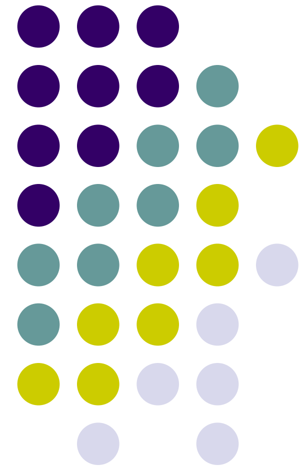
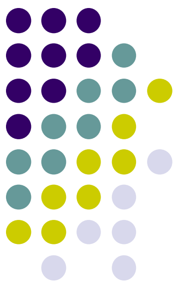
Класи і структури C#



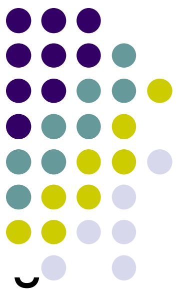
1. Поняття класу в C#, оголошення. Поля та методи.
2. Створення об’єкту. Конструктор класу . Типи передачі параметрів.
3. Перевантаження методів.



1. Поняття класу в C#, оголошення. Поля та методи

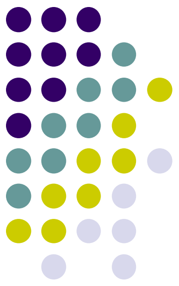
- ***Клас*** – це шаблон який визначає форму, зміст та поведінку об'єкту. Об'єкт – це екземпляр класу. За допомогою класу реалізується перший з основних принципів об'єктно-орієнтованого програмування – ***інкапсуляція***. Складові частини класу називають ***елементами класу.***
- Інкапсуляція – це об'єднання в одному цілому даних та алгоритмів обробки цих даних. Цей термін також включає в себе приховування даних, тобто приховування від зовнішнього користувача деталей реалізації об'єкта.
- Дані зберігаються у вигляді ***полів***, а алгоритми обробки у вигляді ***методів***. Поля, ще іноді називають змінними класу.

1. Поняття класу в C#, оголошення. Поля та методи



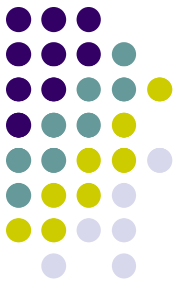
- Елементи класу, які містять виконавчий код, називають ***функціональними елементами***. Вони моделюють поведінку реального об'єкта. До функціональних елементів класу відносять *методи* (включаючи конструктори та деструктори), *властивості, оператори, індексатори та події.*

Методи класу



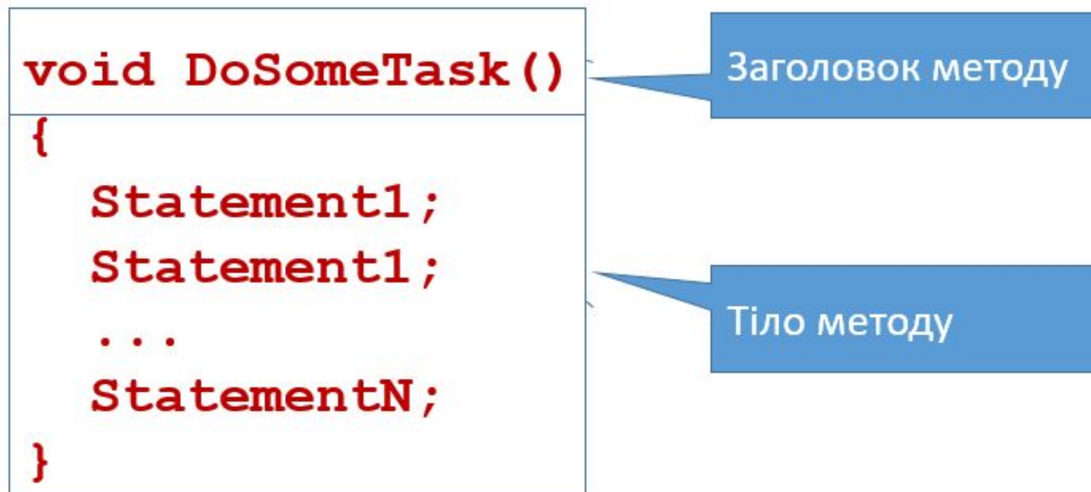
- *Метод* – це іменований блок коду. Методи забезпечують функціональність об'єктів. Об'єкт без методів є пасивною структурою, у якій можна зберігати дані,але вона не виконає ніяких операцій над даними.
- Методи містять більшу частину коду, який забезпечує діяльність програми.
- Код, представлений у методі, можна виконати з будь-якого місця програми в області видимості, використовуючи назву методу.
- Під час роботи програми з методом можна обмінюватися даними, передаючи їх у метод та отримуючи їх від нього.

