
A Terms of cooperation

Obtaining a credit

Each task has a **first deadline** specified in its title. A student may obtain maximum score for solution to an assignment in case he/she presents it during the laboratory class on the first deadline at the latest. Assignments are, in general, evaluated for maximum 10 points. Solutions to some specific assignments may be assessed for more points, which is discussed in the text of the particular tasks.

The **second and very last opportunity to present a solution to an assignment** is the second deadline – i.e. laboratory class next to the first deadline class. Defending a solution to an assignment during a **second deadline** – i.e. laboratory class next to the first deadline class – entitles a student to obtain 50% of the maximum original score at the most.

Evaluation of student solutions

While presenting a solution to an assignment a student should **demonstrate that he/she is aware of its internal architecture** (the applied technologies and combining them or used algorithms).

During the presentation a student should be prepared to answer some control question asked by the person who evaluates the solution. In case the person assessing the solution is not convinced of student's authorship, the **student gets 0 points for the solution and loses any further opportunity to defend the solution to the given assignment**.

The above statement implies that a student **may fail in getting a credit** in case he/she attempts to defend a solution to a mandatory task (see below) and the person evaluating the solution comes to conclusion that the student has not authored the presented solution.

Unit tests

Each solution is required to have defined some unit tests which verify that student's implementation works as expected.

Unit tests should be defined in different source folder of Eclipse project to provide separation from the actual implementation.

In case a student fails to provide unit tests for the given assignment he/she will get maximum 50% of the score for the given deadline – i.e. if a student does not provide unit tests at 2nd deadline he/she will be entitled to get maximum $0.75 * 0.5 = 0.375$ of the maximum score for the particular assignment.

Robustes of student solutions

The presented solutions should validate the input data provided by the user. **The undesired behaviours which entail reducing of awarded number of points during assessment include i.a.:**

- no verification whether a user passed a valid file path or whether file exists;
- missing validation of the input format – in case the format is known in advance (e.g. format of postal code);
- missing verification, whether value of a reference variable/field was set

(i.e. avoiding of `NullPointerException`);

- missing verification whether value of array index does not exceed the actual size of the array/list (`ArrayIndexOutOfBoundsException`).

Evaluation (may undergo some changes in favour of students)

- *3.0 – 20÷35 points*
- *3.5 – 36÷60 points*
- *4.0 – 61÷80 points*
- *4.5 – 81÷110 points*
- *5.0 – 111+ points*

Minimum requirements for getting credit (mandatory tasks)

To get a credit you need to receive at least 5 points for each of the assignments concerning the following topics:

- Generics (assignment 1)
- Collections (assignment 4)
- Input/Output Streams (assignment 8)
- JDBC (assignment 10)

1 Generics (7th October / 9th October)

Create a 2-3 different classes implementing both interfaces: (1) **IAggregable** and (2) **IDeeplyCloneable** specified below – as class **Person** put in a code snippet below.

Create an implementation of **IContainer** which could hold instances of your **IAggregable** and **IDeeplyCloneable**.

```
public interface IAggregable<
    TElement extends IAggregable<TElement, TResult>, TResult>
{
    TResult aggregate(TResult intermediateResult);
}

public interface IDeeplyCloneable<
    TElement extends IDeeplyCloneable<TElement>>
{
    TElement deepClone();
}
```

```

/**
 * a sample class implementing both interfaces
 */

public class Person implements
    IAggregable<Person, Integer>,
    IDeeplyCloneable<Person>
{

    private int _age;

    public Integer aggregate(Integer intermediateResult)
    {
        if (intermediateResult == null) {
            return _age;
        }
        return _age + intermediateResult;
    }

    public Person deepClone()
    {
        Person clone = new Person();
        clone._age = _age;
        return clone;
    }
}

```

```

public interface IContainer<
    TElement extends IAggregable<TElement, TResult>
    & IDeeplyCloneable<TElement>, TResult>
{

    TResult aggregateAllElements();

    TElement cloneElementAtIndex(int index);
}

```

2 Functional programming (14th October / 16th October)

Create a hierarchy of classes for storing information about human resources of a company based on the provided class skeletons.

