Metaprogramming

Matthieu Moy

UCBL

2020-2021

Before we start: Unit tests with JUnit

- Example JUnit test class:

```
class TestPlainJUnit {
    @Test
    void test() {
        assertEquals(4, 2 + 2);
    }

    @Test
    void testExcept() {
        assertThrows(MyException.class, () -> {
            throw new MyException(); // test fails if removed
        });
    }
}
```

JUnit

- Tool to write unit test, i.e. code that tests code
- What JUnit does:
  - Lists all methods with annotation `@Test`
  - Run them
  - Report when assertions (`assertEquals(..., ...)`, ...) fail
- Right now: we'll re-implement a mini JUnit
- In your project (last lab): use JUnit (the real one) in your project

Metaprogramming: programming a program

- Programming = data manipulations
- Meta-programming = consider program as data
- Why?
  - Automatic documentation: read code, write doc
  - Generic programming
    - Java Persistence API (write Java, let it do the SQL) `http://www.vogella.com/tutorials/JavaPersistenceAPI/article.html`
    - XML serialization (annotate Java classes, get XML serialization for free), e.g. Java Architecture for XML Binding (JAXB).
    - ...
  - Static checks (turn runtime errors into compile-time errors)
  - Have fun :-)

Motivating Example: Unit Tests

- A typical unit test framework does (pseudo-code):

```
for (m : thingsToTest) {
    backend.notifyThatTestIsRunning(); // System.out, or GUI
    try {
        m.run(); // Run the test
        backend.notifyThatTestPasses();
    } catch {
        backend.notifyThatTestFails();
    }
}
```

- Types of `m` and `thingsToTest`?
  - `m`: a method, "something that can be ran" ⇝ `java.lang.Runnable` or `java.lang.reflect.Method`.
  - `thingsToTest`: a set of runnables (e.g. `List<Runnable>`)

Home Made Unit Test Framework

- In real life: use JUnit
- This course: write our own framework (Homemade-JUnit), several versions:
  1. Ask the user to list methods to test
  2. Reflexion: list methods in a class, run those starting with `test`
  3. Annotation (= JUnit 4 and 5's solution): user annotates test methods with `@Test`
- Available in the course's repo, `homemade-junit/`.

Homemade-JUnit v0: No Framework

- How to use it:

```
class ClassToTest {
    void testMethod1() { ... }
    void testMethod2() { ... }
}

ClassToTest tc = new ClassToTest();

tc.testMethod1();
tc.testMethod2();
```

- Limitations:
  - User has to call methods explicitly
  - Any code to execute for each method has to be replicated

Homemade-JUnit v0: No Framework

- Tentative extension:

```
ClassToTest tc = new ClassToTest();

int failures = 0;
try {
    tc.testMethod1();
} catch (AssertionError e) {
    failures++;
}
try {
    tc.testMethod2();
} catch (AssertionError e) {
    failures++;
}
System.out.println(failures + " failures");
```

- Ouch, ugly cut-and-paste :-(

Homemade-JUnit v1: Explicit List of Methods

- How to use it:

```
ClassToTest tc = new ClassToTest();

TestRunnerExplicitList runner =
    new TestRunnerExplicitList(tc);
runner.addTestMethod(tc::testMethod1);
runner.addTestMethod(tc::testMethod2);

runner.run();
```

Homemade-JUnit v1: Explicit List of Methods

- How it is implemented (1/2):

```
class TestRunnerExplicitList {
    Object objectUnderTest;
    List<Runnable> methodsToTest = new ArrayList<Runnable>();

    public TestRunnerExplicitList(Object tc) {
        objectUnderTest = tc;
    }

    public void addTestMethod(Runnable m) {
        methodsToTest.add(m);
    }

    public void run() { ... } }
```

Homemade-JUnit v1: Explicit List of Methods

- How it is implemented (2/2, missing exception treatment):

```
public class TestRunnerExplicitList {
    List<Runnable> methodsToTest;
    ...
    public void run() {




        for (Runnable m : methodsToTest) {

            m.run();
        }


    }
}
```

