Universiteit Leiden
The Netherlands

Opleiding Informatica

JShowFlow

A Control Flow Graph Generator for Java Code

David de Muinck Keizer

Supervisors:
Marcello Bonsangue and Frank de Boer

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

18/7/2023

Abstract

From the source code of a program, it can be difficult to fully understand the program's behaviour. This is important to prevent errors, crashes or other unexpected behaviour of the program during runtime. One way to get a better understanding of the behaviour of the source code is by drawing a control flow graph of the code. Such a graph shows all possible paths that might be traversed if the program is executed. It can be time consuming and error-prone to draw a control flow graph by hand, especially when the source code is very large. In this bachelor thesis, we describe JShowFlow, a program that is capable of generating control flow graphs for Java code. This thesis describes a way of turning Java code into graphs and it shows examples of control flow graphs that can be generated using Java code.

Contents

Chapter 1

Introduction

This bachelor thesis is part of the bachelor computer science at the Leiden Institute of Advanced Computer Science at Leiden University and was supervised by Marcello Bonsangue.

When you are developing software, it is almost impossible to write the perfect code without coming across tons of errors first. Most of these errors are detected by the compiler, so that you can solve these errors before executing the program. But even if the compiler cannot detect errors, it is still possible that errors occur during runtime. They can cause the program to crash or they can be revealed by unexpected behaviour of the program. In both cases, unresolved errors can cause dangerous situations. That is why it is important to know how the program will behave in different situations. Such information can be obtained without running the program, just by looking at the source code. This is called static analysis, static program analysis or static code analysis [Wic95]. One way of doing static analysis is by creating a control flow graph. By creating such a graph, one can gain insight into all paths of a program that can be travelled through when the program is executed. This is not only useful to understand the behaviour of your own program, it can also be used to get a better understanding of someone else's code. But drawing such a graph can be time consuming and error-prone, especially for large programs. That is why tools for drawing these graphs can be very useful. There are several tools that can do this, but unfortunately they all have some disadvantages. Some of them require another program to be installed (such as Eclipse), whereas others are not clear enough or contain mistakes. The purpose of this bachelor thesis is to examine how to create an easy to use tool for drawing control flow graphs, that does not have one of these disadvantages. The tool will be designed for Java source code and will be called JShowFlow. It can also be used for programming languages that are somehow similar (such as C++), but since there are some differences between the languages the graphs might not be fully correct.

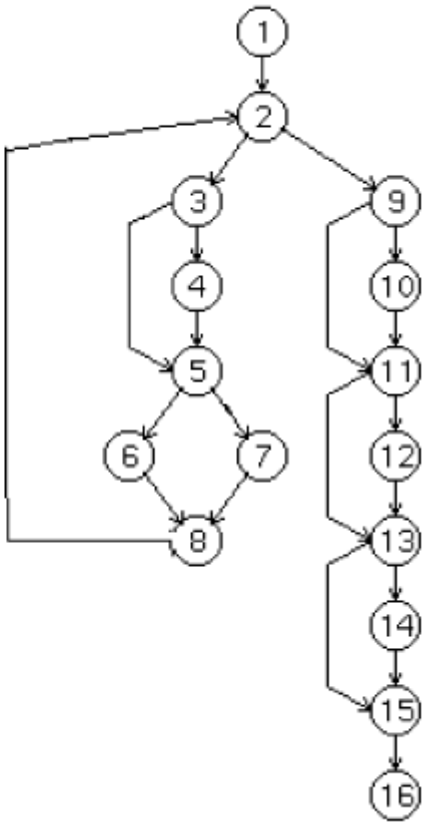
1.1 Control flow graph definition

A basic block is a linear sequence of program instructions having one entry point (the first instruction executed) and one exit point (the last instruction executed). A control flow graph is a directed graph in which the nodes

represent basic blocks and the edges represent control flow paths. Both definitions are described in [All70]. So a control flow graph shows all paths of the program that might be travelled across during the execution of the program.

For example, a control flow graph might look like this:

```
stcode getlist(char *lin, int *i, stcode *status)
/*  1 */ {
/*  1 */   int num, done;
/*  1 */   line2 = 0;
/*  1 */   nlines = 0;
/*  1 */   done = (getone(lin,i,&num,status)!=OK);
/*  2 */   while (!done)
/*  3 */   {
/*  3 */     line1 = line2;
/*  3 */     line2 = num;
/*  3 */     nlines++;
/*  3 */     if (lin[*i]==SEMICOL)
/*  4 */       curln = num;
/*  5 */     if ((lin[*i]==COMMA)||(lin[*i]==SEMICOL))
/*  6 */     {
/*  6 */       *i = *i + 1;
/*  6 */       done = (getone(lin,i,&num,status)!=OK);
/*  6 */     }
/*  7 */      else
/*  7 */        done = 1;
/*  8 */   }
/*  9 */   nlines = min(nlines,2);
/*  9 */   if (nlines == 0)
/* 10 */     line2 = curln;
/* 11 */   if (nlines <= 1)
/* 12 */     line1 = line2;
/* 13 */   if (*status != ERR)
/* 14 */     *status = OK;
/* 15 */   return(*status);
/* 16 */ }
```



In this example the edges do not just point to the next node that represents an instruction, there are also arrows visible that go back to an older instruction or arrows that skip instructions. Those edges represent the end of a `while` loop and an `if` statement (in which the `if` condition evaluates to false), respectively. By doing so, this combination of edges and nodes represents all possible execution paths, so that we know how that piece of code will behave. As we can see, the instructions are represented with just a number inside each node. While this decreases the size of the graph, it also reduces readability. That is why we decided to create a graph in which the nodes contain the text of the instruction, instead of just a number.

1.2 Related work

Drawing a control flow graph by hand can be time consuming, especially when the source code contains many instructions. That is why it is useful to have a tool for generating control flow graphs. There are already several other control flow graph generators, but each of them have some shortcomings.

1.2.1 Dr. Garbage Sourcecode Visualizer

This is an eclipse plugin, so it requires Eclipse to be installed. When using this tool, a control flow graph will appear in a window next to the Java source code in Eclipse. The nodes do not contain any text, only a color. In order to understand which node represent which instruction, one should hover the cursor over a node to let the tool tip appear that contains the information about the node. Although this tool is also capable of creating a control flow graph, it requires a lot more effort to run it and to understand it compared to JShowFlow.

For more information see: `http://www.drgarbage.com/sourcecode-visualizer/`

1.2.2 Control Flow Graph Factory

This tool is also an Eclipse plugin and can be used as an extension for the Sourcecode Visualizer mentioned above, but that it can also be used without the Sourcecode Visualizer. It can create control flow graphs and also edit these graphs. The graphs can be exported as images or as XML or DOT. These nodes do contain text but they can only contain a few characters, whereof some are occupied by a number that does not appear to be very useful. Although the readability is an improvement compared to the Sourcecode Visualizer, it still leaves a lot to be desired.

For more information see: `http://www.drgarbage.com/control-flow-graph-factory/`

1.2.3 Eclipse Control Flow Graph Generator

Created by Aldi Alimucaj, a student of the Konstanz University of Applied Sciences. This is also a plugin for Eclipse and the nodes in the generated graph do not always contain the text of the instruction. For example, it can contain the text "EXPR" instead of "x = 3;". The graph that appears next to the Java source code in Eclipse is not always correct, especially when dealing with `try-catch-finally` statements.

For more information see: `http://eclipsefcg.sourceforge.net/`

1.3 Thesis Overview

Chapter 2 explains the requirements of a control flow graph generator. Chapter 3 shows how to run JShowFlow. Chapter 4 shows several control flow graphs that were created using JShowFlow. In chapter 5 we discuss the things could be implemented to further improve it. Chapter 6 draws some conclusions.

Chapter 2

Method

All of the tools mentioned in Related Work (in chapter 1) are plugins for Eclipse. While Eclipse is a widely used tool for developing Java programs, it is not used by everyone. There are several other programs, such as NetBeans, that can be used as well. For developers that do not use Eclipse, it can be very inconvenient to be forced to install Eclipse, only to use a plugin to create a control flow graph. That is why JShowFlow will not be associated with any source code editor. Using C++, we are planning on building a program that analyzes a .java file, just by looking at the Java source code in the file. Then we will convert that Java source code into a data structure that can be easily read and moved through, so that unnecessarily difficult operations to find particular nodes will not have to be performed. We will call this data structure the Code Diagram. Next, we will create the control flow graph by looping through the code diagram while adding special connections. Then we will create a C++ function that converts the generated control flow graph to a code that can be used to draw the graph: DOT language. And finally, the control flow graph can be drawn using the generated DOT code. The result can be a png file or a svg file that can be viewed with an internet browser.

2.1 Reading the Java source code

We will assume that this Java code is a code without syntactic errors, such that we do not have to recognize errors in the code. For instance, a misplaced bracket in the input code could crash the program, but with the assumption that the input code is correct there is no need to check for such an error. The Java code should be in a file called input.java, which is then read by JShowFlow. The reading of the code itself is straightforward: the program uses a `while` loop to read the input code line by line. In each cycle, the current line is added to the code diagram in way that is explained in the next section. However, if we are dealing with a line we should ignore, such as a comment or a whiteline, it will of course not be added to the Code Diagram.

2.2 Building the Code Diagram

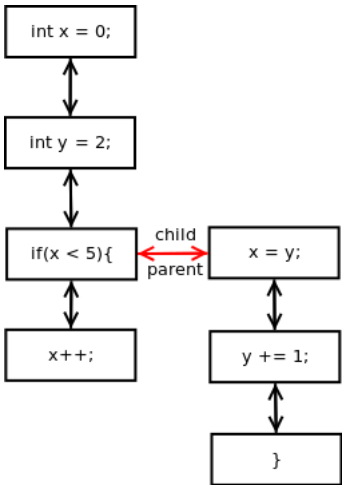
As mentioned above, the Java source file is read line by line. Each line is a string that will be converted to a C++ struct called *Line*. That struct will then be "connected" to the rest of the structs that represents earlier lines that were already read, if any.

Example 2.1

Consider the following code:

```
int x = 0;
int y = 2;
if(x < 5){
    x = y;
    y += 1;
}
x++;
```

In this code, the struct containing the line "int y = 2;" should be connected to the struct containing the line "int x = 0;". Then "if(x < 5){" should be connected to "int y = 2;" the same way. But since the `if` is a block statement, it should have a special connection to the contents of the block statement. We will call that connection a child-parent connection, in which the `if` statement represents the parent of x=y;. All of the other connection are next-previous connections, including the connection between "x = y;" and "y +=1;". By using this child-parent connection, we can create a next-prev connection between the `if` statement and "x++;". When putting all of this into a diagram, in which the structs are represented as blocks and connections as edges, the diagram will look like this:



We call this diagram the code diagram, since it represents the structure of the code in a diagram.

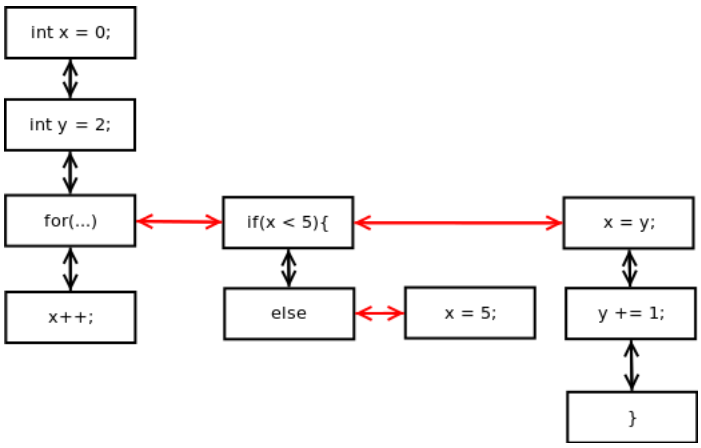
Example 2.2

An `if` statement can also have an `else` statement and block statements can also appear inside other block statements. When using this code as input code:

```
int x = 0;
int y = 2;
```

```
for(int i = 0; i < 10; i++)
    if(x < 5){
        x = y;
        y += 1;
    }
    else
        x = 5;
x++;
```

the code diagram should look like this:



The red, horizontal lines represent the child-parent connections. The vertical edges represent the next-previous connections. It is not difficult to recognize the tree structure in the code diagram. The only difference between a tree and this diagram is that in a tree, parent nodes have connections to all of their children and no connections to their siblings, whereas in this code diagram a child-parent connection is created only with the first child and siblings are connected with a next-previous connection. The reason to implement the code diagram this way is to eliminate the need for including a list of all child structs in a parent struct. By doing so, only one child-parent connection has to be made per parent node. This also makes it more clear that the order of child nodes is important, since they are executed in that order.

2.2.1 Properties of each struct

From the examples above, it is clear that each node has a *next*, a *previous*, a *child* and a *parent* pointer. It depends on the kind of the node which of those pointers are in use.

When converting a line of code into a struct, the original line should be preserved. Otherwise we end up with a code diagram and a control flow graph without text, making it hard to recognize the original code in these generated diagrams. So a string called *text*, which contains the line of code the structs represents, will be included in each struct.

When looking at the input Java code, we can see that there are several types of lines. There are lines ending with a semicolon and several block statements that behave differently. We can determine the type of each line just by looking at it, i.e. performing a string comparison operation. When creating the control flow graph later,

the code diagram is looped through many times in several ways. It would be a real obstacle in performance to perform a string comparison operation each time. Instead, it would be better to include an int called *type* in each struct, that holds a number that represents the type of the line. By doing so, we only have to determine the type of each line once. When creating the control flow graph and looping through the code diagram, only an int comparison is needed to understand what type of line we are dealing with. When comparing two strings, each character in the strings is checked until a difference is found [cpl], while an int comparison only requires a single check. So using int comparisons instead of string comparisons requires less effort.

In the example code above, we could assign type = *1* to the structs of "int x = 0;", "int y = 2;", "x = y;", "y += 1;", "x = 5;" and "x++;" making the number 1 represent each line that ends with a semicolon. "for(int i = 0; i < 10; i++)" could be of type *2*, "if(x < 5){" of type *3*, "}" of type *4* and "else" of type *5*. Each struct also has an *ID*, which is an int, to identify it. Displaying this ID in each node in the control flow graph can be turned on or off.

2.2.2 Extra linked list for other properties

Some information that is needed for creating the code diagram is not covered by the properties of the structs. An example is the property of block statements to be able to contain only one line of code or multiple lines. For instance, an `if` statement without brackets can only contain one instruction. We call a block statement that can only contain one instruction a *single line statement* and a block statement that can contain multiple lines a *multiline statement*.

It is possible that the single instruction of the `if` statement is another block statement that consists of multiple lines, such as a `while` loop. If that is the case, it is important to keep in mind that the `if` statement cannot contain more lines when we are done creating the `while` loop. Otherwise we might add the line after the `while` loop to the `if` statement, while those lines should be added after the `if` statement.

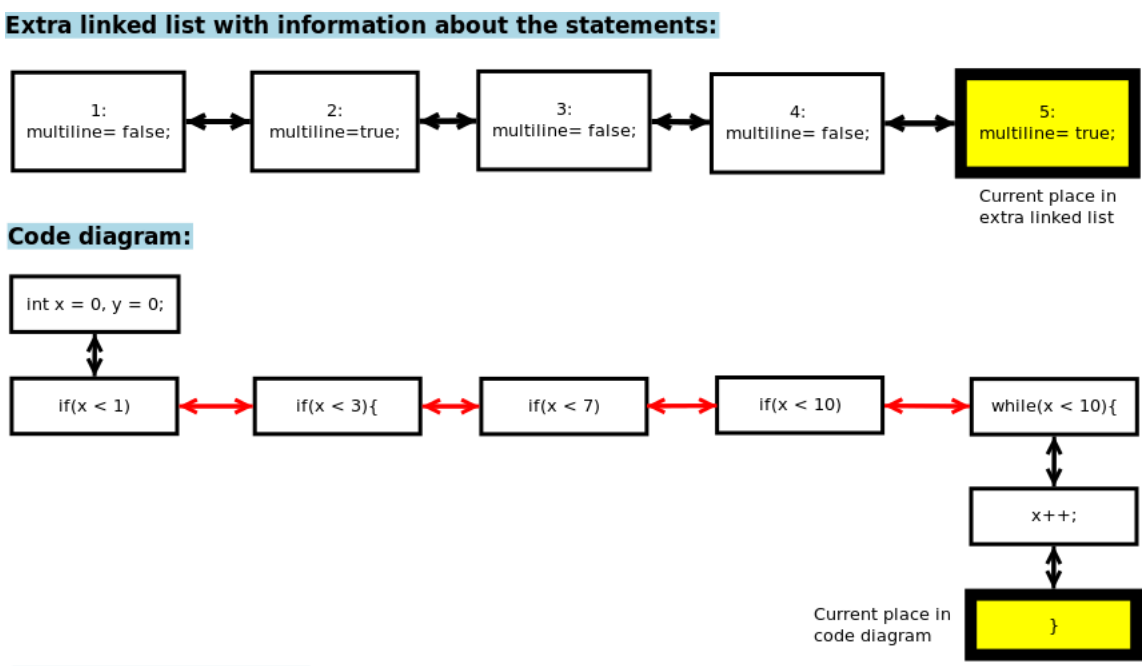
Example

In this example, multiple block statements are nested. Some of the statements are single line statements, while others can contain multiple lines.

```
1  int x = 0, y = 0;
2  if(x < 1)                  //1: single line
3    if(x < 3){               //2: multi line
4      if(x < 7)              //3: single line
5        if(x < 10)           //4: single line
6          while(x < 10){     //5: multi line
7            x++;
8          }                  //Belongs to statement 5
9      y++;                   //Belongs to statement 2
10   }                        //Belongs to statement 2
11 x = y;
```

When we are done processing the bracket that belongs to statement 5 (the `while` loop), the next line is "y++;". In this code, we can see that "y++;" belongs to the second `if` statement. But how do we know this when constructing the code diagram? How do we know to what struct the struct of "y++;" should be connected to? This can be determined by using an extra linked list consisting of another type of structs that contains the information of each statement, including whether a statement is a single line or multiline statement.

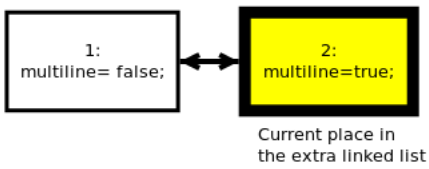
In the above example, assume we are at the point the bracket of statement 5 has just been connected to the first child of statement 5 and now we should determine the location for the next struct.



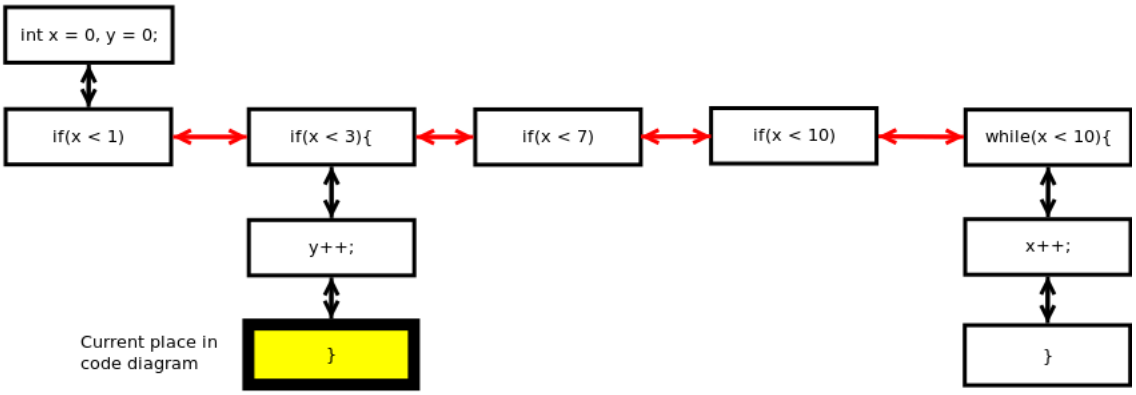
At the top of this image, the extra linked list is displayed. Each number represents the number of the statement that is right below it in the code diagram. Each struct in the extra linked list was created and added to the extra linked list when its corresponding statement in the code diagram was created. Since we just placed a bracket (in the `while` loop) that indicates the end of a statement, we know that we should go back one step in the extra linked list, such that the struct with the number 4 is the current struct in that list. Since multiline is set to false in this struct, this means that statement 4 in the code diagram cannot contain another line of code. Statement 4 is full, so we go back another step in the extra linked list such that the current struct in that list is 3. Again, multiline is set to false so we go back one more step. Then the current struct in the extra linked list is the struct that represents statement 2 in the code diagram. Here multiline is set to true, meaning that it can contain another statement. So we can connect "y++;" to statement 2. The line after that is a bracket, which can be connected to "y++;".

