Main.java

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

package jminusminus;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import static jminusminus.TokenKind.EOF;

/**
 * Driver class for j-- compiler using hand-written front-end. This is the main
 * entry point for the compiler. The compiler proceeds as follows:
 *
 * (1) It reads arguments that affects its behavior.
 *
 * (2) It builds a scanner.
 *
 * (3) It builds a parser (using the scanner) and parses the input for producing
 * an abstact syntax tree (AST).
 *
 * (4) It sends the preAnalyze() message to that AST, which recursively descends
 * the tree so far as the memeber headers for declaring types and members in the
 * symbol table (represented as a string of contexts).
 *
 * (5) It sends the analyze() message to that AST for declaring local variables,
 * and cheking and assigning types to expressions. Analysis also sometimes
 * rewrites some of the abstract syntax trees for clarifying the semantics.
 * Analysis does all of this by recursively descending the AST down to its
 * leaves.
 *
 * (6) Finally, it sends a codegen() message to the AST for generating code.
 * Again, codegen() recursively descends the tree, down to its leaves,
 * generating JVM code for producing a .class or .s (SPIM) file for each defined
 * type (class).
 */

public class Main {

    /** Whether an error occurred during compilation. */
    private static boolean errorHasOccurred;

    /**
     * Entry point.
     */

    public static void main(String args[]) {
        String caller = "java jminusminus.Main";
        String sourceFile = "";
        String debugOption = "";
        String outputDir = ".";
        boolean spimOutput = false;
        String registerAllocation = "";
        errorHasOccurred = false;
        for (int i = 0; i < args.length; i++) {
            if (args[i].equals("j--")) {
                caller = "j--";
            } else if (args[i].endsWith(".java")) {
                sourceFile = args[i];
            } else if (args[i].equals("-t") || args[i].equals("-p")
                    || args[i].equals("-pa") || args[i].equals("-a")) {
                debugOption = args[i];
            } else if (args[i].endsWith("-d") && (i + 1) < args.length) {
                outputDir = args[++i];
            } else if (args[i].endsWith("-s") && (i + 1) < args.length) {
                spimOutput = true;
                registerAllocation = args[++i];
                if (!registerAllocation.equals("naive")
                        && !registerAllocation.equals("linear")
                        && !registerAllocation.equals("graph")
                        || registerAllocation.equals("")) {
                    printUsage(caller);
                    return;
                }
            } else if (args[i].endsWith("-r") && (i + 1) < args.length) {
                NPhysicalRegister.MAX_COUNT = Math.min(18, Integer
                        .parseInt(args[++i]));
                NPhysicalRegister.MAX_COUNT = Math.max(1,
                        NPhysicalRegister.MAX_COUNT);
            } else {
                printUsage(caller);
                return;
            }
        }
        if (sourceFile.equals("")) {
            printUsage(caller);
            return;
        }

        LookaheadScanner scanner = null;
        try {
            scanner = new LookaheadScanner(sourceFile);
        } catch (FileNotFoundException e) {
            System.err.println("Error: file " + sourceFile + " not found.");
            return;
        }

        if (debugOption.equals("-t")) {
            // Just tokenize input and print the tokens to STDOUT
            TokenInfo token;
            do {
                scanner.next();
                token = scanner.token();
                System.out.printf("%d\t : %s = %s\n", token.line(), token
                        .tokenRep(), token.image());
            } while (token.kind() != EOF);
            errorHasOccurred |= scanner.errorHasOccurred();
            return;
        }

        // Parse input
        Parser parser = new Parser(scanner);
        JCompilationUnit ast = parser.compilationUnit();
        errorHasOccurred |= parser.errorHasOccurred();
        if (debugOption.equals("-p")) {
            ast.writeToStdOut(new PrettyPrinter());
            return;
        }
        if (errorHasOccurred) {
            return;
        }

        // Do pre-analysis
        ast.preAnalyze();
        errorHasOccurred |= JAST.compilationUnit.errorHasOccurred();
        if (debugOption.equals("-pa")) {
            ast.writeToStdOut(new PrettyPrinter());
            return;
        }
        if (errorHasOccurred) {
            return;
        }

        // Do analysis
        ast.analyze(null);
        errorHasOccurred |= JAST.compilationUnit.errorHasOccurred();
        if (debugOption.equals("-a")) {
            ast.writeToStdOut(new PrettyPrinter());
            return;
        }
        if (errorHasOccurred) {
            return;
        }

        // Generate JVM code
        CLEmitter clEmitter = new CLEmitter(!spimOutput);
        clEmitter.destinationDir(outputDir);
        ast.codegen(clEmitter);
        errorHasOccurred |= clEmitter.errorHasOccurred();
        if (errorHasOccurred) {
            return;
        }

        // If SPIM output was asked for, convert the in-memory
        // JVM instructions to SPIM using the specified register
        // allocation scheme.
        if (spimOutput) {
            NEmitter nEmitter = new NEmitter(sourceFile, ast.clFiles(),
                    registerAllocation);
            nEmitter.destinationDir(outputDir);
            nEmitter.write();
            errorHasOccurred |= nEmitter.errorHasOccurred();
        }
    }

    /**
     * Return true if an error occurred during compilation; false otherwise.
     *
     * @return true or false.
     */

    public static boolean errorHasOccurred() {
        return errorHasOccurred;
    }

    /**
     * Print command usage to STDOUT.
     *
     * @param caller
     *            denotes how this class is invoked.
     */

    private static void printUsage(String caller) {
        String usage = "Usage: "
                + caller
                + " <options> <source file>\n"
                + "where possible options include:\n"
                + "  -t Only tokenize input and print tokens to STDOUT\n"
                + "  -p Only parse input and print AST to STDOUT\n"
                + "  -pa Only parse and pre-analyze input and print "
                + "AST to STDOUT\n"
                + "  -a Only parse, pre-analyze, and analyze input "
                + "and print AST to STDOUT\n"
                + "  -s <naive|linear|graph> Generate SPIM code\n"
                + "  -r <num> Max. physical registers (1-18) available for allocation; default = 8\n"
                + "  -d <dir> Specify where to place output files; default = .";
        System.out.println(usage);
    }

}
```