Homemade-JUnit v1: Explicit List of Methods

- How it is implemented (2/2, missing exception treatment):

```
public class TestRunnerExplicitList {
    List<Runnable> methodsToTest;
    ...
    public void run() {
        String name =
                objectUnderTest.getClass().getName();
        System.out.println(
                "Testing class " + name + "...");
        for (Runnable m : methodsToTest) {
            System.out.println("  testing one method");
            m.run();
        }
        System.out.println(
                "Testing class " + name + ": DONE");
    }
}
```

Homemade-JUnit v1: Explicit List of Methods

- Pros:
  - Generic code written once, executed once for each test method
  - ‘System.out‘ could be replaced by IDE integration easily
- Cons:
  - User still has to specify list of methods
  - It’s easy to forget one ‘addTestMethod‘ ...
- Next: get the list automatically

Behind `Runnable` Type: Functional Interfaces in Java

- Functional Interface = interface for classes that represent functions = interface containing only one method (optionally annotated with `@FunctionalInterface`)
- Example:

```
@FunctionalInterface
interface IntToInt {
    abstract int run(int x);
}
class C {
    static int increment(int x) { return x + 1; }
}

// Lambda function assigned to functional interface
IntToInt fi = x -> x + 1;
// Reference to method assigned to functional interface
fi = C::increment;
```

Method References: How to Use Them?

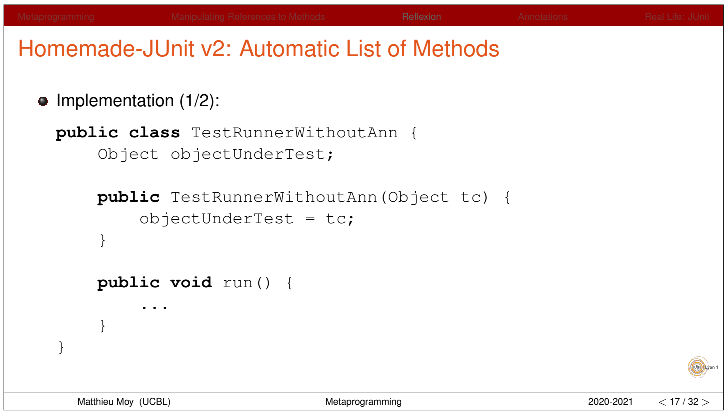
```
ClassToTest tc = new ClassToTest();

// Reference to an instance method of a particular object
Runnable m1 = tc::testMethod1;
m1.run();  // tc.testMethod1();

// Reference to an instance method of an
// arbitrary object of a particular type
Consumer<ClassToTest> m2 = ClassToTest::testMethod2;
m2.accept(tc);  // tc.testMethod2();

BiConsumer<ClassToTest, Integer> m3 = ClassToTest::testMethodWithArg;
m3.accept(tc, 42);  // tc.testMethodWithArg(42)
```

https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html

Homemade-JUnit v2: Automatic List of Methods

- How to use it:

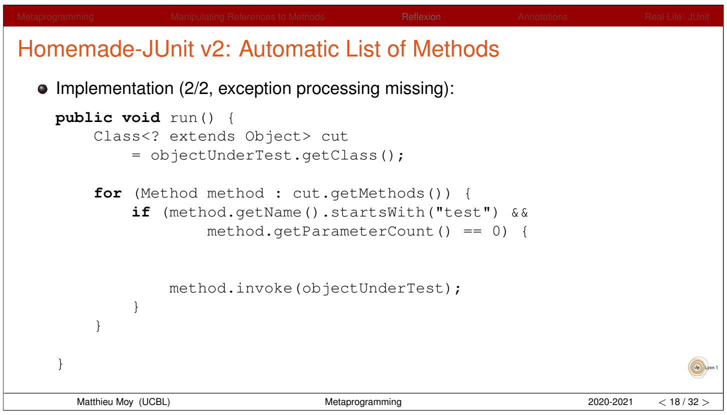
```
ClassToTest tc = new ClassToTest();

TestRunnerWithoutAnn runner = new TestRunnerWithoutAnn(tc);

// Run all methods in ClassToTest
// with name starting with "test"
runner.run();
```



Homemade-JUnit v2: Automatic List of Methods

- Implementation (1/2):

