AP® Computer Science A Magpie Chatbot Lab Student Guide

The AP Program wishes to acknowledge and thank Laurie White of Mercer University, who developed this lab and the accompanying documentation.

Magpie Chatbot Lab: Student Guide

Introduction

From Eliza in the 1960s to Siri and Watson today, the idea of talking to computers in natural language has fascinated people. More and more, computer programs allow people to interact with them by typing English sentences. The field of computer science that addresses how computers can understand human language is called Natural Language Processing (NLP).

NLP is a field that attempts to have computers understand natural (i.e., human) language. There are many exciting breakthroughs in the field. While NLP is a complicated field, it is fairly easy to create a simple program to respond to English sentences.

For this lab, you will explore some of the basics of NLP. As you explore this, you will work with a variety of methods of the `String` class and practice using the `if` statement. You will trace a complicated method to find words in user input.

Activity 1: Getting Acquainted with Chatbots (Optional)

Chatbots are programs that are designed to respond like humans to natural language input. Before you write code to create your own chatbot, you will explore some existing chatbots.

Start

Go to chatbots.org. Try out several of the chatbots and find one to use for this activity. Your teacher may have a specific list of chatbots for you to try.

Exploration

Have several conversations with your chatbot and answer the following questions:

- How does it respond to "where do you come from"?
- What is the most interesting response?
- What is the most peculiar response?
- How does it respond to "asdfghjkl;"?

Exercises

Work with another group and have two different chatbots converse with each other.

Questions
Simple chatbots act by looking for key words or phrases and responding to them.

1. Can you identify keywords to which your chatbot responds?
2. Think of several keywords and the responses they might cause.

Activity 2: Introduction to the `Magpie` Class

In this activity, you will work `Magpie`, with a simple implementation of a chatbot. You will see how it works with some keywords and add keywords of your own.

Prepare
Have available:

- the code for the `Magpie`
- the code for the `MagpieRunner`
- a computer with your Java development tools

Start
Get to know the `Magpie` class. Run it, using the instructions provided by your teacher.

How does it respond to:

- My mother and I talked last night.
- I said no!
- The weather is nice.
- Do you know my brother?

Exploration
Look at the code. See how the `if` statement assigns a value to the response and returns that response. The method `getRandomResponse` picks a response from a group of `String` objects.

Exercises
Alter the code:

- Have it respond “Tell me more about your pets” when the statement contains the word “dog” or “cat.” For example, a possible statement and response would be:

Statement: I like my cat Mittens.

Response: Tell me more about your pets.

- Have it respond favorably when it sees the name of your teacher. Be sure to use appropriate pronouns! For example, a possible statement and response would be:

  Statement: Mr. Finkelstein is telling us about robotics.

  Response: He sounds like a good teacher.

- Have the code check that the statement has at least one character. You can do this by using the `trim` method to remove spaces from the beginning and end, and then checking the length of the trimmed string. If there are no characters, the response should tell the user to enter something. For example, a possible statement and response would be:

  Statement:

  Response: Say something, please.

- Add two more noncommittal responses to the possible random responses.
- Pick three more keywords, such as "no" and "brother" and edit the `getResponse` method to respond to each of these. Enter the three keywords and responses below.

Keyword	Response

- What happens when more than one keyword appears in a string? Consider the string "My mother has a dog but no cat." Explain how to prioritize responses in the reply method.

Question

1. What happens when a keyword is included in another word? Consider statements like "I know all the state capitals" and "I like vegetables smothered in cheese." Explain the problem with the responses to these statements.

Activity 3: Better Keyword Detection

In the previous activity, you discovered that simply searching for collections of letters in a string does not always work as intended. For example, the word “cat” is in the string “Let's play catch!,” but the string has nothing to do with the animal. In this activity, you will trace a method that searches for a full word in the string. It will check the substring before and after the string to ensure that the keyword is actually found.

You will use some more complex `String` methods in this activity. The `String` class has many useful methods, not all of which are included in the AP Computer Science Java Subset. But they can be helpful in certain cases, so you will learn how to use the API to explore all of the methods that are built into Java.

Prepare

Have available:

- the API for the `Magpie` class
- the API for the `String` class
- the code for the `StringExplorer`
- the code for the `Magpie`
- the code for the `MagpieRunner`
- a computer with your Java development tools

Exploration: Using the API

One of the major benefits of using Java as a programming language is that so many library classes have already been created for it.

Open the program `StringExplorer`. It currently has code to illustrate the use of the `indexOf` and `toLowerCase` methods.

Open the API for `String`. Scroll down to the Method Summary section and find the `indexOf(String str)` method. Follow the link and read the description of the `indexOf` method. What value is returned by `indexOf` if the substring does not occur in the string?

Add the following lines to `StringExplorer` to see for yourself that `indexOf` behaves as specified:

```
int notFoundPsn = sample.indexOf("slow");

System.out.println("sample.indexOf(\"slow\") = " + notFoundPsn);
```

Read the description of `indexOf(String str, int fromIndex)`. Add lines to `StringExplorer` that illustrate how this version of `indexOf` differs from the one with one parameter.

This lab activity will use a variety of different `String` methods. Consult the API whenever you see one with which you are unfamiliar.

Exploration: Understand the new method
This version of the `Magpie` class has a method named `findKeyword` to detect keywords. This method will only find exact matches of the keyword, instead of cases where the keyword is embedded in a longer word. Run it, using the instructions provided by your teacher.

```
private int findKeyword(String statement, String goal, int startPos)
{
  String phrase = statement.trim();
  int psn = phrase.toLowerCase().indexOf(goal.toLowerCase(), startPos);

  while (psn >= 0)
  {
    String before = " ", after = " ";
    if (psn > 0)
    {
      before = phrase.substring (psn - 1, psn).toLowerCase();
    }
    if (psn + goal.length() < phrase.length())
    {
      after = phrase.substring(psn + goal.length(),
                               psn + goal.length() + 1).toLowerCase();
    }
    /* determine the values of psn, before, and after at this point in the method. */
    if (((before.compareTo ("a") < 0 ) || (before.compareTo("z") > 0))
         &&
         ((after.compareTo ("a") < 0 ) || (after.compareTo("z") > 0)))
    {
      return psn;
    }
    psn = phrase.indexOf(goal.toLowerCase(), psn + 1);
  }

  return -1;
}
```

Read through the `findKeyword` method. To ensure that you understand it, trace the following method calls.

```
findKeyword("She's my sister", "sister", 0);

findKeyword("Brother Tom is helpful", "brother", 0);

findKeyword("I can't catch wild cats.", "cat", 0);

findKeyword("I know nothing about snow plows.", "no", 0);
```

Write the value of each of the variable `psn`, `before`, and `after` each time the program control reaches the point in the method indicated by the comment.

Example: `findKeyword("yesterday is today's day before.", "day", 0);`

Iteration	`psn`	`before`	`after`
1	6	`"r"`	`" "`
2	15	`"o"`	`"'"`
3	21	`" "`	`" "`

Use a copy of the table below to trace the calls.

Iteration	`psn`	`before`	`after`

Exercise: Use the new method

Repeat the changes you made to the program in Activity 2, using this new method to detect keywords.

Questions: Prepare for the next activity

Single keywords are interesting, but better chatbots look for groups of words. Consider statements like “I like cats,” “I like math class,” and “I like Spain.” All of these have the form “I like *something*.” The response could be “What do you like about *something*?” The next activity will expand on these groups. You will get to add one of your own, so it’s a good idea to start paying close attention to common phrases in your own conversations.

Activity 4: Responses that Transform Statements

As stated previously, single keywords are interesting, but better chatbots look for groups of words. Statements like "I like cats," "I like math class," and "I like Spain" all have the form "I like *something*." The response could be "What do you like about *something*?" This activity will respond to groupings of words.

Prepare

Have available:

- the API for the `Magpie` class
- the API for the `String` class
- the code for the `Magpie`
- the code for the `MagpieRunner`
- a computer with your Java development tools

Exploration

Get to know the revised `Magpie` class. Run it, using the instructions provided by your teacher.

How does it respond to:

- I want to build a robot.
- I want to understand French.
- Do you like me?
- You confuse me.

Exercises

Look at the code. See how it handles "I want to" and you/me statements.

Alter the code:

- Have it respond to "I want *something*" statements with "Would you really be happy if you had *something*?" In doing this, you need to be careful about where you place the check. Be sure you understand why. For example:

  Statement: I want fried chicken.
  Response: Would you really be happy if you had fried chicken?

- Have it respond to statements of the form "I *something* you" with the restructuring "Why do you *something* me?" For example:

  Statement: I like you.
  Response: Why do you like me?

Find an example of when this structure does not work well. How can you improve it?

Activity 5: Arrays and the Magpie (Optional)

When you last worked with the Magpie, default responses were handled with a nested `if` statement. This certainly worked, and you could add more responses, but it was a bit awkward. An easier way to keep track of default responses is with an array. In this activity, you will see how an array makes handling default responses much easier.

Prepare

Have available:

- the code for the `Magpie`
- the code for the `MagpieRunner`
- a computer with your Java development tools

Exploration

Run this version of the `Magpie` class. You should see no difference in its outward behavior. Instead, it has been changed so that its internal structure is different. This is called code refactoring. That's one of the big benefits of dealing with methods as black boxes. As long as they perform the action required, the user does not care about how they perform the action.

Read the code for `getRandomResponse`. Notice that it uses an array of responses.

Exercise

Alter the array to add four additional random responses. Notice that, because the `getRandomResponse` method uses the length attribute of the array, you do not need to change anything else.

Compile and run your code. You should run it until you see all of your new responses.

Current Work in NLP

There is much work going on with Natural Language Processing in a variety of areas:

- Spam filtering uses NLP to determine whether an email message is spam.
- Many businesses use virtual agents to provide assistance to customers on Web sites.
- Information retrieval parses text, such as email messages, and tries to extract relevant information from it. For example, some email programs will suggest additions to online calendars based on text in the message.
- Sentiment analysis takes information retrieval further. Rather than extract information from a single source, it goes to a variety of online resources and accumulates information about a particular topic. For example, sentiment analysis might follow Twitter to see how people are reacting to a particular movie.
- Question answering systems search a large body of knowledge to respond to questions from users. The best known question answering system is IBM's Watson.

Glossary

API — An abbreviation for Application Programming Interface. It is a specification intended to be used as an interface by software components to communicate with each other.

Chatbot — A program that conducts a conversation with a human user.

Code refactoring — A disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.

Magpie — Magpies are large black birds. They are thought to be among the most intelligent birds and are capable of mimicking human speech.

NLP — Natural Language Processing; the field that studies responding to, and processing, human language.

Virtual agent — Also known as an Intelligent Agent. This can be used to gather information from a customer so that appropriate action can be taken in response to a query.

References

General information

http://www.nlp-class.org. A free online NLP class from Stanford. The first video does a good job of explaining the problems and promises of NLP, although much of the material is at a much higher level.

http://nsf.gov/cise/csbytes/newsletter/vol1i4.html. An issue of the NSF Bits and Bytes Newsletter dedicated to NLP. Professor Mari Ostendorf of the University of Washington is featured in the newsletter.

Some chatbots

http://www-03.ibm.com/innovation/us/watson/what-is-watson/science-behind-an-answer.html. Numerous videos about the creation of Watson.

http://nlp-addiction.com/chatbot. Versions of the Eliza program, the first widespread chatbot.

Women in NLP Research

http://dotdiva.org/profiles/laura.html. A virtual nurse created by Laura Pfeifer.

http://people.ischool.berkeley.edu/~hearst. Professor Marti Hearst's homepage. She researches user interfaces to search engines.

Assessment

Multiple-Choice Questions

No questions will come directly from this material and it will not be provided as part of the Quick Reference for the exam. Instead, this material will lead to two types of multiple-choice questions: complex conditional questions and string manipulation questions.

1. Consider the following code segment.

```
if (a < b)
{
  if (b < c)
  {
    if (c < 10)
    {
      System.out.println("one")
    }
    else if (c < a)
    {
      System.out.println("two")
    }
  }
}
else
{
  if (c < a)
  {
    System.out.println("three")
  }
  else
  {
    System.out.println("four")
  }
}
```

For which values of `a`, `b`, and `c` will the code print "`one`"?

I. `a = 5, b = 6, c = 7`
II. `a = 8, b = 7, c = 6`
III. `a = 10, b = 20, c = 30`

(A) I only
(B) II only
(C) III only

(D) I and III

(E) I and II

2. What is the output of the following code segment?

```
String phrase = "Here is the word";
int psn = phrase.indexOf("e");
while (psn >= 0)
{
   System.out.print(psn + " ");
   phrase = phrase.substring(psn + 1);
   psn = phrase.indexOf("e");
}
```

(A) `1 1 6`

(B) `2 2 7`

(C) `1 3 10`

(D) `2 4 11`

(E) Many digits will be printed due to an infinite loop.

Free-Response Question

Free-response questions related to this lab will typically focus on text processing. One possible question is shown below.

When Web developers want to convert plain text documents to HTML, they need to make changes to the text to insert tags and replace certain characters. For this question, you will write part of the class to convert text to HTML. You will write methods to replace underscores `("_")` with tags indicating the text should be in italics.

The class definition for this problem is given below:

```
public class TextFormatter
{
  private String line; //  The line to format

  public TextFormatter (String lineToFormat)
  {  line = lineToFormat;  }


  /**
   * Finds the first single instance of str in line,
   * starting at the position start
   * @param str the string of length 1 to find.
   * Guaranteed to be length 1.
   * @param start the position to start searching.
   * Guaranteed to be in the string line.
   * @return the index of the first single instance of
   * str if the string is found or -1 if it is not found.
   */
  private int findString (String str, int start)
  {  /* To be implemented in part a) */ }


  /**
   * Count the number of times single instances of str appear in
   * the line.
   * @param str the string to find.
   * Guaranteed to be length 1.
   * @return the number of times the string appears in the line
   */
  private int countStrings (String str)
  {  /* To be implemented in part b) */ }


  /**
   * Replace all single instances of underscores in the line given by
   * line with italics tags.  There must be an even number of underscores
```

```
    * in the line, and they will be replaced by <I>, </I>, alternating.
    * @param original a string of length 1 to replace
    * @param replacement the string (of any length) use as a replacement
    * @return the line with single instances of underscores replaced with
    * <I> tags or the original line if there are not an even number of
    * underscores.
    */
   public String convertItalics ()
   {  /* To be implemented in part c) */ }
}
```

a) Write the method `findString`. This method will take a string of length 1 to find in the line, at a given starting point. It will return the location of the goal string. This differs from the `indexOf` method of the `String` class because it requires the string be just a single instance of the string, so if the string appears two or more times consecutively, it will not return any of those values. If there is no single instance, the method should return -1. Consider the following examples, where `line` has the value `"aabaccb"`.

Call	Result	Explanation
`findString("a", 0)`	3	The first single occurrence of `"a"` is in position 3. The `"a"`s in position 0 and 1 are not returned because they are not single instances.
`findString("b", 4)`	6	The first single occurrence of `"b"` at or after position 4 is in position 6.
`findString("c", 0)`	-1	There is no single instance of `"c"` in the line.

b) Write the method `countStrings`. This method will take a string of length 1 to find in the line. It will return the number of times single instances of the goal string appear in the string.

c) Write the method `convertItalics`. If the line has an even number of single underscores, a line with those underscores converted to alternating <I> and </I> tags should be returned. The first underscore will be converted to <I>, the second to </I>, the third to <I>, and so on. If the line does not have an even number of <I> tags, the original line should be returned. Consider the following examples.

line	value returned by convertItalics
This is _very_ good.	This is <I>very</I> good.
This is _very_ _good_.	<I>This</I> is <I>very</I> <I>good</I>.
This is _very good.	This is _very good.
This is __very good.	This is __very good.

