

## Final Examination

---

This is an open-note, open-book exam. You may not use any internet devices. You will be graded on functionality – but good style saves time and helps graders understand what you were attempting. You do not need to include any libraries. You have 180 minutes. We hope this exam is an exciting journey.

First Name:

---

Last Name:

---

SUNET ID

---

I commit to the letter and spirit of the honor code. I agree not to access any unauthorized resources for the duration of the exam.

(sign here) \_\_\_\_\_

**Note:** there is a method reference at the end of this exam.

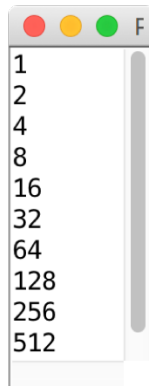


## Problem 1: Java of Wakanda (18 points)

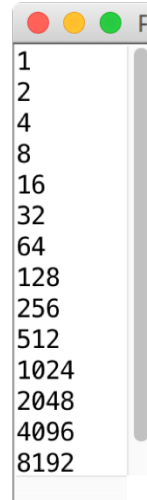
### A) Powers of Two

Fill in the provided `powersOfTwo` method with code that prints out all powers of **two** between 1 and a parameter called 'max' passed into the method, inclusive. You may assume that max is  $\geq 1$ .

When max is  
1,000



When max is  
10,000



```
/* Answer for 1A */
// don't worry about importing packages
// print out all powers of two between 1 and max, inclusive
// max will be a  $\geq 1$ 
private void powersOfTwo(int max) {
    // TODO: your code here.
```



### B) XHosa Trace

This code was found on a recently discovered Wakandan computer. Luckily it is in Java. Unfortunately all of the variable names are in Xhosa, the language of Wakanda. Help us figure out what this program outputs! Trace the program and write down the results of all the print statements. Write a one sentence comment that describes what the method **isiqingatha** does.

```
/** Le nkqubo ihlola ingoma yokwenza i-algorithm kwiJava **/  
public void run() {  
    ArrayList<Double> xabiso = new ArrayList<Double>();  
    for(double a = 0; a < 10; a++) {  
        xabiso.add(a);  
    }  
    xabiso = isiqingatha(xabiso);  
    xabiso = isiqingatha(xabiso);  
}  
  
private ArrayList<Double> isiqingatha(ArrayList<Double> negalelo) {  
    ArrayList<Double> ixabisoElitsha = new ArrayList<Double>();  
    int a = 0;  
    while(a < negalelo.size() - 1) {  
        double isixa = negalelo.get(a) + negalelo.get(a + 1);  
        double umyinge = isixa / 2;  
        ixabisoElitsha.add(umyinge);  
        a += 2;  
    }  
    for(Double into : ixabisoElitsha) {  
        println(into);  
    }  
    println("---");  
    return ixabisoElitsha;  
}
```



```
/* Answer for 1B */  
// What does this program output?
```

```
// Give a one sentence description of the isiqingatha method:
```



### C) Print out all Values

Fill in the provided **printValues** method with code that prints out all values of the parameter **map**. you may assume that each value in the map is unique and thus you don't need to worry about duplicates.

```
/* Answer for 1C */
// don't worry about importing packages
// print out all values in the given hashmap.
// you are guaranteed that each value in the map is unique
// thus you don't need to worry about duplicates.
private void printValues(HashMap<String, String> map) {
    // TODO: your code here.
```



### D) Is Divisible By

Fill in the provided `isDivisibleBy` method with code that returns whether the first parameter **a** is divisible by the second parameter **b**.

We say that **a** is divisible by **b** if and only if (**a** divided by **b**) has remainder 0. You should handle the following edge cases:

- If **a** is less than **b**, you should return false.
- If **a** is  $\leq 0$ , you should return false.
- Similarly if **b** is  $\leq 0$ , you should return false.

```
/* Answer for 1D */
// don't worry about importing packages
// if a is less than b you should return false
// if either a or b is 0 or negative you should return false
// otherwise return true only if a divides by b perfectly
// with no remainder.
private boolean isDivisibleBy(int a, int b) {
    // TODO: your code here
}
```