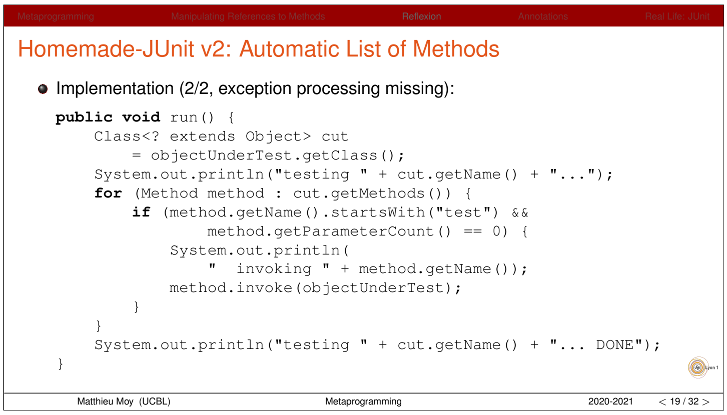
```
public class TestRunnerWithoutAnn {
    Object objectUnderTest;

    public TestRunnerWithoutAnn(Object tc) {
        objectUnderTest = tc;
    }

    public void run() {
        ...
    }
}
```



Homemade-JUnit v2: Automatic List of Methods

- Implementation (2/2, exception processing missing):

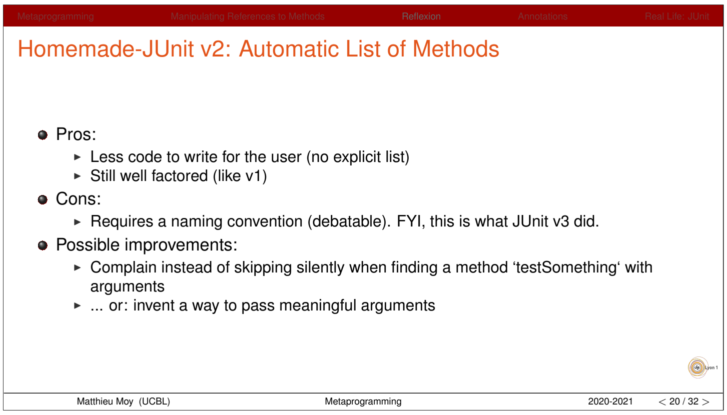
```
public void run() {
    Class<? extends Object> cut
        = objectUnderTest.getClass();

    for (Method method : cut.getMethods()) {
        if (method.getName().startsWith("test") &&
                method.getParameterCount() == 0) {


            method.invoke(objectUnderTest);
        }
    }

}
```



Homemade-JUnit v2: Automatic List of Methods

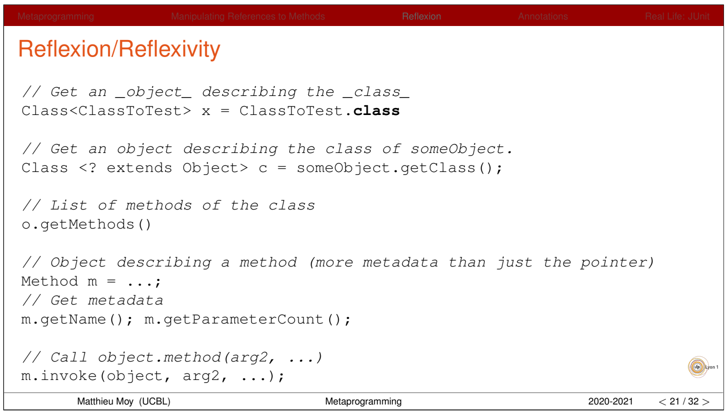
- Implementation (2/2, exception processing missing):

```
public void run() {
    Class<? extends Object> cut
        = objectUnderTest.getClass();
    System.out.println("testing " + cut.getName() + "...");
    for (Method method : cut.getMethods()) {
        if (method.getName().startsWith("test") &&
                method.getParameterCount() == 0) {
            System.out.println(
                "  invoking " + method.getName());
            method.invoke(objectUnderTest);
        }
    }
    System.out.println("testing " + cut.getName() + "... DONE");
}
```