AP® Computer Science A
Elevens Lab
Student Guide

The AP Program wishes to acknowledge and thank the following individuals for their contributions in developing this lab and the accompanying documentation.

Michael Clancy: University of California at Berkeley

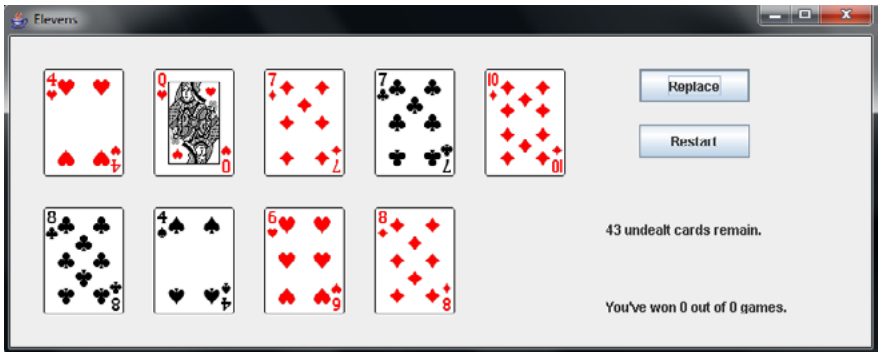
Robert Glen Martin: School for the Talented and Gifted in Dallas, TX

Judith Hromcik: School for the Talented and Gifted in Dallas, TX

Elevens Lab Student Guide

Introduction

The following activities are related to a simple solitaire game called Elevens. You will learn the rules of Elevens, and will be able to play it by using the supplied Graphical User Interface (GUI) shown at the right. You will learn about the design and the Object Oriented Principles that suggested that design. You will also implement much of the code.



43 undealt cards remain.

You've won 0 out of 0 games.

Table of Contents

Activity 1: Design and Create a `Card` Class

Introduction:

In this activity, you will complete a `Card` class that will be used to create card objects.

Think about card games you've played. What kinds of information do these games require a card object to "know"? What kinds of operations do these games require a card object to provide?

Exploration:

Now think about implementing a class to represent a playing card. What instance variables should it have? What methods should it provide? Discuss your ideas for this `Card` class with classmates.

Read the partial implementation of the `Card` class available in the **Activity1 Starter Code** folder. As you read through this class, you will notice the use of the `@Override` annotation before the `toString` method. The Java `@Override` annotation can be used to indicate that a method is intended to override a method in a superclass. In this example, the `Object` class's `toString` method is being overridden in the `Card` class. If the indicated method doesn't override a method, then the Java compiler will give an error message.

Here's a situation where this facility comes in handy. Programmers new to Java often encounter problems matching headings of overridden methods to the superclass's original method heading. For example, in the `Weight` class below, the `tostring` method is intended to be invoked when `toString` is called for a `Weight` object.

```
public class Weight {

   private int pounds;
   private int ounces;
        ...

   public String tostring(String str) {
      return this.pounds + " lb.  " + this.ounces + " oz.";
   }
        ...
}
```

Unfortunately, this doesn't work; the `tostring` method given above has a different name and a different signature from the `Object` class's `toString` method. The correct version below has the correct name `toString` and no parameter:

```
   public String toString() {
      return this.pounds + " lb.  " + this.ounces + " oz.";
   }
```

The `@Override` annotation would cause an error message for the first `tostring` version to alert the programmer of the errors.

Exercises:

1. Complete the implementation of the provided `Card` class. You will be required to complete:

   a. a constructor that takes two `String` parameters that represent the card's rank and suit, and an `int` parameter that represents the point value of the card;

   b. accessor methods for the card's rank, suit, and point value;

   c. a method to test equality between two card objects; and

   d. the `toString` method to create a `String` that contains the rank, suit, and point value of the card object. The string should be in the following format:

      rank `of` *suit* `(point value =` *pointValue*`)`

2. Once you have completed the `Card` class, find the `CardTester.java` file in the **Activity1 Starter Code** folder. Create three `Card` objects and test each method for each `Card` object.

Activity 2: Initial Design of a `Deck` Class

Introduction:

Think about a deck of cards. How would you describe a deck of cards? When you play card games, what kinds of operations do these games require a deck to provide?

Exploration:

Now consider implementing a class to represent a deck of cards. Describe its instance variables and methods, and discuss your design with a classmate.

Read the partial implementation of the `Deck` class available in the **Activity2 Starter Code** folder. This file contains the instance variables, constructor header, and method headers for a `Deck` class general enough to be useful for a variety of card games. Discuss the `Deck` class with your classmates; in particular, make sure you understand the role of each of the parameters to the `Deck` constructor, and of each of the private instance variables in the `Deck` class.

Exercises:

1. Complete the implementation of the `Deck` class by coding each of the following:

   - `Deck` constructor — This constructor receives three arrays as parameters. The arrays contain the ranks, suits, and point values for each card in the deck. The constructor creates an `ArrayList`, and then creates the specified cards and adds them to the list.
     For example, if `ranks = {"A", "B", "C"}`, `suits = {"Giraffes", "Lions"}`, and `values = {2,1,6}`, the constructor would create the following cards:

     ```
     ["A", "Giraffes", 2], ["B", "Giraffes", 1], ["C", "Giraffes", 6],
     ["A", "Lions", 2], ["B", "Lions", 1], ["C", "Lions", 6]
     ```

     and would add each of them to `cards`. The parameter `size` would then be set to the size of `cards`, which in this example is 6.

     Finally, the constructor should shuffle the deck by calling the `shuffle` method. Note that you will not be implementing the `shuffle` method until Activity 4.

   - `isEmpty` — This method should return `true` when the size of the deck is `0`; `false` otherwise.

   - `size` — This method returns the number of cards in the deck that are left to be dealt.

- `deal` — This method "deals" a card by removing a card from the deck and returning it, if there are any cards in the deck left to be dealt. It returns `null` if the deck is empty. There are several ways of accomplishing this task. Here are two possible algorithms:
  Algorithm 1: Because the cards are being held in an `ArrayList`, it would be easy to simply call the `List` method that removes an object at a specified index, and return that object. Removing the object from the end of the list would be more efficient than removing it from the beginning of the list. Note that the use of this algorithm also requires a separate "discard" list to keep track of the dealt cards. This is necessary so that the dealt cards can be reshuffled and dealt again.
  Algorithm 2: It would be more efficient to leave the cards in the list. Instead of removing the card, simply decrement the size instance variable and then return the card at `size`. In this algorithm, the `size` instance variable does double duty; it determines which card to "deal" and it also represents how many cards in the deck are left to be dealt. **This is the algorithm that you should implement.**

2. Once you have completed the `Deck` class, find `DeckTester.java` file in the **Activity2 Starter Code** folder. Add code in the `main` method to create three `Deck` objects and test each method for each `Deck` object.

Questions:

1. Explain in your own words the relationship between a `deck` and a `card`.

2. Consider the deck initialized with the statements below. How many cards does the deck contain?

```
String[] ranks = {"jack", "queen", "king"};
String[] suits = {"blue", "red"};
int[] pointValues = {11, 12, 13};
Deck d = new Deck(ranks, suits, pointValues);
```

3. The game of Twenty-One is played with a deck of 52 cards. Ranks run from ace (highest) down to 2 (lowest). Suits are spades, hearts, diamonds, and clubs as in many other games. A face card has point value 10; an ace has point value 11; point values for 2, …, 10 are 2, …, 10, respectively. Specify the contents of the `ranks`, `suits`, and `pointValues` arrays so that the statement

```
Deck d = new Deck(ranks, suits, pointValues);
```

initializes a deck for a Twenty-One game.

4. Does the order of elements of the `ranks`, `suits`, and `pointValues` arrays matter?

Activity 3: Shuffling the Cards in a Deck

Introduction:

Think about how you shuffle a deck of cards by hand. How well do you think it randomizes the cards in the deck?

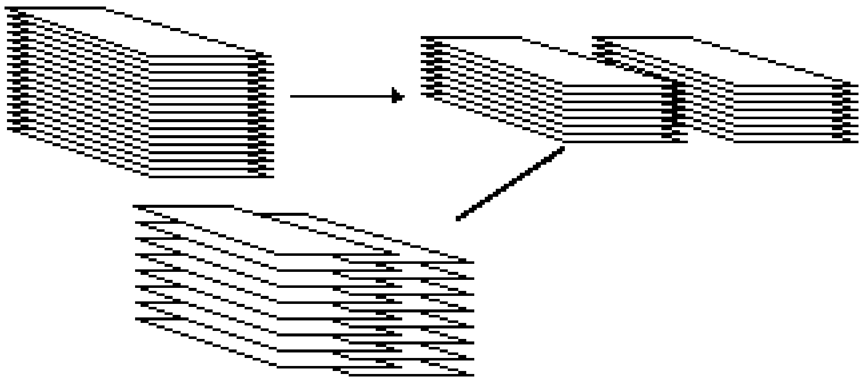
Exploration:

We now consider the *shuffling* of a deck, that is, the *permutation* of its cards into a random-looking sequence. A requirement of the shuffling procedure is that any particular permutation has just as much chance of occurring as any other. We will be using the `Math.random` method to generate random numbers to produce these permutations.

Several ideas for designing a shuffling method come to mind. We will consider two:

Perfect Shuffle

Card players often shuffle by splitting the deck in half and then interleaving the two half-decks, as shown below.



This procedure is called a *perfect shuffle* if the interleaving alternates between the two half-decks. Unfortunately, the perfect shuffle comes nowhere near generating all possible deck permutations. In fact, eight shuffles of a 52-card deck return the deck to its original state!

Consider the following "perfect shuffle" algorithm that starts with an array named `cards` that contains 52 cards and creates an array named `shuffled`.

```
Initialize shuffled to contain 52 "empty" elements.
Set k to 0.
For j = 0 to 25,
    – Copy cards[j] to shuffled[k];
    – Set k to k+2.
Set k to 1.
For j = 26 to 51,
    – Copy cards[j] to shuffled[k];
    – Set k to k+2.
```

This approach moves the first half of `cards` to the even index positions of `shuffled`, and it moves the second half of `cards` to the odd index positions of `shuffled`.

The above algorithm shuffles 52 cards. If an odd number of cards is shuffled, the array `shuffled` has one more even-indexed position than odd-indexed positions. Therefore, the first loop must copy one more card than the second loop does. This requires rounding up when calculating the index of the middle of the deck. In other words, in the first loop `j` must go up to `(cards.length + 1) / 2`, exclusive, and in the second loop `j` most begin at `(cards.length + 1) / 2`.

Selection Shuffle

Consider the following algorithm that starts with an array named `cards` that contains 52 cards and creates an array named `shuffled`. We will call this algorithm the "selection shuffle."

```
Initialize shuffled to contain 52 "empty" elements.
Then for k = 0 to 51,
    – Repeatedly generate a random integer j between 0 and 51, inclusive until
      cards[j] contains a card (not marked as empty);
    – Copy cards[j] to shuffled[k];
    – Set cards[j] to empty.
```

This approach finds a suitable card for the kth position of the deck. Unsuitable candidates are any cards that have already been placed in the deck.

While this is a more promising approach than the perfect shuffle, its big defect is that it runs too slowly. Every time an empty element is selected, it has to loop again. To determine the last element of `shuffled` requires an average of 52 calls to the random number generator.

A better version, the "efficient selection shuffle," works as follows:

```
For k = 51 down to 1,
    – Generate a random integer r between 0 and k, inclusive;
    – Exchange cards[k] and cards[r].
```

This has the same structure as selection sort:

```
For k = 51 down to 1,
    – Find r, the position of the largest value among cards[0] through cards[k];
    – Exchange cards[k] and cards[r].
```

The selection shuffle algorithm does not require to a loop to find the largest (or smallest) value to swap, so it works quickly.

Exercises:

1. Use the file `Shuffler.java`, found in the **Activity3 Starter Code**, to implement the perfect shuffle and the efficient selection shuffle methods as described in the **Exploration** section of this activity. You will be shuffling arrays of integers.

2. `Shuffler.java` also provides a `main` method that calls the shuffling methods. Execute the `main` method and inspect the output to see how well each shuffle method actually randomizes the array elements. You should execute `main` with different values of `SHUFFLE_COUNT` and `VALUE_COUNT`.

Questions:

1. Write a static method named `flip` that simulates a flip of a weighted coin by returning either `"heads"` or `"tails"` each time it is called. The coin is twice as likely to turn up heads as tails. Thus, `flip` should return `"heads"` about twice as often as it returns `"tails."`

2. Write a static method named `arePermutations` that, given two `int` arrays of the same length but with no duplicate elements, returns `true` if one array is a permutation of the other (i.e., the arrays differ only in how their contents are arranged). Otherwise, it should return `false`.

3. Suppose that the initial contents of the `values` array in `Shuffler.java` are `{1, 2, 3, 4}`. For what sequence of random integers would the efficient selection shuffle change `values` to contain `{4, 3, 2, 1}`?

Activity 4: Adding a `Shuffle` Method to the `Deck` Class

Introduction:

You implemented a `Deck` class in Activity 2. This class should be complete except for the `shuffle` method. You also implemented a `DeckTester` class that you used to test your incomplete `Deck` class.

In Activity 3, you implemented methods in the `Shuffler` class, which shuffled integers.

Now you will use what you learned about shuffling in Activity 3 to implement the `Deck shuffle` method.

Exercises:

1. The file `Deck.java`, found in the **Activity4 Starter Code** folder, is a correct solution from Activity 2. Complete the `Deck` class by implementing the `shuffle` method. Use the efficient selection shuffle algorithm from Activity 3.

   Note that the `Deck` constructor creates the deck and then calls the `shuffle` method. The `shuffle` method also needs to reset the value of `size` to indicate that all of the cards can be dealt again.

2. The `DeckTester.java` file, found in the **Activity4 Starter Code** folder, provides a basic set of `Deck` tests. It is similar to the `DeckTester` class you might have written in Activity 2. Add additional code at the bottom of the `main` method to create a standard deck of 52 cards and test the `shuffle` method. You can use the `Deck toString` method to "see" the cards after every shuffle.

Activity 5: Testing with Assertions (Optional)

Introduction:

In the previous activities, you were asked to design and implement the `Card` and `Deck` classes. Upon completion of those tasks, you were instructed to test those classes by creating objects and testing each of the class' methods. In this activity, we will take a more formal approach to testing and introduce the Java `assert` statement.

Exploration:

What is the purpose of testing? Programmers make mistakes. These range from misunderstanding an algorithm or a problem specification (the most serious), to simple typing errors. The purpose of program testing is to find as many of those mistakes as possible. Let's consider some aspects of testing.

Efficient and organized testing

How should we test a program? Clearly, we need to run the program and see whether its behavior matches what is intended. There's more to it than that, though. Good testing is *systematic*. A programmer should pick test cases not at random but in a way more likely to find errors. For example, one should choose test runs that collectively exercise all parts of a program. Code that isn't executed may contain bugs. Good testing is also *programmer-efficient*. Tests should be easy to run. Test cases should be chosen to be as simple as possible while still being complex enough to expose errors. Simple test cases can also make it easier to see what a program is doing wrong.

For this activity, we will focus on finding two kinds of errors:

- Inconsistencies, where two parts of the program have different expectations about a variable's value, for example; and
- Common "typos of the brain," such as off-by-one errors and substitution of one operator for another (such as using `>` instead of `<`).

Our tests will all involve *assertions*, `boolean` expressions that should be `true` if the program is running correctly. We will incorporate our tests in a "tester" class whose `main` method executes the tests.

