Quality Assurance Reflection

Part II - Lecture 9

What do they have in common?

1 *Magic mirror on the wall. . .*

. . . who is the fairest of them all?

Queen, you are full fair, it is true,

but Snow White is fairer than you.

2

SimSim
SimSim
The Simulator Simulator

3

popculturegeek.com

Today's Outline

- Reflection
- The Java Reflection API
- MetaJ

Reflection

By three methods we may learn wisdom:
First, by reflection, which is noblest;
Second, by imitation, which is easiest;
and third by experience, which is the bitterest.
(Confucius)

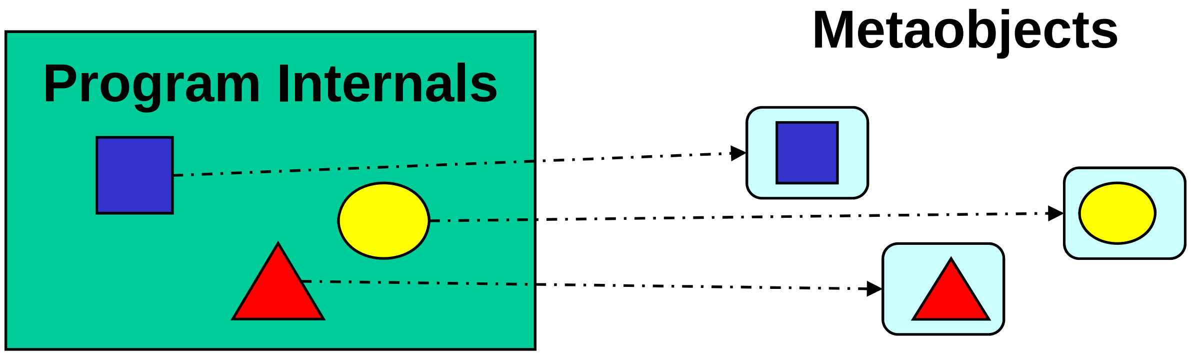
Reflection

The ability of a program to **observe** and possibly **modify** its own **structure** and **behavior**.

- Two kinds of reflection
  - Structural: reflection on data structures & code
  - Behavioral: reflection on program behavior
- Two basic operations
  - Introspection: observe program
  - Intercession: modify it
- Can be **static** (before runtime) or **dynamic** (during runtime)
- Can be dangerous …

Metaobject Protocols (MOPs)

- The way reflection is done in OO languages
- Internal program entities (e.g. types) are represented as metaobjects, which are instances of metaclasses
- Metaobjects are like normal objects, but they serve a special purpose
- The way we handle these metaobjects, i.e. the way methods have to be called in order to do a reflection task, is the **metaobject protocol**
- **Introspecting** the system means getting metaobjects
- **Intercession** means that we can modify them and make those modifications affect the system (sometimes this is done automatically)

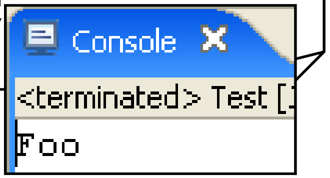


Introspection of Data Structures

- Data structures are usually a static concept
  - They are **defined before compilation**
  - They stay **unchanged during runtime**
- Sometimes we want a program to be able to work with unknown data structures
  - It might get an object, not knowing the exact class
  - A system might allow dynamic loading of classes
- Solution: use introspection
  - Get metaobject for the class of the unknown object
  - Metaobject gives us a description of the unknown object's class

Class
String getName()
Field[] getFields()
Method[] getMethods()
Constructor[] getConstructors()

```
Object f = new Foo();
Class c = f.getClass();
System.out.println(
   c.getName());
```



```
<terminated> Test [
Foo
```

