AP® Computer Science A Elevens Lab Teacher's 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

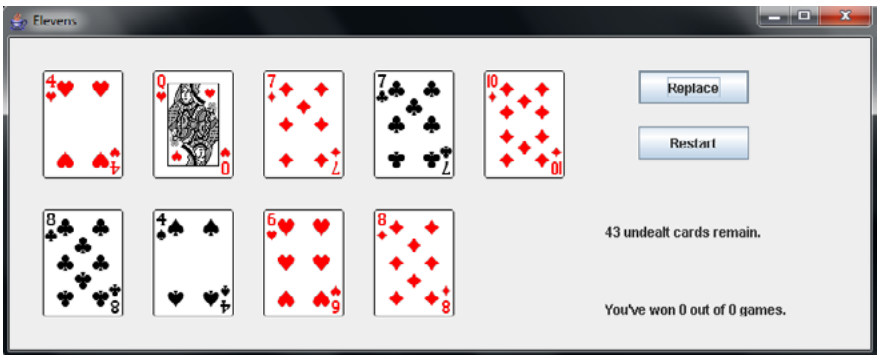
Robert Glen Martin: School for the Talented and Gifted in Dallas, TX

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

Elevens Lab Teacher's Guide

Introduction

Games are known to motivate programming students. Card games, particularly solitaire games, are especially accessible. This curriculum segment attempts to take advantage of students' interest to solidify their understanding of object-oriented design and inheritance, and to introduce the topic of simulation to determine the chances of winning a game.

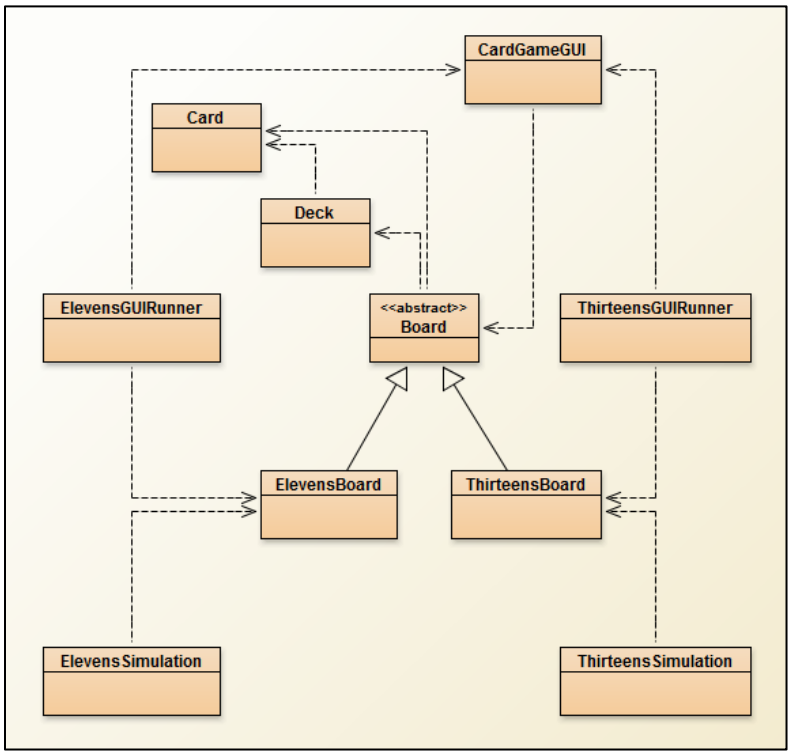


The student activities are related to a simple solitaire game called Elevens. Students will learn the rules of Elevens and will be able to play it by using the supplied Graphical User Interface (GUI) shown at the right. They will learn about the design and the Object-Oriented Principles suggested by that design. They will also receive substantial practice implementing algorithms as they write their portions of the Elevens code.

