# COSC 1047 – Winter 2019 – Assignment 2
# Due: (see D2L)

This assignment is designed to give you some experience writing Java object classes using inheritance and aggregation. This assignment is done using Eclipse. Import the .zip file that is provided on D2L and when it is complete, export it again as a .zip. All submissions should be done through D2L. You can submit and resubmit as many times as you want up to the due date and time. If you have any problems submitting, email me – alangille@cs.laurentian.ca - in a timely manner. **Make sure your name, student number and a brief description of your program appear at the top of each file in comments. Also make sure that you use comments to clarify your code for marking.** ***Failure to do either of these tasks may result in a deduction of marks.***

***Submit solutions to questions 1 and 2 only. But DO ALL questions to be prepared for the tests and exam. Solutions will be provided (via D2L) only for required questions.***

***If you work in partners remember to include both names in the source code. The person who is left out will receive zero.***

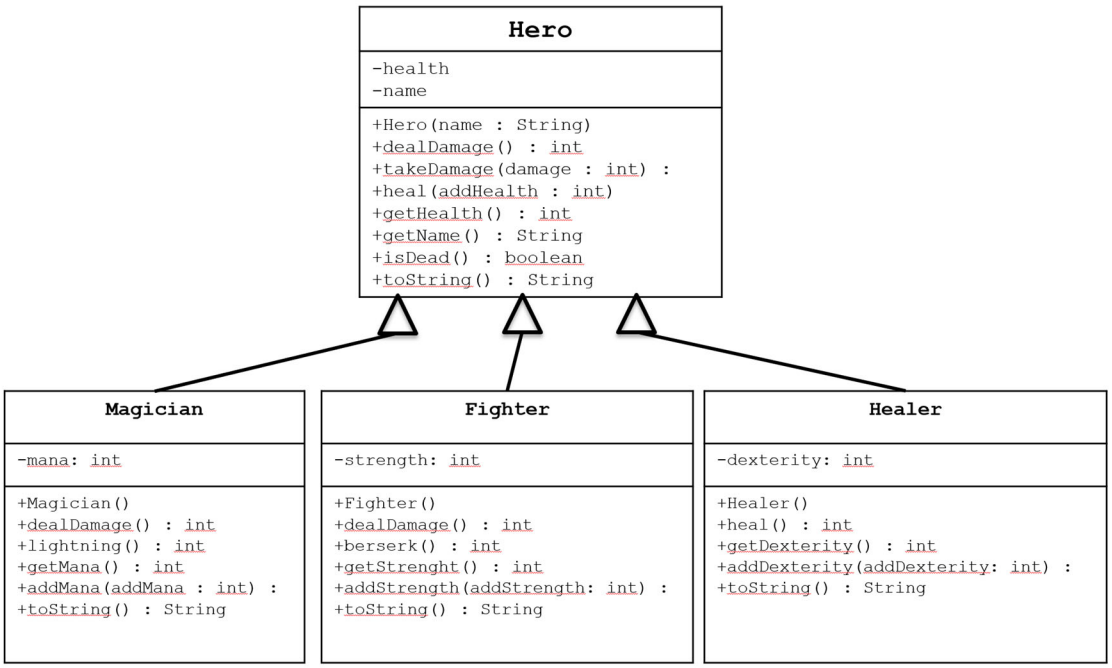
Note: It is imperative that you import the cosc1047-w19-a2.zip file from D2L properly or you won't be able to work in the proper project environment.

---

Grading:

See D2L for rubric.

Question 1. (Magician.java, Healer.java, Fighter.java, HeroTester.java) Consider the abstract hero class and the three subclasses shown below.




Review the hero class to see how it works (it's in the project/package for you already). **Your job is to implement the following subclasses**. Each of the subclasses has an extra data field that relates to their specific hero talent. Some help with these special talents and other methods are below.

**Fighter**:
-The fighter has an extra data field called Strength and a method called Berserk. When the fighter calls Berserk they do damage equal to 1/3 of their existing health BUT they lose 1/4 of their existing health (round to integer values). Using Berserk costs one Strength point. When a fighter is constructed they have 3 strength points.

-the dealDamage() method for the fighter does between 7 and 10 points of damage

**Magician**:
-The magician has an extra data field called Mana and a method called Lightning. When the magician calls Lightning the damage they inflict is quadruple what it would have been otherwise. Call the existing dealDamage function and quadruple the value before dealing it to the enemy. Using Lightning costs 1 Mana point and a magician has 4 Mana points to start with.