Now the current situation looks like this:

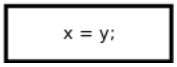
Extra linked list with information about the statements:



Code diagram:



Determine location for:



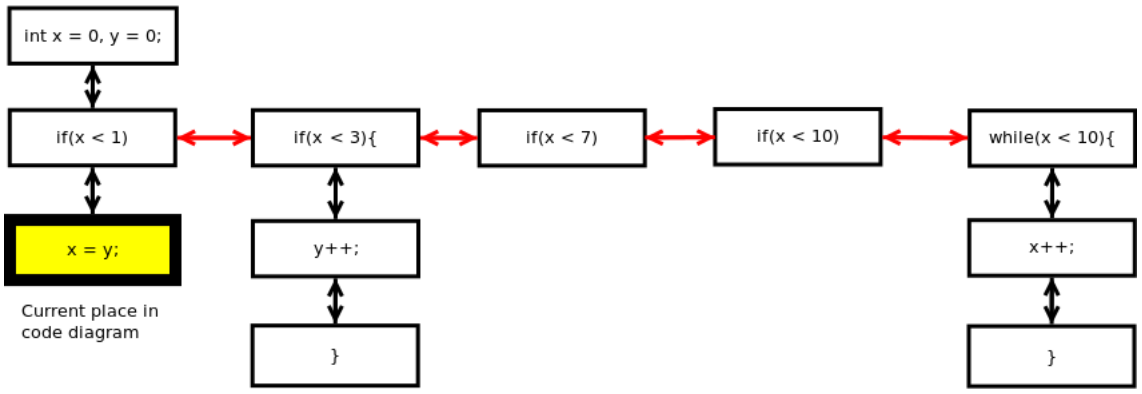
The extra linked list now only contains two structs, since we are done creating the other statements, meaning they do not have to be represented in the extra linked list anymore. Since the bracket, which is the current location in the code diagram, indicated the end of statement 2, we need to go back one step in the extra linked list. Statement 1 is single line, so "x = y;" cannot be put in this `if` statement. So we should go back another step in the extra linked list. But statement 1 is the first item in the extra linked list, so there is no other item to go back to. But we can remove statement 1 from the linked list, such that there are no more items in the list. If there is no item in the list, we do not have to check if "x = y;" can be placed. We can just connect it to the current item (which is statement 1) as next item.

Then the current situation looks like this:

Extra linked list with information about the statements:

(No items)

Code diagram:



Determine location for:

(No item)

Right now there is no more item to place into the code diagram. This means that the construction of the code diagram is finished.

Else statements

But the `bool` multiline is not the only property we should specify to determine where the next line belongs to. Imagine that some of the `if` statements would have an `else` block:

```
int x = 0, y = 0;
if(x < 1)                //1: single line
  if(x < 3){             //2: multiline
    if(x < 7)            //3: single line
      if(x < 10)         //4: single line
        while(x < 10){   //5: multiline
          x++;
        }                //Belongs to statement 5
      else               //else of statement 4, must be inside statement 3
        x--;
    else                 //else of statement 3, must be inside statement 2
      y--;
    y++;                 //Belongs to statement 2
  }                      //Belongs to statement 2
x = y;
```

If we would only check whether a statement is multiline or not, we would conclude that the `else` of statement 4 should be inside statement 2, since that is the first statement that is a multiline statement and can contain another statement. Obviously, this is not correct. To solve this, we use another boolean called *elseused* that tells us if a statement already has an `else`. It is set to false by default and it is also part of the extra linked list, just like the boolean *multiline*. When checking where the first `else` belongs to, we do not only check `if`

statements are multiline statements, but also if their elseused is still false. If so, we can connect the `else` as next item to that statement, which is statement 4 in this case. Now we should set elseused to true, such that the second `else` will not be connected to the first `else`. By doing so, we could also add an infinity amount of `else if` statements as next item to an `if` statement until elseused is set to true.

2.3 Building the control flow graph using the Code Diagram

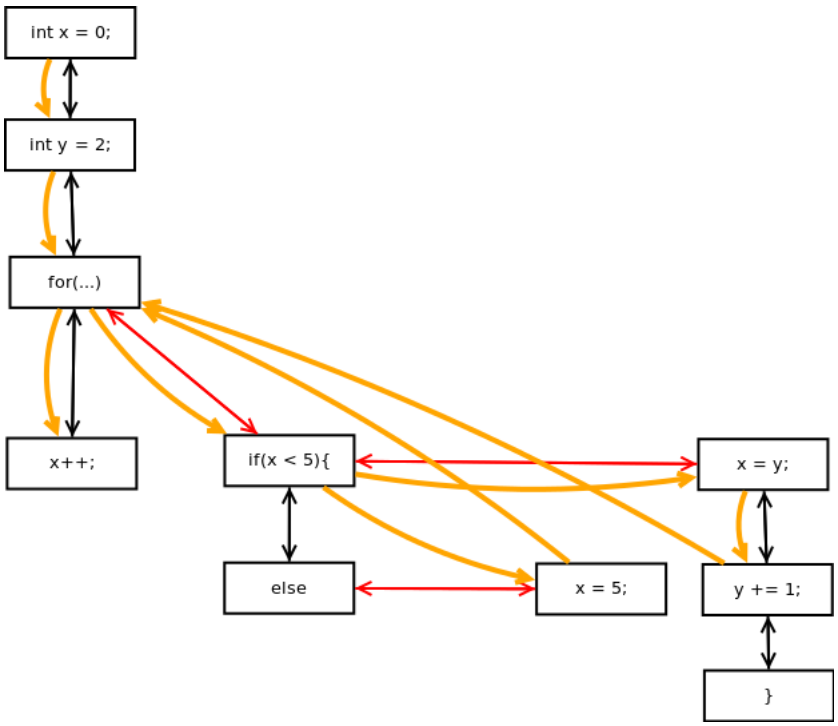
In the previous section, we explained how to turn code into a code diagram. The code diagram consists of several structs, one for each line of code. Each struct contains an ID, the line of code, the type of that line and pointers to connect with others structs.

By using the code diagram, a control flow graph can be created. We can add different types of arrows to the code diagram, that represent the control flow graph. This way, the control flow graph is included in the code diagram. By doing so, we can later write a function that only converts the arrows that belong to the control flow graph into DOT code. The result will be a control flow graph represented by DOT code. The section below will explain more about DOT code. But first, we need to create the control flow graph. It can be done as follows. We start at the first struct in the code diagram. When its next pointer is not NULL, i.e. another struct follows the current struct, an arrow should be drawn to that next struct. Now we go to the next struct in the code diagram and do the same. We repeat this until there is no next struct anymore. While doing this, it is possible that we come across a block statement. In that case, an arrow should be drawn to the first child and then we repeat the process for each child of the block statement recursively. This is how the code diagram is looped through.

However, it is also possible that other arrows than next or child arrows should be drawn. For example, the last child inside a `for` or `while` loop should also have an arrow back to the beginning of that loop, to show that is a loop. To create that arrow we should determine whether the current struct is the last struct inside a loop. If so, draw an arrow to that loop. Another example is the `break` statement, that can break outside a loop or `switch`. In that case we should make the `break` struct point to the struct that follows the loop or `switch`. There are a lot more examples of arrows that are not just the next struct or child struct. In chapter 4 more examples are shown.

Example 2.3

When using the code and code diagram of example 2.2, the code diagram with the arrows of the control flow graph looks like this:



Each curved, orange line represents an arrow that belongs to the control flow graph. Note that there is no arrow to the `else` struct. Instead, two arrows leave the `if` statement: one arrow to its own children and one to the children of the `else` struct. By doing so, we do not have to include the `else` statement itself in the control flow graph. The bracket at the end of the `if` statement is also not included in the control flow graph, since it will not add any value to the control flow graph. The last child of the `if` statement and the last child of the `else` statement both point back to the beginning of the `for` loop, because they are both also the last children in the `for` loop. When the `for` loop has ended, "x++;" will be executed. That is why there is a next arrow from the `for` loop to x++;.

2.4 DOT code

With this new diagram, we can eliminate the arrows that do not belong to the control flow graph. By converting only the control flow graph related arrows to DOT code, this is in fact what will happen.

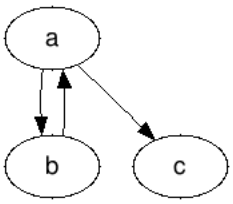
2.4.1 What is the DOT language?

The DOT language can used to describe graphs [Gra]. Edges are created using two horizontal bars (in non-directed graphs) or a horizontal bar followed by a greater than sign (in directed graphs). A->B means that a node with the label "A" points to a node with the label "B". Nodes can be specified before being used, but that is not necessary.

Example 2.4

```
digraph example {
    a -> b;
    a -> c;
    b -> a;
}
```

This DOT code will generate the following graph:

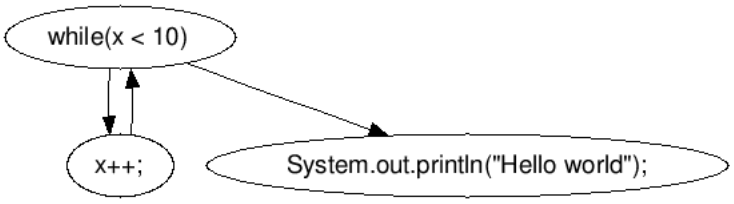


Example 2.5

When defining nodes before they are used, we can add labels the nodes to display different texts inside the nodes. By adding a label for each node, the could could look like this:

```
digraph example {
    a [label="while(x < 10)"];
    b [label="x++;"];
    c [label="System.out.println(\"Hello world\");"];
    a -> b;
    a -> c;
    b -> a;
}
```

This will create the following graph:



2.4.2 Generate the control flow graph with DOT code

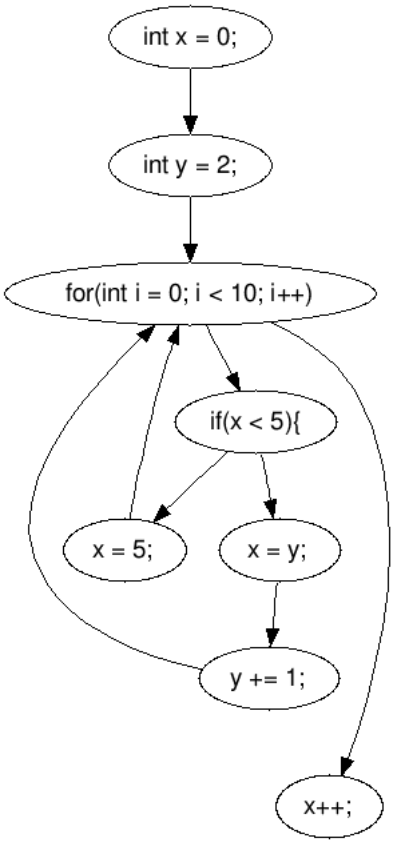
From example 2.5, it is clear that we can create a diagram of code by using labels. In that example, a single character is used to assign a label to. In our control flow graph, we will use a number to assign a label to, because numbers can be increased. Since each struct has its own ID, we can use that int to assign a label and arrows to.

Example 2.6

Using the new diagram from example 2.3, we can write a function that converts the arrows from that diagram to DOT code. Of course, we only convert the arrows that represent the control flow graph. The result will look like this:

```
digraph example {
  1 [label="int x = 0;"];
  2 [label="int y = 2;"];
  3 [label="for(int i = 0; i < 10; i++)"];
  4 [label="if(x < 5){"];
  5 [label="x = y;"];
  6 [label="y += 1;"];
  9 [label="x = 5;"];
  10 [label="x++;"];
  1 -> 2;
  2 -> 3;
  3 -> 4;
  4 -> 5;
  4 -> 9;
  5 -> 6;
  6 -> 3;
  9 -> 3;
  3 -> 10;
}
```

When creating an image using this DOT code, the result looks like this:



In this diagram, we can see two arrows leave the `for` loop. But it is unclear what both arrows mean and when they are used. Fortunately, we can also add labels to edges to make it more clear what is going on. It is also possible to edit the layout of the graph by adding colors or edit the style of the arrow (e.g. "bold", which results in arrows that are clearer to see).

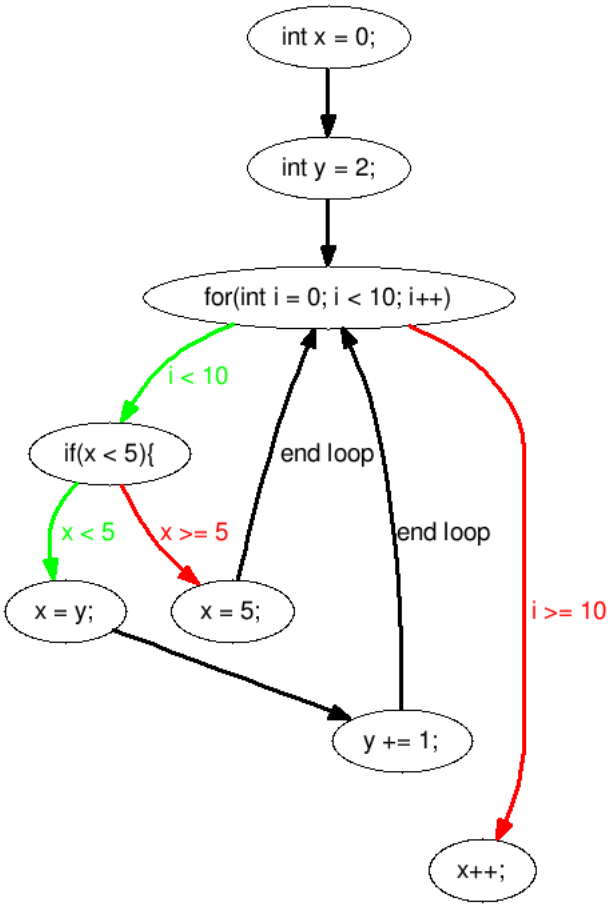
Example 2.7

By adding labels, colors and certain style features, the DOT code could look like this:

```
digraph example {
  1 [label="int x = 0;"];
  2 [label="int y = 2;"];
  3 [label="for(int i = 0; i < 10; i++)"];
  4 [label="if(x < 5){"];
  5 [label="x = y;"];
  6 [label="y += 1;"];
  9 [label="x = 5;"];
  10 [label="x++;"];
  1 -> 2 [style="bold"];
  2 -> 3 [style="bold"];
  3 -> 4 [label="i < 10" color="green" fontcolor="green" style="bold"];
  4 -> 5 [label="i < 5" color="green" fontcolor="green" style="bold"];
  4 -> 9 [label="i >= 5" color="red" fontcolor="red" style="bold"];
  5 -> 6 [style="bold"];
  6 -> 3 [label="end loop" style="bold"];
  9 -> 3 [label="end loop" style="bold"];
  3 -> 10 [label="i >= 10" color="red" fontcolor="red" style="bold"];
```

```
19 }
```

When converting this DOT code to an image, the image looks like this:



Now it is much clearer to read. Note that for both the `for` loop and `if` statement, the condition of the statement was used as label. When x $<$ 5 is false, a red arrow leaves the `if` statement with the label "x $>=$ 5". Right now it easy to see that the opposite of x $<$ 5 is x $>=$ 5, but in general the negated condition is more difficult to determine. Instead, we will strike-out on the label through the `if` statement's condition to display next to the false arrow in the final version of the control flow graph generator.

2.5 Classes and methods

We explained earlier how the code diagram and control flow graph were constructed using a simple input Java code. The examples we showed only contained expressions and statements, while Java code is usually more complicated than that. A Java file can also contain classes and methods.

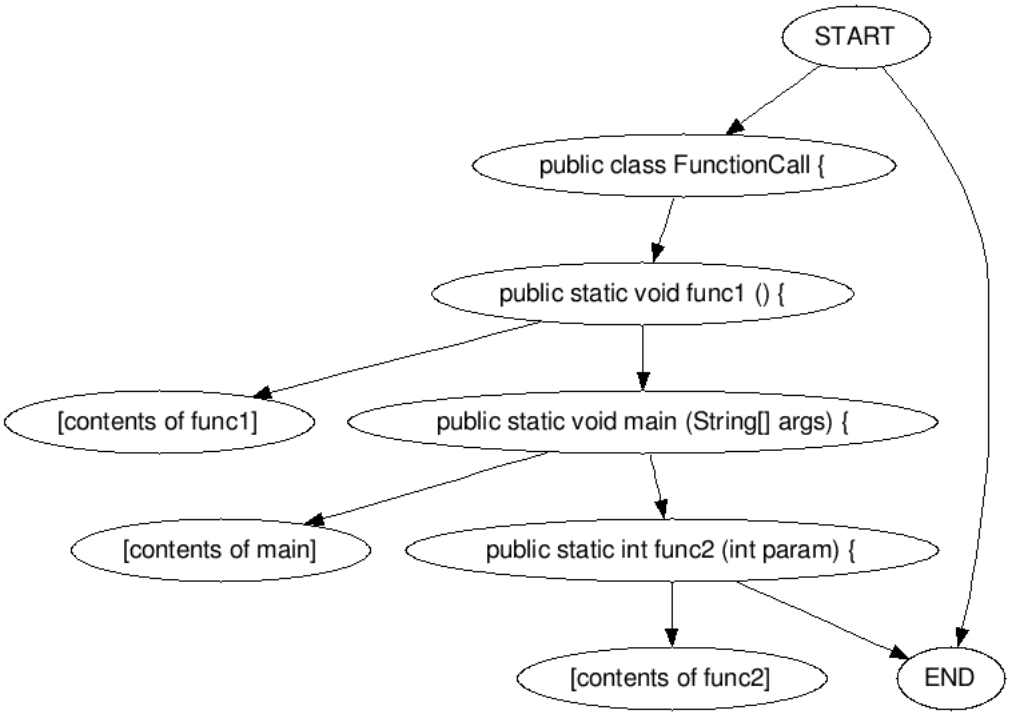
2.5.1 Including classes and methods in the control flow graph

Example 2.8

A Java file with a class and methods:

```
1  public class FunctionCall {
2
3    public static void func1 () {
4      [contents of func1]
5    }
6
7    public static void main (String[] args) {
8      [contents of main]
9    }
10
11   public static int func2 (int param) {
12     [contents of func2]
13   }
14 }
```

We could turn this code into one control flow graph, so that it will look like this:

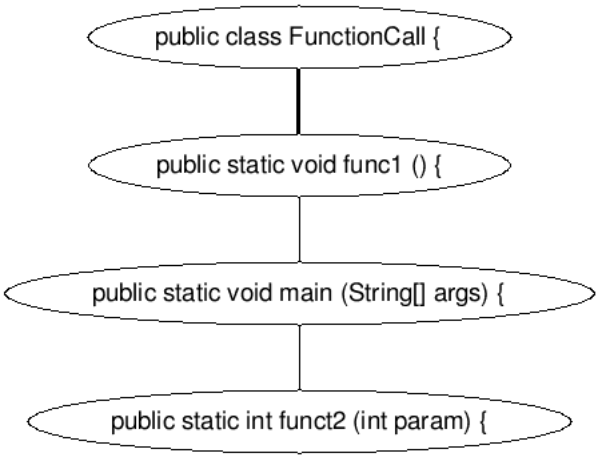


However, the purpose of the graph is to make things clearer, while this graph seems to make things more complex, since there are too many things to look at. Especially when each method has a complicated code, the screen will contain too much information at once. So instead, it would be better to group certain parts to make it easier to read. We decided that there should be one view with all classes and methods, without displaying the contents of the methods. Each method should have its own control flow graph of the contents of that method, that we can view by clicking on that particular method in the view. This makes the view some sort of class

diagram or table of contents. To make it more clear what we mean, here is another example to illustrate the idea.

