Final exam questions and tasks for subject OOP

(Object-oriented programming)

Department: ISE (Information system engineering)

Number of questions: 4
Examination time: ___ minutes

Evaluation criteria

Question#1. Theoretical question (10 points)

- Theoretical definition – 5 points
- Explain with examples – 5 points

Question#2. Practical task (20 points)

Task	Criteria	Points
Task type-2. Relationships between classes	for a-task	4
	for b-task	4
	for c-task	4
	for d-task	4
	for e-task	4
	Total	**20**

Question#3. Practical task (20 points)

Task	Criteria	Points
Task type-3. Inheritance, abstraction, method overriding	Create super classes	2
	Create derived classes	3
	Create abstract class and interfaces	5
	Creating construktors of derived and super classes	5
	Overriding abstract methods	5
	Total	**20**

Question#4. Practical task (20 points)

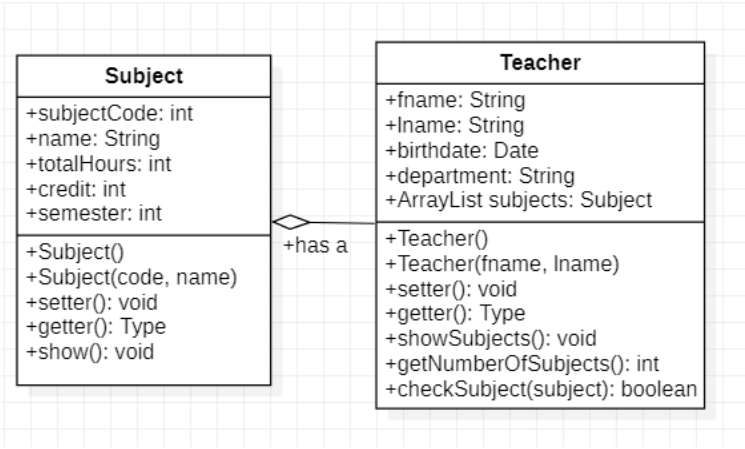
Task	Criteria	Points
Task type-4. GUI programming. JavaFX	Creating JavaFX UI	5
	Setting CSS style	5
	Using correct layouts	5
	Combining fxml file and Controller	5
	Total	**20**

Theoretical question

1. Data types in Java, variable declaration. Using classes and operators for input and output in Java.
2. Conditional (*if, if...else, switch...case*) and loop (*for, while*) operators in Java. Explain each operator with examples
3. Java arrays. Creating 1d and 2d arrays in Java using primitive data types. ArrayList in Java.
4. Char array and Strings in Java. Using String and StringBuilder classes, using methods of String and StringBuilder classes
5. Classes and objects in Java. Class attributes and methods
6. Class variables (local, static, instance). Explain differences between class variables with examples
7. Class constructors. Type of constructors and explain default and parametrized constructors with examples.
8. Array of objects in Java. Intialize array objects using default and parametrized constructors
9. Types of Relationship among Classes in Java. (Uses-A, Has-A, Part-Of)
10. The principle of inheritance between classes in OOP. Create a derived class. Types of inheritance in Java. Give examples
11. The principle of polymorphism in object-oriented programming. Method overloading in the classes. Give examples.
12. The principle of polymorphism in object-oriented programming. Method overriding in the classes. Give examples.
13. The principle of abstraction in object-oriented programming. Creating an abstract method and abstract classes in a Java environment. Interface concept and its use
14. JavaFX library and its capabilities. Tools and settings for GUI programming in Java
15. Layouts in JavaFX GUI. Pane and AnchorPane layout properties
16. Exception handling in the Java. Manage exceptions using keywords try, catch, throw, throws, and finally. Give examples
17. Working with IO streams for files in Java. Use FileOutputStream and FileWriter classes to write data to the file
18. Working with IO streams for files in Java. Use FileInputStream, FileReader and BufferedReader classes to read data from the file
19. Explain the structure of the program created in the JavaFX environment and the meaning of the Application, Controller, FXML files
20. Using layouts in JavaFX GUI. AnchorPane, VBox and HBox layout properties and differences between them.