Create a 3-level hierarchy composed of at least 20 different employees of all the kinds (i.e. trainees, workers and managers). Please note that only a manager can have subordinates.

Using functional programming features available since Java 8 please deliver the following functionality:

1. preparing payroll for all the employees;
2. preparing payroll for subordinates of the given manager;
3. calculating the total cost of bonuses;
4. searching for an employee with the longest seniority;
5. searching for the highest salary without bonus (a number should be returned);
6. searching for the highest salary with bonus (a number should be returned);
7. searching for employees whose surnames begin with 'A' subordinates of the given manager;
8. Searching for employees who earn more than 1000 PLN.

3 Lambda expressions (21st October / 23rd October)

Extend functionality of classes developed as the solution to assignment 2.

Using functional programming mechanism discussed during the lecture implement features described in the published solution skeleton.

4 Collections – introduction (28th October / 30th October)

Create an application which will read input data from the file of the following format:

<FirstName> <Surname> <Birthdate>
<FirstName> <Surname> <Birthdate>
<FirstName> <Surname> <Birthdate>

The contents of an example input file could be as follows:

John Smith 1980-01-03
Mark Brown 1976-02-02

A three composed of (1) first name,(2) surname and (3) birthdate **do not uniquely identify each person** – i.e. there may be multiple people of the same first name and surname and born on the same day.

The data entered from the input file should be stored in several collections:

1. sorted by first name;
2. sorted by surname, first name and birthdate;
3. sorted by birthdate;
4. collection which enables **quick filtering** of data of people born on a specific day.

NOTE: To enforce your custom sorting rules implement Comparable<TValue> and Comparator<TValue> interfaces.

For quick filtering feature use an implementation of Map<TKey, TValue> interface.

For storing birthdate use java.util.Date.

5 Collections – priority queues (4th November / 6th November)

Create a multi-threaded application based on requestor-service architecture.

The requestor threads put tasks in queues.

The service threads get tasks from queue, perform them and then put processing result in the queue with the same priority as the task.

Each task put in the queue by the requestor has a priority: LOW, NORMAL, HIGH.

Tasks are retrieved from the queue based on the priority.

An example for a task to be performed may be calculating sum of two numbers. In such case the task (request) object should contain two components and the result (response) object should contain the sum.

NOTE: The result should always be consumed by the task requestor.

6 Collections – sets, maps and collation (18th November / 20th November)

1. Create a class hierarchy composed of:
 - abstract class Person (social security no., surname, first name, birth date, nationality – preferably enumeration type with Locale instance as field)

Let us assume that social security no. will be a valid Polish PESEL (<https://en.wikipedia.org/wiki/PESEL>).

You may easily generate valid PESEL using <http://www.bogus.ovh.org/generatory/all.html>.

 - Student extending Person (student book no.)
 - Teacher extending Person (academic degree – represented as custom enumeration type, hiredate)
2. Apart from above mentioned classes create custom types which represent:
 - Department (name, employees – list of Teacher instances);
 - Student Group (name) – each group contains 8-10 students;
 - Subject (name, supervising Department, lecturer – instance of Teacher, attending Students).
3. Create classes storing extensions of the above classes (i.e. collections of their instances disallowing duplicates).
4. For extension classes storing information about Person and its derivatives ensure:
 - sorting instances based on Polish collation;
 - filtering people of the selected nationality and sort their names based on the rules for the nationality Locale.
5. Provide implementation of hashCode(), equals() and Comparable.compareTo() in all the above classes. Please make sure that you do not lose any of the created instances when they are put in their extensions.
6. Please create the following nationality instances:
 - Polish;
 - Ukrainian;
 - Belarussian;
 - Slovak;
 - Lithuanian;
 - Latvian;
 - British;
 - Indian;
 - Chinese;
 - Vietnamese.
7. Prepare a generator which automatically generate input data for the unit tests. The generated data set should be composed of at least
 - 100 students of random nationalities;
 - 12 student groups;
 - 3 departments;
 - 10 subjects
 - 10 teachers of random nationalities;

Please make sure that your generated data set is consistent – i.e. students are assigned to groups and attend subjects, subjects have lecturers, etc.