Assertions in Java

Our testing code will use the Java `assert` statement. This statement has the following form:

`assert` *booleanExpression* `:` *StringExpression*`;`

If the value of *booleanExpression* is `true`, the program continues with the next statement. If the value of *booleanExpression* is `false`, the program throws an `AssertionError` exception and prints *StringExpression*. It also prints a *Stack Trace* (more on that later). Here's an example using `assert`.

```
Card c1 = new Card("ace", "hearts", 1);
Card c2 = new Card("ace", "hearts", 1);
assert c1.matches(c2) : "Duplicate cards do not match.";
```

This code creates two new `Card` objects, and then checks that they contain the same information. If `c1.matches(c2)` returns `true`, then execution continues with the next statement. However, if the program has an error which results in `c1.matches(c2)` returning `false`, an `AssertionError` is thrown and the message "Duplicate cards do not match." is output.

Assertions are *disabled* by default. To use them, the command-line option `-ea` (Enable Assertions) is used. For example, the following would run the main method in `CardTester` with assertions enabled:

```
java -ea CardTester
```

Organizing tests of the `Card` class

We move on to test the `Card` class. Cards have a constructor and several methods (`suit`, `rank`, `pointValue`, `matches`, and `toString`). Our tests must cover all of these methods.

First, we create a file named `CardTester.java`. This name was chosen to describe its purpose. Its `main` method will start by creating some `Card` objects that will be used to do the testing:

```
Card c1 = new Card("ace", "hearts", 1);
Card c2 = new Card("ace", "hearts", 1);
Card c3 = new Card("ace", "hearts", 2);
Card c4 = new Card("ace", "spades", 1);
Card c5 = new Card("king", "hearts", 1);
Card c6 = new Card("queen", "clubs", 3);
```

The first two cards are identical. Cards `c3`, `c4`, and `c5` each differ from `c1` in only one of the instance variable values. These "one difference" cards will help us find copy/paste errors; for example, if the body of `suit` was copied from `rank` and pasted without change. The last card is different from the others in all of the values.

We start by testing the `Card` accessor methods. These tests merely check, using cards with completely different information, that what's stored is what was provided in the constructor. Note the inclusion in the `String` message of information about which value was involved in each assertion.

```
assert c1.rank().equals("ace") : "Wrong rank: " + c1.rank();
assert c1.suit().equals("hearts") : "Wrong suit: " + c1.suit();
assert c1.pointValue() == 1 : "Wrong point value: "
   + c1.pointValue();
assert c6.rank().equals("queen") : "Wrong rank: " + c6.rank();
assert c6.suit().equals("clubs") : "Wrong suit: " + c6.suit();
assert c6.pointValue() == 3: "Wrong point value : "
   + c6.pointValue();
```

Next, we test the `Card` method `matches`. Two cards match if and only if they have the same rank, suit, and point values. A likely implementation of `matches` will involve some comparisons and some uses of `&&`. Common bugs are the copy/paste error mentioned above and the substitution of `||` for `&&`. Comparing `c1` to all the others should reveal these kinds of errors.

```
assert c1.matches(c1) : "Card doesn't match itself: " + c1;
assert c1.matches(c2) : "Duplicate cards aren't equal: " + c1;
assert !c1.matches(c3)
   : "Different cards are equal: " + c1 + ", " + c3;
assert !c1.matches(c4)
   : "Different cards are equal: " + c1 + ", " + c4;
assert !c1.matches(c5)
   : "Different cards are equal: " + c1 + ", " + c5;
assert !c1.matches(c6)
   : "Different cards are equal: " + c1 + ", " + c6;
```

Finally, we test `toString`, again on two completely different objects.

```
assert c1.toString().equals("ace of hearts (point value = 1)")
   : "Wrong toString: " + c1;
assert c6.toString().equals("queen of clubs (point value = 3)")
   : "Wrong toString: " + c6;
```

If all of the tests pass, we provide a message that says so:

```
System.out.println("All tests passed!");
```

Systematic testing

Cards didn't involve any data structures more complicated than strings. When testing a class with more complex structures, it makes sense to start small. With a `Deck` class, for example, it might make sense to first provide tests that use a 1-card deck, and then a 2-card deck with different cards.

But, would all these tests get out of control? Just as in other programming you've done, it makes sense to split a long sequence of statements into "helper" methods. The result may be a smaller test program, and some of the assertion sequences might be easier to reuse. The `main` method in a `DeckTester` class might be the following:

```
public static void main(String[] args) {
   test1CardDeck();
   test2CardDeck();
   testShuffle();
   System.out.println("All tests passed!");
}
```

Exercises:

1. The folder **Activity5 Starter Code** contains the four subfolders **Buggy1**, **Buggy2**, **Buggy3**, and **Buggy4**. Each of these contains a different buggy version of the `Deck` class. These buggy decks have been precompiled; only the `Deck.class` bytecode file is included. These buggy versions each contain one error caused by either moving a statement, or substituting one symbol for another, e.g., `1` for `0` or `>` for `<`. Test each of them with the `DeckTester` application provided in each folder:

   ```
   java -ea DeckTester
   ```

   If you are using a Windows-based system, you can just execute the provided **DeckTester.bat** file.

   Each of the four different `DeckTester` runs will produce an `AssertionError` exception, along with information about why the error occurred. For each error that occurs, write down which method or constructor of the buggy `Deck` class could contain the bug, and make an educated guess about the cause of the error. You might find it helpful to refer to your completed `Deck` class from Activity 4.

   Note: The Buggy1 test will produce output similar to the following:

   ```
   ... >java -ea DeckTester
   Exception in thread "main" java.lang.AssertionError: isEmpty is
   false for an empty deck.
      at DeckTester.testEmpty(DeckTester.java:98)
      at DeckTester.test1CardDeck(DeckTester.java:28)
      at DeckTester.main(DeckTester.java:12)
   ```

The last three lines of output are a stack trace that tells you that the

- `AssertionError` occurred in the `testEmpty` method at line 98.
- `testEmpty` method was called from the `test1CardDeck` method at line 28.
- `test1CardDeck` method was called from the `main` method at line 12.

Record your conclusions below:

Buggy1:

Constructor or Method (write method name):

Describe a Possible Code Error:

____

____

____

Buggy2:

Constructor or Method (write method name):

Describe a Possible Code Error:

____

____

____

Buggy3:

Constructor or Method (write method name):

Describe a Possible Code Error:

____

____

____

Buggy4:

Constructor or Method (write method name):

Describe a Possible Code Error:

____

____

____

2. Now, examine the Buggy5 folder. This folder contains a `Deck.java` file with multiple errors. Use `DeckTester` to help you find the errors. Correct each error until the `Deck` class has passed all of its tests.

   Note that you may receive a runtime error other than `AssertionError` when running `DeckTester`. If so, you may find it helpful to switch the order of 1-card deck and 2-card deck tests as follows:

   ```
   public static void main(String[] args) {
      test2CardDeck();   // order swapped
      test1CardDeck();   // order swapped
      testShuffle();
      System.out.println("All tests passed!");
   }
   ```

Activity 6: Playing Elevens

Introduction:

In this activity, the game Elevens will be explained, and you will play an interactive version of the game.

Exploration:

The solitaire game of Elevens uses a deck of 52 cards, with ranks A (ace), 2, 3, 4, 5, 6, 7, 8, 9, 10, J (jack), Q (queen), and K (king), and suits ♣ (clubs), ♦ (diamonds), ♥ (hearts), and ♠ (spades). Here is how it is played.

1. The deck is shuffled, and nine cards are dealt "face up" from the deck to the board.
2. Then the following sequence of steps is repeated:
   a. The player removes each pair of cards (A, 2, … , 10) that total 11, e.g., an 8 and a 3, or a 10 and an A. An ace is worth 1, and suits are ignored when determining cards to remove.
   b. Any triplet consisting of a J, a Q, and a K is also removed by the player. Suits are also ignored when determining which cards to remove.
   c. Cards are dealt from the deck if possible to replace the cards just removed.

The game is won when the deck is empty and no cards remain on the table. Here's a sample game, in which underlined cards are replacements from the deck.

Cards on the Table	**Explanation**
K♠ 10♦ J♣ 2♣ 2♥ 9♦ 3♥ 5♠ 5♦	initial deal
K♠ 10♦ J♣ $\underline{\text{7♦}}$ 2♥ $\underline{\text{Q♠}}$ 3♥ 5♠ 5♦	remove 2♣ (either 2 would work) and 9♦
$\underline{\text{A♠}}$ 10♦ $\underline{\text{9♣}}$ 7♦ 2♥ $\underline{\text{7♣}}$ 3♥ 5♠ 5♦	remove J♣ Q♠ K♠
A♠ 10♦ $\underline{\text{10♠}}$ 7♦ $\underline{\text{3♣}}$ 7♣ 3♥ 5♠ 5♦	remove 9♣ and 2♥ (removing A♠ and 10♦ would have been legal here too)
$\underline{\text{2♠}}$ 10♦ $\underline{\text{9♠}}$ 7♦ 3♣ 7♣ 3♥ 5♠ 5♦	remove A♠ and 10♠ (10♦ could have been removed instead)
$\underline{\text{A♣}}$ 10♦ $\underline{\text{K♦}}$ 7♦ 3♣ 7♣ 3♥ 5♠ 5♦	remove 2♠ and 9♠
$\underline{\text{6♦}}$ $\underline{\text{K♣}}$ K♦ 7♦ 3♣ 7♣ 3♥ 5♠ 5♦	remove A♣ and 10♦

2♦ K♣ K♦ 7♦ 3♣ 7♣ 3♥ 5♠ Q♦	remove 6♦ and one of the 5s; no further plays are possible; game is lost.

An interactive GUI version of Elevens allows one to play by clicking card images and buttons rather than by handling actual cards. When `Elevens.jar` is run, the cards on the board are displayed in a window. Clicking on an unselected card selects it; clicking on a selected card unselects it. Clicking on the **Replace** button first checks that the selection is legal; if so, it does the removal and deals cards to fill the empty slots. Clicking on the **Restart** button restarts the game.

The folder **Activity6 Starter Code** contains the file `Elevens.jar` that, when executed, runs a GUI-based implementation. In a Windows environment, you may be able to run it by double-clicking on it. Otherwise you can run it with the command

```
    java -jar Elevens.jar
```

Play a few games of Elevens. How many did you win?

Questions:

1. List all possible plays for the board 5♠ 4♥ 2♦ 6♣ A♠ J♥ K♦ 5♣ 2♠

2. If the deck is empty and the board has three cards left, must they be J, Q, and K? Why or why not?

3. Does the game involve any strategy? That is, when more than one play is possible, does it matter which one is chosen? Briefly explain your answer.

Activity 7: Elevens Board Class Design

Introduction:

Now that the `Card` and `Deck` classes are completed, the next class to design is `ElevensBoard`. This class will contain the state (instance variables) and behavior (methods) necessary to play the game of Elevens.

Questions:

1. What items would be necessary if you were playing a game of Elevens at your desk (not on the computer)? List the private instance variables needed for the `ElevensBoard` class.

2. Write an algorithm that describes the actions necessary to play the Elevens game.

3. Now examine the partially implemented `ElevensBoard.java` file found in the **Activity7 Starter Code** directory. Does the `ElevensBoard` class contain all the state and behavior necessary to play the game?

4. `ElevensBoard.java` contains three helper methods. These helper methods are `private` because they are only called from the `ElevensBoard` class.

   a. Where is the `dealMyCards` method called in `ElevensBoard`?

   b. Which `public` methods should call the `containsPairSum11` and `containsJQK` methods?

   c. It's important to understand how the `cardIndexes` method works, and how the list that it returns is used. Suppose that `cards` contains the elements shown below. Trace the execution of the `cardIndexes` method to determine what list will be returned. Complete the diagram below by filling in the elements of the returned list, and by showing how those values index `cards`. Note that the returned list may have less than 9 elements.

	0	1	2	3	4	5	6	7	8
`cards ->`	J♥	6♣	`null`	2♠	`null`	`null`	A♠	4♥	`null`

	0	1	2	3	4	5	6	7	8
`returned -> list`									

d. Complete the following `printCards` method to print all of the elements of cards that are indexed by `cIndexes`.

```
public static printCards(ElevensBoard board) {
   List<Integer> cIndexes = board.cardIndexes();

   /* Your code goes here. */









}
```

e. Which one of the methods that you identified in question 4b above needs to call the `cardIndexes` method before calling the `containsPairSum11` and `containsJQK` methods? Why?

Activity 8: Using an Abstract Board Class

Introduction:

The Elevens game belongs to a set of related solitaire games. In this activity you will learn about some of these related games. Then you will see how inheritance can be used to reuse the code that is common to all of these games without rewriting it.

Exploration: Related Games

Thirteens

A game related to Elevens, called *Thirteens*, uses a 10-card board. Ace, 2, … , 10, jack, queen correspond to the point values of 1, 2, …, 10, 11, 12. Pairs of cards whose point values add up to 13 are selected and removed. Kings are selected and removed singly. Chances of winning are claimed to be about 1 out of 2.

Tens

Another relative of Elevens, called *Tens*, uses a 13-card board. Pairs of cards whose point values add to 10 are selected and removed, as are quartets of kings, queens, jacks, and tens, all of the same rank (for example, K♠, K♥, K♦, and K♣). Chances of winning are claimed to be about 1 in 8 games.

Exploration: Abstract Classes

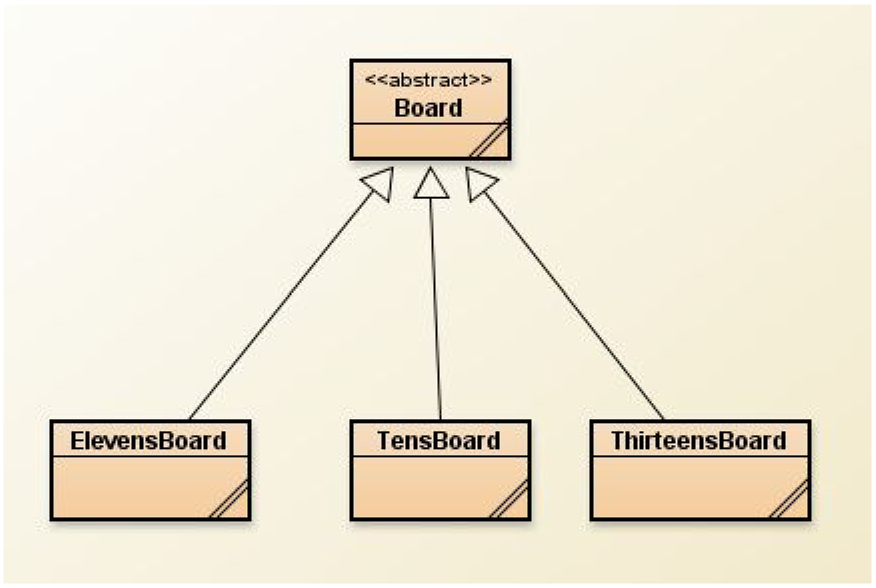
In reading the descriptions of Elevens and its related games, it is evident that these games share common state and behaviors. Each game requires:

- State (instance variables) — a deck of cards and the cards "on the" board.
- Behavior (methods) — to deal the cards, to remove and replace selected cards, to check for a win, to check if selected cards satisfy the rules of the game, to see if there are more legal selections available, and so on.

With all of this state and behavior in common, it would seem that inheritance could allow us to write code once and reuse it, instead of having to copy it for each different game.

But how? If we use the "IS-A" test, a `ThirteensBoard` "IS-A" `ElevensBoard` is not true. They have a lot in common, but an inheritance relationship between the two does not exist. So how do we create an inheritance hierarchy to take advantage of the commonalities between these two related boards?

The answer is to use a common superclass. Take all the state and behavior that these boards have in common and put them into a new `Board` class. Then have `ElevensBoard`, `TensBoard`, and `ThirteensBoard` inherit from the `Board` class. This makes sense because each of them is just a different kind of board. An `ElevensBoard` "IS-A" `Board`, a `ThirteensBoard` "IS-A" `Board`, and a `TensBoard` "IS-A" `Board`. A diagram that shows the inheritance relationships of these classes is included below. Note that `Board` is shown as abstract. We'll discuss why later.



