The purpose of this assignment is to learn how to implement annotations in Java and use reflection to access annotations.

Exercise 1 (acquire the expertise) – CodeGen annotations

The objective of the exercise is to create a program that, given a Java class with annotated fields, generates and prints to console the source code of the corresponding encapsulated Java class.

Example:

Original class

```
public class Example {
  public float ignoredField = 4.2f;

  @Extract
  public int theAnswer = 42;

  @Extract(name = “foo”)
  private String hello = "world";
}
```

Encapsulated class

```
public class Example {
  private int theAnswer = 42;
  private String foo = "world";

  public int getTheAnswer() {
    return theAnswer;
  }

  public String getFoo() {
    return foo;
  }

  public void setTheAnswer(int theAnswer) {
    this.theAnswer = theAnswer;
  }

  public void setFoo(String foo) {
    this.foo = foo;
  }
}
```

Open the *codegen/* folder contents in your IDE. Inside this directory, you'll find:

- the *Main.java* class, which must be completed to read the annotations and generate the encapsulated class,
- the *Extract.java* interface, which is intended to become the annotation,
- the *Example.java* class that includes annotated fields, serving as a test bed for your implementation.

The *Extract* annotation has to be used to annotate those fields that should be present in the generated class and must also provide an annotation parameter (element declaration) to optionally rename fields in the generated class.

To solve the exercise, first implement the *Extract* annotation, by following these steps:

- Introduce the @interface keyword
- Specify the annotation’s @Target
- Specify the annotation’s @Retention
- Introduce the annotation parameter “name” (element declaration) as a String which defaults to empty String (“”)

Finally, implement the *generateImplementation* method in the *Main.java* class to extract the annotations using reflection and generate the output class. The *Main.java* class already contains some template strings (*fieldTemplate, fieldStringTemplate, getterTemplate, setterTemplate*) that can be used to generate fields and methods.

Exercise 2 (problem solving) – Markdown annotations

The objective of this exercise is to develop a program that generates a draft documentation, in Markdown format, for a Java class in which the annotations *@MarkdownDoc* and *@MarkdownDocIgnore* are used. For example, a possible output could be:

```
# Class `assignment05.markdown.Coordinate`
Parent class: `java.lang.Object`
## Interface(s)
 - `java.lang.Comparable`
 - `java.io.Serializable`
## Fields(s)
 - `float lat`
 - `float lon`
## Constructor(s)
 - `assignment05.markdown.Coordinate(float, float)`
## Methods(s)
 - `float getLon()`
 - `boolean isValidLat(float)`
 - `float getLat()`
 - `boolean isValidLon(float)`
 - `int compareTo(java.lang.Object)`
 - `int compareTo(assignment05.markdown.Coordinate)`
 - `double distance(assignment05.markdown.Coordinate)`
```

Open the *markdown/* folder contents in your IDE. It contains an example of an annotated class (*Coordinate.java*), two interfaces that have to be modified into annotations (*MarkdownDoc.java* and *MarkdownDocIgnore.java*) and the *MarkdownGenerator.java* class that has to be implemented to extract the annotations and generate the required output.

When implementing the annotations, please follow these guidelines:

The *@MarkdownDoc* annotation is exclusively applicable to classes and serves to identify classes that should be **analyzed** for documentation. It should also provide parameters to indicate which elements within those classes have to be documented. This includes options for documenting the parent class, interfaces, fields, constructors, and methods. For each of these options, parameters in form of boolean element declarations should be provided, all initially set to true as the default value.

The *@MarkdownDocIgnore* annotation, on the other hand, can only be applied to fields, constructors, and methods. It is used to **annotate** elements that should not be included in the documentation.

Exercise 3 (optional deepening) – Validation

The objective of this exercise is to implement a program that uses annotations to validate the mathematical correctness of methods with parameters of type int and return types of type int.

Consider the following example:

```
@Validate({
    @ValidationItem(params = {1, 2}, result = -1),
    @ValidationItem(params = {-1, 1}, result = -2)
})
public static int sub(int a, int b) {
    return a - b;
}
```

The *sub* method needs to be validated using all the *ValidationItem* annotations, each of which specifies method parameter values in the correct order, along with the expected result. In the first iteration, the program should test the *sub* method with parameters 1 and 2. It should then invoke the method using reflection and compare the obtained result with the expected result specified in the *ValidationItem*

annotation, which is -1. In the second iteration, parameters -1 and 1 should be used to invoke the method and compare the obtained result with -2, which is the expected value provided by the *ValidationItem*.

The `validation/` folder contains initial interfaces for implementing the annotations *@Validate* and *@ValidationItem*, along with an example utility class called *MathOperations* that makes use of these annotations.