Exercises and solutions
Abstract classes, Interfaces, and Patterns

8.1 Parameter variance

First, be sure you understand the co-variance problem stated above. Why is it problematic to execute `aref.Op(sref)` in the class Client?

The parameter variance problem, and the distinction between covariance and contravariance, is not really a topic in C#. The program with the classes A/B/S/T on the previous page compiles and runs without problems. Explain why!

Solution

The program with the classes A/B/S/T is OK because the methods `A.Op` and `B.Op` are different from each other due to the rules about overloading. Recall that two methods of the same name, but with different formal parameter types, can coexist without problems in C#. When one of the methods are called, the static type of the formal parameters are used to find out, which of the methods to call.

Thus, `aref.Op(sref)` call `A.OP`, and not `B.OP`. And therefore there are no problems (no bomb) whatsoever.

8.2 A specialization of Stack

On the slide to which this exercise belongs, we have shown an abstract class `Stack`.

It is noteworthy that the abstract `Stack` is programmed without any instance variables (that is, without any data representation of the stack). Notice also that we have been able to program a single non-abstract method `ToggleTop`, which uses the abstract methods `Top`, `Pop`, and `Push`.

Make a non-abstract specialization of `Stack`, and decide on a reasonable data representation of the stack.

In this exercise it is OK to ignore exception/error handling. You can, for instance, assume that the capacity of the stack is unlimited; That popping an empty stack an empty stack does nothing; And that the top of an empty stack returns the string "Not Possible". In a later lecture we will revisit this exercise in order to introduce exception handling. Exception handling is relevant when we work on full or empty stacks.

Write a client of your stack class, and demonstrate the use of the inherited method `ToggleTop`. If you want, you can also adapt my stack client class .

Solution

Here is my version of the non-abstract specialization of the abstract class `Stack`:

```
using System;

public class StackWithArray: Stack{

  private Object[] elements;
  private int nextPush;
  private int MaxSize;

  public StackWithArray(int MaxSize){
    elements = new Object[MaxSize];
    nextPush = 0;
    this.MaxSize = MaxSize;
  }

   public override void Push(Object el){
     if (!Full){
       elements[nextPush] = el;
       nextPush++;
    }
  }

   public override void Pop(){
   if (!Empty)
     nextPush--;
   }

   public override Object Top{
    get{
      if (!Empty)
         return elements[nextPush-1];
     else return "not possible";}
   }

   public override bool Full{
    get{return nextPush >= MaxSize;}
   }

   public override bool Empty{
```

```
     get{return nextPush == 0;}
    }

    public override int Size{
     get{return nextPush;}
    }

   public override String ToString(){
     string res = "Stack: ";
     for(int i = 0; i < nextPush; i++){
       res += elements[i] + " ";}
     return res;
   }

}
```

Here follows a sample client program:

```
using System;

class C{

  public static void Main(){

    Stack s = new StackWithArray(10);

    Console.WriteLine("Empty: {0}", s.Empty);
    Console.WriteLine("Full: {0}", s.Full);

    s.Push(5); s.Push(6);  s.Push(7);
    s.Push(15); s.Push(16);  s.Push(17);
    s.Push(18); s.Push(19);  s.Push(20);
    s.Push(21);

    Console.WriteLine("{0}", s.Size);
    Console.WriteLine("{0}", s.Top);
    Console.WriteLine("{0}", s);

    s.Pop();s.Pop();s.Pop();s.Pop();s.Pop();s.Pop();s.Pop();

    Console.WriteLine("Empty: {0}", s.Empty);
    Console.WriteLine("Full: {0}", s.Full);
    Console.WriteLine("Top: {0}", s.Top);
```

```
      Console.WriteLine("{0}", s);
      s.ToggleTop();
      Console.WriteLine("{0}", s);

    }

  }
```

8.3 Course and Project classes

In the earlier exercise about courses and projects (found in the lecture about classes) we programmed the classes `BooleanCourse`, `GradedCourse`, and `Project`. Revise and reorganize your solution (or the model solution) such that `BooleanCourse` and `GradedCourse` have a common abstract superclass called `Course`.

Be sure to implement the method `Passed` as an abstract method in class `Course`.

In the `Main` method (of the client class of `Course` and `Project`) you should demonstrate that both boolean courses and graded courses can be referred to by variables of static type `Course`.