Методи класу

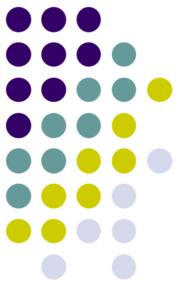


- Найпростіша форма заголовку методу така:

Method тип назва методу (перелік параметрів);

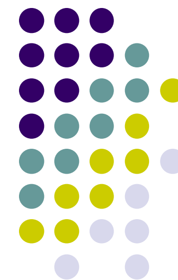


Структура класу

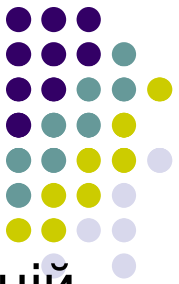


- Дані-елементи класу:
  - константи (неявно статичні) – для потреб класу чи його клієнтів
  - поля класу (статичні поля) – спільні для всіх екземплярів
  - поля екземпляра – стан окремого об’єкта
  - події класу, події екземпляра – це засіб повідомлення про щось варте уваги

1. Поняття класу в C#, оголошення. Поля та методи

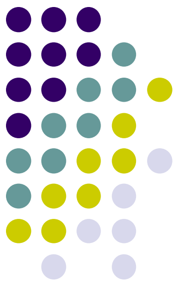


- Функції-елементи класу:
  - методи класу – доступ до полів класу, поведінка класу
  - властивості – набори функцій, доступ до яких нагадує доступ до поля
  - конструктори – функції без типу з іменем класу для ініціалізації екземплярів
  - фіналізатори – ім'я класу з тильдою, працюють перед знищенням екземпляра
  - операції – оператори, перевантажують відомі операції для типів користувача
  - індексатори – індексують об'єкт як масив чи колекцію



# class	# struct
• сукупність даних і функцій	• сукупність даних і функцій
• тип-посилання (купа)	• тип-значення (стек)
• наслідують System.Object	• наслідують System.ValueType
• підтримують одинарне наслідування класів, множинне – інтерфейсів	• підтримують множинне наслідування інтерфейсів, не підтримують – структур
• конструктор за замовчан-ням генерує компілятор; можна перевизначити; оголошення інших конструкторів скасовує автоматичну генерацію	• конструктор за замовчан-ням компілятор генерує завжди; його не можна перевизначити
• поля можна ініціалізувати в оголошенні класу	• ініціалізувати поля в оголошенні структури заборонено

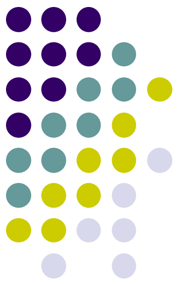
Оголошення класу



```
[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
    }
    ...   }          //
```

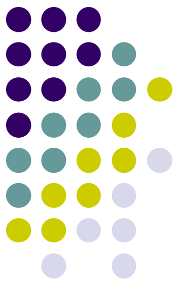
Де [модифікатор_доступу] - дозволяє вказувати рівень доступу до класу або елементів класу

Оголошення класу



- Оголошення класу починають ключовим словом **class**, після якого вказують ім’я класу (назву). Далі між фігурними дужками розміщують оголошення елементів класу. Цю частину оголошення класу називають ***тілом класу***. Елементи класу можна оголошувати у тілі класу в будь-якій послідовності.

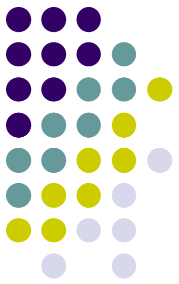
Елементи класу



Найважливішими елементами класу є поля та методи. Поля є елементами даних, а методи – функціональними елементами.