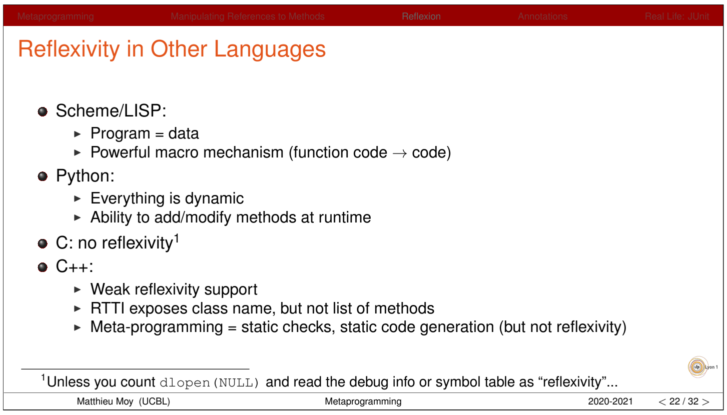
Homemade-JUnit v2: Automatic List of Methods

- Pros:
  - Less code to write for the user (no explicit list)
  - Still well factored (like v1)
- Cons:
  - Requires a naming convention (debatable). FYI, this is what JUnit v3 did.
- Possible improvements:
  - Complain instead of skipping silently when finding a method 'testSomething' with arguments
  - ... or: invent a way to pass meaningful arguments



Reflexion/Reflexivity

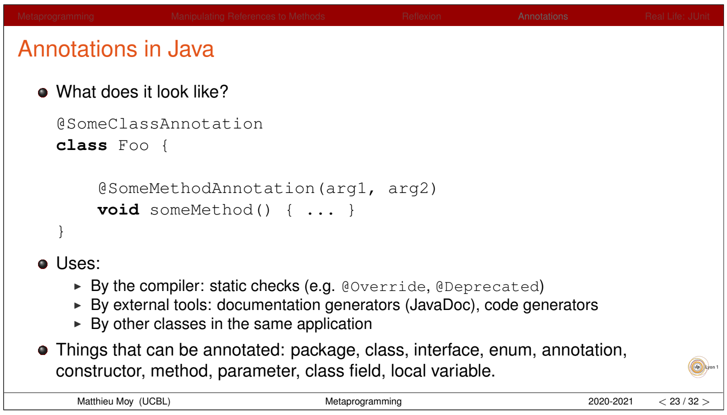
```
// Get an _object_ describing the _class_
Class<ClassToTest> x = ClassToTest.class

// Get an object describing the class of someObject.
Class <? extends Object> c = someObject.getClass();

// List of methods of the class
o.getMethods()

// Object describing a method (more metadata than just the pointer)
Method m = ...;
// Get metadata
m.getName(); m.getParameterCount();

// Call object.method(arg2, ...)
m.invoke(object, arg2, ...);
```



Reflexivity in Other Languages

- Scheme/LISP:
  - Program = data
  - Powerful macro mechanism (function code → code)
- Python:
  - Everything is dynamic
  - Ability to add/modify methods at runtime
- C: no reflexivity[1]
- C++:
  - Weak reflexivity support
  - RTTI exposes class name, but not list of methods
  - Meta-programming = static checks, static code generation (but not reflexivity)

[1] Unless you count `dlopen(NULL)` and read the debug info or symbol table as "reflexivity"...

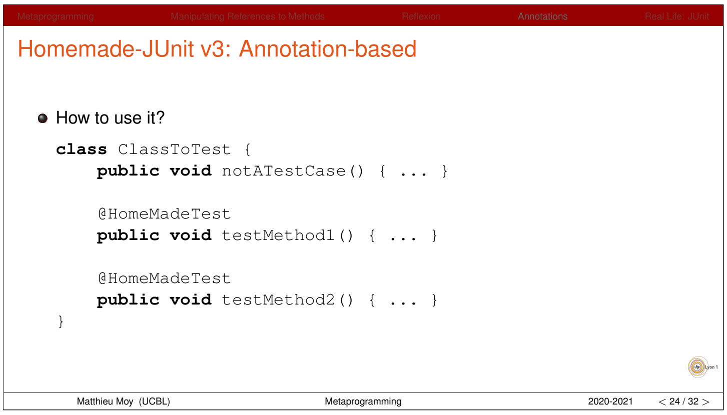


Annotations in Java

- What does it look like?

```
@SomeClassAnnotation
class Foo {

    @SomeMethodAnnotation(arg1, arg2)
    void someMethod() { ... }
}
```

