Assessing Roles of Variables by Program Analysis

Craig Bishop and Colin G. Johnson
Computing Laboratory
University of Kent at Canterbury
Canterbury, Kent, CT2 7NF, England
Email: `C.G.Johnson@kent.ac.uk`

ABSTRACT

The idea of *roles of variables* is to provide a vocabulary for describing the way in which variables are used by experienced programmers. This paper presents work on a system that is designed to automatically check students' role assignments in simple procedural programming. This is achieved by applying program analysis techniques, in particular program slicing and data flow analysis, to programs that students have written and annotated with role assignments.

1. INTRODUCTION

One important area of difficulty in the teaching of programming is bridging the gap between what can be done with a programming language, and what are the typical idioms used by programmers when they use that language. Structures in programming languages are very general; however, in "real" programming these structures are typically used in a small number of ways most of the time.

Typically, this has been regarded as part of the *tacit knowledge* [15] that is acquired by immersion in the activity of programming. However, in recent years a number of attempts have been made to articulate this knowledge in a more concrete fashion. Examples include the idea of *design patterns* [8], which articulate the ways in which high-level behaviour is achieved by the interaction of classes playing certain roles in an object-oriented system; and the idea that is at the core of this paper, *viz.* assigning high-level "semantic" roles to variables alongside the basic labelling which says what sort of thing the variable contains [12]. In general we can see this as a move from expressing static information about the kind of thing something *is* to expressing the kind of thing that it *does.*

One way to support such ideas is to develop computer-based tools to aid learners in applying such concepts. Such tools can help learners by checking whether these semantic ideas are consistent with the syntax they have written, annotating program code that they have written with some interpretation of that code, providing a visual interpretation of the code with regard to the semantic information, et cetera.

The aim of this paper is to discuss such a tool. The aim of it from the learner's point of view is that the learner specifies the role that they believe the variable to be playing in their program, then the tool analyses whether that role has been correctly assigned. This is achieved by carrying out a *slicing* of the program (i.e. rewriting the program so that just those parts of the program that affect a particular variable are included), then constructing a data flow graph from the sliced program. We then analyse properties of that graph, and the sliced program itself, for properties which suggest particular variable roles.

The paper is structured as follows. We begin with two short reviews of the roles of variables concept and the ideas of program analysis and slicing. We then discuss how the tool has been constructed, and go on to discuss ongoing work.

2. ROLES OF VARIABLES

When an experienced programmer creates a variable, they have a number of things in mind. Most obvious are the two things that we teach explicitly to students: the type of information that the variable contains (`int`, `String` ...); and the information that is represented by that variable, e.g. a piece of real-world data, some information that is significant in an algorithm, a piece of information input by a user, ...

However, the experienced programmer will also make use of their (typically tacit) knowledge about how variables are typically used in programming. Occasionally, this knowledge is articulated: for example programmers will talk about a "constant" or a "loop index". Occasionally, (as with the idea of a *constant*) the computer language provides a way to talk about these; on other occasions there is an informal natural language description that is used amongst programmers (e.g. the notion of a *loop index*). However, in many cases there is no explicit term with which to discuss these concepts.

To tackle this the idea of explicitly named *roles of variables* has been introduced by Sajaniemi and colleagues [12]. This work identifies a number of ways in which variables are used in programs, and provides names for these roles. An example of such a role is *gatherer.* A variable playing this role accumulates information as the program goes on, e.g. accumulating an aggregate or average value as data is input, or generating an index or ordering as new items are added.

Interestingly, a small number of such roles describe the vast majority of variable uses in simple procedural programming. It has been demonstrated that ten roles are sufficient to cover 99% of variables used in a large collection of programs [12, 18], and that experienced programmers agree with a high degree of consistency on which roles should be assigned to which variables [1].

These roles can be used in teaching in a number of ways. Primarily, they are used to aid discussion of programming between teachers and students, by providing a set of named concepts that can help to understand how experienced programmers work, and thus provide a bridge between naïve and experienced programmers. Another way in which these have been used is in program visualisation [17], using an intuitively appealing cartoon-like image associated with each role.

Most work on roles of variables has focussed on procedural programming. Recently, some preliminary work has been carried out on analysing roles in an object-oriented framework [5]. However, for the purposes of this paper we focus solely on procedural programs.