- ***Поля*** – це змінні, які належать класові. Вони можуть бути довільного типу (як попередньо визначеного C#, так і користувацького). Подібно до звичайних змінних, поля зберігають дані.
- Найпростіший синтаксис для оголошення поля такий:

Ініціалізація класу



class TheClass

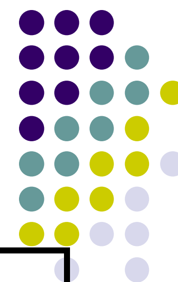
{ int Field1 = 12; *// Ініціалізується значенням 12*

int Field2; *// Ініціалізується значенням 0*

string Field3 = "aaa"; *// Ініціалізується значенням "aaa"*

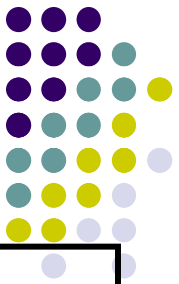
string Field4; *// Ініціалізується значенням null*

Модифікатори доступу C#



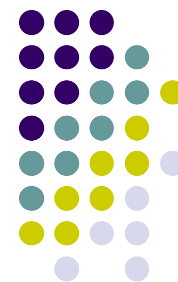
модифікатор	**до чого застосовують**	**призначення**
public (відкритий)	тип, елемент типу	відкриті елементи не мають обмежень доступу
private (закритий)	елемент типу, вкладений тип	закриті елементи доступні тільки в класі (структурі)
protected (захищений)	елемент типу, вкладений тип	захищені елементи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

Інші модифікатори C#



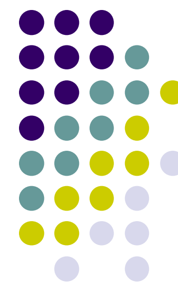
модифікатор	до чого застосовують	призначення
static	клас, поле, метод, властивість	описують дані, поведінку класу
abstract	метод, клас	задати протокол взаємодії, не створювати екземпляри
virtual	метод, властивість	реалізувати пізнє зв'язування, поліморфізм поведінки
override	метод, властивість	
new	метод	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



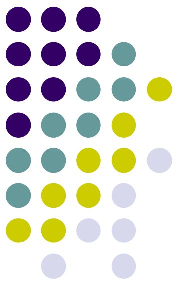
- ***Конструктор*** – це спеціальний метод класу, що викликається при створенні об'єкту класу. Ім'я конструктора завжди співпадає з ім'ям класу.
- Якщо розробником класу явно не визначено жодного конструктора, то для класу автоматично створюється конструктор без аргументів.
- Конструктори не повертають жодних значень, тому при їх оголошенні не потрібно вказувати тип, навіть **void**.

Конструктори



- Клас завжди має хоча б один конструктор. Якщо при оголошенні класу явно не задано жодного конструктора, то використовується конструктор за замовчуванням, який не має параметрів і має порожнє тіло (не робить нічого, тільки створює об'єкт).
- Конструктори можуть бути перевантаженими. Це дозволяє створити множину конструкторів, які можна використовувати для різних режимів ініціалізації новоствореного об’єкта. Потрібний конструктор вказують при створенні екземпляра після оператора new.

Приклад конструктора без аргументів



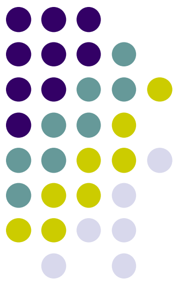
```
public class Car {     //клас машина
public string color;   //колір
public string model;   //модель
public int yearBuilt;  //рік випуску

//конструктор без аргументів
public Car() {
Console.WriteLine("Створюється машина");
}
}
```

Для описаного класу Car об'єкт створюємо за допомогою ключового слова

- new. Car myCar = new Car();
- Transformer myTrans1 = new Transformer();

Конструктор з аргументами



public Car(string c, string m, int yb)

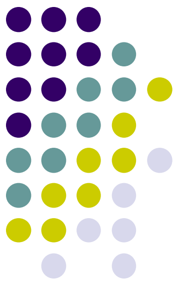
{ //конструктор з аргументами

color = c;

model = m;

