Exercises for Live Coding of “Java Advanced - programming” module

www.sdacademy.pl

Exercise 1.

Create a method that takes a string list as a parameter, then returns the list sorted alphabetically from Z to A.

Exercise 2.

Create a method that takes a string list as a parameter, then returns that list sorted alphabetically from Z to A case-insensitive.

Exercise 3.

Create a method that takes the map as a parameter, where the key is string and the value number, and then prints each map element to the console in the format: Key: <k>, Value: <v>. There should be a comma at the end of every line except the last, and a period at the last.

Example:

Key: Java, Value: 18**,**
Key: Python, Value: 1**,**
…
Key: PHP, Value: 0**.**

Exercise 4.

Create a **Storage** class that will have a private **Map** field, a public constructor, and methods:

- **addToStorage(String key, String value)** → adding elements to the storage
- **printValues(String key)** → displaying all elements under a given key
- **findValues(String value)** → displaying all keys that have a given value

The **Storage** class should allow you to store multiple values under one key.

Exercise 5.

Implement the **SDAHashSet<E>** class that will implement the **HashSet<E>** logic. It should support methods:

- add
- remove
- size
- contains
- clear

Exercise 6.

Create a method that accepts **TreeMap** and prints the first and last **EntrySet** in the console.

software **development** academy

Exercise 7.

Create a class imitating a weapon magazine. The class should be able to define the size of the magazine using the constructor. Implement the methods:

loadBullet(String bullet) → adding a cartridge to the magazine, does not allow loading more cartridges than the capacity of the magazine

isLoaded() → returns information about whether the weapon is loaded (at least one cartridge) or not

shot() → each call shots one bullet (prints string value in console) - the last loaded cartridge - and prepares the next one, loaded before the last one, if there are no more cartridges, it prints "empty magazine" in the console

Exercise 8.

Implement the **Validator** interface, which will include a **boolean validate(Parcel input)** method in its declaration. Create a **Parcel** class with the parameters:

- int **xLength**
- int **yLength**
- int **zLength**
- float **weight**
- boolean **isExpress**

The validator should verify that the sum of the dimensions (x, y, z) does not exceed 300, whether each size is not less than 30, whether the weight does not exceed 30.0 for isExpress = false or 15.0 for isExpress = true. In case of errors, it should inform the user about them.

Exercise 9.

Create a **Point2D** class with fields **double x**, **double y**, getters, setters and constructor. Then create a **Circle** class that will have a constructor:

Circle(Point2D center, Point2D point)

The first parameter specifies the center of the circle, the second is any point on the circle. Based on these points, the **Circle** class is to determine:

- circle radius when calling double **getRadius()** method
- circle circumference when calling double **getPerimeter()** method
- circle area when calling double **getArea()** method
- * (challenging) three points on the circle every 90 degrees from the point given when calling the **List<Point2D> getSlicePoints()** method

Exercise 10.

Create a **MoveDirection** class with fields **double x**, **double y** as well as getters, setters and constructor.
Create a **Movable** interface with the **move(MoveDirection moveDirection)** method.

Implement the interface in the classes from the previous task (**Point2D** and **Circle**). When the **move(MoveDirection moveDirection)** method is called, the objects are to change their position based on the provided direction (**MoveDirection**).

Validate the offset by calling the other **Circle** methods.

Exercise 11.

Create a **Resizable** interface with the **resize(double resizeFactor)** method.

Implement the interface in the class from the previous task (**Circle**). When calling the **resize(double resizeFactor)** method, the circle should change its size by a given factor (1.5, 0.5, 10.0, etc.).

Validate the resizing by calling the other **Circle** methods.

Exercise 12.

Create a **Manufacturer** class that will contain fields: name, year of establishment, country. Include all necessary methods and constructor parameters. Implement the **hashCode()** and **equals()** methods.

Create a **Car** class that will contain fields: name, model, price, year of manufacture, manufacturer list (**Manufacturer**), and engine type (represented as the enum class, e.g. V12, V8, V6, S6, S4, S3). Include all necessary methods and constructor parameters. Implement the **hashcode()** and **equals()** methods.

Exercise 13.

Create a **CarService** class that will contain a list of cars and implement the following methods:

1. adding cars to the list,
2. removing a car from the list,
3. returning a list of all cars,
4. returning cars with a V12 engine,
5. returning cars produced before 1999,
6. returning the most expensive car,
7. returning the cheapest car,
8. returning the car with at least 3 manufacturers,
9. returning a list of all cars sorted according to the passed parameter: ascending / descending,
10. checking if a specific car is on the list,
11. returning a list of cars manufactured by a specific manufacturer,
12. returning the list of cars manufactured by the manufacturer with the year of establishment <,>, <=,> =, =,! = from the given.

Exercise 14.

