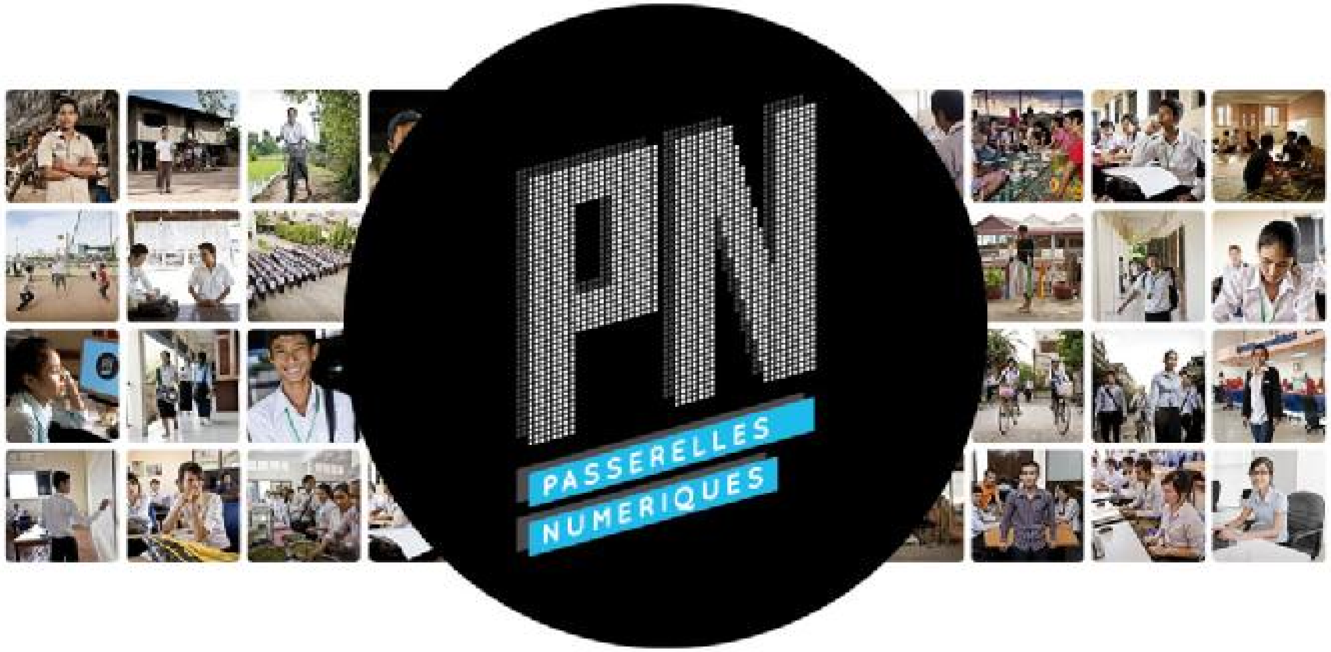


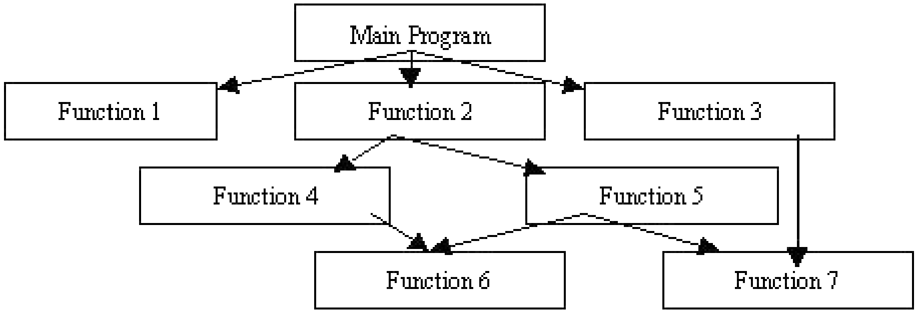
Microsoft®
.NET

# UNIT 3. OOP

- Lession 1. Classes and Objects
- Lesson 2. Properties
- Lesson 3. Indexers

# Introduction

- Disavantage of procedural programming



Main Program

Function 1

Function 2

Function 3

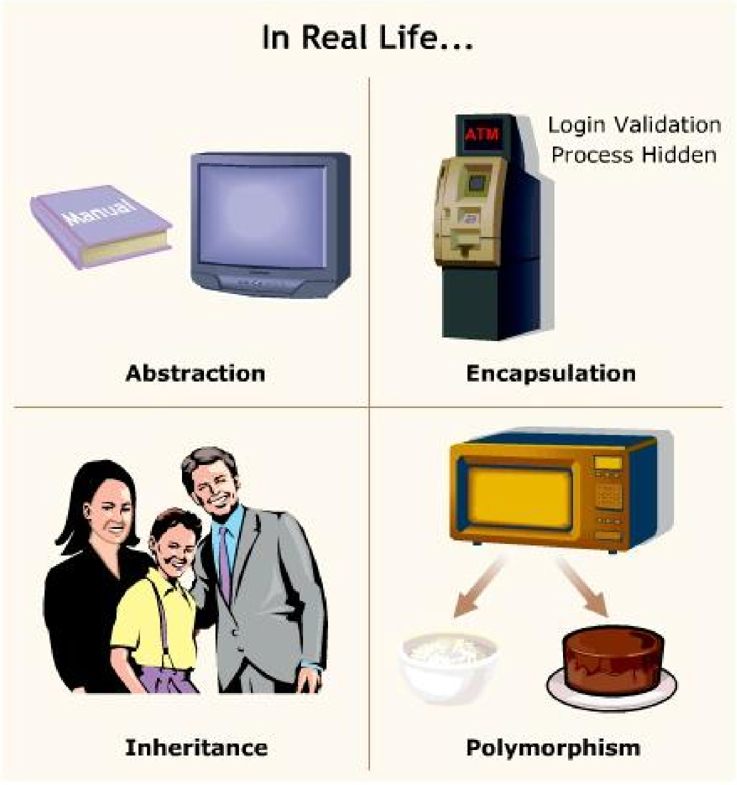
Function 4

Function 5

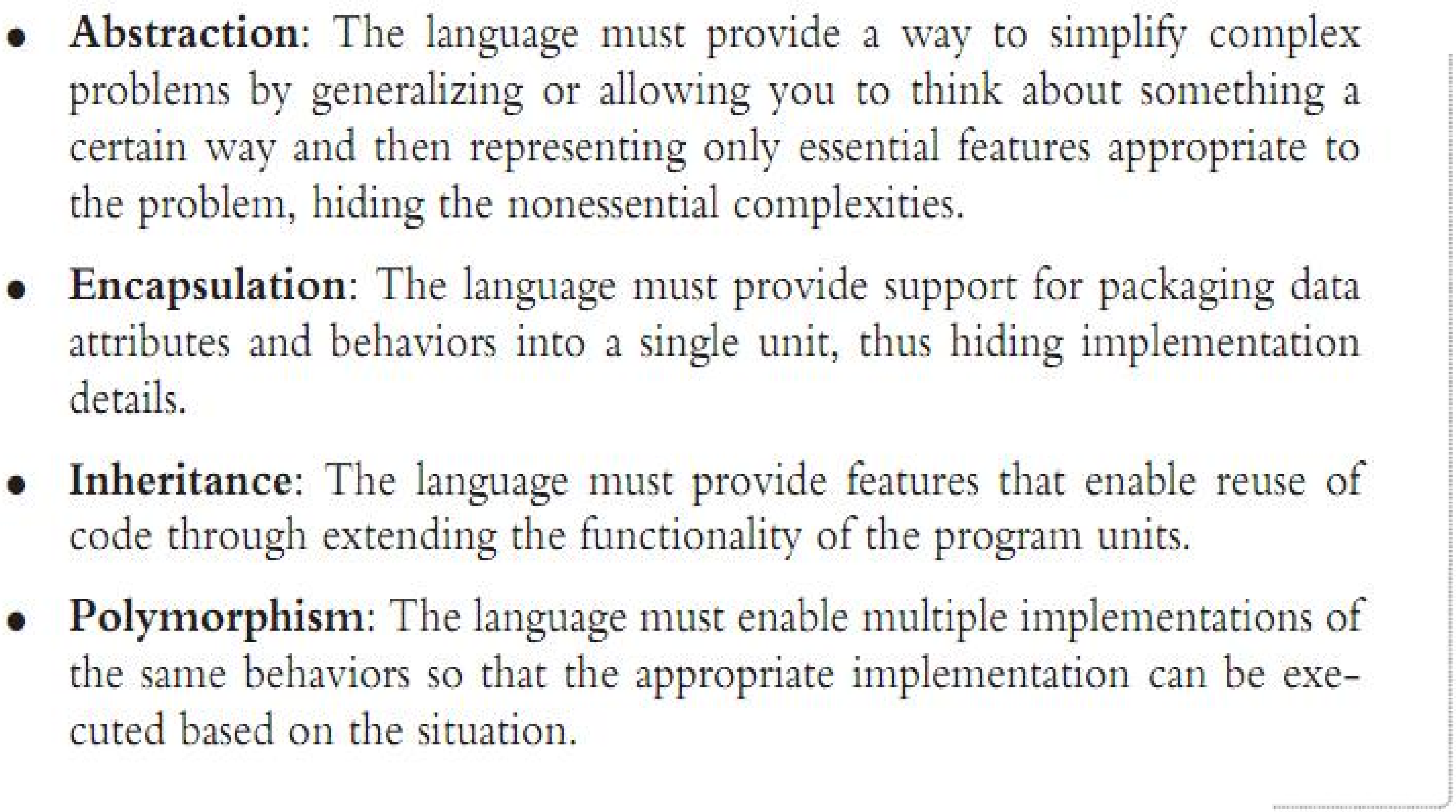
Function 6

Function 7

- Object Oriented Programming



# Object Oriented Language Features



- **Abstraction**: The language must provide a way to simplify complex problems by generalizing or allowing you to think about something a certain way and then representing only essential features appropriate to the problem, hiding the nonessential complexities.
- **Encapsulation**: The language must provide support for packaging data attributes and behaviors into a single unit, thus hiding implementation details.