-the dealDamage() method for the magician does between 4 and 6 points of damage

**Healer**:
-The healer has an extra data field called Dexterity and a method called Heal. When the healer calls Heal each LIVING member of the party receives between 5 and 10 health points (not to pass 100) and each receives one point back to their special skill (dexterity, mana or strength). Calling Heal costs 2 Dexterity points and a healer starts with 4 Dexterity. Hint: The heal method determines how many points are restored to each party member's health. Let your main method do the actual work of healing the surviving heroes.

-the dealDamage() method for the Healer does between 3 and 5 points of damage

If your hero is out of special skill points they can only call the base dealDamage() method (i.e., they cannot use their special skill). Once a hero is dead, they can't do anything … because, they're dead.

In addition to any extra accessors and mutators you should override the toString method for easy printing of the hero state.

---

Write a tester class that does the following:

**Option** 1: Write a hard-coded tester that does the following:

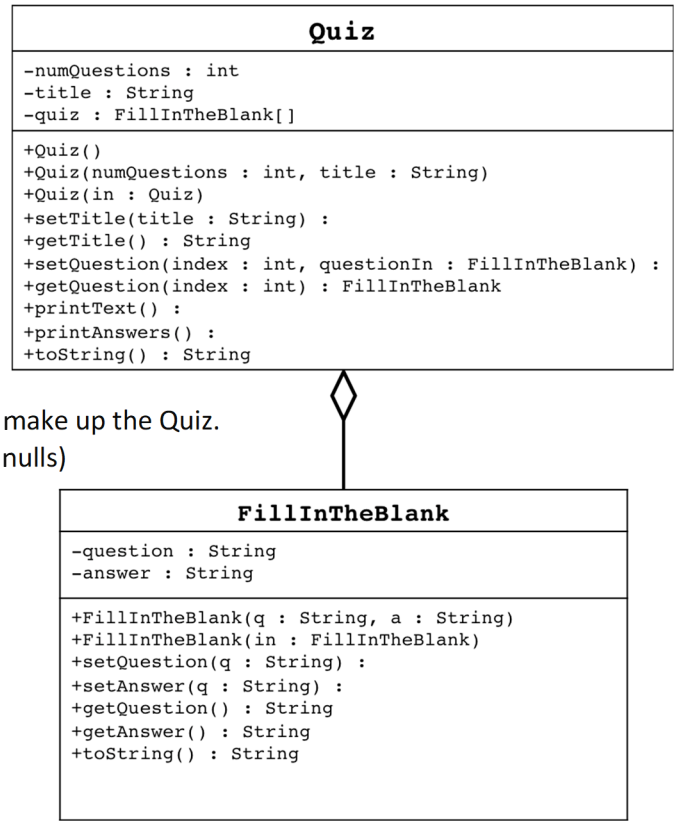
- Create an array of three heroes (one of each type)
- Create an array of 10 goblins (already in the project/package)
- Wage an epic battle between the groups
    - Heroes attack first in sequence (i.e., hero[0] attacks first, then hero[1] and so on)
        - They attack the goblins in sequence (each hero attacks the first goblin in the array until it is vanquished and then they move onto the next)
    - Goblins attack next in order (goblin[0], then goblin[1] and so on) but they attack a random hero.
        - If a hero dies it cannot attack nor be attacked any further, skip it in the rotation. Same for ex-goblins.
- At any point, if all the heroes have met their doom, the goblins win. If any of the heroes live and the goblins are all dispatched, then yay!.
- Your tester should keep you updated on how the battle is progressing by telling you how many goblins are left, what is the state of your heroes after the round, who has died (if anyone) and when there is a win condition.
- Heroes have a 40% chance of using their special skill. The other 60% is regular damage. If a hero tries to use their special skill but they have no special skill points left, they simply do regular damage.
- If you're stuck, try a battle between 1 hero and 1 goblin to get the mechanics figured out, then move onto the arrays.

**Option** 2 (for bragging rights):

Same as option 1 BUT allow the user to select which hero they want to send into battle, which goblin they want to attack and whether or not to do regular damage or use their special attack. The rules of the game must be maintained. They cannot attack with dead heroes and cannot attack dead goblins.