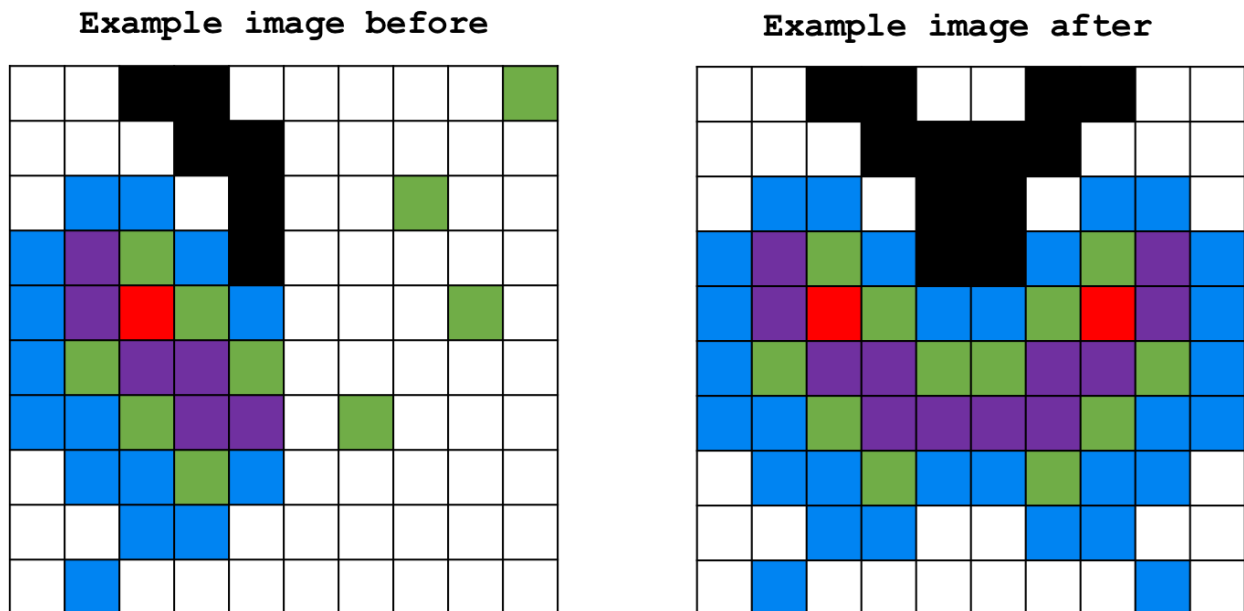


## Problem 2: Mirror Image (12 points)

Write a method:

```
private GImage mirrorImage(GImage input)
```

That takes in a **GImage**, **input**, and returns a new **GImage** where the right side of the image is a mirror reflection of the left side. As an example, consider this small **GImage** which is 10 pixels by 10 pixels:



In the resulting image, the left side is unchanged. However, the right side is replaced with a horizontal reflection of the left side of the image. In the horizontal reflection, each pixel on the right side of the image is copied from a "source" pixel on the left side which is (1) on the same row and (2) an equal distance from the horizontal center of the image.

### Assumptions and Requirements

- All images have an even number of columns.
- The number of rows is not necessarily equal to the number of columns.
- You should not modify the original **GImage**. Instead you should create a new one and return it.



```
/* Answer for Problem 2 */
// don't worry about importing packages

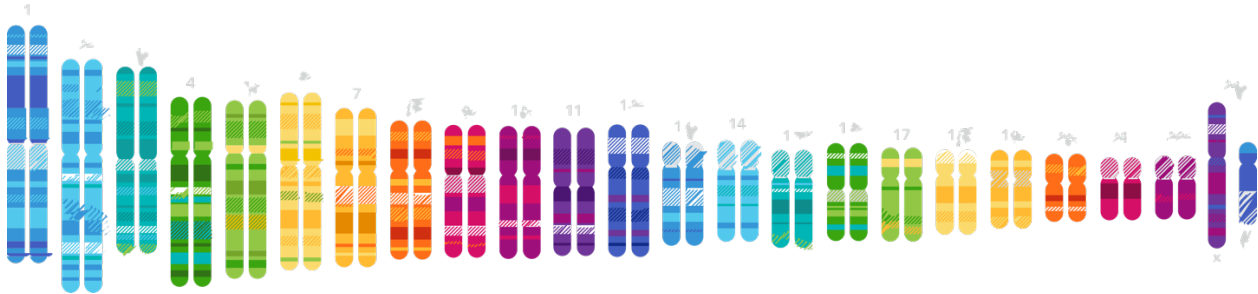
/** Mirror Image */
private GImage mirrorImage(GImage input) {
    int[][] pixels = input.getPixelArray();

    // TODO: your code here
}
```