Example 2.9

With the code from example 2.8, we can create this class diagram:



Now when we click on the node with the text "public static void func1 () ", we should go to another view that displays the control flow graph of the contents of func1. That control flow graph is then generated. So instead of making one large, unclear control flow graph, there will be several control flow graphs, one for each method, that can be shown by clicking on a method in the class diagram. Using DOT code, a node can be made clickable by adding an attribute called *url*. It can be used by adding **url="*location of the control flow graph*"** inside the square brackets in the node specification before or after the *label* attribute. If there are multiple classes in the Java input file, the classes with their methods can be displayed next to or below each other.

2.5.2 Calls from one method to another

By giving each method its own control flow graph, there will be no useless information of other methods on the same screen while we look at the control flow graph of one method. But it is possible that, in some point at the a method, a call will be made to another method. In that case, we will just display that other method as a node in the control flow graph. That node should be clickable, such that a click on that node will show the control flow graph of that method.

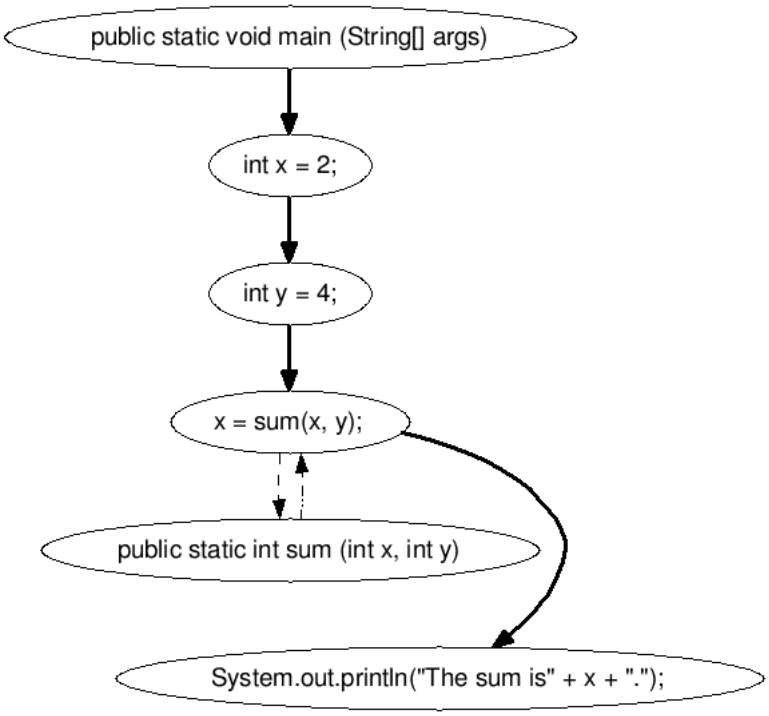
Example 2.10

Consider this Java code:

```
1 public class FunctionCall {
2
3   public static int sum (int x, int y) {
4     return x+y;
5   }
6
7   public static void main (String[] args) {
8     int x = 2;
```

```
9       int y = 4;
10      x = sum(x, y);
11    }
12  }
```

When the control flow graph of the main method is viewed, it could look like this:



There is a dashed line from "x = sum(x, y);" to a node that represents the method sum. When we click on that node, the control flow graph of sum is displayed. Note that there is also a dashed line from that node back to "x = sum(x, y);", since the method sum has a return. If sum would not have a return node (i.e. it would be a void method), there should not be an arrow back. So there must be a function to verify that sum has a return node.

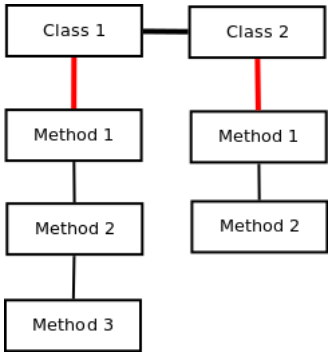
There must also be a way of understanding that "x = sum(x, y);" calls the method "public static int sum (int x, int y)";. This requires a couple things:

- There must be a list with all methods and their classes
- There must be a way of recognizing calls to other methods in a line of code

List with methods

When "x = sum(x, y);" is executed, there should be a way to understand that *sum* represents the struct of the line "public static int sum (int x, int y)". For that we will compare sum to a list of all methods that are in the input Java file, such that we can choose the right one. Constructing this list can be done during the process of turning the Java input code into the code diagram, but that can cause problems. In Java it is possible that a method call occurs to a method that appears below the current method. Imagine that sum would be below the main method, then "public static int sum (int x, int y)" is not in the list of methods yet when "x = sum(x, y);" is being processed. To overcome this error, a list of all methods should be constructed prior to converting the

Java code into the code diagram. This means that the Java input file should be opened and processed multiple times. But since we also need a class diagram, we can use that diagram as method list. For constructing the class diagram, JShowFlow will use the same type of structs that are used for creating the code diagram. By doing so, it will be clear from the structure to which class a method belongs to. This is important to know in cases where different classes have methods with the same name.



Recognize method calls in a line of code

For recognizing a method call in a string, JShowFlow will look for brackets since each method call is followed by brackets. Even if a method without parameters is called, two empty brackets will follow: x = method();. So for finding a method call, JShowFlow looks for the first '('-bracket and then find the word that is in front of that bracket. If that word is not equal to the string "if", "for", "while", "switch", "catch", etc., then it must be a call to another method. This word is then used to compare it with all the methods in the method list, which is the class diagram as we explained above. When a method is called, the compiler always starts to look for a method with that name in the current class, so that is also what we should do. The struct containing the class of the current method is determined, then the struct for the equivalent class in the class diagram is determined. Using that struct, its children, which are all methods, will be compared to the word that was found. If a match was found, a node that represents the found method will be created to include in the control flow graph. A special connection between the method calling struct and that node will then be created, represented by a dashed arrow in the control flow graph. If the found method has a return, a dashed arrow is also drawn back to the method calling node. If there was no match in the current class, the rest of the class diagram will be searched to find a match. The current class will be skipped during this search, since it has already been searched.

It is also possible that a line of code contains multiple method calls, for example: "x = sum(x, y) + sum(x, z) + avg(x, y);". In this case there should be nodes included in the diagram for both sum and avg. Each struct has a list of calls to other methods, which will include sum and avg in this case. The items in that list should be ordered and unique, such that sum will not appear twice in this case.

2.6 Viewing the graph

Each method has its own control flow graph, so for each method a file with DOT code should exist. For this we will create a directory called "methods" in which all of the DOT files will be saved with a filename like "method*[method ID]*.txt", in which *[method ID]* represents the ID of the method.

DOT files can be converted to several extensions in order to view the graph. One extension is .png, such that the graph will be a picture, which is what we want. But when using .png, nodes will not be clickable while being able to click on nodes is very important in the program. Another extension is .svg. By using .svg, we can generate a graph that looks very similar to a graph generated from DOT code and converted to .png, but with .svg clicking on nodes is possible. A .svg file can be opened in an internet browser.

As discussed earlier, the first thing the user should see is the class diagram. This class diagram will be saved in classdiagram.svg. When we open this file in an internet browser, we should be able to click on each method node. When we click on a method, we will be redirected to methods/method*[method ID]*.svg", which will show us the control flow graph of that method in an internet browser. Clicking on the first node in that view should make us return to the class diagram, so that we can view the control flow graphs of other methods.

Chapter 3

How to use JShowFlow

3.1 Preparations

JShowFlow uses C++11 instructions, such as regular expressions, and thus uses a compiler that can run C++11 code.
It is also important to have Graphviz installed for converting DOT code into an image or .svg file. It can be installed by entering this command in the terminal in Ubuntu:

```
sudo apt-get install graphviz
```

It can also be obtained from `http://www.graphviz.org/Download..php`.

3.2 Input Java code

The program reads Java code that is saved in the file *input.java* that must be in the same directory as *cfgg.cpp*. Further we assume that

- there is a blank line at the end of the input.java file.
- the Java code contains at least one class and one method.
- there are no mistakes in the input Java code, that is, it compiles without problems.

3.3 Compiling the C++ code

The file *cfgg.cpp* can be compiled by entering this command in your terminal:

```
./make
```

The command *g++ cfgg.cpp -std=c++11* works as well.

3.4 Executing the program

The program can be run the program by using this command after it compiled successfully:

```
./run
```

Or: *./a.out*

3.5 Viewing the graphs

When the program is executed, the file classdiagram.svg contains the class diagram. You can open this file in your browser to view the classes and methods that are in the *input.java* file. If we would like to view the control flow graph of a method, we can click on that method in the class diagram. Then we will be redirected to the control flow graph of that method. If we want to return to the class diagram, we can click on the first node in the control flow graph of that method.

Important when executing the program multiple times:
Always return to the class diagram before showing the control flow graph of a method. Refreshing the control flow graph of the current method does not always work, because the ID's might be changed. To show the most recent control flow graph of a method after executing the program, we should return to the class diagram, refresh and click on the method we want to view.

Chapter 4

Examples

This chapter shows some examples. The Java code below is used as input code for the examples of the class diagram and the `if` statements.

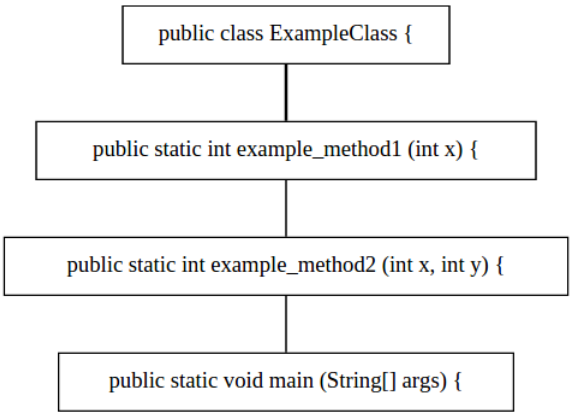
Input code for class diagram and if statements

```
public class ExampleClass {

  //If without else
  public static int example_method1 (int x) {
    if(x < 10){
      x++;
    }
    x = 2;
  }

  //If with else
  public static int example_method2 (int x, int y) {
    if(x < 10){
      x++;
    }
    else{
      y++;
    }
    x = 2;
  }

  //If with elseif and else
  public static void main (String[] args) {
    int x = 0;
    if(x < 10){
        x++;
    else if(x > 10){
        x--;
```

```
29      }
30      else if(x+2 > 10){
31          x--;
32      }
33      else{
34          System.out.println("OK!");
35      }
36      System.out.println("END");
37    }
38  }
```

4.1 Class Diagram

With the code above, the class diagram will look like this:

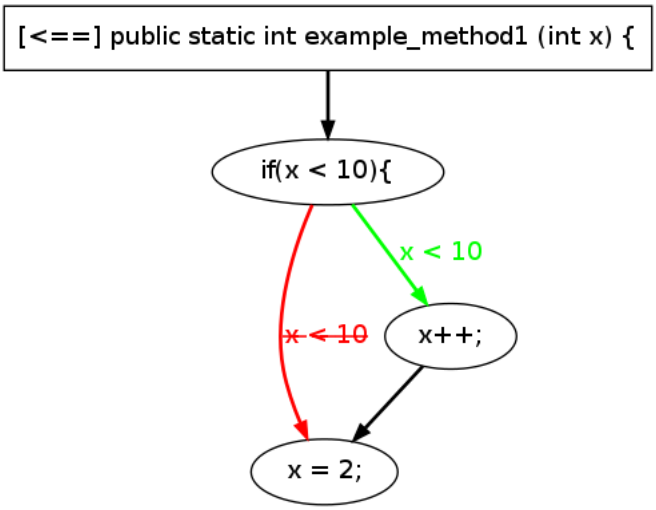


Now we can click on any block to open the control flow graph for the corresponding method.

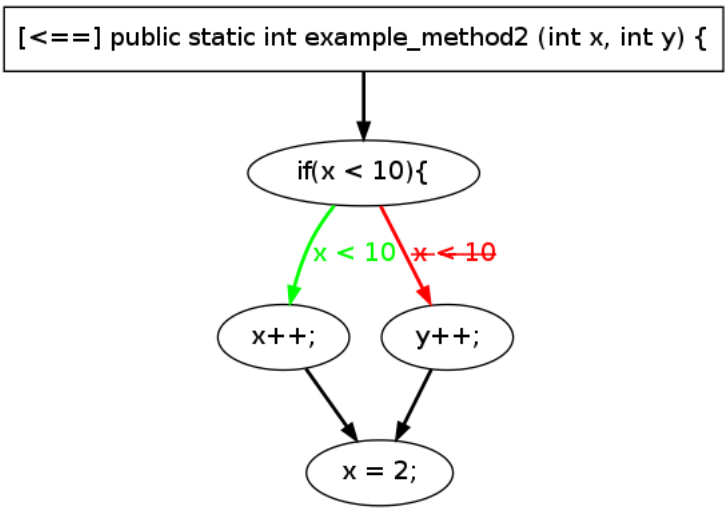
4.2 If statement

An `if` statement node has two leaving arrows: one to its children in case the condition evaluates to true and one to the children of the `else` block in case its condition evaluates to false. If there is no `else` block, the false arrow should point to the item after the `if` statement. Below are some examples that were generated using the code above. If the `if` statements would not have brackets, the results would look the same.

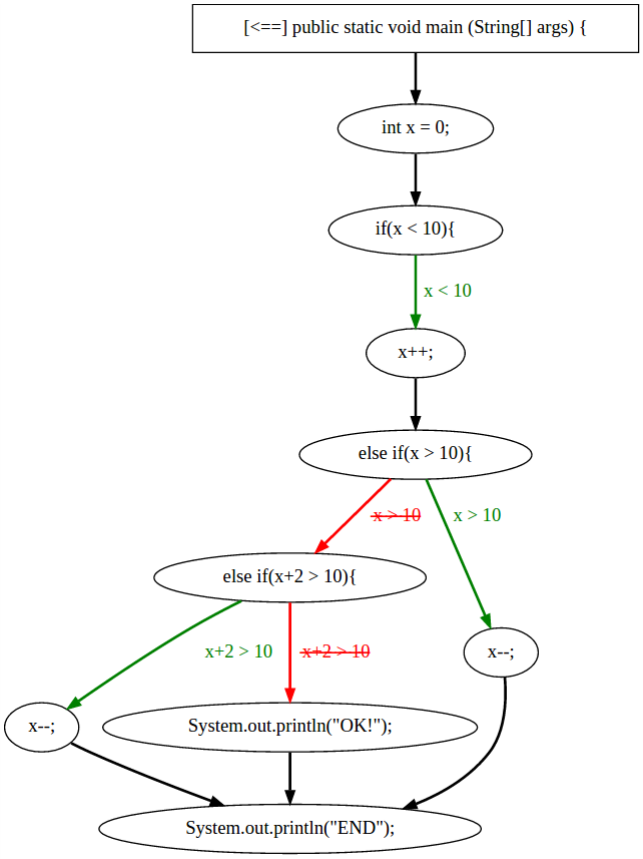
4.2.1 Without else



4.2.2 With else



4.2.3 With else if and else

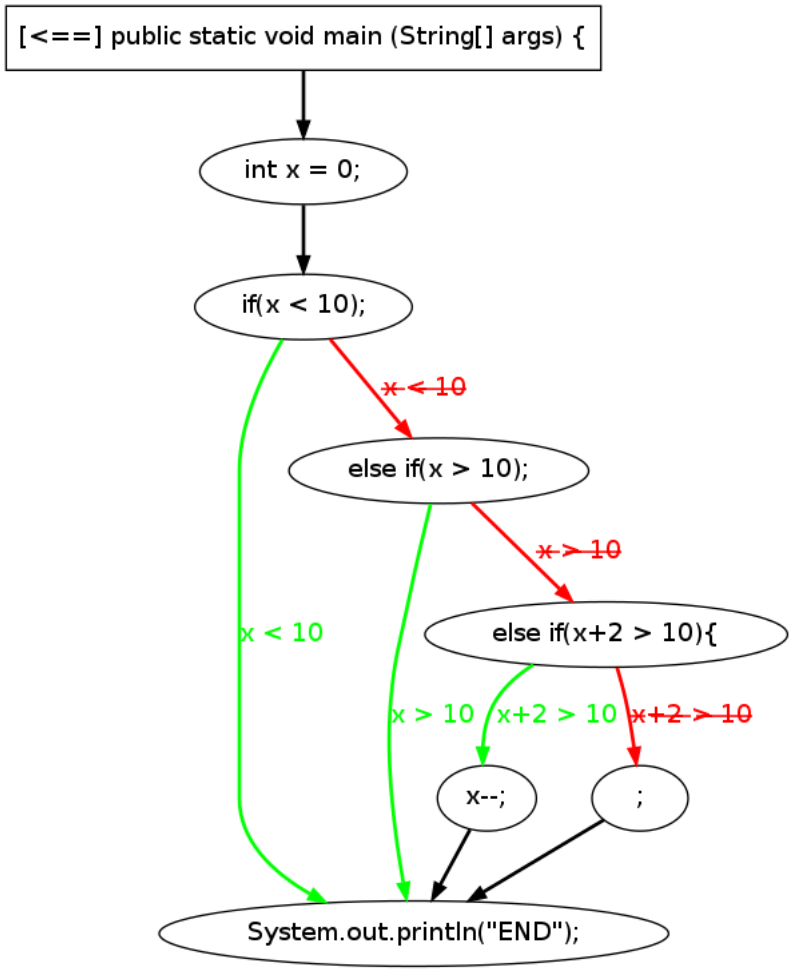


4.2.4 Empty statements

In Java, it is also possible to leave the block statement empty and end its line with a semicolon. This can be alternated with a normal statement, such as an `else if` statement with brackets. When put the following code into the main method:

```
int x = 0;
if(x < 10);
else if(x > 10);
else if(x+2 > 10){
    x--;
}
else;
System.out.println("END");
```

the result will look like this:



the `else`'s trailing semicolon will be seen as child, such that it is clear that there is an empty `else` block.

4.3 Ternary expressions

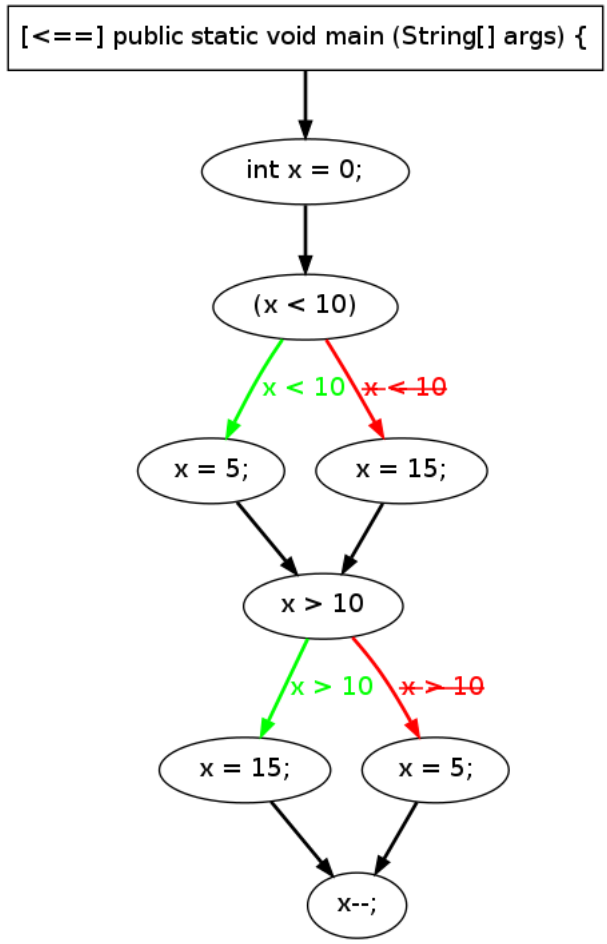
A ternary expression is a way of assigning a value to a variable depending on a certain condition, without using an `if` statement. Instead, the ternary expression contains something that looks like an `if` statement to determine which value it should assign to the variable. When turning a ternary expression into a control flow graph, it becomes more clear that is looks very similar to an `if` statement.

