CIS 400: Object-Oriented Design, Implementation, and Testing

Fall 2024

Exam 1 – 100 points

**This test is closed-notes and closed-computers.**

There are 10 questions.

Name: \_\_\_\_\_\_\_\_\_\_\_\_Solution\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Score: \_\_\_\_\_\_\_\_\_\_\_\_

1. (3 pts) When should we use a private backing field for a property?
   1. Any time we need to give the property a default value
   2. Any time we are implementing a derived property
   3. Any time a property has a “set” accessor
   4. **Any time we need to enforce bounds on a property**
2. (3 pts) When do we use the “override” keyword in C#?
   1. When you want to provide a new implementation for a virtual member that is inherited from the base class
   2. When you want to implement an interface member in a class
   3. When you want to provide a new implementation for an abstract member that is inherited from the base class
   4. **More than one option is correct (BOTH for overriding virtual and abstract)**
3. (3 pts) An abstract class must:
   1. Include at least one abstract method or property
   2. Not include a constructor
   3. Include only abstract methods or properties
   4. **None of the above**
4. (3 pts) Suppose that class *Honeycrisp* extends class *Apple*, and that class *Apple* extends class *Fruit*. Given the following variable declarations:

Fruit (IMenuItem)

Apple (Entrée)

Honeycrisp (ClubSub)

Which of the following statements would compile?

~~I. Honeycrisp h = new Apple();~~

II. Fruit f1 = new Honeycrisp();

III. Fruit f2 = new Apple();

IV. ~~Apple a = new Fruit();~~

1. **II and III**
2. III only
3. I and IV
4. I only
5. (3 pts) Consider the following class definitions:

**public class Media {**

**public void Play() { }**

**public virtual void Stop() { }**

**}**

**public class MusicTrack : Media {**

**public void Pause() { }**

**}**

Given the following declaration:

**Media m = new MusicTrack();**

Which of the following will compile without error?

I. **m.Play();**

II. **m.Stop();**

III. (**m as MusicTrack).Pause();**

~~IV.~~ **~~m.Pause();~~**

* 1. II and III only
  2. I and II only
  3. II and IV only
  4. **I, II, and III**
  5. All of the above will compile

1. (16 pts) Consider the *Socks* class and *IItem* interface from the handout on the last page. As if you are inside the *Socks* class definition, write the necessary code to fulfill the requirements of the *IItem* interface. The count of a socks instance should be 2 initially and should be restricted to a positive even number (if an attempt is made to set it to a value outside these requirements, leave the count unchanged). The price of a socks order is $2.00 per sock, with a 20% discount if more than 4 socks are ordered. You should allow more specific kinds of socks to potentially have different price calculations. (You ONLY need to write code for *Socks* that is not already in the handout.)
2. (25 pts) Write the class *AthleticSocks*, which extends *Socks*. You may wish to include a constructor. Your *AthleticSocks* class should:

* Include a property to represent whether the socks are ankle height, which should initially be true. Include a get and an init accessor.
* Override the inherited *Price* property – the price of white ankle height athletic socks should be $3.00 per sock. Black and gray socks are an additional $1.00 per sock, and non-ankle height socks are an additional $1.50 per sock. There is no discount on bulk orders.

You should NOT make any changes to the *Socks* class in this problem.

public class AthleticSocks : Socks {

public bool Ankle {get; init;} = true;

public override decimal Price {

get {

decimal costPer = 3.00m;

if (Color != SockColor.White) costPer += 1.00m;

if (!Ankle) costPer += 1.50m;

return costPer\*Count;

}

}

public AthleticSocks(SockColor c) : base(c) {}

}

1. (8 pts) Create a new *AthleticSocks* instance for an order of 8 gray non-ankle height athletic socks.

AthleticSocks s = new(SockColor.Gray){Ankle = false};

s.Count = 8;

1. (20 pts) Draw a UML diagram of *IItem*, *Socks*, and *AthleticSocks*. (You do not need to include a separate box for the *SockColor* enumeration.)



1. (16 pts) Complete the following unit tests for the *AthleticSocks* class. If necessary, you may use whatever you want for a sock color.

public class AthleticSocksTests

{

[Fact]

public void AthleticSocksImplementsIItemAndExtendsSocks

{

AthleticSocks s = new(SockColor.White);

Assert.IsAssignableFrom<IItem>(s);

Assert.IsAssignableFrom<Socks>(s);

}

[Theory]

[InlineData(SockColor.White, true, 2, 2\*3)]

[InlineData(SockColor.White, false, 4, 4\*4.5)]

[InlineData(SockColor.Gray, true, 2, 2\*4)]

[InlineData(SockColor.Gray, false, 6, 6\*5.5)]

//maybe two more with black

public void PriceIsCorrect(SockColor c, bool ankleHeight, uint count,

decimal expectedPrice)

{

AthleticSocks s = new(c){Ankle = ankleHeight, Count=count};

Assert.Equal(expectedPrice, s.Price);

}

}

**Feel free to remove this portion to make it easier to work.**

**//The following items are needed for #6-10**

public class Socks : IItem

{

public SockColor Color { get; }

public Socks(SockColor c)

{

Color = c;

}

}

public enum SockColor { White, Black, Gray }

public interface IItem

{

decimal Price { get; }

uint Count { get; set; }

}