### Problem 3: 23 and Character Frequency (18 points)

In this problem you are going to write two methods. The first method calculates frequencies of characters in a string. Among other uses, frequencies allow computer scientists to compute conditional probabilities such as the likelihood of a particular ancestry given a nucleotide.






#### A) Get Character Count

Write a method:

```
private HashMap<Character, Integer> getCharacterCount(String str)
```

that takes in an input string and returns the character frequency **HashMap**. Here are three examples:

Input String	Output HashMap																								
"ATTGCCA" 	<table><tr><td>Key:</td><td>A</td><td>T</td><td>G</td><td>C</td></tr><tr><td>Value:</td><td>2</td><td>3</td><td>1</td><td>2</td></tr></table>	Key:	A	T	G	C	Value:	2	3	1	2														
Key:	A	T	G	C																					
Value:	2	3	1	2																					
"Everyone's welcome!" 	<table><tr><td>Key:</td><td>E</td><td>V</td><td>R</td><td>Y</td><td>O</td><td>N</td><td>S</td><td>W</td><td>L</td><td>C</td><td>M</td></tr><tr><td>Value:</td><td>5</td><td>1</td><td>1</td><td>1</td><td>2</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	Key:	E	V	R	Y	O	N	S	W	L	C	M	Value:	5	1	1	1	2	1	1	1	1	1	1
Key:	E	V	R	Y	O	N	S	W	L	C	M														
Value:	5	1	1	1	2	1	1	1	1	1	1														
"successfully" 	<table><tr><td>Key:</td><td>S</td><td>U</td><td>C</td><td>E</td><td>F</td><td>L</td><td>Y</td></tr><tr><td>Value:</td><td>3</td><td>2</td><td>2</td><td>1</td><td>1</td><td>2</td><td>1</td></tr></table>	Key:	S	U	C	E	F	L	Y	Value:	3	2	2	1	1	2	1								
Key:	S	U	C	E	F	L	Y																		
Value:	3	2	2	1	1	2	1																		

The keys of the character frequency HashMap are the alphabetic characters in the input string, in upper case. The associated values are the number of times the key exists in the input string.

For example the letter '**E**' shows up 5 times in the string "**E**veryone's welcome!" (once as an upper case letter and four times as a lowercase letter). Thus the resulting map for that input would have the key '**E**' associated with the value 5.

You should ignore all non-alphabetic characters and capitalize all letters before counting them. The keys can be added to the hashmap in any order.



```
/* Answer for 3A */  
// don't worry about importing packages  
  
private HashMap<Character, Integer>  
getCharacterCount(ArrayList<String> samples, int i) {  
  
    // TODO: your code here
```



## B) Get First Non Repeat

Write a method, that, given a string input, finds the first non-repeated alphabetic character in the string. A character is "repeated" if it exists more than once in the input:

```
private Character getFirstNonRepeat(String str)
```

Again, ignore capitalization. Here are a few examples:

```
getFirstNonRepeat("ATTTGCCA") would return 'G'.  
Both 'A' and 'T' are repeated in the string. 'G' is the first non-  
repeat.  
  
getFirstNonRepeat("Everyone's Welcome") would return 'V'.  
The letter 'E' is repeated in the string but 'V' is not.  
  
getFirstNonRepeat("successfully") would return 'E'  
all of the letters 'S', 'U', and 'C' are repeated.  
  
getFirstNonRepeat("toot") would return null  
since both the 'T' and the 'O' are repeated.
```

If a string does not contain any non-repeating alphabetic characters, you should return **null**. You may use the method **getCharacterCount** defined in part A.



```
/* Answer for 3B */  
// don't worry about importing packages  
private Character getFirstNonRepeat(String str) {  
  
    // TODO: your code here
```



#### Problem 4: MPedigree Redux (18 points)

In class we talked about the MPedigree application where Bright Simmons put secret codes on medicine boxes in order to help users determine if the medications they bought are authentic or counterfeit:



In this question we are going to write a server which can handle requests from users to check if medicine is authentic. Specifically, you should write a server program **MPedigreeRedux** that can load generated codes from file and respond to "check" web requests. **Importantly**, you are responsible for letting the user know if the code on their recently purchased medicine has been "checked" before.

