

5 The Control Structure Diagram (CSD)

The Control Structure Diagram (CSD) is an algorithmic level diagram intended to improve the comprehensibility of source code by clearly depicting control constructs, control paths, and the overall structure of each program unit. The CSD is an alternative to flow charts and other graphical representations of algorithms. The major goal behind its creation was that it be an intuitive and compact graphical notation that is easy to use manually and relatively straightforward to automate. The CSD is a natural extension to architectural diagrams, such as data flow diagrams, structure charts, module diagrams, and class diagrams.

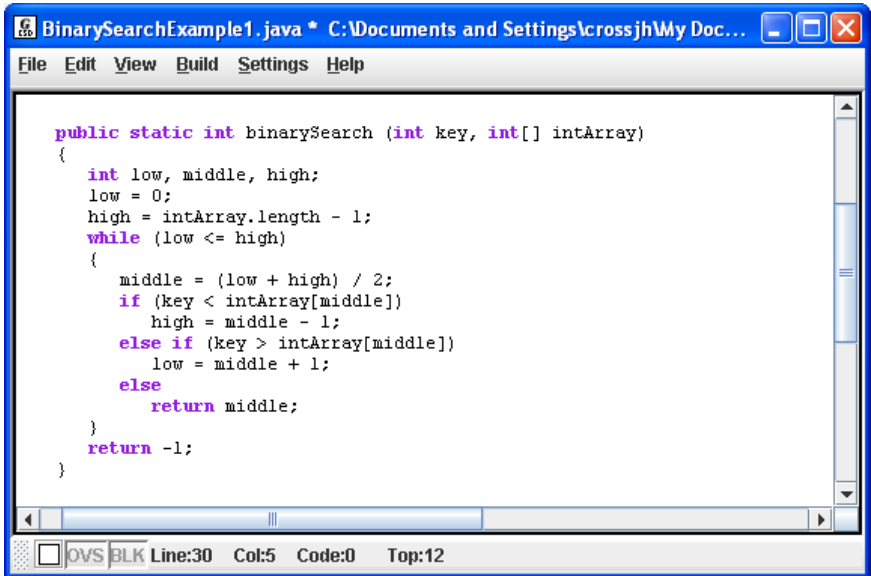
Objectives – When you have completed this tutorial, you should be able to use and understand the graphical notations used in the **CSD** for basic control constructs of modern programming languages, including sequence, selection, iteration, exits, and exception handling.

The details of these objectives are captured in the hyperlinked topics listed below.

- 5.1 An Example to Illustrate the CSD
- 5.2 CSD Program Components/Units
- 5.3 CSD Control Constructs
- 5.4 CSD Templates
- 5.5 Hints on Working with the CSD
- 5.6 Reading Source Code with the CSD
- 5.7 References

5.1 An Example to Illustrate the CSD

Figure 5-1 shows the source code for a Java method called `binarySearch`. The method implements a binary search algorithm by using a `while` loop with an `if..else..if` statement nested within the loop. Even though this is a simple method, displayed with colored keywords and traditional indentation, its readability can be improved by adding the CSD. In addition to the `while` and `if` statements, we see that the method includes the declaration of primitive data (`int`) and two points of exit. The CSD provides visual cues for each of these constructs.



```

public static int binarySearch (int key, int[] intArray)
{
    int low, middle, high;
    low = 0;
    high = intArray.length - 1;
    while (low <= high)
    {
        middle = (low + high) / 2;
        if (key < intArray[middle])
            high = middle - 1;
        else if (key > intArray[middle])
            low = middle + 1;
        else
            return middle;
    }
    return -1;
}

```

Figure 5-1. `binarySearch` method without CSD

Figure 5-2 shows the `binarySearch` method after the CSD has been generated. Although all necessary control information is in the source text, the CSD provides additional visual cues by highlighting the sequence, selection, and iteration in the code. The CSD notation begins with symbol for the method itself followed by the individual statements branching off the stem as it extends downward. The declaration of primitive data is highlighted with the symbol appended to the statement stem. The CSD construct for the `while` statement is represented by the double line “loop” (with break at the top), and

the *if* statement uses the familiar diamond symbol from traditional flowcharts. Finally, the two ways to exit from this method are shown explicitly with an arrow drawn from inside the method through the method stem to the outside.

