Unit 1.2: Object Oriented Programming in C#

Lecturer: Le Thi Bich Tra

Content

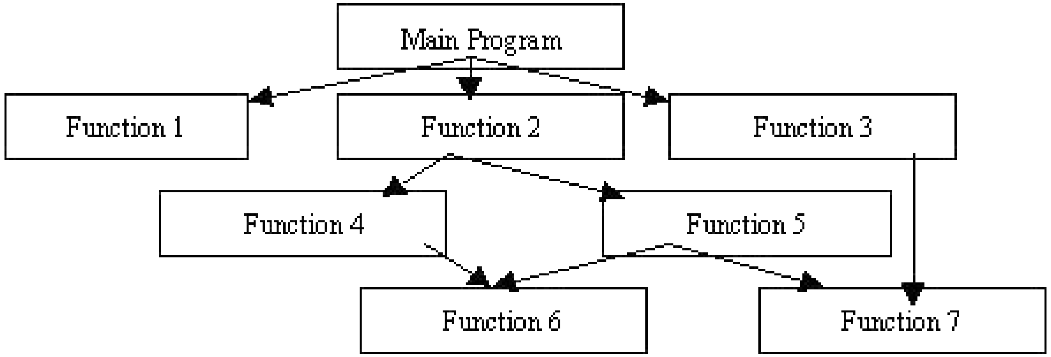
- Lession 1: Class and Object
- Lession 2: Inheritance and Polymorphism
- Lession 3: Interface

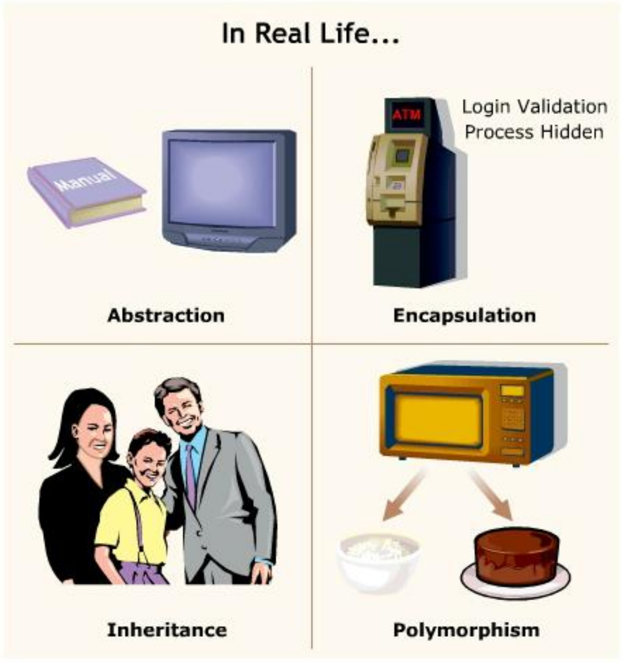
Lession 1: Class and Object

- OOP and its Features?
- Class and Object
- Constructors
- Partial class
- Class Library
- Static keyword
- Indexers

OOP

- Disavantage of procedural programming:





- Object Oriented Programming:

is a programming model where programs are organized around objects and data rather than action and logic.

OOP Features

- Inheritance
- Encapsulation
- Abstraction
- Polymorphism

Object

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

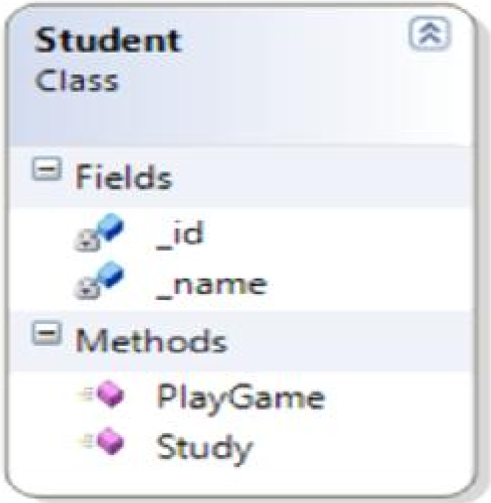
    Object=Characteristics+Behaviours

  - In programming:

    Object=Data+Methods

Class

- 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

› Class

Object Initialization

- Use new keyword

```
Student st = new Student(1,”A”);
```

- Use object initializer to create an object.

```
Student st = new Student { id = 1, name = “A”};
```

Constructor

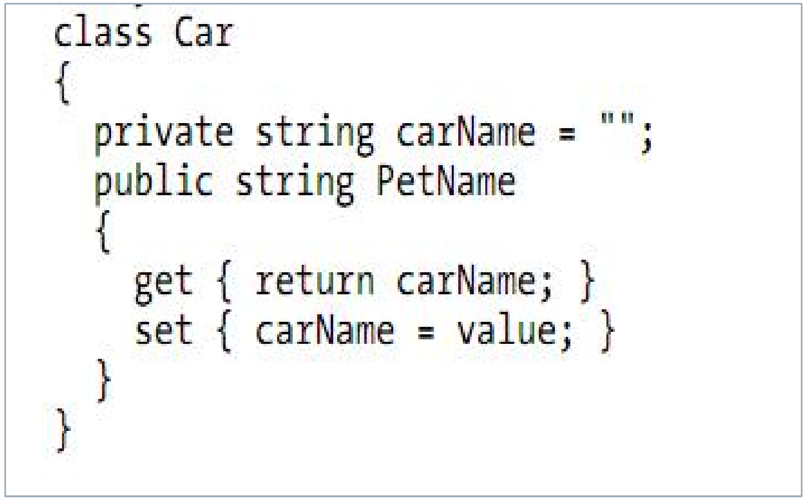
- A constructor is a specialized function that is used to initialize fields. A constructor has the same name as the class
- There are some important rules:
  - Classes with no constructor have an implicit constructor called the default constructor, that is parameterless. The default constructor assigns default values to fields.
  - A constructor returns void but does not have an explicitly declared return type.
- One con-structor may call another
  - Use this keyword

Constructor

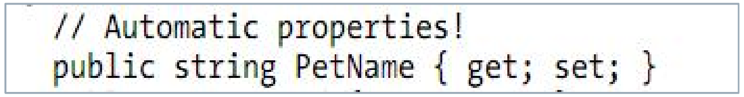
- Static Constructor:
  - You create a static constructor to initialize static fields.
  - Static constructors are not called explicitly with the new statement. They are called when the class is first referenced.
- There are some limitations of the static constructor:
  - Static constructors are parameterless.
  - There is no accessibility specified for Static constructors.