#### The Secret Codes File

In this program you do **not** need to generate codes. Instead, all codes which were put on authentic medicine boxes have been saved in a file called **"secretCodes.txt"**. The file contains one code per line. Here is what the file **"secretCodes.txt"** looks like:

```
18291742
91827482
11220023
...
25041988
```

You will need to load the contents of the secret codes file at the start of your program. If Java throws an Exception while you are reading the file you should print out a message that says "The secret code file could not be loaded".

#### Check Request

The main job of your program is to start a server which can handle web requests to "check" codes



```
private String requestMade(Request r)
```

The server only needs to process requests with the command **"check"** and a single parameter **"code"** (which stores the code that the user wants to validate).

Here is an example server log that shows requests / responses for a working mpedigree redux. This example is based off the codes in the sample **"secretCodes.txt"** file. Note: your server does **not** need to print out anything -- this server log is for demonstration purposes only. Here requests are printed to the console with the command, followed by the parameters in parenthesis, and the returned response:

```
Starting MPedigree Server...
```

```
check (code=91827482)
=> success
```

```
check (code=12345678)
=> invalid code
```

```
check (code=18291742)
=> success
```

```
check (code=91827482)
=> already checked
```

```
check (code=91827482)
=> already checked
```

```
check ()
=> missing parameter
```

```
ping ()
=> unknown command
```

The logic behind your server responses is as follows:

- If the request **command** is not **"check"**, return the string **"unknown command"**.
- If the request is missing a **"code parameter"**, you should return the string **"missing parameter"**.
- If the **code parameter** is not in the codes file, you should return the string **"invalid code"**.
- If the **code parameter** is in the codes file you should return the string **"success"** if and only if this is the **first time** the code has been requested.
- If the **code parameter** is in the codes file and the code has **already** been checked you should return the string **"already checked"**.



Knowing if a particular code has been checked before is an important feature for preventing counterfeiting. You will need to come up with a way to recall if you have already received a request for a give code.

### Request Reference

This program uses the same SimpleServer as Assignment 7. For your convenience here is the documentation of the public methods that you can call on a Request instance:

String	<b>getCommand()</b> Returns the command attached to the request.
boolean	<b>hasParam</b> (String key) Returns whether or not the request has a param with the given key.
String	<b>getParam</b> (String key) Returns the value, in String form, associated with the given param key



```
/* Answer for Problem 4 (two pages) */
// don't worry about importing packages

/** MPedigree Redux **/
public class MPedigreeRedux extends ConsoleProgram implements
SimpleServerListener{

    // the server instance
    private SimpleServer server = new SimpleServer(this, 8000);

    // the file with all of the secret codes
    private String CODE_FILE_NAME = "secretCodes.txt";

    public void run() {
        // TODO: read the code file

        server.start();
    }
}
```



```
public void requestMade(Request r) {  
    // TODO: handle requests
```

```
    }  
}
```