***PLEASE post the output of some of your battles to the forum so that people can see how your tester is working.*****

Question 2: (FillInTheBlank.java, Quiz.java, QuizTester.java) Consider the following aggregation relationship:




You are given the FillInTheBlank class which defines a fill-in-the-blank-type quiz question. Spend a few moments going over this class to understand how it works. A short tester class has also been provided to show you how it works. Add a copy constructor to this class that makes it easy to duplicate an existing question. Write JavaDocs for the FillInTheBlank class.

Next, write a Quiz class that is made up of FillInTheBlank questions. The UML of this class is given.

A few hints:

- numQuestions simply keeps track of how many total questions make up the Quiz.
- The default constructor creates a quiz with 5 empty questions (nulls) and a title of your choosing.
- printTest() should display the title of the test and then all the test questions (with question numbers) and printAnswers() should display only the answers (with question numbers). IF one of the questions has not yet been answered (i.e., it's still null in the array) do not print anything. You can use an if statement to test for nulls. Try it out.
- setQuestion() and getQuestion() should make sure that the index numbers are appropriate and if not should do nothing.
- toString() simply prints the quiz title and the number of questions the quiz can hold.
- Be sure to use DEEP COPY and DEEP RETURN where appropriate.
- If anything else is vague or uncertain use the discussion forum! My Quiz class is 75 lines without comments and with liberal spacing and curly braces.
- Scanners should only appear in your tester class.
- Write JavaDocs for the Quiz class.

Finally write a quiz tester class that creates a simple 5 question quiz. Your tester should allow the user to create the questions. When the quiz has been created use the printTest() and printAnswers() methods to display the results.

No output is shown here. Be creative, be polite (to your users) and if you want to compare and contrast program output use the discussion forum.

Badge Bonus: After you've done the work above in the tester, create a second quiz using your quiz copy constructor. For your second (copied) test, swap the first and last questions and for the middle question change the question AND the answer (do this by access the question's set methods, not by replacing the question). Finally, print the two tests and two answer sets to show that they are distinct and that the changes to one do not affect the other (i.e., your deep copy is working).

Question 3

Design a `ship` class that has the following data fields:

- A data field for the name of the ship (a `string`).
- A data field for the year that the ship was built (an `int`).
- A constructor and appropriate accessors and mutators.
- A `toString` method that displays the ship's name and the year it was built.

Design a `CruiseShip` sub class that extends the Ship class. The `CruiseShip` class should have the following:

- An extra data field for the maximum number of passengers (an `int`).
- A constructor and appropriate accessors and mutators.
- A `toString` method that overrides the `toString` method in the base class. The `CruiseShip` class's `toString` method should also include the max passenger limit.

Design a `CargoShip` class that extends the `Ship` class. The `CargoShip` class should have the following:

- An extra data field for the cargo capacity in tonnage (an `int`).
- A constructor and appropriate accessors and mutators.
- A `toString` method that overrides the `toString` method in the base class. The `CargoShip` class's `toString` method should also include the cargo tonnage.

In the appropriate class write an `equals` method to test if two ships are equal - they should be considered equal if they have the same name and were built in the same year.

Demonstrate the classes in a program (`ShipTester`) that has an array of `Ship` objects (at least 5 of them). Assign various `Ship`, `CruiseShip`, and `CargoShip` objects to the array elements (you can hard-code the objects and data) and print out the initial ship configurations. Show that you can use both the accessor and mutator methods on a sampling of the ships (again, you can hard-code the method arguments here if you like). Also show that your `equals` method can determine whether or not two ships are equal (this means creating at least two ships that have "equal" data).

```
Displaying Ships:

Ship[ Name: Queen Annes Revenge Year built: 1701]
Cruise Ship[ Ship[ Name: USS Enterprise Year built: 2245], Passengers: 2400 ]
Cargo Ship[ Ship[ Name: Black Pearl Year built: 1699], Tonnage: 50000 ]
Cruise Ship[ Ship[ Name: USS Voyager Year built: 2371], Passengers: 2800 ]
Cargo Ship[ Ship[ Name: The Victory Year built: 1790], Tonnage: 33100 ]
--------------------------

Checking ship 2 (accessing):

Name: Black Pearl
Year Built: 1699
Cargo Tonnage: 50000
--------------------------

Changing ship 3 (mutating):

Before: Cruise Ship[ Ship[ Name: USS Voyager Year built: 2371], Passengers: 2800 ]
After: Cruise Ship[ Ship[ Name: USS Enterprise Year built: 2245], Passengers: 2800 ]
--------------------------

Comparing ship 0 to ship 1 - Equal?: false
Comparing ship 1 to ship 3 - Equal?: true
Comparing ship 4 to ship 2 - Equal?: false
```

Question 4.

