Practical works on the course "Programming in Java"  
**Contents**

[**Practical work No.1 Project structure** 2](#_Toc63175912)

[Project Overview 2](#_Toc63175913)

[Objects: Tasks 2](#_Toc63175914)

[Task 1. Creating a base class 2](#_Toc63175915)

[Task 2. Checking the next task execution 3](#_Toc63175916)

[**Practical work No.2 Arrays and Links** 5](#_Toc63175917)

[Task 1. Creating a list based on an array 5](#_Toc63175918)

[Task 2. Selection of tasks 5](#_Toc63175919)

[**Practical work No.3 Exceptions** 6](#_Toc63175920)

[Task 1. Exceptions 6](#_Toc63175921)

[Task 2. Using the linked list 6](#_Toc63175922)

[**Practical work No.4 Inheritance Mechanism** 7](#_Toc63175923)

[Task 1. Inheritance 7](#_Toc63175924)

[Task 2. Abstract Factory 7](#_Toc63175925)

[**Practical work No.5 Iterators, Service Methods** 8](#_Toc63175926)

[Task 1. Iterators 8](#_Toc63175927)

[Task 2. Service methods 8](#_Toc63175928)

[**Practical work No.6 Java 8+** 9](#_Toc63175929)

[Task 1. Using streams 9](#_Toc63175930)

[**Practical work No.7 Service Classes and Collections** 10](#_Toc63175931)

[**Task 1. Transition to the use of a LocalDateTime class** 10](#_Toc63175932)

[**Task 2. Logic of work with the calendar** 10](#_Toc63175933)

[**Practical work No.8 Input/Output and Serialization** 11](#_Toc63175934)

[Task 1. Serialization 11](#_Toc63175935)

[Task 2. Binary input-output 11](#_Toc63175936)

[Task 3. Text input-output 11](#_Toc63175937)

[**Complex practical work No.9 Full-featured "Task Manager" Application** 13](#_Toc63175938)

[General information 13](#_Toc63175939)

[The main requirements 13](#_Toc63175940)

[Application design 13](#_Toc63175941)

[Communication Diagram 13](#_Toc63175942)

[Class Diagram 14](#_Toc63175943)

# **Practical work No.1 Project structure**

## Project Overview

The practical works represent the development of a single project, its evolution from a simple class to a full-featured user application. The work topics include tasks that a user can create for himself/herself, such as "Bowling with friends on Wednesday", or "Running 3 km every day at six o'clock". The project that is the subject of these practical works is an application intended to help the user store, view, and remind to complete the tasks.

## Objects: Tasks

The main objects that the application will work with are tasks.

Tasks have some text that describes the *details of the task*, such as "Cleaning the room."

In addition, tasks can be *active or inactive* . For example, during the holiday, task "Morning run" may be inactive and temporarily not performed.

Integers will be used to describe the start time, indicating, for example, the number of hours that have elapsed since the start of the countdown (for example, from 00:00 on February 1, 2000), and the number 36 will mean 12:00 on February 2, 2000.

Tasks can be scheduled to be performed *once*, for example "Meeting in a cafe on June 26 at 18:00":

Time

“Meeting in a café”

0

June 26

Or the task can be scheduled to be performed *regularly* over a period of time at a given interval (in hours), such as "Morning run from June 1 to June 5 every day at 8:00":

Time

"Morning run"

0

Start time: 08:00   
June 1

End time: 00:00   
June 5

Repetition interval 24 hours

Thus, the goal of the first practical work will be to create a Task object class.

**Details on performing the work**

When performing this work, you must design your class files as described in the Java Code Conventions, which can be found at <https://www.oracle.com/technetwork/java/codeconventions-150003.pdf>. In addition, all public elements must have javadoc, which briefly explains the purpose and use of the element. When writing such documentation, it is not necessary to describe things that are easy to understand from the declaration, such as types of parameters and results. It is necessary to specify non-obvious things: restrictions on parameter values, special resulting values, possible exceptional situations.

## Task 1. Creating a base class

Create a Task class in the ua.sumdu.j2se.studentName.tasks package (replace studentName with your name) using the following public methods:

* The Task (String title, int time) constructor constructs an inactive task to run at a specified time without repeating with a given name.
* The Task (String title, int start, int end, int interval) constructor constructs an inactive task to run within the specified time range (including the start and the end time) with the set repetition interval and with a given name.
* Methods for reading and setting the task name: String getTitle(), void setTitle(String title).
* Methods for reading and setting the task status: boolean isActive(), void setActive(boolean active).
* Methods for reading and changing execution time for non-repetitive tasks:
  + int getTime() – if the task is a repetitive one, the method must return the start time of the repetition;
  + void setTime(int time) – if the task was a repetitive one, it should become non-repetitive.
* Methods for reading and changing execution time for repetitive tasks:
  + int getStartTime() – if the task is a non-repetitive one, the method must return the start time of the execution;
  + int getEndTime() – if the task is a non-repetitive one, the method must return the end time of the execution;
  + int getRepeatInterval() – if the task is a non-repetitive one, the method must return 0;
  + void setTime(int start, int end, int interval) – if the task is a non-repetitive one, it should become repetitive.
* To check the task for repeatability use the boolean isRepeated() method.