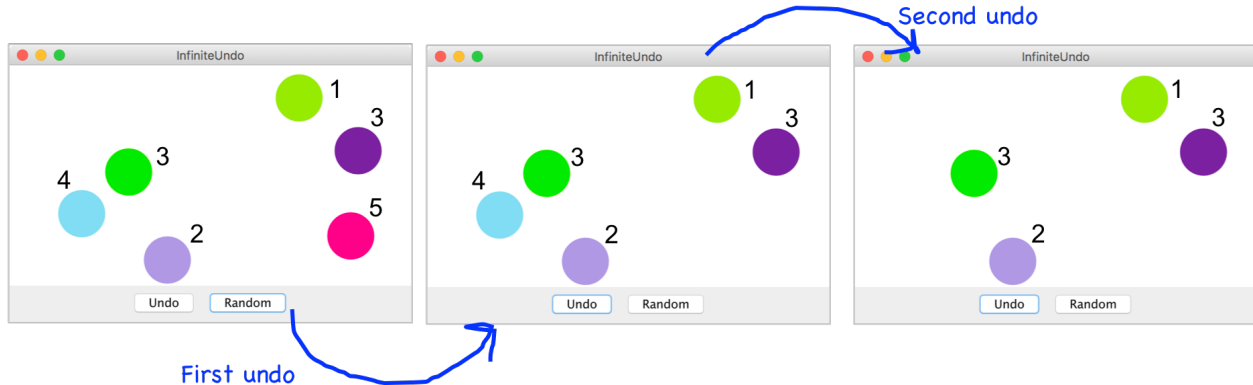
### Problem 5: Infinite Undo (18 points)

One of the greatest features of computers is their ability to "undo" mistakes. If only real life had that property as well... Write a Graphics Program:

```
public InfiniteUndo extends GraphicsProgram
```

That has two buttons, "Undo" and "Random." Each time the user hits the "Random" button a circle with random color is placed on the canvas. When the user hits "Undo" the most recently added circle is removed.

Consider this example. The user has added five circles (which I annotated with the order in which they were put on the screen). She then hits "Undo" twice. The first time she hits undo the **pink (5)** colored circle in the bottom right is removed as it was the most recently added circle. The second time she hits undo the **light blue (4)** circle on the left side of the canvas is removed. You do **not** have to draw numbers annotating when circles were added.



The user can intersperse hitting the Random button and the Undo button as she likes.

### Assumptions and Requirements

- All circles have a diameter of 50 pixels.
- Circles are placed randomly with random color.
- Circles are filled.
- All circles must be placed completely on the screen.
- The screen could initially be any size.
- The screen will not resize.
- If there are no circles on the screen and the user hits Undo, nothing should happen. The program should not throw an error.



```
/* Answer for Problem 5 */  
// don't worry about importing packages  
  
/** Infinite Undo */  
public class InfiniteUndo extends GraphicsProgram {  
  
    // TODO: your code here
```







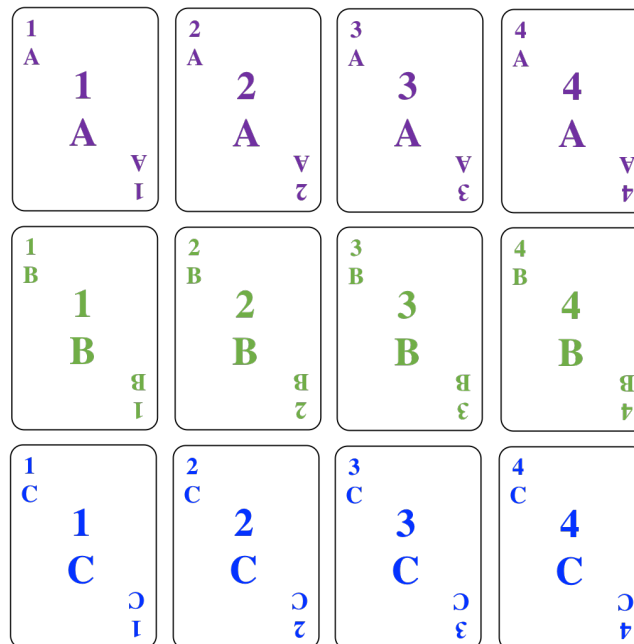
## Problem 6: General Card Deck (18 points)

Write a class:

```
public class GeneralCardDeck
```

which defines a variable type for a general card deck. This general card deck will not necessarily have the 52 cards of a standard deck (13 values with 4 suits). Instead, our card deck will be created with a specified number of values and a specified number of suits. We denote suits with letters starting from 'A'. We denote values as integers starting from '1'.

