

Laboratory 4, Reflection

Main goal

Students will learn about:

- **Nested, inner and anonymous classes** – how to use them, why they can be useful in Java (increasing encapsulation, multiple inheritance, substitute for lambdas)
- **Object lifecycle** – references, initializing classes, `java.lang.ref` package, memory model and memory leaks in Java, garbage collection
- **Flyweight design pattern**
- **Reflection mechanism in Java** – how to use, writing custom annotations, common use cases

Required for the lab

- Basics of object-oriented programming in Java
- Difference between nested (static), inner (non-static) and anonymous classes
- Purpose and capabilities of different kinds of nested classes
- Flyweight design pattern – motivation and general idea
- Reflection mechanism
 - uses and drawbacks
 - `.class` object
 - annotations (`@Override`, `@Deprecated`, writing custom ones)

Plan

In each section you have chunks of code and several unit tests prepared. During the classes you will be provided the address to the repository containing the code package for this lab. Your task is to fill in gaps in the code according to the instruction, so existing tests will pass.

Nested classes

- **Nested (static and non-static) classes**

Java allows defining classes inside other classes and they are called nested classes. They're divided in two categories:

- Static – called static nested classes
- Non-static – called inner classes.

A nested class is a member of its enclosing class. Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private. Static nested classes do not have access to other members of the enclosing class.

They should be used to logically group used classes in only one place, increase encapsulation and create more readable code.

Your task is to refactor classes `Catalogue` and `Position`. `Position` class is used only in `Catalogue`. `Position` should be declared as a nested class in `Catalogue`. You should decide whether this should be static or non-static class.

- **Increasing readability by moving parts of a class to a nested class**

Class `Geometry` contains a list of x and y coordinates. They represent points which build the geometry. Operating on separate x and y coordinates is not intuitive and can lead to errors because they refer to the same point, but are accessed separately.

Extract inner class `Point` and move all logic operating on separate coordinates to that class. It should behave exactly the same as before.

- **Anonymous classes**

Sometimes there is a need to override some behavior in a class only in one place, without declaring any instances. One of common uses is to create so called comparator for a sorting algorithm, which defines how elements should be compared to determine their order. Your task is to create `sortByLength` method which will sort list of `Strings` using custom comparator.

```
public void sortByLength(List<String> strings);
```

The comparator should be implemented as anonymous class implementing `Comparator` interface. You should use `Collections.sort(List T, Comparable c)` method for sorting. Strings should be sorted by their length, from the shortest to longest ones.

Object lifecycle

Objects created using `new` operator are allocated on heap. They're never explicitly released – Garbage Collector is doing this when memory is needed and objects are unreachable. In short, it's checked if there are any references to the object and if there aren't any, then the object may be removed.

- **Finalize method**

It's a method defined in `Object`, similar in behavior to destructors in C++. However, unlike in C++, it's never known when it's going to be executed – it's executed only by Garbage Collector which behavior is nondeterministic. It's described to be “invoked if and **when** the Java™ virtual machine has determined that there is no longer any means by which this object can be accessed by any thread that has not yet died”, which means that there are cases when it's never executed – for example, if there's enough memory available throughout whole program execution. Java provides other means of cleaning up which will be described on next laboratories.

There are however some cases when it can be useful – for example, if your application is long-running (so GC is more likely to run) and you want to provide **extra safety** and check if all resources are released.

Class `OutputStreamOperator` contains buggy code which does not always close `OutputStream` that it's operating on. Write `finalize()` method which closes that stream if it's not closed yet.

Flyweight design pattern

Flyweight pattern is primarily used to reduce the number of objects created, to decrease memory footprint and increase performance.

Flyweight pattern tries to reuse already existing similar kind objects by storing them and creates new object when no matching object is found.

To apply the pattern, you need to create Flyweight factory which returns shared objects. Attached source code contains simple implementation of application drawing shapes – lines and squares. They both implement `Shape` interface. Square contains property defining whether it should be filled with color; line doesn't contain any such property. There are three possible shapes to draw – `SQUARE`, `SQUARE_FILLED` and `LINE`.

`ShapeFactory.getShape(ShapeType st)` method returns new `Shape` instance every time the method is called. Fix the implementation so it will keep created objects cached and when `getShape` is executed many times for the same type (for example `LINE`) it will return the same object. What are the benefits of this approach?

Add another method

```
public Shape getShape(ShapeType st, String color)
```

which behaves in the same way as `getShape`, but caches instances for every color separately.

Reflection

Reflection is a mechanism that allows programs to examine (introspect) or modify the runtime behavior of applications running in the JVM. It is a powerful tool, allowing to change existing code however it should be used with care – it allows breaking contracts in program (for example, you can overwrite access modifiers) and has large negative impact on performance.

- **@Override annotation**

`@Override` is an annotation marking that following method overrides a method from its superclass. The method has to have exactly the same prototype as method in superclass. It is optional when overriding methods.

```
public class OverrideAnnotation {
    @Override
    public String toString(String value) {
        return value;
    }
}
```

What is the problem in this class? Why it doesn't compile? How `@Override` annotation helps to discover potential bugs and what would happen without it?

- **Custom annotations**

Java allows to create own annotations, providing information about classes, methods and fields. Create annotation `@Copyright` taking one parameter, describing copyright info. Write a piece of code showing all method names in given class with their copyright information. Example usage of your annotation:

```
class Example {
    @Copyright(info = "(2009) Luxoft Inc.")
    public void test() {}
}
```

- **Accessing private methods and members**

Consider the following class:

```
public class Authenticator {
    private boolean authenticated = false;
    public void authenticate(String user, String password) {
        if (isValid(user, password)) {
            authenticated = true;
        }
    }
    public boolean isAuthenticated() {
        return authenticated;
    }
    // ...
}
```

Class `Authentication` can be used by an external system to authenticate users. It contains private field `authenticated`, containing `true` if the user is authenticated and `false` otherwise. At first, user is not authenticated and the only way to authenticate is to execute method

`Authenticator.authenticate(String user, String password).`

Your task is to implement method

`AuthenticationOverrider.authenticate()`

which doesn't use `user` or `password`, but after which

`Authenticator.isAuthenticated()`

will return `true`. Do it by two methods:

1. Executing private method
2. Changing private field's value

- **Creating create new instance of a class by its name**

Implement `createInstance` method:.

```
public Object createInstance(String className)
```

Your method should return new instance of class denoted by this `String`.

Hint: use `java.lang.Class` class.

- **Checking object type – instanceof operator**

Write `isNumber` method taking one parameter – `Object`:

```
public boolean isNumber(Object object)
```

The parameter can contain value of any type – `Float`, `List`, `String` etc. The method should return `true` if the value is any kind of number.

Hint – all numeric types (`Float`, `Integer`, ...) extend `Number` class.

Way of getting a credit for the classes

- Completion of tasks in each section
- Taking part in discussions

Literature

1. Nested classes – <http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html>
2. Reflection API – <http://docs.oracle.com/javase/tutorial/reflect/>
3. Java Reflection Tutorial – <http://tutorials.jenkov.com/java-reflection/index.html>
4. Design Patterns: Elements of Reusable Object-Oriented Software – chapter “Flyweight pattern”
5. Flyweight design pattern - <http://javapapers.com/design-patterns/flyweight-design-pattern/>
6. Object lifecycle - http://en.wikibooks.org/wiki/Java_Programming/Object_Lifecycle
7. Object finalization and cleanup - <http://www.javaworld.com/article/2076697/core-java/object-finalization-and-cleanup.html>