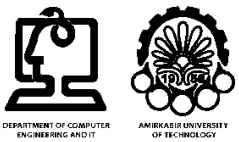

برنامه نویسی پیشرفته: تمرین سری ششم

تاریخ تحویل: تا سه شنبه ۲/۲۰ ساعت ۲۳:۵۵ (حداکثر تا سه روز بعد از این تاریخ قادر به ارسال تمرینات هستید و به ازای هر روز ۲۵% نمره کسر می‌گردد)

توصیه می‌شود درباره تمرینات با همکلاسی‌های خود به صورت گروهی به بحث و تبادل نظر بپردازید اما این به معنای تقلب، کپی کردن و... نمی‌باشد و تمام تمرینات باید توسط شما حل و پیاده سازی شود.

تمام تمرینات در یک پوشه با شماره دانشجویی و شماره تمرین و با ساختار به صورت زیر قرار گیرد، در غیر این صورت به عنوان عدم دریافت تلقی می‌شود.

HW_6_9531901

***توجه:** تمام قسمتهای تمرین در یک پروژه و در پکیج‌های مناسب قرار گیرد و قرار دادن غیرمنطقی تمام کلاسها در یک پکیج باعث کسر نمره خواهد شد.

****توجه:** تمام پروژه باید در یک repository **گیت لب** و به صورت خصوصی ذخیره شده باشد و کامیت‌های شما باید روال منطقی داشته باشند و تنها قرار دادن پروژه در گیت لب نمره‌ای نخواهد داشت.

کارت بازی (۱۰۰ نمره)

هدف این تمرین فهمیدن مفاهیم چندریختی، وراثت، ابسترکتشن و اینترفیس است و در آن میخواهیم یک بازی کارتی (مثل بازی Heroes of Warcraft که در سالهای اخیر محبوبیت بسیاری پیدا کرده است) را شبیه سازی کنیم.

در این تمرین شما باید مفاهیمی مانند بازیکن، زمین بازی و کارت را با استفاده از کلاسهای مختلف پیاده سازی کنید. همچنین یک برنامه تست (DuelMonsters) در پیوست داده شده است که میتوانید برنامه خود را با آن تست کنید. دقت کنید که این برنامه تا وقتی تمام کلاسهای شما به درستی پیاده نشده اند کامپایل نمیشود لذا پیشنهاد می‌شود ابتدا تمام کلاسهای مربوط در DuelMonsters را پیاده کنید تا این کلاس اجرا

شود (توابعی که مقداری برایتان بازمیگردانند را به نحوی هندل کنید که ابتدا یک مقدار الکی برگردانند و هنگامی که کد شما به درستی کامپایل شد کم‌کم به تکمیل بخشهای دیگر کد خود بپردازید).

قوانین بازی

برخی قوانین بازی در کلاس DuelMonsters نوشته شده اند ولی شما باید تمام قوانین زیر را پیاده سازی کنید:

- This game is two players (in the tester, there is just a simple cpu that acts as the second player, and all it does is put down cards from its hand).
- There are three broad categories of Card, which are Monster Cards, Spell Cards, and Trap Cards.
- Monster Cards have an attack power, and can attack an enemy's monster cards.
- Each player can have at most five monster cards on the field at once.
- Monster cards may only attack once per turn.
- Monster cards can only attack Monsters with equal or lower attack power.
- Spell Cards have an effect that happens each turn, and an effect that happens when they are destroyed. A player can have at most five spell cards on the field at once.
- Trap Cards have an instant effect when played, but do not actually have a spot on the field.
- Cards may be Special, which means they have an instant effect when played. All Trap Cards are considered Special.
- Each player has a hand that can have at most five cards.
- Each player has a main deck, from which they can draw one card per turn (though each player starts the game with a full hand). In the event that a player has enough space in their hand to draw another card, but has no more cards in their main deck, they instead lose 500 life points as a penalty.
- Each player has a special deck, which contains only Special cards. They can hold one of these cards in addition to their hand at any given time, and may draw another at the end of their turn should they use the one they have.
- Cards played from the special deck will have an instant effect when played.
- Each player has some number of life points. Life points are lost when monsters battle with each other (the amount lost is the difference between the monsters attack power). When a player loses all their life points, the game ends.
- Players may play as many of the cards from their hand during a turn as they want, and amy also play their special card. The only restriction on the number of cards that can be played is the number of cards that fit on the field.

توضیحات کلاسها:

همانطور که در بالا گفته شد، ابتدا تمام کلاسها و متدهای خواسته شده در کلاس DuelMonsters را پیاده سازی کنید و سپس به سراغ کلاسهای که در پایین گفته شده بروید (هرکلاس را قبل از اینکه به کلاس دیگری بروید کامپایل و تست کنید).

Card.java

Represents a card that can be played in the game. This is an abstract class, and should not be able to be instantiated.

This class has the following private fields, **and associated getter and setter methods for them**:

- `name` which is a String that represents the name of the card.
- `description` which is a String that represents the description of the card.