Table of Contents

Class Overview

Elevens consists of four "core" classes, a class that allows the game to be played using a GUI, a small application which initiates the GUI version of the game, and one that simulates the game. A UML diagram is provided below that shows the important inheritance ⟶▷ and uses ⇢ relationships. This diagram also includes three classes for Thirteens, a related game. The classes are:



- `Card` — A playing card. This is a simple class.
- `Deck` — A deck of cards. It can shuffle and deal its cards.
- `Board` — The 9-card layout. It has all of the board-related game independent methods, as well as the important `isLegal` and `anotherPlayIsPossible` abstract methods needed by the GUI.
- `ElevensBoard` — The "heart" of the Elevens game. It contains the game-specific code, including the implementations of the `isLegal` and `anotherPlayIsPossible` methods.
- `CardGameGUI` — The GUI that students can use to play the game. This class is completely implemented and students are not expected to understand that implementation. It can be used, without modification, for different Elevens-related games, including ones with different numbers of board cards. It relies on the specific `Board` object (such as `ElevensBoard`) and its `isLegal` and `anotherPlayIsPossible` methods to provide the game-dependent behavior. A supplied **`card`** folder contains the images used for the displayed cards.
- `ElevensGUIRunner` — A simple application that initializes the board, initializes the GUI, and then displays the GUI.
- `ElevensSimulation` — Simulates the Elevens game. It uses the `ElevensBoard` `playIfPossible` method to provide the Elevens-specific behavior.
- `ThirteensBoard`, `ThirteensGUIRunner`, `ThirteensSimulation` — The classes for the related Thirteens game.

Activity Overview

The curriculum segment consists of eleven activities. The table below briefly describes each activity, lists its prerequisites, and suggests how much time to spend on it. **It is important to note that these activities can all be done consecutively, or they can be spread out over the year, as you cover the prerequisite topics.**

Activity #	Description	Prerequisites	Java files	Time
1	Design and development of a `Card` class	Objects and classes, `int`s and `String`s	`Card.java`	1 hour
2	Initial design and implementation of a `Deck` class	Arrays, `List`s / `ArrayList`s, conditionals and loops	`Deck.java`	2 hours
3	Shuffling	`Math.random`	`Shuffler.java`	2 hours
4	Implementing the `Deck shuffle` method	`Math.random`	`Deck.java`	1 hour
5 (optional)	Testing with Java `assert` statements		`buggyDecks`, `modified Deck.java`	2 hours
6	Playing Elevens		`Elevens.jar`	1 hour
7	Design of the `ElevensBoard` class	Classes	`ElevensBoard.java` without inheritance	1 hour
8	Using an abstract `Board` class	Inheritance and abstract classes	`Board.java`, `ElevensBoard.java`	1 hour
9	Implementation of the `ElevensBoard` class	Inheritance and abstract classes	`ElevensBoard.java`	2 hours
10 (optional)	Variations on Elevens	Inheritance and abstract classes	`ThirteensBoard.java` `ThirteensGUIRunner.java`	1 hour
11 (optional)	Simulation of the Elevens game	Classes	`ElevensBoard.java` `ElevensSimulation.java` `ThirteensBoard.java` `ThirteensSimulation.java`	2 hours

Preliminaries

Learning Objectives

This curriculum segment provides practice with the following topics from the *AP Computer Science A Course Description* Topic Outline:

I. **Object-Oriented Program Design** — All of the Object-Oriented Program Design topics.
II. **Program Implementation** — All of the Program Implementation topics, except for II.B.4.e Recursion, and the `Double` class from II.C.
III. **Program Analysis** — All of the Program Analysis topics, except for III.F.1. Pre- and post-conditions, and III.H. Numerical representations and limits.
IV. **Standard Data Structures** — All of the Standard Data Structures.
V. **Standard Algorithms** — All of the standard algorithms except for V.B.2. Binary Search, and V.C. Sorting. However, the similarity of the Selection Shuffle algorithm to that of the Selection Sort is discussed.
VI. **Computing in Context** — These topics are not included.

The activities also introduce students to simulation, a rich and socially relevant context in which to practice programming.

What Is Provided

The students' package includes the following:

- **Elevens Lab Student Guide** — The activities in this curriculum segment.
- **Activity Starter Code** directory — Starter code for the students to complete in lab, organized in subdirectories named **Activity1 Starter Code, … , Activity11 Starter Code**.
  Important — Students should only receive the Activity*N* Starter Code folder that they need for a specific Activity. This is because subsequent Activity*N* Starter Code folders contain the complete working solution code for earlier activities.

The teachers' package includes the following:

- **Elevens Lab Teacher's Guide** — This file.
- **Activity Solution Code** — Solution code for each activity, organized in subdirectories named **Activity1 Solution Code, … , Activity11 Solution Code**.

Installation/Setup

Whatever procedures are already in use for students to work with code provided by the instructor should work.