- **Inheritance**: The language must provide features that enable reuse of code through extending the functionality of the program units.
- **Polymorphism**: The language must enable multiple implementations of the same behaviors so that the appropriate implementation can be executed based on the situation.

# LESSION 1. CLASSES AND OBJECTS

- Objects
- Classes
- Instantiating Object

# Objects

- An object is a tangible entity such as…

Objects

# Objects

- Every object has some characteristics and is capable of performing certain actions
  - In the real life:

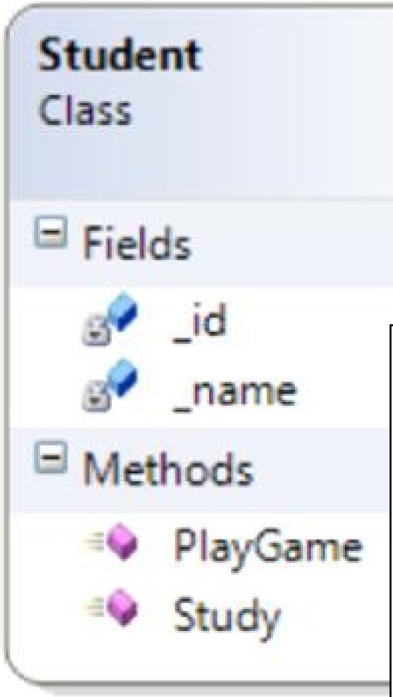
    Object=Characteristics+Behaviours

  - In programming:

    Object=Data+Methods

# Classes

- Several objects have a common characteristics and behavior and thus can be group under a single class.