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.
- E.g:



```
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

Class Library

- .NET provides the capability of creating libraries (components) of a base application rather than an executable (".exe").
- The library project's final build version will be ".DLL" that can be referenced from other outside applications to expose its entire functionality.
- Steps:
  - Create a Class Library
  - Build the class to .dll file
  - Create an another application
  - Right-click on the Reference then "Add reference" then select the path of the dll file.

Static keyword

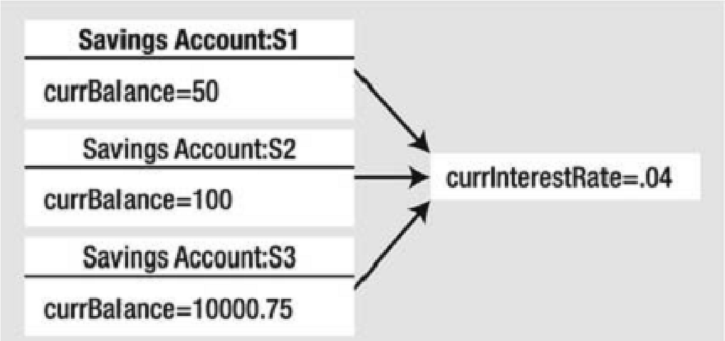
- Use static keyword to declare a static member which belong to the type itself rather than to a specific object.
- Syntax:
  - static <data type> fieldName;
  - static <return type> methodName([parameters]){

        }

Static keyword

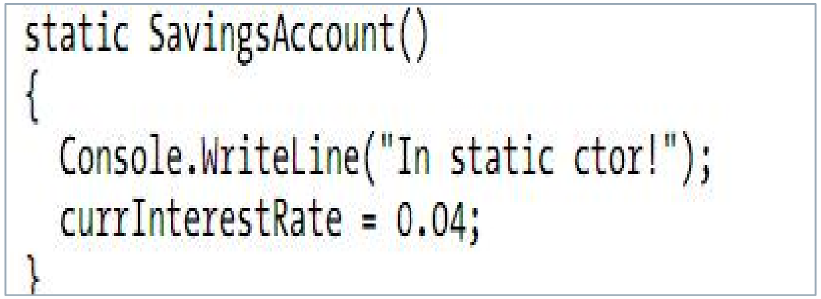
- E.g:

```
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 keyword

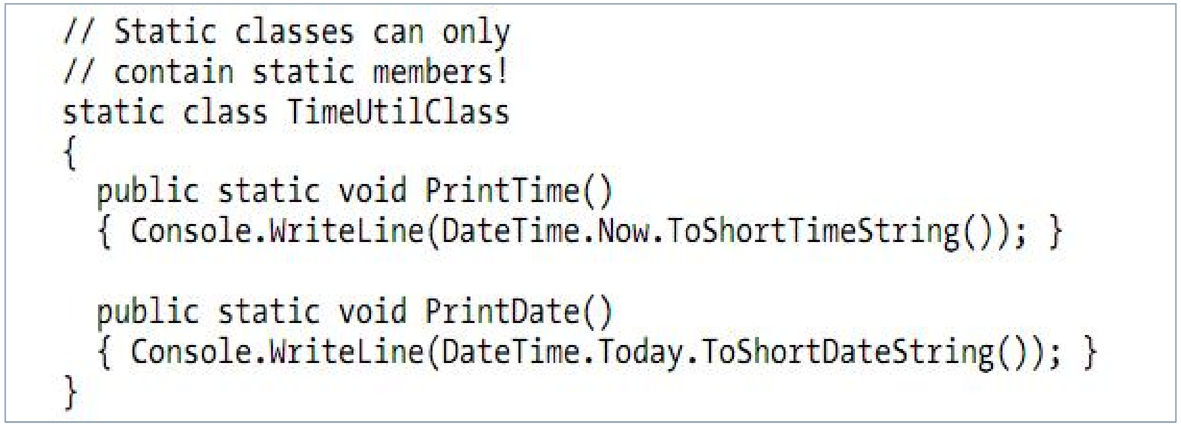
- Static Constructor:
  - You create a static constructor to initialize static fields (when the value is not known at compile time)
  - Static constructors are not called explicitly with the new statement. They are called when the class is first referenced.
- There are some limitations of the static constructor:
  - Static constructors are parameterless.
  - There is no accessibility specified for Static constructors.



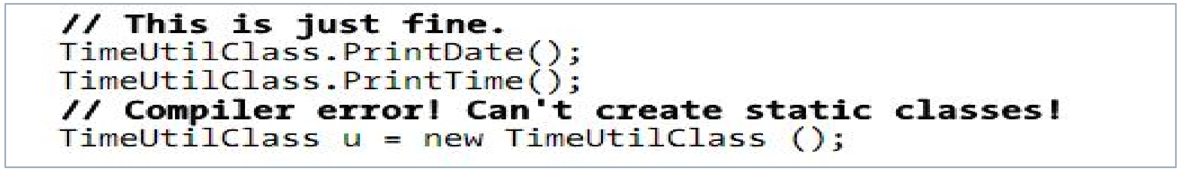
```
static SavingsAccount()
{
  Console.WriteLine("In static ctor!");
  currInterestRate = 0.04;
}
```

Static keyword

- Static class:
  - A static class only contain static members
  - Use static class to contain members that are not associated with a particular object.
  - A static class can not be instantiated



```
// 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 ();
```

Partial class

- The partial keywords allow a class to span multiple source files. When compiled, the elements of the partial types are combined into a single assembly.
- There are some rules for defining a partial class:
  - A partial type must have the same accessibility.
  - Each partial type is preceded with the "partial" keyword.
  - If the partial type is sealed or abstract then the entire class will be sealed and abstract.

Partial class

- Use partial keyword

```
using System;
namespace Demo
{
   public partial class partialclassDemo
   {
      public void method1()
      {
         Console.WriteLine("method from part1 class");
      }
   }
}
```

```
using System;

namespace Demo
{
   public partial class partialclassDemo
   {
      public void method2()
      {
         Console.WriteLine("method from part2 class");
      }
   }
}
```

- Use partial class:

```
using System;
namespace Demo
{
   class Program
   {
      static void Main(string[] args)
      {
         //partial class instance
         partialclassDemo obj = new partialclassDemo();
         obj.method1();
         obj.method2();   }     } }
```