Implement the following functionalities based on 100,000 element arrays with randomly selected values:

1. return a list of unique items,
2. return a list of elements that have been repeated at least once in the generated array,
3. return a list of the 25 most frequently recurring items.

Implement a method that deduplicates items in the list. If a duplicate is found, it replaces it with a new random value that did not occur before. Check if the method worked correctly by calling method number 2.

Exercise 15.

Create a **Car** enum class with FERRARI, PORSCHE, MERCEDES, BMW, OPEL, FIAT, TOYOTA etc. constants. Each vehicle has its own parameters, e.g. price, power, etc. Enum should contain **boolean isPremium()** and **boolean isRegular()** methods. The **isPremium()** method should return the opposite result to the call of the **isRegular()** method.

In addition, the **boolean isFasterThan()** method should be declared and implemented as part of the enum class. This method should accept the **Car** type object and display information that the indicated vehicle is faster or not than the vehicle provided in the argument. To do this, use the **compareTo()** method.

Exercise 16.

Create an **enum Runner** class with constants BEGINNER, INTERMEDIATE, ADVANCED. Enum should have two parameters:

- minimum time of running a marathon in minutes
- maximum running time of the marathon in minutes

In addition, the **Runner** enum should contain the static **getFitnessLevel()** method, which takes any time result of a marathon run, and as a result should return a specific **Runner** object based on the given time.

Exercise 17.

Create a **ConversionType** enum class with the constants METERS_TO_YARDS, YARDS_TO_METERS, CENTIMETERS_TO_ICHES, INCHES_TO_CENTIMETERS, KILOMETERS_TO_MILES, MILES_TO_KILOMETERS. Enum should have a **Converter** type parameter used to perform calculations for a given type.

Then create a **MeasurementConverter** class that will have the **convert(int value, ConvertionType conversionType)** method and based on the value and type of conversion, used the **Converter** of the given type and returned the result.

Exercise 18.

Create a **Computer** class with fields defining computer features: processor, ram, graphics card, company and model. Implement setters, getters, constructor with all fields, **toString()**, **equals()** and **hashcode()** methods.

Instantiate several objects and check how the methods work.

Exercise 19.

Create a **Laptop** class extending the **Computer** class from the previous task. The **Laptop** class should additionally contain the battery parameter.

Implement additional getters, setters, constructor and overwrite the **toString()**, **equals()** and **hashcode()** methods accordingly.

Use a reference to parent class methods.

Exercise 20.

Create an abstract **Shape** class with the abstract methods **calculatePerimeter()** for calculating the perimeter and **calculateArea()** for calculating the area.

Create **Rectangle**, **Triangle**, **Hexagon** classes, extending the **Shape** class, and implementing abstract methods accordingly. Verify the solution correctness.

Exercise 21.

Create an abstract **3DShape** class that extends the **Shape** class from the previous task. Add an additional method **calculateVolume()**.

Create **Cone** and **Qube** classes by extending the **3DShape** class, properly implementing abstract methods. Verify the solution correctness.

Exercise 22.

Create a **Fillable** interface with the **fill()** method. Implement the method in the **3DShape** class from the previous task or separately in the **Cone** and **Qube** classes.

The **fill()** method should take a parameter, e.g. int, and check whether after the action of filling the figure:

- will pour too much water into the figure and overflow,
- fill the figure with water to the brim,
- not pouring enough water.

For each situation, it should write the status information in the console. Use the **calculateVolume()** method.

Exercise 23.

Create a **Zoo** class that will have a collection of animals and will allow you to receive statistics about your animals:

int getNumberOfAllAnimals() → returns the number of all animals

Map <String, Integer> getAnimalsCount() → returns the number of animals of each species

Map <String, Integer> getAnimalsCountSorted() → returns the number of animals of each species sorted based on the number of animals of a given species, where the first element is always the species with the largest number of animals

void addAnimals(String, int) → adds n animals of a given species

software **development** academy

Exercise 24.

Create a **Basket** class that imitates a basket and stores the current number of items in the basket. Add the **addToBasket()** method, which adds the element to the basket (increasing the current state by 1) and the **removeFromBasket()** method, which removes the element from the basket (reducing the current state by 1).

The basket can store from 0 to 10 items. When a user wants to remove an element at 0 items state or add an element at 10 items state, throw the appropriate runtime exception (**BasketFullException** or **BasketEmptyException**).

Exercise 25.

Change the **BasketFullException** and **BasketEmptyException** exceptions from runtime exception type to checked exception type.

Handle them.

Exercise 26a.