**Student**
Class

Fields

_id
_name

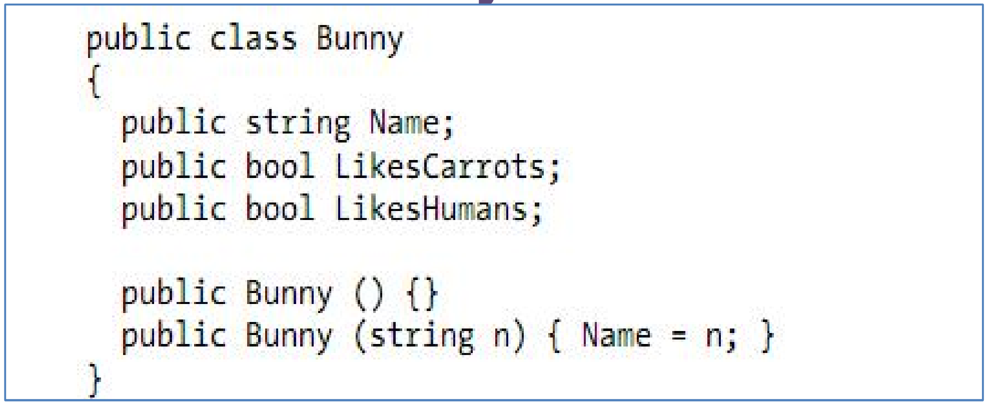
Methods

PlayGame
Study

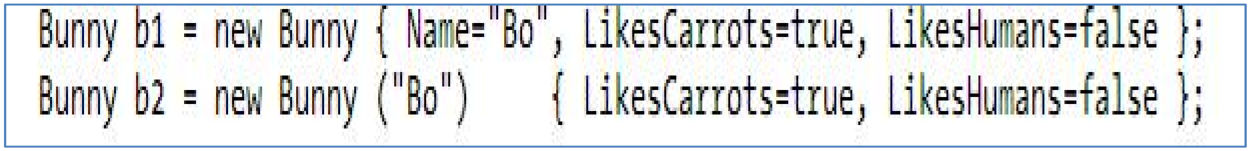
# Object Initialization

- Use new keyword

- Use object initializer to create an object.

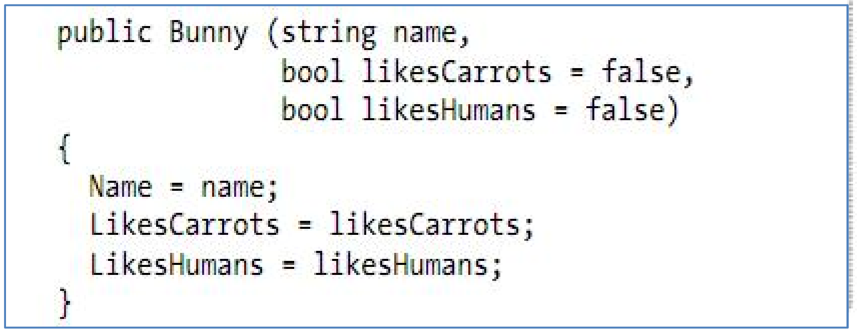


```
public class Bunny
{
  public string Name;
  public bool LikesCarrots;
  public bool LikesHumans;

  public Bunny () {}
  public Bunny (string n) { Name = n; }
}
```



```
Bunny b1 = new Bunny { Name="Bo", LikesCarrots=true, LikesHumans=false };
Bunny b2 = new Bunny ("Bo")     { LikesCarrots=true, LikesHumans=false };
```

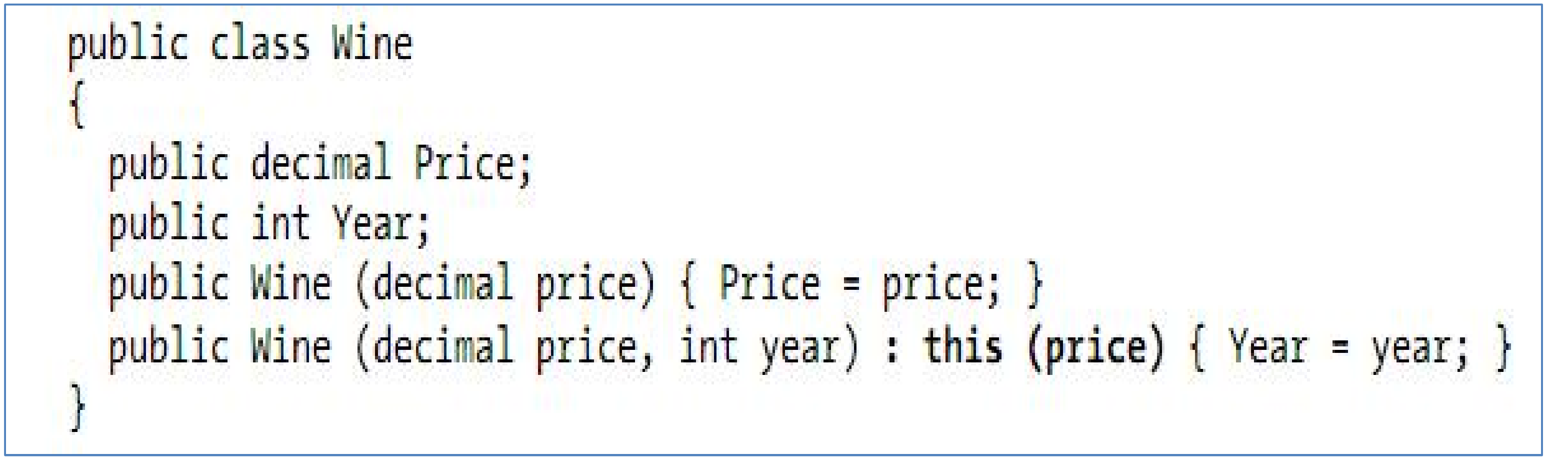
- Use optional parameter



```
public Bunny (string name,
              bool likesCarrots = false,
              bool likesHumans = false)
{
  Name = name;
  LikesCarrots = likesCarrots;
  LikesHumans = likesHumans;
}
```