Solution

Here is my solution:

```
using System;

public abstract class Course{      // Abstract class

  private string name;

  public Course(string name){      // ... with constructor
    this.name = name;
  }

  public abstract bool Passed();

}

public class BooleanCourse: Course{   // Inherits from Course
```

```
  private bool grade;

  public BooleanCourse(string name, bool grade): base(name){
    this.grade = grade;
  }

  public override bool Passed(){
    return grade;
  }

}

public class GradedCourse: Course{   // Inherits from Course

  private int grade;

  public GradedCourse(string name, int grade): base(name){
    this.grade = grade;
  }

  public override bool Passed(){
    return grade >= 2;
  }

}

public class Project{

  private Course c1, c2, c3, c4;   // Course is the common type
                                   // of all four variables.

  public Project(Course c1, Course c2, Course c3, Course c4){   // All parameters of type Course
    this.c1 = c1; this.c2 = c2;
    this.c3 = c3; this.c4 = c4;
  }

  public bool Passed(){
    return
     (c1.Passed() && c2.Passed() && c3.Passed() && c4.Passed()) ||
     (!(c1.Passed()) && c2.Passed() && c3.Passed() && c4.Passed()) ||
     (c1.Passed() && !(c2.Passed()) && c3.Passed() && c4.Passed()) ||
     (c1.Passed() && c2.Passed() && !(c3.Passed()) && c4.Passed()) ||
     (c1.Passed() && c2.Passed() && c3.Passed() && !(c4.Passed()));
  }
```

```
}

public class Program {

  public static void Main(){
    Course c1 = new BooleanCourse("Math", true),          // All variables declared of type Course
           c2 = new BooleanCourse("Geography", true),
           c3 = new GradedCourse("Programming", 0),
           c4 = new GradedCourse("Algorithms", -3);

    Project p = new Project(c1, c2, c3, c4);

    Console.WriteLine("Project Passed: {0}", p.Passed());
  }

}
```

Please take notice of the comments in the source program.

8.4 The interface ITaxable

For the purpose of this exercise you are given a couple of very simple classes called `Bus` and `House`. Class `Bus` specializes the class `Vehicle`. Class `House` specializes the class `FixedProperty`. All the classes are here.

First in this exercise, program an interface `ITaxable` with a parameterless operation `TaxValue`. The operation should return a decimal number.

Next, program variations of class `House` and class `Bus` which implement the interface `ITaxable`. Feel free to invent the concrete taxation of houses and busses. Notice that both class `House` and `Bus` have a superclass, namely `FixedProperty` and `Vehicle`, respectively. Therefore it is essential that taxation is introduced via an interface.

Demonstrate that taxable house objects and taxable bus objects can be used together as objects of type `ITaxable`.

Solution

First, we show the given classes `Bus` and `House` together with their superclasses:

```
using System;

public class FixedProperty{
```

```
  protected string location;
  protected bool inCity;
  protected decimal estimatedValue;

  public FixedProperty(string location, bool inCity, decimal value){
    this.location = location;
    this.inCity = inCity;
    this.estimatedValue = value;
  }

  public FixedProperty(string location):
    this(location,true,1000000){
  }

  public string Location{
    get{
      return location;
    }
  }
}

public class Vehicle{

  protected int registrationNumber;
  protected double maxVelocity;
  protected decimal value;

  public Vehicle(int registrationNumber, double maxVelocity,
                 decimal value){
    this.registrationNumber = registrationNumber;
    this.maxVelocity = maxVelocity;
    this.value = value;
  }

  public int RegistrationNumber{
    get{
      return registrationNumber;
    }
  }

}

public class Bus: Vehicle{

  protected int numberOfSeats;
```

```
  public Bus(int numberOfSeats, int regNumber, decimal value):
      base(regNumber, 80, value){
    this.numberOfSeats = numberOfSeats;
  }

  public int NumberOfSeats{
    get{
     return numberOfSeats;
    }
  }

}

public class House: FixedProperty {

  protected double area;

  public House(string location, bool inCity, double area,
               decimal value):
      base(location, inCity, value){
   this.area = area;
  }

  public double Area{
    get{
     return area;
    }
  }
}
```