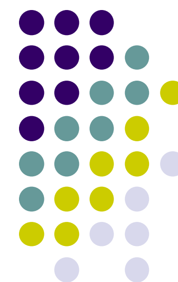
yearBuilt = yb;

Приклад класу, який складається тільки з полів:



```
public class Transformer //клас трансформатор
 {
public int windingCount = 2;//кількість обмоток
public int temperature; //поточна температура
public string power; //потужність
public string model; //марка
public int yearBuilt; //рік випуску
}
```

Використання this



- Ключове слово **this –** використовується для посилання на поточний об'єкт. Воно дозволяє оголошувати аргументи з тими ж назвами, що і поля.

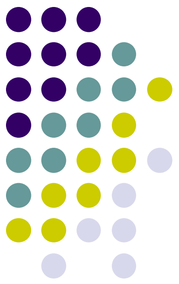
```
public Car(string color, string model, int yearBuilt) {

this.color = color;

this.model = model;
this.yearBuilt = yearBuilt;
}
```

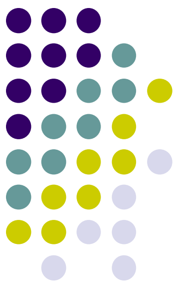
this.color – означає звертання до поля color, а просто color– до аргументу конструктора color.

Конструктори



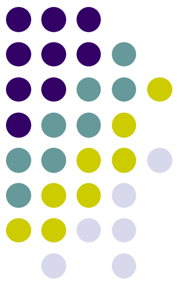
```
class Fraction
{
    private int num;
    private uint den;
// закритий конструктор запобігає створенню екземплярів
    public Fraction() { num = 0; den = 1; }
    public Fraction(int x) { num = x; den = 1; }
    public Fraction(int x, uint y): this(x)
     { if (y > 0) den = y ; }
    public Fraction(Fraction 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
```

Статичний конструктор



- ***Статичний конструктор*** – оголошується за допомогою ключового слова **static**, без модифікаторів та не має аргументів. Викликається один раз перед створенням першого екземпляру класу. Може використовуватись для ініціалізації статичних полів та різного типу налаштувань класу.

Властивості

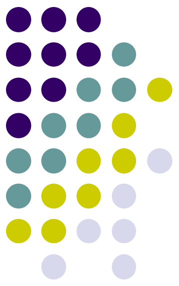


- Це елементи класу, які представляють дані екземпляра чи класу. Властивість має назву та тип; їй можна присвоювати значення, вона є функціональним елементом класу і містить виконавчий код. Властивість не потребує пам’яті для розміщення даних.

Властивість є іменованою парою двох методів, які називають ***методами доступу***:

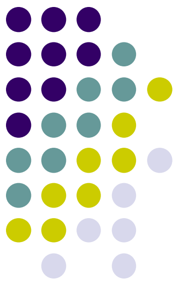
- ► Метод доступу **set** використовують, щоб присвоїти властивості значення.
- ► Метод доступу **get** використовують, щоб прочитати її значення.

Статичні компоненти. Деструктори



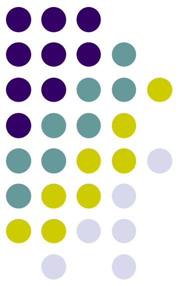
- ***Статичні поля, методи та властивості*** – задаються за допомогою ключового слова **static,** та належать самому класу. Звертання до них виконується **за ім’ям класу**, а не за ім’ям об’єкту.
- Прикладами статичних полів та методів можуть бути поля та методи класу **Math** (такі як Math.PI, Math.Abs(x), Math.Pow(x, y).

Статичні компоненти. Деструктори



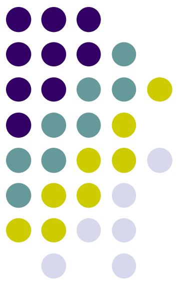
- ***Деструктор*** – це спеціальний метод класу, що викликається для гарантованого звільнення пам’яті під об’єкт. В деструкторі вказуються дії, які необхідно виконати перед тим, як знищити об’єкт. Оголошується за ім’ям класу та за допомогою символу **~**, без модифікаторів. У C# явно викликати деструктор не можливо, оскільки не існує для цього спеціального оператора.