# Overloading constructor

- **One con-structor may call another**

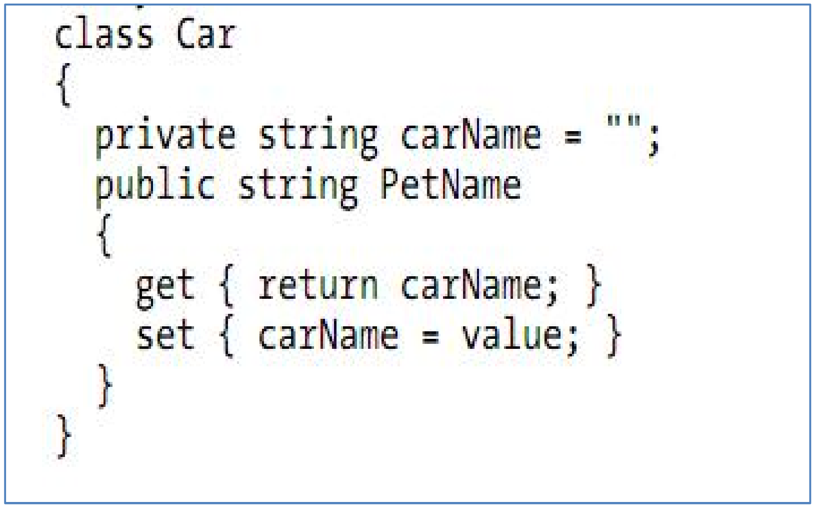


```
public class Wine
{
  public decimal Price;
  public int Year;
  public Wine (decimal price) { Price = price; }
  public Wine (decimal price, int year) : this (price) { Year = year; }
}
```

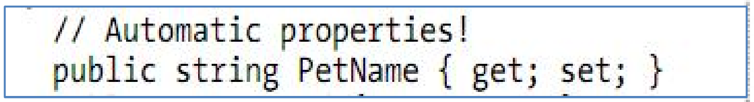
# PROPERTIES

# Properties

- A property is a named set of two matching methods called accessors.
  - The set accessor is used for assigning a value to the property.
  - The get accessor is used for retrieving a value from the property.
- Ex:



```
class Car
{
  private string carName = "";
  public string PetName
  {
    get { return carName; }
    set { carName = value; }
  }
}
```



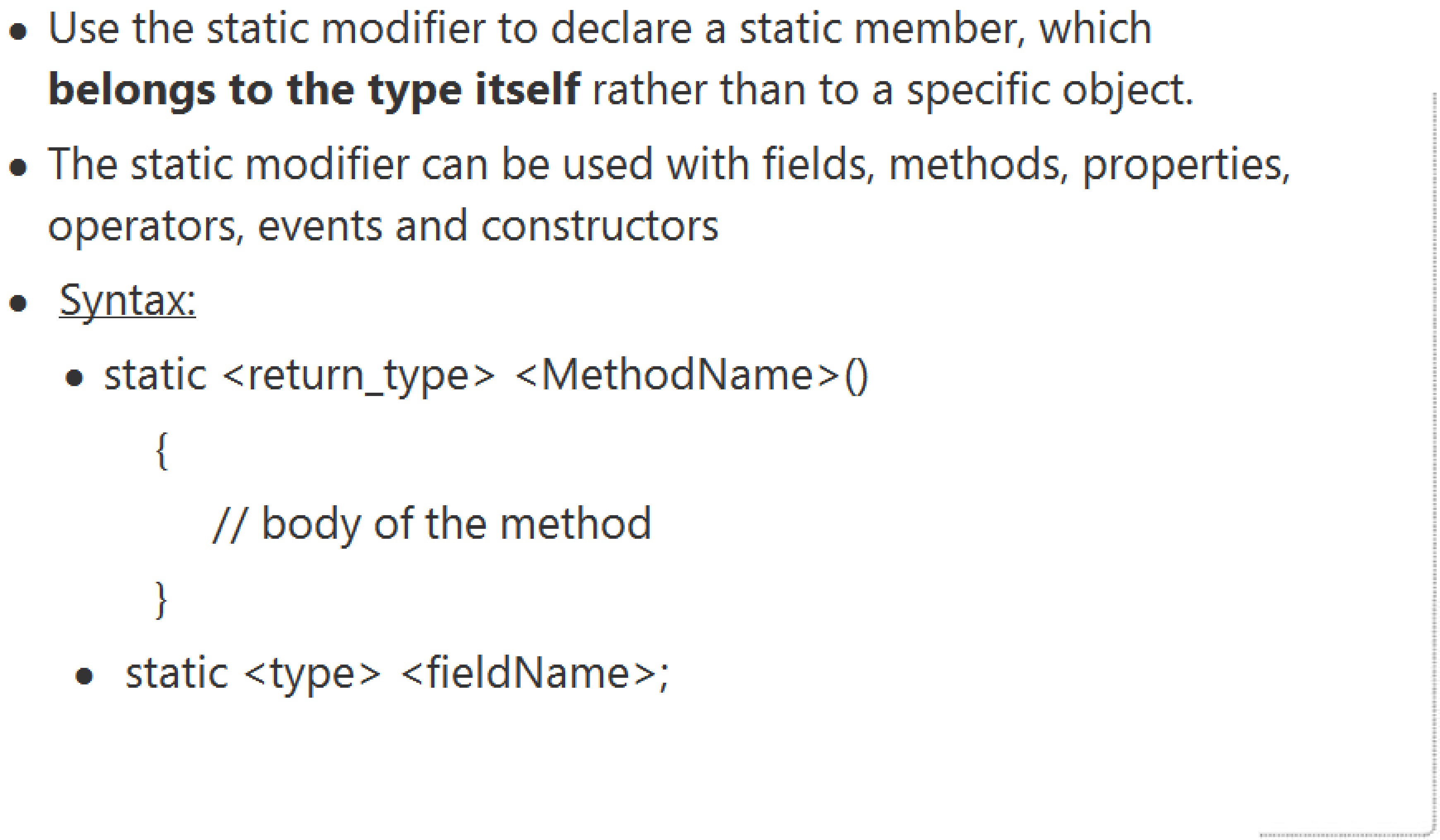
```
// Automatic properties!
public string PetName { get; set; }
```