Student Lab Guide Activity Sections

The Student Lab Guide has the following activity sections:

- Introduction — A brief introduction to the activity.
- Exploration — Detailed information related to the activity.
- Exercises — Writing, running, and/or testing code activities for the student to complete. Students should complete all exercises.
- Questions — Questions related to the topics or techniques that students should have learned from completing the exercises. Variations of the questions would be useful for quizzes or other formative assessments.

Activity Outline

Activity 1: Design and Create a `Card` Class

- Summary: Students think about requirements that various card games impose on cards. Then they complete the implementation of a `Card` class.
- Time Estimate: 1 hour
- Suggested Lesson Plan: Conduct a class discussion on what information a general `Card` class should store, and the resulting implications for program design. Then have the students complete the `Card` class provided. Here are some points that may arise in the discussion:
  - We expect students to come up with rank and suit. Many games like Twenty-One also need some sort of point value assigned to each card (face cards 10, ace 1 or 11).
  - It's useful for a class to have an appropriate `toString` method.
  - Should a `Card` class provide higher-level operations such as "matches" and "outranks"? A client class can implement this on its own, provided that an accessor method for ranks is supplied. Should the `Card` class determine whether one card outranks another or should a client class? Do all games rank cards in the same way? These are interesting points for discussion, but for the set of solitaire games discussed and implemented in this lab, they are not pertinent. Cards in these games are not compared to each other. Our design does include a `matches` method, to see if two cards are identical, for testing purposes only.

Activity 2: Initial Design of a `Deck` Class

- Summary: Students consider the starter code for a general `Deck` class. It is organized to enable reuse in more than one kind of game. A question for discussion is how a deck will know what kind and how many cards it should contain. The design decision for this lab was to have the client of the `Deck` class supply the information necessary to create a deck of cards: the suits, ranks, and point values. Students will complete the `Deck` class, except for the `shuffle` method, and test their implementation.
- Time Estimate: 2 hours
- Suggested Lesson Plan: Have students briefly discuss the design of a deck in small groups. Then have a classroom discussion about our deck design. This discussion should include the relation between `Deck` and `Card`, the three-parameter constructor, and the desire to reuse a `Deck` class for different games

Activity 3: Shuffling the Cards in a Deck

- Summary: Students think about how to shuffle a deck of cards. This activity lists some alternatives for shuffling algorithms, including imitating real-world shuffling. Students are provided with a program that prints the results of shuffling with algorithms that students supply, to get them thinking both about the deficiencies for some of the shuffling alternatives and about the complications of verifying probabilistic behavior.
- Time Estimate: 2 hours
- Suggested Lesson Plan: Conduct a discussion on what makes a good shuffling algorithm, then have students add code for two shuffling methods to the provided `Shuffler` class.

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

- Summary: Students complete the `shuffle` method in the `Deck` class and test it.
- Time Estimate: 1 hour
- Suggested Lesson Plan: This is a lab activity.

Activity 5: Testing with Assertions (Optional)

- Summary: Students are introduced to the Java `assert` statement, and use it to find bugs in versions of the `Deck` class.
- Time Estimate: 2 hours
- Suggested Lesson Plan: This is a lab activity.

Activity 6: Playing Elevens

- Summary: Students are introduced to the Elevens solitaire game. They play a few games (on a GUI-based version we provide), to gather some information about how hard the game is to win.
- Time Estimate: 1 hour
- Suggested Lesson Plan: Lab activity, plus discussion of the analysis exercises.

Activity 7: Elevens Board Class Design

- Summary: Students consider the design of an `ElevensBoard` class.
- Time Estimate: 1 hour

Activity 8: Using an Abstract Board Class

- Summary: Students consider using an abstract class to encapsulate the common state and behaviors for all Elevens-related games. The `ElevensBoard` from Activity 7 will be used to

decide what state and behavior can be moved to the new `Board` class.

- Time Estimate: 1 hour
- Suggested Lesson Plan: First, review the relationship between an abstract class and the corresponding subclass. Then, have your class examine the differences between the Elevens and Thirteens games, and note that the differences in behavior ought to correspond to abstract methods, while the differences in state will correspond to parameters passed to the constructor.

  The difference in board size is implemented merely as a parameter to the constructor. The abstract methods verify a legal selection and check for another possible play. However, there are no abstract methods for actually *making* the card selections from the board, despite the fact that this is one of the features that differ in the various solitaire games. This point may arise in the discussion. The selection of cards is actually done in the `CardGameGUI` class, which is a client of the `Board` class.