Let's see how this works out for dividing up our original `ElevensBoard` code from Activity 7. Because all these games need a deck and the cards on the board, all of the instance variables can go into `Board`. Some methods, like `deal`, will work the same for every game, so they should be in `Board` too. Methods like `containsJQK` are Elevens-specific and should be in `ElevensBoard`. So far, so good.

But what should we do with the `isLegal` and `anotherPlayIsPossible` methods? Every Elevens-related game will have both of these methods, but they need to work differently for each different game. That's exactly why Java has `abstract` methods. Because each of these games needs `isLegal` and `anotherPlayIsPossible` methods, we include those methods in `Board`. However, because the implementation of these methods depends on the specific game, we make them `abstract` in `Board` and don't include their implementations there. Also, because `Board` now contains `abstract` methods, it must also be specified as `abstract`. Finally, we override each of these `abstract` methods in the subclasses to implement their specific behavior for that game.

But if we have to implement `isLegal` and `anotherPlayIsPossible` in each game-specific board class, why do we need to have the `abstract` methods in `Board`? Consider a class the uses a board, such as the GUI program you used in Activity 6. Such a class is called a *client* of the `Board` class.

The GUI program does not actually need to know what kind of a game it is displaying! It only knows that

the board that was provided "IS-A" `Board`, and it only "knows" about the methods in the `Board` class. The GUI program is only able to call `isLegal` and `anotherPlayIsPossible` because they are included in `Board`.

Finally, we need to understand how the GUI program is able to execute the correct `isLegal` and `anotherPlayIsPossible` methods. When the GUI program starts, it is provided an object of a class that inherits from `Board`. If you want to play Elevens, you provide an `ElevensBoard` object. If you want to play Tens, you provide a `TensBoard` object. So, when the GUI program uses that object to call `isLegal` or `anotherPlayIsPossible`, it automatically uses the method implementation included in that particular object. This is known as *polymorphism*.

Questions:

1. Discuss the similarities and differences between *Elevens*, *Thirteens*, and *Tens*.

2. As discussed previously, all of the instance variables are declared in the `Board` class. But it is the `ElevensBoard` class that "knows" the board size, and the ranks, suits, and point values of the cards in the deck. How do the `Board` instance variables get initialized with the `ElevensBoard` values? What is the exact mechanism?

3. Now examine the files `Board.java`, and `ElevensBoard.java`, found in the **Activity8 Starter Code** directory. Identify the `abstract` methods in `Board.java`. See how these methods are implemented in `ElevensBoard`. Do they cover all the differences between *Elevens*, *Thirteens*, and *Tens* as discussed in question 1? Why or why not?

Activity 9: Implementing the Elevens Board

Introduction:

In Activity 8, we *refactored* (reorganized) the original `ElevensBoard` class into a new `Board` class and a much smaller `ElevensBoard` class. The purpose of this change was to allow code reuse in new games such as Tens and Thirteens. Now you will complete the implementation of the methods in the refactored `ElevensBoard` class.

Exercises:

1. Complete the `ElevensBoard` class in the **Activity9 Starter Code** folder, implementing the following methods.

   Abstract methods from the `Board` class:

   a. `isLegal` — This method is described in the method heading and related comments below. The implementation should check the number of cards selected and utilize the `ElevensBoard` helper methods.

```
/**
 * Determines if the selected cards form a valid group for removal.
 * In Elevens, the legal groups are (1) a pair of non-face cards
 * whose values add to 11, and (2) a group of three cards consisting of
 * a jack, a queen, and a king in some order.
 * @param selectedCards the list of the indexes of the selected cards.
 * @return true if the selected cards form a valid group for removal;
 *         false otherwise.
 */
@Override
public boolean isLegal(List<Integer> selectedCards)
```

b. `anotherPlayIsPossible` — This method should also utilize the helper methods. It should be very short.

```
/**
 * Determine if there are any legal plays left on the board.
 * In Elevens, there is a legal play if the board contains
 * (1) a pair of non-face cards whose values add to 11, or (2) a group
 * of three cards consisting of a jack, a queen, and a king in some order.
 * @return true if there is a legal play left on the board;
 *         false otherwise.
 */
@Override
public boolean anotherPlayIsPossible()
```

`ElevensBoard` helper methods:

c. `containsPairSum11` — This method determines if the selected elements of `cards` contain a pair of cards whose point values add to 11.

```
/**
 * Check for an 11-pair in the selected cards.
 * @param selectedCards selects a subset of this board.  It is this list
 *                      of indexes into this board that are searched
 *                      to find an 11-pair.
 * @return true if the board entries indexed in selectedCards
 *              contain an 11-pair; false otherwise.
 */
private boolean containsPairSum11(List<Integer> selectedCards)
```

d. `containsJQK` — This method determines if the selected elements of `cards` contains a jack, a queen, and a king in some order.

```
/**
 * Check for a JQK in the selected cards.
 * @param selectedCards selects a subset of this board.  It is this list
 *                      of indexes into this board that are searched
 *                      to find a JQK-triplet.
 * @return true if the board entries indexed in selectedCards
 *              include a jack, a queen, and a king; false otherwise.
 */
private boolean containsJQK(List<Integer> selectedCards)
```

When you have completed these methods, run the `main` method found in `ElevensGUIRunner.java`. Make sure that the Elevens game works correctly. Note that the cards directory must be in the same directory with your `.class` files.

Questions:

1. The size of the board is one of the differences between *Elevens* and *Thirteens*. Why is `size` not an abstract method?

2. Why are there no abstract methods dealing with the selection of the cards to be removed or replaced in the array `cards`?

3. Another way to create "IS-A" relationships is by implementing interfaces. Suppose that instead of creating an `abstract Board` class, we created the following `Board` interface, and had `ElevensBoard` implement it. Would this new scheme allow the Elevens GUI to call `isLegal` and `anotherPlayIsPossible` polymorphically? Would this alternate design work as well as the `abstract Board` class design? Why or why not?

```
public interface Board
{
   boolean isLegal(List<Integer> selectedCards);

   boolean anotherPlayIsPossible();
}
```

Activity 10: ThirteensBoard (Optional)

Introduction:

The purpose of this activity is to create the Thirteens game using the knowledge gained from implementing the Elevens game.

Exploration:

The rules for the Thirteens game are repeated below:

Thirteens

A game related to Elevens, called Thirteens, uses a 10-card board. Ace, 2, … , 10, jack, queen correspond to the point values of 1, 2, …, 10, 11, 12. Pairs of cards whose point values add up to 13 are selected and removed. Kings are selected and removed singly. Chances of winning are claimed to be about 1 out of 2.

Exercises:

1. The **Activity10 Starter Code** folder contains all the code for a complete working Elevens game. Review the code in the `ElevensBoard` class. Identify the changes that would be necessary to implement the Thirteens game.

2. Copy and paste the `ElevensBoard.java` file into a new file, `ThirteensBoard.java`. Make the necessary changes to this file to implement the Thirteens game.

3. The **Activity10 Starter Code** folder also contains the `ElevensGUIRunner.java` file that is shown below. This program creates the board (an `ElevensBoard` object). Then it creates the GUI (a `CardGameGUI` object). Finally, it displays the GUI by calling its `displayGame` method. Review the code in the `ElevensGUIRunner` class as shown below. Identify the changes that would be necessary to implement the Thirteens game.

```
/**
 * This is a class that plays the GUI version of the Elevens game.
 * See accompanying documents for a description of how Elevens is played.
 */
public class ElevensGUIRunner {

   /**
    * Plays the GUI version of Elevens.
    * @param args is not used.
```

```
     */
    public static void main(String[] args) {
       Board board = new ElevensBoard();
       CardGameGUI gui = new CardGameGUI(board);
       gui.displayGame();
    }
 }
```

4. Copy and paste the `ElevensGUIRunner.java` file into a new file, `ThirteensGUIRunner.java`. Make the necessary changes to this file to implement the Thirteens game.

5. Run the `ThirteensGUIRunner` program and test your new Thirteens game.

Activity 11: Simulation of Elevens (Optional)

Introduction:

We have implemented two different solitaire games that we can play, Elevens and Thirteens. That is perfect if we want to entertain ourselves playing them. But what if we want to answer questions about the games? For example, what percentage of Elevens games can be won? You probably already have some idea about the answer to this question, but you might have to play thousands of games to have any real confidence in your answer. That's where *simulation* comes in.

Exploration:

A common computing application is the *simulation* of some process; in other words, writing a program that imitates the process in some way. The behaviors and state in the program represent key features of, and are said to provide a *model* for, the process. An example is a simulation of a household robot vacuum cleaner. Internally, the simulation program is keeping track of its environment, the battery level, and the amount of dust in its dust container, and is perhaps displaying them on a computer console. Program methods plot the robot's course through rooms in the "house" and determine when it is finished.

A simulation is useful when the process being simulated is too complicated, too slow, too dangerous, or too expensive to observe in the real world. Also, understanding the program helps one understand the process being simulated. For example, the producers of the robot vacuum cleaner would have used a simulation to debug its algorithms before starting to manufacture the actual robots.

A program is called *probabilistic* when its state change is affected by chance. One example is a traffic simulation, which has to account for cars unpredictably entering the traffic zone and driving at varying speeds. A more obvious example is a simulation of a game based on dice or spinners. In Elevens, the probabilistic element is the shuffling of the deck of cards.

To model random events, we use a *pseudo-random number generator* (the "pseudo" is usually omitted). In the `Deck shuffle` method, we used the `Math.random` method to generate our random numbers. This reliance on chance significantly complicates the task of verifying that a simulation is behaving correctly. The programmer needs to have a good idea of what output to expect. Also, a small number of outcomes of the chance events may produce misleading behavior. For example, four flips of a coin may produce all heads, but it would be a mistake to assume that this behavior would happen often. Ten thousand flips would produce the more reasonable outcome of around 50 percent heads and 50 percent tails. The typical probabilistic simulation involves a large number of calls to the random number generator to increase the likelihood that the output reflects expected behavior.

To simulate Elevens, we will need to model the "playing" of the game using program state and behavior. Let's see how the real world relates to the code:

STATE:

Real-world "data"	**Program data**
The deck of cards	A `List` of `Card` objects
The cards on the board	An array of `Card` objects

BEHAVIOR:

Real-world operations	**Program operations**
Locating cards to remove	Searching for specific groups of `Card` objects in the board (an 11-pair or a JQK-triplet)
Removing cards and replacing them	Removing `Card` objects from the board and replacing them with `Card` objects from the deck
Dealing a card	Removing a `Card` object from the deck

We have most of this code already written. We have already taken care of all the state requirements. The deck of cards is modeled by the `cards` list in the `Deck` class. And the cards on the board are represented by the `cards` array in the `Board` class. We have also written methods for most of the necessary behaviors. In fact, we only need to model the additional things you do when you are playing the game yourself!

In the exercises, you will use an `ElevensSimulation` class, which will play games of Elevens. It will need access to methods that mimic the actions you make when you play the game. What do you do when you play the game and why do you do it? Answer these questions:

- What do you do repeatedly to play a game?
- As you are scanning the cards on the board, what are you trying to find?
- Why do you decide to click on a group of cards?
- What happens when you click the **Replace** button?

We will model these behaviors with three new methods in the `ElevensBoard` class:

- `playIfPossible` — Looks for a legal play and makes the play (if found). This is the only new method that `ElevensSimulation` needs to call directly. We could put all of our new code into this method, but it's helpful to divide it up using two new `private` helper methods.
- `playPairSum11IfPossible` — Looks for an 11-pair and replaces it (if found).
- `playJQKIfPossible` — Looks for a JQK-triplet and replaces it (if found).

Next we consider the implementation of `playPairSum11IfPossible`. This method needs to first determine if the board contains an 11-pair. Then, if it finds one, it needs to remove it. Of course, `playPairSum11IfPossible` could call `containsPairSum11` to see if there is an 11-pair on the board. But `containsPairSum11` doesn't return any information about the indexes of the two cards that make up the 11-pair. So, `playPairSum11IfPossible` would have to find the pair again before removing it. To avoid having to find the pair twice, we would need to copy the code from `containsPairSum11` into `playPairSum11IfPossible`. Thankfully, there's a better way.

What if we change the `containsPairSum11` method into a `findPairSum11` method? In other words, instead of having a "contains" method that returns a `boolean` value, we have a "find" method that returns a list of the indexes of the two cards in the pair. If there is no 11-pair on the board, it will return an empty list. Now, `playPairSum11IfPossible` will be able to call `findPairSum11`, and if there is an 11-pair, it will already have the list of indexes needed to call the `replaceSelectedCards` method. This design eliminates both the duplicated code and the double work! We will need to make a similar modification to change the `containsJQK` method into a `findJQK` method for the `playJQKIfPossible` method to call.

Note that it's usually not possible to foresee everything during the initial design of a program. For example, in the GUI version of Elevens, there was no need for "find" methods. The person playing the game had that task. However, the "find" methods became useful when we got into the details of the simulation. So, program designs can and do change.

Of course, when we do an initial program design, we try to accommodate all the needs of the problem as we understand them. But we also try to keep the design flexible, so that we can accommodate future needs. One rule of program design is that methods should be `private`, unless there is a good reason for another class to call those methods. Because we initially made `containsPairSum11 private`, we know that no other class uses it. Therefore, it's safe to rename and change it.

Exercises:

1. First, examine the completed `ElevensSimulation` class in the **Activity11 Starter Code** folder. This simulation creates an `ElevensBoard` object, and uses it to play `GAMES_TO_PLAY` games of Elevens. Note that the only new `ElevensBoard` method used is `playIfPossible`.

2. Now make the necessary changes to the `ElevensBoard` class. Change `containsPairSum11` into `findPairSum11`. You will need to change both the method heading and the method body. Note that the method's comment block has already been changed for you.

3. Change the `isLegal` and `anotherPlayIsPossible` methods to use `findPairSum11` instead of `containsPairSum11`. Note that the board contains an 11-pair if and only if `findPairSum11(cIndexes).size() > 0`.

4. Change `containsJQK` to `findJQK` in a similar fashion to the `containsPairSum11` to `findPairSum11` conversion you did in exercise 2 above. Again, the method's comment block has already been changed for you.

5. Change the `isLegal` and `anotherPlayIsPossible` methods to use `findJQK` instead of `containsJQK`. At this point, the Elevens GUI program should work just as it did before.

6. It's time to complete the `ElevensBoard` methods required by the `ElevensSimulation` class. First complete the `public playIfPossible` method, which is called from the `ElevensSimulation` class. This method will use the `private playPairSum11IfPossible` and `playJQKIfPossible` helper methods. Note that you will have to replace the `return` statement in the stubbed out method.

7. Complete the `private playPairSum11IfPossible` and `playJQKIfPossible` helper methods. Note that you will have to replace the `return` statements in the stubbed out methods.

8. Now it's time to test your simulation-related changes. Make sure that `GAMES_TO_PLAY` and `I_AM_DEBUGGING` are initialized to `1` and `true` respectively in `ElevensSimulation`. Also make sure that `I_AM_DEBUGGING` is initialized to `true` in `ElevensBoard`. Run the `ElevensSimulation` program a few times and examine the output. You should be able to see both 11-pairs and JQK-triplets being correctly identified and removed.

Questions:

1. Set the `I_AM_DEBUGGING` flags to `false` and `GAMES_TO_PLAY` to `10`. Run the `ElevensSimulation` program a few times and record the percentage of games won for each run. What is the range of win percentages that you saw? Were the percentages fairly consistent, or did they vary quite a bit?

2. Increase the number of games to play to `100`. Are the win percentages more consistent from run to run?

3. Experiment with simulating different numbers of games. How many games do you need to play in order to get consistent results from run to run?