# Properties

- Ex1: Definition Point type,which has (x, y) position, has a color (contained in an enum named PointColor (LightBlue, BloodRed, Gold).
  - Provide Constructors to establish (x,y) position and color.
  - Display the status of the points
- Ex2: Build a Rectangle class, which makes use of the Point type to represent its upper-left and bottom-right coordinates, display the status of the rectangle

# STATIC KEYWORD

# Static members



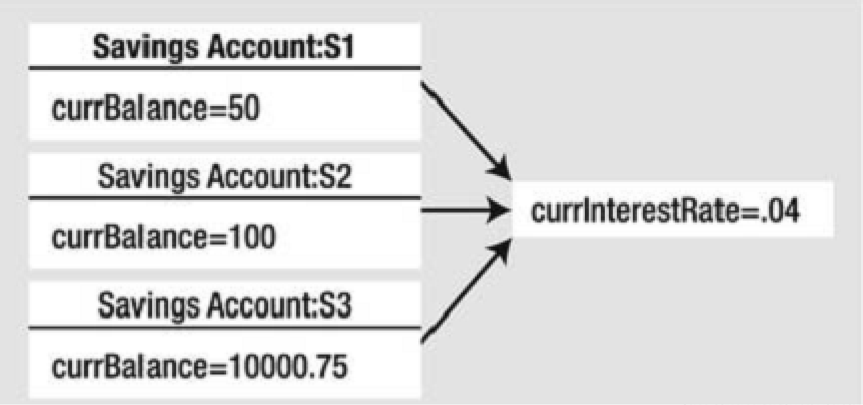
- Use the static modifier to declare a static member, which **belongs to the type itself** rather than to a specific object.
- The static modifier can be used with fields, methods, properties, operators, events and constructors
- Syntax:
  - static <return_type> <MethodName>()

    ```
    {
        // body of the method
    }
    ```
  - static <type> <fieldName>;

# Static members

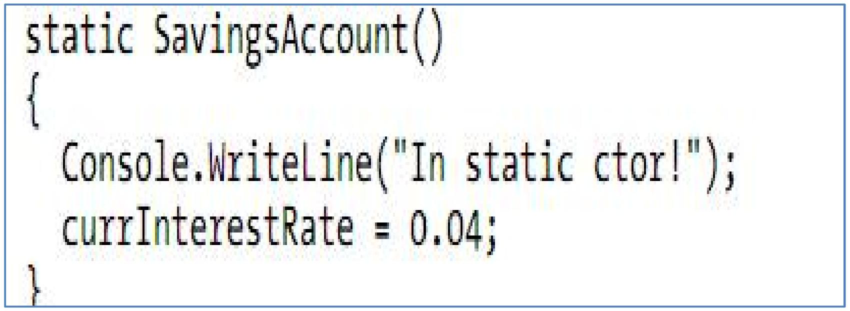
- **Static field data**

```
class SavingsAccount
   {
     // Instance-level data.
     public double currBalance;

     // A static point of data.
     public static double currInterestRate = 0.04;

     public SavingsAccount(double balance)
     {
       currBalance = balance;
     }
   }
```



# Static constructors

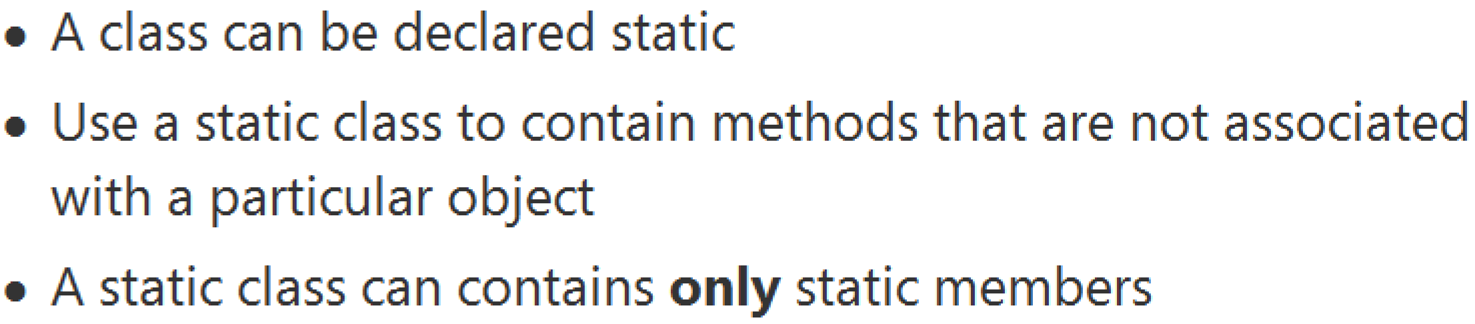
- a static constructor is used to initialize the values of static data when the value is not known at compile time.



```
static SavingsAccount()
{
  Console.WriteLine("In static ctor!");
  currInterestRate = 0.04;
}
```