Activity 9: Implementing the Elevens Board

- Summary: Students complete the `ElevensBoard` class.
- Time Estimate: 2 hours
- Suggested Lesson Plan: This is a lab activity.

Activity 10: ThirteensBoard (Optional)

- Summary: Students analyze **Thirteens**, another member of the **Elevens** family of solitaire games. They then implement `ThirteensBoard` and `ThirteensGUIRunner`, using `ElevensBoard` and `ElevensGUIRunner` as a starting point.
- Time Estimate: 1 hour
- Suggested Lesson Plan: This is a lab activity.

Activity 11: Simulation of Elevens (Optional)

- Summary: Students are introduced to simulation here, in the context of determining how hard Elevens is to win. A new `public` method is added to `ElevensBoard` to facilitate simulation, and a new `ElevensSimulation` class is added to run the simulation. Additionally, some `ElevensBoard` code is reorganized, and some `private` methods are added to better accommodate the added `ElevensBoard` responsibilities.
- Time Estimate: 2 hours
- Suggested Lesson Plan: This is a lab activity, with students reporting their results.

Answers to Student Lab Guide Questions and Exercises

Activity 1: Design and Create a `Card` Class

Answers to the exercises are found in the **Activity1 Solution Code** folder.

Activity 2: Initial Design of a `Deck` Class

Answers to the exercises are found in the **Activity2 Solution Code** folder.

Answers to the questions:

1. A deck is a collection of cards.

2. The size of the constructed deck is the product of the lengths of ranks and suits, so the answer is 6.

3.
```
String[] ranks = {"2", "3", "4", "5", "6", "7", "8", "9", "10",
                  "jack", "queen", "king", "ace"};
String[] suits = {"spades", "hearts", "diamonds", "clubs"};
int[] pointValues = {2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10, 11};
```

4. Elements of suits may appear in any order. Elements of ranks may be reordered, as ranks are not ordered in Elevens, as long as the `pointValues` elements are reordered in the same way. In card games where rank order is important, the sequence of elements in the `ranks` variable should be "in order."

Activity 3: Shuffling the Cards in a Deck

Answers to the exercises are found in the **Activity3 Solution Code** folder.

Answers to the questions:

1.
```
public static String flip() {
   int r = (int) (Math.random() * 3);
   if (r < 2) {
      return "heads";
   } else {
      return "tails";
   }
}
```

2. **Two method solution (using a helper method):**
```
public static boolean arePermutations(int[] a, int[] b) {
   for (int aValue : a) {
      // Make sure that every element of a is somewhere in b.
      if (!contains(b, aValue) {
```

```
            return false;
         }
      }
      return true;
   }

   private static boolean contains(int[] b, int key) {
      for (int bValue: b) {
         if (bValue == key) {
            return true;
         }
      }
      return false;
   }
```

One method solution:

```
public static boolean arePermutations(int[] a, int[] b) {
   for (int aValue : a) {
      // Make sure that every element of a is somewhere in b.
      boolean found = false;
      for (int bValue : b) {
         if (bValue == aValue) {
            found = true;
         }
      }
      if (!found) {
         return false;
      }
   }
   return true;
}
```

3. The sequence 0, 1, 1. The first 0 switches 4 and 1, producing 4, 2, 3, 1; the first 1 switches 2 and 3, producing 4, 3, 2, 1; and the second 1 switches the 3 with itself.

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

Answers to the exercises are found in the **Activity4 Solution Code** folder.

Activity 5: Testing with Assertions (Optional)

Answers to exercises 1–4 will vary, but they should be consistent with the actual code error. The buggy `Deck.java` files are in the **Activity5 Solution Code/Buggy1, … , Activity5 Solution Code/Buggy4** folders. Each error is marked in the code with a `// BUG` comment. The errors are listed below:

1. `isEmpty`: `return size == 0;` was changed to `return size < 0;`

2. Constructor: `for (int j = 0; j < ranks.length; j++)` was changed to
   `for (int j = 1; j < ranks.length; j++)`

3. `shuffle:` `for (int k = cards.size() - 1; k > 0; k--)` was changed to
   `for (int k = cards.size() - 1; k < 0; k--)`

4. `deal:`
```
if (isEmpty()) {
   return null;
}
size--;
```
was changed to
```
size--;
if (isEmpty()) {
   return null;
}
```

5. The buggy `Deck.java` file in the **Activity5 Solution Code/Buggy5** folder contains all of the errors in 1–4 above. The **Activity5 Solution Code/Correct** folder contains the corrected `Deck.java` file.

