Embedded Typesafe Domain Specific Languages for Java

Rein Raudjärv
University of Tartu

Theory Days at Jõulumaa

October 5th 2008

Based on

- Research article “**Embedded Typesafe Domain Specific Languages for Java**”
  - by **Jevgeni Kabanov** and **Rein Raudjärv**
  - Published at **Principles and Practice of Programming in Java '08**

Outline

- Introduction
- Case Study 1: SQL in Java
- Case Study 2: Bytecode Engineering
- Conclusions

Domain Specific Language

- Small ***sub-language*** that has very little overhead when expressing domain specific data and behaviour

- Can be
  - A fully implemented language
  - A specialised API that looks like a sublanguage but is still written using some general-purpose language – ***embedded DSL***

Motivation

- Low overhead
  - You need to write less
  - It is easier to understand
  - Domain experts can understand it
- Type safety
  - You can be sure that certain errors won’t occur
  - You need to write less tests
  - You can use type info to add features

Fluent Interface

```
customer.newOrder()
   .with(6, "TAL")
   .with(5, "HPK").skippable()
   .with(3, "LGV")
   .priorityRush();
```

Java 5 Features and EDSLs

- Java 5 has
  - Parametric polymorphism (**generics**)
  - Static method import

- Java 5 doesn't have
  - Closures or first-class functions
  - Operator overloading
  - Local type inference

SQL in Java

Case Study 1

SQL Example (6 errors)

```
StringBuffer sql = new StringBuffer();
sql.append("SELECT o.sum,(SELECT
first_name,last_name");
sql.append("          FROM person p");
sql.append("         WHERE o.pesron_id=p.id) AS
client");
sql.append(" FORM order o");
sql.append("WHERE o.id = "+orderId);
sql.append("  AND o.status_id IN (?,?)");
PreparedStatement stmt =
conn.prepareStatement(sql.toString());
stmt.setString(1, "PAID");
...
```

SQL Example (6 errors)

```
StringBuffer sql = new StringBuffer();
sql.append("SELECT o.sum,(SELECT
first_name,last_name");
sql.append("          FROM person p");
sql.append("         WHERE o.pesron_id=p.id) AS
client");
sql.append(" FORM order o");
sql.append("WHERE o.id = "+orderId);
sql.append("  AND o.status_id IN (?,?)");
PreparedStatement stmt =
conn.prepareStatement(sql.toString());
stmt.setString(1, "PAID");
...
```

Typesafe SQL Example

```
Person p = new Person();
List<Tuple3<String, Integer, Date>> rows =
  new QueryBuilder(datasource)
    .from(p)
    .where(gt(p.height, 170))
    .select(p.name, p.height, p.birthday)
    .list();
for (Tuple3<String, Integer, Date> row : rows) {
  String name = row.v1;
  Integer height = row.v2;
  Date birthday = row.v3;
  System.out.println(
  name + " " + height + " " + birthday);
}
```

Tuples

```
Person p = new Person();
List<Tuple3<String, Integer, Date>> rows =
  new QueryBuilder(datasource)
    .from(p)
    .where(gt(p.height, 170))
    .select(p.name, p.height, p.birthday)
    .list();
for (Tuple3<String, Integer, Date> row : rows) {
  String name = row.v1;
  Integer height = row.v2;
  Date birthday = row.v3;
  System.out.println(
  name + " " + height + " " + birthday);
}
```

Tuples (2)

- Return ***tuples*** that have precisely the selected data with types known ahead

- Tuple types are ***inferred*** from the select expression (column) types

Tuple2

```java
public class Tuple2<T1, T2> {
  public final T1 v1;
  public final T2 v2;

  public Tuple2(T1 v1, T2 v2) {
    this.v1 = v1;
    this.v2 = v2;
  }
}
```

Typesafe Metadata

```
Person p = new Person();
List<Tuple3<String, Integer, Date>> rows =
  new QueryBuilder(datasource)
    .from(p)
    .where(gt(p.height, 170))
    .select(p.name, p.height, p.birthday)
    .list();
for (Tuple3<String, Integer, Date> row : rows) {
  String name = row.v1;
  Integer height = row.v2;
  Date birthday = row.v3;
  System.out.println(
  name + " " + height + " " + birthday);
}
```

Typesafe Metadata (2)

- ***Metadata used by your DSL should include compile-time type information***
- We make use of pregenerated **metadata dictionary** that contains type information about tables and columns