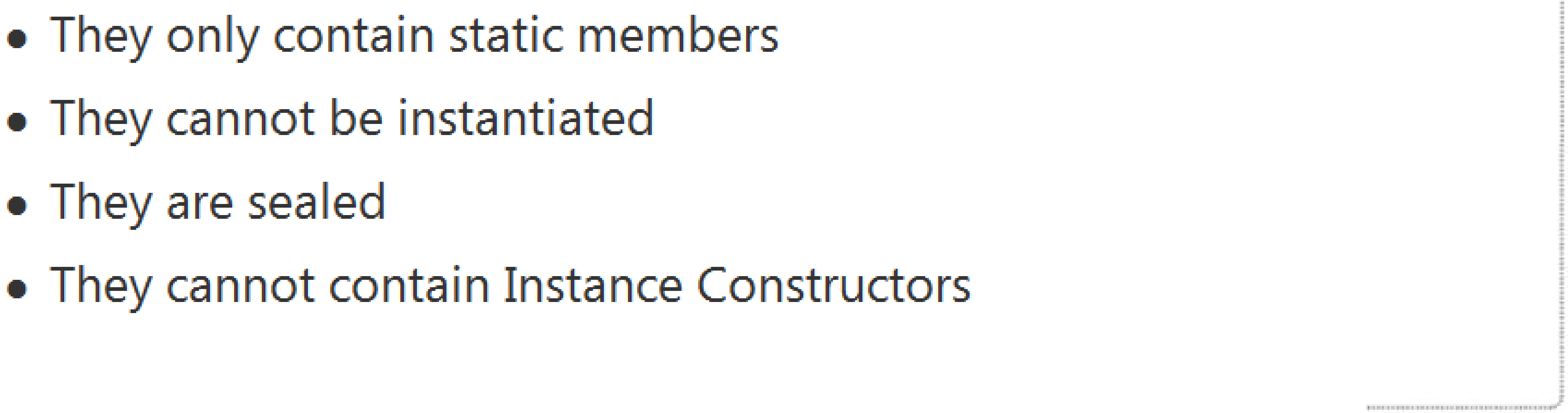
# Static class

- **Definition**



- A class can be declared static
- Use a static class to contain methods that are not associated with a particular object
- A static class can contains **only** static members

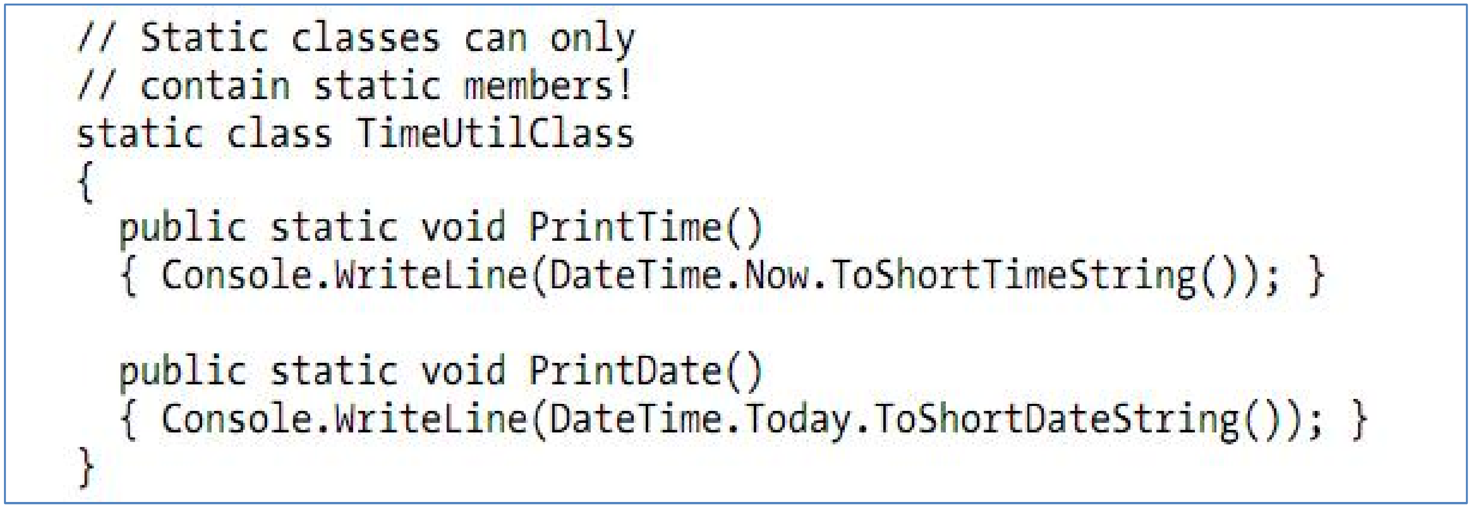
- **Features**



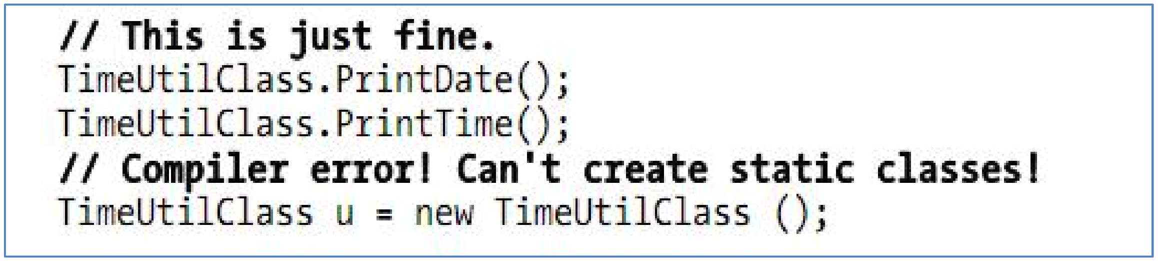
- They only contain static members
- They cannot be instantiated
- They are sealed
- They cannot contain Instance Constructors

# Static class

- **Ex:**



```
// Static classes can only
// contain static members!
static class TimeUtilClass
{
  public static void PrintTime()
  { Console.WriteLine(DateTime.Now.ToShortTimeString()); }

  public static void PrintDate()
  { Console.WriteLine(DateTime.Today.ToShortDateString()); }
}
```