For example a general card deck could be defined to have four values ('1', '2', '3' and '4') and three suits ('A', 'B' and 'C') and thus would have the following twelve cards, each with a unique suit / value combination:

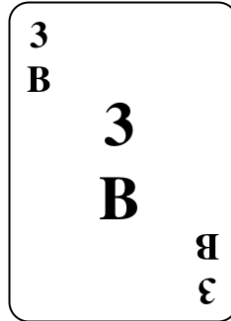


In this question you will *define* the **GeneralCardDeck** variable type. In the next question you will *use* the **GeneralCardDeck** variable type to program a card game. You need to implement the specified constructor and public methods. You may implement the class using any private instance variables that you like.

### Card Representation

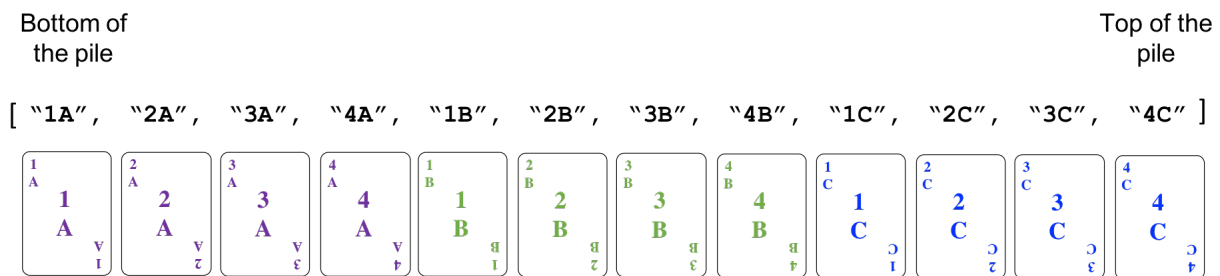
Each card is represented as a two character string. The first character is the value of the card, the second character is the suit. For example this card with value '3' and suit 'B' is represented by the string **"3B"**:





## Deck Representation

The cards in a general card deck are ordered such that they have a "top of the deck" and a "bottom of the deck" -- you must decide how to represent this ordered collection. As an example, if a card deck were made with three suits, four values, and was not shuffled, the deck might look something like this:



Your deck will allow users to take cards from it. However, once constructed, your deck will **not** allow users to add cards. Once a card is removed it will never be put back. Specifically you will need to implement the following.

## Constructor

You must implement a constructor:

```
public GeneralCardDeck(int numValues, int numSuits)
```

that takes in two parameters which define the quantity of cards in the deck: **numValues** and **numSuits**. The constructor should create one card for every value / suit combination. Card suits are represented by characters, starting with 'A', then 'B' and so on. The constructor does not need to shuffle the cards.

You may assume that **numValues** is an integer in the range 1 through 9 inclusive and that **numSuits** is an integer in the range 1 through 26 inclusive. Card values start at '1' and not at '0'.



## Public Methods

You must implement the following four public methods:

### Get Next Card

```
public String getNextCard()
```

Get next card removes a card from the top of the deck and returns the card, in string representation, to the caller. Once a card is removed it is no longer in the deck and that particular card should not be returned by getNextCard again. There is now one fewer card in the deck. If there are no cards left, getNextCard should return the empty string. Here is an example of getNextCard. In this example the method would return the string "4B" and remove that card from the deck:

	Bottom of the pile		Top of the pile
Before:	[	"1A", "2A", "3A", "4A", "1B", "2B", "3B", "4B"	]
After:	[	"1A", "2A", "3A", "4A", "1B", "2B", "3B"	]

### Get Num Remaining

```
public int getNumRemaining()
```

How many cards are remaining in the deck?

### Is Empty

```
public boolean isEmpty()
```

Return **true** if there are still cards in the deck and **false** otherwise.

### Shuffle

```
public void shuffle()
```

Shuffle randomly re-orders the cards in the deck. To shuffle the deck, use the following algorithm. Repeat 100 times: (a) randomly chose two card-positions in the deck and (b) switch the cards at those positions. Here is an example of the inner loop of the algorithm, with both steps a and b:

(a):	[	"1A",	<span style="border: 2px solid blue; border-radius: 50%; padding: 2px;">"2A"</span> ,	"3A",	"4A",	<span style="border: 2px solid blue; border-radius: 50%; padding: 2px;">"1B"</span> ,	"2B",	"3B"	]
(b):	[	"1A",	<span style="color: red;">"1B"</span> ,	"3A",	"4A",	<span style="color: red;">"2A"</span> ,	"2B",	"3B"	]



```
/* Answer for Problem 6 (three pages) */
// don't worry about importing packages

/** General Card Class */
public class GeneralCardDeck {

    // TODO: your instance variables here


    public GeneralCardDeck(int numValues, int numSuits) {
        // TODO: your code here


    }
}
```



```
public void shuffle() {  
    // TODO: your code here
```

```
}
```

```
public String getNextCard() {  
    // TODO: your code here
```

```
}
```



```
public int getNumRemaining() {  
    // TODO: your code here
```

```
}
```

```
public boolean isEmpty() {  
    // TODO: your code here
```

```
}
```

```
}
```



### Problem 7: Simple Set Game (18 points)

Write a `ConsoleProgram`:

```
public SimpleSetGame extends ConsoleProgram
```

That has a user play a game of Simple Set, a basic version of the game Set. In Simple Set, a Card Deck with three suits and three values is created and shuffled. The user is then repeatedly shown the top three cards from the deck and is asked to decide if the three cards make a "Set". Your program will then verify if the user was correct. The game continues until the deck is empty.



*You will implement a simple version of the game of set*

#### What makes a "Set"?

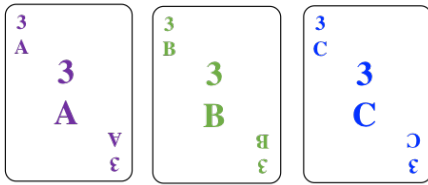
Three cards make a Set if each "feature" is either the same for all three cards OR different for all three cards. Since our Simple Set cards have two "features": value and suit, in order to be a Set three cards **must satisfy both**:

1. All three values are different OR all three values are the same
2. All three suits are different OR all three suits are the same

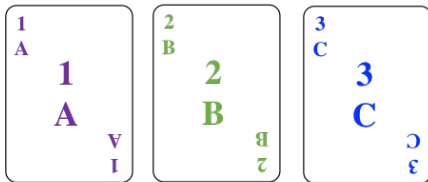
Here are several examples of three cards that are Sets and three cards that are not Sets:



### Yes, these are Sets

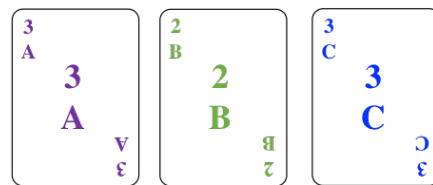


All three have the same value, 3.  
All three have different suits.

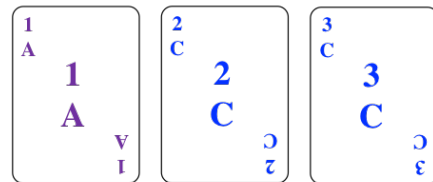


All three have different values.  
All three have different suits.

### No, these are not Sets



The cards neither all have different values  
Nor do they all have the same value.



The cards neither all have different suits  
Nor do they all have the same suit.

### User Interface

When the program starts you should create a **GeneralCardDeck**, the variable type defined in the previous problem, with 3 values and 3 suits. Don't forget to shuffle the deck before starting the loop! Then, until the deck is empty you should:

1. draw three cards from the deck.
2. show the three cards to the user.
3. ask the user to answer, in boolean form, whether or not the three cards make a Set.
4. verify if the user was correct. If they were correct, print "good job!". If they were not correct, tell them the right answer.

Here are three example runs:



Example Run #1

```

Welcome to Simple Set!
2A
3B
1C
Is this a set? true
Good job!

2C
1A
3A
Is this a set? true
It is not a set

3C
2B
1B
Is this a set? false
Good job!

```

Example Run #2

```

Welcome to Simple Set!
3B
1B
2B
Is this a set? false
It is a set

2A
1A
3C
Is this a set? false
Good job!

1C
3A
2C
Is this a set? false
Good job!

```

Example Run #3

```

Welcome to Simple Set!
1C
3C
2A
Is this a set? false
Good job!

3A
1A
2C
Is this a set? false
Good job!

2B
1B
3B
Is this a set? true
Good job!