Metadata Dictionary

```java
public class Person implements Table {
  public String getName() { return "person"; };

  public Column<Person, String> name =
    newColumn(this, "name", String.class);
  public Column<Person, Integer> height =
    newColumn(this, "height", Integer.class);
  public Column<Person, Date> birthday =
    newColumn(this, "birthday", Date.class);
}
```

Restricting Syntax

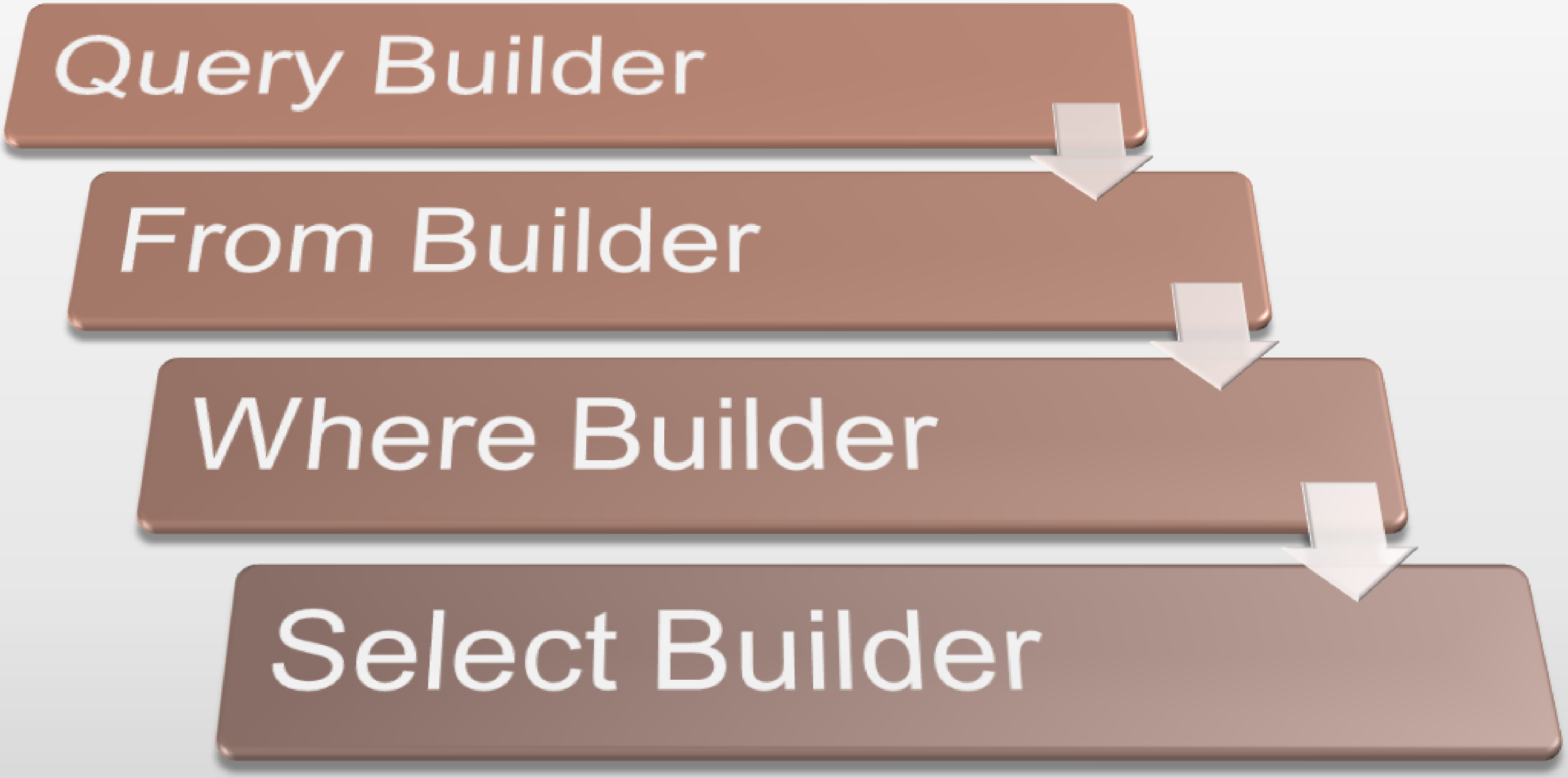
```
Person p = new Person();
List<Tuple3<String, Integer, Date>> rows =
  new QueryBuilder(datasource)
    .from(p)
    .where(gt(p.height, 170))
    .select(p.name, p.height, p.birthday)
    .list();
for (Tuple3<String, Integer, Date> row : rows) {
  String name = row.v1;
  Integer height = row.v2;
  Date birthday = row.v3;
  System.out.println(
  name + " " + height + " " + birthday);
}
```