Consider this Java input code:

```
1 public class ExampleClass {
2
3   public static void main (String[] args) {
4     int x = 0;
5     x = (x < 10)? 5 : 15;
6     x = x > 10? 15 : 5;
7     x--;
8   }
```

```
9
10    public static int example_method1 (int x) {
11      return (x < 10)? 5 : 15;
12    }
13
14  }
```

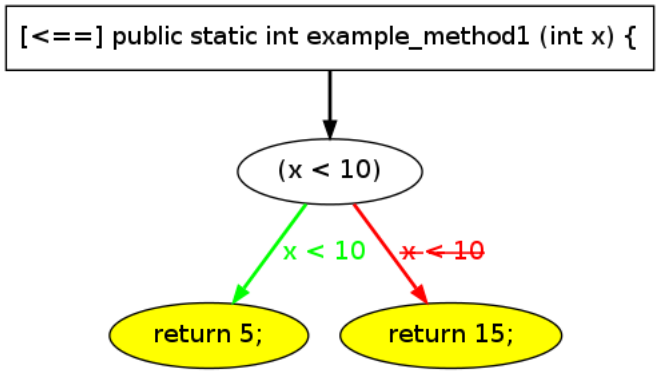
Then the control flow graph for the main method looks like this:



Note that the ternary expressions works both with and without brackets around the condition.

4.4 Ternary return

It is also possible to include a ternary expression inside a return statement. This is done inside example_method1 in the code above. The control flow graph of that method will look like this:



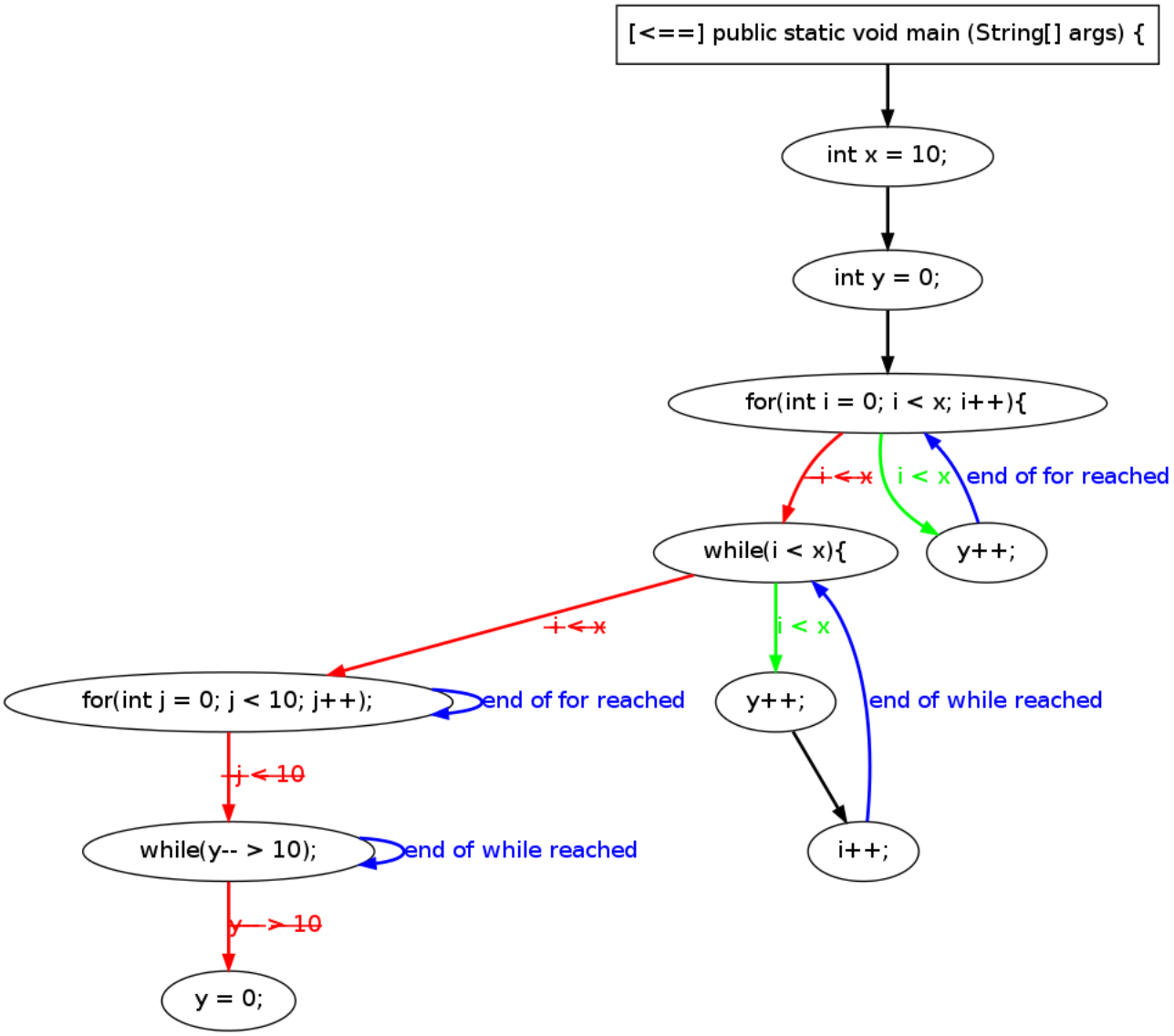
4.5 For and while loop

The control flow graph for a `for` loop and a `while` loop looks the same, since it behaves the same. The only difference is that a `for` loop has a built-in iterator, whereas a `while` loop does not.

When this Java code is put into the main method

```
1  int x = 10;
2  int y = 0;
3  for(int i = 0; i < x; i++){
4      y++;
5  }
6  while(i < x){
7      y++;
8      i++;
9  }
10 for(int j = 0; j < 10; j++);
11 while(y-- > 10);
12 y = 0;
```

its graph looks like this:



The loops have a green arrow to their children in case the condition is still true and a red arrow for when it is not. The blue arrow indicates the end of the loop and is included to show that the loop starts again from the beginning. A `for` loop and a `while` loop can also be written as one line ending with a semicolon. In this case, there is no green arrow because they do not have children. The blue arrow points to the statement itself since that is the only thing that loops. The arrow that leaves the loop is the red false arrow, since it will only leave the statement when the condition is false.

4.5.1 if statement at the end of the loop

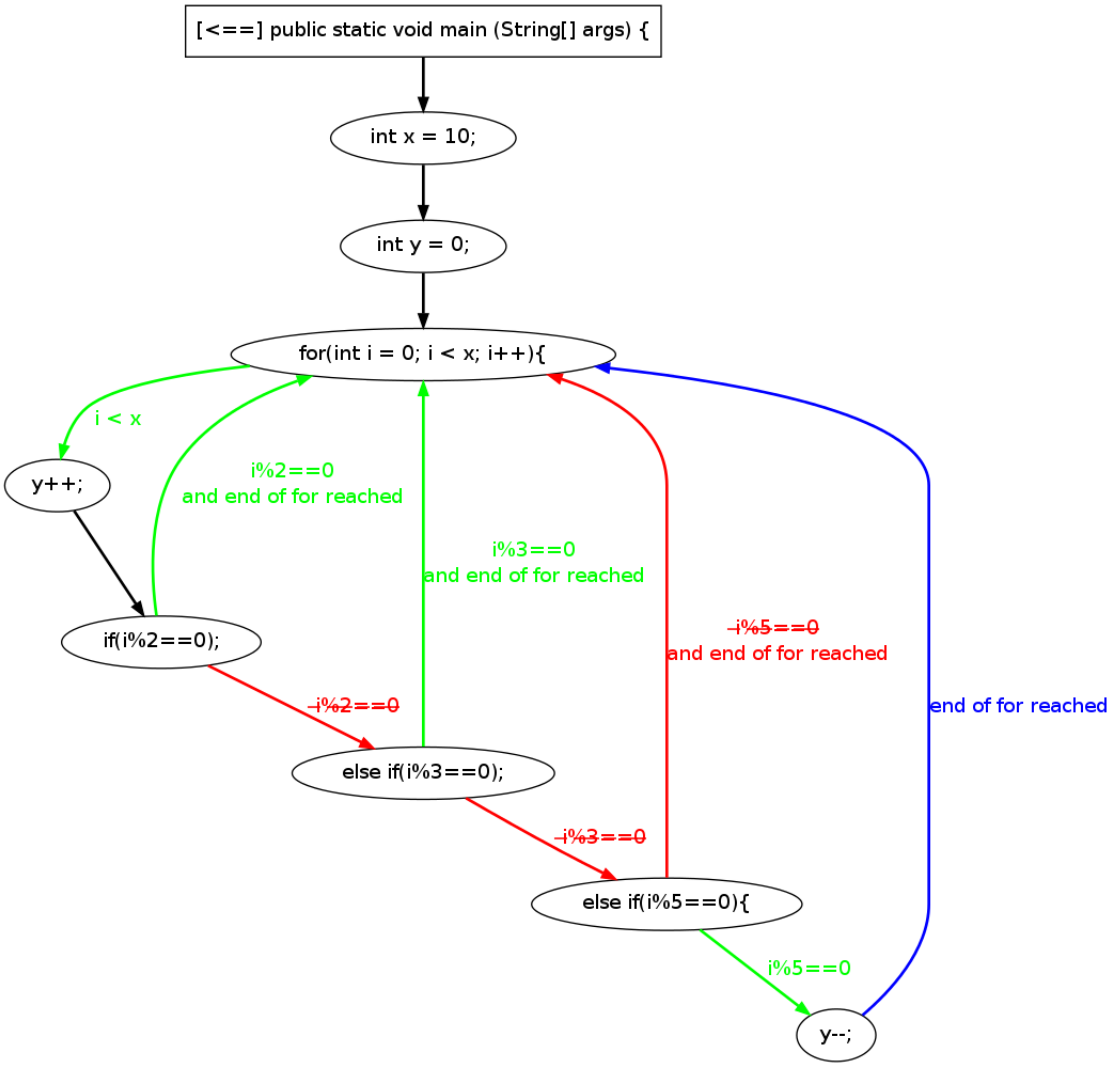
The last item inside the loop will have a blue arrow back to the beginning of the loop. But it is possible that there is more than one last item. For example, when an `if` statement with an `else` is the last item in the loop, both the children of the `if` block and the `else` block are the last items that should have blue arrows back to the beginning of the loop. But it is also possible that such an `if` statement does not have children. In that case its green arrow should go back to the beginning of the loop. If there is no `else` block, its red arrow should go

back to the beginning of the loop, since it is still possible that the `if` condition evaluates to false.

Consider this code:

```
int x = 10;
int y = 0;
for(int i = 0; i < x; i++){
 y++;
 if(i%2==0);
 else if(i%3==0);
 else if(i%5==0){
   y--;
 }
}
```

The result will look like this:



When the condition in the `if` statement is true, it skips the `else if`'s that follow, so its green arrow returns to the beginning. But if its condition evaluates to false, then it goes to the next `else if` with its red arrow. The same principle applies to that next `else if`. The second `else if` however, does have a child so its green arrow points to that child. Now the child is one of the last items so it should have a blue arrow back. Since there is

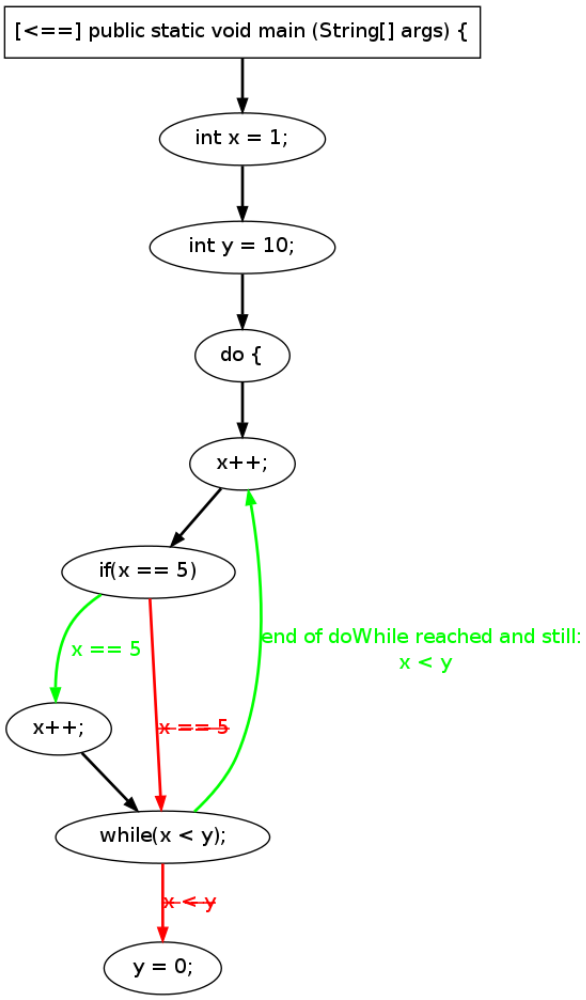
no `else` block, the red false arrow of the second `else if` points back to the beginning of the `for` loop, in case its condition evaluates to false.

4.6 Do-while loop

The `do-while` loop is similar to other loops, but it differs in that each cycle occurs before the condition is checked to be true or false. This also means that the condition is checked before returning back to the beginning of the loop (assuming the condition evaluates to true). That is why we used a green arrow instead of a blue arrow to point back to the beginning of the loop. The red arrow of the `while` part is the arrow that leaves the `do-while` statement if the condition in the `while` is not valid anymore. Here is an example:

```
int x = 1;
int y = 10;
do {
    x++;
    if(x == 5)
        x++;
} while(x < y);
y = 0;
```

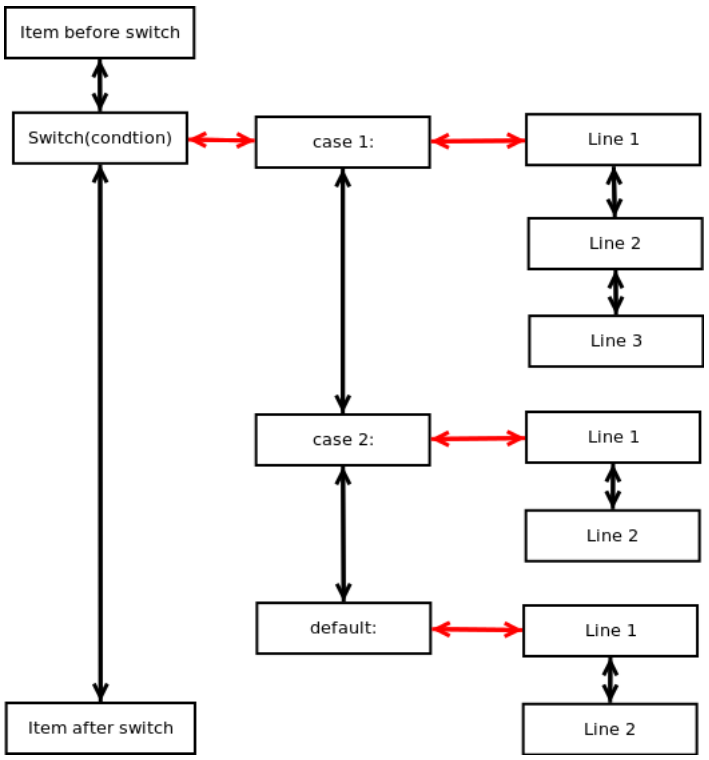
The code above will result in the following graph:



If the `while` part is put below the bracket of the do, the result would be the same.
We decided that, in a `do-while` loop, the beginning of the loop is its first child, since the node that says "do " will not be executed multiple times during runtime.

4.7 Switch statement

For the `switch` statement, it is less obvious how its structure in the code diagram should look. Other block statements have one child arrow to their first child and the rest of the children is connected to that child. A `switch` is a little more complicated, since it has cases, which can be considered as block statements themselves. The only difference is that they are not executed in order if a `break` statement occurs in one of the cases. During creating the program, we discovered that this is the most useful structure for the `switch` in the code diagram.



Using this structure, one can easily distinguish between cases, which makes it easier to create the control flow graph.

When creating the control flow graph of a `switch`, two things are important to check:

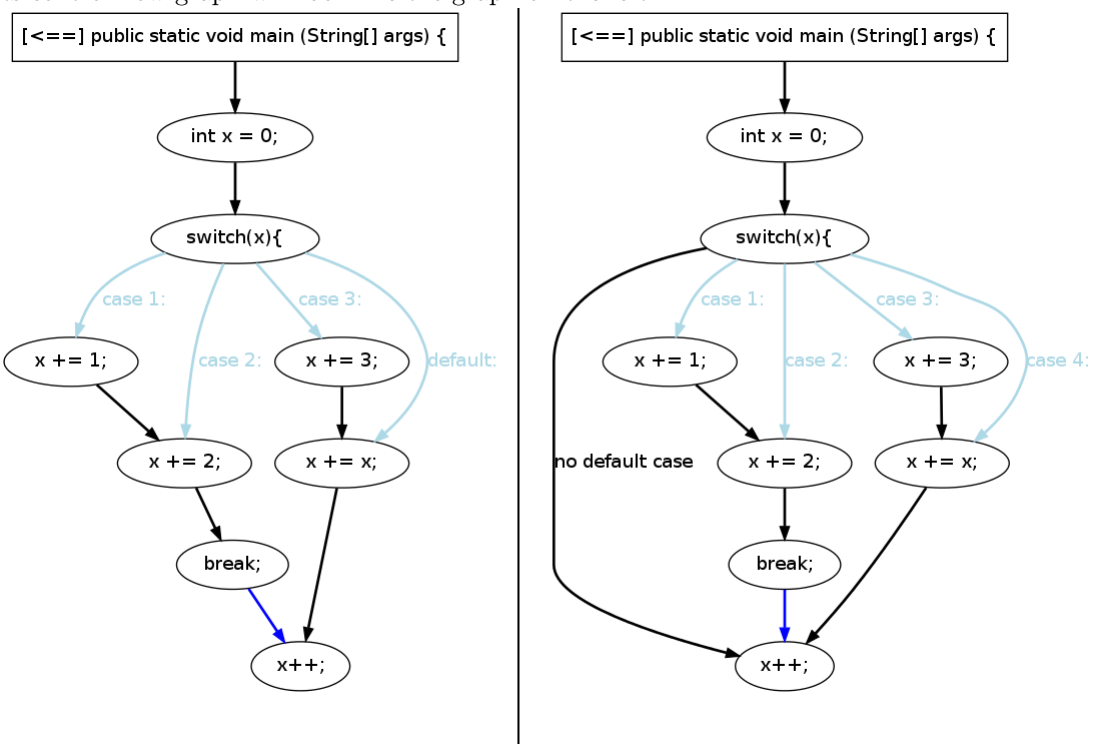
- Does the `switch` have default case? If so, no arrow should be drawn from the `switch` node to the item after the `switch`. A default case means that there is always a case that is entered, so there is no possibility of continuing without entering any `switch` case and thus there should not be an arrow that suggests there is. If, however, there is no default case it is possible that no `switch` case is entered, while the control flow of the program still continues. In that situation it should have an arrow from the `switch` node to the item after the `switch`.
- When creating the control flow graph of a case: does the case have a `break`? If not, then the last item in the case should point to the first child of the next case, or to the item after the `switch` if there is no other case. If the case has a `break`, it should always point to the item after the `switch`.

Consider this code:

```
int x = 0;
switch(x){
  case 1:
    x += 1;
  case 2:
    x += 2;
    break;
  case 3:
    x += 3;
  default:
```

```
    x += x;
}
x++;
```

Its control flow graph will look like the graph on the left.



We did not turn the name of each case ("case 1:" etc.) into a node. Instead, we created an arrow from the `switch` node to the first child of the case. The name of the case appears next to that arrow.

As we can see, there is no arrow from "switch(x){" to "x++;" since there is a default case. If we eliminate the default case by changing it to "case 4:", the graph will look like the graph on the right.