This class has the following constructors:

- One that takes in a String for the name, and a String for the description (in that order) and sets their corresponding instance fields.

This class has the following public methods in addition of the getters and setters:

- A properly overridden equals method that compares cards based on their names and descriptions.
- A properly overridden toString method that returns a String in the format: `name: description`.

Special.java

This is an interface that a card can implement if it can be summoned to the field from the special deck, and instantly cause an effect when summoned.

This interface has the following public methods:

- `void instantEffect(Field owner, Field enemy)`

ObjectDeck.java

Represents a Deck of Objects (think a deck of Cards, but not as specific). This is an abstract class and should not be able to be instantiated. It is used to hold a number of items, and can hold them in the form of an array.

This class has the following constructors:

- One that takes in an array of objects, and stores it internally.

This class has the following public methods:

- `Object deal()` which returns the next item in the deck (should return items starting from the highest index down to the lowest index), or null if no items remain in the deck. When an object is dealt from the deck, it should not be possible for the Deck to deal the item a second time.

- `int size()` which returns the number of items still in the deck.
- `boolean isEmpty()` which returns true if there are no more items in the deck, and false otherwise.

CardDeck.java

Represents a deck of Cards specifically. This class is concrete, and **is an** ObjectDeck, making use of inherited methods so as to reuse as much code as possible (points will be deducted for unnecessarily duplicating code).

This class has the following constructors:

- One that takes in a **variable number** of Cards and stores them internally. Hint: this constructor can be done in one line.

This class has the following public methods:

- You should override the deal method to instead return a Card instead of an Object. Hint: this method can be done in one line.

SpecialDeck.java

Represents a deck of Specials specifically. This class is concrete, and **is an** ObjectDeck, making use of inherited methods so as to reuse as much code as possible (points will be deducted for unnecessarily duplicating code).

This class has the following constructors:

- One that takes in a **variable number** of Specials and stores them internally. Hint: this constructor can be done in one line.

This class has the following public methods:

- You should override the deal method to instead return a Special instead of an Object. Hint: this method can be done in one line.

MonsterCard.java

Represents a Monster card in the game, that can attack other monster cards and thereby inflict damage to the opponent's life points. This **is a** Card, and should reuse as much code as possible. Points will be deducted for unnecessarily duplicating code. This is a concrete class.

This class has the following private fields, **and associated getter and setter methods for them**:

- `int power` which is the current attack power of the monster.
- `int basePower` which is the initial attack power of the monster when it was created. This field does not need a setter, as it should not be changed from the outside.
- `boolean canAttack` which is whether or not the monster can attack on a given turn.

This class has the following constructors:

- One that takes in a String for the name, and a String for the description, an int for the power, and a boolean for if it can attack (in that order) and sets their corresponding instance fields.

- One that does not take in a boolean for if it can attack, and sets a default value of false for canAttack. (Hint: this constructor should be done in one line).

This class has the following public methods in addition of the getters and setters:

- A properly overridden equals method that compares cards based on their names, descriptions, powers, and basePowers. Remember that you can reuse code from the Card class here, and you need not rewrite code for comparing the names and descriptions!
- A properly overridden toString method that returns a String in the format: `name: description | Power: power | Can attack: canAttack`. Again, remember that you have already written some of the code for this!

SpellCard.java

Represents a spell card that has an effect that it applies every turn, and also applies some effect when it gets destroyed. This is an abstract class and should not be able to be instantiated. This **is a** Card, and should reuse as much code as possible.

This class has the following constructors:

- One that takes in a String for the name, and a String for the description (in that order) and sets their corresponding instance fields.

This class has the following public methods:

- `abstract void turnEffect(Field ownerField, Field enemyField)` which is the effect that gets applied every turn.
- `abstract void destroyedEffect(Field ownerField, Field enemyField)` which is the effect that gets applied when the card is destroyed.
- A properly overridden equals method that compares cards based on their names and descriptions. Again, remember to reuse code, and be careful to make sure that this will not return true for a MonsterCard of the same description and name for example.

TrapCard.java

Represents a card that has some instant effect and does not actually get placed onto the Field. This is an abstract class that **is a** Card and also **is** Special.

This class has the following constructors:

- One that takes in a String for the name, and a String for the description (in that order) and sets their corresponding instance fields.

This class has the following public methods:

- A properly overridden equals method that compares cards based on their names and descriptions. Again, remember to reuse code, and be careful to make sure that this will not return true for a MonsterCard of the same description and name for example.

Field.java

Represents a single player's side of the game field that the cards can be placed on. There are two of these in the DuelMonstersAbridged class, where their interactions are managed. This is a concrete class.

This class has the following private fields, **and associated getter methods for them (you need not write setter methods for this class)**:

- `MonsterCard[] monsters` which is an array of length 5 that stores the monsters currently on the field.
- `SpellCard[] spells` which is an array of length 5 that stores the spells currently on the field.

