Part 10

Other useful techniques

Learning Objectives

- You understand the traditional for-loop.
- You understand the issues related to string concatenation and know how to avoid them with the StringBuilder class.
- You understand regular expressions and can write your own ones.
- You understand enumerated (enum) types and know when to use them.
- You know how to use an iterator to go through collections of data.

We'll now take a look at some useful programming techniques and classes.

StringBuilder

Let's look at the following program.

```
String numbers = "";
for (int i = 1; i < 5; i++) {
    numbers = numbers + i;
}
System.out.println(numbers);
```

Sample output

```
1234
```

The program structure is straightforward. A string containing the number 1234 is created, and the string is then outputted.

The program works, but there is a small problem invisible to the user. Calling `numbers + i` creates a *new* string. Let's inspect the program line-

by-line with the repetition block unpacked.

```
String numbers = ""; // creating a new string: ""
int i = 1;
numbers = numbers + i; // creating a new string: "1"
i++;
numbers = numbers + i; // creating a new string: "12"
i++;
numbers = numbers + i; // creating a new string: "123"
i++;
numbers = numbers + i; // creating a new string: "1234"
i++;

System.out.println(numbers); // printing the string
```

In the previous example, five strings are created in total.

Let's look at the same program where a new line is added after each number.

```
String numbers = "";
for (int i = 1; i < 5; i++) {
    numbers = numbers + i + "\n";
}
System.out.println(numbers);
```

Sample output

```
1
2
3
4
```

Each +-operation forms a new string. On the line `numbers + i + "\n";` a string is first created, after which another string is created joining a new line onto the previous string. Let's write this out as well.

```
String numbers = ""; // creating a new string: ""
int i = 1;
// first creating the string "1" and then the string "1\n"
numbers = numbers + i + "\n";
i++;
```

```
// first creating the string "1\n2" and then the string "1\n2\n"
numbers = numbers + i + "\n"
i++;
// first creating the string "1\n2\n3" and then the string "1\n2\n3\n"
numbers = numbers + i + "\n"
i++;
// and so on
numbers = numbers + i + "\n"
i++;

System.out.println(numbers); // outputting the string
```

In the previous example, a total of nine strings is created.

String creation - although unnoticeable at a small scale - is not a quick operation. Space is allocated in memory for each string where the string is then placed. If the string is only needed as part of creating a larger string, performance should be improved.

Java's ready-made StringBuilder class provides a way to concatenate strings without the need to create them. A new StringBuilder object is created with a `new StringBuilder()` call, and content is added to the object using the overloaded `append` method, i.e., there are variations of it for different types of variables. Finally, the StringBuilder object provides a string using the `toString` method.

In the example below, only one string is created.

```
StringBuilder numbers = new StringBuilder();
for (int i = 1; i < 5; i++) {
    numbers.append(i);
}
System.out.println(numbers.toString());
```

Using StringBuilder is more efficient than creating strings with the + operator.

Quiz:

How many strings?

Points:

1/1

How many strings does the following program create?

```
ArrayList<String> words = new ArrayList<>();
words.add("first");
words.add("second");
words.add("third");

String connectedString = "";
for (int i = 0; i < words.size(); i++) {
    connectedString = connectedString + words.get(i);
}
System.out.println(connectedString);
```

Select the correct answer

- 3
- 4
- 5
- 6
- ✓ 7

The answer is correct

Answered Tries remaining: 1

Quiz:
How many strings this time?

Points:
1/1

How many strings does the following program create?

```
ArrayList<String> words= new ArrayList<>();
words.add("first");
words.add("second");
words.add("third");

StringBuilder connectedString = new StringBuilder();
for (int i = 0; i < words.size(); i++) {
    connectedString.append(words.get(i));
}
System.out.println(connectedString.toString());
```

Select the correct answer

- 3
- ✓ 4
- 5
- 6
- 7

The answer is correct

Answered Tries remaining: 2

Regular Expressions

A regular expression defines a set of strings in a compact form. Regular expressions are used, among other things, to verify the correctness of strings. We can assess whether or not a string is in the desired form by using a regular expression that defines the strings considered correct.

Let's look at a problem where we need to check if a student number entered by the user is in the correct format. A student number should begin with "01" followed by 7 digits between 0–9.

You could verify the format of the student number, for instance, by going through the character string representing the student number using the `charAt` method. Another way would be to check that the first character is "0" and call the `Integer.valueOf` method to convert the string to a number. You could then check that the number returned by the `Integer.valueOf` method is less than 20000000.