Lession 2: Inheritace and Polymorphism

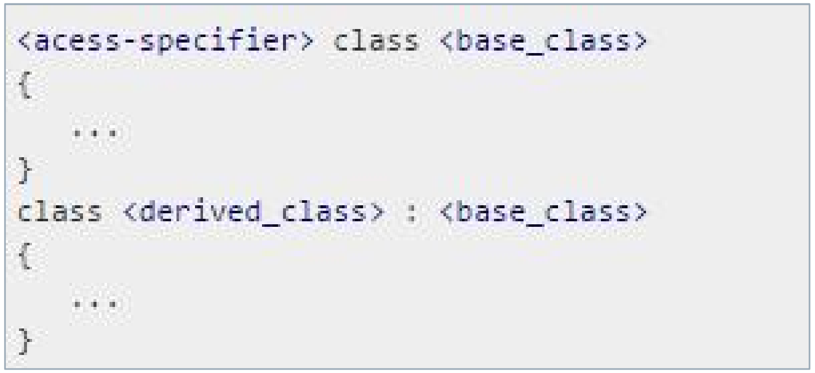
- Inheritance
- Polymorphism
- Casting Operations
- Abstract class

Inheritance

- Inheritance is a way to reuse code of existing class
- A base class (parent class, super class)
- A derived class (child class, subclass)

Inheritance

- The idea of inheritance implements the is-a relationship.
  - E.g: MiniCar is a Car, Dog is a Animal,...
- Don't use it to build has-a relationship
- C# does not support multiple inheritance
- Syntax:



```
<acess-specifier> class <base_class>
{
   ...
}
class <derived_class> : <base_class>
{
   ...
}
```

- Use class diagram with VS

Inheritance

- The base keyword is used to access members of the base class
- To prevent inheritance, use sealed class:
  - Sealed class cannot be used as a base class

Polymorphism

- Polymorphism = ability to take more than one form (objects have more than one type)
  - A class can be used through its parent interface
  - A child class may override some of the behaviors of the parent class
- Polymorphism allows abstract operations to be defined and used
  - Abstract operations are defined in the base class' interface and implemented in the child classes
  - Declared as abstract or virtual

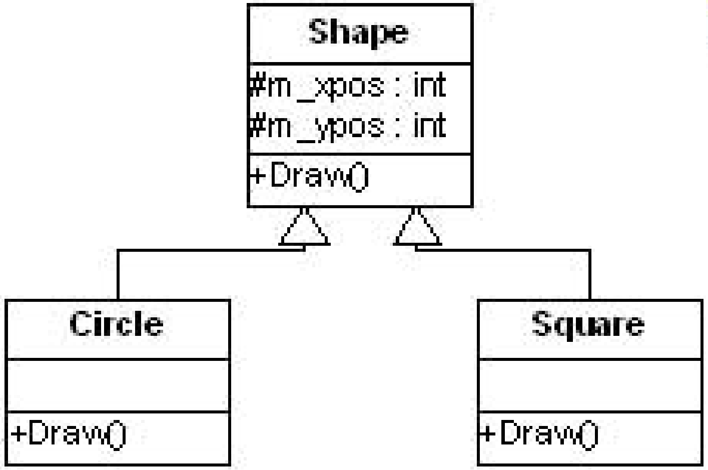
Virtual method

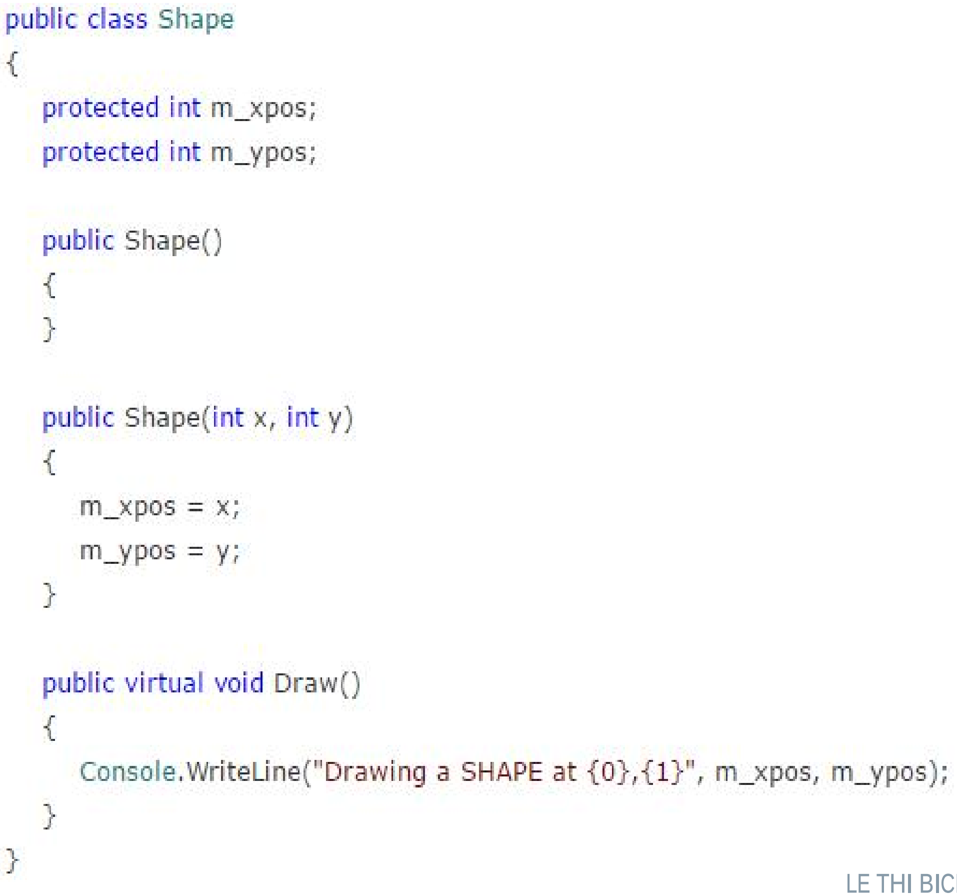
- Virtual method is method that can be used in the same way on instances of base and derived classes but its implementation is different
- A method is said to be a virtual when it is declared as `virtual`
- Methods that are declared as virtual in a base class can be overridden using the keyword `override` in the derived class

Casting operations

- Up-casting:
  - Cast from the base class type to the derived type.
  - Up-casting implicit and is safe.
- Down-casting:
  - Cast
  - Down-casting is explicit cast and is potentially unsafe
  - Be very aware that explicit casting is evaluated at runtime, not compile time.