This class has the following public methods:

- `void cardTurnEffects(Field enemyField)` which should apply the turnEffects for every spell currently on this field, and also set all monsters on the field to be able to attack. This gets run at the end of the player's turn.
- `boolean addMonsterCard(MonsterCard card)` which will add card to the first null spot in the monsters array and return true, or return false if there is no space in the array for it to be added.
- `boolean addSpellCard(SpellCard card)` which will add card to the first null spot in the spells array and return true, or return false if there is no space in the array for it to be added.

Player.java

Represents a player in the game that has decks and a hand and such. This is a concrete class.

This class has the following private fields, **and associated getter and setter methods for them**:

- `CardDeck mainDeck` which will be the main deck for the Player's Cards. You need not write getters and setters for this.
- `SpecialDeck specialDeck` which will be the deck for the Player's Special Cards. You need not write getters and setters for this.
- `Card[] hand` which will be an array of length 5 that stores the Cards in the player's hand.
- `Special nextSpecial` which will be the player's special card that they may play.
- `int lifePoints` the number of life points the player has remaining.

This class has the following constructors:

- One that takes in, in this order, a CardDeck for the main Deck, a SpecialDeck for the special Deck, and an int for the lifepoints, and sets their corresponding instance fields.
- One that does not take in lifepoints, and instead sets lifepoints to a default of 5000 (again, this should be done in one line).

This class has the following public methods:

- `boolean draw(int count)` will attempt to draw count cards from the mainDeck into the hand array. It should fill up null spots in the array starting from the beginning. In the event that there is a null spot in the hand, but not enough cards in the mainDeck to fill it, false should be returned. Otherwise return true.
- `void drawSpecialCard()` should fill nextSpecial with the next card in the specialDeck if nextSpecial is null.
- `void nextTurnPrep()` which will attempt to draw one card from the mainDeck, and if it cannot, will deduct 500 lifepoints. Will also attempt to draw a Special card into the nextSpecial position.

- `boolean playCardFromHand(int whichCard, Field myField)` should add the card at index whichCard in the hand onto myField. Make sure to check that the passed in index is within the bounds of hand. MonsterCards and SpellCards should be added to the field appropriately, while TrapCards should not be added (as they are just for quick play). Returns whether or not the card was successfully played (the play fails if there is not room on the field, or if the index is out of range, or if the card at the selected index is null). Should also remove the played card from the hand.
- `boolean playSpecial(Field myField)` works similarly to playCardFromHand, except it only tries to play nextSpecial. Remember to clear nextSpecial if it is played successfully.
- `void changeLifePoints(int change)` will add change to the current lifePoints.
- `boolean isDefeated()` returns whether or not lifepoints are less than or equal to zero.

BlueEyesWhiteDragon.java

Represents a Blue Eyes White Dragon monster, that also has a Special effect. This is a concrete class.

This class has the following constructors:

- A no-args constructor that sets its name to “Blue Eyes White Dragon”, its description to “The best card.”, its attack to 3000, and makes it so that this card may attack.

This class has the following public methods:

- This class should implement any remaining abstract methods. Since Kaiba loves this card so much, he loves being able to summon a bunch of them to the field all in one turn. So, the special instant effect of this card should be to add two new Blue Eyes White Dragons to its owner’s side of the field (add at most two dragons, or just fill up the remaining spots if there are less than two spaces).

PowerCard.java

Represents a Spell Card that increases the power of monsters on its owner’s side of the field each turn, or decreases their power once destroyed.

This class has the following constructors:

- A no-args constructor that sets its name to “Power Card”, and its description to “Increases power of monsters by 100 each turn.”

This class has the following public methods:

- It should implement any remaining abstract methods such that the effect it has each turn is to increase the power of all monsters on its owner’s side of the field by 100. When destroyed, this card should decrease the power of all monsters on its owner’s side of the field by 300.

DestroySpell.java

Represents a Trap Card that has the ability to destroy an opponent’s Spell Card.

This class has the following constructors:

- A no-args constructor that sets its name to “Destroy Spell” and its description to “Destroys the enemy’s first spell card.”.

This class has the following public methods:

- This should implement any remaining abstract methods such that its instant effect when played is to destroy the enemy’s first spell card (i.e the first spell card it finds when searching through the enemy’s active spell cards on the field).

چند نکته:

- A good way to test your implementation is to play games with the tester and make sure that all of the rules of the game are being followed, and that nothing unexpected is happening. You shouldn’t need to modify the tester too much to this end, but within DuelMonsters.java, there is a method called `private static void setupPlayers()` that puts together the decks and instantiates the players. If you make your own cards to test functionality, you can add them into the decks here.
- The current way the decks are setup, Red Eyes Black Dragon should be the first card that you draw from the mainDeck in the game, and Blue Eyes White Dragon should be your first special card. Your hand should also not be all Red Eyes Black Dragons, as there is only one of those in the Deck.
- You can check the dynamic (runtime) type of an object by using the `instanceof` keyword.
- If you have a couple similar methods, see if you can make a private helper method to avoid having to duplicate code.
- If you have a couple similar methods, see if you can make a private helper method to avoid having to duplicate code.
- **java.util.Arrays is not allowed.**
- Set, Map, List is not allowed too!