Restricting Syntax (2)

- ***At any moment of time the DSL builder should have precisely the methods allowed in the current state***

- SQL query builders allow *from*, *where* and *select* to be called
  - once and only once
  - only in valid order

Builders



Query Builder

From Builder

Where Builder

Select Builder

QueryBuilder

```
class QueryBuilder extends Builder {
  ...
  <T extends Table> FromBuilder<T>
    from(T table);
}
```

FromBuilder

```
class FromBuilder<T extends Table>
  extends Builder {
  ...
  <C1> SelectBuilder1<T, C1>
    select(Col<T, C1> c1);

  <C1, C2> SelectBuilder2<T, C1, C2>
    select(Col<T, C1> c1, Col<T, C2> c2);
  ...
}
```

SelectBuilder

```
class SelectBuilder2<T extends Table,C1,C2>
  extends SelectBuilder<T> {
  ...
  List<Tuple2<C1,C2>> list();
  ...
}
```

Hierarchical Expressions

```
Person p = new Person();
List<Tuple3<String, Integer, Date>> rows =
  new QueryBuilder(datasource)
    .from(p)
    .where(gt(p.height, 170))
    .select(p.name, p.height, p.birthday)
    .list();
for (Tuple3<String, Integer, Date> row : rows) {
  String name = row.v1;
  Integer height = row.v2;
  Date birthday = row.v3;
  System.out.println(
  name + " " + height + " " + birthday);
}
```

Hierarchical Expressions (2)

- ***Use***
  - ***method chaining*** *when you need context*
  - ***static methods*** *when you need hierarchy and extensibility*

```
or(
    eq(p.name, "Peter"),
    gt(p.height, 170)
)
```

Expression

```
public interface Expr<E> {
  String getSql();
  List<Object> getArguments();
}
```

Expressions

```
class Expressions {
  Expr<Bool> and(Expr<Bool>... e)

  <E> Expr<Bool> eq(Expr<E> e1, Expr<E> e2)

  Expr<Bool>
      like(Expr<?> e, Expr<String> pattern)

  <E> Expr<E> constant(E value)

  Expr<String> concat(Expr<String>... e)
  ...
}
```

Unsafe Assumptions

- ***Allow the user to do type unsafe actions, but make sure he has to document his assumptions***

```
Expression<Integer> count =
  unchecked(Integer.class,
      "util.countChildren(id)");
```

Closers – Mixing with Control Flow

```
Person p = new Person();
List<Tuple2<Integer, String>> rows =
  new QueryBuilder(datasource)
    .from(p)
    .closure(new Closure() {
      public void apply(Builder builder) {
        if (searchName != null) {
          builder.addCondition(
            eq(p.name, searchName));
        }
      }
    })
    .select(p.id, p.name)
    .list();
```

Closures (2)

```
interface Closure { void apply(Builder b); }

class SelectBuilderC2<C1,C2>
  extends SelectBuilder {
    SelectBuilderC2<C1,C2>
      closure(Closure closure) {
        closure.apply(this);
        return this;
      }
    }
}
```

Used Patterns

- **Restricting syntax** (builders allow *from*, *where* and *select* to be called only once)
- **Typesafe metadata** (pregenerated metadata dictionary, SelectBuilder encode metadata about its column types)
- **Hierarchical expressions** (with method chaining for main syntax)

Used Patterns (2)

- **Unsafe assumptions** (unchecked expressions declare the expected type)
- **Closures** for mixing with the control flow