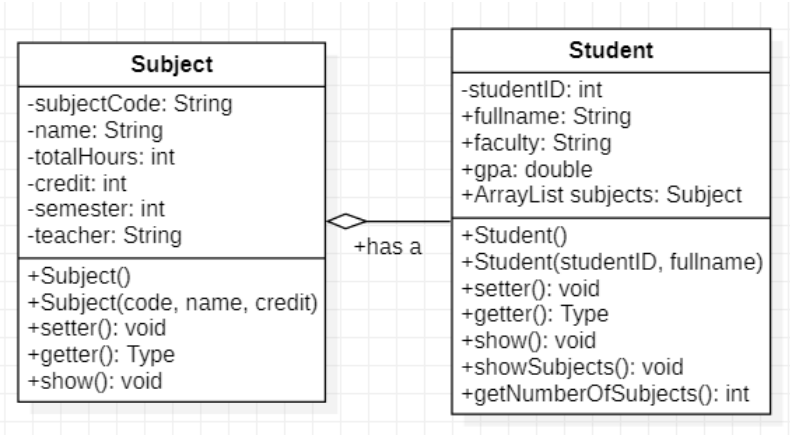
2-Practical task (Class and objects, constructors and methods, relationships between classes)

1. Create the Subject (*subjectCode, name, totalHours, credit, semester*) and Teacher(*fname, lname, birthdate, department, ArrayList<Subject> subjects*) classes listed in the diagram below and create a relationship between them.



   a) Correctly create the Subject class with the attributes, constructor and methods
   b) Correctly create the Teacher class with the attributes, constructor and methods
   c) Create “Has-A” relationship between Subject and Teacher classes
   d) Use the showSubjects () method to display a list of available subjects in the teacher, and if the teacher does not have a subject attached, display the message "This teacher does not have a subject"
   e) Return the number of subjects attached to the teacher using the getNumberOfSubjects() method, check whether the teacher has a subject using the checkSubject() method.

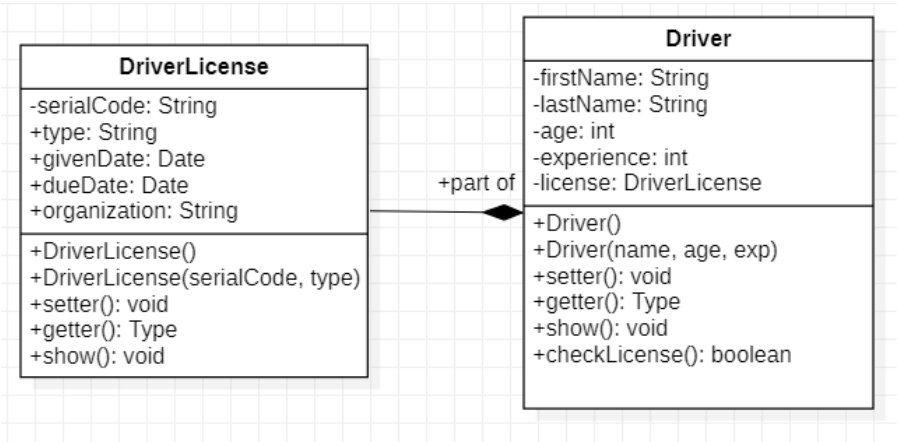
2. Create the Student (*studentID, fullName, faculty, gpa, ArrayList<Subject> subjects*) and Subject(*subjectCode, name, totalHours, credit, semester, teacher*) classes listed in the diagram below and create a relationship between them.



   a) Correctly create the Subject class with the attributes, constructor and methods
   b) Correctly create the Student class with the attributes, constructor and methods
   c) Create “Has-A” relationship between Subject and Student classes
   d) Use the showSubjects () method to display a list of available subjects in the student, and if the student does not have a subject, display the message "No subjects"