Casting operations

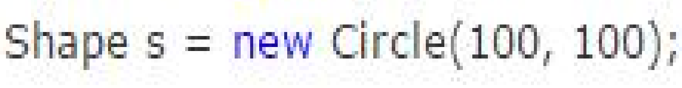




```
public class Shape
{
    protected int m_xpos;
    protected int m_ypos;

    public Shape()
    {
    }

    public Shape(int x, int y)
    {
        m_xpos = x;
        m_ypos = y;
    }

    public virtual void Draw()
    {
        Console.WriteLine("Drawing a SHAPE at {0},{1}", m_xpos, m_ypos);
    }
}
```

Casting operations

- Up-casting



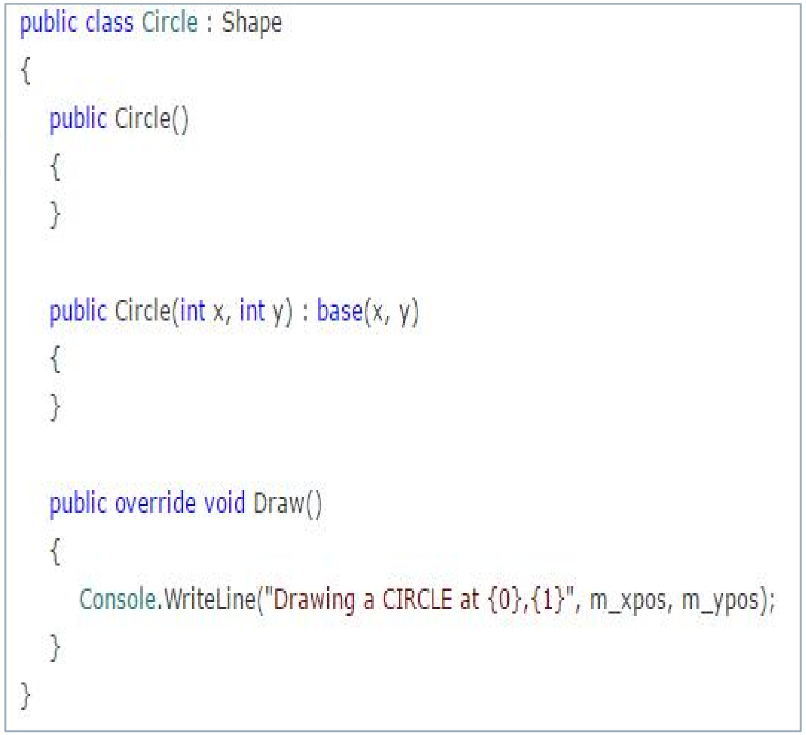
```
Shape s = new Circle(100, 100);
```

- Down-casting

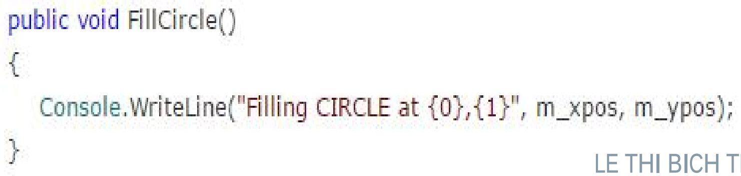
Shape s = new Circle(100, 100);
~~s.fillCircle();~~

Circle c;
c = (Circle)s;
c.fillCircle()

//also write 3 lines above to
((Circle)s).fillCircle()



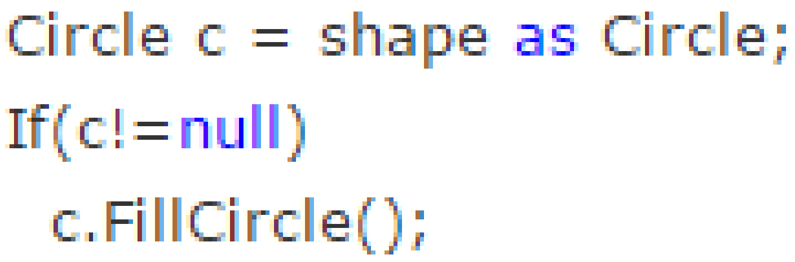
```
public class Circle : Shape
{
    public Circle()
    {
    }

    public Circle(int x, int y) : base(x, y)
    {
    }

    public override void Draw()
    {
        Console.WriteLine("Drawing a CIRCLE at {0},{1}", m_xpos, m_ypos);
    }
}
```



```
public void FillCircle()
{
    Console.WriteLine("Filling CIRCLE at {0},{1}", m_xpos, m_ypos);
}
```

as keyword

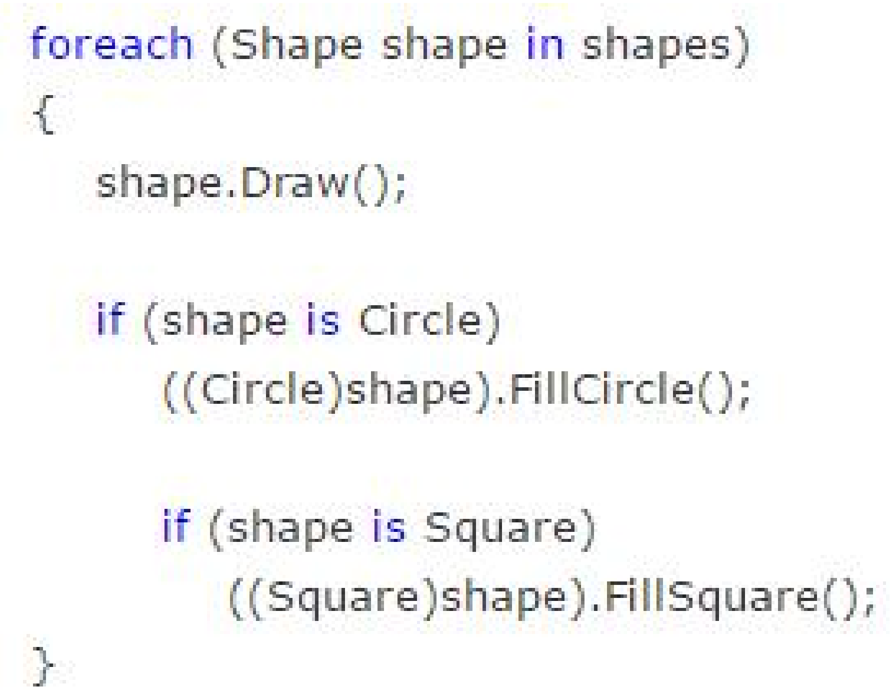
- How to make your program without a runtime exception?
- C# provides the as keyword to quickly determine at runtime whether a given type is compatible with another by checking against a null return value



```
Circle c = shape as Circle;
If(c!=null)
  c.FillCircle();
```

is keyword

- is keyword to determine whether two items are compatible
- the is keyword returns false if the types are incompatible



```
foreach (Shape shape in shapes)
{
    shape.Draw();

    if (shape is Circle)
        ((Circle)shape).FillCircle();

        if (shape is Square)
            ((Square)shape).FillSquare();
}
```

Abstract class

- A class can be declare as abstract class by putting the keyword abstract before the class definition.
- Syntax:

```
public abstract class <class name>{
 }
```

Abstract class

- Purpose:
  - To provide a common definition of a base class that multiple derived classes can share.
  - To implement polymorphism.
- An abstract class cannot be instantiated.