7 Stream (pipeline) processing (25th November / 27th November)

Create a set of utility 3 classes – i.e. a separate class for (1) regular files/directories, (2) zip archives, (3) jar archives – which enable user to:

1. find files or directories by name in the given directory, zip or jar archive respectively;
2. find files with the given content in the given directory, zip or java archive.

Please found your implementation on stream processing – provide both (1) sequential and (2) parallel implementations of particular utility methods – if feasible.

8 Input/Output streams (2nd December / 4th December)

Extend solution to mandatory assignment 4 (Collections) by adding the following features:

1. store your database in a file using `DataOutputStream` – you need to manually define the format of the binary file by specifying what data is stored on given steps;
2. restore your database from binary file with `DataInputStream`.

DO NOT SERIALIZE YOUR DATABASE BY IMPLEMENTING MARKER INTERFACE `Serializable` AND USING `ObjectOutputStream`.

9 Reflection (9th December / 11th December)

Encapsulation is one of good practices in software engineering. Encapsulation hides complexity of implementation beneath the API. However encapsulation impedes covering the concealed implementation with unit test.

Since reflection allows bypassing protection mechanism in Java™ it is very often used for accessing private, protected and package private components.

Create a class representing Polish PESEL number defining:

- private validation methods based on rules specified in <http://en.wikipedia.org/wiki/PESEL>;
- private method extracting birth date (as java.util.Date value) based on the first six digits of PESEL number;
- private method for extracting sex (enum type) of the person based on 10th digit of PESEL number.

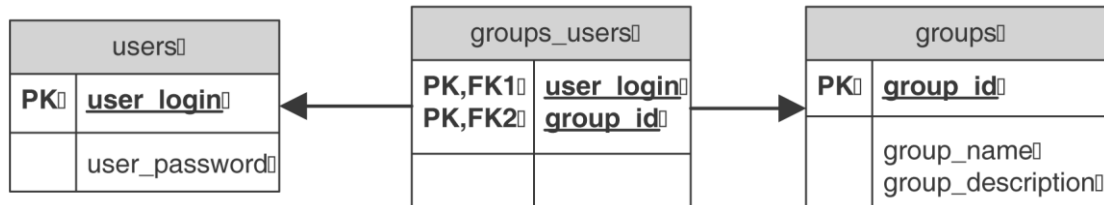
Verify your implementation with unit tests invoking the above mentioned methods with reflection.

10 Java Database Connectivity

(16th December / 18th December)

Develop an application which stores data in a relational database with JDBC.

Application data should be stored in schema as depicted below:



In your application records of (1) users and (2) groups data tables should be represented as DTO classes: (1) User, and (2) Group respectively.

Records of groups_users should be reflected by cardinalities of (1) a list of Users in a Group instance, (2) a list of Groups in User.

Your application should enable all CRUD (Create-Read-Update-Delete) operation, i.e.:

- inserting new Group/User entities into database;
- reading Group/User entities from database based on query;
- updating Group/User entities stored in database;
- deleting Group/User entities and from database;
- creating many-to-many associations between Group and User entities.

Search for Group entities should be based on:

- entity id – i.e. value of primary key in the ‘groups’ data table;
- part of group name with wildcards (e.g. ‘gr%’).

Search for User entities should be based on:

- entity id – i.e. value of primary key in the ‘users’ data table;
- part of user login with wildcards (e.g. ‘%us%’).

All database related operations should be encapsulated in so called repository.

IGroupRepository interface definition

```
public interface IGroupRepository {
    List<Group> findByName(String query);
    Group findById(int id);
    void add(Group user);
    void update(Group user);
    void delete(Group user);

    ... other methods if needed
}
```

UserRepository interface definition

```
public interface IUserRepository {  
    List<User> findByName(String query);  
    User findById(int id);  
    void add(User user);  
    void update(User user);  
    void delete(User user);  
  
    ... other methods if needed  
}
```

REMEMBER THAT YOUR TESTS FOR REPOSITORY METHODS RUN THOSE METHODS WITHIN A TRANSACTION AND ROLL BACK THE CHANGES WHEN THEY FINISH.