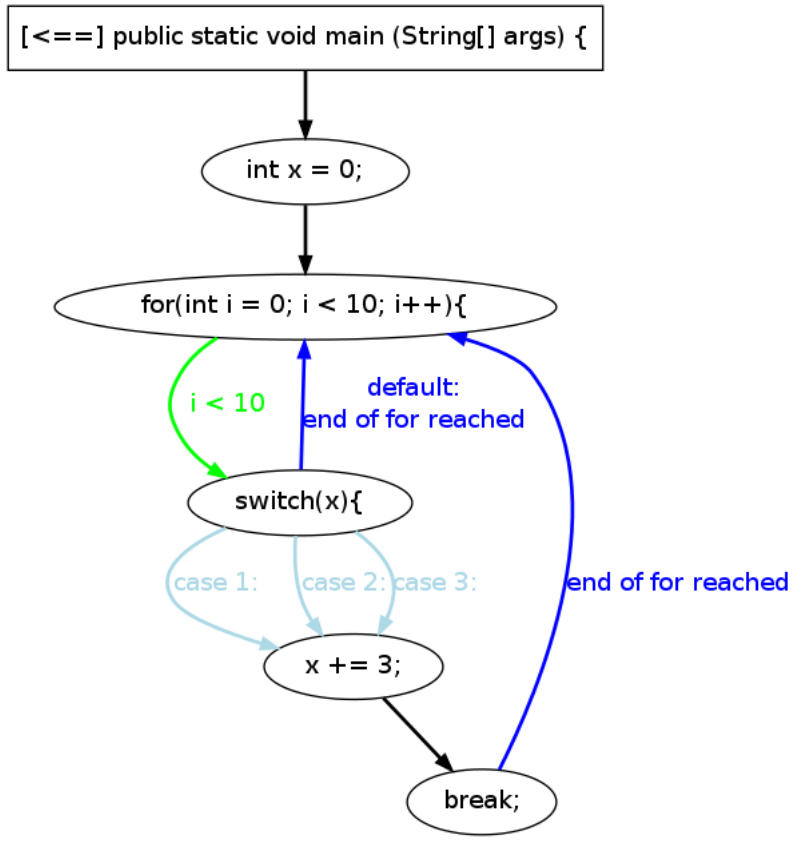
Case 2 has a `break`, so the `break` points to "x++;". Case 1 and case 3 do not have a `break` so their last items point to the next cases. The last case does not have a next case, so the last item in that case also points to "x++;".

4.7.1 Empty switch cases in a switch

If a case is empty, it will point to the next non-empty case. In the code below, the arrows of case 1 and case 2 both point to the same node as the arrow of case 3:

```
int x = 0;
for(int i = 0; i < 10; i++){
  switch(x){
    case 1:
    case 2:
    case 3:
      x += 3;
      break;
    default:
```

```
10    }
11 }
```



Note that the default case at the end is also empty. In C++, the compiler does not allow this and will show an error. But in Java this is possible. There is no more item after the `switch`, so the thing that is executed after the `switch` is the beginning of the `for` loop again. So the default case and the `break` must both point back to the `for` loop.

4.8 Try-catch-finally

The `try-catch-finally` statement consists of three parts. The first part is the `try` part, in which we try to execute some lines. If an exception occurs during one of these lines, the control goes to the first `catch` block that follows. The rest of the code in the `try` block will be ignored. The `catch` block's purpose is to run a piece of code when an exception occurred in the `try` block. A `catch` block can be set to only catch a specific exception, but it is also possible to create a `catch` block that catches all exceptions. Multiple `catch` blocks can be used below each other. If there is more than one `catch` block in our code, the `catch` blocks are tested in order. If the first `catch` block does not catch the exception, the second `catch` block is activated. If that `catch` block does not catch the exception either, the third `catch` block is activated and so on.
The third part is the `finally` block. It does not matter whether an exception will or occur or not, the `finally`

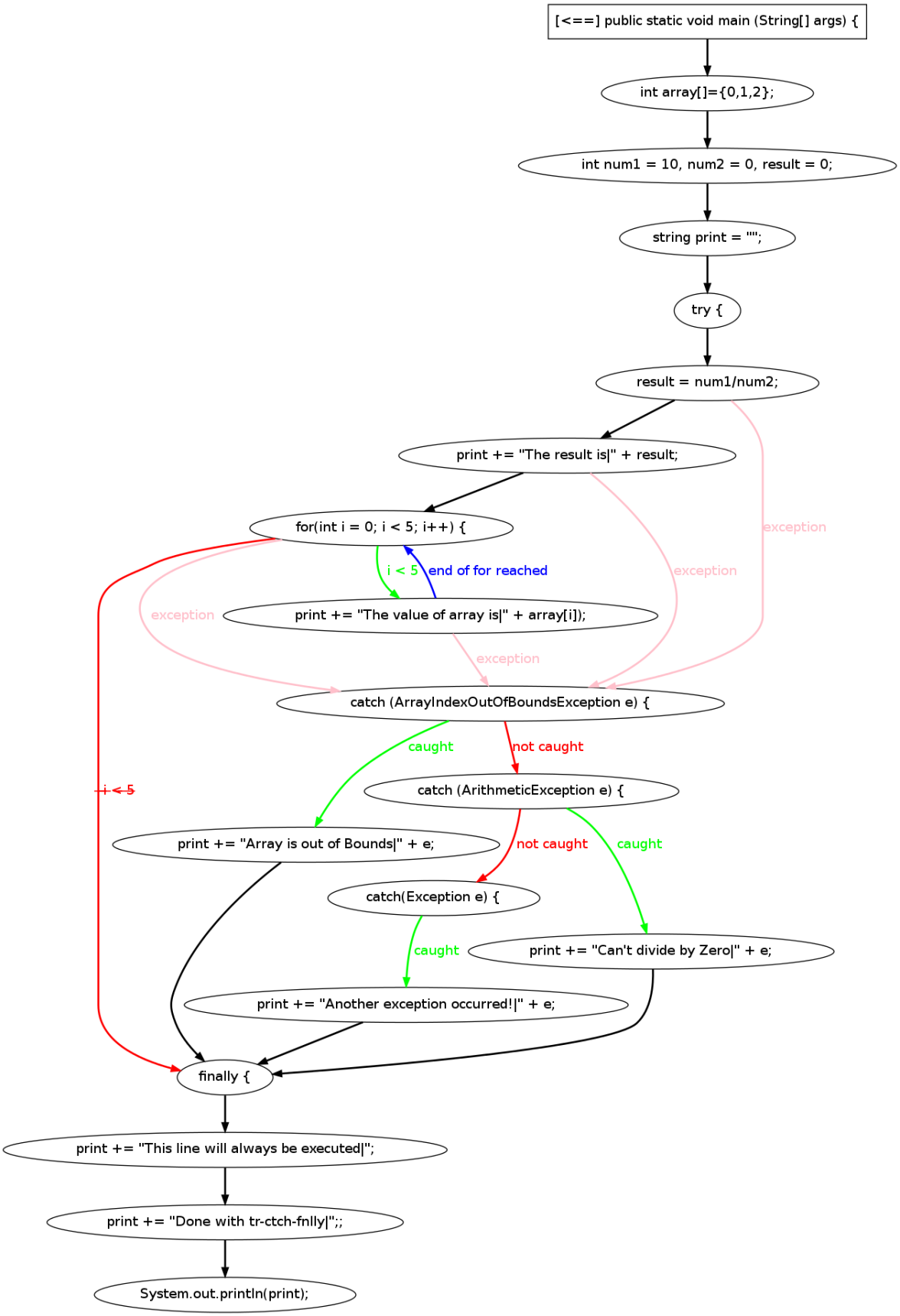
block is always executed.

Example 4.1

In the code below an Arithmetic Exception can occur when we divide by zero. If num2 does not equal zero, then an Array Index Out Of Bounds Exception can occur when an element outside the array is accessed.

```
int array[]={0,1,2};
int num1 = 10, num2 = 0, result = 0;
string print = "";
try {
  //This can cause an Arithmetic Exception (divide by zero)
  result = num1/num2;
  print += "The result is|" + result;

  //This can cause an Array Index Out Of Bounds Exception
  for(int i = 0; i < 5; i++) {
    print += "The value of array is|" + array[i];
  }
}
catch (ArrayIndexOutOfBoundsException e) {
  print += "Array is out of Bounds|" + e;
}
catch (ArithmeticException e) {
  print += "Can't divide by Zero|" + e;
}
catch(Exception e) {
  print += "Another exception occurred!|" + e;
}
finally {
  print += "This line will always be executed|";
}
print += "Done with tr-ctch-fnlly|";;
System.out.println(print);
```

The graph will look like this:



When looking at the graph, please note the following things:

- Each node inside the `try` block has an exception arrow to the first `catch` block. This is because an exception can occur in each node. When it does, the rest of the `try` block will not be executed since the first `catch` block is activated right after the exception occurs.

- The last catch "catch(Exception e)" is a catch that catches all exceptions. This means that it is impossible that the exception is not caught, therefore no red arrow is leaving this node.

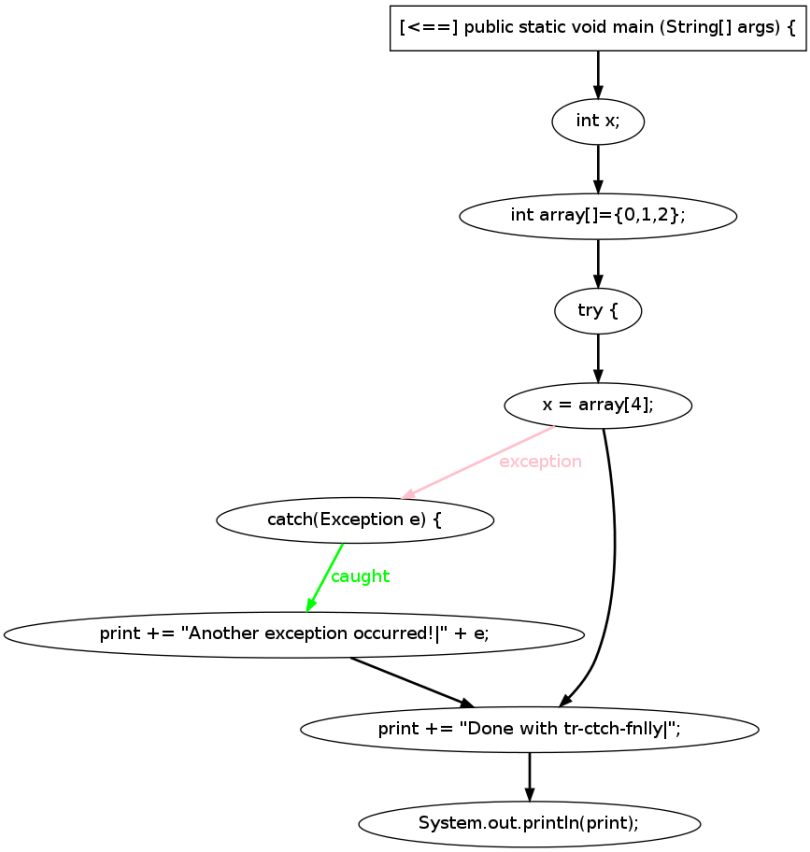
- The `for` loop is the last item inside the `try` block. So if no exception occurs inside the `try` block, this will be the point from which the `finally` block is activated. This is why the false arrow of the `for` loop points to the `finally` block.

Example 4.2

the `catch` blocks are executed in the same order as they appear in the source code until a catch is found that catches the exception. So if a catch that catches all exceptions appears above other catches, all of the other catches will be ignored. This is done in the code below:

```
int x;
int array[]={0,1,2};
try {
  x = array[4];
}
catch(Exception e) {
  print += "Another exception occurred!|" + e;
}
catch (ArrayIndexOutOfBoundsException e) {
  print += "Array is out of Bounds|" + e;
}
catch (ArithmeticException e) {
  print += "Can't divide by Zero|" + e;
}
finally {
  //print += "This line will always be executed|";
}
print += "Done with tr-ctch-fnlly|";
System.out.println(print);
```

The graph will look like this:

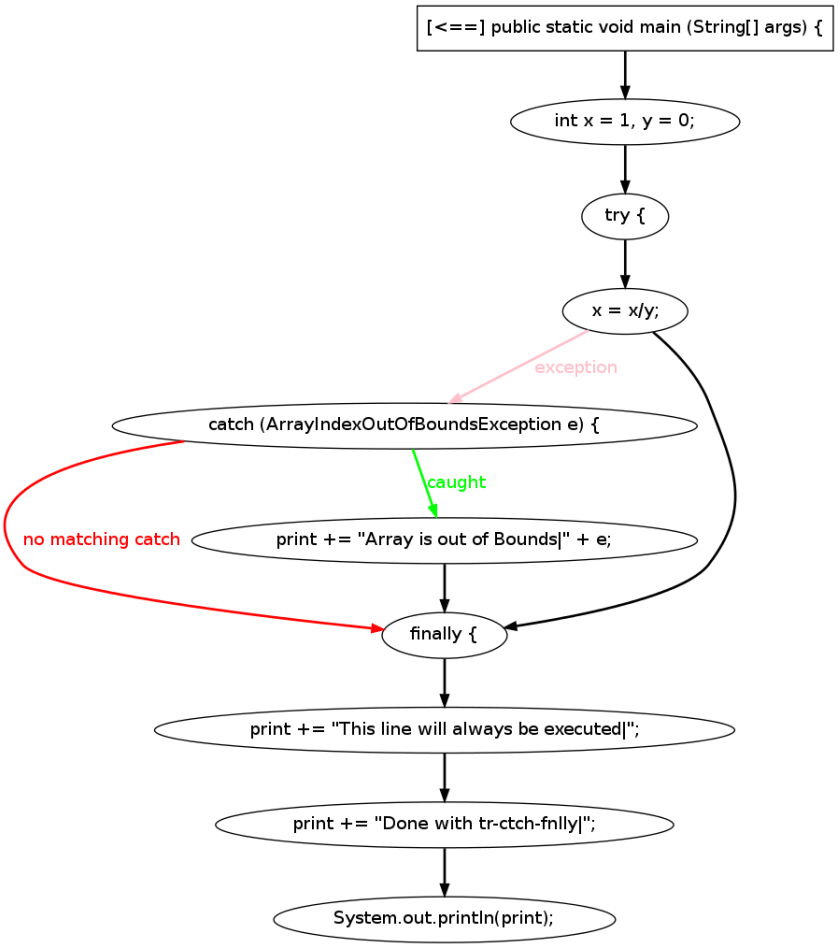


The rest of the catches does not appear in the graph, since the `catch` that catches all exceptions appears above them. There is no chance that another `catch` block will be activated. Please also note that the `finally` block does not appear at all in this graph. This is because its content is commented out, which is why there is no reason to display the "finally {" node either.

Example 4.3

If there is no catch that catches all exceptions, it is also possible that there is no catch that will catch the exception. In that case an arrow with the label "no matching catch" leaves the latest catch node and will point to the `finally` block. Here is an example:

```
int x = 1, y = 0;
try {
   x = x/y;
}
catch (ArrayIndexOutOfBoundsException e) {
  print += "Array is out of Bounds|" + e;
}
finally {
  print += "This line will always be executed|";
}
print += "Done with tr-ctch-fnlly|";
System.out.println(print);
```



4.9 Try-catch-finally with return

Since the program can read Java code that has methods, the code should also be able to handle return nodes properly. Usually this is not very difficult, but situations in which return nodes appear inside a `try-catch-finally` statement turned out to make things a lot more complicated. This is because there are some conflicting rules in the Java language:

1. No others instructions are executed after a return instruction is performed inside a method. The return node returns control to the point from where the method was called.
2. In a `try-catch-finally` statement, control will always flow to the `catch` blocks if an exception occurs.
3. In a `try-catch-finally` statement, the `finally` block is always executed.

So if a return node appears inside a `try` block for example, then rule 1 contradicts rule 2 and rule 3. No more instructions should be executed after the return instruction, but (the `catch` block and) the `finally` block should be executed. So how does Java handle these situations? By testing some pieces of codes, we found out that rule 1 makes an exception for rule 2 and 3. So if a return node appears inside a `try` block, the `finally` block (and the

`catch` block if an exception occurs) will still be executed. This can lead to very confusing situations, especially when `catch` blocks and `finally` blocks have their own return nodes. In those cases, the earlier return node is overwritten by new return nodes in catch and `finally` blocks.

Example 4.4

Consider this Java code with a return node in the `try` block that can cause an exception.

```
public class TestClass {

    public static int example_method1(int i){
    int array[]={5,10,15};
    try{
      return array[i];
    }
    catch (ArrayIndexOutOfBoundsException e) {
      return array[i--];
    }
    finally{
      if(i < 3){
        i++;
      }
      else{
        return array[0];
      }
    }
  }

  public static void main(String[] args) {
    for(int i = -3; i < 15; i++){
      try{
        System.out.println("With i=" + i + ", the returned value is: " + example_method1(i));
      }
      catch(Exception E){
        System.out.println("With i=" + i + ", an exception occurs");
      }
    }
  }
}
```

If we run this code in Eclipse, this will be the output:

```
With i=-3, an exception occurs
With i=-2, an exception occurs
With i=-1, an exception occurs
With i=0, the returned value is: 5
With i=1, the returned value is: 10
With i=2, the returned value is: 15
With i=3, an exception occurs
```

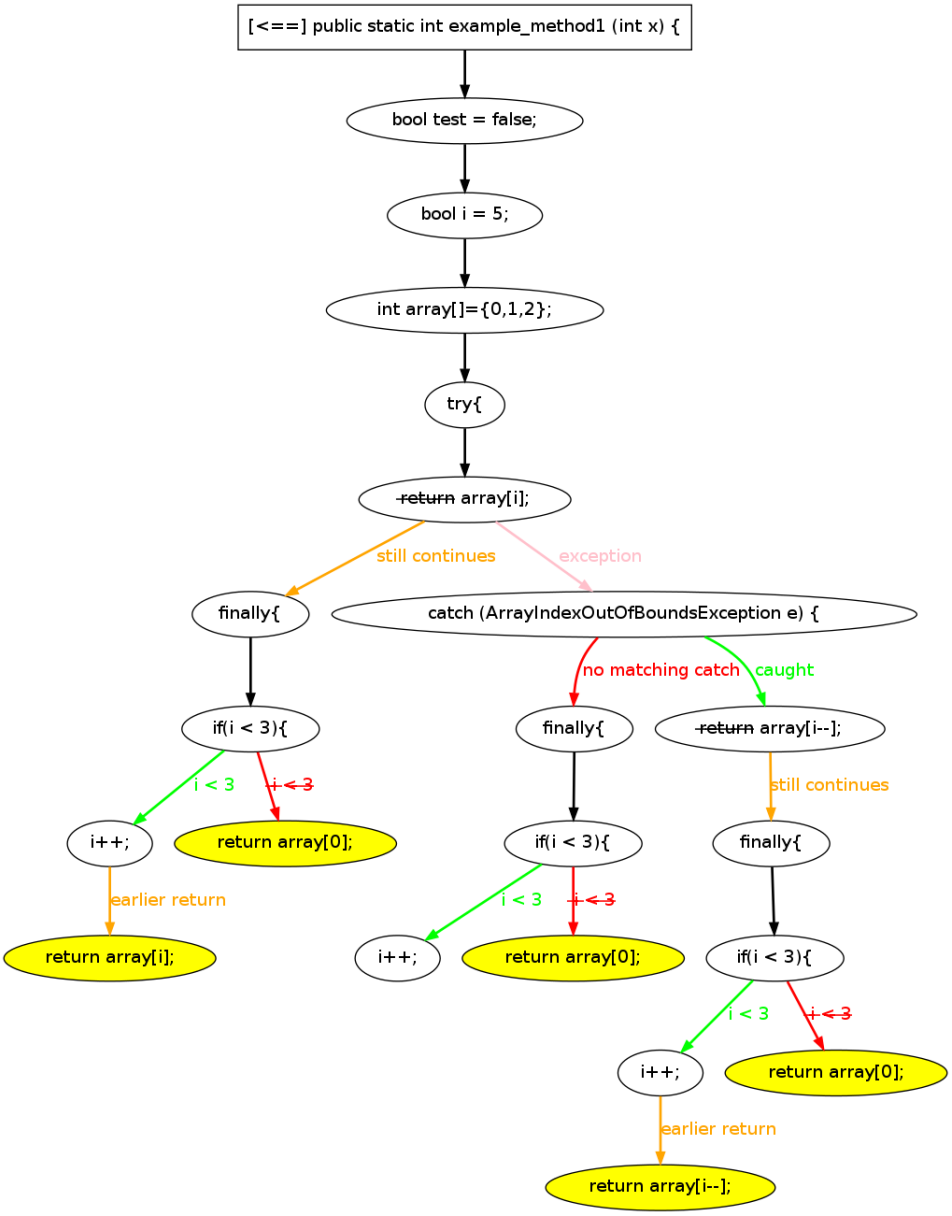
```
With i=4, the returned value is: 5
With i=5, the returned value is: 5
With i=6, the returned value is: 5
With i=7, the returned value is: 5
With i=8, the returned value is: 5
With i=9, the returned value is: 5
With i=10, the returned value is: 5
With i=11, the returned value is: 5
With i=12, the returned value is: 5
With i=13, the returned value is: 5
With i=14, the returned value is: 5
```

This is because this is what happens:

- If i $<$ 0, "return array[i];" in example_method1 will cause an exception. The exception is caught, but the return in the `catch` block causes a new exception, but the `finally` block will still be executed. Since i $<$ 3, the return will not be overwritten inside the `finally` block. The exception keeps existing and will be caught inside the main method.
- If 0 $<=$ i $<$ 3, no exception will occur in the `try` inside the example method, such that "return array[i];" can return a value without any problems.
- If i $==$ 3, then an exception occurs in the `try` block in the example method. The `catch` block that follows causes a new exception, which is not caught in the example method. If it would be "return array[--i];" instead of "return array[i--];", there would be no exception, since i is decreased first, such that i $==$ 2. But right now the exception occurs before decreasing i. The `finally` block will be executed with i $==$ 2, such that the `if` statement will be entered. There is no new return node that overwrites the old one, so the exception keeps existing and will be caught by the main method.
- If i $>$ 3, an exception occurs in the `try` block in the example method. The `catch` block tries a new return, overwriting the return node from the `try`, but that causes an exception too. When the `finally` block is executed, i is not smaller than 3, so "return array[0];" will be executed, overwriting the return node from the `catch` block. The returned value will be array[0], which causes no exception and can be returned without any problems.