Abstract class

- An abstract class can contains non-abstract members and abstract members.
- An abstract member has a signature but no function body and they must be overridden in any derived class.
- Abstract members can be: methods, properties, indexers.
- E.g:

Exercises

- Ex1: Figure, Circle , Square with same method: caculArea()
- Ex2: Human with first name and last name.
  - `Student` which is derived from `Human` and has new field – `grade`
  - `Worker` derived from `Human` with new field `weekSalary` and work-hours per day and method `MoneyPerHour()`
  - Initialize an array of 10 Humans and then display information of each objects.

Lession 3: Interface

- Interface
- Purpose
- Interface vs Abstract class
- Enumerable Interface
- IComparable and IComparer Interface

Interface

- Interface declaration is like a class declaration, but it provides no implementation for its members, since all its members are implicitly abstract.
- Classes and structs that implement the interface must implement all members of the interface.
- In interface can contain only methods, properties, event, and indexers.
- An interface can not be instantiated directly.

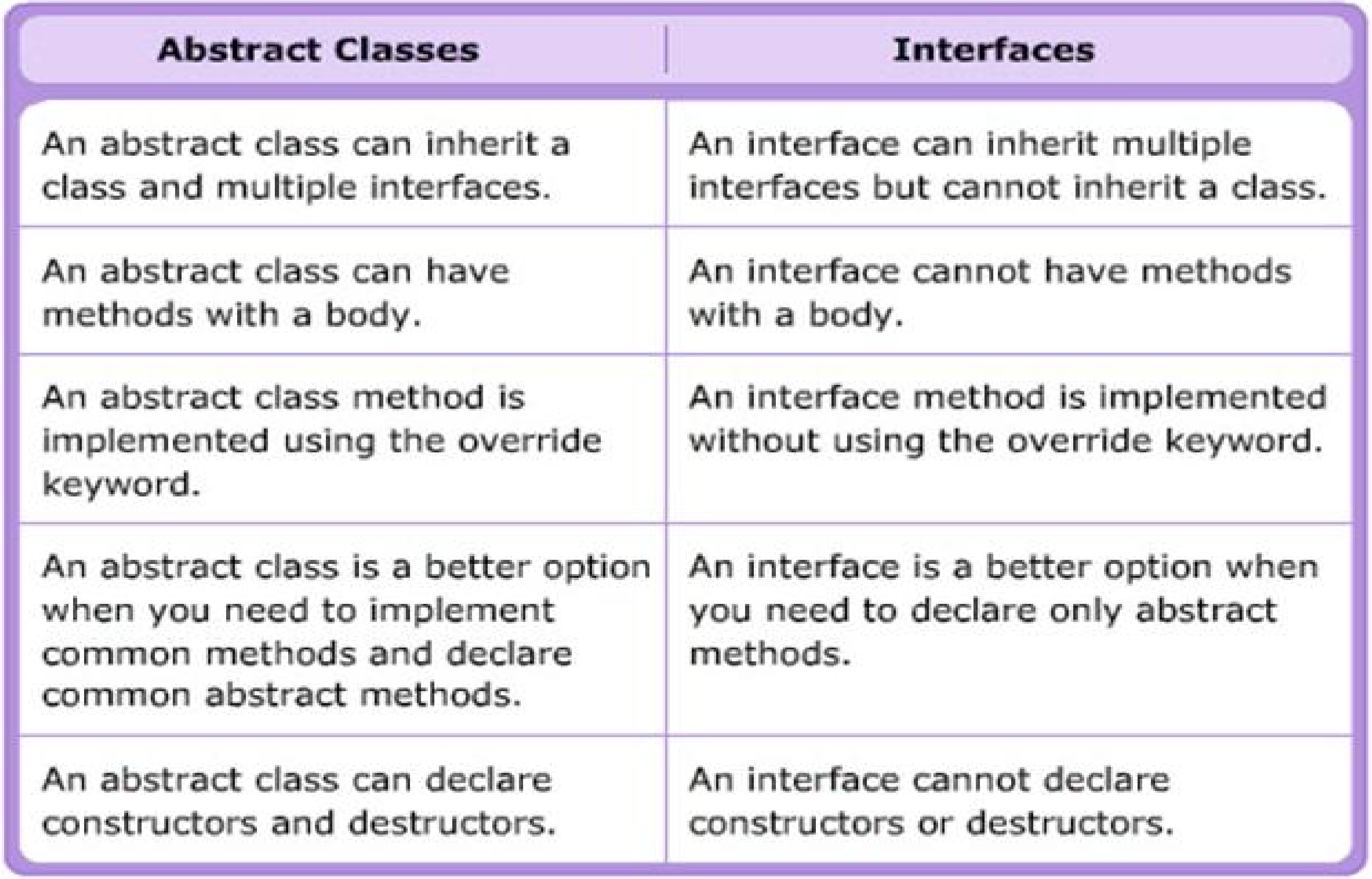
Interface

- Purpose: Multiple inheritance
  - A class (or struct) can implement more than one interface (But one class can only inherit from one base class).
  - A interface can inherit from one or more base interfaces by the inheriting subclass.
  - Both can inherit from multiple interface.

Interface vs Abstract class

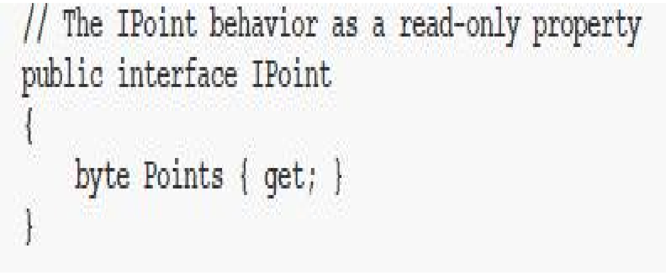
- Similiars:
  - Neither an abstract class nor an interface cannot be instantated.
  - Both contain abstract members which are implement