REMEMBER TO USE `PreparedStatement` INSTEAD OF `Statement` CLASS TO MAKE YOUR IMPLEMENTATION ROBUST TO SQL INJECTION EXPLOIT.

11 Java Persistence API (13th January / 15th January)

Based on the sample presented during the lecture devoted to Java Persistence API develop persistence tier for the domain model depicted in the below UML diagram.

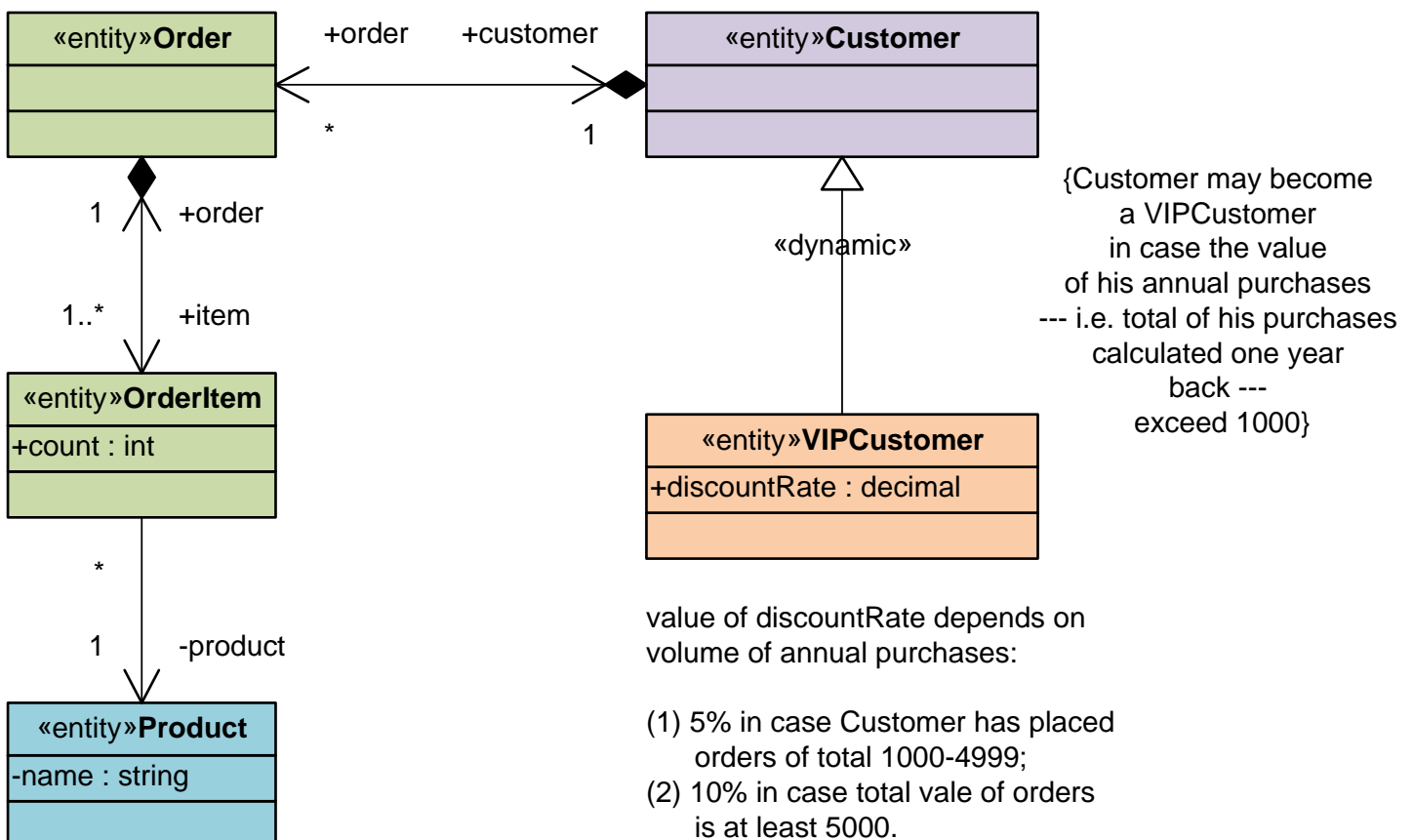
Your persistence tier implementation must (1) follow repository design pattern and (2) utilize one of Java Persistence API implementations.