So return nodes can be overwritten in a `try-catch-finally` statement, but that does not have to be the case. In the example code above, an earlier return node is overwritten when "return array[0];" is executed in the `finally`, but it is not overwritten when i $<$ 3 and "i++;" is executed instead of "return array[0];". Since multiple returns can occur in both the `catch` and `try` blocks, we should know what return we should use if it is not overwritten in the `finally` block. But how can we determine which return we came across earlier when we are already in the `finally`? When running the Java code, we can keep the most recent return in mind, but a control flow graph does not run the code so it does not know which return is the most recent return. Therefore we came up with a different solution: instead of using one `finally` block in the graph, we gave each return node its own sub graph with its own `finally` blocks. By doing so, it is clear what return node should be used if it

is not overwritten in the `finally`. This requires copying the `finally` block. Copying nodes is not something we prepared for, so that is why it is not described in chapter 2. Copying a node means creating a new struct with the same content as the struct from the node that is copied, except for the code diagram arrows and the control flow graph arrows the original struct contains. The source code of JShowFlow also includes a function called "copyTree" which calls the copyNode function recursively for all of its child nodes while connecting them the same way as in the original tree. Using this function the `finally` block can be copied.
If we do this, the graph of example_method1 from Example 4.4 looks like this:



As we can see, the word "return" is struck through in the `try` block's return node. This because there is no return performed at that point. The rest of the return node ("...array[i];") is not struck through, because that part is executed. This is the part that can cause an exception, while not returning.
Each orange arrow with the text "earlier return" is an arrow to a node that is a copy of an earlier return

node that was not overwritten by a new return node. The leftmost return node in this graph is a copy of the return node from the `try`, while the other earlier return node (lowest node in the graph) is a copy of the return in the `catch` block. In case an exception occurs in the `try` block and the exception is not caught, then the `finally` block below the "no matching catch" arrow is used. In this `finally` block, there is no earlier return node attached to "i++;", since we already know that that return node causes an exception.

It is also possible that another `finally` block is executed after the current finally.

Example 4.5

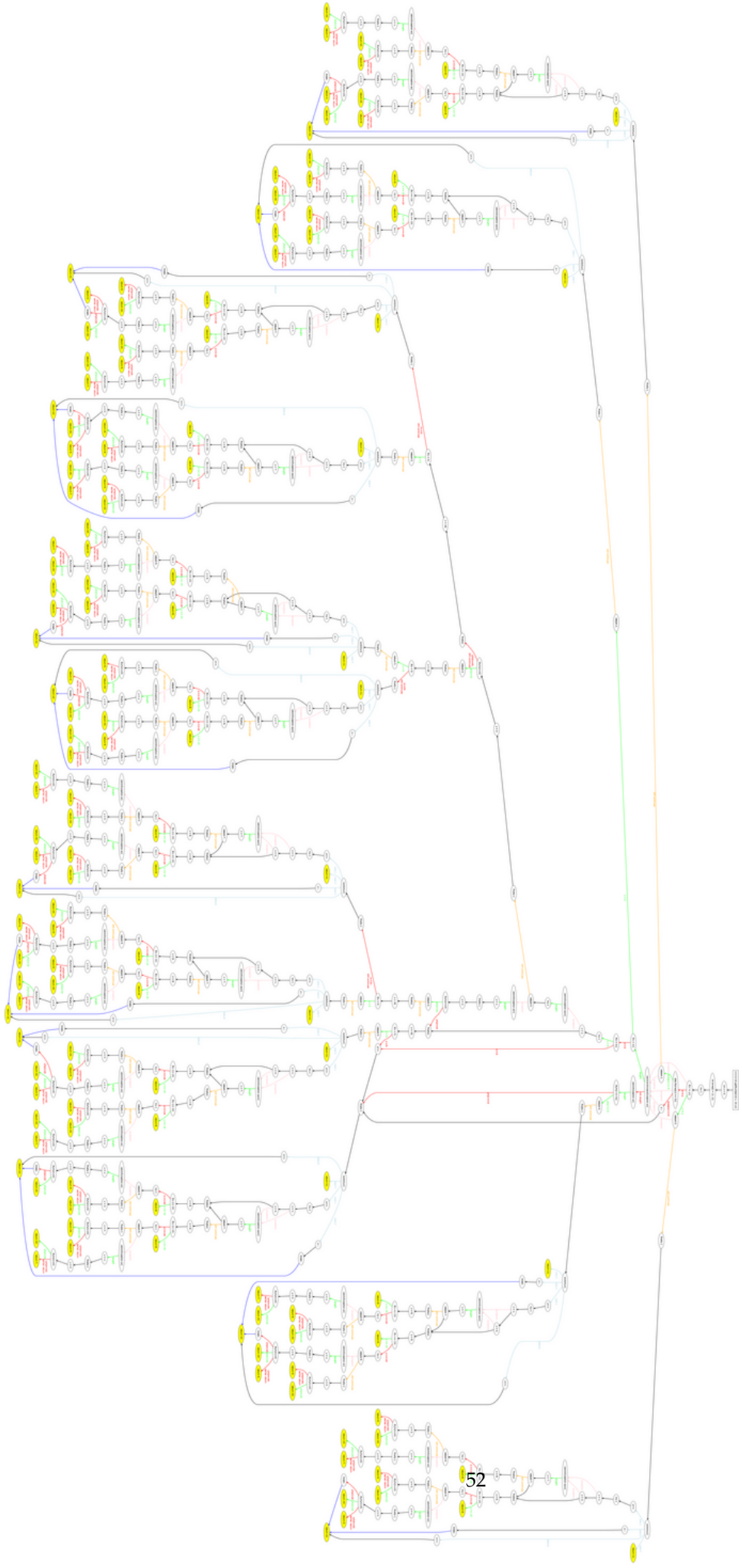
Things get even more complicated if a `try-catch-finally` statement with return occurs inside another `try-catch-finally` statement. In that case, each `finally` block must be executed before anything is returned. This means that, even if return nodes are not overwritten in the `finally` block, they can still be overwritten in the `finally` block of the other try-catch-finally statement that follows after that. But again, it is also possible that the later finally does not overwrite earlier returns, but it must still be executed.

Consider this large code with a `try-catch-finally` statement with return nodes inside another `try-catch-finally` statement with return nodes:

```
1    int x = 0, y = 0, z = 0;
2  int array[2] = {1, 2};
3  try {
4    if(x != y){
5      return 0;
6    }
7    else if(array[x] == 0){
8      return 1;
9    }
10   z--;
11 }
12 catch(ArithmeticException e){
13   if(x > 2){
14     return 2;
15   }
16   if(x > 5){
17     y += y;
18   }
19   y++;
20 }
21 catch(Exception e){
22   if(y%2 == 0){
23     return 4;
24   }
25 }
26 finally {
27   switch(y){
```

```
28        case 1:
29          y++;
30          try {
31            z = 1;
32            z = 4;
33          }
34          catch(Exception eyo){
35            z = 7;
36            return 5;
37          }
38          finally{
39            z = 8;
40            if(x)
41            return 6;
42            try {
43              return 7;
44              x = 9;
45            }
46            catch(Exception e){
47              y = x;
48            }
49            finally {
50              y = 2;
51              if(y%2==0)
52              return 8;
53            }
54          }
55          break;
56        case 2:
57          y--;
58          break;
59        case 3:
60          return 9;
61        case default:
62          x++;
63      }
64      return 10;
65    }
66    while(x < y){
67      x++;
68    }
69    return 11;
70  }
```

Its graph is even bigger:



Unfortunately the graph can not be made readable in this document, but we can get a good impression from the big amount of paths and return nodes that the input code will generate. In the program itself visibility is not a problem since the graph can be viewed in an internet browser. This makes it possible to zoom in to be able to read the graph.

4.10 Break

The `break` statement can appear inside a loop, a `switch` or both. If it occurs in both, it is important to check which statement is the closest to the `break` statement. It is also important to check if the statement in which the `break` occurs is perhaps the last statement inside a loop. If that is the case, then the `break` should point to the beginning of that loop.

4.10.1 Break in a loop

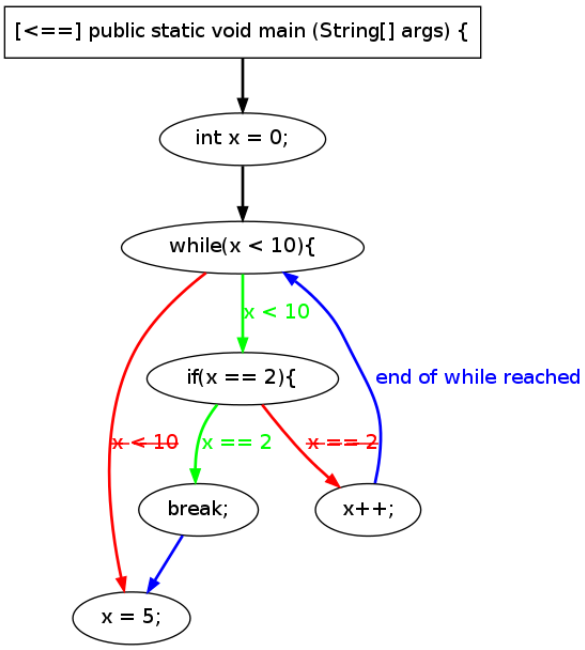
A `break` inside a loop must point to the item after the loop, if available.

Example 4.6.1

Consider this code:

```
1 int x = 0;
2 while(x < 10){
3   if(x == 2){
4     break;
5   }
6   x++;
7 }
8 x = 5;
```

The graph of that method looks like this:



4.10.2 Break in multiple loops

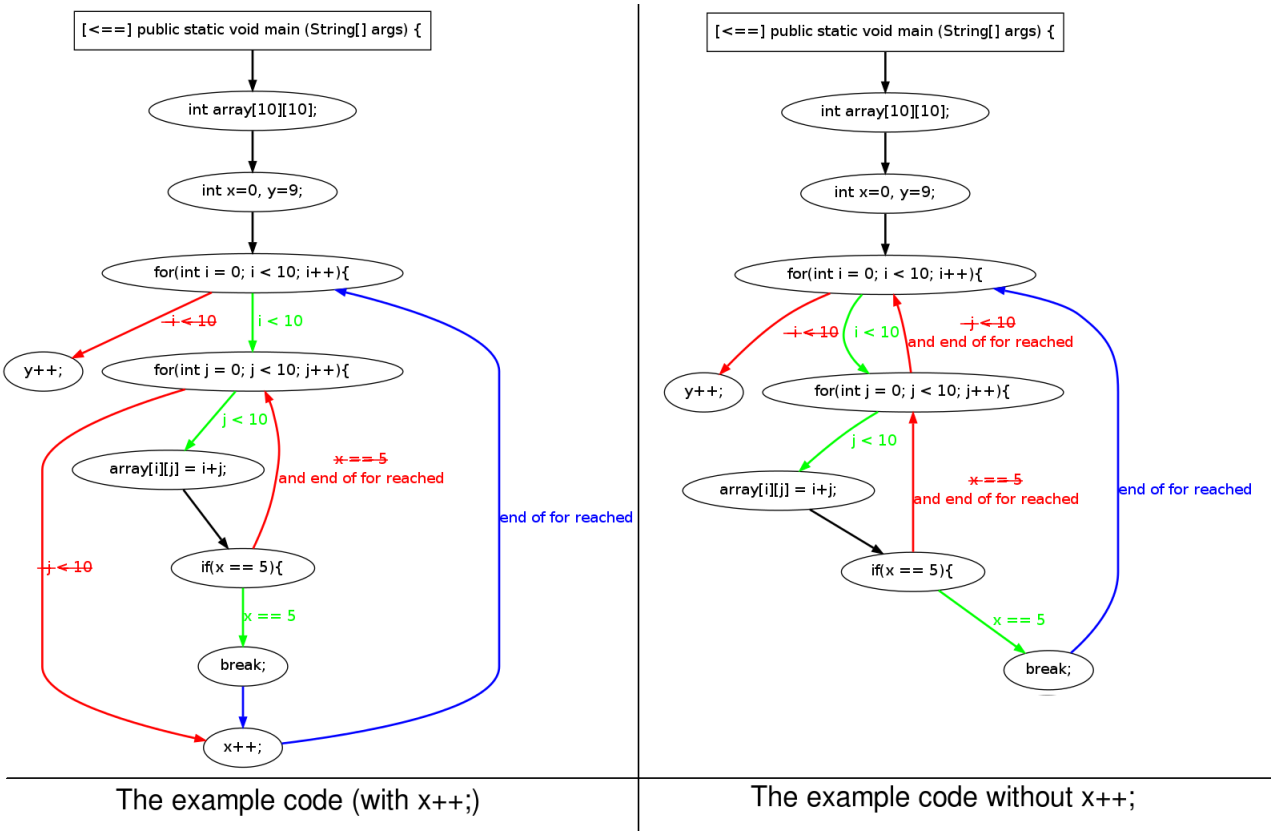
A `break` can also appear inside a loop that is part of another loop.

Example 4.6.2

Consider this code to be in the main method:

```
int array[10][10];
int x=0, y=9;
for(int i = 0; i < 10; i++){
  for(int j = 0; j < 10; j++){
    array[i][j] = i+j;
    if(x == 5){
      break;
    }
  }
  x++;
}
y++;
```

Then the control flow graph of the main method would look like the graph on the left:



The graph on the right is the graph we obtain by eliminating "x++;" in the input code. With "x++;" being part of the code, the **break** statement is straightforward: it should break outside the loop it is part of to the next item after that loop. Without "x++;", there is no item after the loop the **break** is part of. This makes the inner loop the last item inside the outer loop. This means that the inner loop should have an "end of loop" arrow to the outer loop. It also means that the **break** should point to the right item, since "x++;" is not part of the code anymore. If the **break** would point to "y++;", it would break both the inner and outer **for** loop, while a **break** can only break one statement. So instead, it should point to the item "x++;" used to point to, which is the outer **for** loop. This is the node that will be executed after the **break**.

4.10.3 Break inside a switch

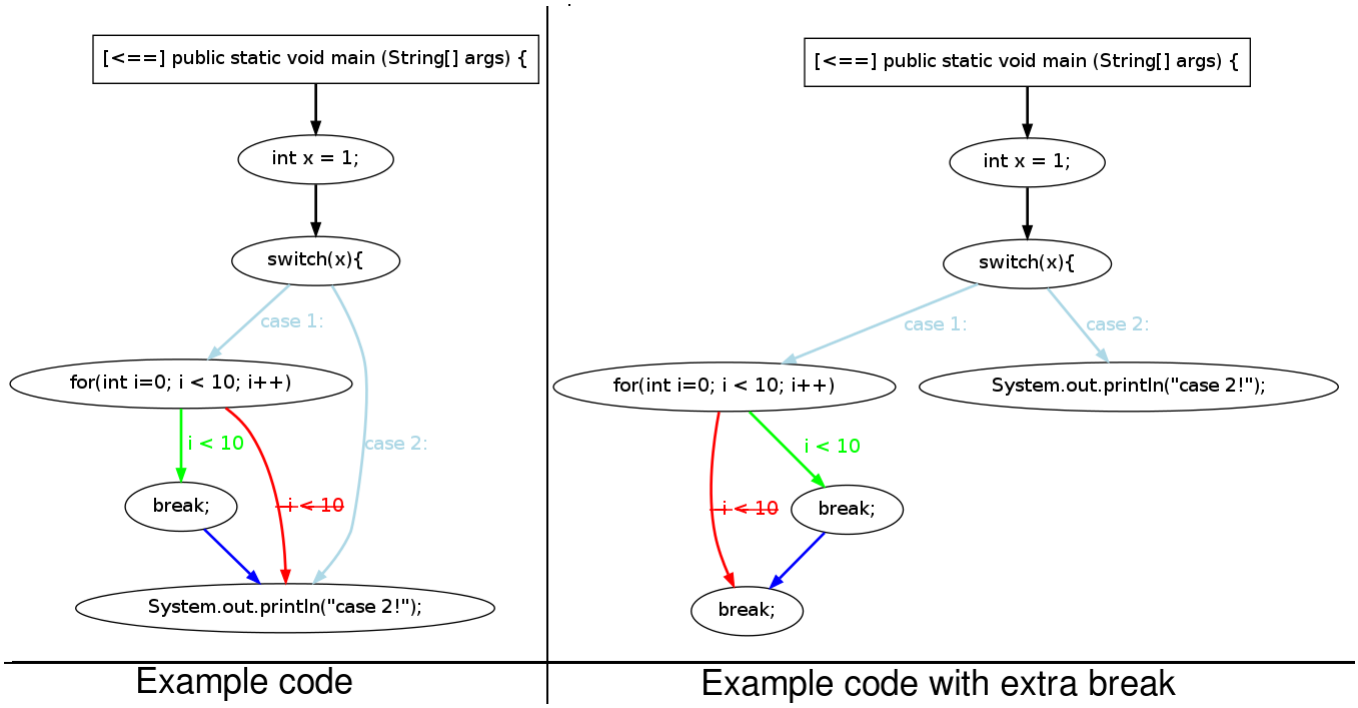
Section 4.7 shows how a **break** behaves inside the **switch** statement. But this does not mean that a **break** inside a **switch** statement always has the purpose to prevent the program from entering the next **switch** case. When loops occur inside **switch** cases, the **break**s purpose depends on where it appears.

Example 4.7

```
1 int x = 1;
2 switch(x){
3   case 1:
```

```
    for(int i=0; i < 10; i++)
    break;
  case 2:
    System.out.println("case 2!");
}
```

In this piece of code, it looks like the `break` belongs to the `switch` case, but it actually breaks the `for` loop that is inside the `switch` case. This `break` does not prevent the program from entering case 2. This example shows that is important to check where a `break` statement belongs to, to understand how it behaves. The `break` is part of the `for` loop and also part of the `switch` case, but since the `for` loop is closer to the `break`, the `break`'s behaviour should apply to the `for` loop. Adding another `break` right below the first `break` will result in the image on the right. That `break` will break the `switch` case since it is not part of the `for` loop. But it does not have anything to break to, since the `switch` does not have a next.