3. PROGRAM SLICING AND PROGRAM ANALYSIS

The standard thing to do with a program is to run it on a set of input data or through interaction with users. However, if we want to *understand* how a program behaves we can do many other things to a program; in general, such techniques are known as *program analysis* [14].

Program analysis techniques typically consist of two stages: transforming the program with regard to the kind of analysis that is to be attempted, then following some information through that transformed program. Typically the aim will be to follow some property of the information processed by the program through *all* possible program paths; often this is achieved through making some (conservative) approximation to that information.

An example of this is *abstract interpretation*, where we choose a property of a variable (e.g. "does this number ever go negative?"), aggregate the values that the variable could take into a set of abstract values (e.g. "known-positive', "known-negative", "don't know"), transform the program so that operations act on that abstraction rather than the concrete value (e.g. "if a known-positive number is added to a known-positive number, then the result is a known-positive number"), then follow the transformed program through all possible branching points examining how the abstracted values of those variables are transformed in the program.

Two program analysis techniques of particular importance to the work in this paper are *program slicing*, which transforms a program with respect to a variable by removing all parts of the program which cannot influence the value of that variable; and *data-flow analysis*, which generates a graph summarising the paths that can be taken through the program and analyses properties of that graph. Details of these will be given as needed during the description of the tool development below.

4. DEVELOPING A TOOL FOR CHECKING ROLES OF VARIABLES

4.1 Overview and Stance

We have been developing a tool that uses program analysis techniques to check variable role assignments in procedural code sections in the Java language. Each time a new variable is introduced the student annotates their code with a structured comment saying what role they believe the variable is playing. The tool then analyses the program to check whether that role assignment is correct. If the tool cannot successfully assign a role to the variable, then it reports a "don't know" value.

The development of this tool is an attempt to chart a middlebrow route between two extremes. By using analysis of the program rather than simply testing it on a number of examples, the tool aims to be more rigorous than a tool that is based simply on sample runs and heuristics. Nonetheless the aim is not to produce a system where the analysis is backed up by a formal proof of correctness of that role assignment.

4.2 Transforming the Program

The input to the tool is an annotated program: a Java program which is self-contained within a single class, where variable declarations are accompanied by a comment which states the role that the student believes the variable to be playing. Here is an example (modified from one of the examples at `http://www.cs.joensuu.fi/~saja/var_roles/`):

```
import java.io.*;

public class Doubles {

    private int counter; //%%counter%%stepper%%
    private int number; //%%number%%most recent holder%%

    public Doubles() {
        do {
            System.out.print("Give amount of loops: ");
            counter = getInput();
        }
        while (counter < 0);
        while (counter > 0) {
            System.out.print("Give some number: ");
            number = getInput();
            System.out.println("Two times " + number
                    + " is " + 2*number);
            counter = counter - 1;
        }
    }

    public int getInput() {
        int returnInt = 0;
        BufferedReader in = new BufferedReader(new
          InputStreamReader(System.in));
        try {
            returnInt = new Integer(in.readLine())
                                .intValue();
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
        return returnInt;
    }
}
```

As discussed in [16, 9] the role of a variable has more to do with the way in which the data flows through the variable than value of the variable itself. A formal way of transforming programs in such a variable-centric fashion is given by the theory of *program slicing* [2, 19, 10]. This was devised by Weiser [20] as a formalisation of the process used in debugging programs whereby programmers consider a variable in a program by working out which parts of the program can impact upon the value of that variable. A *slice* of a program with respect to a variable is the program with all statements which cannot impact on that variable removed.

For the purposes of the present project, there should not be any need to undertake the total decomposition of the source code described in [7] where every variable having an impact on the value of the analysed variable is isolated. Instead it should be sufficient to look for each variable, at where and under what circumstances it is assigned and used (though some provision may need to be made for indirect