Compile the project. It should be compiled without compilation errors, and preferably without design errors.

## Task 2. Checking the next task execution

An important operation when working with tasks is to find the next moment of the task execution.

Yes, if the task is not active, it is never executed, if the task is active and executed only once, then the next moment of execution will be either this single moment, or never (if the task has already been completed):

Time

"Meeting in a cafe"

0

June 26

Regarding this moment,   
the next execution is on June 26

Regarding this moment,   
the task will not be executed anymore

June 23

June 27

If the task is active and repetitive, then similarly the next moment of the task execution is found relative to the given time:

Time

"Morning run"

0

Start time:  
 June 1

End time:  
 August 1

Repetition interval: 24 hours

Regarding these moments, the following ones are marked

Regarding this moment,   
the task will not be executed anymore

You should add the int nextTimeAfter (int current) method to the Task class that returns the next start time of the task execution after the current time. If after the specified time the task is not executed anymore, the method must return -1.

# **Practical work No.2 Arrays and Links**

To perform tasks of the previous practical work, we developed a Task class of objects. Practical work No. 2 deals with the development of a list of tasks that allows you to simultaneously work with several tasks. Tasks in the list can be repeated, the order of the tasks does not matter, but it should not change unless the tasks are added to or removed from the list.

## Task 1. Creating a list based on an array

Create an ArrayTaskList class in the ua.sumdu.j2se.studentName.tasks package (replace studentName with your name) using the following public methods:

* void add (Task task) is a method that adds the specified task to the list.
* boolean remove (Task task)is a method that removes a task from the list and returns true, if such a task was in the list. If the list contains several tasks of the same type, any of them should be removed.
* int size() is a method that returns several tasks from the list.
* Task getTask(int index) is a method that returns a task which takes the specified place in the list; the index of the first task is 0.

Tasks in the list should be stored using an array. The list can contain any number of tasks that can be in the array, but should not occupy much more space than it is required at any particular moment. This means that, for example, if the list contains 5 tasks, then the array for their storage should not occupy space for 100 tasks.

## Task 2. Selection of tasks

Besides, the application should know which tasks from the list are scheduled at least once in a certain interval, for example, which tasks are scheduled for the next week. To implement this, create the ArrayTaskList incoming(int from, int to) method in the ArrayTaskList class. This method returns a subset of tasks that are scheduled for execution at least once after the "from" time, and not later than the "to" time.

**For example,** the list includes the following tasks:

1. Lunch with a beautiful girl on August 24 at 16:00.
2. Morning run from March 1 to September 1 every day.
3. Taking medication from August 20 to September 28 every 12 hours.
4. Meeting with friends on September 1 at 18:00.

Then, the tasks that take place from August 25, 8:00 to August 26, 8:00 are "*Morning run*" and "*Taking medication*".

# **Practical work No.3 Exceptions**

## Task 1. Exceptions

By this moment, there were no information in the tasks about what should happen if the not-admissible operations are performed, for example, trying to retrieve Task 5 out of the list containing three tasks only.

You should analyze the current code and decide, what to do in such situations, so keep in mind that:

1. In the previous tasks, the time was set by an integer, as a number of units of time that have elapsed since a certain starting point. According to this, the time labels cannot be negative.
2. The list of tasks should contain the tasks; that is why one cannot add empty links there (null).
3. The interval of tasks repeating should be more than zero.

The constructor of a Task class should necessarily generate the IllegalArgumentException exception in the case when the time was set as a negative number; and the method getTask should generate IndexOutOfBoundsException in the case when index exceeds the permissible limits for the list.

In all other cases, the decision of what to do in cases of limitation violations and if to generate an exception of a certain type must be reasoned.

## Task 2. Using the linked list

The concept of the task list does not depend on the task saving method; the users of ArrayTaskList class objects may even be unaware of the way this class is implemented. But the implementation through the array has its drawbacks, such as a slow operation of task deletion. Therefore, for scenarios, in which task deletion often takes place, it is necessary to create the list of tasks that will store tasks in a linked list (single-linked, double-linked or another modification to choose from; one cannot use the already existing implementations, such as java.util.LinkedList and others), which does not have this disadvantage. Create the LinkedTaskList class in the same package with the same methods as ArrayTaskList (in the incoming method change the object type that returns to LinkedTaskList). The objects of this class should behave the same as the objects of ArrayTaskList class.