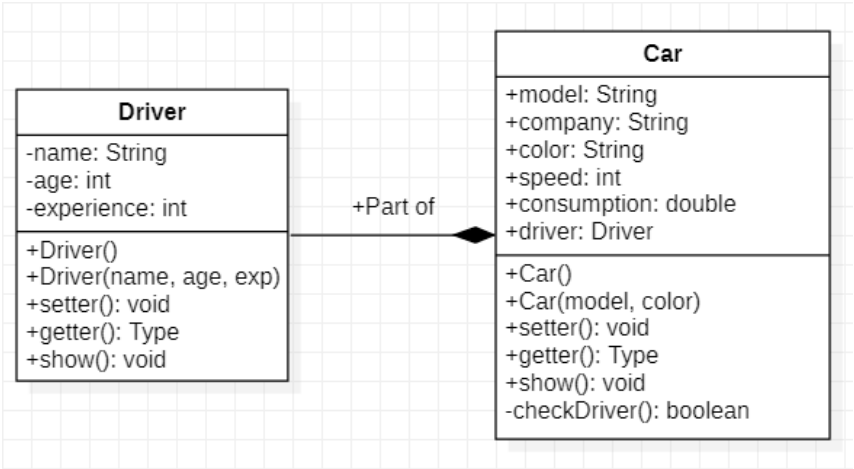
e) Return the number of subjects attached to the student using the getNumberOfSubjects() method.

3. Create the Driver (*firstName, lastName, age, experience, license*) and DriverLicense(*serialCode, type, givenDate, dueDate, organization*) classes listed in the diagram below and create a relationship between them.



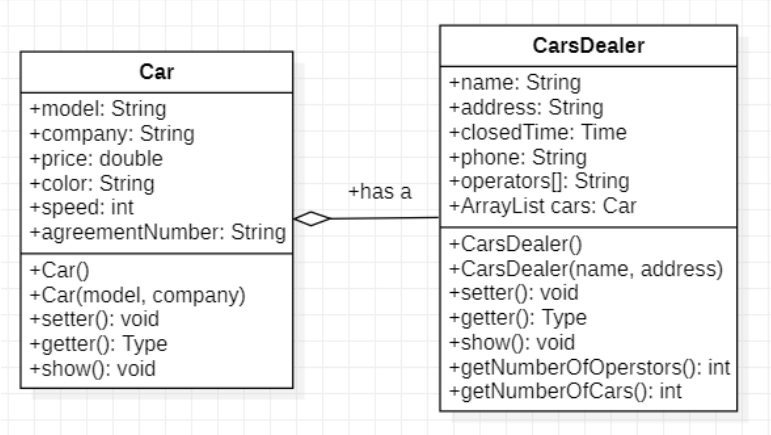
a) Correctly create the Driver class with the attributes, constructor and methods
b) Correctly create the DriverLicense class with the attributes, constructor and methods
c) Create “Part-of” relationship between Driver and DriverLicense classes
d) Use the chechLicense () method to check if the driver has a license
e) If the driver has a license printout Driver and license information using show() method, otherwise the message "This driver does not have a driver's license" will be displayed.

4. Create the Driver (*name, age, experience*) and Car (*model, company, color, speed, consumption, driver*) classes listed in the diagram below and create a relationship between them.



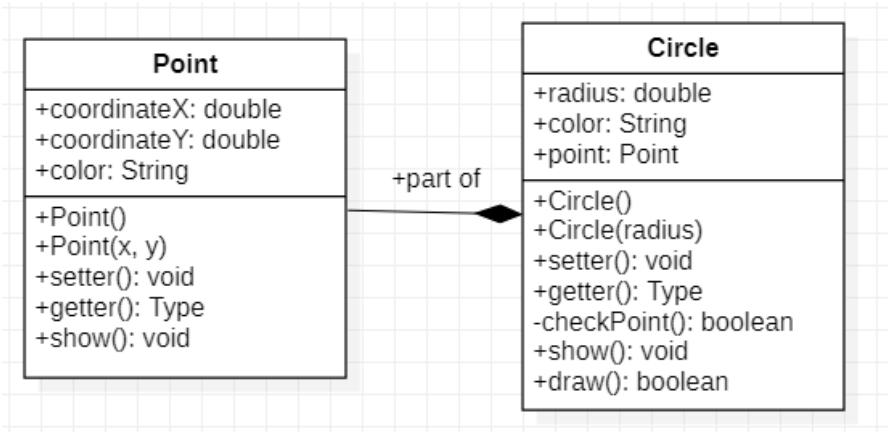
a) Correctly create the Driver class with the attributes, constructor and methods
b) Correctly create the Car class with the attributes, constructor and methods
c) Create “Part-of” relationship between Driver and Car classes
d) Use the chechDriver() method to check if the car has a driver
e) If the car has a driver printout Car and Driver information using show() method, otherwise the message "This car does not have a driver " will be displayed.