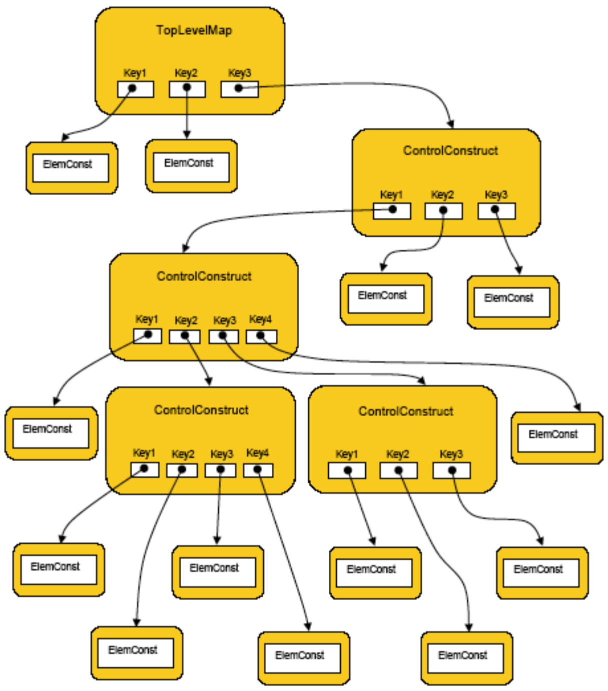
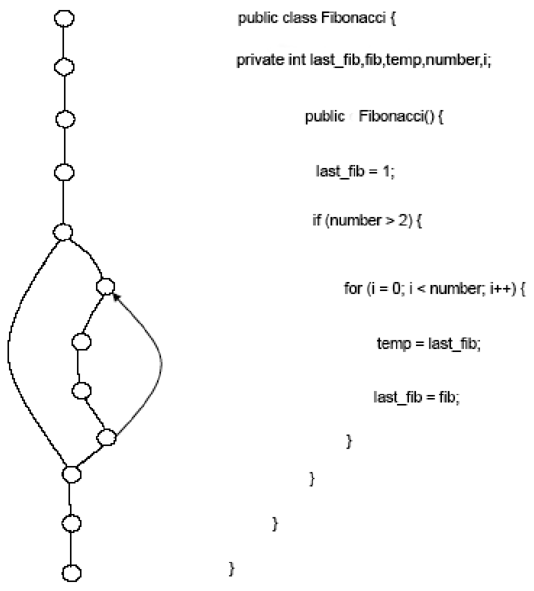


Figure 1: Source Code Map



```
public class Fibonacci {
private int last_fib,fib,temp,number,i;
public   Fibonacci() {
last_fib = 1;
if (number > 2) {
for (i = 0; i < number; i++) {
temp = last_fib;
last_fib = fib;
}
}
}
}
```

Figure 2: Program Slice for `last_fib`.

usage of the variable e.g. when two variables are used in combination to set a flag that acts as a condition for the loop).

In the tool we create the slice from a source code map (illustrated in figure 1); from this statements relating to each variable for which a role has been declared can be extracted. A sensible starting point for this in each program is the map containing the variables to be checked. Taking the key set of the variable map, the source code map can be traversed and each of the control constructs encountered stored in the order that they occur, until a construct (elementary, or control) containing the variable in question, is found. Once an occurrence of the given variable has been found, it is stored with its associated control constructs in a list. The map is traversed again looking for the next occurrence of the variable, with any control constructs and variable statements stored in a separate list from the first. This process can be repeated until every occurrence of the variable has been found. In this way it is possible to identify the position of each incidence of the variable being analysed in the program. This process can be repeated for all variables in the program for which a role has been declared. It is then possible to reconstruct the program from the perspective only of a given variable. This is the program slice. Figure 2 shows a program slice diagram for the variable `last_fib` from the Fibonacci program given in full in the Appendix.

4.3 Analysing the slice

Having obtained a slice of the program with respect to the variable role, we now need to analyse the slice to determine whether the role specified in the annotation is the actual role played. In doing this we are primarily concerned with *data flow* analysis of the program, as it is such data flow properties that primarily define the role that the variable is playing.

In [13] Moonen describes a generic architecture for data flow analysis, and concludes that there is no need to be specific about control constructs within the program being analysed. Any loop can be modelled simply as a loop having a condition under which it should continue to run. Similarly, any branch can be modelled as a branch and condition under which it should be followed in the sequence of executed statements. Moonen usefully differentiates between *variable definition* comprising input statements and those where the variable appears on the left hand side of equations, and *variable use* such as output statements and those where the variable appears on the right hand side of equations. According to [1], the role of a variable captures its behaviour by characterising the dynamic nature of the variable, i.e. the sequence of its successive values as related to other variables and external events. The analysis of training programs undertaken in the present project does not entirely support the view in [12] that usage does has no effect on the role of a variable, but the idea of separating statements involving variables into those in which the variable is defined and those in which the variable is used, provides a useful starting point for breaking down the sliced statements. From the available training programs, four distinct categories of variable statement are identified as follows:

1. **Assignment statements**, i.e. those where the variable appears on the left hand side of an equation as described in [13]. Java appears to have some advantages over other programming languages such as Pascal in this respect, as direct assignment statements always contain an $=$ operator on the left hand side of which appears the variable in question[1].