Приклад класу, в якому оголошується деструктор



```
class MyClass
{ int[] A; // внутрішній масив A
    // конструктор класу
public MyClass()
{
// виділення пам'яті для масиву A
A = new int[100];
}
// деструктор класу
~MyClass()
{
// дії, які потрібно виконати, якщо буде відбуватись "збір сміття", //
і може викликатись деструктор //
}
```

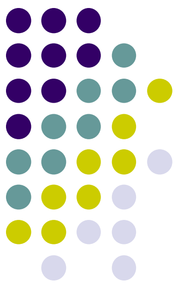
3. Перевантаження методів



Клас може містити кілька методів з однаковою назвою. Такі методи називають *перевантаженими*. Але кожен з перевантажених методів повинен мати різну сигнатуру. *Сигнатура метода* містить:

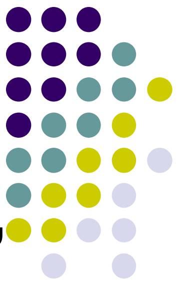
- назву методу,
- кількість параметрів,
- тип і порядок параметрів,
- модифікатори параметрів.

3. Перевантаження методів



- Тип поверненого значення не є частиною сигнатури. Назви формальних параметрів також не є частиною сигнатури – лише їх тип.
- *Перевантаження методу в класі* – це оголошення іншого методу з таким самим іменем у класі, але з відмінними параметрами. Параметри перевантаженого методу повинні відрізнятись типами або кількістю.

Структури

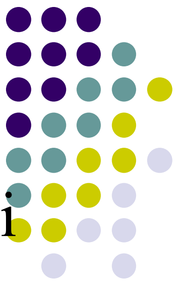


- Структура- це спеціальний тип даних, який створює користувач для опрацювання інформації про об’єкти з деякої предметної області. Така інформація може складатися з даних різних типів. Структура складається з набору полів – даних різних типів, і **форма оголошення структури така:**

struct *ім’я типу структури* : *інтерфейси*

{ // оголошення членів та методів структури // ... };

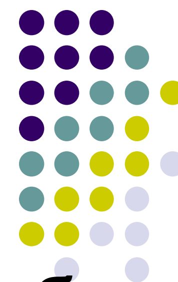
Оголошення структури



- *імʼя типу структури* – назва структурного типу на основі якого будуть оголошуватись обʼєкти (змінні, екземпляри структури);
- *інтерфейси* – перелік інтерфейсів, методи яких потрібно реалізувати в тілі структури.

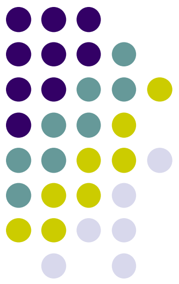
Наприклад: структура, що описує інформацію про студента

```
struct Student
 {
public string name; // прізвище
public string surname; // ім'я
public int year; // рік вступу до навчального закладу
public int birth_year; // рік народження
 public string address; // адреса проживання
```



Відмінності між класами

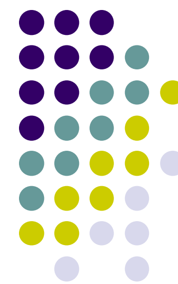
- Структури відносяться до типів значення, тобто виділяти для них пам'ять з допомогою оператора new не обов'язково, а класи відносяться до типів посилання. Це означає, що для об'єкту класу обов'язково потрібно виділяти пам'ять з допомогою оператора new.
- на відміну від класів, у структурах члени даних не можуть бути оголошені з ключовим словом protected
- на відміну від класів, структури не підтримують деструкторів



В структурі можна реалізовувати:

- поля;
- методи;
- інтерфейси;
- індексатори;
- властивості;
- події;
- конструктори (крім конструктора за замовчуванням).

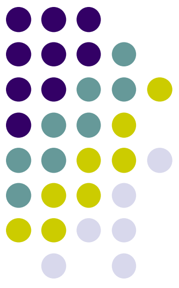
Приклад



Реалізація явно заданого конструктора для структури типу **Point,** що описує точку на координатній площині.