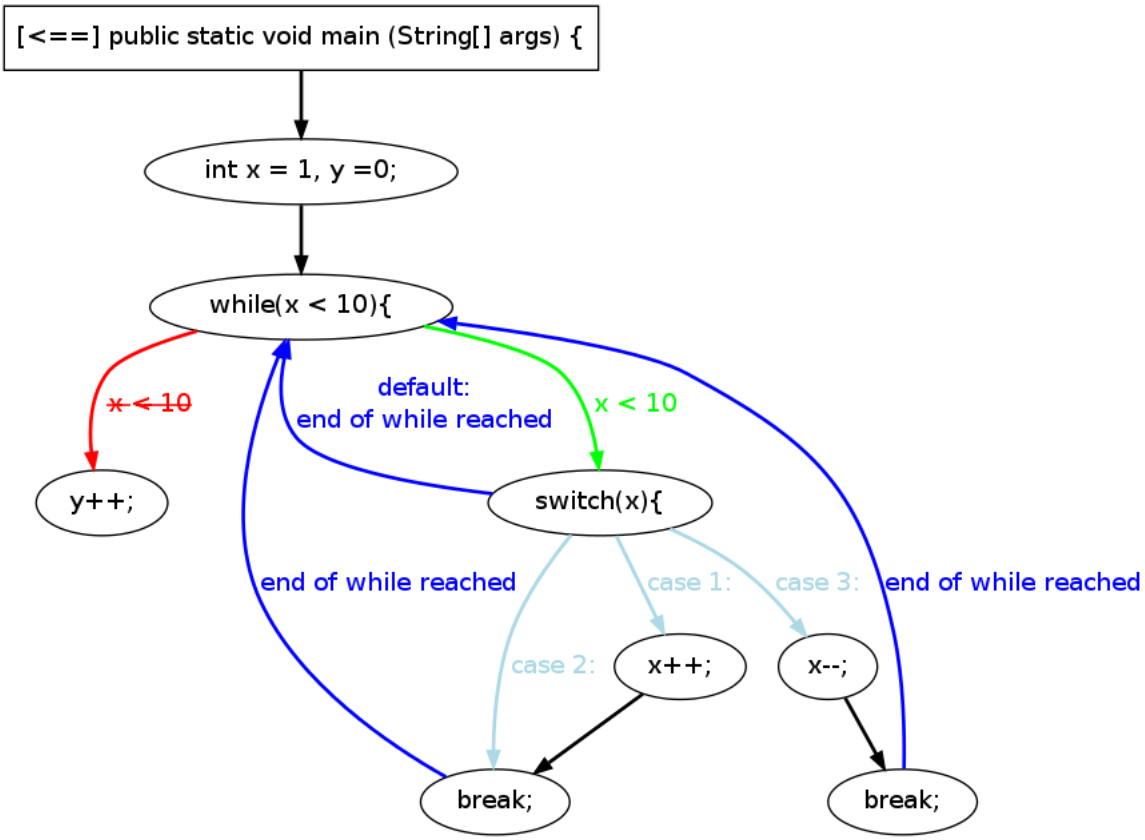
Example 4.8

A loop can occur inside a `switch` case, but a `switch` can also appear inside a loop. If the `switch` is the last item inside that loop, each `break` should point back to the beginning of that loop.

```
int x = 1, y =0;
while(x < 10){
  switch(x){
    case 1:
      x++;
    case 2:
      break;
    case 3:
```

```
        x--;
        break;
    }
}
y++;
```

The graph of the code looks like this:



Since there is no default case in the `switch`, the `switch` also points back to the beginning of the loop in case x does not equal 1, 2 or 3.

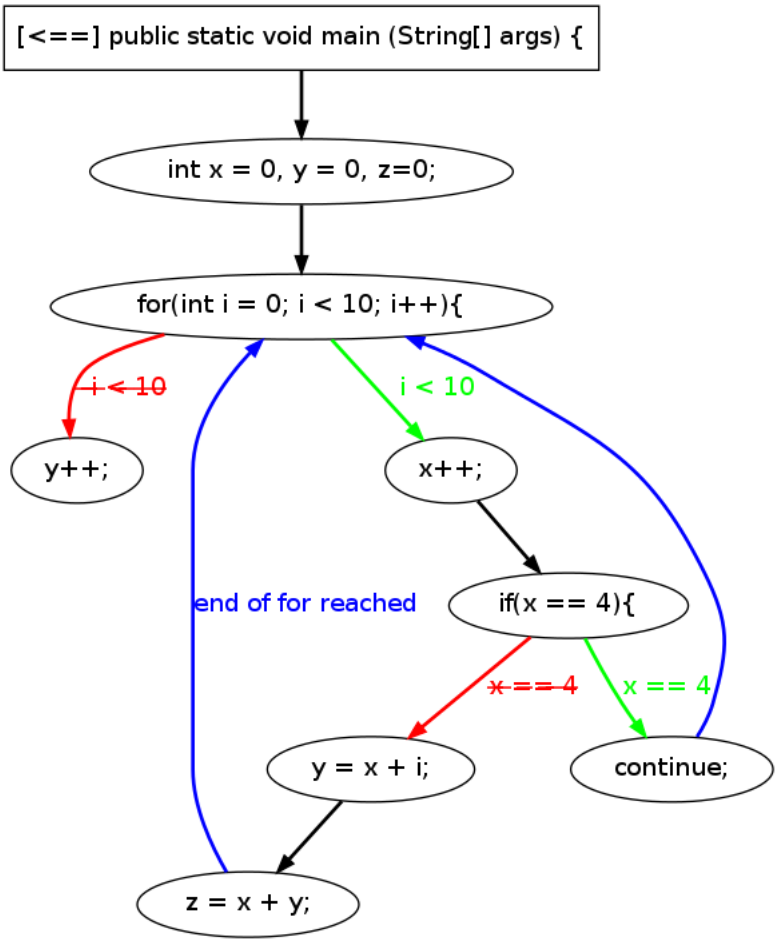
4.11 Continue statement

The `continue` statement is easier to process than the `break` statement, since it can only mean one thing: return to the beginning of the loop.

Example 4.9

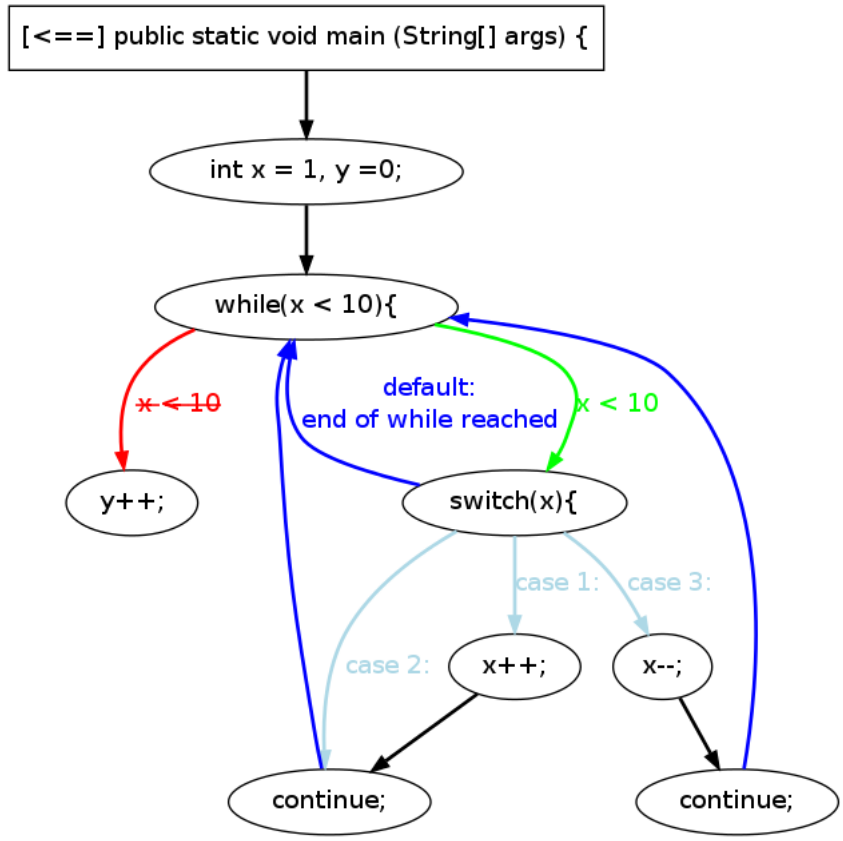
```
int x = 0, y = 0, z=0;
for(int i = 0; i < 10; i++){
  x++;
  if(x == 4){
```

```
5         continue;
6     }
7     y = x + i;
8     z = x + y;
9 }
10 y++;
```



If the `continue` occurs in nested loops, it will point to the innermost loop. In other words, if the `for` loop would occur in another loop, the `continue` statement will still point to the `for` loop.

If a `break` occurs inside a loop, it means that the program should leave the loop from that point. If a `continue` statement appears inside a loop, it means that the program should return to the loop. So the `break` statement and the `continue` statement are often each others opposites. However, in some situations they can behave exactly the same. In the code of Example 4.8, the `break`s will also return to the `while` loop, just as `continue` statements would. Replacing the `break` statements with `continue` statement will result in exactly the same graph:



4.12 Method calls

Another method can be called inside a line of code. That method can be in the same class or in another class as the current method. In chapter 2 we discussed how a call to another method can be linked to the correct method in the class diagram. Here we will show the results.

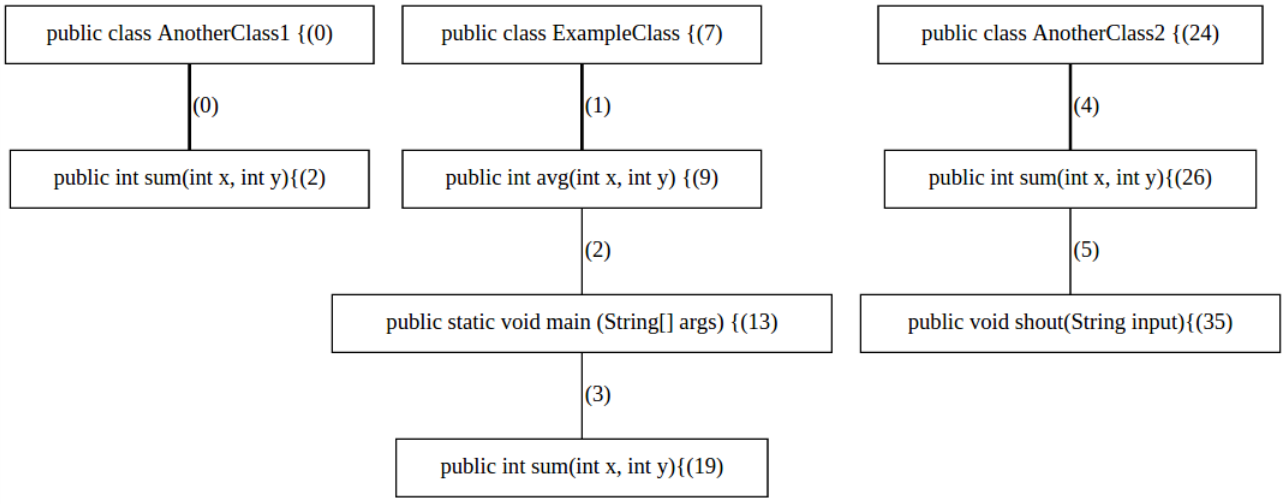
Example 4.10

We will use this piece of code with multiple classes:

```
1  public class AnotherClass1 {
2
3    public int sum(int x, int y){
4      return (x+y);
5    }
6
7  }
8  public class ExampleClass {
9
10   public int avg(int x, int y) {
11     return (x+y)/2;
12   }
13
```

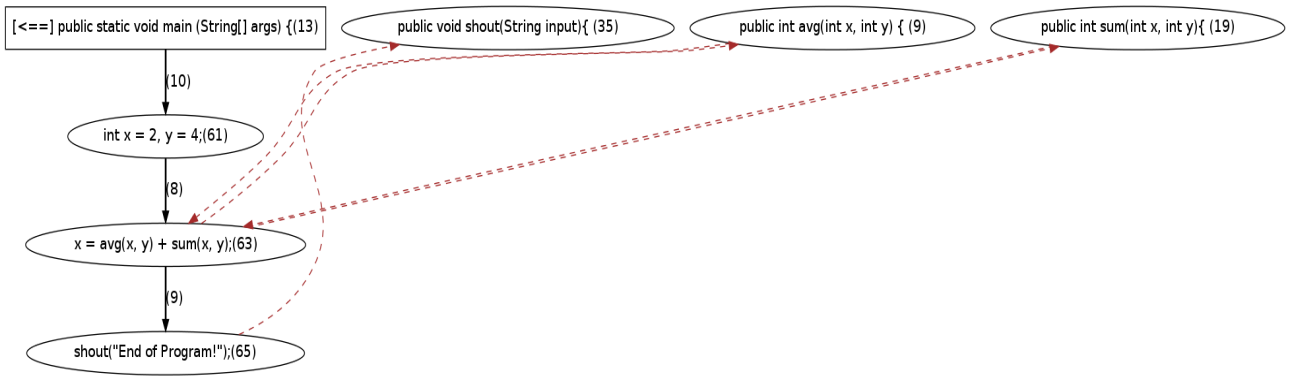
```
14    public static void main (String[] args) {
15      int x = 2, y = 4;
16      x = avg(x, y) + sum(x, y);
17      shout("End of Program!");
18    }
19
20    public int sum(int x, int y){
21      return x+y;
22    }
23
24  }
25  public class AnotherClass2 {
26
27    public int sum(int x, int y){
28      return x+y;
29    }
30    public void shout(String input){
31      System.out.println(input);
32    }
33  }
```

The class diagram for this piece of code looks like this:



As we can see, there are numbers at the end of each block. This is the ID of each struct or arrow. Showing the ID's can be turned on or off in JShowFlow with the variable *showID* at the beginning of the code. In previous graphs we turned it off because it would be only distracting. But here it is turned on to show to which method a line with a method call points to.

Now the graph of the main method looks like this:



When looking at the graph we can notice the following things:

- The line "x = avg(x, y) + sum(x, y);" has two method calls, one to the method avg and one to the method sum.
- The method sum is below the main method, but it can still be found and linked to the method call.
- AnotherClass1 and AnotherClass2 both also contain a method called sum. JShowFlow knows that it should look in the current class first before searching other classes. That is why it picked the sum method from the current class. This can be verified by looking at the ID's in the control flow graph and comparing these with those in the class diagram. It points to sum with the ID being 19. In the class diagram we can see that this is the sum method from the ExampleClass, which is the current class.
- The line "shout("End of Program!");" calls for a method inside another class that is below the current class. That method can also be found and linked to the call without problems.
- There are dashed arrows to the three methods that are called, but only avg and sum have an arrow back to the node that calls the methods. This is because the method shout does not have a return node, so it does not return anything back to the main method.

When we see this graph in a browser, we can click on a node that represents one of the other methods. When we do that, we will see the control flow graph of that method.

4.13 Examples of other things that should work

During creating the control flow graph generator, we encountered a great amount of situations we were not prepared for. Some of them are discussed earlier, some other situations are discussed in this section.

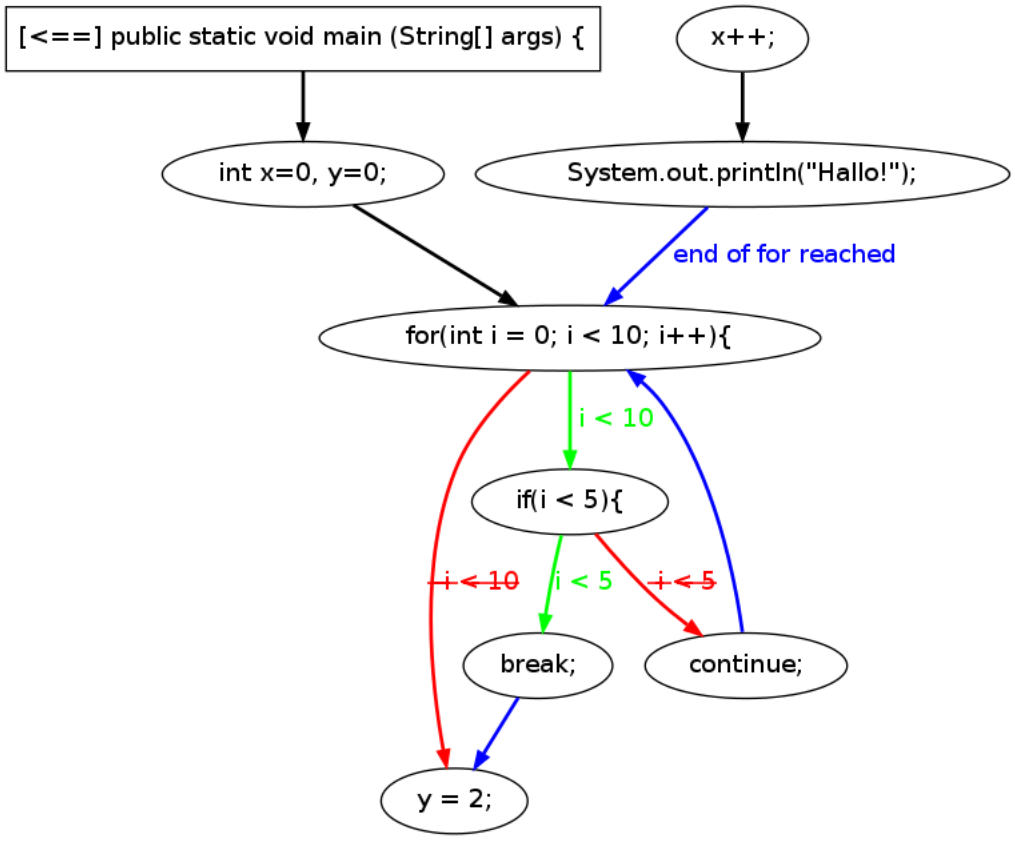
4.13.1 Statically unreachable nodes should not appear

At one point we encountered the problem that nodes that are not used still appear.

Example 4.11 Consider this code:

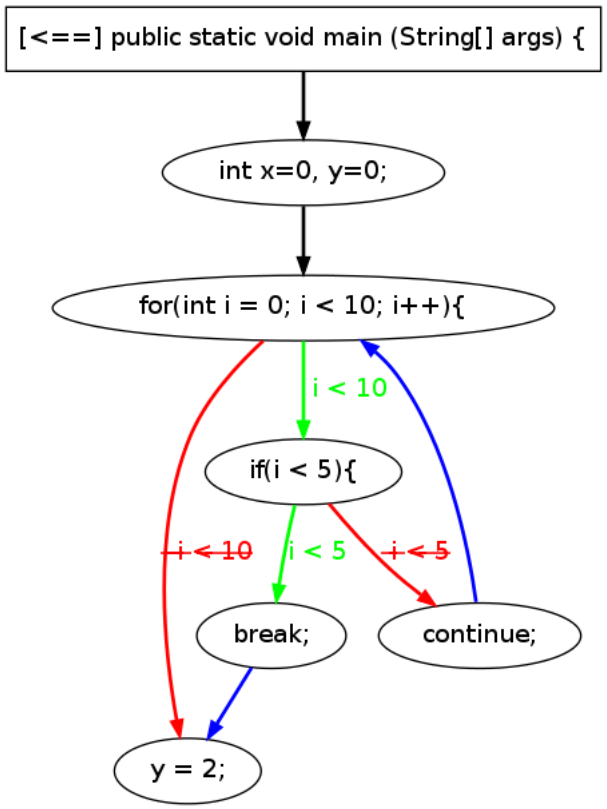
```
1  int x=0, y=0;
2  for(int i = 0; i < 10; i++){
3    if(i < 5){
4      break;
5    }
6    else{
7      continue;
8    }
9    x++;
10   System.out.println("Hallo!");
11 }
12 y = 2;
13 }
```

In this code the lines "x++;" and "System.out.println("Hallo!");" are unused, i.e. unreachable code. The `break` will point to "y = 2;" and the `continue` will point back to the beginning of the `for` loop. If we would allow unused code to appear in the graph, the graph would look like this:



This looks very ugly and confusing, while the purpose of a control flow graph is to make things clearer. So that is why we thought it was important to solve this bug. We gave each struct a boolean called *used*. The boolean of the another struct is set to true when the current struct gets a control flow arrow to that other struct. In

this example, the `bool` of the struct of "x++;" will not be set to true since there is no other node that creates an arrow to this struct. This also means that we do not look at "x++;" to see to what other nodes it should point to. If we would make "x++;" point to "System.out.println("Hallo!");", then both of the structs become used and will still appear. By skipping "x++;", the node "System.out.println("Hallo!");" is also skipped.
When converting the graph into DOT code, each struct with a boolean *used* set to *false* will also be skipped and therefore ommitted in the control flow graph. This will make the graph look like this:



So JShowFlow will omit nodes that are syntactically unreachable. However, JShowFlow is not capable of detecting nodes that are semantically unreachable. For example, consider this code:

```
1 if(x < 1 && x > 4){
2     x++;
3 }
```

The line "x++;" will never be executed, since x cannot be smaller than 1 while it is greater than 4 at the same time. Situations like this will become clear during execution, therefore it is impossible to detect semantically unreachable nodes by performing static analysis.

4.13.2 Statements spread over multiple lines

The disadvantage of reading the input code line by line is that it will lead to problems when a line of code is spread over multiple lines in the input document.