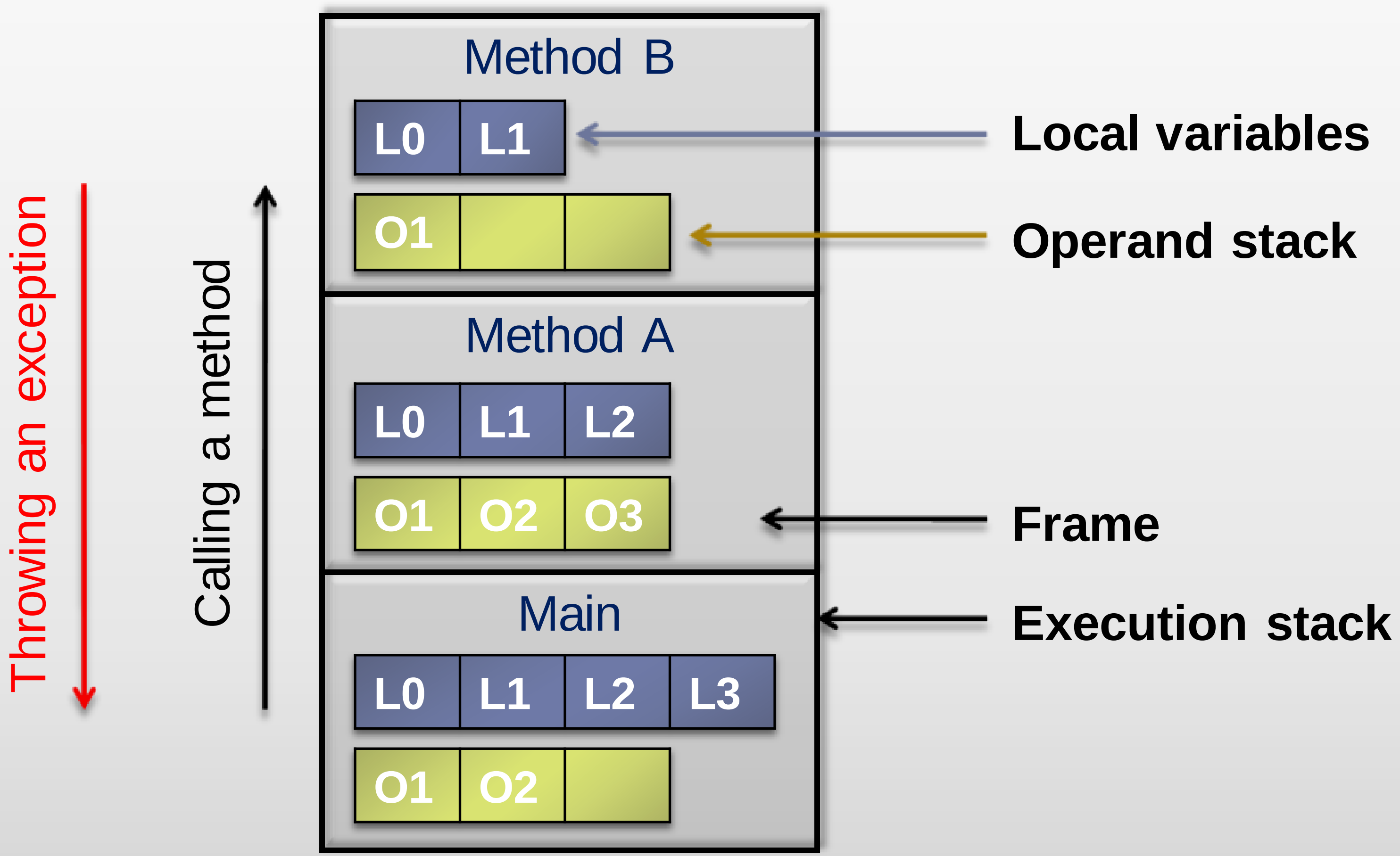
Engineering Java Bytecode

Case Study 2

Java Class Definition

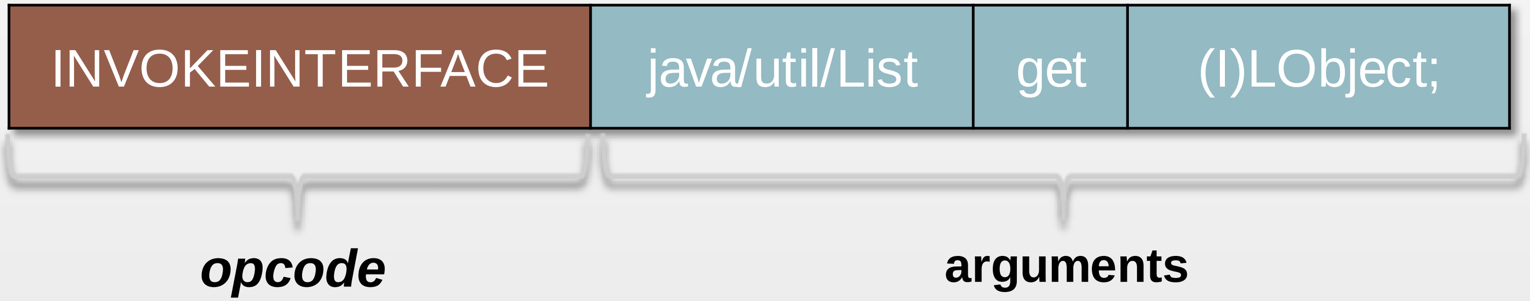
Modifiers, name, super class, interfaces	
Enclosing class reference	
Annotation*	
Inner class*	Name
Field*	Modifiers, name, type
	Annotation*
Method*	Modifiers, name, return and parameter types
	Annotation*
	Compiled code

