Java Workbook

10. Reflections

~by Andrzej "Andret2344" Chmiel

Exercise 1: Mutable immutability

Given the following code:

```
class MainMutableImmutability {
    public static void main(final String[] args) {
        final String text = "bike";
        reverse(text);
        System.out.println(text);
    }
}
```

Create the missing `reverse` method so the code compiles and prints out this:

```
ekib
```

Do not change the body of the `main` method.

Exercise 2: Test engine

Create an annotation `@MyTest` that behaves likes a custom test engine and will run all the annotated methods from the given class, and a main class that runs the test methods. Given the following code:

```
class TestSuite {
    // Test is SUCCEEDED
    @MyTest
    public void testSoemthing() {
        System.out.println("I'm testing something!");
    }

    // Test is SUCCEEDED
    @MyTest(iParams = {1, 2, 4})
    public void testWithIParam(final int param) {
        System.out.printf("I was invoked with an integer parameter: %s\n", param);
    }

    // Method not invoked
    public void notATest() {
        System.out.println("I'm not a test.");
    }

    // Test FAILED
    @MyTest
    public void imFailue() {
        throw new NullPointerException();
    }

    // Test SUCCEEDED
    @MyTest(expected = ArrayIndexOutOfBoundsException.class)
    public void imThrowingAIOOBE() {
        new int[]{1, 2, 3}[7] = 0;
    }

    // Test FAILED
    @MyTest(expected = NullPointerException.class)
    public void imNotThrowingNPE() {
        new int[]{1, 2, 3}[7] = 0;
    }
}
```

All the annotated methods should run and the main class should inform whether the test method ran successfully or not (i.e., a thrown exception). In the case of parameters, the method should be invoked for each of them.

Exercise 3: Class commenter

Create a `@Commenter` compile annotation that analyses an annotated class and generates a separate class that includes comments regarding the annotated class.

```
//Przykładowa klasa
@Commenter
public class App {
    public int intField;
    public boolean booleanField;
    public char charField;
    public float floatField;
    public Object objectField;
    public String stringField;
    public int[] intArrayField;

    public App() {
        System.out.println("Initializes an object");
    }

    public static List<App> create(final int count) {
        Stream.generate(App::new)
                .limit(count)
                .toList();
    }
}
```

An example output for the class above:

```
public class AppCommenter {
    // The element intField is a FIELD
    // The element booleanField is a FIELD
    // The element charField is a FIELD
    // The element floatField is a FIELD
    // The element objectField is a FIELD
    // The element stringField is a FIELD
    // The element intArrayField is a FIELD
    // The element App() is a CONSTRUCTOR
    // The element create(int) is a METHOD
}
```

Exercise 4: Prioritizing

Given the following code that calls methods from the `MyClass` class in random order using the reflection mechanism.

```
class MainPrioritizing {
    public static void main(String[] args) throws ReflectiveOperationException {
        final Method[] methods = MyClass.class.getDeclaredMethods();
        final MyClass instance = new MyClass();
        for (final Method method : methods) {
            method.invoke(instance);
        }
    }
}

class MyClass {
    public void test1() {
        System.out.println("test1");
    }

    public void test2() {
        System.out.println("test2");
    }
    public void test3() {
        System.out.println("test3");
    }

    public void test4() {
        System.out.println("test4");
    }

    public void test5() {
        System.out.println("test5");
    }
}
```

Create a solution that uses annotations to prioritize methods' calls. Include that:

- The priority value must be passed as the annotation's parameter.
- The priority value can be skipped, then it is considered to be neutral (0).
- The annotation can be skipped, then the priority value is considered to be neutral (0).
- If multiple methods have the same priority value, the order doesn't matter.

Plus, there should be an annotation that allows ignoring a method, that is not calling it. Don't modify the methods in `MyClass`, just add them annotations.

Exercise 5: Forbidden overloading

Given the following code:

```
class OverloadingForbiddenException extends Exception {
    public OverloadingForbiddenException(final String message) {
        super(message);
    }
}

class OverloadingChecker {
    public static <T> void check(final Class<T> clazz)
            throws OverloadingForbiddenException {

    }
}

class MainForbiddenOverloading {
    public static void main(final String[] args){
        try {
            OverloadingChecker.check(TestClass.class);
        } catch (OverloadingForbiddenException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Fill in the `check` method in the `OverloadingChecker` class so that it checks if any of methods in the given `clazz` is overloaded and throws an exception if so.

Exercise 6: Forbidden overriding

Given the following code:

```
class OverridingForbiddenException extends Exception {
    public OverridingForbiddenException(final String message) {
        super(message);
    }
}

class OverridingChecker {
    public static <T> void check(final Class<T> clazz)
            throws OverridingForbiddenException {

    }
}

class MainForbiddenOverriding {
    public static void main(final String[] args){
        try {
            OverridingChecker.check(TestClass.class);
        } catch (OverloadingForbiddenException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Fill in the `check` method in the `OverridingChecker` class so that it checks if any of methods in the given `clazz` is overriden and throws an exception if so.

Exercise 7: Deserializing

An object of some class has been serialized to the `object.obj` file that content looks like this:

```
aced 0005 7372 000d 7a61 6430 342e 5374
7564 656e 7454 9046 575d c8ca 4402 0003
4900 0361 6765 4c00 0966 6972 7374 4e61
6d65 7400 124c 6a61 7661 2f6c 616e 672f
5374 7269 6e67 3b4c 0008 6c61 7374 4e61
6d65 7100 7e00 0178 7000 0000 1c74 0006
5061 7765 c582 7400 0642 6f67 6461 6e
```

Unfortunately, the class got lost. Using the reflection mechanism, deserialize the object and print it out.

Exercise 8: Various cases of `IOException`

Throw as many exceptions being subclasses of the `IOException` as possible, not importing any of them. In the main method, use `throws Exception`.

Exercise 9: Various strange cases of `IOException`

Throw as many exceptions being subclasses of the `IOException` as possible, not importing any of them, but using the `throw` keyword. In the main method, use `throws Exception`.

Exercise 10: Going upstairs

Given the following code:

```
class MainGoingUpstairs {
    public static void main(final String[] args) {
        upstairs(FileNotFoundException.class);
        upstairs(String.class);
        upstairs(StackOverflowError.class);
        upstairs(BufferedInputStream.class);
    }

    private static void upstairs(final Class<?> clazz) {

    }
}
```

Fill in the `upstairs` method, so it prints out all the ancestors of the given class from the given class (the youngest) to the oldest, finishing on `"null"` if no parent is available. So, running the code prints out this:

```
FileNotFoundException -> IOException -> Exception -> Throwable -> Object -> null
String -> Object -> null
StackOverflowError -> VirtualMachineError -> Error -> Throwable -> Object -> null
BufferedInputStream -> FilterInputStream -> InputStream -> Object -> null
```