Student ID:	Q1
Student Name:	Q2
Student Signature:	Q3a
	Q3b
	Q3c
	Q4a
	Q4b
	Q5_________
	Total

CENG 431 Building Software Systems
Midterm Exam

April 18, 2022

Closed Book Exam. Answer the following questions in 120 minutes.

Questions:

1. **List the 5 of the General Responsibility Assignment Software Patterns (GRASP) and explain each using maximum 5 sentences (15 points).**

   Information expert: It is the most basic responsibility assignment principle. It assigns a responsibility to the class that has the information necessary to fulfill it.

   Creator: Creator is a GRASP Pattern which helps to decide which class should be responsible for creating a new instance of a class.

   High Cohesion: High cohesion is an evaluative pattern that attempts to keep objects appropriately focused, manageable and understandable.

   Low Coupling: Low coupling is an evaluative pattern that dictates how to assign responsibilities to support
   - **lower dependency between the classes**
   - **change in one class having a lower impact on other classes**
   - **higher reuse potential**

   Controller: The controller pattern assigns the responsibility of dealing with system events to a non-UI class that represents the overall system or a use case scenario. A controller object is a non-user interface object responsible for receiving or handling a system event.

2. **List SOLID principles (10 points) and give an example for implementation of the "O" principle in Java (15 points).**

   Single Responsibility

Open/Closed

Liskov Substitution

Interface Segregation

Dependency Inversion

```
public class Shape {

	private String shapeType;
	private double shapeParameter;

	public String getShapeType() {
		return shapeType;
	}

	public void setShapeType(String shapeType) {
		this.shapeType = shapeType;
	}

	public double getShapeParameter() {
		return shapeParameter;
	}

	public void setShapeParameter(double shapeParameter) {
		this.shapeParameter = shapeParameter;
	}

	public double calculateArea() {

		if (shapeType.equals("circle")) {
			return Math.PI * Math.pow(shapeParameter, 2);
		} else if (shapeType.equals("square")) {
			return Math.pow(shapeParameter, 2);
		}

		return -1;
	}

	}
```

The above example violates the open/closed principle since when we want to add a new type of shape such as Rectangle, we need to modify the if/else statements of the method calculateArea().

3. **Design the following domain in UML with class diagram (15 pts):**

- **A paper entry is either a paragraph, figure, or table.**
- **A paper entry is presented in a section.**

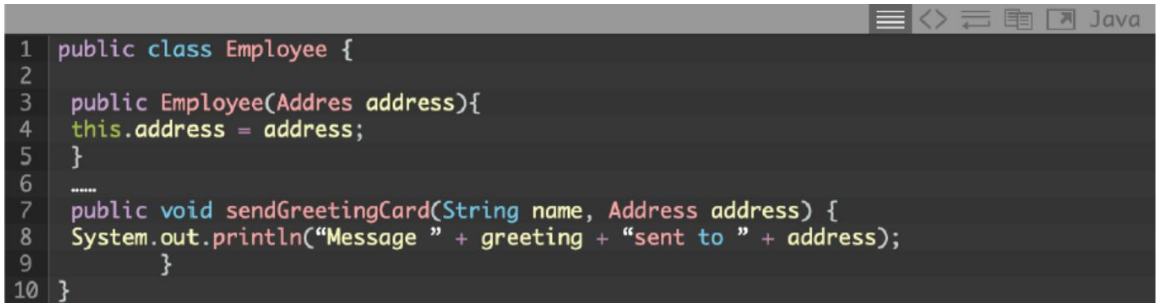
- **A section can contain both paper entries and other sections.**

Write necessary classes in Java (15 pts). Draw sequence diagram for listing the number of paragraphs, figures, and tables in a paper (10 pts).