# 

# **Practical work No.4 Inheritance Mechanism**

## Task 1. Inheritance

The methods in the ArrayTaskList and LinkedTaskList classes have almost the same destination, which allows to create a software of higher abstract level.

* Since both ArrayTaskList and LinkedTaskList classes implement the same data type, an AbstractTaskList class should be created to describe the methods specific to the task list as abstract and to inherit the two list classes from this class.
* As far as the incoming method implementation does not depend on the method of saving tasks, it can be implemented in the AbstractTaskList class (however, the object type, returned with it, also changes to AbstractTaskList), thus reducing the code duplication.

That is why we receive:

* the AbstractTaskList class that describes operations, which can be performed with the task list, and implements the methods that do not depend on the saving method;
* the ArrayTaskList class that is inherited from the AbstractTaskList class and implements the abstract operations of the AbstractTaskList class that depend on the saving method, using the array;
* the LinkedTaskList class that is inherited from AbstractTaskList and implements the abstract operations of the AbstractTaskList class that depend on the saving method, using the linked list.

## Task 2. Abstract Factory

In this stage, we have two classes – ArrayTaskList and LinkedTaskList, and a parent class – AbstractTaskList, which common for both of them. It would be rather convenient to have a separate class that would have a method that creates object of ArrayTaskList and LinkedTaskList depending on the parameter that is passed to it. That is the basic of an **abstract factory** pattern.

* Create a ListTypes class with a types field of enum type. The types field should contain ARRAY and LINKED values.
* Create a TaskListFactory class with the static method
  + AbstractTaskList createTaskList(ListTypes.types type). This method, according to the type parameter, should return the object of ArrayTaskList or LinkedTaskList class.

# **Practical work No.5 Iterators, Service Methods**

## Task 1. Iterators

For the abstract description of objects that contains the sequence of objects of another type, the standard library stores the java.util.Iterable<T> interface, where T is a type of objects – constituents. As far as AbstractTaskList class describes the collection of objects-tasks, it should implement this interface.

At this, the Iterable<T> interface describes an abstract iteration on objects selection, and the implementation of this interface should be optimal from the point of view of internal structure of the selection. For example, LinkedTaskList should not start searching for an element from the very beginning during the transition from the current element to the next one.

It is also necessary to be aware of the fact, that one task list can simultaneously have several independent iterators.

## Task 2. Service methods

Implement the service methods from the Object class:

1. The equals and hashCode methods in all classes. The list of tasks is considered to be the same if it contains the same tasks in the same order. The tasks are equal when their features are equal.
2. The toString method for all classes. The string display should provide the maximum of information available in a convenient form.
3. There should be an option of cloning Tasks and task lists. The implementation should not involve the call of class constructor.

# **Practical work No.6 Java 8+**

## Task 1. Using streams

Java 8 introduced additional possibilities for work with collections, using the functional programming.

1. It is necessary to add to the AbstractTaskListclass the Stream<Task> getStream() method, which allows to work with collections as with the streams.
2. This method should be overridden in child classes.
3. Move the incoming(int from, int to)method implementation from the ArrayTaskList and LinkedTaskList classes to the AbstractTaskList class. It means that the ArrayTaskList and LinkedTaskList classes should not contain the incoming(int from, int to) method.
4. Describe the logic of the incoming(int from, int to) method in AbstractTaskList, using the stream API, appeared in Java 8.
5. Prohibit overriding the incoming(int from, int to) method in child classes, it should be implemented only in the AbstractTaskList class.

**Practical work No.7 Service Classes and Collections**

**Task 1. Transition to the use of a LocalDateTime class**

In previous tasks, we used integers to work with time points, but the standard library contains java.time.LocalDateTime class for reperesnting date and time. The use of integers should be replaced by LocalDateTime to work with time points.

However, it is necessary to consider that the objects of the LocalDateTime class can change unlike integers (if the setTime method is called there), so it is better to copy such objects, saving only the local copy.

To compare dates, use methods of the LocalDateTime class instead of converting dates to an integer.

**Task 2. Logic of work with the calendar**

At the moment, the task list classes contain both task storage logic and task timing logic (incoming method).

To allow the usage of any collection for saving tasks, it is necessary to create a separate class for working with tasks collection – the Tasks class, and to move the incoming method there as a static method. Whereby, the method should be processed separately from the task list – method signature will be the following Iterable<Task> incoming(Iterable<Task> tasks, LocalDateTime start, LocalDateTime end).

Besides, it is necessary to implement another static method in the Tasks class –SortedMap<LocalDateTime, Set<Task>> calendar(Iterable<Task> tasks, LocalDateTime start, LocalDateTime end). This method builds the calendar of tasks for the specified period – a table, where each date corresponds to a set of tasks that should be performed at this time. At this, one task can refer to several dates, if it should be performed multiple times for the specified period.

