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

Spring 2025

Exam 1 – 100 points

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

There are 10 questions.

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

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

1. (3 pts) What is a derived property?
   1. A property that has a private backing field
   2. A property whose value is computed from other properties or fields
   3. A property with enforced bounds
   4. A property whose implementation is overridden in a child class
2. (3 pts) When do we use the “virtual” keyword in C#?
   1. When you want to allow a child class to write a new version of the property or method
   2. When you want to require a child class to write a new version of the property or method
   3. When you are not providing implementation details for the property or method
   4. When the property or method was listed in an interface that a class is implementing
3. (3 pts) When a property is declared *static*, this means:
   1. The value of the property cannot be changed
   2. The class containing the property must also be declared *static*
   3. The property cannot use auto property syntax
   4. The property cannot access any non-static fields or properties
4. (3 pts) Suppose that class *Dessert* extends class *Food*, and that class *Cake* extends class *Dessert*. Which of the following statements would compile?

I. Dessert d1 = new Food();

II. **Food f = new Cake();**

III. Cake c = new Dessert();

IV. **Dessert d2 = new Cake();**

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

**public abstract class Account {**

**public abstract decimal Balance {get;}**

**public void Deposit(decimal amount) {...}**

**public void Withdraw(decimal amount) {...}**

**}**

**public class CheckingAccount : Account {**

**public override decimal Balance {get { ... } }**

**}**

Given the following declaration:

**Account a = new CheckingAccount();**

Which of the following will compile without error?

I. **a.Deposit(100);**

II. **a.Withdraw(50);**

III. **Console.WriteLine((a as Account).Balance);**

IV. **Console.WriteLine((Account)a.Balance);**

1. I and II only
2. III and IV only
3. I, II, and III only
4. Each line will compile without error
5. The original declaration will not compile because *Account* is abstract
6. (3 pts) When do we use the “override” keyword in C#?
7. When you want to provide a new implementation for an inherited virtual property in a child class
8. When you want to provide an implementation for an inherited abstract property in a child class
9. When you want to provide an implementation for a property defined in an interface
10. (a) and (b) only are correct
11. (a), (b), and (c) are all correct
12. (15 pts) Consider the *Bread* class in the handout. As if you were inside that class, write a code fragment where you rewrite the *Count* property to enforce that the count must always be between \_*minCount* and \_*maxCount*. (If an attempt is made to set the count to something outside those bounds, leave the count unchanged.) You may assume that the *Bread* constructor is also changed so that the initial count is \_*minCount*. The *Count* property should NOT be virtual.
13. (30 pts) Write the class *Breadsticks*, which extends *Bread*. You may wish to include a constructor. Your *Breadsticks* class should:

* Ensure that its *Name* property has the value “*Breadsticks*”
* Ensure that the smallest count of a breadsticks order is 4 and that the largest count of a breadsticks order is 6
* Include a property that represents whether the *Breadsticks* contain cheese. Make sure this property can only be given a value when a new *Breadsticks* object is initialized, and that by default breadsticks do not contain cheese.
* Ensure that the cost per breadstick is $0.75 for regular breadsticks and $1.15 for breadsticks with cheese.

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

public class Breadsticks : Bread {

public override string Name => “Breadsticks”;

private bool \_cheese = false;

public bool Cheese {

get => \_cheese;

init {

\_cheese = value;

if (Cheese) \_costPerEach = 1.15m;

else \_costPerEach = 0.75m;

}

}

public Breadsticks() : base(4, 6) {

\_costPerEach = 0.75m;

}

}

Breadsticks b = new() {Cheese = true};

1. (20 pts) Draw a UML diagram that includes *Bread*, *BreadBasket*, *ICollection<T>*, and *Breadsticks*. In the case of *BreadBasket*, you ONLY need to include the members listed and not the additional members required by *ICollection<T>*. For *ICollection<T>*, you do not need to list any members. You do not need to list any private or protected members for any class.
2. (17 pts) Complete the following unit test for the *Breadsticks* class.

public class BreadstickTests

{

[Theory]

[InlineData( 4, true, 4\*1.15 )]

[InlineData( 5, true, 5\*1.15 )]

[InlineData( 6, true, 6\*1.15 )]

[InlineData( )]

[InlineData( )]

[InlineData( )]

public void CostIsCorrect(uint count, bool cheese, decimal expected)

{

Breadsticks b = new(){Cheese = cheese};

b.Count = count;

Assert.Equal(expected, b.Cost, 2);

}

}

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

**//The following items are needed for #7-10**

public abstract class Bread

{

private uint \_minCount;

private uint \_maxCount;

public uint Count { get; set; }

public abstract string Name {get;}

protected decimal \_costPerEach;

public decimal Cost => \_costPerEach\*Count;

public Bread(uint min, uint max) {

\_minCount = min;

\_maxCount = max;

}

}

public class BreadBasket: ICollection<Bread>

{

private List<Bread> \_basket = new();

public void AddBread(Bread b)

{

\_basket.Add(b);

}

//other members, as required by ICollection<T>

}