INSTRUCTIONS - CHALLENGE 5

Okay. If, for some reason, you opened this before reading the comment at the top of the Main Class,
go back and read that first. If you're here because that comment told you to come here, then keep reading!

This challenge is similar to the last one. But we're going to start with one of the major differences: Interfaces.

1. To start, I want you to right-click on the package "hearthstoneCardStyles" and select new > interface.
    a. Name your new interface "Card" and hit finish.
        i. Before we continue, here's some basic information about interfaces: Interfaces are like outlines for how Classes are supposed to be constructed. They work like Abstract classes with 2 main differences.
            1. Firstly, they AREN'T considered the parent of the class that implements them, so "super" does NOT lead to them. This means that a Class can implement MULTIPLE interfaces because there’s no code that can break from using multiple of them.
            2. Secondly, interfaces cannot include any implemented code. This means that they can only DEFINE methods and cannot create implementations for those methods. You may be wondering why this is useful. It’s not ALWAYS useful, but if you want to force the user to conform to a specific standard or style without writing the code for them, interfaces are key. When used in tandem with Abstract classes, they create powerful frame-working tools.

    b. Okay, now that you've got the interface open, I want you to write some weird code... write the code that I’ve given you below after the line "public interface Card {".

```
public enum Rarity {
        NONE, COMMON, UNCOMMON, RARE, LEGENDARY
}

public enum GameClass {
        DRUID, HUNTER, MAGE, PALADIN, PREIST, ROGUE, SHAMAN, WARLOCK, WARRIOR, NEUTRAL
}

public enum Set {
        BASIC(true), CLASSIC(true), WHISPERS_OF_THE_OLD_GODS(true), ONE_NIGHT_IN_KARAZHAN(true), MEAN_STREETS_OF_GADGETZAN(true),
JOURNEY_TO_UNGORO(true), KNIGHTS_OF_THE_FROZEN_THRONE(true), KOBOLDS_AND_CATACOMBS(true), NAXXRAMAS(false), GOBLINS_VS_GNOMES(false),
BLACKROCK_MOUNTAIN(false), THE_GRAND_TOURNAMENT(false), LEAGUE_OF_EXPLORERS(false), HALL_OF_FAME(false);

        private final boolean standard;


        Set(boolean isStandard) {
                this.standard = isStandard;
        }


        public boolean isStandard() {
                return standard;
        }


        public boolean isWild() {
                return !standard;
        }
}

public enum Type {
        MINION, SPELL, WEAPON, HERO
}
```

c. The code that I just had you write contains a bunch of variable definitions called enums. Enums are special types of variables that can be called by their names and used AS their names. For example, the first set of enums I had you set up are the possible Rarity values of any given card. To access the rarity value, All you need to do is call "Rarity." (the “.” Is the important part here. They work like classes with static variables) and then the value of the enum you want, so if I wanted the Rarity value “uncommon”, I'd just call "Rarity.UNCOMMON;" They're their own data type, so they can be compared without fear of strange input. I.e. doing something like this is perfectly fine:

```
if(passedInCard.rarity == Rarity.COMMON) {
    //Do something here
}
```

Enums are just a cool way to deal with a small list of possible values for something. You probably noticed that the final enum "Set" has some code within the codeblock that defines it... That's because Enums are objects. Don't worry too much about what it's doing. Just know that it allows you to determine if a Set is in the standard pool or the wild pool by calling Set.WHATEVER_SET_YOU_WANT.isStandard(); or Set.WHATEVER_SET_YOU_WANT.isWild();

d. Now that the confusing part is almost over, we'll move on to the method definitions. Every Hearthstone card shares some basic information (regardless of the card type), so, let’s create some methods that reflect that. The methods we'll need to create are:

```
isGolden()              -> Returns a boolean
getName()               -> Returns a String
getDescription()        -> Returns a String
getManaCost()           -> Returns an int
getCurrentManaCost()    -> Returns an int
getType()               -> Returns an enum called Type (they work like objects, so just
make sure the return value is "Type")
getRarity()             -> Returns an enum called Rarity
getGameClass()          -> Returns an enum called GameClass
getSet()                -> Returns an enum called Set
isStandard()            -> Returns a boolean
isWild()                -> Returns a boolean
```

e. Creating these methods is only slightly different than using an abstract class to create them, in fact, the only difference is that you don't need the abstract keyword. They're built like this:

```
[scope] [optional modifier] [return type] [method name] ([Method
parameters]);   //<-- Remember to always end in a semicolon when inside an
interface.
```

So, for example, the first method, isGolden() would look like this:

```
public boolean isGolden();
```

f. After outlining all the methods that I gave you, your first interface is done. Something cool you should know about interfaces is that they can extend other interfaces, just like classes can extend other classes. We'll be doing just that in a bit, but first. I want you to create an abstract class called "PlaceholderCard". Make sure you check the checkbox to make it abstract and click the "add" button to add an interface that it inherits from the interface

you should choose is the "Card" interface you just created. Use the search function if you don't see it in the list of interfaces. You should notice, after clicking finish, that it auto-filled in some information. That's good as it's less code for you to write. Now, we need to adjust the constructor so that it takes in more of the information that we need.

