06 - Java Reflection and Annotation

Reflections in Java

Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. Encoding the execution state as data is called reification.

Reflection is the composition of:

- Introspection: observe and reason about the state;
- Intercession: modify the state.

There are two types of reflection:

- Structural reflection: form of intercession, can inspect the structure of objects/classes at runtime. Complete reification of both:
  - The program currently executed;
  - The program's abstract data types.
- Behavioural reflection: form of intercession, can modify the behaviour of objects/classes at runtime. Complete reification of both:
  - Semantics and implementation of the language;
  - Data and implementation of the run-time system.

Examples of reflection are debuggers, test tools, visual development environments (e.g. a tool that composes beans in a graphical way)...

Reflection comes with 3 main drawbacks:

- Performance overhead: it is about runtime components, thus it is not possible to statically optimize them;
- Security;
- Exposure of Internals: reflective code may access internals (like private fields), breaking abstraction;

Java supports introspection and reflective invocation, but can't change code (it can find the methods of a class and invoke them, but not change their code). The JVM accomplish this by keeping an object of class `java.lang.Class` associated to every type (primitive, loaded, synthesized), which reflects the type it represents. Class objects are constructed automatically by the JVM as the corresponding classes are loaded. We can retrieve them in 3 ways:

- `Object.getClass()`: e.g. `"foo".getClass();`;
- `type.class`: e.g. `String.class`, `int[][][].class`;
- `Class.forName(String)`: e.g. `Class.forName("java.util.List”);`.

Once we have retrieved a class object, we can retrieve:

- Class name;
- Class modifiers;
- Obtain info on fields, methods and constructors. Each of them we have an associated class (`Field`, `Method`, `Constructor`), which all implements the interface `Member`.

An exception are the `generic` classes: due to Java's type erasure, at the end of the compilation generics are substituted with Object, for retro-compatibility reasons. For example an object `HashSet<String>` will be changed into `HashSet` (without string).

Java's reflection can also be used to:

- Create objects of type that is not known at compile time (e.g. it gets loaded through the internet). E.g.:
  - Using Default Constructors: `Class c = Class.forName(...)` -> `c.newInstance()`;
  - Using Constructors with Arguments: `Constructor ctor = c.getConstructor(argsClass)` (where `argsClass` is an array of classes) -> `ctor.newInstance(args)` (where `args` is an array of objects containing the arguments to pass to the constructor);
- Access members (fields/methods) not known at compile time. E.g.:
  - Getting Field Values: `Field f = c.getField(name)` -> `f.get(i)` (where `i` is an instance of Class `c`);
  - Setting Field Values: same thing, but with `f.set(i, value)`;
  - Invoking Methods: `Method m = c.getMethod(name, params)` -> `m.invoke(i, args)`;

Generally, operations forbidden by privacy rules fail even if invoked through reflection (e.g. changing a final, reading/writing a private field, invoking a private method...). However, if there is no security manager or if the security manager allows it, the programmer can request access to the objects `Field`, `Method` and `Constructor` previously seen, which all inherit from the class `AccessibleObject`. This class contains 3 main methods:

- `isAccessible()`: check if accessible;
- `setAccessible(boolean flag)`: set accessibility;
- `setAccessible(AccessibleObject[] array, boolean flag)`: set accessibility of multiple objects.

Annotations

Annotations are user-defined modifiers (metadata) not embedded in the language, applicable to almost any syntactic element (classes, fields, parameters...).

They are made of:

- Name: mandatory;
- Attributes: optional (key-value) pairs;

Syntax:
``@annName{name_1 = constExp_1, ..., name_k = constExp_k}`

The Java compiler defines and recognizes a small set of predefined annotations:

- `@Override`: explicit override;
- `@Deprecated`: declares that the annotated element is not necessarily included in future releases of the Java API;
- `@SuppressWarnings`: instruct the compiler to avoid issuing warnings;
- Moreover, we can annotate annotations to describe their meta-data:
  - `@Target`: constrains the program elements to which the annotation can be applied;
  - `@Retention`: defines the expire time of the annotation:
    - SOURCE: source code only;
    - CLASS: default, kept by compiler and inserted in the class file;
    - RUNTIME: available at runtime;
  - `@Inherited`: the annotation is inherited by subclasses.

Then, programmers can define new annotations, but they are ignored on compilation. Still, they can be used to:

- Document;
- Static analysis in the bytecode;
- Inspect the annotations placed on a class at runtime thanks to reflection;

To declare a new annotation type, the syntax is similar to interfaces. E.g.:

```java
@interface InfoCode {
        String author ();
        String date ();
        int ver () default 1;
        int rev () default 0;
        String [] changes () default {};
}
```

- Each method determines the name of an attribute and its type (the return type);
- A default value can be specified for each attribute;
- Attribute types can only be `primitive`, `String`, `Class`, `Enum`, `Annotation` or an array of those types.

Annotations can be recovered through Java's Reflection API, through:

- `c.getAnnotations()`: returns the Annotations inside `c`;
- `c.getAnnotation(annotationClass)`: only returns the annotation of the specified type;