C# Fundamentals: The C# Language

Use C# and an Integrated Development Environment (Visual Studio Community) to complete the following coding exercises.

In cases where you need to display values to the screen, you may use Console.WriteLine() to display the values to console. If you wish to test your code, you can run it inside the static void Main(stringp[args] method of a Main class.

NOTE: The following questions are written presuming they were to be solved in Java, however, they can also be solved in C#. Please refer to instructions.txt if it exists for that particular exercise. Otherwise, you can follow the instructions below and still implement it in C# appropriately (you’ll find the syntax is very similar!). Also, to solve these problems you will have to leverage the training videos and your creativity to research specific implantation considerations.

## 01 Circle Area Exercise

Create a static C# function ‘calculateCircleArea’ that takes the circle radius as a parameter. The function should return the area of the circle as a double. Hint: you can access the constant PI inside the Math class, as Math.PI.

**Example**:

calculateCircleArea(15.5);// returns ‘754.7676350249478’

## 02 Polygon Perimeter Exercise

Create a static C# function ‘calculatePerimeter’ that takes a single parameter with an array of float numbers, containing the length of all the sides of the polygon. The length of this array may be variable, so a polygon with any number of sides could be passed. The function should return as a float the sum of all numbers in the array.

**Example:**

**float**[] polygonSideLengths = {7.5f, 10.4f, 3.7f, 16f, 20f};// A pentagon

*calculatePerimeter*(polygonSideLengths); // Returns ’57.6’

## 03 User Class Exercise

Create a new class, ‘User’, containing:

* Two private class members, ‘name’ (String) and ‘score’ (int).
* Methods that will allow the developer to get and set the value of the name and score members.
* A public, empty, default constructor.
* A method ‘increaseScoreByOne’ that takes no parameters, and returns nothing, but increases the value of the score of this object by one.

**Example:**

User sampleUser = **new** User();  
sampleUser.setName(**"Henry"**);  
sampleUser.setScore(1336);  
sampleUser.increaseScoreByOne();  
  
System.***out***.println(**"User "** + sampleUser.getName() + **" has a score of: "** + sampleUser.getScore()); // Prints ‘User Henry has a score of: 1337’

## 04 Multiple Constructors Exercise: Triangle Class

Create a new class, ‘Triangle’, containing:

* Three private float members, ‘a’, ‘b’ and ‘c’ representing the three sides of a triangle.
* A constructor with three parameters, ‘a’, ‘b’, and ‘c’, that would create a Triangle whose sides would have the specified measure.
* A constructor with only one parameter, ‘sideLength’, representing the length of a single side, and assuming all sides are equal (that would hence create an equilateral triangle.)
* A public function ‘calculatePerimeter’ that takes no parameters and returns the perimeter of this triangle.

**Example:**

Triangle scaleneTriangle = **new** Triangle(10, 9, 7.5f);  
**float** scaleneTrianglePerimeter = scaleneTriangle.calculatePerimeter();  
System.***out***.println(**"Scalene triangle perimeter = "** + scaleneTrianglePerimeter);

// Prints ‘Scalene triangle perimeter = 26.5’  
  
Triangle equilateralTriangle = **new** Triangle(10);  
**float** equilateralTrianglePerimeter = equilateralTriangle.calculatePerimeter();  
System.***out***.println(**"Equilateral triangle perimeter = "** + equilateralTrianglePerimeter);

// Prints ‘Equilateral triangle perimeter = 30.0’

## 05 Method Overload Exercise: Shopping Cart Class

Given the following class representing an Item:

**public class** Item {  
  
 **private float** price;  
  
 **public float** getPrice() {  
 **return** price;  
 }  
  
 **public** Item(**float** price) {  
 **this**.price = price;  
 }  
}

And the following incomplete template of a new class, ‘ShoppingCart’:

**public class** ShoppingCart {  
  
 **private float** totalPrice;  
 **private int** numberOfItems;  
  
 **public float** getTotalPrice() {  
 **return** totalPrice;  
 }  
  
 **public int** getNumberOfItems() {  
 **return** numberOfItems;  
 }  
  
 */\*\* Add your code here \*\*/*}

Add the following public methods, all named ‘addItems’, to the ShoppingCart class (so they overload each other:)

* A method ‘addItems’ that takes two parameters, a float ‘price’ and an int ‘numberOfItems’, and adds them to the ‘totalPrice’ and ‘numberOfItems’ of the class.
* A method ‘addItems’ that takes one parameter, a float ‘price’, and adds it to the totalPrice, increasing the numberOfItems by one.
* A method ‘addItems’ that takes one parameter, an Item object ‘item’, and adds its price to the totalPrice, increasing the numberOfItems by one.
* A method ‘addItems’ that takes a variable number of parameters of type Item, addItems(Item... list) and adds their prices to the totalPrice, plus their number to numberOfItems.