5. Create the keltirilgan Car (*model, company, price, color, speed, agreementNumber*) and CarsDealer(*name, address, closedTime, phone, operators[], ArrayList<Car> cars*) classes listed in the diagram below and create a relationship between them.



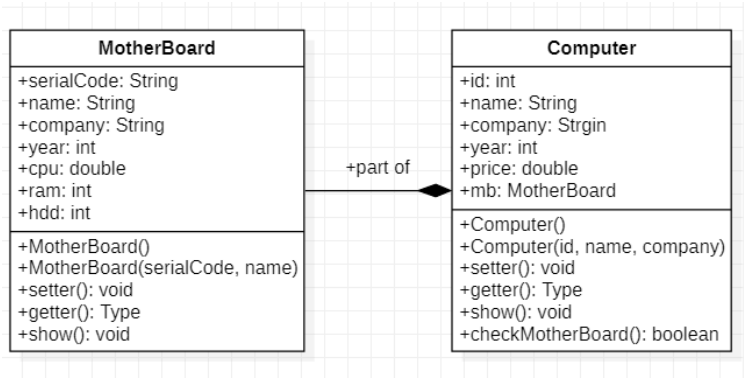
a) Correctly create the Car class with the attributes, constructor and methods
b) Correctly create the CarsDealer class with the attributes, constructor and methods
c) Create "Has-A" relationship between Car and CarsDealer classes
d) In the CarsDealer class, print the list of operators using showOperators() method, if not exsists, display the message "No operators in this dealer".
e) Return the number of operators and cars in the carshop, using the method getNumberOfOperators() and getNumberOfCars() respectively.

6. Create the keltirilgan Point(*coordinateX, coordinateY, color*) and Circle(*radius, color, point*) classes listed in the diagram below and create a relationship between them.



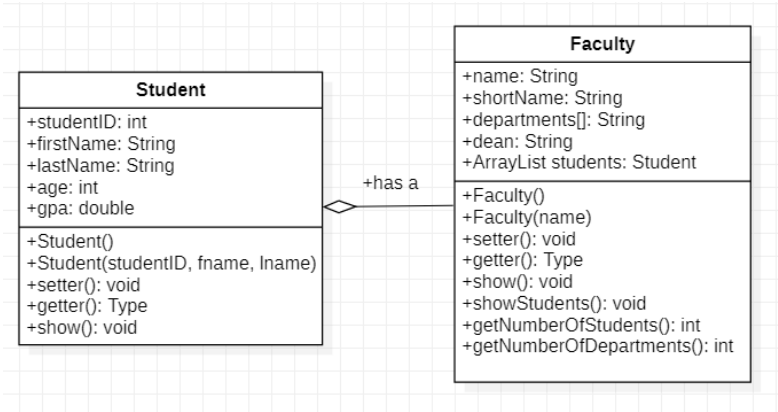
a) Correctly create the Point class with the attributes, constructor and methods
b) Correctly create the Circle class with the attributes, constructor and methods
c) Create "Has-A" relationship between Point and Circle classes
d) Use checkPoint() method to check Point to draw circle, if yes, use draw() and show() method of Circle class
e) If no print the error message "Mark the point to draw a circle".

7. Create the keltirilgan MotherBoard (*serialCode, name, company, year, cpu, ram, hdd*) and Computer(*id, name, company, year, price, mb*) classes listed in the diagram below and create a relationship between them.



a) Correctly create the MotherBoard class with the attributes, constructor and methods
b) Correctly create the Computer class with the attributes, constructor and methods
c) Use the default constructor to generate random values for object attributes regarding data types
d) Create "Part-of" relationship between MotherBoard and Computer classes
e) Use the chechMotherBoard () method to check if the current "computer" has a "motherboard" and, if available, use the show () method of the Computer class, otherwise print the message "This computer does not have a motherboard".