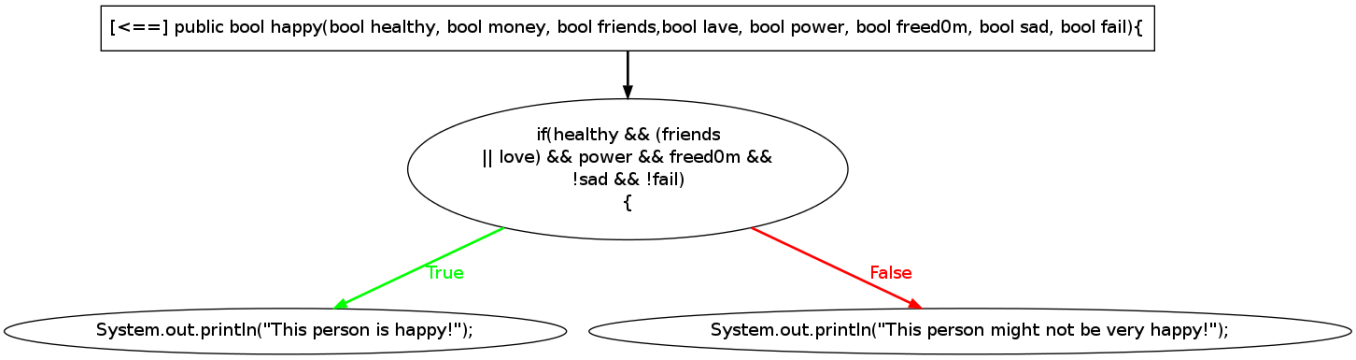
Example 4.12

```
1  public class ExampleClass {
2
3    public bool happy(bool healthy, bool money, bool friends,
4          bool lave, bool power, bool freed0m, bool sad, bool fail)
5    {
6      if(healthy && (friends
7      || love) && power && freed0m &&
8      !sad && !fail)
9      {
10       System.out.println("This person is happy!");
11     }
12     else
13     {
14       System.out.println("This person might not be very happy!");
15     }
16   }
17
18   public static void main (String[] args) {
19     happy(true, true, true, true, true,
20     true, false, false)
21     ;
22     System.out.println("End.");
23   }
24 }
```

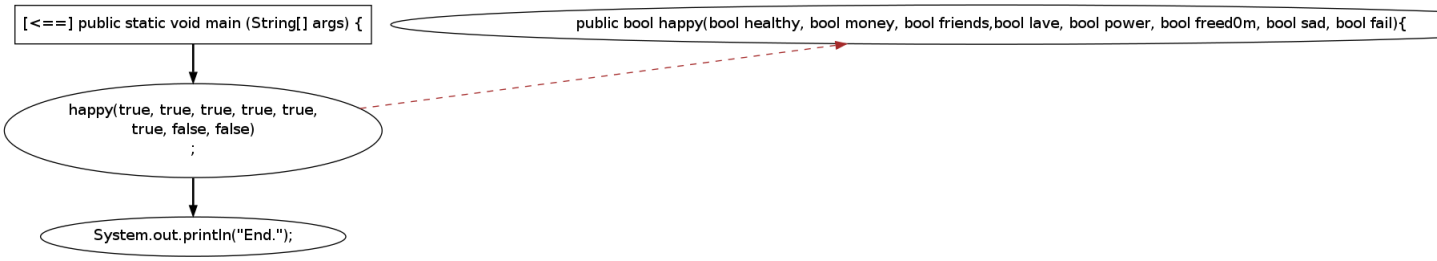
In this code, the method declaration of the method happy is spread over three lines. The {-bracket appears on the last of those three lines. When creating the class diagram, the type of each line is determined to understand whether a line is a class or method. The first line is not considered to be a method, but it will be saved. When the bracket is detected, the previous lines are analyzed to see whether they are a method if they are combined. If so, a node is created with all of those lines.

Inside the method occurs an `if` statement that is spread over four lines. A similar process happens there, except for the fact we already know that we are dealing with an `if` statement since the first line contains the word **if** with a '(' after it. All lines that follow will be collected and combined until and ')' and {-bracket are detected, or only a ')'-bracket. In the main method, the call to the method happy is also spread over three lines. Each line of which the type cannot be determined is considered to be of the same type as lines ending with a semicolon. All of those lines are collected and combined until a semicolon is detected.

The graph for the method happy looks likes this:



And the graph of the main method looks like this:



4.13.3 Comments and white lines in Java Code

Something that often appears in code is a comment. Obviously, we do not want the program process comments. For comments that start with a double backslash it is quite easy. When reading the input file line by line, we start with verifying that the line does not start with two blackslashes. If it does, the line should be ignored and we should immediately read the next line before doing anything else. But there is also another type of comment: a comment consisting of multiple lines. It start with the sign "/*" and ends with the "*/" sign. When one of these signs is detected, we know the line is a comment. But what about the lines in between? They do not have any of those signs and could look like regular Java code.

For this we use the variable **mlcomment**, which is set to true as soon as the "/*" sign is detected. When mlcomment is set to true, the program knows that we are currently dealing with comments and that everything should be ignored until the "*/" sign is detected, which indicates the end of the comment.

For white lines we use the same principle as for comments that start with the double backslash: if a line is found to be blank, it can be skipped such that the next line can be loaded before doing anything else.

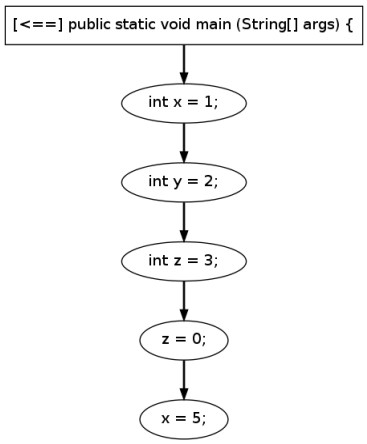
Example 4.13

The example code below contains the two types of comments and also whitelines:

```
1 //Below the variables are declared:
2 int x = 1;
3 int y = 2;
```

```
4  int z = 3;
5
6  /*
7  int a = -1;
8  int b = -2;
9  int c = -3;
10 */
11
12 /* If you would add y to x, you would get
13 x + y = 1 + 2 = 3, which is equal to z. So if
14 you would substract y from z, you will get
15 z - y = 3 - 2 = 1 - x */
16 z = 0;
17 x = 5;
18 //y = 9;
```

The graph looks just as expected. It only includes nodes of the lines that are meant to be included:



Chapter 5

Improvements

This whole bachelor project took a lot more time than we excepted. Since we wanted the program to be as much complete as possible, we spent a lot of time implementing features, studying different situations and solving bugs. This led to the current source code, that consists of nearly 3000 lines of code. But no matter how much time you spend on a project, there is always room for improvement. That is why there comes a point where you must decide to stop implementing new features and stop improving the code. This inevitably leads to some features not being implemented. Some of those features are discussed in this chapter.

5.1 Multiple instructions on one line

The program is capable of handling one instructions spread over multiple lines, but not the other way around: multiple instructions combined on one line.
Here are some example with explanations in the comments:

Example 5.1

```
1  //This will not be seen as two different lines:
2  int a; int b;
3
4  //Comments next to an instruction does not work:
5  int x = 0;  //this is a comment next to int x = 0;
6  int y = 0;  //the program thinks this is is still part of the instruction
7
8  /* JShowFlow program thinks that "x=y;" is still part
9  of this comment*/ x = y;
10
11 //These \verb|if| statements below do not work properly:
12 if(x == 3) y++;        //1) The program will see the line as
13                         //something like if(x == 3 || y++);
```

```
if(x == 3){ y++;       //2)
}
if(x == 3){            //3) This one causes a segmentation fault
  y++; }
if(x == 3){ y++; }     //4)
y++; if(x == 3){       //5) This one works, but y++; is being ignored.
}

/* The program can handle a while condition spread over multiple lines,
but not if it is part of a \verb|do-while| while one part appears on the same line as the
closing bracket of the do. This will not cause a segmentation fault, but
it will only show the "|| y == 0);" part if the comments behind it are removed.*/
do{
  x += 2;
} while(x < 10       //<== on the same line as the bracket
|| y == 0);          //<== multiple lines

//This causes a segmentation fault because the first child of each
//case is put on the same line as the case declaration.
switch(x){
  case 1: x++;
  case 2: x++;
          break;
  default: y--;
}
```

A way to solve this is to adjust the process of creating the code diagram. For example, we could include a few steps to recognize combined lines. Then we should split the line and turn them into different structs. This means that we should also include a few steps to decide where to split. The code above does not work properly, but there are some exceptions in which it does work:

Example 5.2

```
int x=0, int y=0;
if(x < 10){
  x = 0;
//The line below does work. Even if the
//second bracket is put a line below the \verb|else| if
} else if(x == 10) {
  y = 0;
//Works too:
} else
{
  y++;
}

do {
  x++;
```

```
//As we saw earlier, this works too:
} while(x < 10);

try{
  x++;
//No problems:
} catch(Exception E) {
  x--;
//This works too:
} finally {
  y = x;
}
```

These are all examples of a }-bracket followed by another line. We think that java code that looks this way is more common than other statements that are combined on one line. That is why we implemented a solution for these specific cases. In order to make the code of Example 5.1 work, this solution could be used as well when some adjustments are made to cover more cases.

5.2 Goto and labels

The goto statement is a statement that can make the program jump from one line to a completely different line, making it difficult to understand the control flow of the program. Therefore, its use is often considered bad practice. Currently the goto statement is not implemented in java, although it is a reserved keyword [Orab]. That is why we decided that this is less important to implement.

An alternative for the goto statements can be the `break` and `continue` statements, since they can jump to another line as well. Both are currently included in JShowFlow, but their features could be extended. In Java, `break` and `continue` statements can have labels [Oraa]. Here is an example:

Example 5.3

```
search:
for (i = 0; i < arrayOfInts.length; i++) {
    for (j = 0; j < arrayOfInts[i].length;
         j++) {
        if (arrayOfInts[i][j] == searchfor) {
            foundIt = true;
            break search;
        }
    }
}
x++;
```

The `break` statement terminates the labeled statement, which is the outer `for` loop. Control flow is transferred to the statement immediately following the labeled (terminated) statement, which is "x++;" in this case.
A labeled `continue` statement works the same way, but instead of breaking the outer `for` loop, it makes the outer `for` loop continue with the next cycle.
In the source code of JShowFlow, we could assign a type to lines ending with a colon. For example, the line "search:" in the code above could be of type 27. When a labeled `break` is detected during the control flow graph construction, the program should search for the corresponding label (which is a struct of type 27). If an instruction occurs after the item after that label (in this case: "x++;"), an arrow should be drawn from the `break` to that instruction. If it is a `continue`, an arrow should be drawn to the statement right after the label (in this case: the outer `for` loop).

5.3 Break and continue in try-catch-finally statement

When the `break` statement occurs inside the `try` or `catch` of a `try-catch-finally` statement that is part of loop, the `finally` block must still be executed. From each end point of the `finally` block, an arrow with the text "break;" should be drawn to the item after the `try-catch-finally` statement.

Example 5.4

```
int x=0, int y=0;
while(x < 10){
  try{
    break;
  }
  catch(Exception E){
    break;
  }
  finally{
    if(x == y){
      x = y;
    }
    else if(x < y){
      y = 9;
    }
    else {
      x = 9;
    }
  }
  x++;
}
y++;
```

Currently, the `break` skips the rest of the loop, including the `finally`. The `break` should point to the `finally`, and the lines "x=y;", "y = 9;" and "x = 9; should point to "y++;" while skipping the line "x++;". If this

`try-catch-finally` statement occurs inside another `try-catch-finally` statement that is also part of the loop, the `finally` block of that second `try-catch-finally` statement should also be executed. The same principle applies when these statements are part of a `switch` statement.
If a `continue` statement would occur in the `finally`, the `break` statement in the `try` would be overwritten by the `continue` statement in the `finally`, just like return statements can be overwritten when they occur inside a `try-catch-finally` statement.
For return statements inside `try-catch-finally` statements we already have an implemented solution. The way the `break` statement behaves in this case is similar to the behaviour of a return statement in this case, so we think that the same solution can be used to implement a solution for this problem. If the `break` statement would be replaced with a `continue` statement, the same principle still applies, with the only difference being that the lines "x=y;", "y = 9;" and "x = 9; should point back to the beginning of the loop.

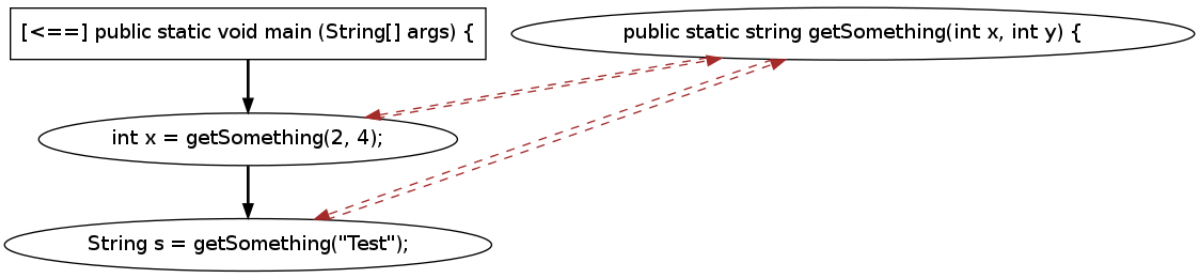
5.4 Method calls

5.4.1 Methods with the same name

If a call for a method is detected in a line of code, that call will be linked to the corresponding method. The way this is done is by searching the classdiagram for a method with a name that is the same as the method call. As we can see in chapter 4, this works very well. However, in some situations it is not enough to only look at the name of a method. Imagine that multiple methods have the same name, but different input variables. JShowFlow cannot distinguish between these methods and will pick the first one with the same name. This means that it might pick the wrong method. Here is an example in which it goes wrong:

```
1  public class AnotherClass {
2
3    public String getSomething (String name){
4       return "Hello";
5    }
6  }
7  public class ExampleClass {
8
9    public static void main (String[] args) {
10       int x = getSomething(2, 4);
11       String s = getSomething("Test");
12   }
13
14   public static string getSomething(int x, int y) {
15       return x+y;
16   }
17 }
```

The control flow graph of the main method looks like this:



There are two methods with the name "getSomething". Since one of them is in the same class as the main method, that method is the first detected method with that name and thus it will use that one. This is incorrect, since the second line in the main method refers to the getSomething method in AnotherClass. In this example, a solution could be checking the amount of input variables, since they differ in both methods. However, the amount of input variables could be equal in another situation in which only the type of input variables differs. This requires understanding the type of each variable, which is a very complicated process. For this, we need to construct a list of each variable and its type. This means we need to have a way of recognizing variable declarations. Keywords such as "int" and "String" can be recognized, but when the user has defined its own types for some variables, it is harder to recognize. We could write a regular expression that looks for something like two words with a space in between and a semicolon after the second word. Of course it is more complicated than that, but with some adjustments to that description of the regular expression, it must be possible to recognize declarations. Then we could create the variable list while reading the input code line by line. Using that list, we can get the type of each variable, which makes it possible to distinguish between two methods with the same name.

5.4.2 Inheritance and late bindings

Inheritance is currently not included in JShowFlow. This means that there is no check to see whether a method call is a call to a method in the current class, superclass or subclass. As explained earlier, JShowFlow always starts with finding a method with the corresponding name in the current class when linking a method call to a method. When dealing with inheritance, this is not always correct. For example, consider this code:

```
1  public class vehicle {
2
3    public int speed;
4
5    public vehicle() {
6      speed = 0;
7    }
8
9    public void speedUp() {
10     speed++;
11   }
12
13   public void brake() {
14     speed--;
```

```
15    }
16
17    public void stop(){
18      while(speed > 0){
19        this.brake();
20      }
21    }
22
23  }
24
25  class car extends vehicle {
26
27    public void speedUp() {
28      speed += 5;
29    }
30
31    public void brake() {
32      speed -= 5;
33    }
34
35  }
```

Car is a subclass of the class *vehicle* and it uses its own versions of the methods *speedUp()* and *brake()*. When the method *stop()* is called, the vehicle will brake until its speed equals 0.

Imagine that a new instance of the class *car* is created in the main method of another class:

```
1  public static void main(String[] args) {
2      car car1 = new car();
3      car1.speedUp();
4      car1.stop();
5  }
```

The method *stop()* is called on the last line in this main method. Since it is a *car*, the line "this.brake();" in the method *stop()* refers to the method *brake()* inside the subclass *car*. JShowFlow does not understand this correctly and will link "this.brake();" to the method *brake()* in the current class, which is the superclass *vehicle*. This is not correct in this situation, but it would be correct if an instance of the superclass is created without using the subclass. So unfortunately, there is no telling to what method "this.brake();" refers to just by looking at this code. This an example of a late binding, which is a binding that happens at runtime instead of compile time [jav]. Therefore, it cannot be included in a control flow graph the same way as the earlier examples of method calls.

However, instead of trying to link the call to one method, we could show a list of all possible candidates. The list could be put in a box that appears next to the graph and will be connected to the call with dashed arrows, just like regular method calls. Constructing this list requires finding all possible candidates. When creating the class diagram, each subclass should be included while having a special connection to the superclass. By doing so, the class diagram can be used to search for methods in subclasses to include in the list of possible candidates for a method call.

5.4.3 Other sources

While linking a method to a method call, only the class diagram is searched through. So only the content in input.java is checked. If a method is defined in something that is imported in input.java, the link cannot be made. To solve this, we should also search through the packages and other things that are imported into input.java to find the corresponding method.

5.5 Ternary expression with else if

Ternary expressions, such as "x = (y < 10)? 5 : 10;" and ternary returns such as "return (y < 10)? 5 : 10;" are both implemented. However, a ternary expression can also have some sort of `else if` part. The ternary expression "x = (y < 0)? 5 : (y < 10)? 10 : (y < 15)? 15 : 20;" can also be written as this code:

```
if(y < 0){
  x = 5;
}
else if(y < 10){
  x = 10;
}
else if(y < 15){
  x = 15;
}
else{
  x = 20;
}
```

Currently, the program can only handle ternary expressions with one ternary operator. Such a ternary expression is then translated to something that looks very similar to an `if` statement with an `else` block. To make more complicated ternary expressions possible, one could use a similar process while also including a way of translating the extra ternary operators into something that is similar to an `else if` statement as shown by the example above.

Chapter 6

Conclusions and Future Work

When you are a programmer, it is important to understand how your code behaves. When your program crashes or behaves differently than expected, it can lead to dangerous situations. Even when your program runs as expected, it could go wrong in situations you did not think of. Control flow graphs show all paths that might be traversed through a program during its execution, so they can be very useful for understanding how your code behaves. Drawing control flow graphs by hand can be time consuming and error-prone, which is why JShowFlow is needed. With JShowFlow, we can now generate control flow graphs for Java code as we saw in chapter 4, by using the techniques explained in chapter 2. By generating graphs that are minimalist while providing all necessary information, it can be a very handy tool for gaining insight in the behaviour of your Java program. It can also be a very useful tool for understanding someone else's code. The program does not need any other program such as Eclipse, so it is a very easy to use tool for every Java programmer. This is not the first control flow graph generator, but existing control flow graph generators described in chapter 1 do need another program and generate graphs that are not as clear as the ones we saw in chapter 4. There are some limitations to JShowFlow that are discussed in chapter 5. The proposed solutions in that chapter can be used for future work. Another idea for future work is creating control flow graph generators for other languages. With some small modifications the program could be used for C++ and other similar languages. Languages that have a different syntax (such as Python or Perl) require many more adjustments, but the same ideas can be used.

Bibliography

[All70] Frances E. Allen. Control flow analysis. 1970.

[cpl] cplusplus.com. Function strcmp. Retrieved from http://www.cplusplus.com/reference/cstring/strcmp/.

[Gra] Graphviz. The dot language. Retrieved from http://www.graphviz.org/content/dot-language.

[jav] javatpoint.com. Static binding and dynamic binding. Retrieved from https://www.javatpoint.com/static-binding-and-dynamic-binding/.

[Oraa] Oracle. Branching statements. Retrieved from https://docs.oracle.com/javase/tutorial/java/nutsandbolts/branch.html.

[Orab] Oracle. Java language keywords. Retrieved from http://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html.

[Wic95] A. A.; Clutterbuck D. L.; Winsbarrow L. A.; Ward N. J.; Marsh D. W. R. Wichmann, B. A.; Canning. Industrial Perspective on Static Analysis. 1995.