Here follows the the interface `ITaxable` together with the classes `TaxableBus` and `TaxableHouse`. Notice the way they the two classes implement the interface `ITaxable`. At the bottom, the class `App` serves as a client that demonstrates how taxable houses and taxable busses can be used together.

```
using System;

interface ITaxable{
  decimal TaxValue();
}

class TaxableBus: Bus, ITaxable {

  public TaxableBus(int numberOfSeats, int regNumber, decimal value):
    base(numberOfSeats, regNumber, value){
```

```
  }

  public decimal TaxValue(){
    return (value / 10) + 105M * numberOfSeats;
  }

}

class TaxableHouse: House, ITaxable {

  public TaxableHouse(string location, bool inCity, double area, decimal value):
    base(location, inCity, area, value){
  }

  public decimal TaxValue(){
    if (inCity)
      return (estimatedValue / 1000.0M) * 5M + 5M * (decimal)area;
    else
      return (estimatedValue / 1000.0M) * 3M;
  }
}


class App{

  public static void Main(){

     ITaxable[] tObjects =
        new ITaxable[]{
           new TaxableHouse("Aalborg", true, 700, 500000M),
           new TaxableHouse("Skjern", true, 1800, 300000M),
           new TaxableBus(50, 12345, 300000M),
           new TaxableBus(10, 345678, 25000M)};

     foreach(ITaxable it in tObjects)
       Console.WriteLine("{0}", it.TaxValue());
  }
}
```

8.5 An abstract GameObject class

On the slide, to which this exercise belongs, we have written an interface `IGameObject` which is implemented by both class `Die` and class `Card`.

Restructure this program such that class `Die` and class `Card` both inherit an abstract class `GameObject`. You should write the class `GameObject`.

The client program should survive this restructuring. (You may, however, need to change the name of the type `IGameObject` to `GameObject`). Compile and run the given client program with your classes.

Solution

The abtract class `GameObject` is here:

```
public enum GameObjectMedium {Paper, Plastic, Electronic}

public abstract class GameObject{

  public abstract int GameValue{
    get;
  }

  public abstract GameObjectMedium Medium{
    get;
  }
}
```

The class `Die` specializes the abstract class `GameObject`:

```
using System;

public class Die: GameObject {
  private int numberOfEyes;
  private Random randomNumberSupplier;
  private readonly int maxNumberOfEyes;

  public Die (): this(6){}

  public Die (int maxNumberOfEyes){
    randomNumberSupplier =
      new Random(unchecked((int)DateTime.Now.Ticks));
    this.maxNumberOfEyes = maxNumberOfEyes;
    numberOfEyes = NewTossHowManyEyes();
  }

  public void Toss (){
    numberOfEyes = NewTossHowManyEyes();
  }
```

```
  private int NewTossHowManyEyes (){
    return randomNumberSupplier.Next(1,maxNumberOfEyes + 1);
  }

  public int NumberOfEyes() {
    return numberOfEyes;
  }

  public override String ToString(){
    return String.Format("Die[{0}]: {1}", maxNumberOfEyes, numberOfEyes);
  }

  public override int GameValue{
    get{
      return numberOfEyes;
    }
  }

  public override GameObjectMedium Medium{
    get{
      return
       GameObjectMedium.Plastic;
    }
  }

}
```

Similarly, the class `Card` specializes the abstract class `GameObject`:

```
using System;

public class Card: GameObject{
  public enum CardSuite { spades, hearts, clubs, diamonds };
  public enum CardValue { two = 2, three = 3, four = 4, five = 5,
                          six = 6, seven = 7, eight = 8, nine = 9,
                          ten = 10, jack = 11, queen = 12, king = 13,
                          ace = 14 };

  private CardSuite suite;
  private CardValue value;

  public Card(CardSuite suite, CardValue value){
    this.suite = suite;
```

```
    this.value = value;
  }

  public CardSuite Suite{
    get { return this.suite; }
  }

  public CardValue Value{
    get { return this.value; }
  }

  public override String ToString(){
    return String.Format("Suite:{0}, Value:{1}", suite, value);
  }

  public override int GameValue{
    get { return (int)(this.value); }
  }

  public override GameObjectMedium Medium{
    get{
      return GameObjectMedium.Paper;
    }
  }
}
```

