­­­Homework 5: Character Behaviors

**Due Tuesday, October 31th by 11:59pm.**

In software development, you often run into the case where it would be nice to create variants of a class. There may be multiple characters in an RPG, but many of the features of those characters are the same. For example, all characters might have a health attribute you need to track.

You’re going to create a class that can hold the common information about characters in your game, and then make a few more specific classes that inherit from the first one. Lastly, you’ll create two characters and make them fight.

This homework will be used as the basis for the next homework, so be sure to finish it!

# Task Overview

This is a brief overview of the tasks you must complete for this assignment. Specifics are given in the corresponding sections later in this document. ***Read the entire document before starting.***

* Create a **CommonCharacter** class, which holds common information about characters
* Create **two** **more specific classes** that inherit from CommonCharacter
* Add **specified methods** to these child classes
* **Test your methods** in Main by making two characters and having them fight

This homework is broken into three activities. Read on for each activity.

# Activity 1: Using Inheritance

Start by brainstorming two different types of combat-related characters (like rogues and fighters, or archers and wizards). Next, determine common combat characteristics that both characters share (like strength, dexterity or level). They are also required to have a health field, as you will make these characters battle later in this assignment.

## Coding the Common Character

Now for the coding: Create a class called CommonCharacter. The common attributes you thought of earlier will become the fields of this class. Be sure to include a “health” field and a Random object field. The class will also need appropriate constructors (a default and parameterized), and properties to provide public access to your fields. The properties should mostly only have “get” sections, unless the corresponding field represents something that can change over time.

Remember, if we want child classes to eventually have access to these fields, they must be defined as **protected** instead of private.

## Coding the More Specific Classes

Next, create two more classes, each of which inherits from the CommonCharacter class you created above. These child classes will contain fields and properties that are unique to these more specific characters. Since these are child classes, they’ll automatically have access to any protected fields from the parent class. However, each of these classes must have at least *two additional fields* (on top of what they get from CommonCharacter). These classes should also have two constructors (a default and a parameterized), which will initialize all fields with useful starting values.

Your project should only instantiate a single Random object. To share that Random object among multiple instances of your child classes, create it inside of the Main method and pass it into each child class’s constructor as a parameter. Remember to save the incoming Random object in the class’s corresponding field.

**Example:** All the character types share the idea of a health value, defining how healthy or injured they are at the moment. All the character types also share a Random object. These would be defined as fields in the parent class. The same might be true for attributes like strength and dexterity. However, your specialized Rogue class might have a stabbingSkill field in addition to any fields it inherits from CommonCharacter.

## Testing

After you have created these classes, add some code to the Main method to test their functionality. Create several objects of your child classes (not CommonCharacter) using the different constructors. Print out some of the characters’ data (using properties) to make sure the characters are set up correctly.

# Activity 2: Adding Behaviors

Up to now, the characters may be fairly uninteresting. You have some fields and ways to access and potentially change those variables. However, the characters don’t really do anything.

In order to improve on them, you need to implement the following methods in **each child class** you created in Activity 1. These must be implemented exactly as defined below, with the specified method signatures and return types. However, the *contents* of each method will be slightly different in each child class.

## Comments

Some of the following methods may have calculations that depend on a child class’s stats. You are required to add comments explaining why (or how) your calculations work in their intended manner. For example, write comments that explain why your code causes a rogue to flee combat if it’s below 20 health while your fighter remains on the battlefield until death.

## public override String ToString()

Returns a String containing all of the character’s fields (with appropriate labels). We haven’t discussed ToString() or override quite yet (that’s coming up next week). If Visual Studio tries to automatically fill in the method for you, remove the “base.ToString()” part and instead return a string that you make yourself.

A Fighter class might return a string similar to the following:

“This Fighter has a dexterity of 1, a strength of 6 and currently has 14 health left. They also get 3 bonus damage when attacking, and their armor reduces incoming damage by 1.”

A Rogue class might return a string similar to the following:

“This Rogue has a dexterity of 4, a strength of 2 and currently has 12 health left. Their rank 10 invisibility cloak means they have a 10% chance to take no damage when attacked, and their stabbing skill of 2 means they deal 2 times the amount of damage when attacking.”

## public int Attack()

This method should calculate the amount of damage this character deals during a single attack. The final returned value should be based on several of the overall stats (fields) of each class. To make this more interesting, the calculation will contain one or more random numbers. The exact calculations you use are up to you, but they must incorporate several fields. Below you’ll find some examples based on the example stats given on the previous pages.

