Metaprogramming

Prof. Dr. Ralf Lämmel
Universität Koblenz-Landau
Software Languages Team

Motivation

Elevator speech

So programs are programs and data is data.
Or perhaps programs could be data?
Or rather data could be programs?
What if programs process data representing programs?
Those are *meta*programs.
(*Reflection* is an important way of doing *metaprogramming*.)
What if data represents data about programs?
This is *meta*data.

Programming is for kids.
Metaprogramming is for nerds.

(C) 2010-2015 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

Metaprogramming terminology

- A ***metaprogram*** **i**s a program that manipulates other programs. That is, programs correspond to data of metaprograms.
- The language in which the metaprogram is written is called the ***metalanguage***. The language of the programs that are manipulated is called the ***object language***.
- The ability of a programming language to be its own metalanguage is called ***reflection*** . If reflection is limited to “read access”, then this is called ***introspection***.
- Reflection can be achieved in very different ways:
  - **API-based access to program (Java).**
  - Special language and compiler support (C++ templates).

(C) 2010-2015 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

Examples of metaprograms

- A Java program that pretty prints arithmetic expressions.
- … evaluates them.
- … transforms them (e.g., optimizes them).
- A Java compiler because:
  - it inspects Java code, and
  - it produces byte code.
- If the Java compiler is written in Java, it is a *reflective* meta-program; if it is written in Haskell instead, it is just a meta-program.
- A byte-code engineering program because it analyses, generates, or transforms programs in JVM byte code.
- …

(C) 2010-2015 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

See package **helloworld** of
https://github.com/101companies/101repo/tree/master/technologies/Java_platform/samples/javaReflectionSamples

The "Hello World" of java.lang.reflect

DEMO

Scenario

- Create an object through reflection API
- Invoke a method through reflection API

Why is this interesting?

- Imagine the class or the method are not known until ***runtime***.

(C) 2010-2014 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

Reflection

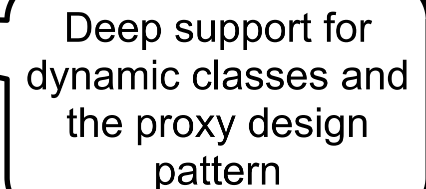
Java's "built-in" reflection

Basics (Covered today)

- **java.lang.Class**
- **java.lang.reflect.Field**
- **java.lang.reflect.Method**
- **java.lang.reflect.Constructor**

Advanced (Optional material included)

- java.lang.reflect.Proxy



(C) 2010-2015 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

java.lang.Class

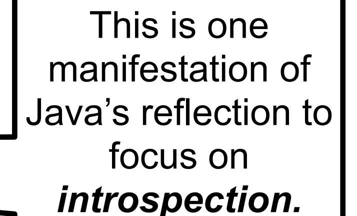
- Represents classes (and interfaces) in a Java app.
- Each object knows of its class: obj.getClass()
- Each class knows of its class: String.class
- Primitive types have an associated class.
- Even void has an associated class.
- Class *cannot be regularly instantiated*.
- Instead the class loader uses defineClass:
  - Input: byte code
  - Output: an instance of type Class

To treat argument and result types homogeneously and precisely

(C) 2010-2015 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

java.lang.Class interface

- Map strings to class objects (forName)
- Getter
  - Name
  - Constructors
  - Fields
  - Methods
  - Interfaces
  - Annotations
  - Package
  - Superclass
  - Enclosing class
- **NO SETTERS**
- Various other inspectors (“tests”)
- Instantiation (but see constructors)
- Cast operations
- …



(C) 2010-2015 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

java.lang.reflect.Method

- Represents methods
  - … of classes and interfaces
  - static, initialization, instance, and abstract methods
- Constructors are treated very similar; see:
  - java.lang.reflect.Constructor
- Does not provide byte-code access.
- Returned by getters on java.lang.Class.
- **NO METHOD BODIES**

(C) 2010-2015 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

java.lang.reflect.Method interface

- Getters
  - Name
  - Parameter types
  - Return type
  - Type parameters
  - Class
  - Modifiers
  - Annotations
  - Parameter annotations
  - …
- ***Invocation***

(C) 2010-2015 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

java.lang.reflect.Field

- Provides information about fields.
- Provides dynamic access to fields.

(C) 2010-2015 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

java.lang.reflect.Field interface

- Getters
  - Name
  - ***Value*** (for a given object)
  - Modifiers
  - Annotations
  - …
- Setters
  - ***Value*** (for a given object)

(C) 2010-2015 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

Reflection examples

See package **dumper** of
https://github.com/101companies/101repo/tree/master/technologies/Java_platform/samples/javaReflectionSamples

An object dumper

DEMO

Scenario

- Given an object of an arbitrary type.
- Provide a deep dump on the state of the object.
- That is, go over the fields recursively.

