JMessageExpression.java

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

package jminusminus;

import java.util.ArrayList;

import static jminusminus.CLConstants.*;

/**
 * The AST node for a message expression that has a target, optionally an
 * ambiguous part, a message name, and zero or more actual arguments.
 */

class JMessageExpression extends JExpression {

    /** The target expression. */
    private JExpression target;

    /** The ambiguous part that is reclassfied in analyze(). */
    private AmbiguousName ambiguousPart;

    /** The message name. */
    private String messageName;

    /** Message arguments. */
    private ArrayList<JExpression> arguments;

    /** Types of arguments. */
    private Type[] argTypes;

    /** The Method representing this message. */
    private Method method;

    /**
     * Construct an AST node for a message expression without an ambiguous part.
     *
     * @param line
     *            line in which the expression occurs in the source file.
     * @param target
     *            the target expression.
     * @param messageName
     *            the message name.
     * @param arguments
     *            the ambiguousPart arguments.
     */

    protected JMessageExpression(int line, JExpression target,
            String messageName, ArrayList<JExpression> arguments) {
        this(line, target, null, messageName, arguments);
    }

    /**
     * Construct an AST node for a message expression having an ambiguous part.
     *
     * @param line
     *            line in which the expression occurs in the source file.
     * @param target
     *            the target expression.
     * @param ambiguousPart
     *            the ambiguous part.
     * @param messageName
     *            the message name.
     * @param arguments
     *            the arguments.
     */

    protected JMessageExpression(int line, JExpression target,
            AmbiguousName ambiguousPart, String messageName,
            ArrayList<JExpression> arguments) {
        super(line);
        this.target = target;
        this.ambiguousPart = ambiguousPart;
        this.messageName = messageName;
        this.arguments = arguments;
    }

    /**
     * Analysis of a message expression involves: (1) reclassifying any
     * ambiguous part, (2) analyzing and computing the types for the actual
     * arguments, (3) determining the type we are currently in (for checking
     * access), (4) analyzing the target and determining its type, (5) finding
     * the appropriate Method, (6) checking accessibility, and (7) determining
     * the result type.
     *
     * @param context
     *            context in which names are resolved.
     * @return the analyzed (and possibly rewritten) AST subtree.
     */

    public JExpression analyze(Context context) {
        // Reclassify the ambiguous part
        if (ambiguousPart != null) {
            JExpression expr = ambiguousPart.reclassify(context);
            if (expr != null) {
                if (target == null) {
                    target = expr;
                } else {
                    // Can't even happen syntactically
                    JAST.compilationUnit.reportSemanticError(line(),
                            "Badly formed suffix");
                }
            }
        }

        // Then analyze the arguments, collecting
        // their types (in Class form) as argTypes
        argTypes = new Type[arguments.size()];
        for (int i = 0; i < arguments.size(); i++) {
            arguments.set(i, (JExpression) arguments.get(i).analyze(context));
            argTypes[i] = arguments.get(i).type();
        }

        // Where are we now? (For access)
        Type thisType = ((JTypeDecl) context.classContext.definition())
                .thisType();

        // Then analyze the target
        if (target == null) {
            // Implied this (or, implied type for statics)
            if (!context.methodContext().isStatic()) {
                target = new JThis(line()).analyze(context);
            } else {
                target = new JVariable(line(), context.definingType()
                        .toString()).analyze(context);
            }
        } else {
            target = (JExpression) target.analyze(context);
            if (target.type().isPrimitive()) {
                JAST.compilationUnit.reportSemanticError(line(),
                        "cannot invoke a message on a primitive type:"
                                + target.type());
            }
        }

        // Find appropriate Method for this message expression
        method = target.type().methodFor(messageName, argTypes);
        if (method == null) {
            JAST.compilationUnit.reportSemanticError(line(),
                    "Cannot find method for: "
                            + Type.signatureFor(messageName, argTypes));
            type = Type.ANY;
        } else {
            context.definingType().checkAccess(line, (Member) method);
            type = method.returnType();

            // Non-static method cannot be referenced from a static context.
            if (!method.isStatic()) {
                if (target instanceof JVariable
                        && ((JVariable) target).iDefn() instanceof TypeNameDefn)
    {
                    JAST.compilationUnit
                            .reportSemanticError(
                                    line(),
                                    "Non-static method "
                                            + Type.signatureFor(messageName,
                                                    argTypes)
                                            + "cannot be referenced from a static
    context");
                }
            }
        }
        return this;
    }

    /**
     * Code generation for a message expression involves generating code for
     * loading the target onto the stack, generating code to load the actual
     * arguments onto the stack, and then invoking the named Method. Notice that
     * if this is a statement expression (as marked by a parent
     * JStatementExpression) then we also generate code for popping the stacked
     * value for any non-void invocation.
     * 
     * @param output
     *            the code emitter (basically an abstraction for producing the
     *            .class file).
     */

    public void codegen(CLEmitter output) {
        if (!method.isStatic()) {
            target.codegen(output);
        }
        for (JExpression argument : arguments) {
            argument.codegen(output);
        }
        int mnemonic = method.isStatic() ? INVOKESTATIC : target.type()
                .isInterface() ? INVOKEINTERFACE : INVOKEVIRTUAL;
        output.addMemberAccessInstruction(mnemonic, target.type().jvmName(),
                messageName, method.toDescriptor());
        if (isStatementExpression && type != Type.VOID) {
            // Pop any value left on the stack
            output.addNoArgInstruction(POP);
        }
    }

    /**
     * The semantics of j-- require that we implement short-circuiting branching
     * in implementing message expressions.
     * 
     * @param output
     *            the code emitter (basically an abstraction for producing the
     *            .class file).
     * @param targetLabel
     *            the label to which we should branch.
     * @param onTrue
```

```java
     *             do we branch on true?
     */

    public void codegen(CLEmitter output, String targetLabel, boolean onTrue) {
        // Push the value
        codegen(output);

        if (onTrue) {
            // Branch on true
            output.addBranchInstruction(IFNE, targetLabel);
        } else {
            // Branch on false
            output.addBranchInstruction(IFEQ, targetLabel);
        }
    }

    /**
     * @inheritDoc
     */

    public void writeToStdOut(PrettyPrinter p) {
        p.printf("<JMessageExpression line=\"%d\" name=\"%s\">\n", line(),
                messageName);
        p.indentRight();
        if (target != null) {
            p.println("<Target>");
            p.indentRight();
            target.writeToStdOut(p);
            p.indentLeft();
            p.println("</Target>");
        }
        if (arguments != null) {
            p.println("<Arguments>");
            for (JExpression argument : arguments) {
                p.indentRight();
                p.println("<Argument>");
                p.indentRight();
                argument.writeToStdOut(p);
                p.indentLeft();
                p.println("</Argument>");
                p.indentLeft();
            }
            p.println("</Arguments>");
        }
        p.indentLeft();
        p.println("</JMessageExpression>");
    }

}
```