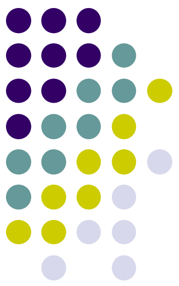
```
struct Point
{
public int x, y; // явно заданий конструктор
public Point(int nx, int ny)
    {
      x = nx; y = ny;
    }
 }
// виклик конструктора при оголошенні екземпляру (об’єкту) структури.
Point P = new Point(5, 6); // P.x = 5; P.y = 6
```

Оголошення, використання методу



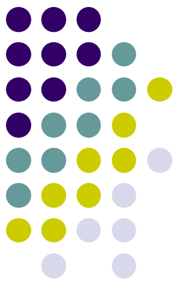
- [модифікатори] тип ім'я ([параметри]) { тіло методу }
- параметри-значення і параметри-посилання (всі ініціалізовані перед передачею):
  - типи-значення передаються за значенням – копіюються
  - типи-посилання передаються за посиланням – копіюється посилання
  - рядки передаються як значення, бо зміна 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();
  }
}
```

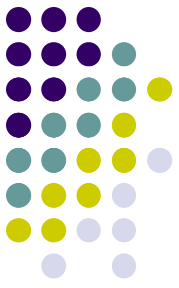
Автоматичні властивості та ще дещо



```
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

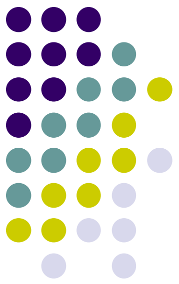


```
using System;
using System.Drawing;
namespace
   Wrox.ProCSharp.StaticConstructorSample
{
  public class UserPreferences
  {
    public static readonly Color BackColor;
    static UserPreferences()
    {
      DateTime now = DateTime.Now;
      if (now.DayOfWeek ==
            DayOfWeek.Saturday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

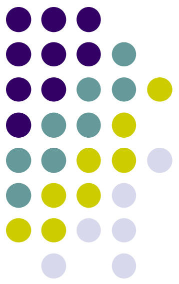
```
namespace Wrox
{ public static class MoneyExtension
  {
    public static void AddToAmount(
this Money money, decimal amountToAdd)
    {
      money.Amount += amountToAdd;
    }
  }
}
```

```
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);

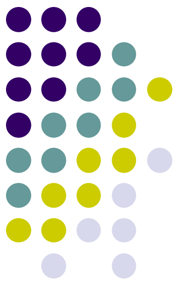
     Console.WriteLine(
          "cash.ToString() returns: " +
          cash.ToString());
        Console.ReadLine();
     }
  }
```

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

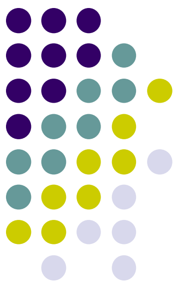


```
public class IdInfo
{
   public int IdNumber;
   public IdInfo(int IdNumber)
   {
      this.IdNumber = IdNumber;
   }
}

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;
   }
}

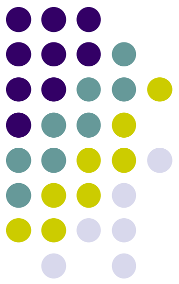
public class Example
{
   public static void Main()
   {
      Person p1 = new Person();
      Person p2 = p1.ShallowCopy();
      Person p3 = p1.DeepCopy();  …
   }
}
```

Анонімні типи C#



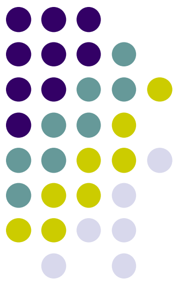
- інкапсуляція в один об’єкт набору властивостей тільки для читання
- тип виводить компілятор за складом ініціалізатора, “придумує” ім’я класу; можна привести до типу object (System.Object)
- ```
  var cap = new { FirstName = "James", MiddleName = 'T',
     LastName = "Kirk" };
  var doc = new { FirstName = "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



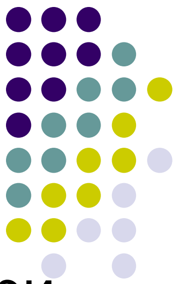
- Наслідування реалізації
  - підклас успадковує набір полів (вкладений об’єкт) і наслідує поведінку (успадковані та перевизначені методи)
  - підклас уточнює, розширяє функціональність базового класу
  - базовий клас реалізує спільну для підкласів функціональність
  - конструктори не наслідуються
- 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
  { … }
  ```
  ```

