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.

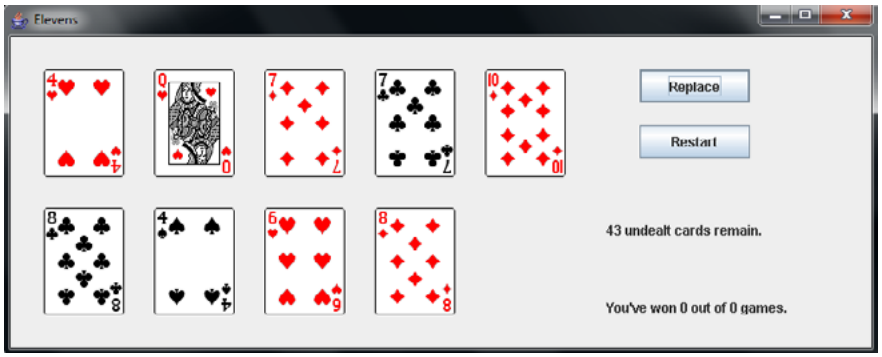


Table of Contents

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?

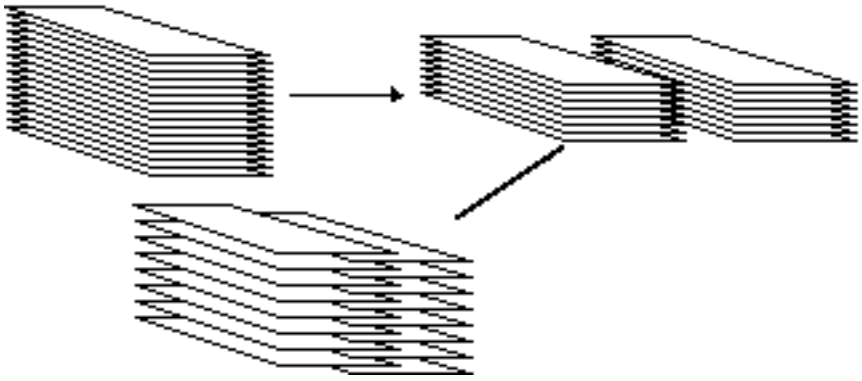
Exploration:

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 k^{th} 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♣	<u>7♦</u>	2♥	<u>Q♠</u>	3♥	5♠	5♦	remove 2♣ (either 2 would work) and 9♦
<u>A♠</u>	10♦	<u>9♣</u>	7♦	2♥	<u>7♣</u>	3♥	5♠	5♦	remove J♣ Q♠ K♠
A♠	10♦	<u>10♠</u>	7♦	<u>3♣</u>	7♣	3♥	5♠	5♦	remove 9♣ and 2♥ (removing A♠ and 10♦ would have been legal here too)
<u>2♠</u>	10♦	<u>9♠</u>	7♦	3♣	7♣	3♥	5♠	5♦	remove A♠ and 10♠ (10♦ could have been removed instead)
<u>A♣</u>	10♦	<u>K♦</u>	7♦	3♣	7♣	3♥	5♠	5♦	remove 2♠ and 9♠
<u>6♦</u>	<u>K♣</u>	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.

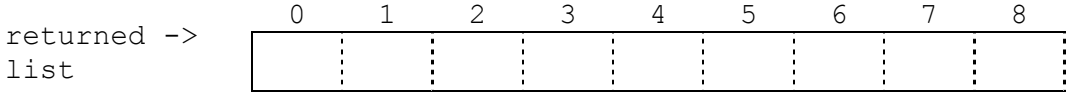
3. Now examine the partially implemented `ElevensBoard.java` file found in the **Activity7 Starter Code** directory. Does the `ElevensBoard` class contain all the state and behavior necessary to play the game?

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.



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

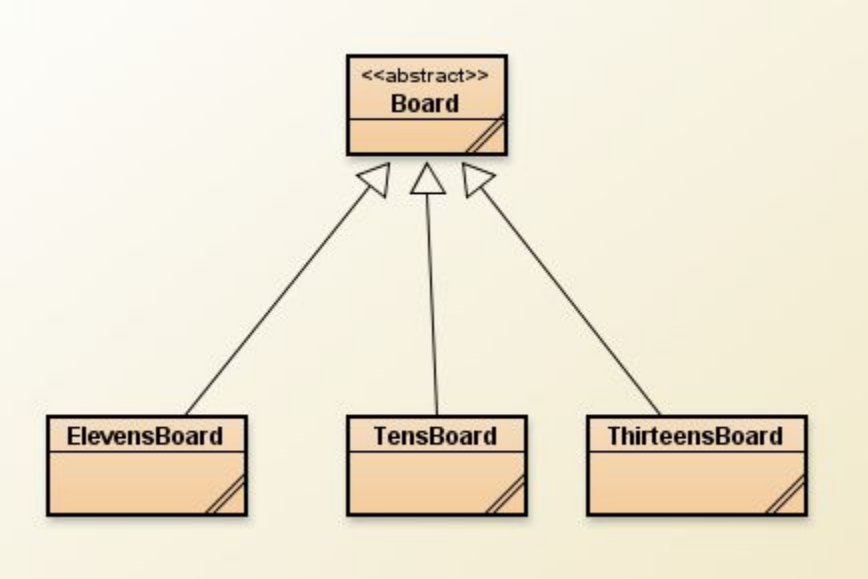
In reading the descriptions of Elevens and its related games, it is evident that these games share common state and behaviors. Each game requires:

- 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).