Contents

Properties (Solutions)

Questions

1. What is the right syntax for an automatic property? *Select all that apply.*

   ☒ `public int Width { get; set; }`
   ☒ `public int Width { set; get; }`
   ☐ `public int Width { Set; Get; }`
   ☐ `public int Width { Get; Set; }`
   ☐ `public int Width { set(); get();}`
   ☐ `public int Width { get(); set();}`

2. Which of the following statements is *false*?

   ☐ Properties can be static.
   ☒ `get` and `set` accessors must always have bodies.
   ☐ Properties have headers.
   ☐ `get` and `set` accessors correspond to "getter" and "setter" methods for attributes.

3. Consider the following implementation of a class called `Pet`:

```
class Pet{
    private string name;
    public string Name{
        get;
        set { name = value; }
    }
}
```

   This code will give a compilation error. Why?

   ☒ The `set` accessor has a body, but the `get` accessor does not.
   ☐ The instance variable for `name` is declared, but no value is assigned.
   ☐ `value` is not a keyword and hasn't been declared, so it is meaningless here.

☐ The access modifier for `name` is **`private`**, but it should be **`public`**.

Circle Example

For the following questions, imagine you've implemented a `Circle` class, with the attribute `private decimal diameter;` and a "getter" and "setter" method for that attribute. You've created an object in this `Circle` class called `myCircle`. If you were to implement the class with properties instead:

1. What would calling the `get` accessor do?

   ☒ Return the value of `diameter`
   ☐ Assign a value to `diameter`

2. What would calling the `set` accessor do?

   ☐ Return the value of `diameter`
   ☒ Assign a value to `diameter`

3. The statement `myCircle.GetDiameter();` would have to be rewritten. How would you rewrite it?

   ☒ `myCircle.Diameter;`
   ☐ `myCircle.diameter;`
   ☐ `Diameter.myCircle;`
   ☐ `myCircle = Diameter;`

4. The statement `myCircle.SetDiameter(5.0m);` would also need to be rewritten. How would you rewrite it?

   ☐ `myCircle.diameter = 5.0m;`
   ☐ `Diameter.myCircle(5.0m);`
   ☒ `myCircle.Diameter = 5.0m;`
   ☐ `myCircle.diameter(5.0m);`

You would now like to add a `Color` property of type `string` to your `Circle` class.

1. How would you declare the instance variable?

   ☐ `public color string;`
   ☐ `public string Color;`
   ☐ `private string Color;`
   ☒ `private string color;`

2. How would you format the property header?

   ☐ `public string color;`
   ☒ `public string Color;`
   ☐ `private Color string;`

- ☐ `private string color;`

3. What would the `get` accessor's body look like, in its most basic possible form?

   - ☐ `color;`
   - ☐ `color = value;`
   - ☒ `return color;`
   - ☐ `string color;`

4. What would the `set` accessor's body look like, in its most basic possible form?

   - ☐ `color;`
   - ☒ `color = value;`
   - ☐ `return color;`
   - ☐ `string color;`

5. In the `Main` method, you would like to assign the value `"yellow"` to `color`. Which statement would do that?

   - ☐ `yellow.myCircle;`
   - ☒ `myCircle.Color = "yellow";`
   - ☐ `myCircle.yellow = Color;`
   - ☐ `myCircle = "yellow";`

Plant Example

For the next questions, consider the following implementation of a class called `Plant`:

```
class Plant{
    private string species;
    public string Species
        {get;} = "Helianthus annus";
    private static bool hasChloroplasts;
    public static bool HasChloroplasts
        {get;} = true;
}
```

1. Will this code compile? Why or why not?

   - ☐ No, because there are no `set` accessors, and properties must have one.
   - ☐ No, because a property cannot be assigned a default value.
   - ☐ No, because a `get` accessor must always have a body.
   - ☒ Yes, because properties are *not* required to have `set` accessors.
   - ☐ Yes, because a default value must be assigned for each property.

Suppose you've created an object in the `Plant` class called `myPlant`.

1. In the Main method, what would the statement

   `Console.Write(myPlant.Species);`

   do?

   - ☒ Display the current value of `species`, `"Helianthus annus"`.
   - ☐ Rename the `myPlant` object to `Species`.
   - ☐ It won't do anything–the code for the class doesn't compile.
   - ☐ It won't do anything–the property is write-only.

2. The `HasChloroplasts` property is **`static`**. What does this mean? *Select all that apply.*

   - ☐ Every object in the `Plant` class has its own `HasChloroplasts` property.
   - ☒ The property is shared across the class and all of its instances.
   - ☒ The property can be accessed without creating a `Plant` object.
   - ☐ The property's value cannot be changed from the default.

