Identifying and Correcting Java Programming Errors for Introductory Computer Science Students

Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri, Ph.D.
Mathematics and Computer Science Department
Bryn Mawr College
101 N. Merion Avenue
Bryn Mawr, PA 19010-2899
{rmercuri, amisra}@brynmawr.edu
{maria.hristova, megan.rutter}@alumnae.brynmawr.edu

Abstract

Programming in Java can be a daunting task for introductory students, one that is only compounded by the cryptic compiler error messages they see when they first start to write actual code. This article details a project conducted by faculty and advanced students in the creation of an educational tool for Java programming, called Expresso. This paper discusses some existing programming tools, explains their drawbacks, and describes why Expresso is different. We also include a detailed list of typical errors made by novice programmers, used in the construction of the Expresso tool.

Categories & Subject Descriptors

D.2.5 [**Software Engineering**]: Coding Tools and Techniques -- *Object-Oriented Programming*.

K.3.2 [**Computers and Education**]: Computer and Information Science Education.

General Terms

Languages, Design, Human Factors.

Keywords

Java, CS1, programming, syntax, semantics, logic.

1 Premise of Study

In recent years, a significant number of colleges have converted their introductory computer science courses from C/C++ to Java. This transition has brought attention to many programming errors and misconceptions that exist across this particular family of languages or are Java specific. For beginning programmers, it is often hard to comprehend linguistic intricacies inherent in the design of languages like Java, leading to a range of common difficulties during the coding process.

Despite extensive coverage of these types of errors in textbooks and lectures, we have observed that these still persist when students actually write programs. Though certain compilers may flag some of these mistakes, often the Java error messages are so cryptic to students that they have a hard time simply identifying their errors, let alone making corrections.

In order to investigate this problem, we formed a collaborative project between computer science faculty and students who had previously served as Java teaching assistants. The study was divided into three phases as follows:

Phase 1: Identification of Errors

We conducted a survey of college professors, teaching assistants, and students in Computer Science to identify relevant Java programming mistakes. Combining the survey results with our own experiences as teaching assistants, we obtained a final target list of errors and misconceptions.

Phase 2: Choice of Implementation

Next, we decided how to implement the error-detection advisory tool. As a part of this process, we examined some existing educational programming environments. We had initially contemplated writing ours as a preprocessor or a dynamic and interactive part of an editor. Later, we considered implementing it as a post-compiler designed to translate cryptic compiler messages into comments that introductory students could more easily comprehend. We finally decided to write our tool as a pre-compiler, a program to be run before compilation.

Phase 3: Implementation and Evaluation

We then wrote code to implement our program. After developing a working version, we evaluated it using our own test suite, with the eventual intention of having a Java class use it as well.

2 List of Common Java Errors

We started by collecting a list of Java programming errors based on those reported by the teaching assistants. Next, we contacted computer science professors from 58 schools listed in the US News and World Report's Top 50 Liberal Arts Colleges for 2002 and members of the Special Interest Group on Computer

Science Education (SIGCSE) of the Association for Computing Machinery. We asked the teachers what they felt were the five most commonly made programming errors by introductory Java students, as well as what were the three hardest programming errors to find and/or fix given the compiler messages. Finally we collected data from students enrolled in introductory undergraduate Java courses.

After collecting the survey data, we combined the information provided by faculty and students and divided it into three general categories: syntax errors, semantic errors, and logic errors. Syntax errors refer to mistakes in the spelling, punctuation and order of words in the program. Frequently, syntax errors render a program incomprehensible to compilers and are thus easily identified. While compilers often detect the obvious syntax errors, their error messages do not necessarily point the students in the right direction needed to fix the code. Semantic errors deal with the meaning of the code. In particular, we identified errors that are likely to ensue from a mistaken idea of how the language interprets certain instructions. Most of these types of errors are specific to Java and similar languages but occur on a more abstract level than the syntax errors. Logic errors are often the most general errors. We included here those errors that tend to arise from fallacious thinking by the programmer rather than from language peculiarities, although some may manifest themselves via improper syntax or semantics.

Disagreement on the correct category for a given error is, of course, possible. In fact, several of the errors in our list seem to belong in more than one category—sometimes in all three. We attempted to separate them based more on the thought process of an introductory Java programmer than on that of a compiler. This distinction enabled us to approach errors from the user's point of view, and consequently to provide meaningful error messages.