8. Create the Student (*studentID, firstName, lastName, age, gpa*) and Faculty(*name, shortName, departments[], dean, ArrayList<Student> students*) classes listed in the diagram below and create a relationship between them.



a) Correctly create the Student class with the attributes, constructor and methods
b) Correctly create the Faculty class with the attributes, constructor and methods
c) Create "Has-A" relationship between Student and Faculty classes
d) Print the list of students in the faculty using the showStudents() method, if students are not attached to the faculty, print out the message "student is not available"
e) Return the number of students and departments in the faculty, respectively, using the methods getNumberOfStudents() and getNumberOfDepartments().

3-Practical task (Inheritance, abstraction, interface, method overriding)

1. Establish inheritance between given classes (*PNGImage, Image*). Create a program by overriding methods of interface or abstract class (*ImageView*) get results. Use your own logic and data to override abstract methods and show result (print messages, calculate parameters).

Derived class	Super class	Interface
PNGImage **<u>Atributes:</u>** data: int[][] contrast: double **<u>Constructors:</u>** PNGImage() setter(): void getter(): Type	**Image** **<u>Atributes:</u>** name: String height: int width: int **<u>Constructors:</u>** Image()	**Imageview** **<u>Abstract methods:</u>** show(): void open(): void getSize(): String

2. Establish inheritance between given classes (*JPEGImage, Image*). Create a program by overriding methods of interface or abstract class (*ImageView*) get results. Use your own logic and data to override abstract methods and show result (print messages, calculate parameters).

Derived class	Super class	Interface
JPEGImage **<u>Attributes:</u>** data: int[][] verticalResolution: double horizontalResolution: double **<u>Constructors:</u>** JPEGImage() setter(): void getter(): Type	**Image** **<u>Attributes:</u>** name: String height: int width: int **<u>Constructors:</u>** Image()	**Imageview** **<u>Abstract methods:</u>** show(): void open(): void getSize(): String

3. Establish inheritance between given classes (*Circle, Shape*). Create a program by overriding methods of interface or abstract class (*Drawable*) get results. Use your own logic and data to override abstract methods and show result (print messages, calculate parameters).

Derived class	Super class	Interface
Circle **<u>Attributes:</u>** radius: double **<u>Constructors:</u>** Circle() setter(): void getter(): Type	**Shape** **<u>Attributes:</u>** color: String typeShape: String **<u>Constructors:</u>** Shape()	**Drawable** **<u>Abstract method:</u>** draw(): void area(): double perimeter(): double

4. Establish inheritance between given classes (*Rectangle, Shape*). Create a program by overriding methods of interface or abstract class (*Drawable*) get results. Use your own logic and data to override abstract methods and show result (print messages, calculate parameters).

Derived class	Super class	Interface
Rectangle **<u>Attributes:</u>** sideA: double sideB: double **<u>Constructors:</u>** Rectangle() setter(): void getter(): Type	**Shape** **<u>Attributes</u>** color: String typeShape: String **<u>Constructors:</u>** Shape()	**Drawable** **<u>Abstract methods:</u>** draw(): void area(): double perimeter(): double

5. Establish inheritance between given classes (*Triangle, Shape*). Create a program by overriding methods of interface or abstract class (*Drawable*) get results. Use your own logic and data to override abstract methods and show result (print messages, calculate parameters).

Derived class	Super class	Interface
Triangle **<u>Attributes:</u>** sideA: double sideB: double sideC: double **<u>Constructors:</u>** Rectangle() setter(): void getter(): Type	**Shape** **<u>Attributes:</u>** color: String typeShape: String **<u>Constructors:</u>** Shape()	**Drawable** **<u>Abstract methods:</u>** draw(): void area(): double perimeter(): double

6. Establish inheritance between given classes (*Car, Transport*). Create a program by overriding methods of interface or abstract class (*Vehicle*) get results. Use your own logic and data to override abstract methods and show result (print messages, calculate parameters).