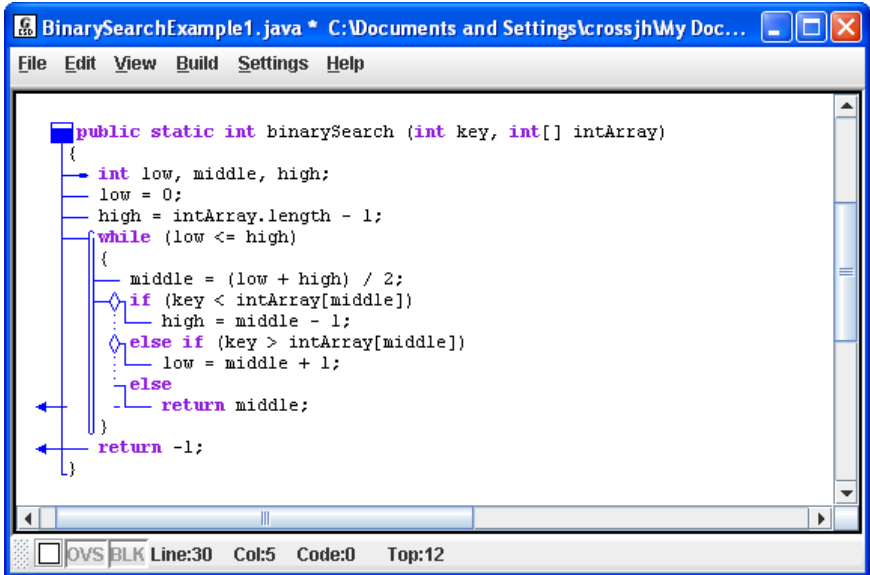

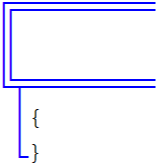
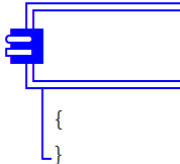

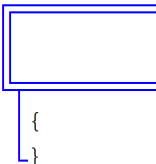
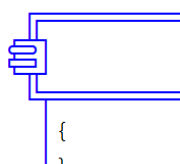
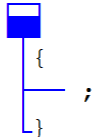
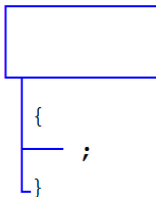
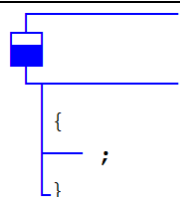


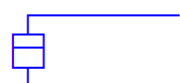


Figure 5-2. binarySearch with CSD

While this is a small piece of code, it does illustrate the basic CSD constructs. However, the true utility of the CSD can be realized best when reading or writing larger, more complex programs, especially when control constructs become deeply nested. A number of studies involving the CSD have been done and others are in progress. In one of these, the CSD was shown to be preferred significantly over four other notations: flowchart, Nasi-Schneiderman chart, Warnier-Orr diagram, and the action diagram [Cross 1998]. In a several later studies, empirical experiments were done in which source code with the CSD was compared to source code without the CSD. In each of these studies, the CSD was shown to provide significant advantages in numerous code reading activities [Hendrix 2002]. In the following sections, the CSD notation is described in more detail.

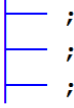
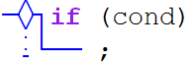
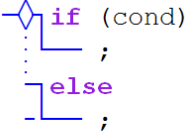
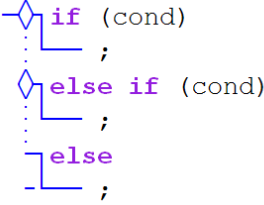
5.2 CSD Program Components/Units

The CSD includes graphical constructs for the following components or program units: class, abstract class, method, and abstract method. The construct for each component includes a unit symbol, a box notation, and a combination of the symbol and box notation. The symbol notation provides a visual cue as to the specific type of program component. It has the most compact vertical spacing in that it retains the line spacing of source code without the CSD. The box notation provides a useful amount of vertical separation similar to skipping lines between components. The symbol and box notation is simply a combination of the first two. The symbol and box notation is simply a combination of the first two. Most of the examples in this handbook use the symbol notation because of its compactness. CSD notation for program components is illustrated in the table below.

Component	Symbol Notation	Box Notation	Symbol and Box Notation
class or Ada package			
abstract class			
method or function or procedure			
abstract method			

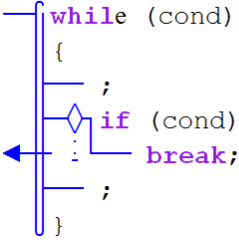
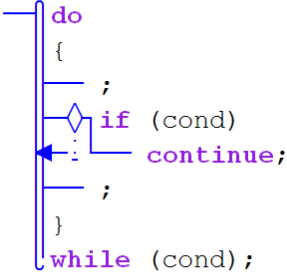
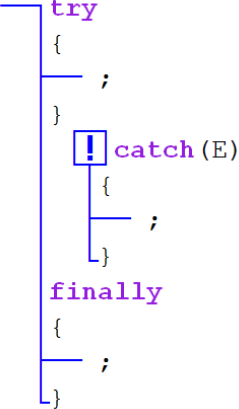
5.3 CSD Control Constructs

The basic CSD control constructs for Java are grouped in the following categories: sequence, selection, iteration, and exception handling, as described in the table below. The semi-colons in the examples are placeholders for statements in the language.

