Technische Hochschule Rosenheim

Modul - Fortgeschrittene Programmierkonzepte

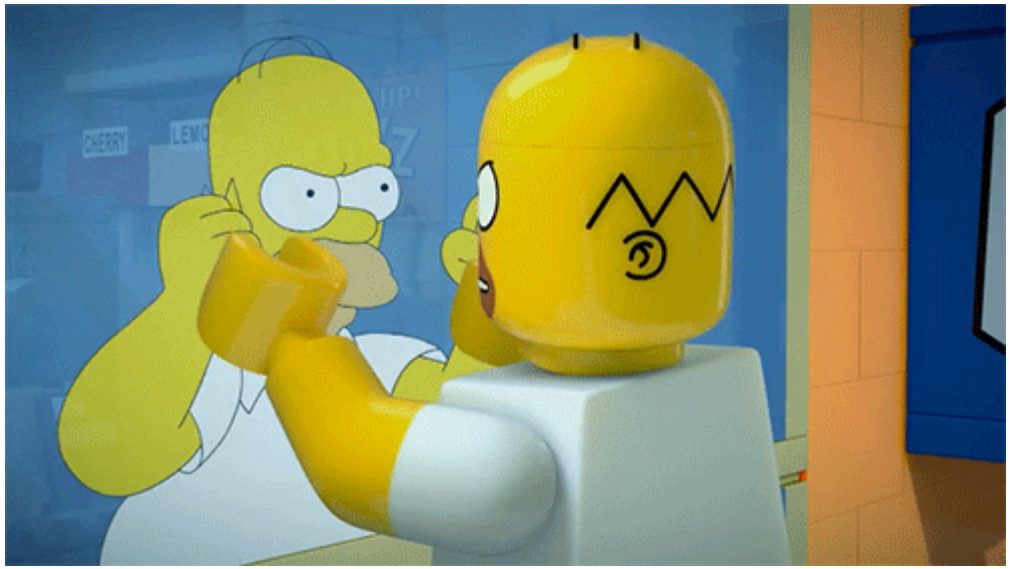
Bachelor Informatik

06 - Reflection und Annotations

Prof. Dr. Marcel Tilly

Fakultät für Informatik, Cloud Computing

Agenda for today



- `java.lang.Class<T>`: your ticket to reflection
- Messing with Objects
- Basic Java beans (a simple plugin architecture)
- Annotations

Reflection

Technische Hochschule Rosenheim

Type introspection is the ability of programming languages to determine the type (or its properties) of an arbitrary object at runtime.

Java takes this concept one step further with *reflection*, allowing not only to determine the type at runtime, but also modifying it.

At its core, reflection builds on `java.lang.Class`, a generic class that *is* the definition of classes.

Note that most of the classes we will be using when dealing with reflection are in the package `java.lang.reflection` (package summary), and (almost) none of them have public constructors -- the JVM will take care of the instantiation.

Classes

Technische Hochschule Rosenheim

There are different ways to get to the class object of a Class:

```
// at compile time
Class<String> klass1 = String.class;

// or at runtime
Class<? extends String> klass2 = "Hello, World!".getClass();
Class<?> klass3 = Class.forName("java.lang.String");
Class<String> klass4 = (Class<String>) new String().getClass();

System.out.println(klass1.toString());  // java.lang.String
System.out.println(klass2.toString());  // ditto...
System.out.println(klass3.toString());
System.out.println(klass4.toString());

klass1.getName();          // java.lang.String
klass1.getSimpleName();    // String
```

Class: Class

Technische Hochschule Rosenheim

As a note ...

- use `Class<T>` to get the class name if known at compile time (or use an unchecked cast at runtime)
- use the `?` (wildcard) and appropriate bounds for *any* type.

So what can you do with a `Class<?>` object?

Basic Type Information

Technische Hochschule Rosenheim

You can use `Class<?>` object to get basic type information:

```
// all of these will return true
int.class.isPrimitive();

String[].class.isArray();

Comparable.class.isInterface();

(new Object() {})
    .getClass()
    .isAnonymousClass();          // also: local and member classes

Deprecated.class.isAnnotation();  // we will talk about those later
```

As you can see, even primitive type like `int` or `float` have class objects (which are, by the way, different from their wrapper types `Integer.class` etc.!).