Derived class	Super class	Interface
Car **<u>Attributes:</u>** model: String price: double consumption: double **<u>Constructors:</u>** Car() setter(): void getter(): Type	**Transport** **<u>Attributes:</u>** color: String company: String fuelType: int **<u>Constructors:</u>** Transport()	**Vehicle** **<u>Abstract methods:</u>** start(): void stop(): void checkSpeed(): int

7. Establish inheritance between given classes (*Bus, Transport*). Create a program by overriding methods of interface or abstract class (*Vehicle*) get results. Use your own logic and data to override abstract methods and show result (print messages, calculate parameters).

Derived class	Super class	Interface
Bus **<u>Attributes:</u>** model: String passangers: int	**Transport** **<u>Attributes:</u>** color: String company: String	**Vehicle** **<u>Abstract methods:</u>** start(): void stop(): void

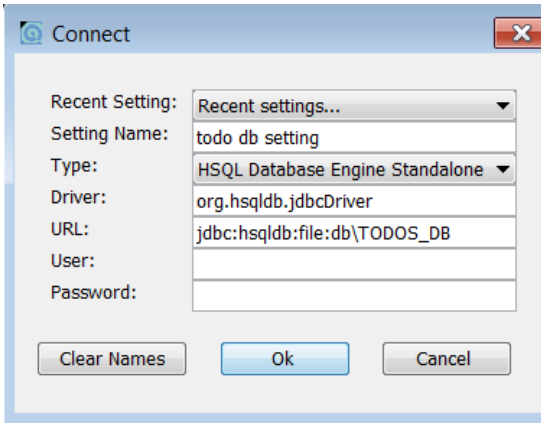
consumption: double **<u>Constructors:</u>** Bus() setter(): void getter(): Type	fuelType: int **<u>Constructors:</u>** Transport()	checkSpeed(): int

8. Establish inheritance between given classes (*Airplane*, *Transport*). Create a program by overriding methods of interface or abstract class (*Vehicle*) get results. Use your own logic and data to override abstract methods and show result (print messages, calculate parameters).

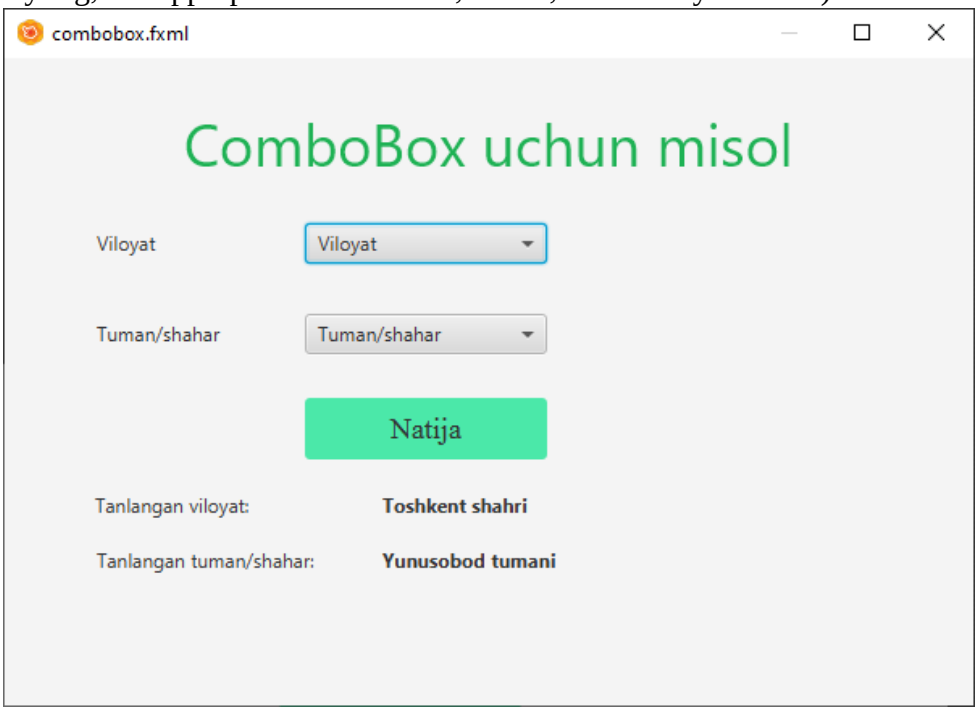
Derived class	Super class	Interface
Airplane **<u>Attributes:</u>** model: String passengers: int type: String **<u>Constructors:</u>** Airplane() setter(): void getter(): Type	**Transport** **<u>Attributes:</u>** color: String company: String fuelType: int **<u>Constructors:</u>** Transport()	**Vehicle** **<u>Abstract methods:</u>** start(): void stop(): void takeoff(): void checkSpeed(): int