For example, a Fighter’s attack calculation might be:

attack = (random number between 0 and *strength*) \* (*dexterity*) + (*bonus damage*)

However, a Rogue’s attack calculation might be:

attack = (random number between 0 and *strength*) \* (*dexterity*) \* (*stabbing skill*)

## public void TakeDamage(int damage)

This method will reduce the character’s health by the specified amount of damage. If the character has a stat that can potentially adjust the amount of incoming damage (or ignore it entirely), apply that here before reducing their health.

After adjusting the incoming damage, make sure that the damage is a positive number (otherwise this method will end up healing the character). Again, the exact calculations for how damage is mitigated depends on the characters you’ve created and their specific stats. Below are some examples.

Here is an example of a Fighter’s damage calculation:

damage = damage – (*armor rating*)

However, a Rogue might have a small chance of completely ignoring damage. This kind of effect would require a random number to be generated. If that random number is within a particular range (based on a corresponding stat like *invisibilityCloak*), then the incoming damage would be set to zero before the calculation.

## public bool HasFled()

This method should return true when a character would flee combat, based on some condition. The condition should be different for each character, and may or may not depend on their stats.

For example, a Rogue might flee combat if they are currently below 20 health.

However, a Fighter might want to fight to the death, and thus would never, ever flee combat.

## public bool IsDead()

This method examines a characters health value to determine if they’re dead. If they are at or below zero, it should return true. Otherwise, the character is still alive and the method returns false. This method will generally be the same for all characters.

## Testing

Now that you’ve completed more specific child classes, be sure to test them out. Go back to the Main method and remove the testing code from earlier. This time, create one of each child character you’ve defined (for instance, one Fighter object and one Rogue object).

Print their stats using their ToString() methods. Call each of their other methods several times to ensure they’re working as expected, printing the results of the methods (or the newly changed stats like Health) as you go.

Do not move on to the last activity until you’re confident **all of your methods** are working appropriately.

# Activity 3: To Battle

The last part of this homework is to make your two characters fight until one has fled or has otherwise been defeated. Start your characters with ample health so that the fight isn’t over after one round. Start off by printing a welcome message and each character’s full stats.

## Rounds

A single round of the fight consists of each character performing an attack, and applying the damage of that attack to the opposing character. For instance, the results of the Fighter’s Attack() method must be passed into the Rogue’s TakeDamage() method, and vice versa.

As you generate these attack numbers, print them out so the user can see how much damage the characters are doing to each other. At the end of the round, print the current health values.

## To the Death (or the Flee)

Since you don’t know how many “rounds” of fighting it will take for the battle to be over, you’ll need to loop until one of the characters has fled or is dead.

If you want to “pause” your program for a few milliseconds so the user has time to read what’s happening as the battle progresses, use the following line of code inside the loop:

System.Threading.Thread.Sleep(500); // 500 milliseconds is half a second

## Winner

Once the fight is over, announce the winner and their method of victory. The Fighter might win because the Rogue fled, or because the Rogue has died. Note: depending on how you’ve implemented your battle loop, there may be a chance that both characters deal lethal damage to each other in the last round. Your code should account for that circumstance, too.

# Sample Run

Welcome to the Battle! Today the Fighter is taking on the Rogue.

This Fighter has a dexterity of 1, a strength of 6 and currently has 50 health left. They also get 3 bonus damage when attacking, and their armor reduces incoming damage by 1.

This Rogue has a dexterity of 4, a strength of 2 and currently has 30 health left. Their rank 10 invisibility cloak means they have a 10% chance to take no damage when attacked, and their stabbing skill of 2 means they deal 2 times the amount of damage when attacking.

Round 1 ---------------

The Fighter deals 7 damage to the Rogue.

The Rogue deals 8 damage to the Fighter.

The Fighter has 43 health left.

The Rogue has 23 health left.

Round 2 ---------------

The Fighter deals 3 damage to the Rogue.

The Rogue deals 16 damage to the Fighter.

The Fighter has 28 health left.

The Rogue has 20 health left.

Round 3 ---------------

The Fighter deals 5 damage to the Rogue.

The Rogue deals 0 damage to the Fighter.

The Fighter has 28 health left.

The Rogue has 15 health left.

\*\* Battle Finished \*\*

The Rogue has fled

The Fighter is the winner!

The rogue flees when his health drops  
below 20.

Although the fighter has received 8 damage   
from the Rogue, his armor reduced the  
amount of damage taken by 1,  
therefore receiving 7 damage.