[1]Indirect assignment of a variable e.g. via input of an integer

2. **Usage statements**, i.e. those where the variable is either output to terminal (for use by the program user), directly used to assign some value to another variable, or as input to a method. This is roughly equivalent to the description of variable use in [13].

3. **Conditional statements**, i.e. those statements where the variable appears either as the condition for a loop, or as a condition for a branch in the software.

4. **Other statements**, e.g. those where the variable is declared, or returned.

In order to aid with the construction of analyses that would suggest variable roles, a tool was written which took example programs and listed each variable together with a description of how that variable was assigned and used (in particular where the variable was assigned and modified with respect to loop structures in the program).

By comparing these assignment and usage conditions with the roles we identified 21 conditions which can be used to build up checks for role assignment (table 1).

By using these conditions we have built a set of conditions for each role under which that role will *fail* to apply. This gives us a way of checking the student's role assertion and providing feedback if the role assignment appears to be incorrect.

As an example consider the role *most recent holder.* This has the failure condition

$$\begin{aligned} Y = \quad & F \vee \neg A \vee (Q \wedge R) \vee (Q \wedge X) \vee M \vee K \vee I \vee J \vee \\ & S \vee H \vee L \vee (\neg G \wedge D \wedge \neg P) \\ & \vee (\neg T \wedge \neg U) \vee (\neg B \wedge (\neg C \vee D \vee E)) \end{aligned}$$

where Y is a predicate indicating that the rule assignment is correct. If any of the subconditions are not met, feedback can be given appropriately using the messages in table 2.

In developing the rule sets, we have endeavoured where possible to create a rule set for each role that has minimal overlap with rule sets for other roles. This is to stop incidences where a variable appears to conform to the rule set of more than one role.

One exception to this is the case of the rule set for most recent holder. In that rule set, one of the rules indicating that the variable is not a most recent holder is as follows: $R = A \wedge B$, where R signifies an incorrect role declaration, A signifies that the variable is of type array, and B signifies that the variable is not playing the role fixed value. This break with convention is to enable the program to recognize in programs such as `Histogram.java` (see Appendix), that the float Array (amount) has the role fixed value and not the role most recent holder. The problem in that case is that the array appears to be used within the loop in which it is assigned, even though only one of the array values is being used and that value is actually assigned only once (i.e. it is a fixed value). At present, the variable is reckoned by the analysis software to have the role fixed value, only because even though it appears to be used in the loop within which it is assigned, it is also of type Array.

Moreover, the analysis program recognises the variable as not having the role most recent holder, only because one of the rules is that the variable shall not be an Array and also

value to a method, is not covered in the present version of the software.

A	variable is assigned in a loop.
B	variable is used in its assignment loop.
C	variable is used conditionally in its assignment loop.
D	variable is used directly for its assigning loop condition.
E	variable is used indirectly for its assigning loop condition.
F	variable is assigned in "for" loop statement.
G	variable is used directly in the program.
H	variable is assigned in a branch for which it is part of the condition.
I	variable appears directly on both sides of assignment statement.
J	variable appears indirectly on both sides of assignment statement.
K	variable is directly toggled within a loop.
L	variable is indirectly toggled within loop.
M	variable is incremented/decremented within a loop.
N	variable is used outside of loop in which it is assigned.
O	variable is assigned in loop before it is used in that loop.
P	variable is used conditionally for a loop outside of its assignment loop.
Q	variable appears in array organizing type statement.
R	variable is of type array.
S	variable is assigned within a loop with a combination of other variables, values and operators.
T	variable is assigned with the output from a method call.
U	variable is assigned with a value resulting from instantiation of a new object or directly with boolean value.
(X)	(variable is playing role *fixed value*).