Interface vs Abstract class

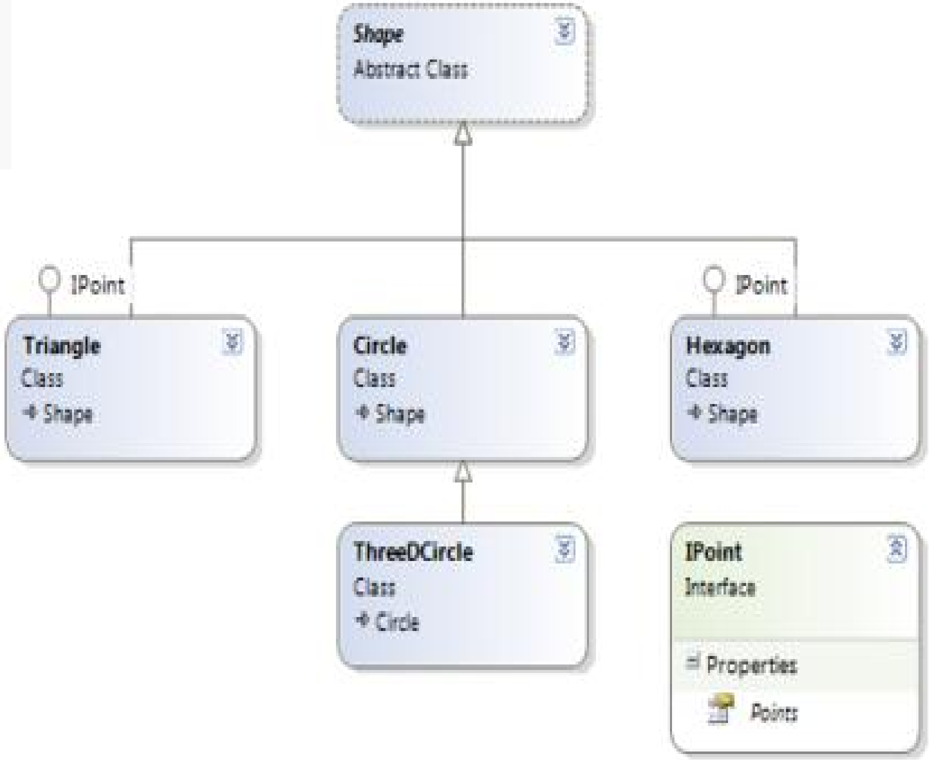


Abstract Classes	Interfaces
An abstract class can inherit a class and multiple interfaces.	An interface can inherit multiple interfaces but cannot inherit a class.
An abstract class can have methods with a body.	An interface cannot have methods with a body.
An abstract class method is implemented using the override keyword.	An interface method is implemented without using the override keyword.
An abstract class is a better option when you need to implement common methods and declare common abstract methods.	An interface is a better option when you need to declare only abstract methods.
An abstract class can declare constructors and destructors.	An interface cannot declare constructors or destructors.

Interfaces

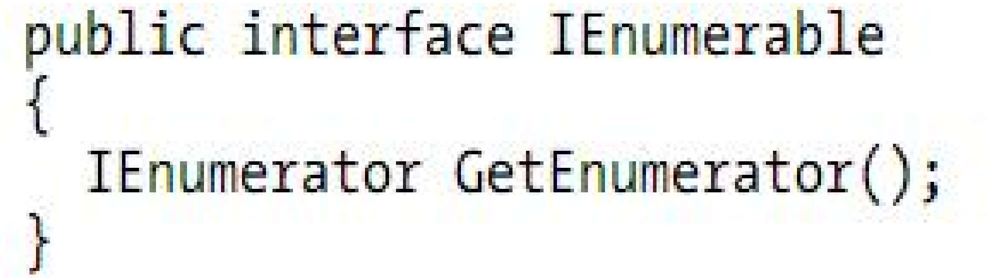


```
// The IPoint behavior as a read-only property
public interface IPoint
{
    byte Points { get; }
}
```



Enumerable Interface

- Definition:

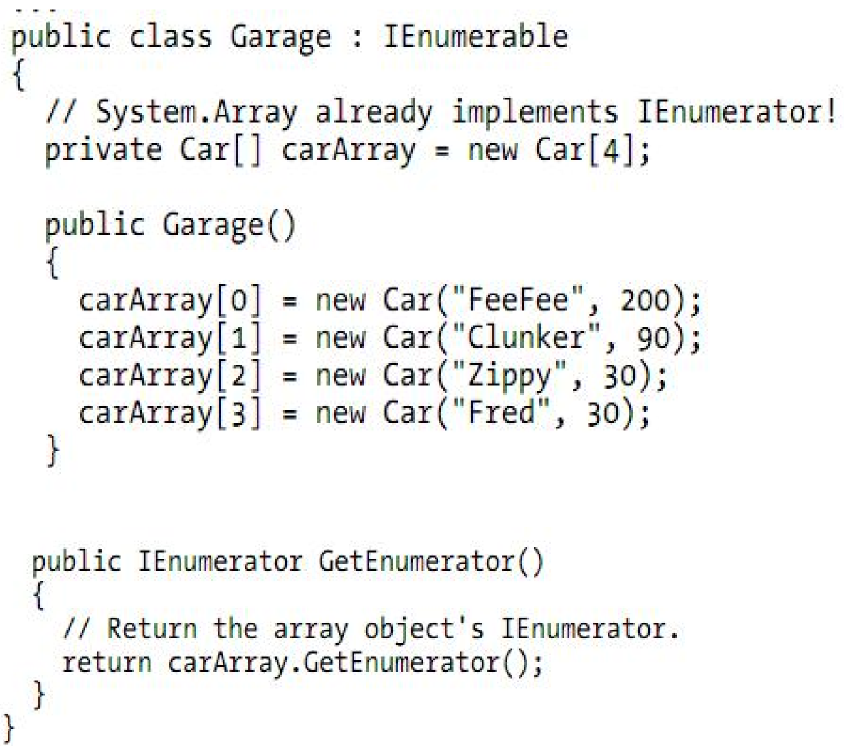


```
public interface IEnumerable
{
   IEnumerator GetEnumerator();
}
```

- The GetEnumerator() method returns a reference to yet another interface named System.Collections.IEnumerator