Type Internals

Technische Hochschule Rosenheim

You can get even more ...

```
// family affairs...
String.class.getSuperClass();    // Object.class

// constructors
String.class.getConstructors();  // returns Constructor<String>[]

// public methods
String.class.getMethod("charAt", int.class);
String.class.getMethods();       // returns Method[]

// public fields (attributes)
// Comparator<String>
String.class.getField("CASE_INSENSITIVE_ORDER");
String.class.getFields();        // returns Field[]

// public annotations (more on this later)
String.class.getAnnotation(Deprecated.class);          // null...
String.class.getAnnotationsByType(Deprecated.class);   // []
String.class.getAnnotations();  // returns Annotation[]
```

Some Remarks

Technische Hochschule Rosenheim

These methods may throw `NoSuch{Method,Field}Exception`s and `SecurityException`s (more on security later).

You can distinguish between *declared* fields (methods, ...), and "just" fields: `.getFields()` (and `.getMethods()` etc.) will return *the public* fields of an object, including those inherited by base classes.

Use `.getDeclaredFields()` (and `.getDeclaredMethods()` etc.) to retrieve *all* fields declared in this particular class.

Object Instantiation

Technische Hochschule Rosenheim

As you can see, you can also query for *constructors* of a class. This is the base for creating new instances based on class definitions:

```
Class<?> klass = "".getClass();
String magic = (String) klass.newInstance();  // unchecked cast...
```

Sticking with the example of strings, the documentation tells us that there exist non-default constructors. Consider for example the `String(byte[] bytes)` constructor:

Object Instantiation

Technische Hochschule Rosenheim

```
Constructor<String> cons = (Constructor<String>)
    String.class.getConstructor(byte[].class);

String crazy = cons.newInstance(new byte[] {1, 2, 3, 4});
```

Note that this may trigger quite a few exceptions: `InstantiationException`, `IllegalAccessException`, `IllegalArgumentException`, `InvocationTargetException`, all of which make sense if you extrapolate their meaning from the name.

Messing with Objects

Technische Hochschule Rosenheim

Modifying Fields (Attributes)

Remember Rumpelstiltskin?

In short: A very ambitious father claimed his daughter could produce gold. Put under pressure by the king, she cuts a deal with an imp: it spins the gold and in return she would sacrifice her first child --

unless she could guess the imp's name!

Objects

Technische Hochschule Rosenheim

Ok, here's the mean imp and the poor daughter:

```
class Imp {
    private String name = "Rumpelstiltskin";
    boolean guess(String guess) {
        return guess.equals(name);
    }
}
```

```
class Daughter {
    public static void main(String... args) {
        Imp imp = new Imp();

        // to save your child...
        imp.guess(???);
    }
}
```

More about it ...

Technische Hochschule Rosenheim

Since `Imp.name` is private, the imp feels safe (it's dancing around the fire pit...). But can we help the daugher save her firstborn?

Yes, we can!

Technische Hochschule Rosenheim

Using reflection, we will sneak through the imp's "head" to find the string variable that encodes the name.

```
Imp imp = new Imp();
String oracle = null;
// get all fields
for (Field f : imp.getClass().getDeclaredFields()) {

    f.setAccessible(true);  // oops, you said private? :-)

    // looking for private String
    if (Modifier.isPrivate(f.getModifiers())
            && f.getType() == String.class) {
        oracle = (String) f.get(imp);  // heureka!
    }
}
imp.guess(oracle);  // true :-)
```

Alternatively

```
Imp imp = new Imp();

for (Field f : imp.getClass().getDeclaredFields()) {
    f.setAccessible(true);
    if (Modifier.isPrivate(f.getModifiers())
        && f.getType() == String.class) {
        f.set(imp, "Pinocchio");  // oops :-)
    }
}

imp.guess("Pinocchio");  // true :-)
```

The `Field` class allows us to retrieve and modify both the modifiers and the values of fields (given an instance).

Calling Functions

Technische Hochschule Rosenheim

Similar to accessing and modifying fields, you can enumerate and invoke methods. Sticking with the imp above, what if the imp's name were well-known, but nobody knew how to ask for a guess?

```
class WeirdImp {
    static final String name = "Rumpelstiltskin";
    private boolean saewlkhasdfwds(String slaskdjh) {
        return name.equals(slaskdjh);
    }
}
```

Accessability

Technische Hochschule Rosenheim

This time, the `name` is well known, but the guessing function is hidden. Again, reflection to the rescue.

```
WeirdImp weirdo = new WeirdImp();
for (Method m : weirdo.getClass().getDeclaredMethods()) {
    m.setAccessible(true);

    // ...returns boolean?
    if (m.getReturnType() == boolean.class
            // ...has one arg?
            && m.getParameterCount() == 1
            // which is String?
            && m.getParameterTypes()[0] == String.class) {
        System.out.println(m.invoke(weirdo, Weirdo.name));
    }
}
```

Basic Java Beans

Technische Hochschule Rosenheim

Reflection can be used to facilitate an architecture where code is dynamically loaded at runtime. This is often called a *plugin mechanism*, and Java Beans have been around for quite a long time.