4-Practical task (GUI programming. JavaFX**)**

1. Create the following “DB-Connection” form using an **AnchorPane** layout.
   Use the following requirements for CSS styling:
   JavaFX Label (Font familiy: Bell MT, Bold, Size: 18px, Color: black)
   JavaFX Button (backgroundColor: #165ecc, textColor: #07ff20, width: 100px, height: 50px)
   JavaFX TextField (Font familiy: Candara, Size: 14px, backgroundColor: #f4f4f4, borderColor: brown, width: 250px, height: 40px)
   JavaFX ComboBox (backgroundColor: #f4f4f4, borderColor: : #165ecc, width: 250px, height: 40px)
   VBox (spacing: 30px)

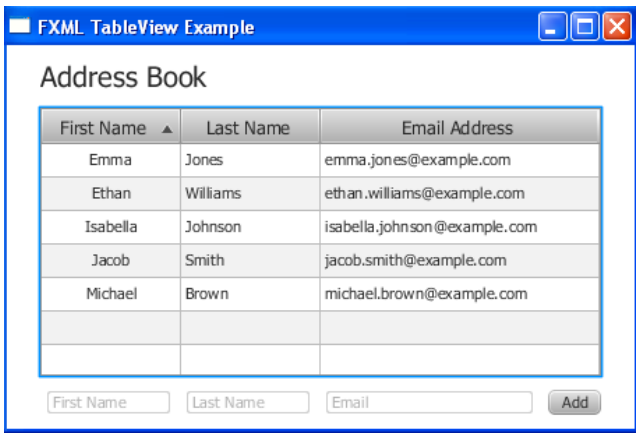


2. Create the following “address” form using an **AnchorPane** layout (with given CSS styling, use appropriate UI controls, colors, font family and size)

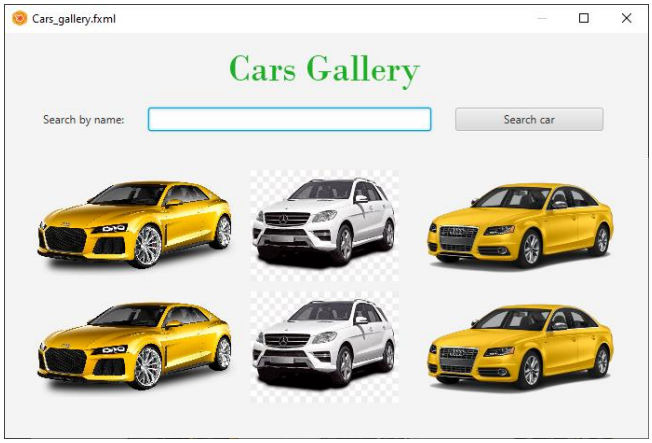


3. Create the following “orders” form using an **AnchorPane** layout (with with given CSS styling, use appropriate UI controls, colors, font family and size)

4. Create the following "address-book" form using an **AnchorPane** layout. Use the following requirements for CSS styling:
JavaFX Text (Font familiy: Temis New Roman, Size: 40px, Color: #1eb025)
JavaFX Button (backgroundColor: blue, textColor: #c9cc16, width: 70px, height: 30px)
JavaFX TextField (Font familiy: Candara, Size: 11px, backgroundColor: #f4f4f4, borderColor: brown, width: 140px, height: 30px, placeHolder: same with the given picture)