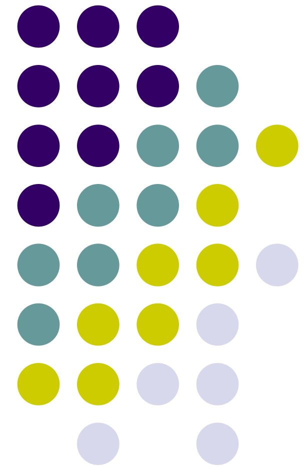
Сумісність і приведення



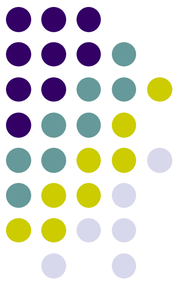
- У посиланні на базовий клас можна зберігати екземпляри підкласів
  - ```
    MyClass P = new SubClass(30, 45);
    MyClass Q = new MyClass(29);
    object R = new MyClass(72); // неявне приведення
    ```
  - ```
    void DeelWith(MyClass M)
    {   Console.WriteLine(M.ToString());  }
    DeelWith(P); 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 в C#

1. Поняття інтерфейсу в C#, оголошення.
2. Реалізація інтерфейсу.
3. Спадкування інтерфейсів.



Інтерфейси

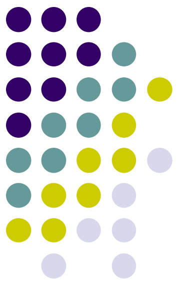


- це посилальний тип, який задає множину функціональних елементів, але не реалізовує їх. Це набір методів (властивостей, подій, індексаторів), реалізацію яких повинен забезпечити клас.
- Жоден метод інтерфейсу не може мати реалізації.
- Інтерфейс показує, що клас повинен робити, але не визначає, як саме. При впровадженні інтерфейсу у клас мають бути реалізовані всі методи інтерфейсу, але кожен клас може реалізувати їх по-іншому.

Інтерфейси як засіб визначення типу

- **Основне призначення** - Об’єднання в одну іменовану функціональну групу методів, властивостей і подій
  - Гарантія реалізації класом оголошеної інтерфейсом поведінки
  - CLR утворює об'єкт-тип
- **Наслідування інтерфейсів** - обмежений варіант множинного наслідування в CLR
  - Використання об’єктів типів, які реалізують інтерфейс, у контексті інтерфейсного типу як базового
  - Реалізація поліморфної поведінки незалежними типами
  - Наслідування інтерфейсів інтерфейсом – структурування функціональності

Оголошення інтерфейсу

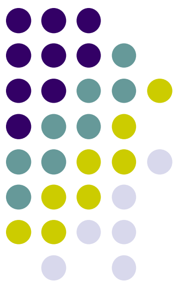


- **Інтерфейс** оголошується за межами класу, за допомогою ключового слова interface:

```
interface ISomeInterface
{
  // Тіло інтерфейсу
}
```

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

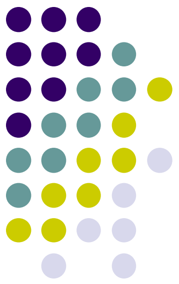
Тіло інтерфейсу



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

```
interface ISomeInterface
{
   string SomeProperty {get; set; } // Властивість

   void SomeMethod (int a); // Метод
}
```



- **Інтерфейс може містити лише оголошення методів, властивостей, індексаторів та подій.**

method

indexer

property

event

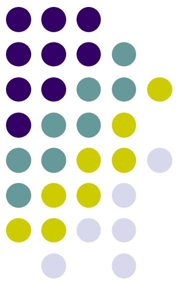
```
interface IMyInterface
{
  void Process(int arg1, double arg2);

  float this [int index] { get; set; }

  string Name { get; set; }

  event MouseEventHandler Mouse;
}
```