(C) 2010-2014 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

http://101companies.org/wiki/Contribution:javaReflection

Total and cut
with the help of reflection.

DEMO

Please note the conciseness of the code compared to javaComposition, for example.

(C) 2010-2014 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

A challenge for (Java) reflection

Given a class, what are the subclasses of it?
Alas, there is no such member on “Class”.

The set of classes known to the system is simply the set of classes loaded so far. One could assume a scan over the full classpath to get to know more classes, but automatically loading all these classes is expensive and may not be intended.

(C) 2010-2015 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

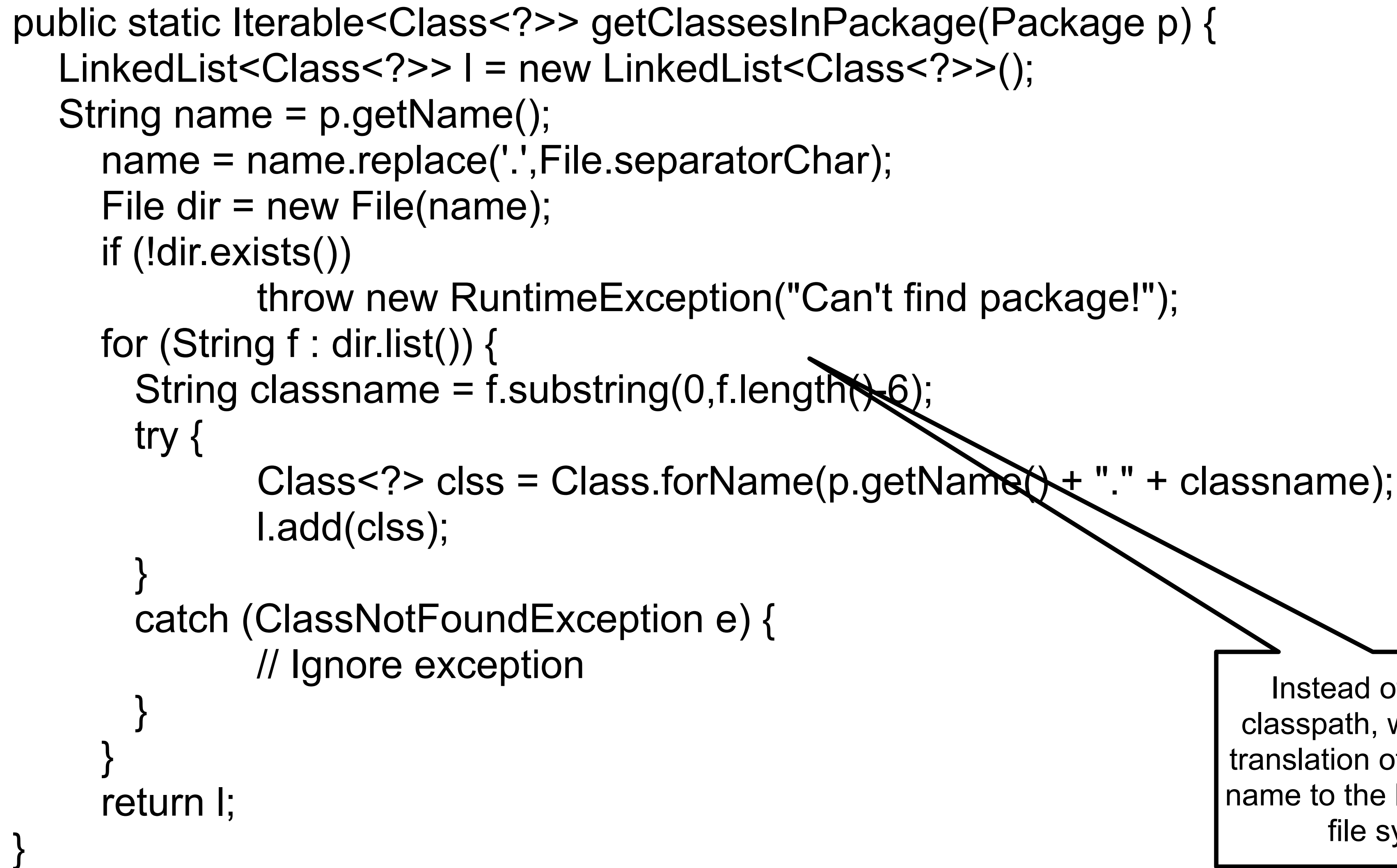
See package **packagelist** of
https://github.com/101companies/101repo/tree/master/technologies/Java_platform/samples/javaReflectionSamples