Enumerable Interface

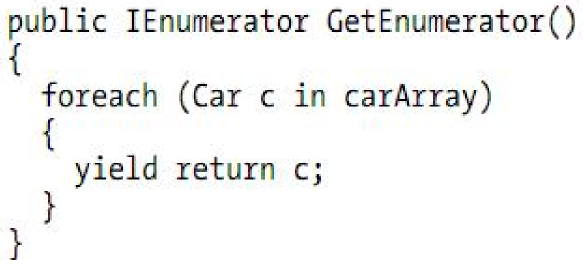
- Implement IEnumerable interface



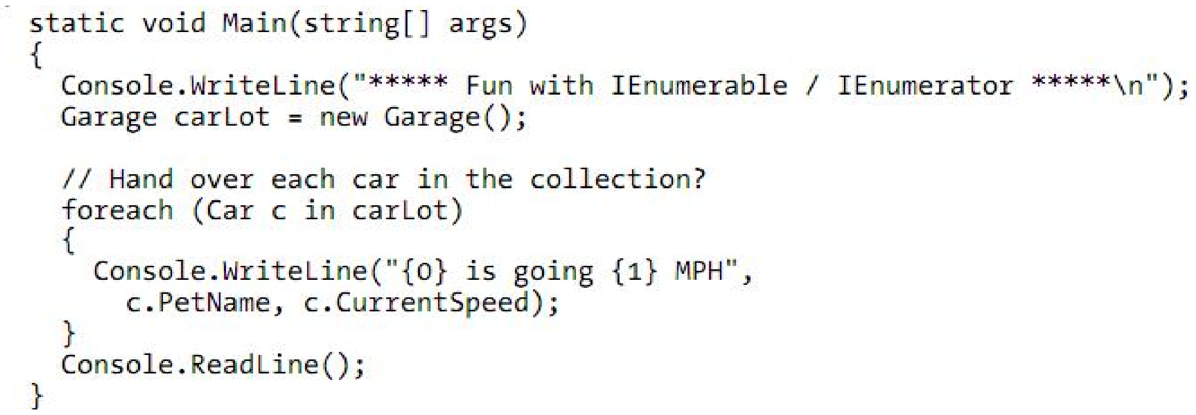
```
...
public class Garage : IEnumerable
{
  // System.Array already implements IEnumerator!
  private Car[] carArray = new Car[4];

  public Garage()
  {
    carArray[0] = new Car("FeeFee", 200);
    carArray[1] = new Car("Clunker", 90);
    carArray[2] = new Car("Zippy", 30);
    carArray[3] = new Car("Fred", 30);
  }


 public IEnumerator GetEnumerator()
 {
   // Return the array object's IEnumerator.
   return carArray.GetEnumerator();
 }
}
```

Enumerable Interface

› The yield keyword is used to specify the value (or values) to be returned to the caller’s foreach construct.



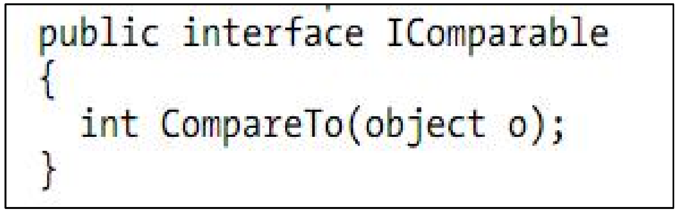
```
public IEnumerator GetEnumerator()
{
  foreach (Car c in carArray)
  {
    yield return c;
  }
}
```



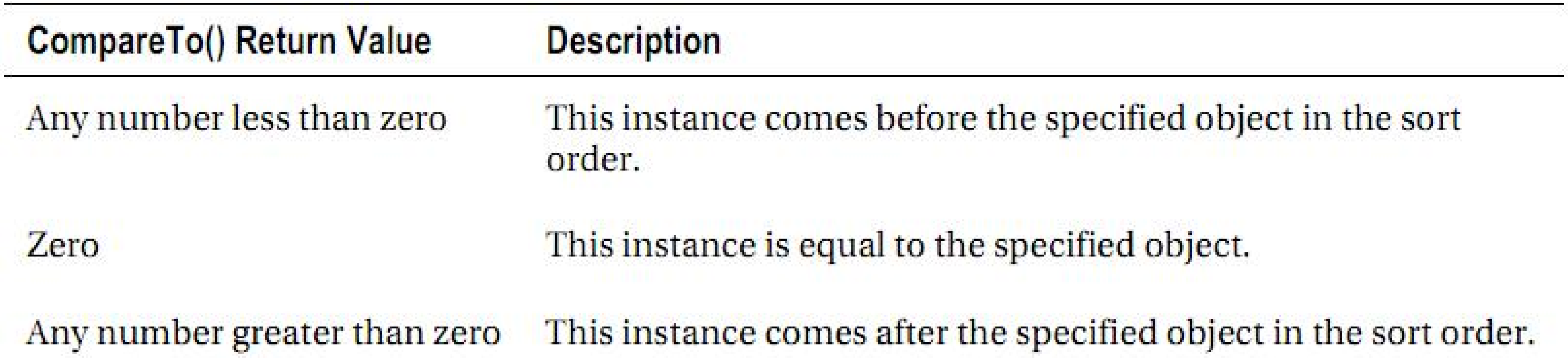
```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with IEnumerable / IEnumerator *****\n");
  Garage carLot = new Garage();

  // Hand over each car in the collection?
  foreach (Car c in carLot)
  {
    Console.WriteLine("{0} is going {1} MPH",
      c.PetName, c.CurrentSpeed);
  }
  Console.ReadLine();
}
```

IComparable Interface

- The IComparable interface specifies a behavior that allows an object to be sorted based on some specified key.
- Definition:



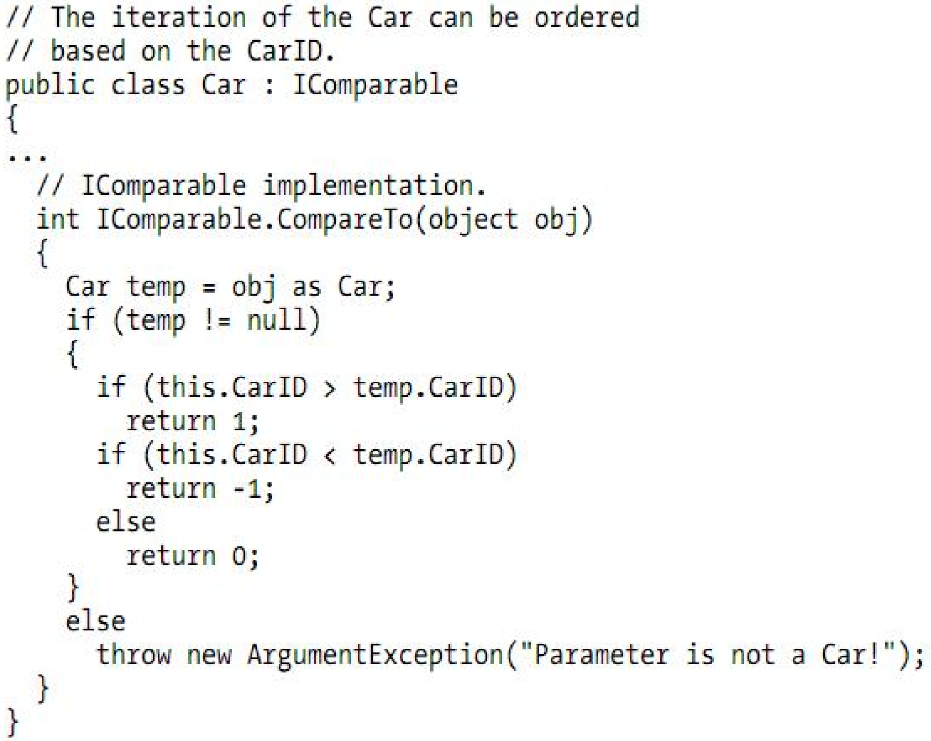
```
public interface IComparable
{
  int CompareTo(object o);
}
```



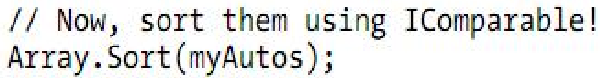
CompareTo() Return Value	Description
Any number less than zero	This instance comes before the specified object in the sort order.
Zero	This instance is equal to the specified object.
Any number greater than zero	This instance comes after the specified object in the sort order.