The next step was figuring out which errors we wanted our program to consider. There were some errors that didn't need to be included because we felt that the compiler our students would be using (CodeWarrior) satisfactorily identified the error in a way that the student would be able to fix it. There were other errors (mostly from the surveys and questionnaires) that we chose not to include because they dealt with Java concepts that were too complex or too compiler-specific and therefore didn't apply to our introductory students. The other errors that we didn't include were errors that we either did not understand, or that we understood but could not implement (due to time constraints, and the programming backgrounds of the students writing the pre-compiler). Further errors included in this category were ones that we simply could not check for because they were run-time errors, not compile-time errors.

The original list contained 62 reported errors. Of these, 20 were identified as those we felt were essential to the educational aspect of our project. These are listed below.

Syntax Errors

1. = versus ==

Confusing the assignment operator (=) with the comparison operator (==). This can lead to inadvertent reassignment in conditional expressions.

2. == versus *.equals* (faulty string comparisons)

Use of == instead of *.equals* to compare strings. While *.equals* compares the values of two strings, == compares only their memory locations.

3. mismatching, miscounting and/or misuse of *{ }*, *[ ]*, *( )*, *" "*, and *' '*

Unbalanced parentheses, brackets, square brackets and quotation marks, or using these different symbols interchangeably.

4. && vs. & and || vs. |

Confusing "short-circuit" evaluators (*&&* and ||) with conventional logical operators (*&* and |). Depending on the value of the first condition, "short-circuit" evaluators may ignore the second condition. This can cause problems if the user relies on the second condition being evaluated.

5. incorrect semi-colon after an *if* selection structure before the *if* statement or after the *for* or *while* repetition structure before the respective *for* or *while* loop

Inserting a semi-colon after the parentheses defining *if*, *for*, or *while* conditions results in the program's doing nothing when the *if* condition is true, or for the duration of the *for* or *while* loop. This is valid code but generally undesirable and often inadvertently done by beginning programmers.

6. wrong separators in *for* loops (using commas instead of semi-colons)

Separating the initialization, testing and update clauses of a *for* structure with commas or other punctuation in place of semi-colons.

7. an *if* followed by a bracket instead of by a parenthesis

Inserting the condition of an *if* statement within brackets instead of parentheses.

8. using keywords as method names or variable names

There are 50 keywords in Java that cannot be used outside of their given purpose. This means that programmer-defined functions and new variables cannot have keywords as their names. Note: It is possible to use the same words in code because Java is case-sensitive. However, this is not recommended as it decreases readability.

9. invoking methods with wrong arguments

The types of the arguments in a method call must match the types of the arguments in that method's definition.

10. forgetting parentheses after method call

When a method is called, it is always followed by parentheses, sometimes with arguments in them.

11. incorrect semicolon at the end of a method header

There should never be a semicolon at the end of a method header (the first line of a method definition).

12. leaving a space after a period when calling a specific method

There should never be a space after a period, excluding, of course, comments and strings.

13. >= and =<

The equal sign in a greater than or equal (or less than or equal) comparison operator always follows the greater than sign (or less than sign).

Semantic Errors

1. invoking class method on object

A common error is trying to invoke a method that belongs to a class on a variable or an object directly. The concept of having a method executed usually is perceived as having to perform it directly on the variables or objects that it needs to be applied. Since Java has a complicated object hierarchy, one can not invoke a class method on an object directly, rather this is done by giving the object or variable type first before the "." operator.

Logic Errors

1. improper casting

This problem occurs when a variable is declared rather than cast (i.e. the casting parentheses are missing). One possible result may involve truncation of important data. An error of this type can also occur as a result of integer division. Introductory students tend to believe that numbers are just numbers and fail to comprehend the differences and data type necessities between *int* and *float*, for example.

2. invoking a non-void method in a statement that requires a return value

A method that is supposed to return a variable of some type (i.e. it must be made equal to a variable of the return type) is instead called as a void method or as a statement. If this mistake is made, the value returned by the method is lost since it is not stored anywhere.

3. flow reaches end of non-void method

A non-void method is supposed to return a value of some type, but the return statement is missing due to misunderstanding about the role or type of the method or just forgetfulness.