Activity 6: Playing Elevens

Answers to the questions:

Question from the **Exploration** section:
Q. Play a few games of Elevens. How many did you win?
A. Students should win about 1 out of 10 games.

1. There are two possible plays from 5♠ 4♥ 2♦ 6♣ A♠ J♥ K♦ 5♣ 2♠ :
   The 5♠ with the 6♣ make 11. The 5♣ with the 6♣ also make 11.

2. The deck and the board satisfy three invariant relations before and after each play.

   - The number of face cards in the deck plus the number of face cards in the board is divisible by 3.
   - The number of jacks, the number of queens, and the number of kings are all equal to each other.
   - The number of nonface cards in the deck plus the number of nonface cards in the board is even.

   Thus, if the deck is empty and the board contains three cards, they must all be face cards. Because there have to be equal numbers of each face card, the three must be the remaining JQK.

   One can also argue by cases as follows.

- If there are three face cards left, there must have been three previous plays of face cards. If the remaining cards are not JQK, then each face card rank wasn't played the same number of times. This can't happen.

- If there are two or one face cards left, we get a contradiction using the same reasoning as in the first case.

- If none of the three cards is a face card, there must have been a play of an odd number of nonface cards earlier in the game, as there are 40 nonface cards in all. This play would have been illegal.

3. The game doesn't involve any strategy. When there is a choice between two or more different plays, it doesn't matter in which order they are played.

Activity 7: Elevens Board Class Design

Answers to the questions:

1. Deck of cards and a list of cards on the board. The `ElevensBoard` class would need `Deck` and `Card[]` instance variables.

2. Answers may vary. One possible answer is:

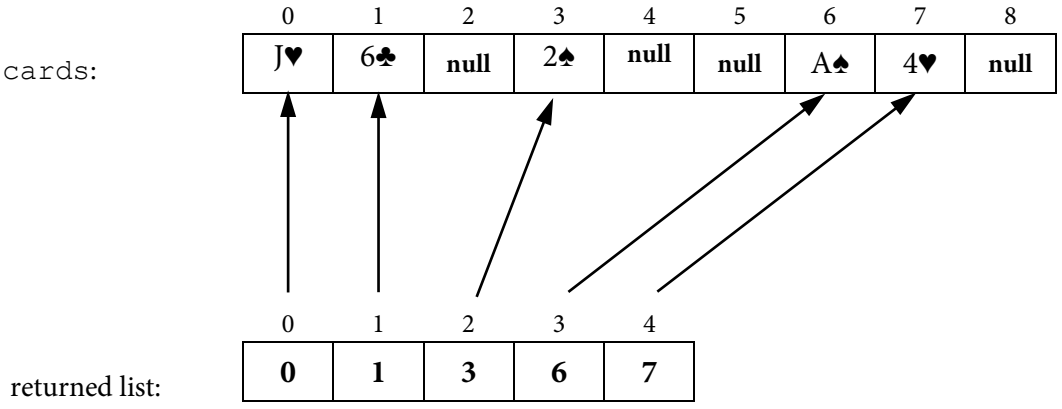
```
Shuffle the deck;
Deal nine cards;
While there is a possible move,
    If a pair exists that sums to 11,
        Remove the pair;
        Replace the two removed cards if possible;
    Else if a triplet that contains J,Q,K exists,
        Remove the triplet;
        Replace the three removed cards is possible;
If there are no cards left on the board, you win, else you lose.
```

3. In the `ElevensBoard` class, as written, there are no methods that actually select the cards to be removed, only ones to check already selected cards.