Here is the client class, which shows a mixed used of dies and playing cards on the basis of the abstract class `GameObject`:

```
using System;
using System.Collections.Generic;

class Client{

  public static void Main(){

    Die d1 = new Die(),
        d2 = new Die(10),
        d3 = new Die(18);

    Card cs1 =  new Card(Card.CardSuite.spades, Card.CardValue.queen),
         cs2 =  new Card(Card.CardSuite.clubs, Card.CardValue.four),
         cs3 =  new Card(Card.CardSuite.diamonds, Card.CardValue.ace);
```

```
      List lst = new List();

      lst.Add(d1);  lst.Add(d2);  lst.Add(d3);
      lst.Add(cs1); lst.Add(cs2); lst.Add(cs3);

      foreach(GameObject gao in lst){
        Console.WriteLine("{0}: {1} {2}",
                          gao, gao.GameValue, gao.Medium);
      }
    }
  }
```

Only few modifications where necessary in our transition from the interface `IGameObject` to the abstract class `GameObject:`

1. The methods in `GameObject` must use the modifiers **`public`** and **`abstract`**.
2. In `Die` and `Card`, the methods `GameValue` and `Medium` must **`override`** the abstract methods.
3. We must change the name of the type `IGameObject` to `GameObject` in the client program.

8.6 Comparable Dice

In this exercise we will arrange that two dice can be compared to each other. The result of `die1.CompareTo(die2)` is an integer. If the integer is negative, `die1` is considered less than `die2`; If zero, `die1` is considered equal to `die2`; And if positive, `die1` is considered greater than `die2`. When two dice can be compared to each other, it is possible sort an array of dice with the standard `Sort` method in C#.

Program a version of class `Die` which implements the interface `System.IComparable`.

Consult the documentation of the (overloaded) static method `System.Array.Sort` and locate the `Sort` method which relies on `IComparable` elements.

Make an array of dice and sort them by use of the `Sort` method.

Solution

We show a solution which relies on the non-generic version of class `IComparable`

The listing below is relatively long. You should focus on the method `CompareTo` in class `Die`. Notice the necessary casting of `other` to `Die` in the body of `CompareTo`.

Also notice the use of `Array.Sort(Array)` at the very bottom of the client class `App`. There are many overloaded `Sort` methods in class `Array`. According to the documentation, the method `Array.Sort(Array)` relies on `Icomparable` elements.

```
using System;

public class Die: IComparable {
  private int numberOfEyes;
  private Random randomNumberSupplier;
  private const int maxNumberOfEyes = 6;

  public Die(){
    randomNumberSupplier = Random.Instance();
    numberOfEyes = NewTossHowManyEyes();
  }

  public void Toss(){
    numberOfEyes = NewTossHowManyEyes();
  }

  private int NewTossHowManyEyes (){
    return randomNumberSupplier.Next(1,maxNumberOfEyes + 1);
  }

  public int NumberOfEyes() {
    return numberOfEyes;
  }

  public override String ToString(){
    return String.Format("[{0}]", numberOfEyes);
  }

  public int CompareTo(Object other){
    return this.numberOfEyes.CompareTo(((Die)other).numberOfEyes);
  }
}

public class Random {

  // Singleton pattern:
  // Keeps track of unique instance of this class
  private static Random uniqueInstance = null;

  // Holds the instance of System.Random
  private System.Random systemRandom;
```

```
  // Singleton pattern: Private constructor.
  private Random(){
    systemRandom = new System.Random(unchecked((int)DateTime.Now.Ticks));
  }

  public static Random Instance(){
    if (uniqueInstance == null)
      uniqueInstance = new Random();
    return uniqueInstance;
  }

  public int Next(int lower, int upper){
    // delegate to systemRandom
    return systemRandom.Next(lower,upper);
  }

}

class App{

  public static void Main(){

    Die[] dice = new Die[]{new Die(), new Die(), new Die(), new Die(),
                           new Die(), new Die(), new Die(), new Die(),
                          };
    Console.WriteLine("Before sorting");
    foreach(Die d in dice)
      Console.Write("{0} ", d);

    Console.WriteLine();

    Console.WriteLine("After sorting");
    Array.Sort(dice);
    foreach(Die d in dice)
      Console.Write("{0} ", d);

  }

}
```

The following output appears when I run the program:

```
 Before sorting
```

```
[3] [6] [3] [6] [5] [3] [1] [1]
After sorting
[1] [1] [3] [3] [3] [5] [6] [6]
```

You do probably not get the same output if you run the program, because randomness is involved in the class `Die`.

Generated: Monday February 7, 2011, 12:18:08