Checking correctness with the help of regular expressions is done by first defining a suitable regular expression. We can then use the `matches` method of the `String` class, which checks whether the string matches the regular expression given as a parameter. For the student number, the appropriate regular expression is `"01[0-9]{7}"`, and checking the student number entered by a user is done as follows:

```
System.out.print("Provide a student number: ");
String number = scanner.nextLine();

if (number.matches("01[0-9]{7}")) {
    System.out.println("Correct format.");
} else {
    System.out.println("Incorrect format.");
}
```

Let's go through the most common characters used in regular expressions.

Alternation (Vertical Line)

A vertical line indicates that parts of a regular expressions are optional. For example, `00|111|0000` defines the strings `00`, `111` and `0000`. The `respond` method returns `true` if the string matches any one of the specified group of alternatives.

```
String string = "00";

if (string.matches("00|111|0000")) {
    System.out.println("The string contained one of the three alternatives");
```

```
} else {
    System.out.println("The string contained none of the alternatives");
}
```

Sample output

```
The string contained one of the three alternatives
```

The regular expression `00|111|0000` demands that the string is exactly it specifies it to be - there is no *"contains"* functionality.

```
String string = "1111";

if (string.matches("00|111|0000")) {
    System.out.println("The string contained one of the three alternatives");
} else {
    System.out.println("The string contained none of the three alternatives");
}
```

Sample output

```
The string contained none of the three alternatives
```

Affecting Part of a String (Parentheses)

You can use parentheses to determine which part of a regular expression is affected by the rules inside the parentheses. Say we want to allow the strings `00000` and `00001`. We can do that by placing a vertical bar in between them this way `00000|00001`. Parentheses allow us to limit the option to a specific part of the string. The expression `0000(0|1)` specifies the strings `00000` and `00001`.

Similarly, the regular expression `car(|s|)` defines the singular (car) and plural (cars) forms of the word car.

Quantifiers

What is often desired is that a particular sub-string is repeated in a string. The following expressions are available in regular expressions:

- The quantifier `*` repeats 0 ... times, for example;

```
String string = "trololololololo";

if (string.matches("trolo(lo)*")) {
    System.out.println("Correct form.");
} else {
    System.out.println("Incorrect form.");
}
```

Sample output

Correct form.

- The quantifier `+` repeats 1 ... times, for example;

```
String string = "trololololololo";

if (string.matches("tro(lo)+")) {
    System.out.println("Correct form.");
} else {
    System.out.println("Incorrect form.");
}
```

Sample output

Correct form.

```
String string = "nanananananananana Batmaan!";

if (string.matches("(na)+ Batmaan!")) {
    System.out.println("Correct form.");
} else {
    System.out.println("Incorrect form.");
}
```

Sample output

Correct form.

- The quantifier `?` repeats 0 or 1 times, for example:

```
String string = "You have to accidentally the whole meme";

if (string.matches("You have to accidentally (delete )?the whole meme")) {
    System.out.println("Correct form.");
} else {
    System.out.println("Incorrect form.");
}
```

Sample output

```
Correct form.
```

- The quantifier **{a}** repeats a times, for example:

```
String string = "1010";

if (string.matches("(10){2}")) {
    System.out.println("Correct form.");
} else {
    System.out.println("Incorrect form.");
}
```

Sample output

```
Correct form.
```

- The quantifier **{a,b}** repeats a ... b times, for example:

```
String string = "1";

if (string.matches("1{2,4}")) {
    System.out.println("Correct form.");
} else {
    System.out.println("Incorrect form.");
}
```

Sample output

```
Incorrect form.
```

- The quantifier **{a,}** repeats a ... times, for example:

```
String string = "11111";

if (string.matches("1{2,}")) {
    System.out.println("Correct form.");
} else {
    System.out.println("Incorrect form.");
}
```

Sample output

```
Correct form.
```

You can use more than one quantifier in a single regular expression. For example, the regular expression `5{3}(1|0)*5{3}` defines strings that begin and end with three fives. An unlimited number of ones and zeros are allowed in between.

Character Classes (Square Brackets)

A character class can be used to specify a set of characters in a compact way. Characters are enclosed in square brackets, and a range is indicated with a dash. For example, `[145]` means `(1|4|5)` and `[2-36-9]` means `(2|3|6|7|8|9)`. Similarly, the entry `[a-c]*` defines a regular expression that requires the string to contain only `a`, `b` and `c`.