- Uses:
  - By the compiler: static checks (e.g. `@Override`, `@Deprecated`)
  - By external tools: documentation generators (JavaDoc), code generators
  - By other classes in the same application
- Things that can be annotated: package, class, interface, enum, annotation, constructor, method, parameter, class field, local variable.



Homemade-JUnit v3: Annotation-based

- How to use it?

```
class ClassToTest {
    public void notATestCase() { ... }

    @HomeMadeTest
    public void testMethod1() { ... }

    @HomeMadeTest
    public void testMethod2() { ... }
}
```

Homemade-JUnit v3: Annotation-based

- Implementation: declare annotation

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface HomeMadeTest {
    // Nothing!
}
```

- An object of type `HomeMadeTest` attached to each method decorated with `@HomeMadeTest`
- Don't forget `Retention(RetentionPolicy.RUNTIME)`: default is `CLASS` which keeps the annotations in `.class` files, but doesn't load them at runtime.

Homemade-JUnit v3: Annotation-based

```
public class TestRunnerWithAnn {
    Object objectUnderTest;
    public TestRunnerWithAnn(Object tc) { objectUnderTest = tc; }

    public void run() {
        Class<? extends Object> cut
                = objectUnderTest.getClass();
        for (Method method : cut.getMethods()) {
            processMethod(method);
        }
    }

    void processMethod(Method method) { ... }
}
```

Homemade-JUnit v3: Annotation-based

```
private void processMethod(Method method) {
    HomeMadeTest a = method.getAnnotation(HomeMadeTest.class);
    if (a != null) {
        method.invoke(objectUnderTest);
    }
}
```

Homemade-JUnit v3.1: Parameterized Tests

- Sometimes, one wants to run the same test with multiple inputs
- Non-meta-programming way:

```
tc.testMethodWithArg(1);
tc.testMethodWithArg(2);
tc.testMethodWithArg(33);
```

- Our annotation-based way:

```
@HomeMadeTest
@HomeMadeArgs({1, 2, 33})
public void testMethodWithArg(int x) {
    ...
}
```

Homemade-JUnit v3.1: Parameterized Tests

- Annotation declaration:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface HomeMadeArgs {
    int[] value();
}
```

Homemade-JUnit v3.1: Parameterized Tests

- Implementation:

```
private void processMethod(Method method) {
    HomeMadeTest a = method.getAnnotation(HomeMadeTest.class);
    if (a != null) {
        HomeMadeArgs args = method.getAnnotation(HomeMadeArgs.class);
        if (args != null) {
            for (int arg : args.value()) {
                method.invoke(objectUnderTest, arg);
            }
        } else {
            method.invoke(objectUnderTest);
        }
    }
}
```

JUnit and Annotations

- Example JUnit test class:

```
class TestPlainJUnit {
    @Test
    void test() {
        assertEquals(4, 2 + 2);
    }

    @Test
    void testExcept() {
        assertThrows(MyException.class, () -> {
            throw new MyException(); // test fails if removed
        });
    }
}
```

JUnit and Annotations

```
public class FibonacciTest {
    @ParameterizedTest
    @ValueSource(ints = { 1, 3, 5, 15, Integer.MAX_VALUE }) // six numbers
    void isOdd_ShouldReturnTrueForOddNumbers(int number) {
        assertEquals(1, number % 2);
    }

    @ParameterizedTest
    @CsvSource({"1,1", "2,2", "3,3", "4,5", "5,8"})
    void testFibo(String n, String expected_fibo_n) {
        assertEquals(Integer.parseInt(expected_fibo_n),
                     Fibo.fibo(Integer.parseInt(n)));
    }
}
```

https://www.baeldung.com/parameterized-tests-junit-5