Querying database should be take advantage of criteria – i.e. in the same way as it has been developed in the demo sample.

NOTE: Please remember to wrap tests with transactions – as it has been presented during the lecture.

NOTE: Since this this assignment is more difficult a successful deliverable can be evaluated for up to 25 points.

HINT: Dynamic inheritance actually does not need to be implemented as inheritance at all – any solution which meets the requirement and follows good practices is acceptable.



12 Java Concurrency (20th January / 22nd January)

Develop a GUI application running in a fixed-size thread pool tasks submitted by the user.

Each task can be in either of 4 different statuses:

- PENDING
- RUNNING
- ACCOMPLISHED
- FAILED

The user interface of the application a table (JTable) composed of the following 3 columns:

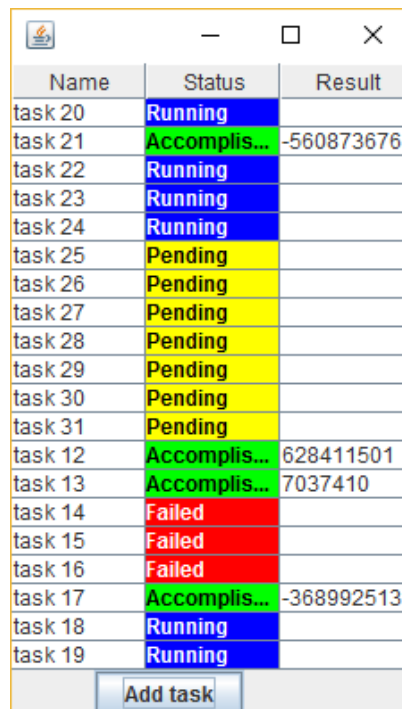
- 1st column – task name;
- 2nd column – processing status (see above) – for better visualization you can use different foreground/background color of the table cell presenting status a task;
- 3rd column – processing result in case task status is ACCOMPLISHED.

Table content should periodically refreshed (every 200 milliseconds) based on task list polling. The size of task list must never exceed predefined limit (e.g. 20 items).

A user adds tasks to the list by pressing a button placed beneath the above mentioned table. In case the number of tasks reaches the predefined limit a new task may only replace a task in the list only if it is either (1) ACCOMPLISHED or (2) FAILED.

Each task should be processed by one of the pooled threads. The thread pool size should be predefined (e.g. to 5). Execution of each task should last between 2 and 10 seconds, so that status change is perceivable to the human user.

NOTE: Since this assignment requires creating graphical user interface based on Model-View-Controller design pattern a successful solution will be evaluated up to 15 points.



The screenshot shows a Java Swing window with a title bar. Inside, there is a JTable with three columns: 'Name', 'Status', and 'Result'. The table contains 20 rows of task data. The 'Status' column uses color-coding: blue for 'Running', green for 'Accomplis...', yellow for 'Pending', and red for 'Failed'. The 'Result' column shows numerical values for completed tasks. Below the table is a button labeled 'Add task'.

Name	Status	Result
task 20	Running	
task 21	Accomplis...	-560873676
task 22	Running	
task 23	Running	
task 24	Running	
task 25	Pending	
task 26	Pending	
task 27	Pending	
task 28	Pending	
task 29	Pending	
task 30	Pending	
task 31	Pending	
task 12	Accomplis...	628411501
task 13	Accomplis...	7037410
task 14	Failed	
task 15	Failed	
task 16	Failed	
task 17	Accomplis...	-368992513
task 18	Running	
task 19	Running	

13 Synchronization of concurrent threads (27th January / 29th January)

Modify your solution to assignment 12 so that the tasks are added by client threads managed by a separate thread pool.

A client thread is suspended until processing the task it has submitted finishes.