Find and load all classes of a package

DEMO

(C) 2010-2014 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

Find (and load) all classes in a given package. The search relies on a directory listing.



```java
public static Iterable<Class<?>> getClassesInPackage(Package p) {
    LinkedList<Class<?>> l = new LinkedList<Class<?>>();
    String name = p.getName();
        name = name.replace('.',File.separatorChar);
        File dir = new File(name);
        if (!dir.exists())
                throw new RuntimeException("Can't find package!");
        for (String f : dir.list()) {
            String classname = f.substring(0,f.length()-6);
            try {
                Class<?> clss = Class.forName(p.getName() + "." + classname);
                l.add(clss);
            }
            catch (ClassNotFoundException e) {
                // Ignore exception
            }
        }
        return l;
}
```


(C) 2010-2015 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

Metadata

Metaprograms vs. metadata

- ***Metaprograms*** are programs that generate, analyze, or control other programs.

- ***Metadata*** (generally) is data that is attached to programs or other data to provide additional information (in particular, application-specific information) for programs and data.

- In Java, we can use ***annotations*** for metadata. It happens that metaprograms often need annotations.

(C) 2010-2015 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

Sample annotation

```
@Test
public void testTotal() throws Exception {
    Document doc = DOMUtilities.loadDocument(sampleCompany);
    double total = total(doc);
    assertEquals(399747, total, 0);
}
```

(C) 2010-2015 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

Annotations

Data associated with methods, fields, class, etc. that does not affect the semantics of a program, but controls metaprograms over the annotated object programs.

Scenarios:

- Identify test methods and control execution (JUnit)
- Control serialization of objects as XML (JAXB)
- ... many other frameworks ...

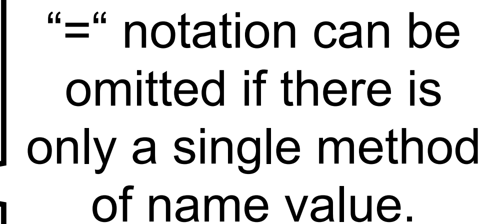
.NET uses the term “custom attributes”; Java “adopted” them and called them annotations

(C) 2010-2015 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

Sample annotation

```
@RequestForEnhancement(
   id       = 2868724,
   synopsis = "Enable time-travel",
   engineer = "Mr. Peabody",
   date     = "4/1/3007"
)

Public void travelThroughTime(Date destination) { ...
}
```



(C) 2010-2015 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

Annotation declaration for sample annotation

```
/**
 * Describes a Request-For-Enhancement(RFE)
 * for the API element at hand
 */

public @interface RequestForEnhancement {
    int    id();
    String synopsis();
    String engineer() default "[unassigned]";
    String date() default "[unimplemented]";
}
```

Thus, annotation declarations are somewhat specific interface declarations.

(C) 2010-2015 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

Annotation rules

- An annotation decl takes the form of interface decl except that …
  - they start with an 'at' sign @;
  - their method declarations must not have any parameters;
  - … and must not have any throws clauses;
  - their return types of the method must be one of the following:
    - Primitive types;
    - String;
    - Class;
    - Enum;
    - Array types of the above types;
  - there may be a default for each method.

(C) 2010-2015 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

Meta-annotations aka Metametadata

- @Documented – controls whether or not the declarations should be visible when targets are documented in any way.
- @Inherited – controls whether or not a subclass (or a member thereof) inherits the annotation when the superclass (or a member thereof) was target.
- @Retention – (runtime, source, …) controls whether the annotation will be available, accessible during runtime.
- @Target – (field, method, type, …, annotation) the kind of declaration to which the annotation can be attached.

(C) 2010-2015 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

See package **junitlight** of
https://github.com/101companies/101repo/tree/master/technologies/Java_platform/samples/javaReflectionSamples

A tiny approximation of JUnit

DEMO

Scenario

- Take a class name as argument.
- Find all “test” methods in the class.
- Execute them.
- Keep a record of success and failure.