**For example,** the list includes the following tasks:

1. Lunch with a beautiful girl on August 24 at 16:00.
2. Morning run from March 1 to September 1 every day at 8:15.
3. Taking medication from August 20 at 8:15 to August 28 every 12 hours.
4. Meeting with friends on September 1 at 18:00.

So the calendar of tasks for the period from August 25, 8:00 till August 26, 8:00 will be as follows:

|  |  |
| --- | --- |
| Date | Tasks |
| August 25, 8:15 | Morning run, taking medication |
| August 25, 20:15 | Taking medication |

While performing the work it is necessary to keep in mind that all collection classes in java.util can change their content after creation. For example, when filling Set in the calendar, which is located in Map, it should not contain all tasks at once, they can be added to the Set later.

# **Practical work No.8 Input/Output and Serialization**

To implement the full program on work with tasks, it is necessary to allow saving tasks and task lists on the disc and transfer through the net. This requires the implementation of read and write methods for task lists in different formats.

## Task 1. Serialization

The AbstractTaskList and Task implementation classes should be created in such a way that they can be serialized by standard methods.

## Task 2. Binary input-output

It is necessary to create a TaskIO class with the following static methods:

* void write(AbstractTaskList tasks, OutputStream out) – writes the tasks from the list to the stream in a binary format, described below.
* void read(AbstractTaskList tasks, InputStream in) – reads tasks from the stream to the current task list.
* void writeBinary(AbstractTaskList tasks, File file) – writes tasks from the list to the file.
* void readBinary(AbstractTaskList tasks, File file) – reads tasks from the file to the task list.

The format of task list presentation in a binary code:

The number of tasks

The length of the name

Name

Activity: 0/1

Repetition interval

Start time

Execution time

End time

If repeated

If **not** repeated

× number of tasks

It is useful to use the DataInput and DataOutput types for working with binary streams. Write and read the dates by converting them into integers.

## Task 3. Text input-output

It is necessary to describe the static methods in TaskIO class:

* void write(AbstractTaskList tasks, Writer out) – writes tasks from the list to the stream in the JSON format.
* void read(AbstractTaskList tasks, Reader in) – reads tasks from the stream to the list.
* void writeText(AbstractTaskList tasks, File file) – writes tasks to the file in JSON format
* void readText(AbstractTaskList tasks, File file) – reads tasks from the file.

The Gson library should be used for working with JSON format. This library can be connected using the Maven dependency:

<dependency>

<groupId>com.google.code.gson</groupId>

<artifactId>gson</artifactId>

<version>2.8.5</version>

</dependency>

This dependency should be included in the <dependencies> tag.

# **Complex practical work No.9 Full-featured "Task Manager" Application**

## General information

After performing all practical tasks, you have a library of classes for work with the calendar of tasks. Now your task is to create a convenient full-featured user-friendly application using this library.

## The main requirements

1. An application can have a TUI (console) or a GUI. A student can choose.
2. An application should keep the data between the starts in the file system.
3. An application should provide a user with the following basic functions:
   1. create new tasks;
   2. change parameters of the existing tasks;
   3. delete tasks;
   4. view the information about the existing tasks;
   5. view the calendar of events scheduled for a certain period of time.
4. The application should notify the user in time when some task should be performed.
5. An application should be resistant to errors: display notifications that are easy-to-understand for users, maintain performance, control user's input.
6. An application should use log4j orslf4j library to log in.

## Application design

Before you start working on the code, you need to think about the design of the application, which results in two main documents – the external design in the form of a communication diagram, and the internal design in the form of a class diagram.

### Communication Diagram

This diagram should show the actions a user can perform with the system and their results, for example:

calendar

1. Getting started
2. List of all tasks
3. Calendar for the week
4. Introduction of a new task
5. Detailed information about the task
6. End of work

add/edit ID

save/discard

view ID

activate/deactivaterd

back

quit

back

add

### Class Diagram

This diagram displays internal architecture of an application, a system, a subsystem and modules, as well as methods of their interaction, for example:

Notification subsystem

Model

**Task**  
+ nextTimeAfter(Date): Date  
+ title: String  
+ isActive: boolean

**Tasks**  
+ incoming(List<Task> tasks,   
 Date from, Date to): Set<Task>  
+ calendar(List<Task> tasks,  
 Date from, Date to):   
 Map<Date,Set<Task>>

uses

**NotificationManager**

uses

***<Notification>***+ notify(Set<Task>)

***ScreenNotification***+ notify(Set<Task>)

***MailNotification***+ notify(Set<Task>)

uses

implements

implements

The application architecture should be designed so that separate subsystems and modules are as independent as possible, and interact via interfaces; it is preferable to use the MVC patterns, Observer/EventListener, etc.