4. a) The method, `dealMyCards`, is called in the `ElevensBoard` constructor and the `newGame` method.

b) The methods `isLegal` and `anotherPlayIsPossible` should call the `containsPairSum11` and `containsJQK` methods.

c)



d)

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

   for (Integer kObj : cIndexes) {
      int k = kObj.intValue();
      System.out.println(board.cardAt(k));
   }
}
```

e) The method `anotherPlayIsPossible` needs to call the `cardIndexes` method before calling the `containsPairSum11` and `containsJQK` methods. It needs to do this in order to get the indexes of all the cards on the board (non-`null` cards) so that it can check to see if the board contains another pair of cards that sum to 11 or a JQK-triplet.

Activity 8: Using an Abstract Board Class

Answers to the questions:

1. Similarities:
   - They are all single player games.
   - They are all played on a board with cards.
   - The cards are selected for removal based on one of two rules: the cards' point values add up to a specific sum, or there is a specific group of face cards.

Differences:

- The number of cards on the board is different.
- The sums differ.
- The specific groups of face cards differ.

2. The `ElevensBoard` constructor passes to the `Board` constructor the information needed to initialize the instance variables declared in the abstract `Board` class. This is accomplished through the following use of super:

```
super(BOARD_SIZE, RANKS, SUITS, POINT_VALUES);
```

3. No. The `abstract` methods will have to be implemented differently in the Tens and Thirteens games and will need different `private` helper methods to accomplish their tasks.

Activity 9: Implementing the Elevens Board

Answers to the exercises are found in the **Activity9 Solution Code** folder.

Notes to the teacher about the exercises:

Three of the four `ElevensBoard` methods that students implement in this activity have `selectedCards` parameters: `isLegal`, `containsPairSum11`, and `containsJQK`. The elements of these `selectedCards` lists are used as indexes in conjunction with the `Board` `cardAt` accessor method to access the elements of the `Board` `cards` array. However, the `cards` array contains `null` entries when there are fewer than nine cards remaining on the board. These three methods would need to handle these `null` entries if any of the `selectedCards` entries accessed one of these `null` `cards` entries.

However, students do not need to handle this situation in their code because the elements of `selectedCards` can't index `null` `cards` entries:

- The `isLegal` method is only called from the `CardGameGUI` class. `CardGameGUI` constructs the `selectedCards` list based on the card images selected by the player. Since `null` cards can't be selected by the player, the `selectedCards` list passed to `isLegal` will not index any `null` cards.

- The `containsPairSum11` and `containsJQK` methods are only called from `isLegal` and `anotherPlayIsPossible`:

    - Since the `isLegal` method passes on the `selectedCards` list it received from `CardGameGUI`, the `selectedCards` list will not index any `null` cards.

- The `anotherPlayIsPossible` method calls the `Board` `cardIndexes` method to construct the `selectedCards` list that it passes. Since the purpose of the `cardIndexes` method is to create a list of the indexes of all the non-`null` cards, this list will not index any `null` cards.

Answers to questions:

1. The size of the board is just an integer that needs to be passed to the `Board` constructor to size the array that holds the cards on the board.

2. The actual selection of the cards is done by the person playing the game and is "read" by the `CardGameGUIobject` when the player presses the **Replace** button. There is no method, abstract or non-abstract, necessary for selecting cards.

3. Yes, an interface would allow the `CardGameGUI` class to call `isLegal` and `anotherPlayIsPossible` polymorphically**.**

   No, it would not work as well. The `private` instance variables shared by all boards cannot be declared in an interface, and methods cannot be implemented in an interface. All of the boards share behaviors that can be coded without knowing which kind of board it is. If an interface were used instead of an abstract class, each subclass of `Board` would have to declare their own `private` instance variables and rewrite the same implementations for the commonly shared methods.

Activity 10: ThirteensBoard (Optional)

Answers to the exercises are found in the **Activity10 Solution Code** folder.

Activity 11: Simulation of Elevens (Optional)

Answers to the exercises are found in the **Activity11 Solution Code** folder.

Answers to questions from the **Exploration** section:

Q. What do you do repeatedly to play a game?
A. You scan the board for 11-pairs and JQK-triplets.

Q. As you are scanning the cards on the board, what are you trying to find?
A. You're trying to find 11-pairs and JQK-triplets.

Q. Why do you decide to click on a group of cards?
A. Because it's an 11-pair or a JQK-triplet.

A. What happens when you click the **Replace** button?
Q. The cards are removed and replaced with new ones, if the deck has more cards.
Spots are left empty when there are no more cards.

Answers to the questions:

1. Answers will vary, but the percentages should not be consistent. The simulation should usually display percentages in the 0.0 percent to 20.0 percent range. 10.0 percent should be displayed the most often.

2. Yes, the results should be somewhat more consistent. The observed range should be less than the 20 percent seen in the 10-game runs. However, the results from run to run will still vary significantly.

3. Answers will vary, but it takes quite a few runs to achieve fairly consistent results. Even 100,000 game runs will typically have win percentages that vary by a few tenths of a percent.

4. Again, answers will vary. Although the percentage of games won will be much higher for the Thirteens game, it takes runs of many games to achieve reasonably consistent results.

Group Work

Gathering information on card games is a great exercise for the whole class. Subsequent programming in activities 2, 4, 5, 9, and 11 are quite appropriate for solution in partnership. Exercises in Activity 9 and Activity 11 are especially good for a large group or class discussion, even for students who have not completed the exercises.

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() {
```