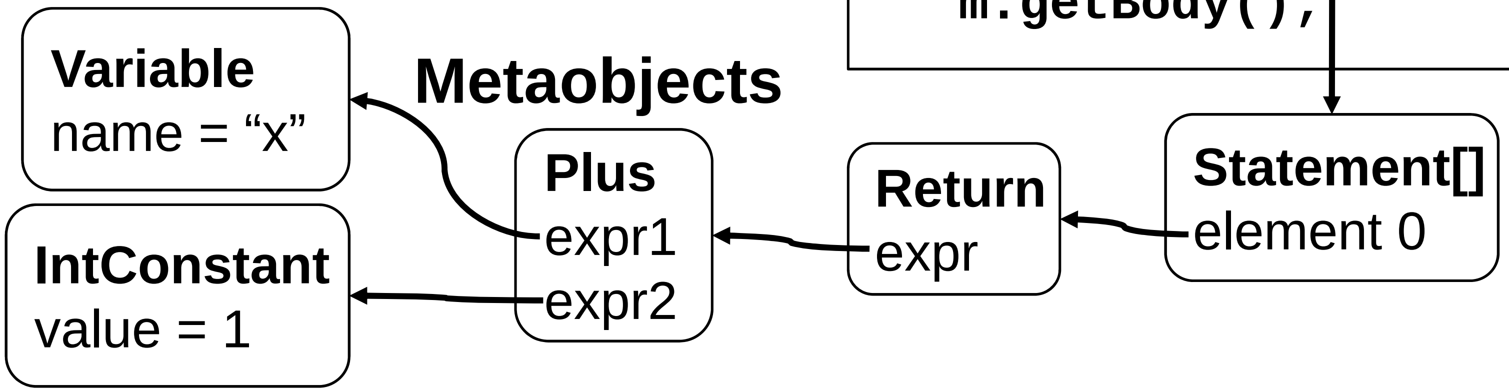
Introspection of Code

- Only supported in few languages
- Possibility to look into method bodies and see all the statements
- Program code is usually represented ("reified") as abstract syntax tree (AST)
- Metaclasses for different statements, expressions, …

```
class Foo {
   int inc(int x) {
      return x + 1;
}  }
```

Pseudo-code

```
Object f = new Foo();
Method m = f.getClass()
   .getMethod("inc");
Statement[] stmts =
   m.getBody();
```



Intercession

- Data structures / program code can be modified during runtime
- Can be done, for example, simply through modification of metaobjects
- Rare feature because it can be dangerous (confusing and unsafe)
- Invariants like types and program code usually important for us to understand complex systems

Pseudo-code

```
Object f = new Foo();
Method m = f.getClass()
   .getMethod("inc");
Statement[] stmts = m.getBody();
Return rstmt = (Return) stmts[0];
Plus pexpr = (Plus) rstmt.expr;
pexpr.expr2 = new IntConstant(99);
```

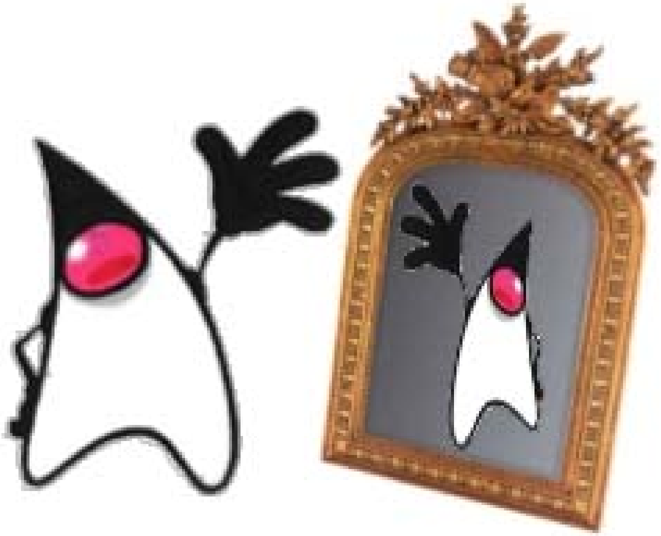
Before

```
class Foo {
   int inc(int x) {
      return x+1;
}  }
```

After

```
class Foo {
   int inc(int x) {
      return x+99;
}  }
```

The Java Reflection API



Reflection in Java

- Java does not support full dynamic reflection
- Only some introspection
  - Introspection of **types**
  - Introspection of **method signatures**
  - Introspective **access** to types and methods
    - Instantiation
    - Field access (read and write)
    - Method invocation
- For safety: exceptions are thrown when something doesn’t work (e.g. **`NoSuchFieldException`**, **`NoSuchMethodException`**, **`SecurityException`**)