Consider this simple example: We want to have a game loader that can load arbitrary text-based games which are provided as a third party `.jar` file.

```
package reflection;
public interface TextBasedGame {
    void run(InputStream in, PrintStream out) throws IOException;
}
```

Beans Example

Technische Hochschule Rosenheim

A simple *parrot* (echoing) game could be:

```
package reflection.games;
public class Parrot implements TextBasedGame {
    @Override
    public void run(InputStream in, PrintStream out)
        throws IOException {

        BufferedReader reader = new BufferedReader(
            new InputStreamReader(in));
        out.println("Welcome to parrot. Please say something");

        String line;
        while ((line = reader.readLine()) != null
            && line.length() > 0) {
            out.println("You said: " + line);
        }
        out.println("Bye!");
    }
}
```

Load Class

Technische Hochschule Rosenheim

These games all implement the `TextBasedGame` interface, and their `.class` files can be packaged into a jar.

Later, if you know the location of the jar file, you can load classes by-name:

```
package reflection;
public class TextGameLoader {
    public static void main(String... args) throws Exception {

        // load classes from jar file
        URL url = new URL("jar:file:/...hsro-inf-fpk/example/games.jar!/");
        URLClassLoader cl = URLClassLoader.newInstance(new URL[] {url});

        // you can play "Addition" or "Parrot"
        final String choice = "Parrot";
        TextBasedGame g = (TextBasedGame) cl.loadClass
            ("reflection.games." + choice)
            .newInstance();
        g.run(System.in, System.out);
    }
}
```

Security

Technische Hochschule Rosenheim

The previous sections showed clearly how powerful the tools of reflection are. Naturally, security is a concern: what if someone loads your jars, enumerates all classes, and then tries to steal passwords from a user?

This has indeed been done, and is the reason why Java was historically considered insecure or even unsafe to use. However, newer versions of Java have a sophisticated system of permissions and security settings that can limit or prevent reflection (and other critical functionality).

Two things that do *not* work, at least out of the box:

- While you can do a forced write on `final` *fields*, this typically does not affect the code at runtime, since the values are already bound at compiletime.
- It is impossible to swap out *methods* since class definitions are read-only and read-once. If you wanted to facilitate that, you would have to write your own class loader.

Object Comparison

Technische Hochschule Rosenheim

One last word regarding reflection and object comparison. As you know, we distinguish two types of equality: reference and content-based equality.

```
class K {
    K(String s) {
        this.s = s;
    }
}

int a = 1;
int b = 1;

a == b;  // true: for primitive types, the values are compared

K k1 = new K("Hans");
K k2 = new K("Hans");

k1 == k2;        // false: they point to different memory
k1.equals(k2);   // ???
```

equals()-Method

Technische Hochschule Rosenheim

If you don't overwrite the `.equals` method in your class, the default version `Object.equals` will check for reference equality. If you overwrite it, you need to make sure to

- test for `null`
- test for reference equality (same memory?)
- test for same *type*
- call `super.equals()`, if it was overwritten there
- compare all attributes

equals()-Method

Technische Hochschule Rosenheim

Consider this implementation of `equals`:

```
public boolean equals(Object o) {
    if (o == null) return false;
    if (o == this) return true;   // same memory
    // version A
    if (!this.getClass().equals(o.getClass()))
        return false;
    // version B
    if (this.getClass() != o.getClass())
        return false;
    // version C
    if (!(o instanceof K))
        return false;
    if (!super.equals(o))
        return false;
    // now compare attributes
    return this.s.equals(((K) o).s);
}
```

What works?

Explanation

Here's the question: Which of the versions A, B or C are correct ways to test if the *types* of the two objects match?

A) is correct, since we compare both runtime classes with `.equals`.

B) is correct, since the class objects are shared among all instances (and parametrized generics; recall type erasure).

C) is however *incorrect*: the `instanceof` operator would also return `true` if there is a match on an interface of derived class.

Annotations

Technische Hochschule Rosenheim

Annotations are *meta-information*, they can be attached to classes, methods or fields, and can be read by the compiler or using reflection at runtime.

They are denoted using the `@` character, for example `@Override`.