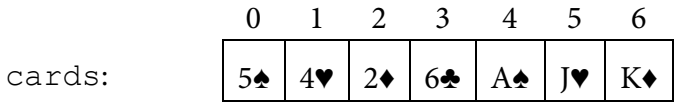
(b) Compare the two versions of the `shuffle` method and answer the following question. Which `shuffle` method is more efficient, the `Deck` class's original `shuffle` method or the `shuffle` method described in part (a)? **Justify your answer**.

(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:

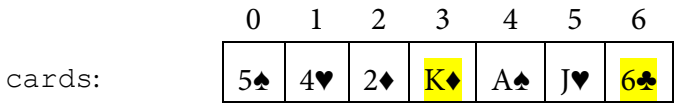
- If the deck is empty, return `null`
- Randomly generate a number that can index a card in the portion of the deck that contains cards that have not been dealt
- Decrement `size`
- Swap the selected card with the card found at position `size`
- Return the selected card

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




```
size: 7
```

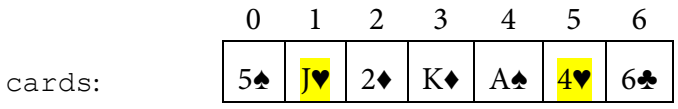
and the number 3 is randomly generated for the index. After the changes, `cards` and `size` would be as follows:




```
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:




```
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() {
```

Sample Exam Question Answers

Answers to multiple-choice questions

1. C — 20 pairs that sum to 11 and 4 triplets of JQK
2. E
3. C
4. D

Answers to free-response questions

1 (a)

```
public void shuffle() {
   size = cards.size();
   List<Card> temp = new ArrayList<Card>();
   for (int j = 0; j < size; j++) {
      int index = (int) (Math.random() * cards.size());
      temp.add(cards.remove(index));
   }
   cards = temp;

}
```

1 (b)

The original `shuffle` method is more efficient than the newly implemented `shuffle` method. The original method runs `size - 1` times and makes 1 swap per loop iteration. The new `shuffle` method removes a card from the list `cards` each loop iteration and adds it to the end of the temporary list. Removing a card from the `cards` list requires shifting higher index elements in the `cards` list to a lower index position. This happens `size` times.

1 (c)

```
public Card deal() {
   if (isEmpty()) {
      return null;
   }
   int index = (int) (Math.random() * size);
   size--;
   Card temp = cards.get(index);       // Get the card to return.
   cards.set(index, cards.get(size)); // Swap the card at the end
   cards.set(size, temp);              // with the card to return.
   return temp;                        // Return the card.
}
```

2 (a)

```
private int findMaxLength() {
   int max = 0;
   for (String str : wordList) {
      if (str.length() > max) {
         max = str.length();
      }
   }
   return max;
}
```

2 (b)

```
private void addCompoundWords(List<String> compoundWords, int letterSum)
{
   for (int j = 0; j < wordList.size() - 1; j++) {
      String w1 = wordList.get(j);
      for (int k = j + 1; k < wordList.size(); k++) {
         String w2 = wordList.get(k);

         int numLetters = w1.length() + w2.length();

         if (numLetters == letterSum) {
            String word1 = w1 + w2;
            String word2 = w2 + w1;
            if (inDictionary(word1)) {
               compoundWords.add(word1);
            }
            if (inDictionary(word2)) {
               compoundWords.add(word2);
            }
         }
      }
   }
}
```

2 (c)

```
public List<String> buildWords() {
   List<String> compoundWords = new ArrayList<String>();
   int max = findMaxLength();
   for (int j = 3; j <= max; j++) {
      addCompoundWords(compoundWords, j);
   }
   return compoundWords;
}
```