4. methods with parameters: confusion between declaring parameters of a method and passing parameters in a method invocation

When a method is defined, the parameter types need to be declared. However, in a method invocation the types of the variables passed are not given, only the variable names. There exists a confusion between passing parameters, declaring them, and identifying them in the method's definition.

5. incompatibility between the declared return type of a method and in its invocation

A non-void method that is supposed to return a value of a particular data type but the variable that will receive the return value is of an incompatible type.

6. class declared abstract because of missing function

A class that implements some interface but is missing one of the major methods that the interface must define and support.

3 Implementation of the Educational Tool

We examined existing tools such as TA Online[1], DrScheme[2], and BlueJ[3], and drew ideas and inspiration from these when deciding how to write our educational tool and in selecting what goals to focus on.

Of the tools we examined, none were suitable for our needs. TA Online provides a catalog of over 100 commonly made Java mistakes, but it is meant to be used as a reference and does not interact with the students' code. DrScheme is a highly interactive environment that highlights erroneous code, but is implemented for the Scheme language. Our resulting tool eventually focused on many of the same problems with the Java language as the DrScheme tool did, as much as there exists correspondence between these vastly different languages. BlueJ is also an interactive tool, and it is intended for use with Java, but we found that it creates some distance between the student programmer and their code. For example, there is no need to declare a main function for every program because it is generated by one of the BlueJ template classes. While this may save hand-waving and explanations like "just write it and you will understand it later," in the early weeks of a course, it may cause students to be unaware of the need for such functions, and create a dependency that could later result in programming difficulties. Since a lot of code is generated for the student, we felt that they may even need to re-learn some Java basics when a regular compiler is used.

Our tool differs from the ones we examined because it specifically does not eliminate the need for understandable compiler error messages; rather, our tool enhances the functions of a compiler. The intention was to create a helpful interactive tool that would do a better job generating error messages than existing compilers and also provide suggestions on how to fix the code. Our tool is targeted for use during the beginning process of learning programming and we believe that the need for students to use it should decline as they become more proficient with Java and gain a better understanding of the essential programming concepts.

Once the final list of errors our program would consider was developed, we began implementing the code. We decided to call the program "Expresso" (misspelling intentional) and we wrote it in C++ as a multiple-pass pre-processor. The first pass inputs the programmer's file and removes comments, while keeping track of line numbers, and storing the resulting characters in a vector. The second pass removes white space and tokenizes the file, storing the result as a vector of words. Words were identified using punctuation and white space as delimiters. The final pass detects the mistakes, printing out an error message when appropriate.

We created a test suite of short programs in order to exercise the individual errors that Expresso could identify. This test suite can be useful, beyond the debugging process for our Expresso project, as example code for students to help them understand the types of errors they are making. This could be presented by the teacher or TA in a classroom setting, or individually when working with a student who is having difficulty programming.

Included (at the end of this article) is an example Java program, combining a number of different types of errors, that was used to demonstrate the Expresso code. The output generated by Expresso is also shown. Note that error messages do not all appear in sequential line number order, due to the multi-pass process.

4 Conclusions and Future of the Project

We believe that the Expresso tool will be useful for students in introductory Java programming classes. Additional helpful materials generated by our project included the list of common Java programming errors and the test suite. The error messages and analysis could be adapted for use with other similar languages, such as C and C++.

Future work on this project includes an assessment of Expresso in an actual classroom environment, by using it in conjunction with Java courses on our campus. Feedback from students and professors will provide insight as to whether it is an effective tool, and also how it might be improved. It will be interesting to observe whether students' programming skills change over time with the use of this tool. We would also like to combine it with an exsiting Integrated Development Environment in order to facilitate its use.

5 Acknowledgements

This project was funded by the Collaborative Research Experience for Women, part of the Computer Research Association. Thanks also to Bryn Mawr College Computer Science Division Chair, Deepak Kumar for assistance in obtaining this grant.