(CardGame.java, ComicBook.java, InventoryTest.java) Consider the following interface:

```
interface InventoryItem {
// Return the quantity of that item currently in stock.
public int getQuantity();
// Get the price of an individual of an item.
public double getRetailPrice();
// Change the quantity of an item absolutely.
public void setQuantity(int qty);
// Adjust the retail price of an item
public void setRetailPrice(double price);
}
```

The idea here is to pretend that you are creating a very simple inventory system for a friend who is opening a new store. Write at least two object classes representing card games and comic books that implement this interface. Each item type should have data fields that define the quantity of the item in stock and the per-unit price of that item. A card game should also have a simple description data field while a comic book should have a title and an issue. Write appropriate constructors, toStrings, and any missing accessors or mutators for your classes.

Write an application class called InventoryTracker that demonstrates your inventory objects. Create (hard-code) several different items of each type and store them in a ***single*** array. Show that you can print their state using toString. Also show that you can use a polymorphic loop to determine the total inventory quantity (total number of items), total inventory value as well as the most and least valuable inventory items in the store (value is the quantity × the retail price).

```
Creating a store inventory array of 5 items...
Stocking the inventory items...Printing store inventory:

Comic Book[ Superman, Issue 2, Retail Price: 100.0, Qty in Stock: 2]
Card Game[ Uno, Retail Price: 9.95, Qty in Stock: 3]
Card Game[ Munchkin, Retail Price: 29.95, Qty in Stock: 10]
Comic Book[ Buffy the Vampire Slayer, Issue 6, Retail Price: 9.95, Qty in Stock: 2]
Comic Book[ X-men, Issue 50, Retail Price: 8.95, Qty in Stock: 12]

Assessing value of in-stock items...

Total Inventory Quantity (num items):29
Total Inventory Value:656.65
Inventory Item with Max Value of 299.5 is: Card Game[ Munchkin, Retail Price: 29.95, Qty in Stock: 10]
Inventory Item with Min Value of 19.9 is: Comic Book[ Buffy the Vampire Slayer, Issue 6, Retail Price:
9.95, Qty in Stock: 2]
```

(5 marks ) After you have accomplished the work above, change your tester so that it allows the user to enter the inventory items instead of hard-coding them. Your tester:

- Prompts the user for the size of the array
- Uses a loop to prompt the user for each item in the array
- Prompt the user for the type of item and then the necessary data to construct the item
- Then run the remainder of the program as shown above.

Question 5: (Unicycle.java, UnicycleTester.java) In the project, you are given object classes for Frames and Wheels. Your job is to create a new Object class Unicycle that has as data field a Frame object and a Wheel object. It also has a String data field which represents the customer for whom the Unicycle is being built. I suggest taking a few minutes to look over Frame and Wheel to see what they do. Your Unicycle class should have the following:

- A default, no argument constructor that creates a unicycle with your name as the customer and a frame/wheel combination of your choice.
- A fully-argumented constructor that takes in a wheel, frame and customer name.
- A copy constructor that creates a new Unicycle that has the same properties as the incoming argument Unicycle.
- Get and set methods for all of the data fields.
- An appropriate toString method.

The idea here is to practice deep-copy for your constructors, set methods and return methods. Wherever appropriate you must avoid using shallow copy. Write a hard-coded tester to make sure that your objects are working properly and that there are no "data leaks" (this is what happens when one object changes and those changes are reflected in another).

Bonus: Create an interactive tester class that collects from the user all of the information required to create their own custom Unicycle and allows them to modify if they choose to.

My hard-coded tester output:

```
Creating Unicycle using default constructor:
Unicycle[Aaron Langille, Frame[16, blue], Wheel[12, learner]]

Creating new frame and wheel for another unicycle:
Frame[18, green]
Wheel[16, mountain]

Creating new custom unicycle from the parts for Peter Parker (aka Spiderman):
Unicycle[Peter Parker, Frame[18, green], Wheel[16, mountain]]

Pete changed his mind and wants to alter the unicycle to be a red and blue learner:
Unicycle[Peter Parker, Frame[18, red and blue], Wheel[16, learner]]
```