The constructor should take in the following information in order:

```
[Variable Type] - [Variable Name]

String - name
String - description
int - manaCost
Type - type
Rarity - rarity
GameClass - gameClass
Set - set
boolean golden
```

g. You'll notice that we can use the Enum name as a variable type, that's another huge advantage of them. Okay, now, inside the constructor, I want you to take all of those passed-in variables and save them inside some instance variables. BUT I want the instance variable's scope to be "protected" NOT "public". I would also like you to create an extra instance variable called currentManaCost and ALSO set its value to the manaCost that will be passed in (do that along with saving manaCost in an instance variable called manaCost, like normal).

h. Now that you've stored all of the passed-in variables into instance variables, I want you to go through all of the @override methods and return their correct values. (i.e. "public boolean isGolden()" should return the instance variable golden. The last two are a bit tricky, isStandard and isWild aren't simply returning a variable. They need to return the method stored within our "Set" enum. So for isStandard() you should return set.isStandard(); and for is wild, you should return set.isWild();

i. One last thing that I'd like you to do, I want you to paste the following code at the bottom of the class (above the last curly brace):

```
		@Override
	public String toString() {
		String s = "_____\n";
		s += "|" + name + "\n|\n|" + description + "\n|\n|Cost " + manaCost + "\n|";
		if (this.type == Type.WEAPON) {
			Weapon weapon = (Weapon) this;
			s += "Attack: " + weapon.attack + "\n|Durability: " + weapon.durability + "\n|\n|Abilities:";

			if (weapon.abilities != null) {
				for (WeaponCard.Ability ability : weapon.abilities) {
					s += "\n|" + ability.name();
				}
			} else {
				s += "NONE";
			}

			s += "\n|\n|";

		} else if (this.type == Type.MINION) {
			Minion minion = (Minion) this;

			s += "Attack: " + minion.attack + "\n|Health: " + minion.health + "\n|\n|Abilities:";

			if (minion.abilities != null) {
				for (MinionCard.Ability ability : minion.abilities) {
					s += "\n|" + ability.name();
				}
			} else {
				s += "NONE";
			}

			s += "\n|\n|Creature Type: " + minion.creatureType.name() + "\n|\n|";

		} else if (this.type == Type.SPELL) {
			Spell spell = (Spell) this;

			s += "Spell Type: " + spell.spellType.name() + "\n|\n|";

		} else {
			Hero hero = (Hero) this;

			s += "\n|\n|Abilities:";

			if (hero.abilities != null) {
				for (HeroCard.Ability ability : hero.abilities) {
					s += "\n|" + ability.name();
				}
			} else {
				s += "NONE";
			}

			s += "\n|\n|";

		}

		s += "Type: " + type.name() + "\n|Rarity: " + rarity.name() + "\n|Class: " + gameClass.name() +
"\n|Set: " + set.name();
		s += "\n_____";
		return s;
	}
```

   j. That's the toString method. I wrote it in a specific way so that the output of cards will look the way mine do. It'll be full of errors right now. Don't worry about it, we haven't implemented all of the classes or interfaces yet so as soon as those are done, it'll stop erroring.

2. Next, we're going to extend the “Card” interface into more specific types of interfaces: Minions, Spells, Heroes, and Weapons. We’ll start with minions. So, create a new interface and call it MinionCard. Make sure you select "add" and choose the "Card" interface for it to extend. Once you've done that, hit finish and we'll start creating the interface.

a. The first thing we need to do is create a enum called CreatureType:

```
public enum CreatureType {

        //Enum types go here, seperated by commas. They DON'T end in a semicolon.

}
```

b. The types of enums want you to add to that are as follows (Make sure they're in all caps, seperated by a comma, and DON'T end with a semi-colon):

```
GENERAL
BEAST
DEMON
DRAGON
ELEMENTAL
MECH
MURLOC
PIRATE
TOTEM
```

If you need help, refer to the "Card" interface and make yours look like the ones in there.

c. Next, you'll be creating ANOTHER enum. This one should be called "Ability". The types you need to put into it are as follows:

```
CHARGE
TAUNT
TRIGGERED_EFFECT
DEATHRATTLE
INSPIRE
POISON
SPELL DAMAGE
ONGOING_EFFECT
STEALTH
LIFESTEAL
WINDFURY
SILENCE
ENRAGE
FREEZE
DIVINE_SHIELD
IMMUNE
```

d. Once you've completed those, we can move on to method definitions. Minion cards have a few methods that will be specific to them that you'll need to implement. Those are:

```
getAttack()            -> Returns an int
getCurrentAttack()     -> Returns an int
getHealth()            -> Returns an int
getCurrentHealth()     -> Returns an int
getAbilities()         -> Returns an Ability enum array (i.e. Ability[])
getCreatureType()      -> Return a CreatureType enum (i.e. CreatureType)
```

You're going to get pretty good at making these, because we're about to make 3 more interfaces that extend the "Card" interface.