References

[1] TA Online: Common Java Compiler Errors, Dept. of Computer Science, University of Arizona, Feb. 2002. http://www.cs.arizona.edu/people/teena/ta_online/

[2] PLT Scheme: Software: DrScheme Home Page, Jan. 2002. http://www.plt-scheme.org/software/drscheme/

[3] Kolling, Michael, The BlueJ Tutorial, Jan. 2002. http://www.bluej.org/tutorial/tutorial.pdf

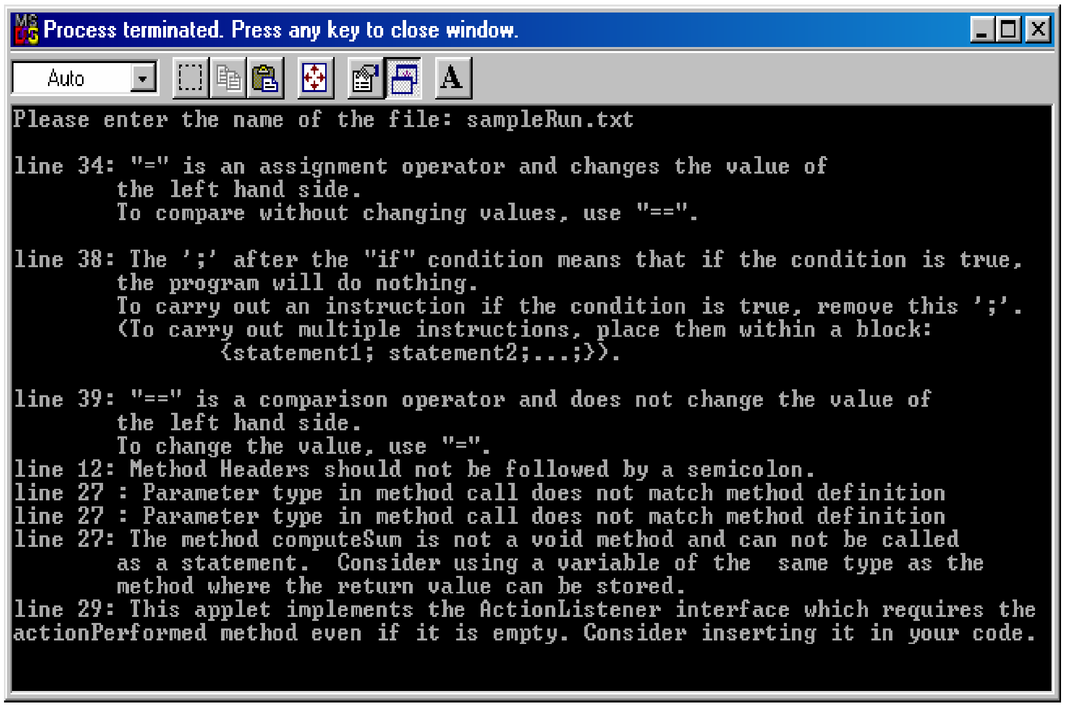
Sample program containing Java errors:

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
public class SampleRun extends Applet implements
ActionListener {
        Label sampleLabel = new Label("Sample Label",
Label.LEFT);
        Panel samplePanel = new Panel();
        public void init( );{
                samplePanel.setLayout(new
FlowLayout(FlowLayout.LEFT));
                samplePanel.add(sampleLabel);
                add(samplePanel);
                repaint();
        }
        public void paint(Graphics g){
                char bear = 'p';
                String cheese = "appleSauce";
                computeSum(cheese, bear);
        }

        public int computeSum(int apple, int sauce){

                int appleSauce;
                if (apple = sauce){
                        appleSauce= apple + sauce;
                }
                else if (apple > sauce);{
                        appleSauce == apple - sauce;
                }
                else{
                        appleSauce = 49;
                }
        return appleSauce;
        }
}
```

Output of Expresso run on sample program:



```
Please enter the name of the file: sampleRun.txt

line 34: "=" is an assignment operator and changes the value of
        the left hand side.
        To compare without changing values, use "==".

line 38: The ';' after the "if" condition means that if the condition is true,
        the program will do nothing.
        To carry out an instruction if the condition is true, remove this ';'.
        (To carry out multiple instructions, place them within a block:
                {statement1; statement2;...;}).

line 39: "==" is a comparison operator and does not change the value of
        the left hand side.
        To change the value, use "=".
line 12: Method Headers should not be followed by a semicolon.
line 27 : Parameter type in method call does not match method definition
line 27 : Parameter type in method call does not match method definition
line 27: The method computeSum is not a void method and can not be called
        as a statement.  Consider using a variable of the  same type as the
        method where the return value can be stored.
line 29: This applet implements the ActionListener interface which requires the
actionPerformed method even if it is empty. Consider inserting it in your code.
```