Quiz:
Regular expression

Points:
1/1

Which string would the following regular expression accept?

```
String regex = "ab(ba)+";

String word = /* word here*/;

if (word.matches(regex)) {
    System.out.println("Correct");
}
```

Select the correct answer

babab

abbbbbb

ababab

✓ abba

aaaaaaaa

bb

The answer is correct

Answered Tries remaining: 2

Programming exercise:

Regular expressions (3 parts)

Points 3/3

NB! By submitting a solution to a part of an exercise which has multiple parts, you can get part of the exercise points. You can submit a part by using the 'submit' button on NetBeans. More on the programming exercise submission instructions: Exercise submission instructions.

Let's practice using regular expressions a little. The methods in this exercise should be created in the class `Checker`.

Part 1: Day of week

Use regular expressions to create the method `public boolean isDayOfWeek(String string)`, which returns `true` if the parameter string is an abbreviation of a day of the week (mon, tue, wed, thu, fri, sat, sun)

Example outputs of a program that uses the method:

Sample output

Enter a string: tue
The form is correct.

Sample output

Enter a string: abc
The form is incorrect.

Part 2: Vowel check

NB. For simplicity's sake, in this exercises **the letters that are considered vowels are: a, e, i, o, and u**.

Create the method `public boolean allVowels(String string)` that uses a regular expression to check whether all the characters in the parameter string are vowels.

Example outputs of a program that uses the method:

Sample output

Enter a string: oi
The form is correct.

Sample output

Enter a string: queue
The form is incorrect.

Part 3: Time of day

Regular expressions come in handy in certain situations. In some cases the expressions become too complex, and the "correctness" of the string is best checked with some other style. Or it could be beneficial to use regular expressions for only some part of the check.

Create the method `public boolean timeOfDay(String string)`. It should use a regular expression to check whether the parameter string expresses a time of day in the form `hh:mm:ss` (hours, minutes, and seconds each always take up two spaces).

NB. In this exercise we use the 24-hour clock. So the acceptable values are between 00:00:00 and 23:59:59.

Example outputs of a program that uses the method:

Sample output

```
Enter a string: 17:23:05
The form is correct.
```

Sample output

```
Enter a string: abc
The form is incorrect.
```

Sample output

```
Enter a string: 33:33:33
The form is incorrect.
```

Exercise submission instructions

How to see the solution

Almost all programming languages support regular expressions nowadays. The theory of regular expressions is one of the topics

considered in the course Computational Models (TKT-20005). You can find more regular expressions by googling *regular expressions java*, for instance.

Enumerated Type - Enum

If we know the possible values of a variable in advance, we can use a class of type `enum`, i.e., *enumerated type* to represent the values. Enumerated types are their own type in addition to being normal classes and interfaces. An enumerated type is defined by the keyword `enum`. For example, the following `Suit` enum class defines four constant values: `DIAMOND`, `SPADE`, `CLUB` and `HEART`.

```
public enum Suit {
    DIAMOND, SPADE, CLUB, HEART
}
```

In its simplest form, `enum` lists the constant values it declares, separated by a comma. Enum types, i.e., constants, are conventionally written with capital letters.

An Enum is (usually) written in its own file, much like a class or interface. In NetBeans, you can create an Enum by selecting *new/other/java/java enum* from project.

The following is a `Card` class where the suit is represented by an enum:

```
public class Card {

    private int value;
    private Suit suit;

    public Card(int value, Suit suit) {
        this.value = value;
        this.suit = suit;
    }

    @Override
    public String toString() {
        return suit + " " + value;
    }

    public Suit getSuit() {
```

```
        return suit;
    }

    public int getValue() {
        return value;
    }
}
```

The card is used in the following way:

```
Card first = new Card(10, Suit.HEART);

System.out.println(first);

if (first.getSuit() == Suit.SPADE) {
    System.out.println("is a spade");
} else {
    System.out.println("is not a spade");
}
```

The output:

Sample output

```
HEARTS 10
is not a spade
```

We see that the Enum values are outputted nicely! Oracle has a site related to the `enum` data type at http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html .

Comparing Enums

In the example above, two enums were compared with equal signs. How does this work?

Each enum field gets a unique number code, and they can be compared using the equals sign. Just as other classes in Java, these values also inherit the Object class and its equals method. The equals method compares this numeric identifier in enum types too.