4. Optional — Repeat the above steps for the Thirteens game.

Glossary

assertion: Boolean expressions that should be true if the program is running correctly. The Java `assert` statement can be used to check assertions in a program.

class invariant: A logical statement relating to the values of the instance variables of a class that is always true between calls to the class's methods (also referred to as a "data invariant"). ("Invariant" means "not varying" or "not changing.")

client class: A class that uses another class (e.g., The `Deck` class is a client of the `Card` class.).

helper method: A method, usually `private`, that is called by another method. Helper methods are used to simplify the calling method. They also facilitate code reuse when they provide a function that can be used by more than one calling method.

loop invariant: A logical statement that is always true when execution reaches a loop's termination test.

model: A class with behaviors and state that represent key features of some "real-world" object or process. We say that a class models the "real-world" object. For example, the `Deck` class models a real deck of cards.

perfect shuffle: A card-shuffling method that starts with dividing the deck into two stacks, then interleaving the cards, first a card from stack 1, then a card from stack 2, then another card from stack 1, another from stack 2, and so on.

permutation: A rearrangement of a given sequence of values. There are six permutations of the sequence [1,2,3], namely [1,2,3] (the "identity" permutation), [1,3,2], [2,1,3], [2,3,1], [3,1,2], and [3,2,1]. If the given sequence contains duplicate values, so will its permutations. For example, the permutations of [1,1,2] are [1,1,2], [1,2,1], and [2,1,1].

polymorphism: A process that Java uses where the method to execute is based on the object executing the method. For example, if `board.anotherPlayIsPossible()` is executed, and `board` references an `ElevensBoard` object, then the `ElevensBoard` `anotherPlayIsPossible` method will be called.

probabilistic: Based on chance or involving the use of randomness.

pseudo-random number generator: A procedure that produces a sequence of values that passes various statistical tests for randomness (e.g., any value is just as likely to occur in a given position in the sequence as any other).

random number generator: See **pseudo-random number generator**.

refactor: Reorganizing code. One example of refactoring is creating helper methods to simplify code or eliminate duplicate code. Another is splitting a class into a superclass and a subclass, putting the code that would be common to other subclasses into the new superclass.

selection shuffle: A card-shuffling method that works similarly to the selection sort. It randomly selects a card for each position in the deck from the remaining unselected cards.

shuffle: A method of permuting (mixing up) the cards in a deck. See **perfect shuffle** and **selection shuffle**.

simulation: Imitation, using a computer program, of some real-world process. The "actors" in the process correspond to objects and variables in the simulation, while the interactions between the actors correspond to program methods.

systematic: Performed using a logical step-by-step process.

truncation: Removal of the fractional part of a real or double value, producing an integer.

References

The Complete Book of Solitaire and Patience Games, by Albert H. Morehead and Geoffrey Mott-Smith, Bantam Books (1977).

Sample Exam Questions

Multiple-Choice Questions

1. For a single winning game, how many iterations are completed by the inner `while` loop in the `ElevensSimulation` class?

   A. 22
   B. 23
   C. 24
   D. 26
   E. Different winning games will require differing iteration counts.

2. Which of the following code segments correctly stores in `int` variables `r1` and `r2` random unequal integers, each of which can be between `0` and `n-1`, inclusive?

   A.
   ```
   r1 = (int) (Math.random() * n - 1);
   r2 = (int) (Math.random() * n);
   ```

   B.
   ```
   r1 = (int) (Math.random() * (n - 1));
   r2 = (int) (Math.random() * (n - 1));
   if (r1 == r2) {
      r2 = (int) (Math.random() * (n - 1));
   }
   ```

   C.
   ```
   r1 = (int) (Math.random() * n);
   r2 = (int) (Math.random() * n);
   if (r1 == r2) {
      r2 = (int) (Math.random() * n);
   }
   ```

   D.
   ```
   r1 = (int) (Math.random() * (n - 1));
   r2 = (int) (Math.random() * n);
   while (r1 == r2) {
      r2 = (int) (Math.random() * n);
   }
   ```

   E.
   ```
   r1 = (int) (Math.random() * n);
   r2 = (int) (Math.random() * n);
   while (r1 == r2) {
      r2 = (int) (Math.random() * n);
   }
   ```

3. Consider the first line of the `ElevensBoard` class declaration below.

```
public class ElevensBoard extends Board
```

Which of the following methods must be implemented as a result of the way the `ElevensBoard` class has been declared?

I. `toString`
II. `anotherPlayIsPossible`
III. `isLegal`
IV. `dealMyCards`

A. I only
B. I, II, and IV only
C. II and III only
D. I and IV only
E. I, II, III, and IV

4. Consider the following declarations where … is a valid `Board` constructor parameter list.

I. `Board board = new Board(…);`
II. `Board board = new ElevensBoard();`
III. `ElevensBoard board = new ElevensBoard();`

Which of these declarations is(are) legal?

A. I only
B. I and II only
C. I and III only
D. II and III only
E. I, II, and III

Free-Response Questions

1. Consider the partial implementation of the `Deck` class as shown below:

```
public class Deck {

   /** cards contains all the cards in the deck.
    */
   private List<Card> cards;

   /** size is the number of not-yet-dealt cards.
    *  Cards are dealt from the top (highest index) down.
    *  The next card to be dealt is at size - 1.
    */
   private int size;

   /** Determines if this deck is empty (no undealt cards).
    *  @return true if this deck is empty, false otherwise.
    */
   public boolean isEmpty() {
      return size == 0;
   }

   /** Accesses the number of undealt cards in this deck.
    *  @return the number of undealt cards in this deck.
    */
   public int size() {
      return size;
   }

   /** Randomly permute the given collection of cards
    *  and reset the size to represent the entire deck.
    */
   public void shuffle() {
      for (int k = cards.size() - 1; k > 0; k--) {
         int howMany = k + 1;
         int start = 0;
         int randPos = (int) (Math.random() * howMany) + start;
         Card temp = cards.get(k);
         cards.set(k, cards.get(randPos));
         cards.set(randPos, temp);
      }
      size = cards.size();
   }

   /** Deals a card from this deck.
    *  @return the card just dealt, or null if all the cards have been
    *          previously dealt.
    */
   public Card deal() {
      if (isEmpty()) {
         return null;
      }
      size--;
      Card c = cards.get(size);
      return c;
   }

   // Constructor and other methods not shown.

}
```

(a) The current `shuffle` method uses the following algorithm:

- `for k = cards.size() - 1` down to `1`
  - Generate a random number `r` between `0` and `k`, inclusive
  - Swap the card found at position `k` in `cards` with card found at position `r` in `cards`
- Set `size` to `cards.size()`

Consider a different algorithm that uses a temporary `ArrayList` and that will do the following:

- Set `size` to `cards.size()`
- Create a temporary `ArrayList`
- Do this `size` number of times
  - Generate a random number `r` that can index any card in `cards`
  - Remove the card found at position `r` in `cards` and add it to the end of the temporary `ArrayList`
- Set `cards` to the temporary `ArrayList`

Complete the `shuffle` method using this new algorithm.

```
public void shuffle() {
```

(b) Compare the two versions of the `shuffle` method and answer the following question. Which `shuffle` method is more efficient, the `Deck` class's original `shuffle` method or the `shuffle` method described in part (a)? **Justify your answer**.

(c) Another way to handle shuffling the deck is to remove the `shuffle` method from the `Deck` class and give the responsibility to the `deal` method.

In this algorithm, the `deal` method performs a single card "just in time" shuffle immediately before dealing a card. This algorithm will do the following:

- If the deck is empty, return `null`
- Randomly generate a number that can index a card in the portion of the deck that contains cards that have not been dealt
- Decrement `size`
- Swap the selected card with the card found at position `size`
- Return the selected card

For example, if `cards` and `size` start out like this:

`cards:`

0	1	2	3	4	5	6
5♠	4♥	2♦	6♣	A♠	J♥	K♦

`size: 7`

and the number 3 is randomly generated for the index. After the changes, `cards` and `size` would be as follows:

`cards:`

0	1	2	3	4	5	6
5♠	4♥	2♦	K♦	A♠	J♥	6♣

`size: 6`

and the 6♣ would be returned and is now in the dealt portion of the deck.

The number 1 is the next randomly generated index. After the changes, `cards` and `size` would be as follows:

`cards:`

0	1	2	3	4	5	6
5♠	J♥	2♦	K♦	A♠	4♥	6♣

`size: 5`

and the 4♥ would be returned. Now both the 4♥ and the 6♣ are in the dealt portion of the deck.

Complete the `deal` method using this new algorithm.

```
public Card deal() {
```

2. Consider the following class that creates a list of compound words:

```
public class CompoundWordCreator
{
   private List<String> wordList; //contains no duplicates

   /** @return true if word is in the dictionary; false otherwise
    */
   private boolean inDictionary(String word) {
      /* implementation not shown */
   }

   /** Combines all pairs of words in wordlist whose lengths sum to letterSum,
    *  and adds the new words to the list compoundWords if the new words were
    *  found in the dictionary. Words should not be combined with themselves.
    */
   private void addCompoundWords(List<String> compoundWords, int letterSum) {
      /* to be completed in part (b) */
   }

   /** precondition: wordList.size() > 0
    *  @return the length of the longest word in wordList
    */
   private int findMaxLength() {
      /* to be completed in part (a) */
   }

   /** precondition: wordList.size() > 0
    *  @return a list of compound words found in the dictionary that were created
    *          by combining strings in the list wordList
    *  postcondition: for each word, w, in list, inDictionary(w) == true and
    *                 3 <= w.length() <= findMaxLength().
    */
   public List<String> buildWords() {
      /* to be completed in part (c) */
   }

   // Constructors, other methods, and instance variables are not shown.
}
```

(a) Method `findMaxLength` should find and return the length of the longest word in `wordList`. Complete method `findMaxLength`.

```
/** precondition: wordList.size() > 0
  * @return the length of the longest word in wordList
  */
private int findMaxLength() {
```

(b) The list, `wordList`, contains a list of unique words that can be found in the dictionary. Method `addCompoundWords` creates two new words for each pair of words in `wordList` whose lengths add to `letterSum`. (For example, if “less” and “time” are chosen as a pair of words whose lengths sum to **8**, then the two new words created would be “lesstime” and “timeless.”) Once the words are created, it checks to see if the new words are in the dictionary. Any new word that is found in the dictionary will be added to `compoundWords`. A word should not be combined with itself to create a new word.

Complete method `addCompoundWords`.

```
/** Combines all pairs of words in wordlist whose lengths sum to letterSum,
  * and adds the new words to the list compoundWords if the new words were
  * found in the dictionary. Words should not be combined with themselves
  */
private void addCompoundWords(List<String> compoundWords, int letterSum) {
```

(c) The `buildWords` method builds a list of new words. It finds these new words by considering all pairs of unique words from `wordList`. If their combined lengths are between 3 and the length of the largest word, inclusive, they are concatenated to create two new words. New words that can be found in the dictionary are added to `compoundWords`. You may use any methods found in the `CompoundWordCreator` class declaration. You may assume that anything you wrote in part (a) and part (b) works as intended.

Complete method `buildWords` below.

```
/** precondition: wordList.size() > 0
 *  @return a list of compound words found in the dictionary that were created
 *          by combining strings in the list wordList
 *  postcondition: for each word, w, in the returned list, inDictionary(w) == true and
 *                 3 <= w.length() <= findMaxLength().
 */
public List<String> buildWords() {
```

AP® Computer Science A
Picture Lab
Student Guide

The AP Program wishes to acknowledge and thank Barbara Ericson of the Georgia Institute of Technology, who developed this lab and the accompanying documentation.

Picture Lab: Student Guide

Introduction

In this lab you will be writing methods that modify digital pictures. In writing these methods you will learn how to traverse a two-dimensional array of integers or objects. You will also be introduced to nested loops, binary numbers, interfaces, and inheritance.

Activities

You will be working through a set of activities. These activities will help you learn about how:

- digital pictures are represented on a computer;
- the binary number system is used to represent values;
- to create colors using light;
- Java handles two-dimensional arrays;
- data from a picture is stored; and
- to modify a digital picture.

Set-up

You will need the `pixLab` folder and a Java Development Kit, also known as a JDK (see http://www.oracle.com/technetwork/java/javase/downloads/index.html). A development environment is also useful. DrJava is a free development environment for Java that allows students to try out code in an interactions pane. It also has a debugger, and can be downloaded from http://drjava.org. However, you can use any development environment with this lab. Just open the files in the `classes` folder and compile them. Please note that there are two small pictures in the `classes` folder that need to remain there: `leftArrow.gif` and `rightArrow.gif`. If you copy the Java source files to another folder you must copy these gif files as well.

Keep the `images` folder and the `classes` folder together in the `pixLab` folder. The `FileChooser` expects the images to be in a folder called `images`, at the same level as the `classes` folder. If it does not find the images there it also looks in the same folder as the class files that are executing. If you wish to modify this, change the `FileChooser.java` class to specify the folder where the pictures are stored. For example, if you want to store the images in "`r://student/images/`," change the following line in the method `getMediaDirectory()` in `FileChooser.java`:

```
URL fileURL = new URL(classURL,"../images/");
```

And modify it to

```
URL fileURL = new URL("r://student/images/");
```

Then recompile.

A1: Introduction to digital pictures and color

If you look at an advertisement for a digital camera, it will tell you how many *megapixels* the camera can record. What is a megapixel? A digital camera has sensors that record color at millions of points arranged in rows and columns (Figure 1). Each point is a *pixel* or *picture (abbreviated **pix**) **el**ement*. A *megapixel* is one million pixels. A 16.2 megapixel camera can store the color at over 16 million pixels. That's a lot of pixels! Do you really need all of them? If you are sending a small version of your picture to a friend's phone, then just a few megapixels will be plenty. But, if you are printing a huge poster from a picture or you want to zoom in on part of the picture, then more pixels will give you more detail.

How is the color of a pixel recorded? It can be represented using the RGB (Red, Green, Blue) color model, which stores values for red, green, and blue, each ranging from 0 to 255. You can make yellow by combining red and green. That probably sounds strange, but combining pixels isn't the same as mixing paint to make a color. The computer uses light to display color, not paint. Tilt the bottom of a CD in white light and you will see lots of colors. The CD acts as a prism and lets you see all the colors in white light. The RGB color model sometimes also stores an alpha value as well as the red, green, and blue values. The alpha value indicates how transparent or opaque the color is. A color that is transparent will let you see some of the color beneath it.

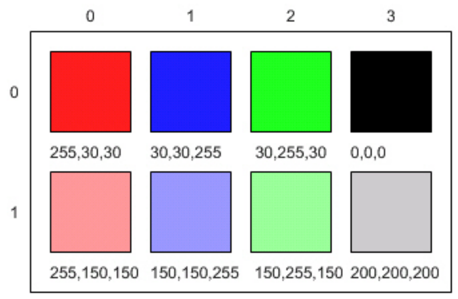


Figure 1: RGB values and the resulting colors displayed in rows and columns

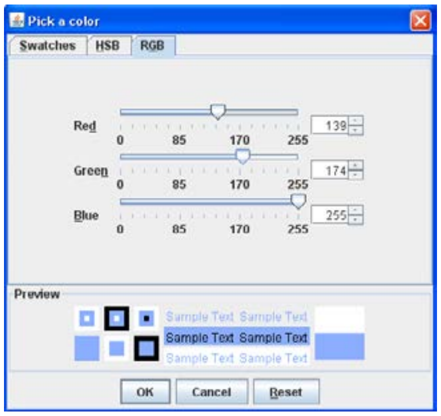
How does the computer represent the values from 0 to 255? A decimal number uses the digits 0 to 9 and powers of 10 to represent values. The decimal number 325 means 5 ones (10^0) plus 2 tens (10^1) plus 3 hundreds (10^2) for a total of three hundred and twenty-five. Computers use *binary numbers*, which use the digits 0 and 1 and powers of 2 to represent values using groups of bits. A *bit* is a **b**inary dig**it,** which can be either 0 or 1. A group of 8 bits is called a *byte*. The binary number 110 means 0 ones (2^0) plus 1 two (2^1) plus 1 four (2^2), for a total of 6.