Java Execution Model



Instruction Example

Instruction:

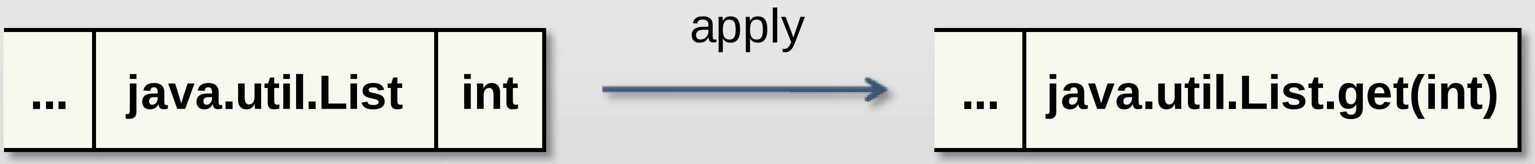


INVOKEINTERFACE | java/util/List | get | (I)LObject;

opcode

arguments

Operands stack when applying the instruction:



... | **java.util.List** | **int**

apply

... | **java.util.List.get(int)**

Bytecode Engineering

- **ASM** – Java bytecode engineering library
  - Visitor-based API
  - Tree-based API
- Completely untyped
- Produced bytecode is only verified at runtime

Hello, World!

```java
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello, World!");
  }
}
```

Hello, World! in Bytecode

```
public class HelloWorld {
    public <init>()V
        ALOAD 0
        INVOKESPECIAL Object.<init>()V
        RETURN
    public static main([LString;)V
        GETSTATIC System.out : LPrintStream;
        LDC "Hello, World!"
        INVOKEVIRTUAL PrintStream.println(LString;)V
        RETURN
}
```

Hello, World! in ASM

```
ClassWriter cw = new ClassWriter(0);
    MethodVisitor mv;
    cw.visit(V1_6, ACC_PUBLIC + ACC_SUPER, "HelloWorld", null,
"java/lang/Object", null);
    {
    mv = cw.visitMethod(ACC_PUBLIC + ACC_STATIC, "main",
"([Ljava/lang/String;)V", null, null);
    mv.visitCode();
    mv.visitFieldInsn(GETSTATIC, "java/lang/System", "out",
"Ljava/io/PrintStream;");
    mv.visitLdcInsn("Hello, World!");
    mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream",
"println", "(Ljava/lang/String;)V");
    mv.visitInsn(RETURN);
    mv.visitMaxs(2, 1);
    mv.visitEnd();
    }
    cw.visitEnd();
```

Hello, World! in DSL

```
new ClassBuilder(cw, V1_4, ACC_PUBLIC,
        "HelloWorld", "java/lang/Object", null)
    .beginStaticMethod(ACC_PUBLIC | ACC_STATIC,
       "main", void.class, String[].class)
    .getStatic(System.class, "out",
       PrintStream.class)
    .push("Hello, World!")
    .invoke()
      .param(String.class)
      .virtVoid(PrintStream.class, "println")
    .returnVoid()
    .endMethod();
```

Possible Mistakes

- Not enough stack elements for the instruction
- Stack elements have the wrong type
- Local variables have the wrong type
- Using illegal modifiers or opcodes

Similar Patterns

- **Typesafe metadata**
  - Class literals
  - Track types of the stack elements and local variables
- **Closures** for mixing with the control flow

Different Patterns

- **Restricting syntax**
  - Hide methods that consume more stack slots than available
- **Unsafe assumptions**
  - Deprecate methods instead of omitting them
  - Allow assuming stack slot and local variable types

Conclusions

- Not just method chaining!
  - Static methods
  - Typesafe metadata
  - Closures
- More safety
  - Generics
  - Type lists
  - Unsafe assumptions

Future Work

- SQL DSL developed as Open Source
  - **Squill**
  - http://squill.dev.java.net

Questions?