Java Workbook

02. Collections and generics

~by Andrzej "Andret2344" Chmiel

Exercise 1: Sorting

Given the following code:

```
class MainLengthSorter {
    public static void main(final String[] args) {
        final List<String> strings = List.of("john", "smith", "tom",
                "alice", "rick", "ann", "java", "c", "painter");
        strings.sort(/* HERE */ );
        for (final String s : strings) {
            System.out.println(s);
        }
    }
}
```

Fill in the pinpointed place, so it prints this:

```
c
ann
tom
java
john
rick
alice
smith
painter
```

Exercise 2: Dictionary

Given the following code:

```
class MainDictionaries {
    public static void main(final String[] args) {
        final PolishDictionary p1 = new PolishDictionary();
        final int beginningSerialNumber = p1.getSerialNumber();
        final PolishDictionary p2 = new PolishDictionary();
        if (beginningSerialNumber != p1.getSerialNumber()) {
            System.out.println("ERROR");
        }
        if (p1.getSerialNumber() == p2.getSerialNumber()) {
            System.out.println("ERROR");
        }
        if (PolishDictionary.getUserCounter() != 2) {
            System.out.println("ERROR");
        }
        if (AbstractDictionary.getUserCounter() != 2) {
            System.out.println("ERROR");
        }
        final AbstractDictionary d1 = new AbstractDictionary() {
            @Override
            public String translate(String word) {
                return null;
            }
        };
        if (beginningSerialNumber != p1.getSerialNumber()) {
            System.out.println("ERROR");
        }
        if (PolishDictionary.getUserCounter() != 2) {
            System.out.println("ERROR");
        }
        if (AbstractDictionary.getUserCounter() != 3) {
            System.out.println("ERROR");
        }
        if (d1.getSerialNumber() == p1.getSerialNumber()) {
            System.out.println("ERROR");
        }
        if (d1.getSerialNumber() == p2.getSerialNumber()) {
            System.out.println("ERROR");
        }
        System.out.println("OK");
    }
}
```

Create classes so that this code compiles and prints no errors.

An `AbstractDictionary` is an abstract class that contains a field `dict` that:

- is of type `Map<String, String>` (Polish version is the key, English version is the value).
- points to a single object on the memory heap, and it mustn't change.
- is available for inheriting classes
- is initialized only once with pairs: `północ-north`, `południe-south`, `wschód-east`, `zachód-west`.

The `AbstractDictionary` class additionally contains methods:

- `abstract String translate(String)`,
- `int getSerialNumber()` that returns a unique immutable number for every `AbstractDictionary` object
- `static int getUserCount()` that returns the number of `AbstractDictionary` objects independently on its implementations

A `PolishDictionary` class inherits from the `AbstractDictionary` class and contains methods:

- `String translate(String)` that returns the English version of the given Polish word,
- `static int getUserCount()` that returns the number of `PolishDictionary` objects independently on its implementations

Exercise 3: Incorrect inheritance

Fix the following code so it is compliant with OOP paradigms:

```
record Student(String firstName, String lastName, int age) {
}

class Course extends ArrayList<Student> {
    private final String name;

    public Course(final String name) {
        this.name = name;
    }

    public void addStudent(final Student student) {
        add(student);
    }

    public void printStudents() {
        forEach(System.out::println);
    }
}

class MainWrongInheritance {
    public static void main(final String[] args) {
        final Student s1 = new Student("John", "Smith", 44);
        final Student s2 = new Student("Rick", "Ricky", 33);
        final Course course = new Course("Literature");
        course.addStudent(s1);
        course.addStudent(s2);
        course.printStudents();
    }
}
```

Exercise 4: List filtering

Given the following code:

```
class MainCaseInsensitiveCollections {
    public static void main(final String[] args) {
        /* HERE */
        caseInsensitiveStrings.add("john");
        caseInsensitiveStrings.add("mouse");
        caseInsensitiveStrings.add("JAVA");
        caseInsensitiveStrings.add("JOHN");
        caseInsensitiveStrings.add("MONKEY");
        caseInsensitiveStrings.add("java");
        caseInsensitiveStrings.add("python");
        caseInsensitiveStrings.add("monkey");
        for (final String text : caseInsensitiveStrings) {
            System.out.println(text);
        }
    }
}
```

Fill in the code in the pinpointed place, so it compiles and prints out this:

```
python
mouse
MONKEY
john
JAVA
```

Exercise 5: A vet

Create a `Vet` class that allows storing scheduled visits in one place. The field should be named `scheduler` and have the type `Map`. The `Vet` class should contain the following methods:

- `makeSingleAppointment`
- `makeManyAppointments`
- `getUnmodifiableScheduler`
- `getScheduler`