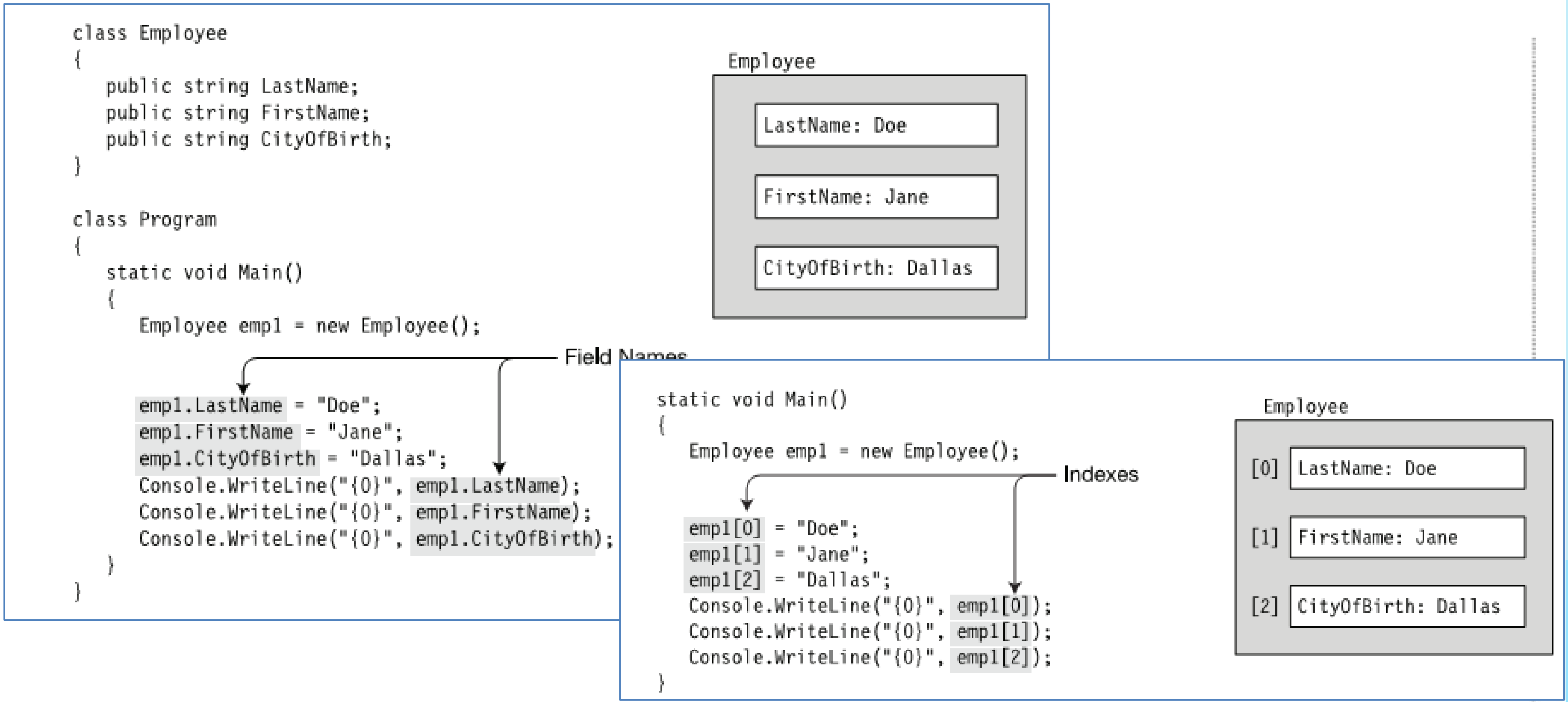


```
// This is just fine.
TimeUtilClass.PrintDate();
TimeUtilClass.PrintTime();
// Compiler error! Can't create static classes!
TimeUtilClass u = new TimeUtilClass ();
```

# INDEXERS

# Indexers

- Ex: without indexer



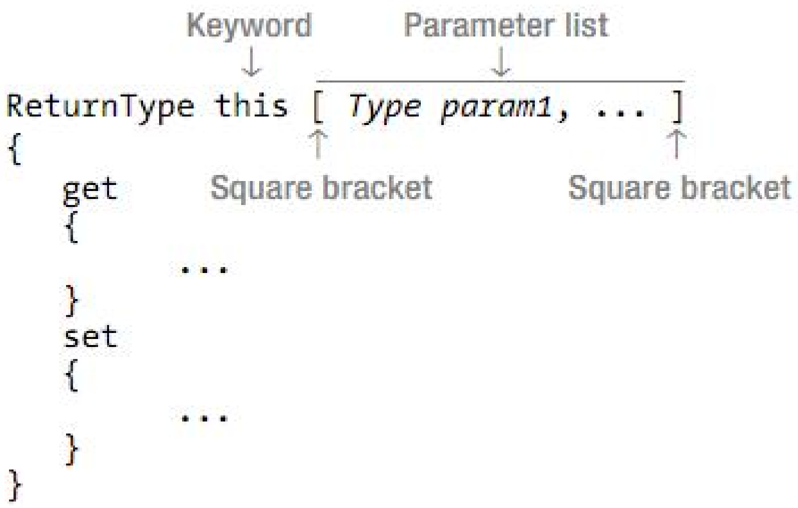
```
class Employee
{
   public string LastName;
   public string FirstName;
   public string CityOfBirth;
}

class Program
{
   static void Main()
   {
      Employee emp1 = new Employee();

      emp1.LastName = "Doe";
      emp1.FirstName = "Jane";
      emp1.CityOfBirth = "Dallas";
      Console.WriteLine("{0}", emp1.LastName);
      Console.WriteLine("{0}", emp1.FirstName);
      Console.WriteLine("{0}", emp1.CityOfBirth);
   }
}
```

```
static void Main()
{
   Employee emp1 = new Employee();

   emp1[0] = "Doe";
   emp1[1] = "Jane";
   emp1[2] = "Dallas";
   Console.WriteLine("{0}", emp1[0]);
   Console.WriteLine("{0}", emp1[1]);
   Console.WriteLine("{0}", emp1[2]);
}
```