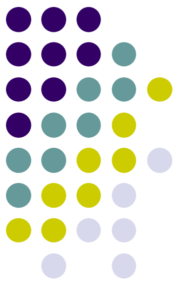
Реалізація інтерфейсу



Щоб вказати, що клас реалізовує інтерфейс, необхідно після імені класу і двокрапки вказати ім'я інтерфейсу:

```
class SomeClass: ISomeInterface // реалізація інтерфейсу ISomeInterface
{
   // Тіло класу
}
```

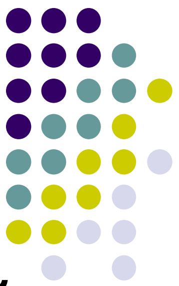
Реалізація інтерфейсу



- Якщо клас, у який впроваджують інтерфейс, є похідним, то його базовий клас у списку успадкованих класів повинен бути першим. Після назви базового класу через кому можна вказати довільну кількість інтерфейсів:

```
class TheClass : TheBaseClass, IInterface1,
   IInterface2
{ ...
}
```

Реалізація інтерфейсу



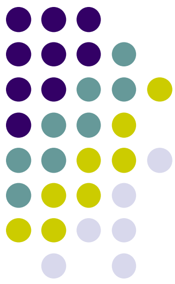
Клас, який реалізовує інтерфейс, повинен надати реалізацію всіх членів інтерфейсу.

В інтерфейсі розміщують виключно оголошення функціональних елементів, а їх тіла повинні бути реалізовані у класах, які спадкують інтерфейс.

Приклад

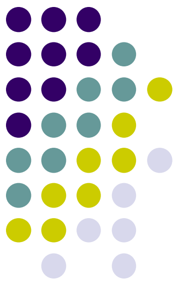
Існує клас геометричних фігур Прямокутник і Коло. У обох класів повинні бути методи обчислення периметра і площі. Ці методи ми можна представити інтерфейсом:

```
interface IGeometrical    // оголошення інтерфейсу
{
void GetPerimeter ();
   void GetArea ();
}
```



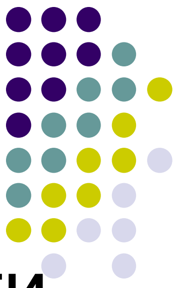
```
class Rectangle: IGeometrical    // реалізація інтерфейсу
{
   public void GetPerimeter ()
   {
    Console.WriteLine ("(a + b) * 2");
   }
   public void GetArea ()
   {
    Console.WriteLine ("a * b");
   }}
class Circle: Igeometrical      // реалізація інтерфейсу
{
   public void GetPerimeter ()
   {
    Console.WriteLine ("2 * pi * r");
   }
    public void GetArea ()
   {
    Console.WriteLine ("pi * r ^ 2");
   }}
      f.GetArea ();
    }
    Console.ReadLine ();
   }}
```

Приклад



```
class Program
{
  static void Main (string [] args)
  {
   List <IGeometrical> figure = new List <IGeometrical> ();
   figure.Add (new Rectangle ());
      foreach (IGeometrical f in figure)
   {
    f.GetPerimeter ();
    f.GetArea ();
   }
   Console.ReadLine ();
  }}
```

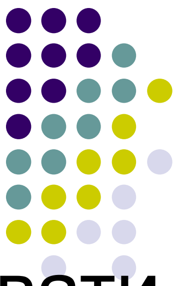
Спадкування інтерфейсів



Інтерфейс, як і клас, може спадкувати структуру іншого інтерфейсу. Таким чином, на основі одних інтерфейсів можна створювати інші інтерфейси. Щоб вказати, що інтерфейс успадковує інші інтерфейси, імена базових інтерфейсів слід вказати так само, як при спадкуванні класів – в окремому переліку:

```
interface IInterface : IInterface1, IInterface2
{ ...
```

Спадкування інтерфейсів



На відміну від класів, які можуть успадкувати тільки один базовий клас, інтерфейси можуть спадкувати довільну кількість базових інтерфейсів. У свою чергу, кожен базовий інтерфейс може спадкувати інші інтерфейси.

Результуючий інтерфейс крім своїх елементів містить елементи всіх успадкованих інтерфейсів. Відповідно клас, який впроваджує такий інтерфейс, також повинен реалізувати всі елементи всіх успадкованих інтерфейсів.