```
1   // Copyright 2013 Bill Campbell, Swami Iyer and Bahar Akbal-Delibas
2
3   package jminusminus;
4
5   /**
6    * A wrapper for members (eg Fields, Methods, Constructors) in the Java API.
7    * Members are used in message expressions, field selections, and new object
8    * construction operations.
9    */
10
11  abstract class Member {
12
13      /**
14       * Return the member's (simple) name.
15       *
16       * @return the name.
17       */
18
19      public String name() {
20          return member().getName();
21      }
22
23      /**
24       * Return the type in which this member was declared.
25       *
26       * @return the declaring type.
27       */
28
29      public Type declaringType() {
30          return Type.typeFor(member().getDeclaringClass());
31      }
32
33      /**
34       * Has this member been declared with the static modifier?
35       *
36       * @return is the member static?
37       */
38
39      public boolean isStatic() {
40          return java.lang.reflect.Modifier.isStatic(member().getModifiers());
41      }
42
43      /**
44       * Has this member been declared with the public modifier?
45       *
46       * @return is the member public?
47       */
48
49      public boolean isPublic() {
50          return java.lang.reflect.Modifier.isPublic(member().getModifiers());
51      }
52
53      /**
54       * Has this member been declared with the protected modifier?
55       *
56       * @return is the member protected?
57       */
58
59      public boolean isProtected() {
60          return java.lang.reflect.Modifier.isProtected(member().getModifiers());
61      }
62
63      /**
64       * Has this member been declared with the private modifier?
65       *
66       * @return is the member private?
```

```
67       */
68
69      public boolean isPrivate() {
70          return java.lang.reflect.Modifier.isPrivate(member().getModifiers());
71      }
72
73      /**
74       * Has this member been declared with the abstract modifier?
75       *
76       * @return is the member abstract?
77       */
78
79      public boolean isAbstract() {
80          return java.lang.reflect.Modifier.isAbstract(member().getModifiers());
81      }
82
83      /**
84       * Has this member been declared with the final modifier? Note,we cannot
85       * declare anything final as it is not in our lexicon. But we may refer to
86       * names that are declared final in the java API and so we (may) want to
87       * honor those declarations.
88       *
89       * @return is the member final?
90       */
91
92      public boolean isFinal() {
93          return java.lang.reflect.Modifier.isFinal(member().getModifiers());
94      }
95
96      /**
97       * Return the member's internal representation.
98       *
99       * @return the internal representation.
100      */
101
102     protected abstract java.lang.reflect.Member member();
103
104 }
105
106 /**
107  * A Method knows its descriptor (its signature in JVM format), and its return
108  * type.
109  */
110
111 class Method extends Member {
112
113     /** Internal representation of this method. */
114     private java.lang.reflect.Method method;
115
116     /**
117      * Construct a Method is constructed from its internal representation.
118      *
119      * @param method
120      *             a Java method in the relection API.
121      */
122
123     public Method(java.lang.reflect.Method method) {
124         this.method = method;
125     }
126
127     /**
128      * Return the JVM descriptor for this method.
129      *
130      * @return the descriptor.
131      */
132
133     public String toDescriptor() {
134         String descriptor = "(";
135         for (Class paramType : method.getParameterTypes()) {
```

```
            descriptor += Type.typeFor(paramType).toDescriptor();
        }
        descriptor += ")" + Type.typeFor(method.getReturnType()).toDescriptor();
        return descriptor;
    }

    /**
     * Return the Java representation for this method.
     *
     * @return the descriptor.
     */

    public String toString() {
        String str = name() + "(";
        for (Class paramType : method.getParameterTypes()) {
            str += Type.typeFor(paramType).toString();
        }
        str += ")";
        return str;
    }

    /**
     * Return the method's return type.
     *
     * @return the return type.
     */

    public Type returnType() {
        return Type.typeFor(method.getReturnType());
    }

    /**
     * Method equality is defined HERE as having override-equivalent signatures.
     *
     * @param that
     *            the method we are comparing this to.
     * @return true iff the methods are override-equivalent.
     */

    public boolean equals(Method that) {
        return Type.argTypesMatch(this.method.getParameterTypes(), that.method
                .getParameterTypes());
    }

    /**
     * Return the member's internal representation.
     *
     * @return the internal representation.
     */

    protected java.lang.reflect.Member member() {
        return method;
    }

}

/**
 * A Field knows its type.
 */

class Field extends Member {

    /** Internal representation of this field. */
    private java.lang.reflect.Field field;

    /**
     * Construct a Field is constructed from its internal representation.
     *
     * @param field
```

```
205      *            a Java field in the relection API.
206      */
207
208     public Field(java.lang.reflect.Field field) {
209         this.field = field;
210     }
211
212     /**
213      * Return the field's type.
214      *
215      * @return the field's type.
216      */
217
218     public Type type() {
219         return Type.typeFor(field.getType());
220     }
221
222     /**
223      * @inheritDoc
224      */
225
226     protected java.lang.reflect.Member member() {
227         return field;
228     }
229
230 }
231
232 /**
233  * A Constructor knows its JVM descriptor.
234  */
235
236 class Constructor extends Member {
237
238     /** Internal representation of this constructor. */
239     java.lang.reflect.Constructor constructor;
240
241     /**
242      * Construct a Constructor from its internal representation.
243      *
244      * @param constructor
245      *            a Java constructor in the relection API.
246      */
247
248     public Constructor(java.lang.reflect.Constructor constructor) {
249         this.constructor = constructor;
250     }
251
252     /**
253      * Return the JVM descriptor for this constructor.
254      *
255      * @return the descriptor.
256      */
257
258     public String toDescriptor() {
259         String descriptor = "(";
260         for (Class paramType : constructor.getParameterTypes()) {
261             descriptor += Type.typeFor(paramType).toDescriptor();
262         }
263         descriptor += ")V";
264         return descriptor;
265     }
266
267     /**
268      * Return the type declaring this constructor.
269      *
270      * @return the declaring type.
271      */
272
273     public Type declaringType() {
```

```
274         return Type.typeFor(constructor.getDeclaringClass());
275     }
276 
277     /**
278      * @inheritDoc
279      */
280 
281     protected java.lang.reflect.Member member() {
282         return constructor;
283     }
284 
285 }
286 
```