Questions

1. How many bits does it take to represent the values from 0 to 255?
2. How many bytes does it take to represent a color in the RBG color model?
3. How many pixels are in a picture that is 640 pixels wide and 480 pixels high?

A2: Picking a color

Run the `main` method in `ColorChooser.java`. This will pop up a window (Figure 2) asking you to pick a color. Click on the RGB tab and move the sliders to make different colors.



Red 139
Green 174
Blue 255

Figure 2: The Color Chooser (This is the version from Java 6.)

When you click the OK button, the red, green, and blue values for the color you picked will be displayed as shown below. The `Color` class has a `toString` method that displays the class name followed by the red, green, and blue values. The `toString` method is automatically called when you print an object.

```
java.awt.Color[r=139,g=174,b=255]
```

Java represents color using the `java.awt.Color` class. This is the *full name* for the `Color` class, which includes the *package* name of `java.awt` followed by a period and then the class name `Color`. Java groups related classes into *packages*. The *awt* stands for Abstract Windowing Toolkit, which is the package that contains the original Graphical User Interface (GUI) classes developed for Java. You can use just the short name for a class, like `Color`, as long as you include an import statement at the beginning of a class source file, as shown below. The `Picture` class contains the following import statement.

```
import java.awt.Color;
```

Use the `ColorChooser` class (run the `main` method) to answer the following questions.

Questions

1. How can you make pink?
2. How can you make yellow?
3. How can you make purple?

4. How can you make white?
5. How can you make dark gray?

A3: Exploring a picture

Run the `main` method in `PictureExplorer.java`. This will load a picture of a beach from a file, make a copy of that picture in memory, and show it in the explorer tool (Figure 3). It makes a copy of the picture to make it easier to explore a picture both before and after any changes. You can use the explorer tool to explore the pixels in a picture. Click any location (pixel) in the picture and it will display the row index, column index, and red, green, and blue values for that location. The location will be highlighted with yellow crosshairs. You can click on the arrow keys or even type in values and hit the enter button to update the display. You can also use the menu to change the zoom level.

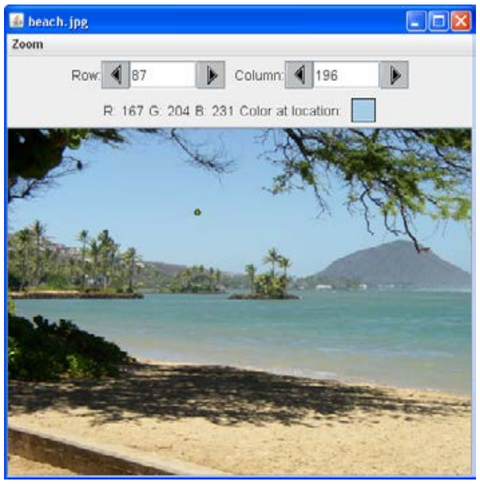


Figure 3: The Picture Explorer

Questions

1. What is the row index for the top left corner of the picture?
2. What is the column index for the top left corner of the picture?
3. The width of this picture is 640. What is the right most column index?
4. The height of this picture is 480. What is the bottom most row index?
5. Does the row index increase from left to right or top to bottom?
6. Does the column index increase from left to right or top to bottom?
7. Set the zoom to 500%. Can you see squares of color? This is called *pixelation*. Pixelation means displaying a picture so magnified that the individual pixels look like small squares.

Creating and exploring other pictures

Here is the `main` method in the class `PictureExplorer`. Every class in Java can have a `main` method, and it is where execution starts when you execute the command `java` *`ClassName`*.

```
public static void main( String args[])
{
  Picture pix = new Picture("beach.jpg");
  pix.explore();
}
```

The body of the `main` method declares a reference to a `Picture` object named `pix` and sets that variable to refer to a `Picture` object created from the data stored in a JPEG file named "`beach.jpg`" in the `images` folder. A JPEG file is one that follows an international standard for storing picture data using *lossy compression*. *Lossy compression* means that the amount of data that is stored is much smaller than the available data, but the part that is not stored is data we won't miss.

Exercises

1. Modify the `main` method in the `PictureExplorer` class to create and explore a different picture from the `images` folder.
2. Add a picture to the `images` folder and then create and explore that picture in the `main` method. If the picture is very large (for instance, one from a digital camera), you can scale it using the `scale` method in the `Picture` class.
   For example, you can make a new picture ("`smallMyPicture.jpg`" in the `images` folder) one-fourth the size of the original ("`myPicture.jpg`") using:

   ```
   Picture p = new Picture("myPicture.jpg");
   Picture smallP = p.scale(0.25,0.25);
   smallP.write("smallMyPicture.jpg");
   ```

A4: Two-dimensional arrays in Java

In this activity you will work with integer data stored in a two-dimensional array. Some programming languages use a one-dimensional (1D) array to represent a two-dimensional (2D) array with the data in either *row-major* or *column-major order*. *Row-major order* in a 1D array means that all the data for the first row is stored before the data for the next row in the 1D array. *Column-major order* in a 1D array means that all the data for the first column is stored before the data for the next column in the 1D array. The order matters, because you need to calculate the position in the 1D array based on the order, the number of rows and columns, and the current column and row numbers (indices). The rows and columns are numbered (indexed) and often that numbering starts at 0 as it does in Java. The top left row has an index of 0 and the top left column has an index of 0. The row number (index) increases from top to bottom and the column number (index) increases from left to right as shown below.

	0	1	2
0	1	2	3
1	4	5	6

If the above 2D array is stored in a 1D array in row-major order it would be:

0	1	2	3	4	5
1	2	3	4	5	6

If the above 2D array is stored in a 1D array in column-major order it would be:

0	1	2	3	4	5
1	4	2	5	3	6

Java actually uses arrays of arrays to represent 2D arrays. This means that each element in the outer array is a reference to another array. The data can be in either row-major or column-major order (Figure 4). The AP Computer Science A course specification tells you to assume that all 2D arrays are row-major, which means that the outer array in Java represents the rows and the inner arrays represent the columns.

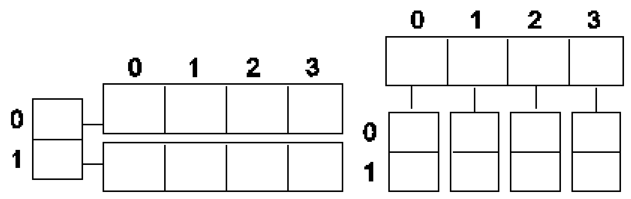


Figure 4: A row-major 2D array (left) and a column-major 2D array (right)

The following table shows the Java syntax and examples for tasks with 2D arrays. Java supports 2D arrays of primitive and object types.

Task	Java Syntax	Examples

Declare a 2D array	`type[][] name`	`int[][] matrix`
		`Pixel[][] pixels`

Create a 2D array	`new type[nRows][nCols]`	`new int[5][8]`
		`new Pixel[numRows][numCols]`

Access an element	`name[row][col]`	`int value = matrix[3][2];`
		`Pixel pixel = pixels[r][c];`

Set the value of an element	`name[row][col] = value`	`matrix[3][2] = 8;`
		`pixels[r][c] = aPixel;`

Get the number of rows	`name.length`	`matrix.length` `pixels.length`

Get the number of columns	`name[0].length`	`matrix[0].length`
		`pixels[0].length`

To loop through the values in a 2D array you must have two indexes. One index is used to change the row index and one is used to change the column index. You can use *nested loops,* which is one `for` loop inside of another, to loop through all the values in a 2D array.

Here is a method in the `IntArrayWorker` class that totals all the values in a 2D array of integers in a private instance variable (field in the class) named `matrix`. Notice the nested `for` loop and how it uses `matrix.length` to get the number of rows and `matrix[0].length` to get the number of columns. Since `matrix[0]` returns the inner array in a 2D array, you can use `matrix[0].length` to get the number of columns.

```
public int getTotal()
{
  int total = 0;
  for (int row = 0; row < matrix.length; row++)
  {
    for (int col = 0; col < matrix[0].length; col++)
    {
      total = total + matrix[row][col];
    }
  }
  return total;
}
```

Because Java two-dimensional arrays are actually arrays of arrays, you can also get the total using nested `for-each` loops as shown in `getTotalNested` below. The outer loop will loop through the outer array (each of the rows) and the inner loop will loop through the inner array (columns in that row). You can use a nested `for-each` loop whenever you want to loop through all items in a 2D array and you don't need to know the row index or column index.

```
public int getTotalNested()
{
  int total = 0;
  for (int[] rowArray : matrix)
  {
    for (int item : rowArray)
    {
      total = total + item;
    }
  }
  return total;
}
```

Exercises

1. Write a `getCount` method in the `IntArrayWorker` class that returns the count of the number of times a passed integer value is found in the matrix. There is already a method to test this in `IntArrayWorkerTester`. Just uncomment the method `testGetCount()` and the call to it in the `main` method of `IntArrayWorkerTester`.
2. Write a `getLargest` method in the `IntArrayWorker` class that returns the largest value in the matrix. There is already a method to test this in `IntArrayWorkerTester`. Just uncomment the method `testGetLargest()` and the call to it in the `main` method of `IntArrayWorkerTester`.
3. Write a `getColTotal` method in the `IntArrayWorker` class that returns the total of all integers in a specified column. There is already a method to test this in `IntArrayWorkerTester`. Just uncomment the method `testGetColTotal()` and the call to it in the `main` method of `IntArrayWorkerTester`.

A5: Modifying a picture

Even though digital pictures have millions of pixels, modern computers are so fast that they can process all of them quickly. You will write methods in the `Picture` class that modify digital pictures. The `Picture` class inherits from the `SimplePicture` class and the `SimplePicture` class implements the `DigitalPicture` interface as shown in the Unified Modeling Language (UML) class diagram in Figure 5.

A UML class diagram shows classes and the relationships between the classes. Each class is shown in a box with the class name at the top. The middle area shows attributes (instance or class variables) and the bottom area shows methods. The open triangle points to the class that the connected class inherits from. The straight line links show associations between classes. Association is also called a “has-a” relationship. The numbers at the end of the association links give the number of objects associated with an object at the other end. For example, in Figure 5 it shows that one `Pixel` object has one `Color` object associated with it and that a `Color` object can have zero to many `Pixel` objects associated with it. You may notice that the UML class diagram doesn't look exactly like Java code. UML isn't language specific.

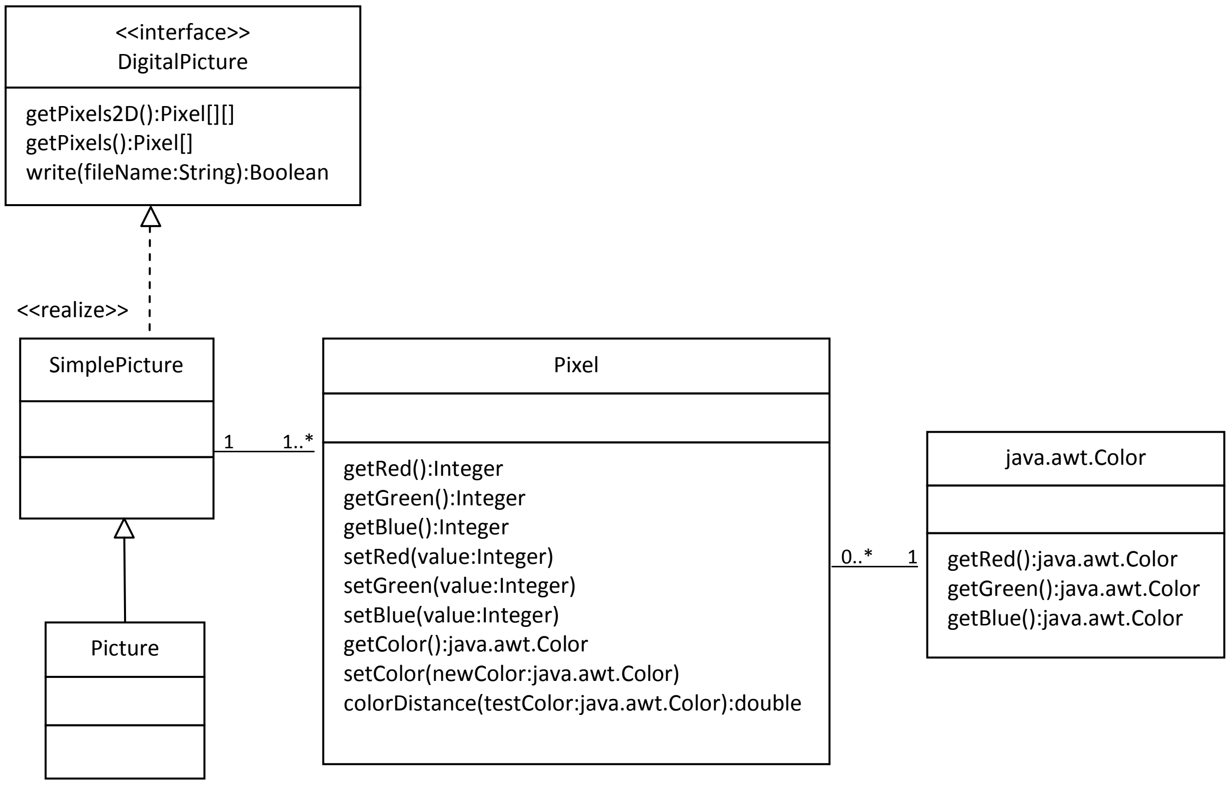


Figure 5: A UML Class Diagram

Questions

1. Open `Picture.java` and look for the method `getPixels2D`. Is it there?
2. Open `SimplePicture.java` and look for the method `getPixels2D`. Is it there?
3. Does the following code compile?
   ```
   DigitalPicture p = new DigitalPicture();
   ```
4. Assuming that a no-argument constructor exists for `SimplePicture`, would the following code compile?
   ```
   DigitalPicture p = new SimplePicture();
   ```
5. Assuming that a no-argument constructor exists for `Picture`, does the following code compile?
   ```
   DigitalPicture p = new Picture();
   ```
6. Assuming that a no-argument constructor exists for `Picture`, does the following code compile?
   ```
   SimplePicture p = new Picture();
   ```
7. Assuming that a no-argument constructor exists for `SimplePicture`, does the following code compile?
   ```
   Picture p = new SimplePicture();
   ```

`DigitalPicture` is an *interface*. An *interface* most often only has public abstract methods. An *abstract method* is not allowed to have a body. Notice that none of the methods declared in `DigitalPicture` have a body. If a method can't have a body, what good is it?

Interfaces are useful for separating **what** from **how**. An interface specifies **what** an object of that type needs to be able to do but not **how** it does it. You cannot create an object using an interface type. A class can *implement* (*realize*) an interface as `SimplePicture` does. A non-abstract class provides bodies for all the methods declared in the interface, either directly or through inheritance. You can declare a variable to be of an interface type and then set that variable to refer to an object of any class that implements that interface. For example, Java has a `List` interface that declares the methods that a list should have such as `add`, `remove`, and `get`, etc. But, if you want to create a `List` object you will create an `ArrayList` object. It is recommended that you declare a variable to be of type `List`, not `ArrayList`, as shown below (for a list of names).

```
List<String> nameList = new ArrayList<String>();
```

Why wouldn't you just declare `nameList` to be of the type `ArrayList<String>`? There are other classes in Java that implement the `List` interface. By declaring `nameList` to be of the type `List<String>` instead of `ArrayList<String>`, it is easy to change your mind in the future and use another class that implements the same interface. Interfaces give you some flexibility and reduce the number of changes you might need to make in the future, as long as your code only uses the functionality defined by the interface.