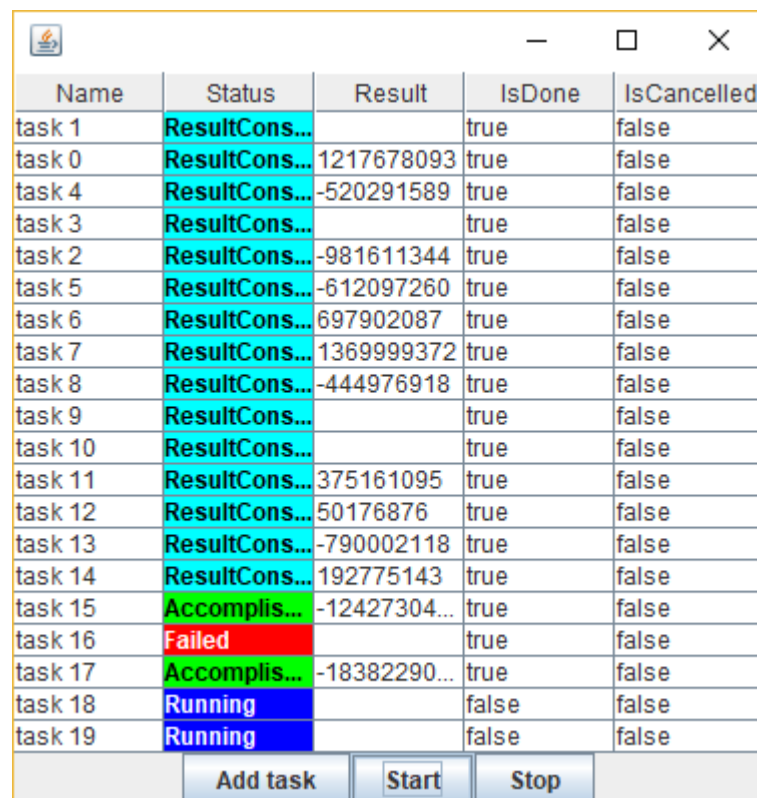
In order to achieve inactive waiting for task completion – i.e. with no need to verify task status at some intervals (and so introducing some inefficiencies) – the client thread should await on condition created specifically for the given task.

Extend the hitherto set of task statuses with one more – RESULT CONSUMED.

If number of tasks put in the queue (visualized as JTable) reaches the configured limit, the client thread can only replace tasks of status RESULT CONSUMED.

In order to make each status change perceivable to the human user submitting the new task should occur 3 seconds after status of the given task has been changed to RESULT CONSUMED.

The client threads are started and stopped with the two buttons beneath the data grid (JTable) presenting the current state of the simulation.



The screenshot shows a Java Swing window with a title bar containing a standard icon, a minus sign, a square icon, and a close button. The window contains a JTable with 5 columns: Name, Status, Result, IsDone, and IsCancelled. The table lists 20 tasks (task 0 to task 19). The Status column is color-coded: 'ResultCons...' in cyan, 'Accomplis...' in green, 'Failed' in red, and 'Running' in blue. The Result column contains numerical values for tasks 0-14 and 17, and is empty for tasks 1, 3, 5, 7, 9, 11, 13, 15, 16, 18, and 19. The IsDone column is 'true' for tasks 0-17 and 'false' for tasks 18-19. The IsCancelled column is 'false' for all tasks. Below the table are three buttons: 'Add task', 'Start', and 'Stop'.

Name	Status	Result	IsDone	IsCancelled
task 1	ResultCons...		true	false
task 0	ResultCons...	1217678093	true	false
task 4	ResultCons...	-520291589	true	false
task 3	ResultCons...		true	false
task 2	ResultCons...	-981611344	true	false
task 5	ResultCons...	-612097260	true	false
task 6	ResultCons...	697902087	true	false
task 7	ResultCons...	1369999372	true	false
task 8	ResultCons...	-444976918	true	false
task 9	ResultCons...		true	false
task 10	ResultCons...		true	false
task 11	ResultCons...	375161095	true	false
task 12	ResultCons...	50176876	true	false
task 13	ResultCons...	-790002118	true	false
task 14	ResultCons...	192775143	true	false
task 15	Accomplis...	-12427304...	true	false
task 16	Failed		true	false
task 17	Accomplis...	-18382290...	true	false
task 18	Running		false	false
task 19	Running		false	false

Buttons: Add task, Start, Stop