3. The statement `myPlant.Species = "Coffea arabica";` would not compile. Why not?

   - ☐ The syntax is wrong.
   - ☐ Only a **`static`** property's default value can be changed.
   - ☐ The code for the class doesn't compile.
   - ☒ The property only has a `get` accessor, so it is read-only.

4. What modification to the `Plant` class would allow the statement `myPlant.Species = "Coffea arabica";` to compile?

   - ☐ Remove the default value, `"Helianthus annus"`.
   - ☒ Add **`set`**`;` to the `Species` property.
   - ☐ Add **`set`**`;` to the `HasChloroplasts` property.
   - ☐ Make the entire class **`static`**.
   - ☐ Change the access modifier for `species` from **`private`** to **`public`**

Problems

1. Consider the following implementation of a `Rectangle` class:

```
class Rectangle
{
  private int length;
  private int width;

  public void SetLength(int lengthParameter)
  {
    length = lengthParameter;
  }

  public int GetLength()
  {
    return length;
  }

  public void SetWidth(int widthParameter)
  {
    width = widthParameter;
  }

  public int GetWidth()
  {
    return width;
  }

  public int ComputeArea()
  {
    return length * width;
  }
}
```

(a) Write a `Main` method that

i. Creates a `Rectangle` object,
ii. Sets its width to 5,
iii. Sets its length to 10,
iv. Displays its area.

Solution

```
using System;
class Program
{
public static void Main()
{
    Rectangle test = new Rectangle(); // 1
    test.SetWidth(5); // 2
    test.SetLength(10); // 3
    Console.WriteLine(test.ComputeArea()); //
↪   4
}
}
```

(b) Write an implementation of the `Rectangle` class *using only properties* (included for the `ComputeArea()`).

Solution

```
class Rectangle{
  public int Length{get; set;}
  public int Width{get; set;}
  public int Area{get{return Length * Width;}}
}
```

(c) Write a `Main` method that performs the same tasks as above, but using the properties you just defined.

Solution

```
using System;
class Program
{
    public static void Main()
    {
        Rectangle test = new Rectangle(); // 1
        test.Width = 5; // 2
        test.Length = 10; // 3
        Console.WriteLine(test.Area); // 4
    }
}
```

2. Implement a `SDCard` class to represent SD cards. Add attributes to your answer if needed.

   (a) Implement a `Nickname` `string` property using automatic properties.
   Solution
   ```
   public string Nickname {get; set;}
   ```
   (b) Implement a `Capacity` `int` property whose setter raises an `ArgumentException` exception if the value passed as argument is not 8, 16, 32, 64 or 128. The getter should simply return the value stored.
   Solution
   ```
   private int capacity;
   public int Capacity {
       set {
           if (value == 8 || value == 16 || value ==
            ↪  32 || value == 64 || value == 128)
               capacity = value;
           else
               throw new ArgumentException();
        }
        get { return capacity; }
   }
   ```

(c) Implement a `CapacityInGb` `int` property with only a getter, that returns the `Capacity` times 8.
Solution

```
public int CapacityInGb {
    get {return capacity * 8;}
}
```

(d) Implement a `ToString` method that returns a `string` containing the nickname of the SD card, its capacity in gigabytes (GB, from question 2.), and its capacity in gigabits (Gb, from question 3.).

Solution

```
public override string ToString(){
    return "Name: " + Nickname + "\nCapacity: " +
     ↪  Capacity + "GB" + "\nCapacity in Gb: " +
     ↪  CapacityInGb + "Gb";
}
```

Solution

A complete solution gives:

```
using System;

class SDCard
{
  public string Nickname { get; set; }
  private int capacity;
  public int Capacity
  {
    set
    {
      if (
        value == 8
        || value == 16
        || value == 32
        || value == 64
        || value == 128
      )
        capacity = value;
      else
        throw new ArgumentException();
    }
    get { return capacity; }
  }
  public int CapacityInGb
  {
    get { return capacity * 8; }
  }

  public override string ToString()
  {
    return "Name: "
      + Nickname
      + "\nCapacity: "
      + Capacity
      + "GB"
      + "\nCapacity in Gb: "
      + CapacityInGb
      + "Gb";
  }
}
```

(Download this code)[1]

And a possible test program is:

```
using System;

class Program
{
  static void Main()
  {
    SDCard test = new SDCard();
    test.Nickname = "Blue";
    test.Capacity = 8;
    Console.WriteLine(test);
    try
    {
      test.Capacity = 7;
    }
    catch (Exception e)
    {
      Console.WriteLine(e.Message);
    }
  }
}
```

(Download this code)[2]

[1]code/projects/SDCard.zip
[2]code/projects/SDCard.zip