3. Create a new interface called "SpellCard". Make sure it extends the "Card" interface.
    a. We'll need to add a single enum type to this interface. It should be called "SpellType" and the types should be:

        NONE
        SECRET
        QUEST

    b. You'll also need to define a single method in this interface:

        getSpellType() -> Returns a SpellType enum (i.e. SpellType)

    That's it for the SpellCard interface! Easy enough.

4. Next, create a new interface called "WeaponCard". Make sure it extends the "Card" interface.
    a. Weapon card needs a single enum. It should be called "Ability", and the types should be:

        TRIGGERED_EFFECT
        DEATHRATTLE
        POISON
        SPELL_DAMAGE
        ONGOING_EFFECT
        LIFESTEAL
        WINDFURY
        FREEZE
        IMMUNE

    b. The interface will also need 5 method definitions:

        getAttack() -> Returns an int
        getCurrentAttack() -> Returns an int
        getDurability() -> Returns an int
        getCurrentDurability() -> Returns an int
        getAbilities() -> Returns an Ability enum array (i.e. Ability[])

        Once you've done that we can move on to the final interface type!

5. Create a new interface called "HeroCard". Make sure it extends the "Card" interface.
    a. HeroCard will have one enum called Ability. Its types should be:

        BATTLECRY
        ONGOING_EFFECT
        TRIGGERED_EFFECT

    b. It should also have one method definition.

        getAbilities() -> Returns an Ability enum array (i.e. Ability[])

    c. You should've noticed by now that some of our enums have the same name. We can do this because, in order to access the enum, we need to have access to an implementation of its parent class or interface first. This means that if I call "Ability." inside a class that implements MinionCard, I'll get the minion card list, and if I do it inside "HeroCard", I'll get the HeroCard list. In a situation that I could potentially get both, I would need to specify which type I wanted by calling the interface name first, like so: HeroCard.Ability.BATTLECRY or MinionCard.Ability.DEATHRATTLE. Easy enough.

6. Now we just need to create classes that implement our interfaces and class definitions! We'll start with the easy one.
    a. Create a new class called "Spell". Make its SuperClass "PlaceholderCard" and then make it implement the interface "SpellCard". Once you've done that, click finish and we'll get to work!
        i. The first thing you should do is get rid of the "Type type" variable that's being passed into the constructor. Once you've deleted that, you should get an error in the "super()" call. Replace the "type" variable that was being passed in with "Type.SPELL". We're calling the enum type from the "Card" interface and using it here We have access to it because PlaceholderCard implements Card.
        ii. Next, add one more variable to be passed in at the end of the constructor "SpellType spellType". Then, store this variable in an instance variable with "protected" scope.
        iii. Finally, go down to the @override method and make sure you're returning the instance variable "spellType". That's it for the "Spell" Class. Next, we'll do Weapon.
    b. Create a new class called "Weapon". Make sure its superclass is "PlaceholderCard" and that it implements "WeaponCard".
        i. First things first, delete the "Type type" variable in the constructor and change it in the super() call to be "Type.WEAPON".
        ii. Then, at the end of the constructor you'll need to add 3 variables: int attack, int durability, Ability[] abilities
        iii. Once you add those, store them in instance variables (with protected scope). Also, create 2 more instance variables called currentAttack and currentDurability and set them to attack and durability respectively.
        iv. Next, ensure all of the override methods are returning the correct values. (I.e. getDurability returns the instance variable durability, etc.)
    c. Next, we'll do Hero. Create a class called "Hero". Make sure its parent/superclass is "PlaceholderCard" and that it implements HeroCard.
        i. Again, get rid of Type type, and make sure that you replace "type" in super() with "Type.HERO". You only need to pass in one variable for this one: Ability[] abilities. Make sure you store it in a protected instance variable and return it properly in the override method.
    d. Last but not least is Minion. Create class called "Minion". Make sure its parent is "PlaceholderCard" and that it implements MinionCard.
        i. Make sure you remove "Type type" from the constructor and "type" from the super() call, replacing it with "Type.MINION".
        ii. You'll need to take in 4 more variables in the constructor: int attack, int health, Ability[] abilities, CreatureType creatureType
        iii. Make sure you store them in protected instance variables. Also, create two more instance variables called currentAttack and currentHealth. Set them equal to attack and health respectively.
        iv. After you do that, ensure all of the override methods are returning the correct values!
7. That's it! Run the program and see if your output matches mine!