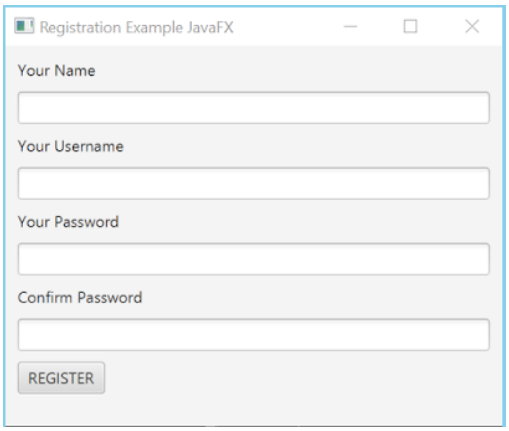


5. Create the following "cars-gallery" form using an **AnchorPane** layout. Use the following requirements for CSS styling:
JavaFX Text (Font familiy: Bodini MT, Size: 40px, Color: #1eb025)
JavaFX Label (Font familiy: Bell MT, Bold, Size: 24px, Color: blue)
JavaFX Button (backgroundColor: blue, textColor: #c9cc16, width: 100px, height: 40px)
JavaFX TextField (Font familiy: Candara, Size: 14px, backgroundColor: #f4f4f4, borderColor: brown, width: 200px, height: 40px)



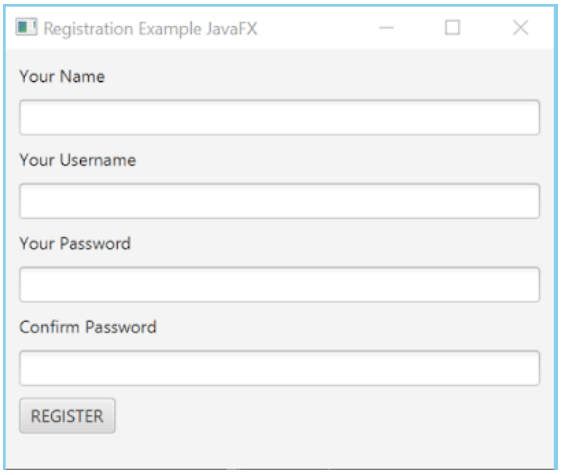
6. Create the following “registration” form using an **AnchorPane** layout.

   Use the following requirements for CSS styling:
   JavaFX Label (Font familiy: Bell MT, Bold, Size: 24px, Color: blue)
   JavaFX Button (backgroundColor: blue, textColor: #c9cc16, width: 150px, height: 60px)
   JavaFX TextField (Font familiy: Candara, Size: 14px, backgroundColor: #f4f4f4, borderColor: brown, width: 275px, height: 40px, placeHolder: same with the label name)



7. Create the following “registration” form using an **HBox and VBox** layout.

   Use the following requirements for CSS styling:
   JavaFX Label (Font familiy: Bell MT, Bold, Size: 24px, Color: blue)
   JavaFX Button (backgroundColor: green, textColor: #c9cc16, width: 120px, height: 60px)
   JavaFX TextField (Font familiy: Candara, Size: 12px, backgroundColor: #efefef, borderColor: gray, width: 275px, height: 40px)



8. Create the following “login” form using an **HBox and VBox** layout.

   Use the following requirements for CSS styling:
   JavaFX Text (Font familiy: Bodini MT, Size: 40px, Color: #1eb025)
   JavaFX Label (Font familiy: Bell MT, Bold, Size: 22px, Color: #3355ab)
   JavaFX Button (backgroundColor: green, textColor: #c9cc16, width: 120px, height: 60px)
   JavaFX Checkbox Label (Font familiy: Bell MT, Italic, Size: 10px, Color: #000000)

9. Create the following “login” form using an **AnchorPane** layout.

Use the following requirements for CSS styling:
JavaFX Text (Font familiy: Bodini MT, Size: 36px, Color: #1eb254)
JavaFX Label (Font familiy: Bell MT, Bold, Size: 24px, Color: #2244ab)
JavaFX Button (backgroundColor: green, textColor: #c9cc16, width: 100px, height: 50px)
JavaFX Checkbox Label (Font familiy: Bell MT, Italic, Size: 12px, Color: #000000)

10. Create UI of “Game of Fifteen” using **HBox** and **GridPane** layout