Because `DigitalPicture` declares a `getPixels2D` method that returns a two-dimensional array of `Pixel` objects, `SimplePicture` implements that interface, and `Picture` inherits from `SimplePicture`, you can use the `getPixels2D` method on a `Picture` object. You can loop through all the `Pixel` objects in the two-dimensional array to modify the picture. You can get and set the red, green, and/or blue value for a `Pixel` object. You can also get and/or set the `Color` value for a `Pixel` object. You can create a new `Color` object using a constructor that takes the red, green, and blue values as integers as shown below.

```
Color myColor = new Color(255,30,120);
```

What do you think you will see if you modify the beach picture in the `images` folder to set all the blue values to zero? Do you think you will still see a beach? Run the `main` method in the `Picture` class. The body of the `main` method will create a `Picture` object named `beach` from the "`beach.jpg`" file, open an explorer on a copy of the picture (in memory), call the method that sets the blue values at all pixels to zero, and then open an explorer on a copy of the resulting picture.

The following code is the `main` method from the `Picture` class.

```
    public static void main(String[] args)
    {
      Picture beach = new Picture("beach.jpg");
      beach.explore();
      beach.zeroBlue();
      beach.explore();
    }
```

Exercises

1. Open `PictureTester.java` and run its `main` method. You should get the same results as running the `main` method in the `Picture` class. The `PictureTester` class contains class (static) methods for testing the methods that are in the `Picture` class.
2. Uncomment the appropriate test method in the `main` method of `PictureTester` to test any of the other methods in `Picture.java`. You can comment out the tests you don't want to run. You can also add new test methods to `PictureTester` to test any methods you create in the `Picture` class.

The method `zeroBlue` in the `Picture` class gets a two-dimensional array of `Pixel` objects from the current picture (the picture the method was called on). It then declares a variable that will refer to a `Pixel` object named `pixelObj`. It uses a nested `for-each` loop to loop through all the pixels in the picture. Inside the body of the nested `for-each` loop it sets the blue value for the current pixel to zero. Note that you cannot change the elements of an array when you use a `for-each` loop. If, however, the array elements are references to objects that have methods that allow changes, you can change the internal state of objects referenced in the array (pixels).

The following code is the `zeroBlue` method in the `Picture` class.

```
public void zeroBlue()
{
  Pixel[][] pixels = this.getPixels2D();
  for (Pixel[] rowArray : pixels)
  {
    for (Pixel pixelObj : rowArray)
    {
      pixelObj.setBlue(0);
    }
  }
}
```

Exercises

3. Using the `zeroBlue` method as a starting point, write the method `keepOnlyBlue` that will keep only the blue values, that is, it will set the red and green values to zero. Create a class (static) method to test this new method in the class `PictureTester`. Be sure to call the new test method in the `main` method in `PictureTester`.
4. Write the `negate` method to negate all the pixels in a picture. To negate a picture, set the red value to 255 minus the current red value, the green value to 255 minus the current green value and the blue value to 255 minus the current blue value. Create a class (static) method to test this new method in the class `PictureTester`. Be sure to call the new test method in the `main` method in `PictureTester`.
5. Write the `grayscale` method to turn the picture into shades of gray. Set the red, green, and blue values to the average of the current red, green, and blue values (add all three values and divide by 3). Create a class (static) method to test this new method in the class `PictureTester`. Be sure to call the new test method in the `main` method in `PictureTester`.
6. Challenge — Explore the "`water.jpg`" picture in the `images` folder. Write a method `fixUnderwater()` to modify the pixel colors to make the fish easier to see. Create a class (static) method to test this new method in the class `PictureTester`. Be sure to call the new test method in the `main` method in `PictureTester`.

A6: Mirroring pictures

Car designers at General Motors Research Labs only sculpt half of a car out of clay and then use a vertical mirror to reflect that half to see the whole car. What if we want to see what a picture would look like if we placed a mirror on a vertical line in the center of the width of the picture to reflect the left side (Figure 6)?

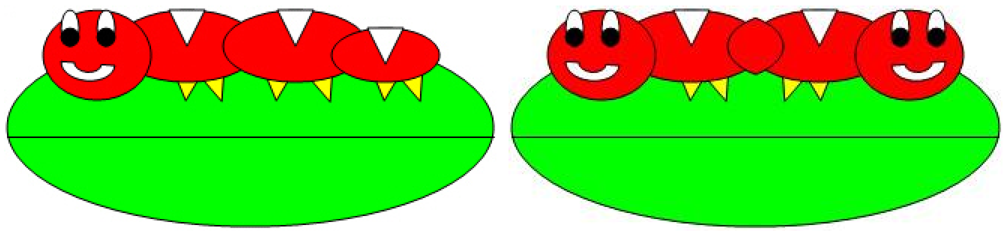
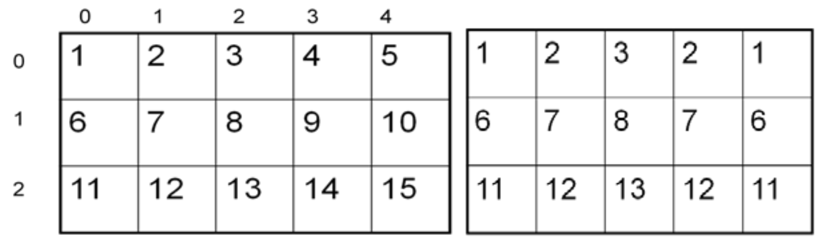


Figure 6: Original picture (left) and picture after mirroring (right)

How can we write a method to mirror a picture in this way? One way to figure out the *algorithm,* which is a description of the steps for solving a problem, is to try it on smaller and simpler data. Figure 7 shows the result of mirroring a two-dimensional array of numbers from left to right vertically.



	0	1	2	3	4
0	1	2	3	4	5
1	6	7	8	9	10
2	11	12	13	14	15

1	2	3	2	1
6	7	8	7	6
11	12	13	12	11

Figure 7: Two-Dimensional array of numbers (left) and mirrored result (right)

Can you figure out the *algorithm* for this process? Test your algorithm on different sizes of two-dimensional arrays of integers. Will it work for 2D arrays with an odd number of columns? Will it work for 2D arrays with an even number of columns?

One algorithm is to loop through all the rows and half the columns. You need to get a pixel from the left side of the picture and a pixel from the right side of the picture, which is the same distance from the right end as the left pixel is from the left end. Set the color of the right pixel to the color of the left pixel. The column number at the right end is the number of columns, also known as the width, minus one. So assuming there are at least 3 pixels in a row, the first left pixel will be at row=0, col=0 and the first right pixel will be at row=0, col=width-1. The second left pixel will be at row=0, col=1 and the corresponding right pixel will be at row=0, col=width-1-1. The third left pixel will be at row=0, col=2 and its right pixel will be at row=0, col=width-1-2. Each time the left pixel is at (current row value, current column value), the corresponding right pixel is at (current row value, width - 1 - (current column value)).

The following method implements this algorithm. Note that, because the method is not looping through all the pixels, it cannot use a nested `for-each` loop.

```
public void mirrorVertical()
{
  Pixel[][] pixels = this.getPixels2D();
  Pixel leftPixel = null;
  Pixel rightPixel = null;
  int width = pixels[0].length;
  for (int row = 0; row < pixels.length; row++)
  {
    for (int col = 0; col < width / 2; col++)
    {
      leftPixel = pixels[row][col];
      rightPixel = pixels[row][width – 1 - col];
      rightPixel.setColor(leftPixel.getColor());
    }
  }
}
```

You can test this with the `testMirrorVertical` method in `PictureTester`.

Exercises

1. Write the method `mirrorVerticalRightToLeft` that mirrors a picture around a mirror placed vertically from right to left. Hint: you can copy the body of `mirrorVertical` and only change one line in the body of the method to accomplish this. Write a class (static) test method called `testMirrorVertical` in `PictureTester` to test this new method and call it in the `main` method.
2. Write the method `mirrorHorizontal` that mirrors a picture around a mirror placed horizontally at the middle of the height of the picture. Mirror from top to bottom as shown in the pictures below (Figure 8). Write a class (static) test method in `PictureTester` to test this new method and call it in the `main` method.

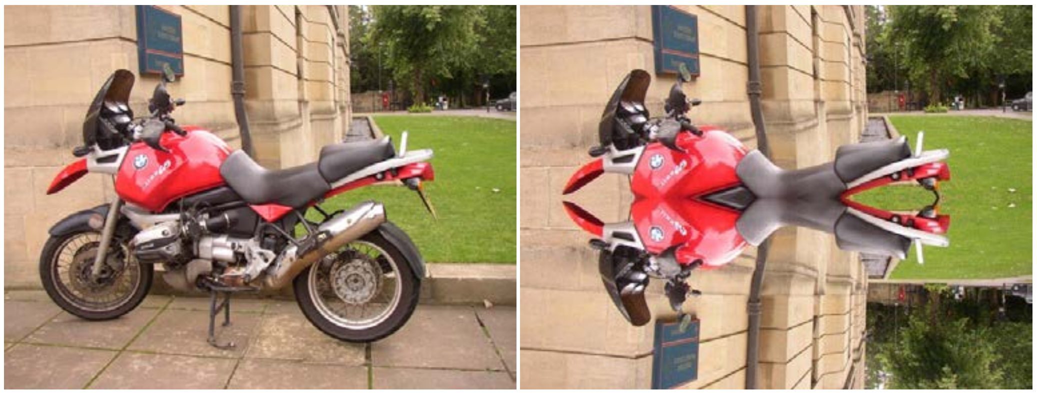


Figure 8: Original picture (left) and mirrored from top to bottom (right)

3. Write the method `mirrorHorizontalBotToTop` that mirrors the picture around a mirror placed horizontally from bottom to top. Hint: you can copy the body of `mirrorHorizontal` and only change one line to accomplish this. Write a class (static) test method in `PictureTester` to test this new method and call it in the `main` method.
4. Challenge — Work in groups to figure out the algorithm for the method `mirrorDiagonal` that mirrors just a square part of the picture from bottom left to top right around a mirror placed on the diagonal line (the diagonal line is the one where the row index equals the column index). This will copy the triangular area to the left and below the diagonal line as shown below. This is like folding a square piece of paper from the bottom left to the top right, painting just the bottom left triangle and then (while the paint is still wet) folding the paper up to the top right again. The paint would be copied from the bottom left to the top right as shown in the pictures below (Figure 9). Write a class (static) test method in `PictureTester` to test this new method and call it in the `main` method.

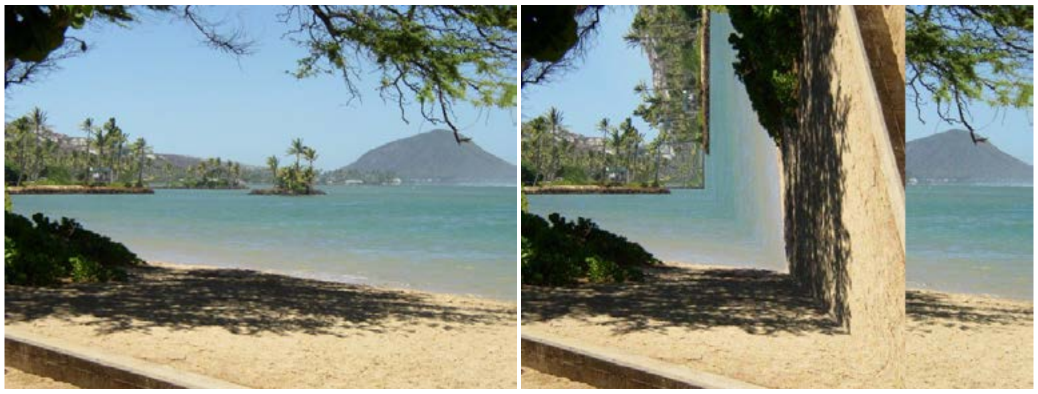


Figure 9: Original picture (left) and mirrored around the diagonal line with copying from bottom left to top right (right)

A7: Mirroring part of a picture

Sometimes you only want to mirror part of a picture. For example, Figure 10 shows a temple in Greece that is missing a part of the roof called the pediment. You can use the explorer tool to find the area that you want to mirror to produce the picture on the right. If you do this you will find that you can mirror the rows from 27 to 96 (inclusive) and the columns from 13 to 275 (inclusive). You can change the starting and ending points for the row and column values to mirror just part of the picture.

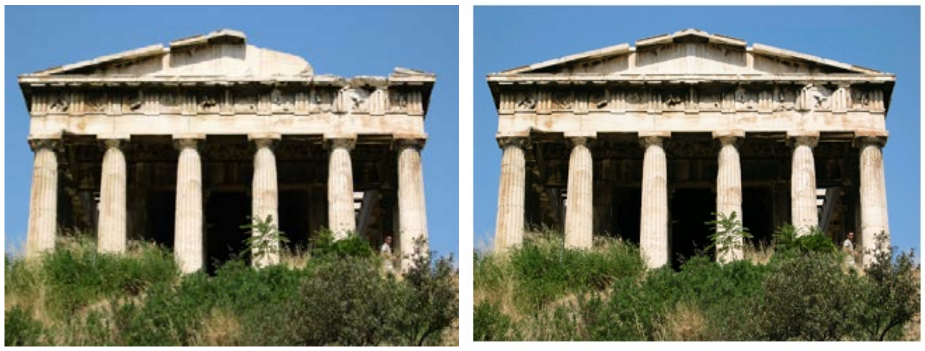


Figure 10: Greek temple before (left) and after (right) mirroring the pediment

To work with just part of a picture, change the starting and ending values for the nested `for` loops as shown in the following `mirrorTemple` method. This method also calculates the distance the current column is from the `mirrorPoint` and then adds that distance to the `mirrorPoint` to get the column to copy to.

```
public void mirrorTemple()
{
  int mirrorPoint = 276;
  Pixel leftPixel = null;
  Pixel rightPixel = null;
  int count = 0;
  Pixel[][] pixels = this.getPixels2D();

  // loop through the rows
  for (int row = 27; row < 97; row++)
  {
    // loop from 13 to just before the mirror point
    for (int col = 13; col < mirrorPoint; col++)
    {

      leftPixel = pixels[row][col];
      rightPixel = pixels[row]
                       [mirrorPoint - col + mirrorPoint];
      rightPixel.setColor(leftPixel.getColor());
    }
```

```
      }
   }
```

You can test this with the `testMirrorTemple` method in `PictureTester`.

How many times was `leftPixel = pixels[row][col];` executed? The formula for the number of times a nested loop executes is the number of times the *outer loop* executes multiplied by the number of times the *inner loop* executes. The outer loop is the one looping through the rows, because it is outside the other loop. The inner loop is the one looping through the columns, because it is inside the row loop.

How many times does the outer loop execute? The outer loop starts with `row` equal to 27 and ends when it reaches 97, so the last time through the loop `row` is 96. To calculate the number of times a loop executes, subtract the starting value from the ending value and add one. The outer loop executes 96 – 27 + 1 times, which equals 70 times. The inner loop starts with `col` equal to 13 and ends when it reaches 276, so, the last time through the loop, `col` will be 275. It executes 275 – 13 + 1 times, which equals 263 times. The total is 70 * 263, which equals 18,410.

Questions

1. How many times would the body of this nested `for` loop execute?
   ```
   for (int row = 7; row < 17; row++)
     for (int col = 6; col < 15; col++)
   ```
2. How many times would the body of this nested `for` loop execute?
   ```
   for (int row = 5; row <= 11; row++)
     for (int col = 3; col <= 18; col++)
   ```

Exercises

1. Check the calculation of the number of times the body of the nested loop executes by adding an integer `count` variable to the `mirrorTemple` method that starts out at 0 and increments inside the body of the loop. Print the value of `count` after the nested loop ends.
2. Write the method `mirrorArms` to mirror the arms on the snowman (“`snowman.jpg`”) to make a snowman with 4 arms. Write a class (static) test method in `PictureTester` to test this new method and call it in the `main` method.
3. Write the method `mirrorGull` to mirror the seagull (“`seagull.jpg`”) to the right so that there are two seagulls on the beach near each other. Write a class (static) test method in `PictureTester` to test this new method and call it in the `main` method.

