Member.java

```java
// Copyright 2013 Bill Campbell, Swami Iyer and Bahar Akbal-Delibas

package jminusminus;

/**
 * A wrapper for members (eg Fields, Methods, Constructors) in the Java API.
 * Members are used in message expressions, field selections, and new object
 * construction operations.
 */

abstract class Member {

    /**
     * Return the member's (simple) name.
     *
     * @return the name.
     */

    public String name() {
        return member().getName();
    }

    /**
     * Return the type in which this member was declared.
     *
     * @return the declaring type.
     */

    public Type declaringType() {
        return Type.typeFor(member().getDeclaringClass());
    }

    /**
     * Has this member been declared with the static modifier?
     *
     * @return is the member static?
     */

    public boolean isStatic() {
        return java.lang.reflect.Modifier.isStatic(member().getModifiers());
    }

    /**
     * Has this member been declared with the public modifier?
     *
     * @return is the member public?
     */

    public boolean isPublic() {
        return java.lang.reflect.Modifier.isPublic(member().getModifiers());
    }

    /**
     * Has this member been declared with the protected modifier?
     *
     * @return is the member protected?
     */

    public boolean isProtected() {
        return java.lang.reflect.Modifier.isProtected(member().getModifiers());
    }

    /**
     * Has this member been declared with the private modifier?
     *
     * @return is the member private?
     */

    public boolean isPrivate() {
        return java.lang.reflect.Modifier.isPrivate(member().getModifiers());
    }

    /**
     * Has this member been declared with the abstract modifier?
     *
     * @return is the member abstract?
     */

    public boolean isAbstract() {
        return java.lang.reflect.Modifier.isAbstract(member().getModifiers());
    }

    /**
     * Has this member been declared with the final modifier? Note,we cannot
     * declare anything final as it is not in our lexicon. But we may refer to
     * names that are declared final in the java API and so we (may) want to
     * honor those declarations.
     *
     * @return is the member final?
     */

    public boolean isFinal() {
        return java.lang.reflect.Modifier.isFinal(member().getModifiers());
    }

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

    protected abstract java.lang.reflect.Member member();

}

/**
 * A Method knows its descriptor (its signature in JVM format), and its return
 * type.
 */

class Method extends Member {

    /** Internal representation of this method. */
    private java.lang.reflect.Method method;

    /**
     * Construct a Method is constructed from its internal representation.
     *
     * @param method
     *            a Java method in the relection API.
     */

    public Method(java.lang.reflect.Method method) {
        this.method = method;
    }

    /**
     * Return the JVM descriptor for this method.
     *
     * @return the descriptor.
     */

    public String toDescriptor() {
        String descriptor = "(";
        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
     *              a Java field in the relection API.
     */

    public Field(java.lang.reflect.Field field) {
        this.field = field;
    }

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

    public Type type() {
        return Type.typeFor(field.getType());
    }

    /**
     * @inheritDoc
     */

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

}

/**
 * A Constructor knows its JVM descriptor.
 */

class Constructor extends Member {

    /** Internal representation of this constructor. */
    java.lang.reflect.Constructor constructor;

    /**
     * Construct a Constructor from its internal representation.
     *
     * @param constructor
     *              a Java constructor in the relection API.
     */

    public Constructor(java.lang.reflect.Constructor constructor) {
        this.constructor = constructor;
    }

    /**
     * Return the JVM descriptor for this constructor.
     *
     * @return the descriptor.
     */

    public String toDescriptor() {
        String descriptor = "(";
        for (Class paramType : constructor.getParameterTypes()) {
            descriptor += Type.typeFor(paramType).toDescriptor();
        }
        descriptor += ")V";
        return descriptor;
    }

    /**
     * Return the type declaring this constructor.
     *
     * @return the declaring type.
     */

    public Type declaringType() {
```

```java
        return Type.typeFor(constructor.getDeclaringClass());
    }

    /**
     * @inheritDoc
     */

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

}

```