(C) 2010-2014 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

More references.
FYI

Extra meta-programming power for Java

- BCEL, ASM, ... – Byte-code engineering libraries
- Javassist – Byte-code editing also at source level
- com.sun.source.tree – Compiler Tree API
- … various other technologies

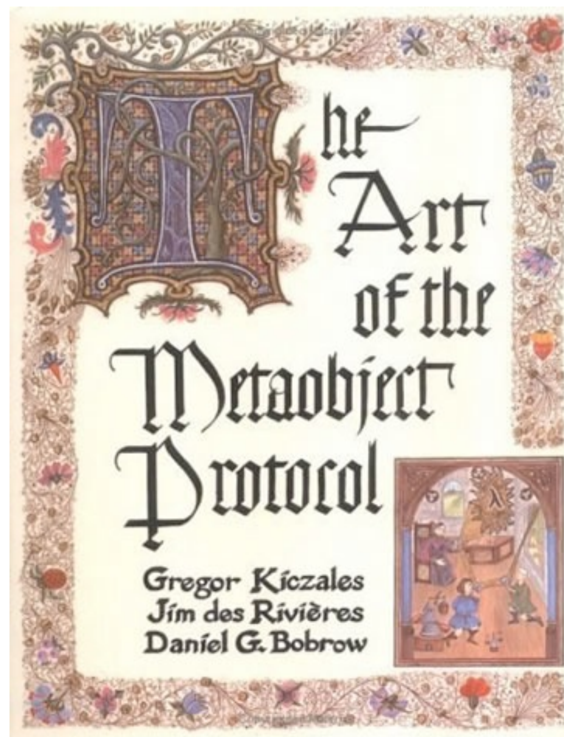
Related topics: code generation and analysis

- “Code-Generation Techniques for Java”, article at OnJava.com
- “Generate Java Code”, article at codefutures.com
- “Source Code Analysis Using Java 6 APIs", article Java.net

References - FYI

(C) 2010-2015 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

Further reading: Beyond Java



July 1991
7 x 9, 348 pp., 9 illus.
$45.00/£29.95 (PAPER)
Short

ISBN-10:
0-262-61074-4
ISBN-13:
978-0-262-61074-2

References - FYI

(C) 2010-2015 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

Seminal references

- Brian C. Smith. Reflection and semantics in lisp. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 23--35. ACM Press, January 1984.
- Daniel P. Friedman and Mitchell Wand. Reification: Reflection without metaphysics. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 348--355, Austin, Texas, August 1984. ACM Press.
- Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. *Lisp and Symbolic Computation*, 1(1):11--38, June 1988.
- Stanley Jefferson and Daniel P. Friedman. A simple reflective interpreter. *Lisp and Symbolic Computation*, 9(2/3):181--202, May/June 1996.

References - FYI

(C) 2010-2015 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

Online resources on Java reflection

References - FYI

Series at IBM developerWorks

- Part 1: Classes and class loading
- Part 2: Introducing reflection
- Part 3: Applied reflection
- Part 4: Class transformation with Javassist
- Part 5: Transforming classes on-the-fly
- Part 6: Aspect-oriented changes with Javassist
- Part 7: Bytecode engineering with BCEL
- Part 8: Replacing reflection with code generation

Miscellaneous

- Another simple reflection tutorial
- "Reflection API Code Samples", samples @ SDN
- More on class loading:
  - "Inside Class Loaders", article at OnJava.com
  - "Create a custom Java 1.2-style ClassLoader", article at JavaWorld.com