Using functional programming mechanisms based on the given structure, display:

1. a list of all Models,
2. a list of all cars,
3. list of all manufacturer names,
4. list of all manufacturers' establishment years,
5. list of all model names,
6. list of all years of starting production of models,
7. list of all car names,
8. list of all car descriptions,
9. only models with an even year of production start,
10. only cars from manufacturers with an even year of foundation,
11. only cars with an even year of starting production of the model and an odd year of establishing the manufacturer,
12. only CABRIO cars with an odd year of starting model production and an even year of establishing the manufacturer,
13. only cars of the SEDAN type from a model newer than 2019 and the manufacturer's founding year less than 1919

Exercise 26b.

```
enum CarType {

COUPE, CABRIO, SEDAN, HATCHBACK

}

class Car {
   public String name;
   public String description;
   public CarType carType;


public Car(String name, String description, CarType carType) {

this.name = name;

this.description = description;

this.carType = carType;

}}
```

```
class Model {
   public String name;
   public int productionStartYear;
   List<Car> cars;


public Model(String name, int productionStartYear, List<Car> cars) {

this.name = name;

this.productionStartYear = productionStartYear;

this.cars = cars;

} }

class Manufacturer {
   public String name;
   public int yearOfCreation;
   List<Model> models;


public Manufacturer(String name, int yearOfCreation, List<Model> models) {

this.name = name;

this.yearOfCreation = yearOfCreation;

this.models = models;

} }
```

software development academy

www.sdacademy.pl

Exercise 27.

Design the **Joiner<T>** class, which in the constructor will take a separator (string) and have a **join()** method that allows you to specify any number of T-type objects. This method will return a string joining all passed elements by calling their **toString()** method and separating them with a separator.

eg. for **Integers** and separator "**-**" it should return: 1-2-3-4 ...

Exercise 28.

Extend the **ArrayList<E>** class by implementing the **getEveryNthElement()** method. This method returns a list of elements and takes two parameters: the element index from which it starts and a number specifying which element to choose.

For the list: [a, b, c, d, e, f, g] and parameters: startIndex = 2 and skip = 3 it should return [c, g]

Exercise 29.

Implement the generic **partOf** method, which will return % of items satisfying the condition based on any type of array and the function indicated as parameters.

Exercise 30.

Write a program that will read any file and save it in the same folder. The content and title of the new file should be reversed (from the back).

Exercise 31.

Write a program that will count the occurrences of each word in a text file and then save the results in the form of a table in a new file.

software **development** academy

Exercise 32.

Write a program that will be able to save the list of items (e.g. cars) to a file in such a format that it can also read this file and create a new list of items and write it to the console.

Exercise 33.

Write a program that will display all photos (.png, .jpg) in a given directory and subdirectories.

Exercise 34.

Create a class that extends the **Thread** class, e.g. **MyThread**, overload the **run()** method so it displays the thread name in the console by reading it from the static method of the current thread:

```
Thread.currentThread().getName()
```

Create a class with the **public static void main (String [] args)** method. Inside the main method create a type variable of our class created above, e.g. **MyThread** and initialize the class. Perform the **start()** method on the variable.

Exercise 35.

Create a class implementing **Runnable**, e.g. **MyRunnable**. Implement the **run()** method, which will display the name of the current thread in the same way as in the previous exercise.

Create a class with the **public static void main (String [] args)** method. Inside the main method, create a class type variable created above, e.g. **MyRunnable**, and initialize the class.

Create a **Thread** variable and initialize it by passing the **Runnable** interface implementation as the constructor parameter. Perform the **start()** method on a **Thread** type variable.

Exercise 36.

Create a **ThreadPlaygroundRunnable** class that implements the **Runnable** interface having a name field of type **String** with a public constructor for that field. The class should implement the **run()** method from the **Runnable** interface. This method should contain a loop iterating up to 10 and print the name of the current thread using Thread.currentThread().getName(), the name given in the constructor and the current iteration number.

Create a class that has two private static **Thread** fields and a standard **public static void main (String [] args)** method. Then initialize the **Thread** fields using the constructor that accepts the **Runnable** object and pass **ThreadPlaygroundRunnable** creating it using the constructor, each time giving a different name.

On each thread (Thread) use the **start()** method.

Exercise 37.

Create a class containing the standard static method main and a variable of type **Executor** and using the factory method **newFixedThreadPool** of the **Executors** class to create a pool of 2 executors.

In iteration, add 10 **ThreadPlaygroundRunnable** objects from the previous task to the executor. Use any string and current iteration number as the name.

Exercise 38.

Write an application that will simulate a coffee making machine. In the event that any coffee brewing service finds an empty water tank, the thread should stop. When the water is refilled in the machine, the thread should be excited.

Exercise 39.

Write a program that will solve the problem below.

There is a system that stores current results in competitions. Many screens read the current results, while several sensors update these results. Write a program that allows continuous reading of data through many screens (threads) and updating data by many sensors (threads).

To update results, the sensor must provide current results along with new ones. The system then verifies that the sensor has current data and updates the data. If the system data has changed during this time, the sensor's data update is rejected.

For simplicity, you can assume that the sensor is reading data, waiting a random amount of time, and increasing the data by a random value.