Table 1: The 21 rules (plus one role assignment that plays the role of an additional rule) from which the role assessments are constructed.

conform to the rule set for fixed value. This combination of rules is circular and self satisfying. What more, it effectively means that it is not possible for a variable to be of type Array and have the role most recent holder. Clearly, this is not an adequate rule for either fixed value, or most recent holder, and it serves to highlight the additional difficulties faced when trying to define rules that hold for variables of type array and other primitive types. It also perhaps serves to highlight the sort of problem that may be encountered if trying to deal with other more complex variables such as those that refer to an object. Further analysis is therefore required for variables of type Array, perhaps to the extent that additional categories of variable statement should be defined for Array type variables, such as “Array value assignment”, and “Array value usage”.

It is desirable to add information to the output of the program that can be used to illustrate why a declared role is thought to be incorrect, so prioritisation is used for the rules within the rule set. Prioritisation is also required, because some rules are more specific than others. E.g. if a variable having a declared role of fixed value is actually a gatherer, it may fail the fixed value check because it is used within the loop within which it is assigned, as well as because the variable appears on both sides of an assignment statement. In such cases it is preferable to identify the reason for the incorrect role declaration as the detection of a specific gatherer (on both side of assignment) type statement, rather than the more generic reason that a fixed value variable shall not

Condition under which variable is adjudged not to be fixed value.	**Message generated**
F	"assigned in for loop statement"
$\neg A$	"not assigned in loop"
$(Q \wedge R)$	"appears to be organizer statement"
$(Q \wedge X)$	"is array filled in loop"
M	"incremented/decremented within loop"
K	"toggled within loop"
I	"appears on both sides of assignment"
J	"indirectly appears on both sides of assignment"
S	"assigned in loop with combination of other variables, operators and constants"
H	"used as condition for branch in which assigned"
L	"appears to be indirectly toggled within loop"
$(\neg G \wedge D \wedge \neg P)$	"condition for loop in which assigned and limited use outside of loop"
$(\neg T \wedge \neg U)$	"always assigned in loop with value of variable"
$(\neg B \wedge (\neg C \vee D \vee E))$	"not used in loop in which assigned"

Table 2: Messages given on failure: Most Wanted Holder.

be used in the same loop in which it is assigned. This prioritisation also supports the suggestion of roles to the programmer, as the offending statement (e.g. "variable appears on both sides of assignment statement") can be presented with the incorrect role detection announcement, reason for the detection, and suggested role for the variable.

5. CURRENT STATE AND FUTURE PLANS

The software has been implemented, and is currently able to handle procedural code sections in Java. There are a small number of limitations on the programs that will be processed by the tool: in particular variable overriding and `switch`/`case` statements are not currently supported.

At present we have focused our attention on using the analysis to identify faults in the student's role assignment. A second approach, which would add robustness to the role checking, would be to identify *positive* features in the data flow graph which indicate that a particular role is certainly being played. This could involve additional analysis techniques, for example *model checking* [6, 11], which allows statements about variables in temporal logic (e.g. "if A is positive, then at some time before the program ends B is guaranteed to be negative, regardless of the path taken through the program") to be confirmed or counterexamples found.

We have tested the program on the seventeen programs given at `http://www.cs.joensuu.fi/~saja/var_roles/` (which were used for the experiments described in [1, 5]) and all roles are correctly identified by the tool (also we have shown that experiments with erroneous variants on those programs correctly identify the errors). Clearly, this is a good thing; nonetheless such programs will have been devised specifically as clean examples of the role concept, and as a result future evaluation work will focus on a wider variety of programs.

We have incorporated the role analysis tool into the BlueJ programming/learning environment (`http://www.bluej.org/`) and hope to use this in practical evaluation of the tool in real classroom settings.

Further details of this work are given in [3] and on our web site at `http://www.cs.kent.ac.uk/people/staff/cgj/research/rolesOfVariables.html`.

6. REFERENCES

[1] Mordechai Ben-Ari and Jorma Sajaniemi. Roles of variables as seen by CS educators. In Boyle [4], pages 52–56.

[2] David Binkley and Keith Brian Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.

[3] Craig Bishop. Roles of variables and program analysis. Master's thesis, University of Kent at Canterbury, Computing Laboratory, 2005.

[4] Roger Boyle, editor. *Proceedings of the 9th Annual ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE 2004)*. Association for Computing Machinery, 2004.

[5] Pauli Byckling, Petri Gerdt, and Jorma Sajaniemi. Roles of variables in object-oriented programming. In *Proceedings of the Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2005) Educators Symposium.* ACM Press, 2005.

[6] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, 1999.