Then, using your completed class, invoke its methods completing the following main method in a Main class:

**public class** Main {  
  
 **public static void** main(String[] args) {  
 ShoppingCart shoppingCart = **new** ShoppingCart();  
  
 Item item1 = **new** Item(10.5f);  
 Item item2 = **new** Item(20.75f);  
 Item item3 = **new** Item(9.99f);  
 Item item4 = **new** Item(14f);  
 Item item5 = **new** Item(100f);  
 Item item6 = **new** Item(55.5f);  
  
 *// Add item1 individually  
 // Add the remaining items (item2, item3, item4, item5 and item6) in a single operation  
 // Add another item of price 33.5  
 // Add two items in a single operation, with a combined price of 50* System.out.println(shoppingCart.getNumberOfItems() + **" items were added,"** + **" with a total price of $"** + shoppingCart.getTotalPrice());  
 }  
}

## 06 Class Inheritance Exercise: Employee Class

Create an Employee class with:

* A private Integer (non-primitive) field named ‘idNumber’.
* A public method with no parameters called ‘getIdNumber’ that returns the ‘idNumber’ of the class.
* A constructor that takes one Integer parameter and saves its value in the ‘idNumber’ field.
* A public method with no parameters called ‘hasAdministratorRights’ that always returns ‘false’ for an Employee.
* A method overriding the ‘equals’ method of the base Object class, so two Employee objects can be compared in a specific way. We consider two Employee objects to be equal if the two of them have the same ‘idNumber’ **value**.

Then create a Manager class that:

* Inherits from the ‘Employee’ class (a Manager is a special case of an Employee.)
* Overrides the ‘hasAdministratorRights’ method to always return true.

**Example:**

Employee worker = **new** Employee(1);  
Employee theBoss = **new** Manager(2);

System.***out***.println(**"Are these employees the same? "** + (worker.equals(theBoss)));

// Prints ‘Are these employees the same? false’

Employee anEqualBoss = new Manager(2);

System.***out***.println(**"Are these employees the same? "** + (theBoss.equals(anEqualBoss)));

// Prints ‘Are these employees the same? true’

System.***out***.println(**"Has The Boss admin rights? "** + (theBoss.hasAdministratorRights()));

// Prints ‘Has The Boss admin rights? True’

## 07 Exceptions: Glass Class

Create a custom Exception named ‘GlassOverflownException’ whose constructor takes one String parameter, ‘message’, containing a description about the cause of such exception. We will use these exceptions to represent the error situation of trying to fill up a glass with more water than it can hold.

Create a class ‘Glass’ with:

* Two private float fields, ‘maxGallons’ and ‘currentGallons’, representing the maximum liquid capacity of the glass, and the current volume of liquid contained by the glass.
* A constructor that takes a single parameter ‘maxGallons’ and assigns it to the ‘maxGallons’ field.
* A public method ‘addWater’ that takes a single parameter ‘gallonsToAdd’, adds that value to the current gallons, and throws a ‘GlassOverflownException’ if the added gallons, plus the current gallons, surpass the maximum gallons that the glass can hold. Ideally, this exception would hold a descriptive message with the details of the error.

**Example:**

Glass hugeGlass = **new** Glass(1.5f);  
  
**try**{  
 hugeGlass.addWater(20f);  
} **catch**(GlassOverflownException e) {  
 System.***out***.println(e);

// Prints com.daugherty.GlassOverflownException: glass capacity exceeded; you have spilt 18.5 gallons

}

## 08 Packages and Interfaces: Building a Calculator Watch

Create the geekiest calculator watch ever, following these steps:

1. Structure your code in packages. Create com.daugherty.time package to store our time-related interfaces, com.daugherty.math package to store math-oriented interfaces, and com.daugherty.gadgets package to store the calculator watch classes.
2. Create the interface ‘IWatch’ that exposes a single method named ‘getUNIXEpochTime’, that receives no parameters, and returns a long with the [Unix Epoch Time](https://en.wikipedia.org/wiki/Unix_time).
3. Create the interface ‘ICalculator’ that exposes a single method named ‘calculateAdd’, which receives two int parameters, ‘numberOne’ and ‘numberTwo’, and returns an int with its sum.
4. Create the ‘CalculatorWatch’ class, that implements both interfaces.
   1. Hint: System.currentTimeMillis()returns the current Unix Epoch time.

**Example:**

CalculatorWatch calculatorWatch = **new** CalculatorWatch();  
  
**int** addedNumbers = calculatorWatch.calculateAdd(2, 2);  
**long** currentUNIXTime = calculatorWatch.getUNIXEpochTime();  
  
System.***out***.println(**"The result of the addition is "** + addedNumbers + **" and the current UNIX time is "** + currentUNIXTime);

// Prints: ‘The result of the addition is 4 and the current UNIX time is 1467411884801’