4. **What is the goal behind Dependecy Injection (10 points)? Explain implementation of Dependecy Injection with one example in Java (10 points).**

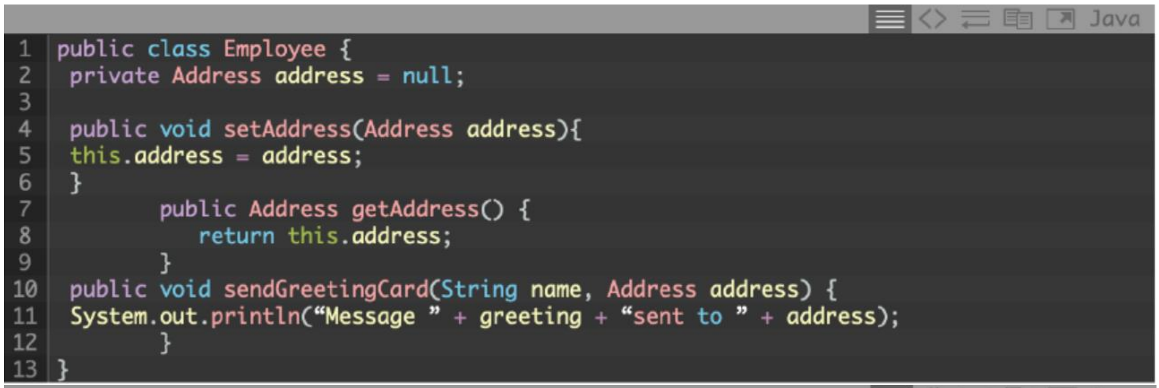
The goal of the dependency injection technique is to allow the creation of dependent objects outside of a class and provides those objects to a class through different ways.

Assume that there are Address and Employee classes. Instead of Employee class creates the Address object, one of the three types of injections given below is applied.



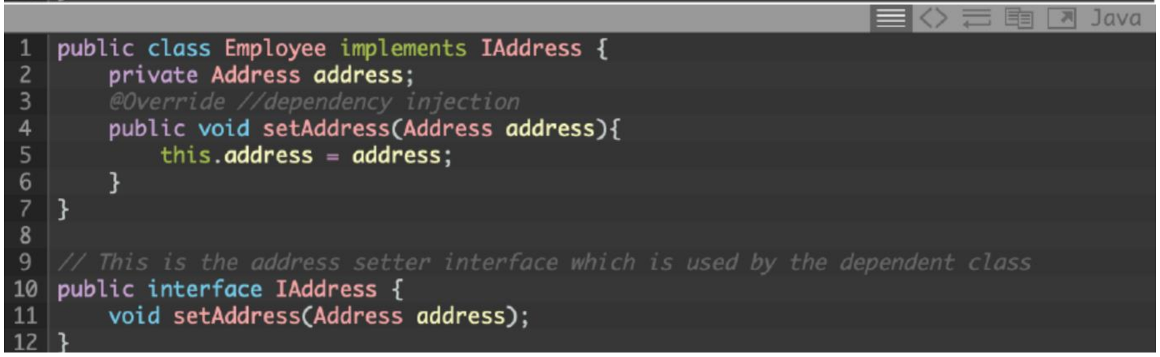
```java
public class Employee {

 public Employee(Addres address){
 this.address = address;
 }
 ......
 public void sendGreetingCard(String name, Address address) {
 System.out.println(“Message ” + greeting + “sent to ” + address);
        }
}
```



```java
public class Employee {
 private Address address = null;

 public void setAddress(Address address){
 this.address = address;
 }
        public Address getAddress() {
           return this.address;
        }
 public void sendGreetingCard(String name, Address address) {
 System.out.println(“Message ” + greeting + “sent to ” + address);
        }
}
```



```java
public class Employee implements IAddress {
    private Address address;
    @Override //dependency injection
    public void setAddress(Address address){
        this.address = address;
    }
}

// This is the address setter interface which is used by the dependent class
public interface IAddress {
    void setAddress(Address address);
}
```

5. **Compare UML sequence diagram to UML state diagram using maximum 3 sentences (10 points).**

   A sequence diagram typically shows the execution of a particular use case for the application and the objects that are involved in carrying out that use case. It could either show a single path, or all of the various paths, through the use case, starting with an actor (user, external system, event) initiating an action.

   State diagrams show the various states that are valid for an object. This type of diagram shows what actions are valid for a given object, depending on what state it is currently in.