The numeric identifier of an enum field value can be found with `ordinal()`. The method actually returns an order number - if the

enum value is presented first, its order number is 0. If its second, the order number is 1, and so on.

```
public enum Suits {
    DIAMOND, CLUB, HEART, SPADE
}
```

```
System.out.println(Suit.DIAMOND.ordinal());
System.out.println(Suit.HEART.ordinal());
```

Sample output

```
0
3
```

Object References In Enums

Enumerated types may contain object reference variables. The values of the reference variables should be set in an internal constructor of the class defining the enumerated type, i.e., within a constructor having a `private` access modifier. Enum type classes cannot have a `public` constructor.

In the following example, we have an enum `Color` that contains the constants RED, GREEN and BLUE. The constants have been declared with object reference variables referring to their color codes:

```
public enum Color {
    // constructor parameters are defined as
    // the constants are enumerated
    RED("#FF0000"),
    GREEN("#00FF00"),
    BLUE("#0000FF");

    private String code;        // object reference variable

    private Color(String code) { // constructor
        this.code = code;
    }
```

```
    public String getCode() {
        return this.code;
    }
}
```

The enum `Color` can be used like so:

```
System.out.println(Color.GREEN.getCode());
```

Sample output

```
#00FF00
```

Iterator

Let's look at the following `Hand` class that represents the set of cards that a player is holding:

```
public class Hand {
    private List<Card> cards;

    public Hand() {
        this.cards = new ArrayList<>();
    }

    public void add(Card card) {
        this.cards.add(card);
    }

    public void print() {
        this.cards.stream().forEach(card -> {
            System.out.println(card);
        });
    }
}
```

The `print` method of the class prints each card in the current hand.

ArrayList and other "object containers" that implement the *Collection* interface implement the *Iterable* interface, and they can also be iterated over with the help of an *iterator* - an object specifically designed to go

through a particular type of object collection. The following is a version of printing the cards that uses an iterator:

```
public void print() {
    Iterator<Card> iterator = cards.iterator();

    while (iterator.hasNext()) {
        System.out.println(iterator.next());
    }
}
```

The iterator is requested from the `cards` list containing cards. The iterator can be thought of as a "finger" that always points to a particular object inside the list. Initially it points to the first item, then to the next, and so on... until all the objects have been gone through with the help of the "finger".

The iterator offers a few methods. The `hasNext()` method is used to ask if there are any objects still to be iterated over. If there are, the next object in line can be requested from the iterator using the `next()` method. This method returns the next object in line to be processed and moves the iterator, or "finger", to point to the following object in the collection.

The object reference returned by the iterator's next method can of course also be stored in a variable. As such, the `print` method could also be written in the following way.

```
public void print(){
    Iterator<Card> iterator = cards.iterator();

    while (iterator.hasNext()) {
        Card nextInLine = iterator.next();
        System.out.println(nextInLine);
    }
}
```

Let's now consider a use case for an iterator. We'll first approach the issue problematically to provide motivation for the coming solution. We attempt to create a method that removes cards from a given stream with a value lower than the given value.

```
public class Hand {
    // ...

    public void removeWorst(int value) {
        this.cards.stream().forEach(card -> {
            if (card.getValue() < value) {
                cards.remove(card);
            }
        });
    }
}
```

Executing the method results in an error.

Sample output

```
Exception in thread "main" java.util.ConcurrentModificationException
at ...
Java Result: 1
```

The reason for this error lies in the fact that when a list is iterated over using the forEach method, it's assumed that the list is not modified during the traversal. Modifying the list (in this case deleting elements) causes an error - we can think of the forEach method as getting "confused" here.

If you want to remove some of the objects from the list during a traversal, you can do so using an iterator. Calling the `remove` method of the iterator object neatly removes form the list the item returned by the iterator with the previous `next` call. Here's a working example of the version of the method:

```
public class Hand {
    // ...

    public void removeWorst(int value) {
        Iterator<Card> iterator = cards.iterator();

        while (iterator.hasNext()) {
            if (iterator.next().getValue() < value) {
                // removing from the list the element returned by the previous
                iterator.remove();
            }
        }
```

```
}
```

Programming exercise:

Enum and Iterator (4 parts)

Points 4/4

NB! By submitting a solution to a part of an exercise which has multiple parts, you can get part of the exercise points. You can submit a part by using the 'submit' button on NetBeans. More on the programming exercise submission instructions: Exercise submission instructions.