IComparable Interface

- Implement IComparable



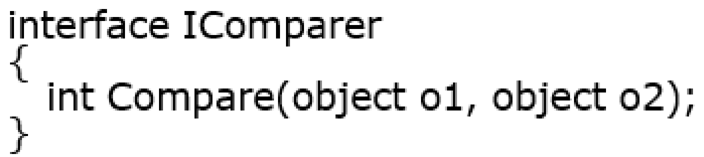
```
// The iteration of the Car can be ordered
// based on the CarID.
public class Car : IComparable
{
...
  // IComparable implementation.
  int IComparable.CompareTo(object obj)
  {
    Car temp = obj as Car;
    if (temp != null)
    {
      if (this.CarID > temp.CarID)
        return 1;
      if (this.CarID < temp.CarID)
        return -1;
      else
        return 0;
    }
    else
      throw new ArgumentException("Parameter is not a Car!");
  }
}
```



```
// Now, sort them using IComparable!
Array.Sort(myAutos);
```

IComparer Interface

- Specifying Multiple Sort Orders
- IComparer interface is defined within the System.Collections namespace as follows:

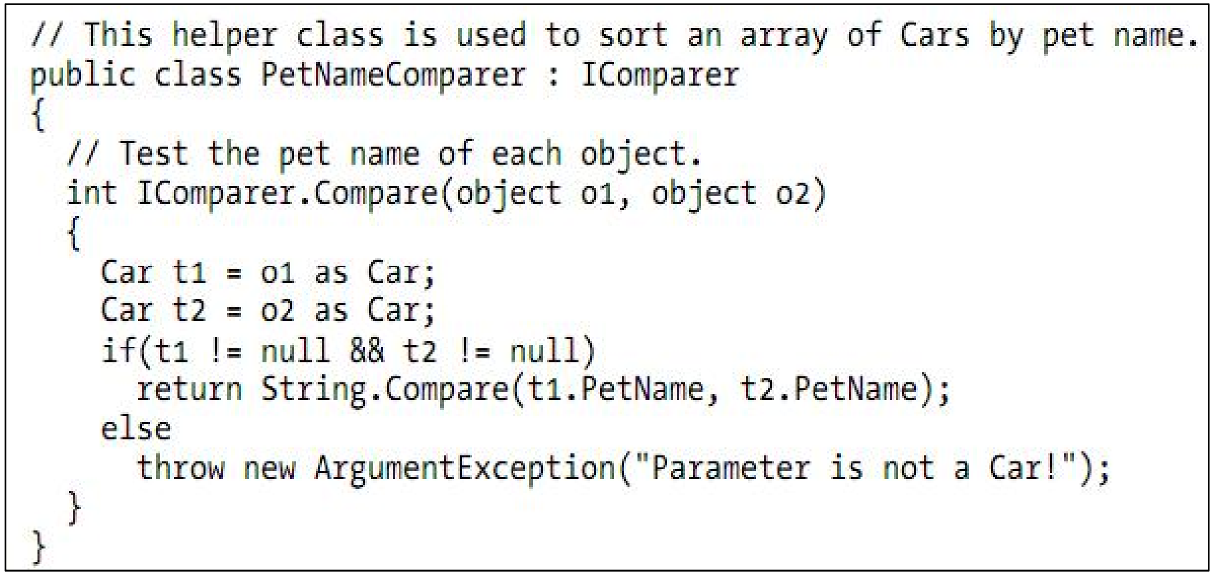


```
interface IComparer
{
   int Compare(object o1, object o2);
}
```

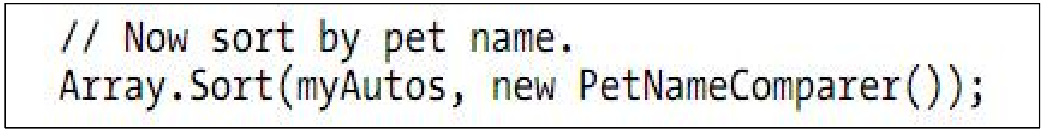
- You must implement this interface on any number of helper classes, one for each sort order

IComparer Interface

- E.g:



```
// This helper class is used to sort an array of Cars by pet name.
public class PetNameComparer : IComparer
{
  // Test the pet name of each object.
  int IComparer.Compare(object o1, object o2)
  {
    Car t1 = o1 as Car;
    Car t2 = o2 as Car;
    if(t1 != null && t2 != null)
      return String.Compare(t1.PetName, t2.PetName);
    else
      throw new ArgumentException("Parameter is not a Car!");
  }
}
```



```
// Now sort by pet name.
Array.Sort(myAutos, new PetNameComparer());
```

IComparer Interface

- Custom static property in order to help the object user along when sorting by a specific data point.

```
public class Car : IComparable
{
  ...
  // Property to return the PetNameComparer.
  public static IComparer SortByPetName
  { get
              {
                 return (IComparer)new PetNameComparer();
              }
   }
}
```

```
// Sorting by pet name made a bit cleaner.
Array.Sort(myAutos, Car.SortByPetName);
```

Exercise 1

- Write a base class named: MenuItem. It should have a name. It should have a method called printToScreen, to print the name to screen; this method should be virtual so that we can implement polymorphism. It should have a set method or a constructor, or both, related member variable name.
- Write 2 derived classes: Beverage and Snack.
- Beverage should have 3 prices: small, medium and large sizes.
- It should have a method named printToScreen. And you should implement the keyword override
- Please write a constructor or a set method to handle the prices of the three sizes
- Snack has 1 member variable: price
- It also has printToScreen method
- It should have a constructor, or a set method to handle the price of a snack object.
- In the main program, declare an array of 7 objects of MenuItem
- Then use 4 elements to store information of the drinks as shown below.
- The last 3 elements will have information of 3 snack items
- Then display the menu similarly to the screen below.

Exercises 2

- Create a base class and name this class as Employee. It should have 1 member variable: empName.
- 1 constructor so that the employee name can be saved for each object.
- A method called calcPaidCheck. And it is a virtual method
- Create a derived class named HourlyWorker
- Create an override method calcPaidCheck. Paid amount = hourly rate x hours worked

- Create a derived class name SalaryWorker
- Create an override method CalcPaidCheck and the paid amount with be annual-salary/12
- In the main method, create an array of Employee, and test with 2 worker employees and 2
- salary employees

Thank You !