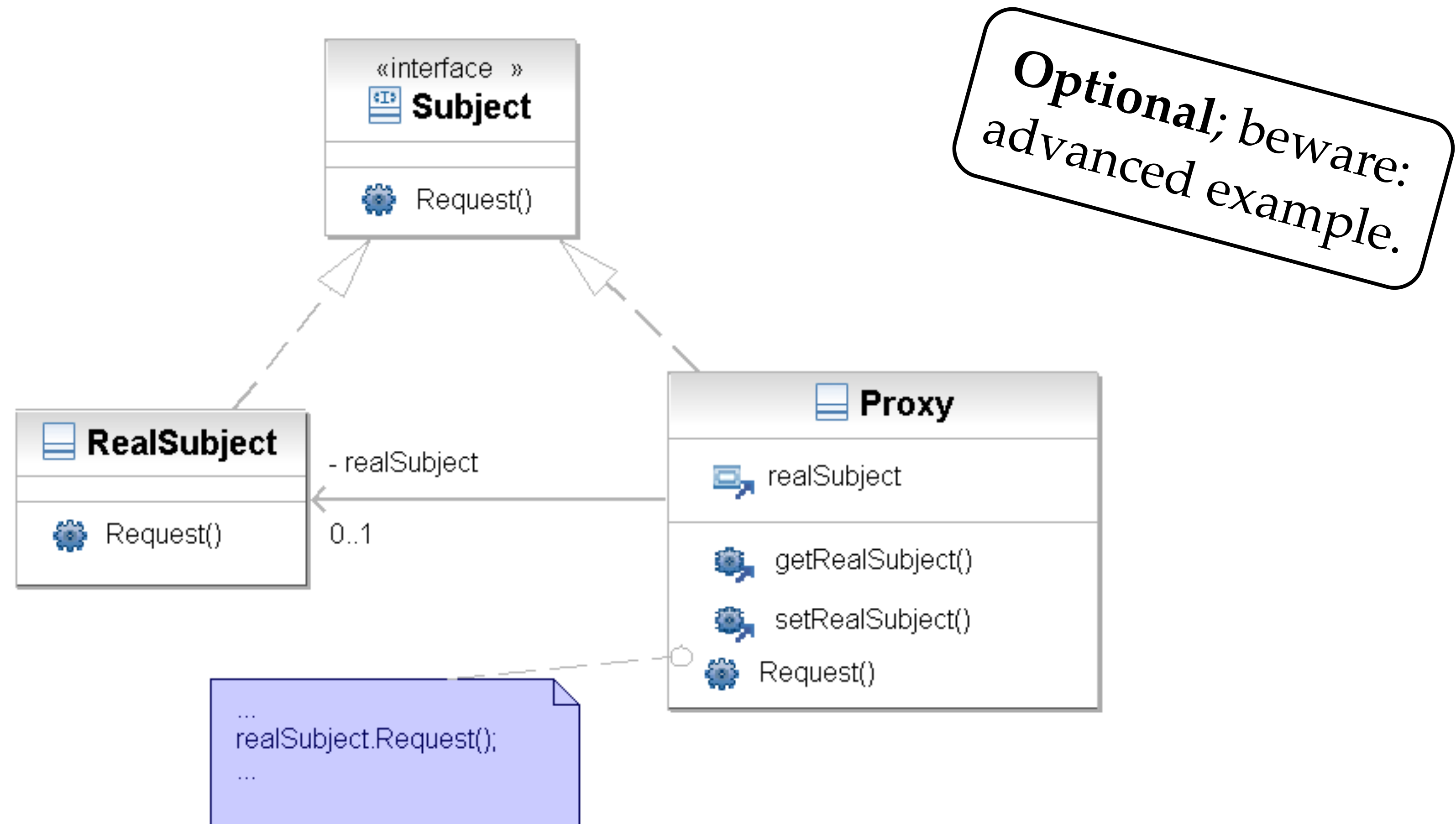
(C) 2010-2015 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

Optional; beware: advanced example.

Reflective proxies

Reflection-based proxies

Remember the Proxy Pattern?



(C) 2010-2014 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

See package **tracer** of
https://github.com/101companies/101repo/tree/master/technologies/Java_platform/samples/javaReflectionSamples

A tracer proxy for any given object

DEMO

Optional; beware: advanced example.

Scenario
- Taken an object of any type.
- Return a proxy for that object.
- Forward all methods from proxy to real subject.
- In addition, produce tracing information.

(C) 2010-2014 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

java.lang.reflect.Proxy

Optional; beware: advanced example.

- Proxy provides static methods for creating *dynamic proxy classes and instances*, and it is also the superclass of all dynamic proxy classes created by those methods.
- A (*dynamic) proxy class* is a class that implements a list of interfaces specified at runtime when the class is created.
- A *proxy interface* is such an interface that is implemented by a proxy class.
- A *proxy instance* is an instance of a proxy class. Each proxy instance has an associated *invocation handler* object, which implements the interface InvocationHandler. A method invocation on a proxy instance through one of its proxy interfaces will be dispatched to the invoke method of the instance's invocation handler.

(C) 2010-2015 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

java.lang.reflect.Proxy

If *XYZ* is an interface passed as an argument in the creation of the proxy class, and *obj* is an instance of the proxy class, then the following operations are valid:

- *obj* instanceof *XYZ*
- (*XYZ*) *obj*

Optional; beware: advanced example.

(C) 2010-2015 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)

Summary

- Meta-programming comes in many forms:
  - Analyze
  - Transform
  - Generate
  - Compose

  Code, where code is …
  - Java source code, or
  - JVM byte code.
- Meta-programming is used/needed all over the place:
  - By the compiler
  - By IDE and tool support (JUnit, …)
  - By application generators
  - By mapping tools (X/O/R)

(C) 2010-2015 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)