Let's implement a program for managing employee data of a small company.

Part 1: Education

Make an enumerated type (enum) `Education`. It should have the enumerators `PHD` (Doctoral degree), `MA` (Masters degree), `BA` (Bachelors degree) and `HS` (High School diploma).

Part 2: Person

Make a class `Person`. The Person constructor takes a name and the education as parameters. A Person has a method `public Education getEducation()`, which returns the education of the person. A Person also has a `toString` -method which works as follows:

```
Person anna = new Person("Anna", Education.PHD);
System.out.println(anna);
```

Sample output

```
Anna, PHD
```

Part 3: Employees

Make a class `Employees`. Employees -object contains a list of Person -objects. The class has a constructor which takes no parameters, and the following methods:

- `public void add(Person personToAdd)` adds the given person to the employees list
- `public void add(List<Person> peopleToAdd)` adds the given list of people to the employees list
- `public void print()` prints all employees
- `public void print(Education education)` prints the employees whose education matches the education given as a parameter.

NB: The `print` method of the `Employees` class must be implemented using an iterator!

Part 4: Firing an employee

Make a method `public void fire(Education education)` for the `Employees` class. The method removes all employees whose education matches the education given as parameter from the employees list.

NB: Implement the method using an iterator!

See an example of using the class below:

```
Employees university = new Employees();
university.add(new Person("Petrus", Education.PHD));
university.add(new Person("Arto", Education.HS));
university.add(new Person("Elina", Education.PHD));

university.print();

university.fire(Education.HS);

System.out.println("==");

university.print();
```

Prints:

Sample output

Part 5: Petrus, PHD
Arto, HS
Elina, PHD

Petrus, PHD
Elina, PHD

Exercise submission instructions

How to see the solution

Programming exercise:
Sort them cards! (6 parts)

Points
6/6

NB! By submitting a solution to a part of an exercise which has multiple parts, you can get part of the exercise points. You can submit a part by using the 'submit' button on NetBeans. More on the programming exercise submission instructions: Exercise submission instructions.

The exercise template has a class that represents a playing card. Each card has a value and a suit. A card's value is represented as a number *2, 3, ..., 14* and its suit as *Club, Diamond, Heart* or *Spade*. Ace's value is 14. The value is represented with an integer, and the suit as an enum. Cards also have a method toString, which can be used to print the value and the suit in a readable form.

New cards can be created like this:

```
Card first = new Card(2, Suit.DIAMOND);
Card second = new Card(14, Suit.SPADE);
Card third = new Card(12, Suit.HEART);

System.out.println(first);
```

```
System.out.println(second);
System.out.println(third);
```

The output:

Sample output

```
DIAMOND 2
SPADE A
HEART Q
```

Part 1: Comparable Card class

Change the Card class to be `Comparable`. Implement the `compareTo` method so that using it sorts the cards in ascending order based on their value. If the cards being compared have the same value, they are sorted by *club first, diamond second, heart third, and spade last.*

Reading Ordinal method of Enum will help you out in sorting the cards by their suit.
So, for this sorting, the least valuable card is two of clubs and most valuable card is the ace of spades.

Part 2: Hand

Create a class `Hand` to represent the cards in a player's hand. Add the following methods to the class:

- `public void add(Card card)` adds a card to the hand
- `public void print()` prints the cards in hand as shown in the example below

```
Hand hand = new Hand();

hand.add(new Card(2, Suit.DIAMOND));
hand.add(new Card(14, Suit.SPADE));
hand.add(new Card(12, Suit.HEART));
hand.add(new Card(2, Suit.SPADE));

hand.print();
```

Outputs:

Sample output

```
DIAMOND 2
SPADE A
HEART Q
SPADE 2
```

Use an ArrayList to store the cards.

Part 3: Sorting the hand

Add a method `public void sort()` to the Hand class, which sorts the cards in the hand. After sorting, the cards are printed in order:

```
Hand hand = new Hand();

hand.add(new Card(2, Suit.DIAMOND));
hand.add(new Card(14, Suit.SPADE));
hand.add(new Card(12, Suit.HEART));
hand.add(new Card(2, Suit.SPADE));

hand.sort();

hand.print();
```

Output:

Sample output

```
DIAMOND 2
SPADE 2
HEART Q
SPADE A
```

Part 4: Comparing hands

In a card game, hands are ranked based on the sum of values of its cards. Modify the `Hand` class to be comparable based on this criteria, i.e. change the class so that interface `Comparable<Hand>` applies to it.