Defining Annotations

Technische Hochschule Rosenheim

Annotations are similar to interfaces: both as in syntax and as in a method or field can have multiple annotations.

```
public @interface Fixed {
    String author() ;
    String date() ;
    String bugsFixed() default "" ;
}
```

This defines the annotation `@Fixed(...)` with three arguments; the last one is optional and defaults to the empty string.

```
@Fixed(author="mustermann", date="2011-11-11")
void method() { ... }
```

Meta-Annotations

Technische Hochschule Rosenheim

Even annotations can be annotated. *Meta annotations* define where and how annotations can be used.

- `@Target({ElementType.FIELD, ElementType.METHOD})`: Use this to limit your custom annotation to fields or methods.
- `@Retention(RetentionPolicy.{RUNTIME,CLASS,SOURCE)}`: This controls if the annotation is available at runtime, in the class file, or only in the source code.
- `@Inherited`: Use this to make an annotation to be passed on to deriving classes.

Method Annotations

Technische Hochschule Rosenheim

In general, there are *marker anotations* (e.g. `@Deprecated`) without arguments, *value annotations* (e.g. `@SuppressWarnings("...")`) that take exactly one value, and more sophisticated annotations (e.g. `@Fixed(...)` above).

```
class K {
    @Override
    public boolean equals(Object o) {
        // ...
    }
    @Deprecated
    public void useSomethingElseNow() {
        // ...
    }
    @SuppressWarnings("unchecked")
    public void nastyCasts() {

    }
}
```

Marker Annotations

Technische Hochschule Rosenheim

What are they good for?

- `@Override` is used to signal the intent of overwriting; results in compile error if its actually no overwrite (e.g. `@Override public boolean equals(K k)`)
- `@Deprecated` marks a method not to be used anymore; it might be removed in the future.
- `@SuppressWarnings(...)` turns off certain compiler warnings

Technische Hochschule Rosenheim

Type (Attribute) Annotations

`@NonNull`: The compiler can determine cases where a code path might receive a null value, without ever having to debug a `NullPointerException`.

`@ReadOnly`: The compiler will flag any attempt to change the object.

`@Regex`: Provides compile-time verification that a `String` intended to be used as a regular expression is a properly formatted regular expression.

`@Tainted` and `@Untainted`: Identity types of data that should not be used together, such as remote user input being used in system commands, or sensitive information in log streams.

```
abstract void method(@NonNull String value, @Regex re);
```

Annotations: JUnit5

Technische Hochschule Rosenheim

The new JUnit5 test drivers inspect test classes for certain annotations.

```
class MyTest {
    BufferedReader reader;

    @BeforeAll
    void setUp() {
        reader = new BufferedReader();  // ...
    }

    @Test
    void testSomeClass() {
        // ...
    }
}
```

Most of the time, you will get around with `@BeforeAll`, `@AfterAll` and `@Test`; see this complete list of annotations.

Annotations: Gson by Google

Technische Hochschule Rosenheim

Gson by Google helps with de/serializing objects (see today's assignment).

It allows you to map between JSON and Java objects:

```
class Klass {
    private int value1 = 1;
    private String value2 = "abc";
    @SerializedName("odd-name") private String oddName = "1337";
    private transient int value3 = 3;  // will be excluded

    Klass() {
        // default constructor (required)
    }
}
```

How to use it?

```
// Serialization
Klass obj = new Klass();
Gson gson = new Gson();
String json = gson.toJson(obj);
// ==> json is {"value1":1,"value2":"abc","odd-name": "1337"}

// Deserialization
Klass obj2 = gson.fromJson(json, Klass.class);
// ==> obj2 is just like obj
```

Annotations: Butterknife

Butterknife: GUI bindings for Android ridiculously simplified

```
class ExampleActivity extends Activity {
    @BindView(R.id.title) TextView title;
    @BindView(R.id.subtitle) TextView subtitle;
    @BindView(R.id.footer) TextView footer;

    @Override public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.simple_activity);
        ButterKnife.bind(this);
        // TODO Use fields...
    }
}
```

Annotations: Retrofit

Technische Hochschule Rosenheim

Retrofit: consume REST interfaces without any pain

```
public interface GitHubService {
  @GET("users/{user}/repos")
  Call<List<Repo>> listRepos(@Path("user") String user);
}

Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("https://api.github.com/")
    .build();

GitHubService service = retrofit.create(GitHubService.class);

Call<List<Repo>> repos = service.listRepos("octocat");
```

Summary

Technische Hochschule Rosenheim

Lessons learned today:

- Reflections
  - Classes, Methods, Attributes
  - Security
- Annotations
- Frameworks
  - Butterknofe
  - GSON
  - Retrofit

Final Thought!