A8: Creating a collage

You can copy one picture to another by copying the color from the pixels in one picture to the pixels in the other picture. To do this you will need to keep track of the row and column information for both the picture you are copying from and the picture you are copying to, as shown in the following `copy` method. The easiest way to do this is to declare and initialize both a `fromRow` and `toRow` in the outer `for` loop and increment them both at the end of the loop. A `for` loop can have more than one variable declaration and initialization and/or modification. Just separate the items with commas. Note that the inner loop has both a `fromCol` and a `toCol` declared, initialized, and incremented.

```
public void copy(Picture fromPic,
                 int startRow, int startCol)
{
  Pixel fromPixel = null;
  Pixel toPixel = null;
  Pixel[][] toPixels = this.getPixels2D();
  Pixel[][] fromPixels = fromPic.getPixels2D();
  for (int fromRow = 0, toRow = startRow;
       fromRow < fromPixels.length &&
       toRow < toPixels.length;
       fromRow++, toRow++)
  {
    for (int fromCol = 0, toCol = startCol;
         fromCol < fromPixels[0].length &&
         toCol < toPixels[0].length;
         fromCol++, toCol++)
    {
      fromPixel = fromPixels[fromRow][fromCol];
      toPixel = toPixels[toRow][toCol];
      toPixel.setColor(fromPixel.getColor());
    }
  }
}
```

You can create a collage by copying several small pictures onto a larger picture. You can do some picture manipulations like zero blue before you copy the picture as well. You can even mirror the result to get a nice artistic effect (Figure 11).

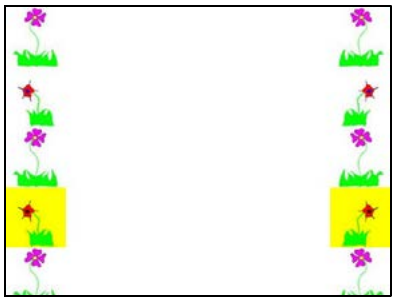


Figure 11: Collage with vertical mirror

The following method shows how to create a simple collage using the `copy` method.

```
public void createCollage()
{
  Picture flower1 = new Picture("flower1.jpg");
  Picture flower2 = new Picture("flower2.jpg");
  this.copy(flower1,0,0);
  this.copy(flower2,100,0);
  this.copy(flower1,200,0);
  Picture flowerNoBlue = new Picture(flower2);
  flowerNoBlue.zeroBlue();
  this.copy(flowerNoBlue,300,0);
  this.copy(flower1,400,0);
  this.copy(flower2,500,0);
  this.mirrorVertical();
  this.write("collage.jpg");
}
```

Notice that the `Picture` method `write` can be used to save a copy of the final collage to your disk as a JPEG picture file. You can also specify the full path name of where to write the picture (“`c:\temp\collage.jpg`”). Be sure to include the extension `(.jpg)` as well so that your computer knows the file type.

You can test this with the `testCollage` method in `PictureTester`.

Exercises

1. Create a second `copy` method that adds parameters to allow you to copy just part of the `fromPic`. You will need to add parameters that specify the start row, end row, start column,

and end column to copy from. Write a class (static) test method in `PictureTester` to test this new method and call it in the `main` method.

2. Create a `myCollage` method that has at least three pictures (can be the same picture) copied three times with three different picture manipulations and at least one mirroring. Write a class (static) test method in `PictureTester` to test this new method and call it in the `main` method.

A9: Simple edge detection

Detecting edges is a common image processing problem. For example, digital cameras often feature face detection. Some robotic competitions require the robots to find a ball using a digital camera, so the robot needs to be able to "see" a ball.

One way to look for an edge in a picture is to compare the color at the current pixel with the pixel in the next column to the right. If the colors differ by more than some specified amount, this indicates that an edge has been detected and the current pixel color should be set to black. Otherwise, the current pixel is not part of an edge and its color should be set to white (Figure 12). How do you calculate the difference between two colors? The formula for the difference between two points (x_1,y_1) and (x_2,y_2) is the square root of $((x_2 - x_1)^2 + (y_2 - y_1)^2)$. The difference between two colors $(red_1,green_1,blue_1)$ and $(red_2,green_2,blue_2)$ is the square root of $((red_2 - red_1)^2 + (green_2 - green_1)^2 + (blue_2 - blue_1)^2)$. The `colorDistance` method in the `Pixel` class uses this calculation to return the difference between the current pixel color and a passed color.

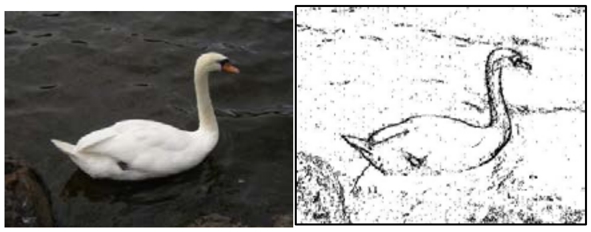


Figure 12: Original picture and after edge detection

The following method implements this simple algorithm. Notice that the nested `for` loop stops earlier than when it reaches the number of columns. That is because in the nested loop the current color is compared to the color at the pixel in the next column. If the loop continued to the last column this would cause an out-of-bounds error.

```
public void edgeDetection(int edgeDist)
{
  Pixel leftPixel = null;
  Pixel rightPixel = null;
  Pixel[][] pixels = this.getPixels2D();
  Color rightColor = null;
  for (int row = 0; row < pixels.length; row++)
  {
    for (int col = 0;
         col < pixels[0].length-1; col++)
    {
      leftPixel = pixels[row][col];
      rightPixel = pixels[row][col+1];
      rightColor = rightPixel.getColor();
      if (leftPixel.colorDistance(rightColor) >
          edgeDist)
        leftPixel.setColor(Color.BLACK);
      else
        leftPixel.setColor(Color.WHITE);
    }
  }
}
```

You can test this with the `testEdgeDetection` method in `PictureTester`.

Exercises

1. Notice that the current edge detection method works best when there are big color changes from left to right but not when the changes are from top to bottom. Add another loop that compares the current pixel with the one below and sets the current pixel color to black as well when the color distance is greater than the specified edge distance.
2. Work in groups to come up with another algorithm for edge detection.

How image processing is related to new scientific breakthroughs

Many of today's important scientific breakthroughs are being made by large, interdisciplinary collaborations of scientists working in geographically widely distributed locations, producing, collecting, and analyzing vast and complex datasets.

One of the computer scientists who works on a large interdisciplinary scientific team is Dr. Cecilia Aragon. She is an associate professor in the Department of Human Centered Design & Engineering and the eScience Institute at the University of Washington, where she directs the Scientific Collaboration and Creativity Lab. Previously, she was a computer scientist in the Computational Research Division at Lawrence Berkeley National Laboratory for six years, after earning her Ph.D. in Computer Science from UC Berkeley in 2004. She earned her B.S. in mathematics from the California Institute of Technology.

Her current research focuses on human-computer interaction (HCI) and computer-supported cooperative work (CSCW) in scientific collaborations, distributed creativity, information visualization, and the visual understanding of very large data sets. She is interested in how social media and new methods of computer-mediated communication are changing scientific practice. She has developed novel visual interfaces for collaborative exploration of very large scientific data sets, and has authored or co-authored many papers in the areas of computer-supported cooperative work, human-computer interaction, visualization, visual analytics, image processing, machine learning, cyberinfrastructure, and astrophysics.

In 2008, she received the Presidential Early Career Award for Scientists and Engineers (PECASE) for her work in collaborative data-intensive science. Her research has been recognized with four Best Paper awards since 2004, and she was named one of the Top 25 Women of 2009 by Hispanic Business Magazine. She was the architect of the Sunfall data visualization and workflow management system for the Nearby Supernova Factory, which helped advance the study of supernovae in order to reduce the statistical uncertainties on key cosmological parameters that categorize dark energy, one of the grand challenges in physics today.

Cecilia Aragon is also one of the most skilled aerobatic pilots flying today. A two-time member of the U.S. Aerobatic Team, she was a medalist at the 1993 U.S. National Championships and the 1994 World Aerobatic Championships, and was the California State Aerobatic Champion.

Glossary

1. Abstract class — You cannot create an object of an abstract class type. But, you can create an object of a subclass of an abstract class (as long as the subclass is not also an abstract class).
2. Abstract method — An abstract method cannot have a method body in the class where the method is declared to be abstract.
3. Algorithm — A step-by-step description of how to solve a problem.
4. AWT — The Abstract Windowing Toolkit. It is the package that contains the original Graphical User Interface (GUI) classes developed for Java.
5. Binary number — A binary number contains only the digits 0 and 1. Each place is a power of 2 starting with 2^0 on the right. The decimal number 6 would be 110 in binary. That would be $0 * 2^0 + 1 * 2^1 + 1 * 2^2 = 6$.
6. Bit — A **b**inary dig**it**, which means that it has a value of either 0 or 1.
7. Byte — A consecutive group of 8 bits.
8. Column-major order — An order for storing two-dimensional array data in a one-dimensional array, so that all the data for the first column is stored before all the data for the second column and so on. In a two-dimensional array represented using an array of arrays (like in Java) this means that the outer array represents the columns and the inner arrays represent the rows.
9. Digital camera — A camera that can take digital pictures.
10. Digital picture — A picture that can be stored on a computer.
11. Inheritance — In Java, a class can specify the parent class from which it inherits instance variables (object fields) and object methods. Even though instance variables may be inherited, if they are declared to be private they cannot be directly accessed using dot notation in the inheriting class. Private methods that are inherited can also not be directly called in an inheriting class.
12. Inner loop — In a nested loop (a loop inside of another loop) the loop that is inside of another loop is considered the inner loop.
13. Interface — A special type of class that can only have public abstract methods in it and/or static constants.
14. Lossy compression — Lossy compression means that the amount of data that is stored is much smaller than the available data, but the part that is not stored is data that humans would not miss.
15. Media computation — A method of teaching programming by having students write programs that manipulate media: pictures, sounds, text, movies. This approach was developed by Dr. Mark Guzdial at Georgia Tech.
16. Megapixel — One million pixels.
17. Nested loop — One loop inside of another loop.
18. Outer loop — In a nested loop (a loop inside of another loop) the loop that is outside of another loop is considered the outer loop.
19. Package — A package in Java is a group of related classes.
20. Pixel — A picture (abbreviated **pix**) **el**ement.

21. RGB model — Represents color as amounts of red, green, and blue light. It sometimes also includes alpha, which is the amount of transparency.
22. Row-major order — An order for storing two-dimensional array data in a one-dimensional array, so that all the data for the first row is stored before all the data for the second row, and so on. In a two-dimensional array represented using an array of arrays (like in Java) this means that the outer array represents the rows and the inner arrays represent the columns.
23. Subclass — A class that has inherited from another class.
24. Superclass — A class that another class has inherited from.
25. UML —Unified Modeling Language. It is a general purpose modeling language used in object-oriented software development.

References

Dann, W., Cooper, S., & Ericson, B. (2009) *Exploring Wonderland: Java Programming Using Alice and Media Computation*. Englewood, NJ: Prentice-Hall.

Guzdial, M., & Ericson B. (2006) *Introduction to Computing and Programming in Java: A Multimedia Approach*. Englewood, NJ: Prentice-Hall.

Guzdial, M., & Ericson, B. (2009) *Introduction to Computing and Programming in Python: A Multimedia Approach*. (2nd ed.). Englewood, NJ: Prentice-Hall.

Guzdial, M., & Ericson, B. (2010) *Problem Solving with Data Structures using Java: A Multimedia Approach*. Englewood, NJ: Prentice-Hall.

Quick Reference

```
DigitalPicture Interface
Pixel[][] getPixels2D()         // implemented in SimplePicture
void explore()                  // implemented in SimplePicture
boolean write(String fileName)  // implemented in SimplePicture
```

```
SimplePicture Class (implements Digital Picture)
public SimplePicture()
public SimplePicture(int width, int height)
public SimplePicture(SimplePicture copyPicture)
public SinplePicture(String fileName)
public Pixel[][] getPixels2D()
public void explore()
public boolean write(String fileName)
```

```
Picture Class (extends SimplePicture)
public Picture()
public Picture(int height, int width)
public Picture(Picture copyPicture)
public Picture(String fileName)
public Pixel[][] getPixels2D()         // from SimplePicture
public void explore()                  // from SimplePicture
public boolean write(String fileName)  // from SimplePicture
```

```
Pixel Class
public double colorDistance(Color testColor)
public double getAverage()
public int getRed()
public int getGreen()
public int getBlue()
public Color getColor()
public int getRow()
public int getCol()
public void setRed(int value)
public void setGreen(int value)
public void setBlue(int value)
public void setColor(Color newColor)
```

```
java.awt.Color Class
public Color(int r, int g, int b)
public int getRed()
public int getGreen()
public int getBlue()
```

Group Work
All of these activities can be done in groups. Students can discuss the questions in the activities and answer them in groups. The exercises can also be done in groups. The only activity where you may want to have students work by themselves is the collage exercise (A8). This is to enable each student to create a collage that is meaningful to them.

Assessment

Sample Free-Response Questions

1. Write two unrelated methods for the `Picture` class.
    a. Write a method `int getCountRedOverValue(int value)` that returns a count of the number of pixels in the current picture that have a red value more than the parameter `value`.
    b. Write a method `setRedToHalfValueInTopHalf()` that sets the red value for all pixels in the top half of the picture to half the current red value.
2. Write two unrelated methods for the `Picture` class.

a. Write a method `clearBlueOverValue(int value)` that sets the blue value to 0 for every pixel that has a current blue value great than the parameter `value`.
b. Write a method `int[] getAverageForColumn(int col)` that creates and returns an array of integers the size of the number of columns that contains the average of the red, green, and blue values for each pixel in the column at index `col`.

The code for each of these methods is in `Picture.java` in the `finalClasses` folder.

Sample Multiple-Choice Questions

1. Which of the following will compile without error?

```
I.    DigitalPicture p = new Picture();

II.   DigitalPicture p = new SimplePicture();

III.  Picture p = new SimplePicture();
```

a. I, II, and III
b. II only
c. III only
d. I and II
e. II and III

Answer d is correct.

2. If the following code is in a method in the `Picture` class, what will the value of `count` be after the following code executes?

```
int count = 0;
for (int row = 5; row < 12; row++)
{
  for (int col = 8; col < 13; col++)
  {
    count++;
  }
}
```

a. 13
b. 25

c. 35
d. 42
e. 48

3. Which of the following sets the blue value to zero for all pixels in the bottom half of the picture?

I.

```
public void method1()
{
  Pixel[][] pixels = this.getPixels2D();
  Pixel pixelObj = null;
  int height = pixels.length;
  for (int row = 0; row < height / 2; row++)
  {
    for (int col = 0; col < pixels[0].length; col++)
    {
      pixelObj = pixels[row][col];
      pixelObj.setBlue(0);
    }
  }
}
```

II.

```
public void method2()
{
  Pixel[][] pixels = this.getPixels2D();
  Pixel pixelObj = null;
  int height = pixels.length;
  for (int row = height / 2; row < height; row++)
  {
    for (int col = 0; col < pixels[0].length; col++)
    {
      pixelObj = pixels[row][col];
      pixelObj.setBlue(0);
    }
  }
}
```

III.

```
public void method3()
{
  Pixel[][] pixels = this.getPixels2D();
  Pixel pixelObj = null;
  int height = pixels.length;
  for (int row = height-1; row >= height / 2; row--)
  {
    for (int col = 0; col < pixels[0].length; col++)
    {
      pixelObj = pixels[row][col];
      pixelObj.setBlue(0);
    }
  }
}
```

a. I, II, and III
b. II only
c. III only
d. I and II
e. II and III