```

### GeneralCardDeck Reference

You should use the **GeneralCardDeck** variable type that you defined in the previous problem. Regardless of whether you answered the last question, you may assume these methods have been implemented correctly. For your convenience here is the documentation of the public methods that you can call on a **GeneralCardDeck** instance:

	<b>GeneralCardDeck</b> (int numValues, int numSuits) Creates a card deck with numValues × numSuits cards.
String	<b>getNextCard</b> () Returns the next card from the deck, in String form.
int	<b>getNumRemaining</b> () Returns how many cards are still in the deck.
boolean	<b>isEmpty</b> () Returns true only if there are more cards left in the deck.



```
/* Answer for Problem 7 */  
// don't worry about importing packages  
  
/** Simple Set Game */  
public class SimpleSetGame extends ConsoleProgram {  
  
    // TODO: your code here
```







## Reference

### ArrayList

Mostly Complete ArrayList Documentation

Return	Method and Description
boolean	<b>add</b> (E e) Appends the specified element to the end of this list.
void	<b>add</b> (int index, E element) Inserts the specified element at the specified position in this list.
boolean	<b>addAll</b> (Collection<? extends E> c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator.
boolean	<b>addAll</b> (int index, Collection<? extends E> c) Inserts all of the elements in the specified collection into this list, starting at the specified position.
void	<b>clear</b> () Removes all of the elements from this list.
Object	<b>clone</b> () Returns a shallow copy of this ArrayList instance.
boolean	<b>contains</b> (Object o) Returns <b>true</b> if this list contains the specified element.
E	<b>get</b> (int index) Returns the element at the specified position in this list.
int	<b>indexOf</b> (Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	<b>isEmpty</b> () Returns <b>true</b> if this list contains no elements.
int	<b>lastIndexOf</b> (Object o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
E	<b>remove</b> (int index) Removes the element at the specified position in this list.
boolean	<b>remove</b> (Object o) Removes the first occurrence of the specified element from this list, if it is present.
boolean	<b>removeAll</b> (Collection<?> c) Removes from this list all of its elements that are contained in the specified collection.
protected void	<b>removeRange</b> (int fromIndex, int toIndex) Removes from this list all of the elements whose index is between <b>fromIndex</b> , inclusive, and <b>toIndex</b> , exclusive.



boolean	<b>retainAll</b> (Collection<?> c) Retains only the elements in this list that are contained in the specified collection.
E	<b>set</b> (int index, E element) Replaces the element at the specified position in this list with the specified element.
int	<b>size</b> () Returns the number of elements in this list.
List<E>	<b>subList</b> (int fromIndex, int toIndex) Returns a view of the portion of this list between the specified <b>fromIndex</b> , inclusive, and <b>toIndex</b> , exclusive.
Object []	<b>toArray</b> () Returns an array containing all of the elements in this list in proper sequence (from first to last element).

## HashMap

### Mostly Complete HashMap Documentation

Return	Method and Description
void	<b>clear</b> () Removes all of the mappings from this map.
Object	<b>clone</b> () Returns a shallow copy of this <b>HashMap</b> instance: the keys and values themselves are not cloned.
boolean	<b>containsKey</b> (Object key) Returns <b>true</b> if this map contains a mapping for the specified key.
boolean	<b>containsValue</b> (Object value) Returns <b>true</b> if this map maps one or more keys to the specified value.
V	<b>get</b> (Object key) Returns the value to which the specified key is mapped, or <b>null</b> if this map contains no mapping for the key.
boolean	<b>isEmpty</b> () Returns <b>true</b> if this map contains no key-value mappings.
Set<K>	<b>keySet</b> () Returns a Set view of the keys contained in this map.
V	<b>put</b> (K key, V value) Associates the specified value with the specified key in this map.
void	<b>putAll</b> (Map<? extends K, ? extends V> m) Copies all of the mappings from the specified map to this map.
V	<b>remove</b> (Object key) Removes the mapping for the specified key from this map if present.
int	<b>size</b> () Returns the number of key-value mappings in this map.
Collection<V>	<b>values</b> () Returns a Collection view of the values contained in this map.



```
/** Scratch Paper **/  
// Random message for the CS106A teaching staff?
```