Java Reflection Example: Introspection

```
import java.lang.reflect.*;

public class Test {
   public static void main(String[] args) {
      Object o = new Integer(1);
      Class c = o.getClass();
      System.out.println(c.getName()); // java.lang.Integer
      System.out.println(
         c.getSuperclass().getName()); // java.lang.Number
      System.out.println(
         c.getPackage().getName());    // java.lang
      for(Field f : c.getFields())
         System.out.println(f);    // … int … MIN_VALUE , …
      for(Method m : c.getMethods())
         System.out.println(m);    // … int … hashCode() , …
      for(Constructor ct : c.getConstructors())
         System.out.println(ct);   // … Integer(int) , …
}  }
```

Java Reflection Example: Introspective Access I

```
public class Foo {
   public void hello() {
      System.out.println("hello!");
}  }
```

```
import java.lang.reflect.*;

public class Test {
   public static void main(String[] args) {
      try {
         Class c = Class.forName("Foo");
         Method m = c.getMethod("hello", null);
         m.invoke(c.newInstance(), null);
      } catch(Exception e) {
         e.printStackTrace();
      }
}  }
```

Java Reflection Example: Introspective Access II

```
public class Foo { public int x; }
```

```
import java.lang.reflect.*;

public class Test {
   public static void main(String[] args) {
      try {
         Class c = Class.forName("Foo");
         Field fieldx = c.getField("x");
         Object foo = c.newInstance();
         fieldx.set(foo, 99);
         System.out.println(fieldx.get(foo));
      } catch(Exception e) {
         e.printStackTrace();
      }
}  }
```

Class Class<T>

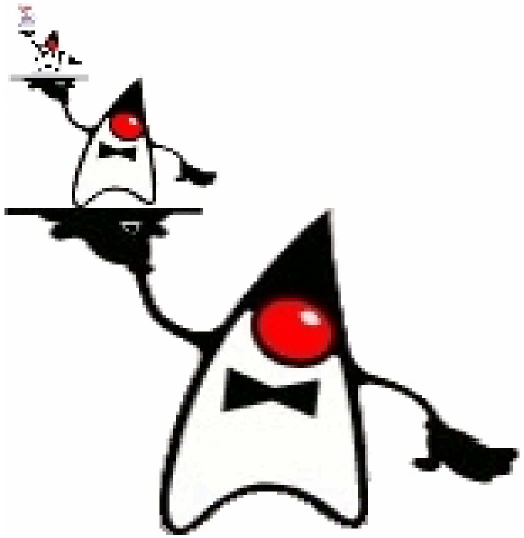
- static Class<?> forName(String className)
- String getName()
- String getSimpleName()
- Class<? super T> getSuperclass()
- Field[] getDeclaredFields()
- Field[] getFields()
- Field getDeclaredField(String name)
- Field getField(String name)
- Constructor<T> getConstructor(Class... paramTypes)
- Method getMethod(String name, Class... paramTypes)
- T newInstance()
- boolean isArray()
- boolean isInterface()
- boolean isPrimitive()

Class Field and Class Method

- Class Field
  - `String getName()`
  - `Class<?> getType()`
  - `Object get(Object obj)`
  - `int getInt(Object obj)`
  - `boolean getBoolean(Object obj)`
  - `void set(Object obj, Object value)`

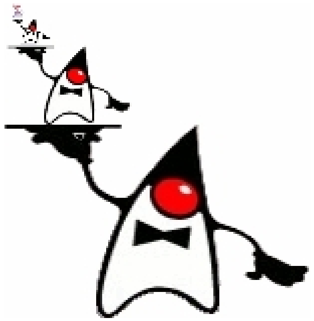
- Class Method
  - `String getName()`
  - `Class<?>[] getParameterTypes()`
  - `Class<?> getReturnType()`
  - `Object invoke(Object obj, Object... args)`