The following code should compile and work correctly.

```
final Vet vet = new Vet();
final Dog dog = new Dog();
vet.makeSingleAppointment(LocalDateTime.of(2016, 2, 29, 12, 0, 0), dog);

final Map<LocalDateTime, Cat> cats = new HashMap<>();
cats.put(LocalDateTime.of(2016, 2, 29, 12, 0, 1), new Cat());
cats.put(LocalDateTime.of(2016, 2, 29, 12, 0, 2), new Cat());
vet.makeManyAppointments(cats);

vet.getScheduler()
        .put(LocalDateTime.of(2016, 2, 29, 12, 0, 4), new Snake());

// should throw an exception
vet.getUnmodifiableScheduler()
        .put(LocalDateTime.of(2016, 2, 29, 12, 0, 3), new Snake());

final Animal a = vet.getUnmodifiableScheduler()
        .get(LocalDateTime.of(2016, 2, 29, 12, 0, 4));
System.out.println(a);

final Object o = vet.getScheduler()
        .get(LocalDateTime.of(2016, 2, 29, 12, 0, 4));
System.out.println(o);

// should not compile
Animal a = vet.getScheduler()
        .get(LocalDateTime.of(2016, 2, 29, 12, 0, 4));
```

Exercise 6: Storage

Create a `Storage` class that allows storing values assigned to keys. Each key should have assigned multiple values. The `Storage` class should contain the following methods:

- `addToStorage(String key, String value)` - adds the `value` to the `key`.
- `findValues(String key)` - returns all values for the given `key`.
- `findKeys(String value)` - returns all keys that have the given `value` assigned.
- `countValues()` - returns the count of all values in the storage.
- `countUniqueValues()` - returns the count of all unique values in the storage.

So running the following code:

```
class MainStorage {
    public static void main(final String[] args) {
        final Storage storage = new Storage();
        storage.addToStorage("key1", "value1");
        storage.addToStorage("key1", "value2");
        storage.addToStorage("key2", "value1");
        storage.addToStorage("key2", "value2");
        storage.addToStorage("key3", "value3");
        System.out.println(storage.findValues("key1"));
        System.out.println(storage.findKeys("value2"));
        System.out.println(storage.countValues());
        System.out.println(storage.countUniqueValues());
    }
}
```

prints out this:

```
[value1, value2]
[key1, key2]
5
3
```

Exercise 7: What the generics

Given the following code:

```
class MainGenerics {
    public static void main(final String[] args) {
        final List<Integer> ints = Arrays.asList(1, 2, 3);
        System.out.println(sum(ints));
        final List<Number> numbers = new ArrayList<>();
        numbers.add(1.5);
        numbers.add(2.7);
        System.out.println(sum(numbers));
        add(numbers, 7);
        System.out.println(nums);
    }

    private static void add(final List<? extends Integer> numbers, final int n) {
        for (int i = 0; i < n; i++)
            numbers.add(i);
        }
    }

    private static int sum(final List<? super Integer> numbers) {
        int result = 0;
        for (final int i : numbers) {
            result += i.intValue();
        }
        return result;
    }
}
```

Not changing the `main` method, fix the `add` and `sum` methods so the code compiles and prints out this:

```
6
3
[1.5, 2.7, 0, 1, 2, 3, 4, 5, 6]
```

Exercise 8: Values joining

Create a generic class `Joiner<T>` that in its only constructor accepts a separator as a `String` and contains a `join` method that allows passing any number of `T` values. The method should return a string representation of all given arguments joined with the separator. So running the following code:

```
class MainJoining {
    public static void main(final String[] args) {
        System.out.println(new Joiner<>("-").join(1, 2, 3, 4));
    }
}
```

prints out this: `1-2-3-4`.

Exercise 9: Two constructors

Given the following class:

```
class Reservation {
    private final String title;
    private final double price;
    private final List<String> names;
}
```

Create constructors, so objects can be created in both ways:

```
new Reservation("Le petit prince", 99.99, List.of("J. K. Rowling", "R. Riordan"));
new Reservation("Le petit prince", 99.99, "J. K. Rowling");
```

Use as little assignments as possible.
* Try to achieve the same result, by converting this class to a record.

Exercise 10: Comparable lists

Implement the `ListUtil` class that is used to manage lists of comparable objects. The class should contain:

- A constructor accepting a list as a parameter.
- Methods `lessThan`, `equalTo`, and `greaterThan` that accept one object of type `E` and return `List<E>` matching the given condition.

So running the following code:

```
class MainComparableLists {
    public static void printList(final List<?> list) {
        final String result = list.stream()
                .map(String::valueOf)
                .collect(Collectors.joining(", "));
        System.out.println(result);
    }

    public static void main(final String[] args) {
        final List<Integer> list = Arrays.asList(10, 23, 22, 54, 0, -26, 2314, 1000);
        ListUtil<Integer> listUtil = new ListUtil<>(list);
        System.out.print("Lower than 23: ");
        printList(listUtil.lowerThan(23));
        System.out.println();
        System.out.print("Equal to 10: ");
        printList(listUtil.equalTo(10));
        System.out.println();
        System.out.print("Greater than -25: ");
        printList(listUtil.greaterThan(-25));
        /* should not compile */
        // final List<int[]> list2 = Arrays.asList(new int[0], new int[]{1});
        // final ListUtil<int[]> listUtil2 = new ListUtil<>(list2);
    }
}
```

prints out this:

```
Lower than 23: 10, 22, 0, -26,
Equal to 10: 10,
Greater than -25: 10, 23, 22, 54, 0, 2314, 1000,
```