- With indexed fieds

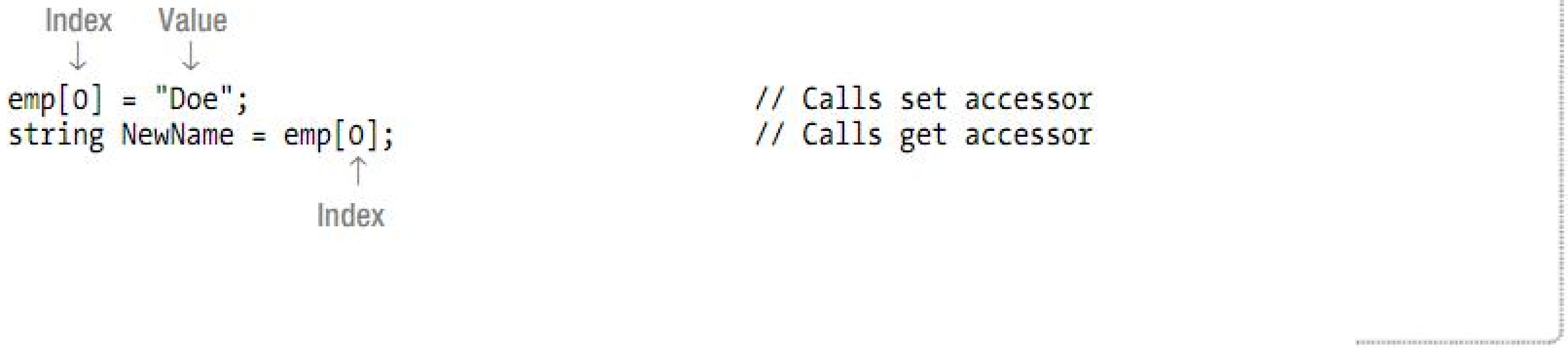
# Indexers

- **Declare an indexer:**
  - **Indexer not have a name**



```
ReturnType this [ Type param1, ... ]
{
    get
    {
            ...
    }
    set
    {
            ...
    }
}
```

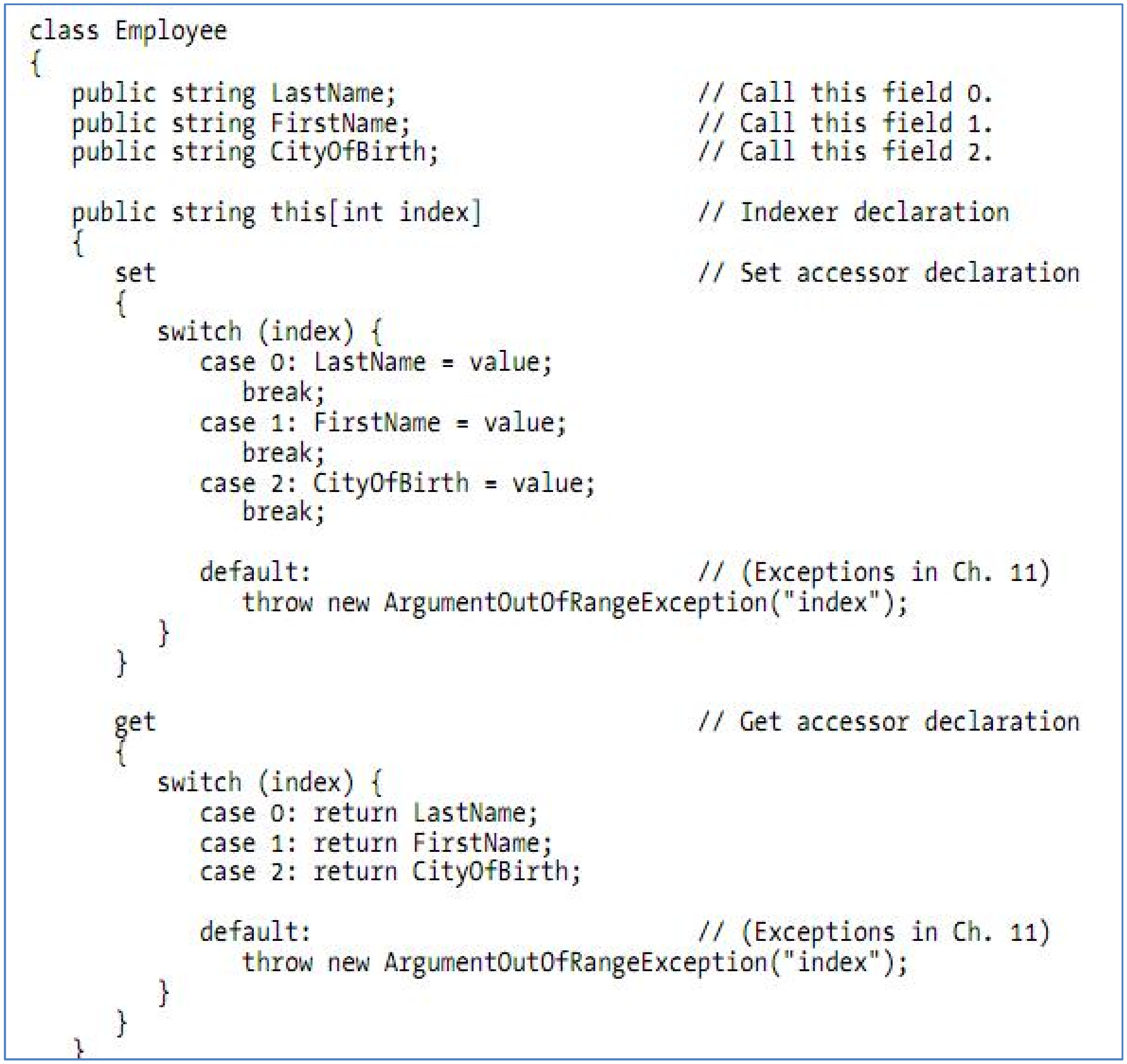
- **Use indexer**



```
emp[0] = "Doe";                          // Calls set accessor
string NewName = emp[0];                 // Calls get accessor
```

# Indexers

- **Ex:**



```
class Employee
{
   public string LastName;                     // Call this field 0.
   public string FirstName;                    // Call this field 1.
   public string CityOfBirth;                  // Call this field 2.

   public string this[int index]               // Indexer declaration
   {
      set                                      // Set accessor declaration
      {
         switch (index) {
            case 0: LastName = value;
               break;
            case 1: FirstName = value;
               break;
            case 2: CityOfBirth = value;
               break;

            default:                           // (Exceptions in Ch. 11)
               throw new ArgumentOutOfRangeException("index");
         }
      }

      get                                      // Get accessor declaration
      {
         switch (index) {
            case 0: return LastName;
            case 1: return FirstName;
            case 2: return CityOfBirth;

            default:                           // (Exceptions in Ch. 11)
               throw new ArgumentOutOfRangeException("index");
         }
      }
   }
```