MetaJ



MetaJ

- **Reflective interpreter** for Java-like language
- Research prototype written by Rémi Douence and Mario Südholt http://www.emn.fr/x-info/sudholt/research/metaj/
- Offers **complete dynamic structural & behavioral reflection**: you can change classes, code, and even the interpreter itself
- In other words: nearly everything can be changed by a program
- However: if you change the wrong thing you crash
- Common challenge of reflective languages: balancing act between **power** and **safety**



Example: Dynamic Reflection with MetaJ

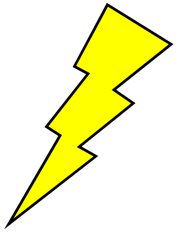
```
class Pair { String fst; String snd; }

class PrintablePair extends Pair {
   String toString() {
       return "(" + fst + ", " + snd + ")";
}  }

class Main {
   void main() {
     Pair p = new Pair("1", "2");
     Class metaClass = reify(Pair);   // Introspection
     if (metaClass.getExtendsLink() == null)
        System.out.println("Pair has no superclass!!!");
     Instance metaInstance = reify(p); // Intercession
     metaInstance.instanceLink = PrintablePair;
     System.out.println(p.toString());
}  }
```

MetaJ's Metaclasses

```
class Class {
  Class extendsLink;    // superclass
  DataList dataList;    // field list
  MethodList methodList;
  Instance instantiate() { ... } // "new" operator
}

class Method {
  private StringList args; // parameter names
  private Exp body;        // method body

  Data apply(Environment argsE, Instance i) {
      ...
      return this.body.eval(argsE);
  }
}
```

Safety Issues

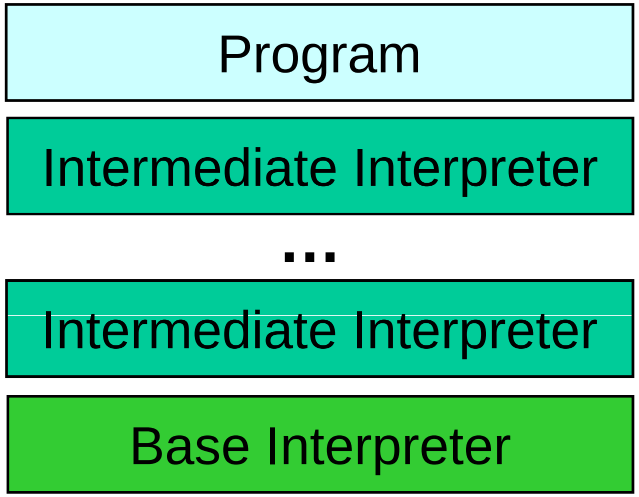
```
class Pair {
   String fst; String snd;
   String toString() {
      return "(" + fst + ", " + snd + ")";
}  }

class NotAPair {
  int fst;
}

class Main {
   void main() {
      Pair p = new Pair("1", "2");
      Instance metaPair = reify(p);
      metaPair.instanceLink = NotAPair;
      p.fst = 99;
      System.out.println(p.toString());
}  }
```



Behavioral Reflection

SE Software Engineering
The University of Auckland

- Not only the running program can be modified (structural reflection), but also the runtime system (behavioral reflection)
- In MetaJ: the interpreter itself can be changed
- "**Reflective Towers**" meta-architecture
  - Unchangeable base interpreter which interprets the program
  - Possibility to insert a new intermediate interpreter between the base interpreter and the program
  - Intermediate interpreter can be arbitrarily modified with reflection (like program)
  - Can change the **operational semantics** of the language
  - Base interpreter interprets the intermediate interpreter on top of it which in turn interprets the program
  - We can insert as many intermediate interpreters as we like (tower of interpreters)



Today's Summary

- **Reflection** is the ability of a program to observe and possibly modify its own structure and behavior
  - **Introspection** and **intercession**
  - OO languages use **metaobject protocols** (MOPs) with metaclasses and metaobjects
- **Java** only supports some introspection and introspective access to methods and fields
- Other languages (e.g. **MetaJ**) offer full reflection
- However: reflection can be **dangerous** (lead to hard-to-find bugs)

Quiz

1. What are the two basic operations of reflection? What do they do?

2. What kind of reflection is Java capable of?

3. Why can reflection be dangerous?