Here's an example of a program that compares the hands:

```
Hand hand1 = new Hand();

hand1.add(new Card(2, Suit.DIAMOND));
hand1.add(new Card(14, Suit.SPADE));
hand1.add(new Card(12, Suit.HEART));
hand1.add(new Card(2, Suit.SPADE));

Hand hand2 = new Hand();

hand2.add(new Card(11, Suit.DIAMOND));
hand2.add(new Card(11, Suit.SPADE));
hand2.add(new Card(11, Suit.HEART));

int comparison = hand1.compareTo(hand2);

if (comparison < 0) {
    System.out.println("better hand is");
    hand2.print();
} else if (comparison > 0){
    System.out.println("better hand is");
    hand1.print();
} else {
    System.out.println("hands are equal");
}
```

Output

Sample output

```
better hand is
DIAMOND J
SPADE J
HEART J
```

Part 5: Sorting cards with different criteria

What if we want to sort the cards in different ways, e.g. sorting all the cards of the same suit in a row. A class can only have one `compareTo` method, so we'll need something else to sort the cards in to a different order.

Alternative sorting systems are possible through different sorting classes. Such a class must have the `Comparator<Card>` interface. An object of the sorting class will then compare two cards give as parameters. The class only has one method, compare(Card c1, Card c2), which returns a negative value if the card c1 should be sorted before card c2, a positive value if card c2 comes before card c1, and zero if they are equal.

The idea is to create a different sorting class for each different way of sorting the cards, e.g. cards of the same suit in a row.:

```
import java.util.Comparator;

public class SortBySuit implements Comparator<Card> {
    public int compare(Card c1, Card c2) {
        return c1.getSuit().ordinal() - c2.getSuit().ordinal();
    }
}
```

When sorting the cards by suit, use the same order as with the `compareTo` method: *clubs first, diamonds second, hearts third, spades last.*

Sorting still works with the sort method of Collections class. As its other parameter, the method now receives the object that has the sorting logic.

```
ArrayList<Card> cards = new ArrayList<>();

cards.add(new Card(3, Suit.SPADE));
cards.add(new Card(2, Suit.DIAMOND));
cards.add(new Card(14, Suit.SPADE));
cards.add(new Card(12, Suit.HEART));
cards.add(new Card(2, Suit.SPADE));

SortBySuit sortBySuitSorter = new SortBySuit();
Collections.sort(cards, sortBySuitSorter);

cards.stream().forEach(c -> System.out.println(c));
```

Output:

Sample output

```
DIAMOND 2
HEART Q
SPADE 3
SPADE A
SPADE 2
```

The sorting object can also be created directly when sort method is called.

```
Collections.sort(cards, new SortBySuit());
```

It can even be done with a lambda function, without ever creating the sorting class.

```
Collections.sort(cards, (c1, c2) -> c1.getSuit().ordinal() - c2.getSuit().c
```

You can learn more about creating sorting classes here.

Now, create a class `BySuitInValueOrder` class that has the Comparator interface, which sorts the cards in the same order as in the above example, except that now the cards are sorted by value inside their suit.

Part 6: Sorting the hand by suit

Add a method `public void sortBySuit()` to class `Hand`. When the method is called, it sorts the cards in the hand with the same logic as in the previous part. After being sorted, the cards are printed in the following order:

```
Hand hand = new Hand();

hand.add(new Card(12, Suit.HEART));
hand.add(new Card(4, Suit.SPADE));
hand.add(new Card(2, Suit.DIAMOND));
hand.add(new Card(14, Suit.SPADE));
hand.add(new Card(7, Suit.HEART));
```

```
hand.add(new Card(2, Suit.SPADE));

hand.sortBySuit();

hand.print();
```

Output:

Sample output

```
DIAMOND 2
HEART 7
HEART Q
SPADE 2
SPADE 4
SPADE A
```

Exercise submission instructions

How to see the solution

You have reached the end of this section! Continue to the next section:

→ 4. Summary

Remember to check your points from the ball on the bottom-right corner of the material!

In this part:

Source code of the material

This course is created by the Agile Education Research -research group of the University of Helsinki.

Credits and about the material.

HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

MASSIIVISET AVOIMET VERKKOKURSSIT
MASSIVE OPEN ONLINE COURSES · MOOC.FI