[7] Keith B. Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1994.

[9] Petri Gerdt and Jorma Sajaniemi. An approach to automatic detection of variable roles in program animation. In A. Korhonen, editor, *Proceedings of the Third Program Visualization Workshop.* University of Warwick Department of Computer Science Report CS-RR-407, 2004.

[10] Mark Harman and Rob Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.

[11] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, 2000.

[12] Marja Kuittinen and Jorma Sajaniemi. Teaching roles of variables in elementary programming courses. In Boyle [4], pages 57–61.

[13] Leon Moonen. A generic architecture for data flow analysis to support reverse engineering. In *Proceedings*

of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications, 1997.

[14] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer, 1999.

[15] Michael Polyani. *Personal Knowledge: Towards a Post-Critical Philosophy.* Routledge and Kegan Paul, 1958.

[16] Jorma Sajaniemi. An empirical analysis of roles of variables in novice-level procedural programs. In *Proceedings of the 2002 IEEE Symposium on Human-Centric Computing Languages and Environments*, pages 37–39, 2002.

[17] Jorma Sajaniemi and Marja Kuittinen. Visualizing roles of variables in program animation. *Information Visualization*, 3(3):137–153, 2004.

[18] Jorma Sajaniemi and Marja Kuittinen. An experiment on using roles of variables in teaching introductory programming. *Computer Science Education*, 15(1):59–82, 2005.

[19] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3), 1995.

[20] Mark D. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 4:352–357, 1984.

Appendix: program examples

The Fibonacci and Histogram programs (adapted from `http://www.cs.joensuu.fi/~saja/var_roles/`) used as examples in several sections above.

```
import java.io.*;

public class Fibonacci {
    private int last_fib;   /*%%last_fib%%follower%%*/
    private int fib;        //%%fib%%gatherer%%
    private int temp;       //%%temp%%temporary%%
    private int number;     //%%number%%fixed value%%
    private int i;          //%%i%%stepper%%
    /**
     * Constructor for objects of class Fibonacci
     */
    public Fibonacci() {
        last_fib = 1;
        fib = 1; /*another comment*/ number = getNumber();
        if (number <= 2) {
            System.out.println("The first and
                                  second numbers are 1.");
        } else {
            System.out.println("Value 1 is: 1");
            System.out.println("Value 2 is: 1");
            for (i = 3; i <= number; i++) {
                temp = last_fib;
                last_fib = fib;
                fib = fib + temp;
                System.out.println("Value " + i +
                                     " is: " + fib);
            }
        }
    }

    /**
     * Read number of integers in sequence from terminal
     * @return  The number of values to be in the sequence
     */
    private int getNumber() {
        BufferedReader in = new BufferedReader(new
           InputStreamReader(System.in));
        int n = 0;
        while (n < 1) {
            String tempN = "";
            System.out.print("Enter number in sequence: ");
            try {
                tempN = in.readLine();}
            catch (IOException e) {
                System.out.println(e.getMessage());
            }
            Integer tempInt = new Integer(tempN);
            n = tempInt.intValue();
        }
        return n;
    }
}
```

```
import java.io.*;

public class Histogram {
    //Draw a histogram

    private final int longest = 40;  //Longest bar
    private float[] amount = new float[12];
    //Data for drawing%%amount%%fixed value%%
    private float max;
    //Maximum data element %%max%%most wanted holder%%
    private int month;
    //Current month %%month%%stepper%%
    private int i; //%%i%%stepper%%

    public Histogram() {
        max = 0;
        for (month = 1; month <= 12; month ++) {
            System.out.println("Enter amount for month "
                                   + month + ": ");
            amount[month - 1] = getInput();
            if (max < amount[month - 1]) {
               max = amount[month - 1];
            }
        }
        System.out.println();
        for (month = 1; month <= 12; month++) {
            System.out.print(month + ": ");
            if (month < 10) System.out.print(" ");
            for (i = 0; i <= amount[month-1]
                  / max * longest; i++)
              System.out.print('*');
            System.out.println();
        }
    }

    private float getInput(){
        float returnFloat = 0;
        BufferedReader in = new BufferedReader(new
  InputStreamReader(System.in));
        try {
            String inputString = in.readLine();
            Float tempFloat = new Float(inputString);
            returnFloat = tempFloat.floatValue();
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
        return returnFloat;
    }
}
```