Sequence		Sequential flow is represented in the CSD by a vertical stem with a small horizontal stem for each individual statement on a particular level of control.
<p>Selection</p> <p>if</p> <p>if..else</p> <p>if..else..if</p>	  	<p>For selection statements, the True/False condition itself is marked with a small diamond, just as in a flow chart. The statements to be executed if the condition is true are marked by a solid line leading from the right of the decision diamond.</p> <p>The control path for a false condition is marked with a dotted line leading from the bottom of the diamond to another decision diamond, an else clause, a default clause, or the end of the decision statement.</p> <p>By placing the second <i>if</i> on the same line with the first <i>else</i>, the unnecessary indentation of nested <i>if</i> statements is avoided. However, if the deep nesting effect is desired, the second <i>if</i> can be placed on the line after the else.</p>

<p>Selection (cont'd)</p> <p>switch</p>	<pre> switch (item) { case a: ; break; case b: ; break; default: ; } </pre>	<p>The semantics of the <i>switch</i> statement are different from those of <i>if</i> statements. The <i>expression</i> (of integral or enum type) is evaluated, and then control is transferred to the case label matching the result or to the default label if there is no match. If a <i>break</i> statement is placed at the end of the sequence within a case, control passes “out” (as indicated by the arrow) and to the end of the <i>switch</i> statement after the sequence is executed. Notice the similarity of the CSD notation for the <i>switch</i> and <i>if</i> statements when the <i>break</i> is used in this conventional way. The reason for this is that, although different semantically, we humans tend to process them the same way (e.g., if expr is not equal to case 1, then take the false path to case 2 and see if they are equal, and so on). However, the <i>break</i> statement can be omitted as illustrated next.</p>
<p>switch</p> <p>when break is omitted</p>	<pre> switch (expr) { case 1: ; case 2: ; case 3: ; case 4: ; } </pre>	<p>When the break statement is omitted from end of the sequence within a case, control falls through to the next case. In the example at left, case 1 has a <i>break</i> statement at the end of its sequence, which will pass control to the end of the switch (as indicated by the arrow). However, case 2, case 3, and case 4 do not use the <i>break</i> statement. The CSD notation clearly indicates that once the flow of control reaches case 2, it will also execute the sequences in case 3 and case 4.</p>

		<p>The diamonds in front of case 3 and case 4 have arrows pointing to each case to remind the user that these are entry points for the switch. When the break statement precedes the next case (as in case 1), the arrows are unnecessary.</p>
<p>Iteration</p> <p>while loop (pre-test)</p> <p>for loop (discrete)</p> <p>do loop (post-test)</p>	<pre> while (cond) { ; } for (i=0; i<j; i++) { ; } do { ; } while (cond); </pre>	<p>The CSD notation for the while statement is a loop construct represented by the double line, which is continuous except for the small gap on the line with the while. The gap indicates the control flow can exit the loop at that point or continue, depending on the value of the boolean condition. The sequence within the while will be executed zero or more times.</p> <p>The for statement is represented in a similar way. The for statement is designed to iterate a discrete number of times based on an index, test expression, and index increment. In the example at left, the for index is initialized to 0, the condition is $i < j$, and the index increment is $i++$. The sequence within the for will be executed zero or more times.</p> <p>The do statement is similar to the while except that the loop condition is at the end of the loop instead of the beginning. Thus, the body of the loop is guaranteed to execute at least once.</p>

<p>break in loop</p>	 <pre> while (cond) { ; if (cond) break; ; } </pre>	<p>The break statement can be used to transfer control flow out of any loop (while, for, do) body, as indicated by the arrow, and down to the statement past the end of the loop. Typically, this would be done in conjunction with an if statement. If the break is used alone (e.g., without the if statement), the statements in the loop body beyond the break will never be executed.</p>
<p>Iteration (cont'd)</p> <p>continue</p>	 <pre> do { ; if (cond) continue; ; } while (cond); </pre>	<p>The continue statement is similar to the break statement, but the loop condition is evaluated and if true, the body of the loop body is executed again. Hence, as indicated by the arrow, control is not transferred out of the loop, but rather to top or bottom of the loop (while, for, do).</p>
<p>Exception Handling</p>	 <pre> try { ; } catch (E) { ; } finally { ; } </pre>	<p>In Java, the control construct for exception handling is the try..catch statement with optional finally clause. In the example at left, if stmt1 generates an exception E, then control is transferred to the corresponding catch clause. After the catch body is executed, the finally clause (if present) is executed. If no exception occurs in the try block, when it completes, the finally clause (if present) is executed.</p>

<p>With a return</p>	<pre> try { ; ; return; } ! catch (E) { ; } finally { ; } </pre>	<p>The <i>try..catch</i> statement can have multiple <i>catch</i> clauses, one for each exception to be handled.</p> <p>By definition, the <i>finally</i> clause is always executed no matter how the <i>try</i> block is exited. In the example at left, a <i>return</i> statement causes flow of control to leave the try block. The CSD indicates that flow of control passes to the finally clause, which is executed prior to leaving the <i>try</i> block. The CSD uses this same convention for <i>break</i> and <i>continue</i> when these cause a <i>try</i> block to be exited.</p> <p>When try blocks are nested and <i>break</i>, <i>continue</i>, and <i>return</i> statements occur at the different levels of the nesting, the actual control flow can become quite counterintuitive. The CSD can be used to clarify the control flow.</p>
-----------------------------	--	---

5.4 CSD Templates

In Figure 5-3, the basic CSD control constructs, described above, are shown in the CSD window. These are generated automatically based on the text in the window. In addition to being typed or read from a file, the text can be inserted from a list of templates by selecting **Templates** on the CSD window tool bar.

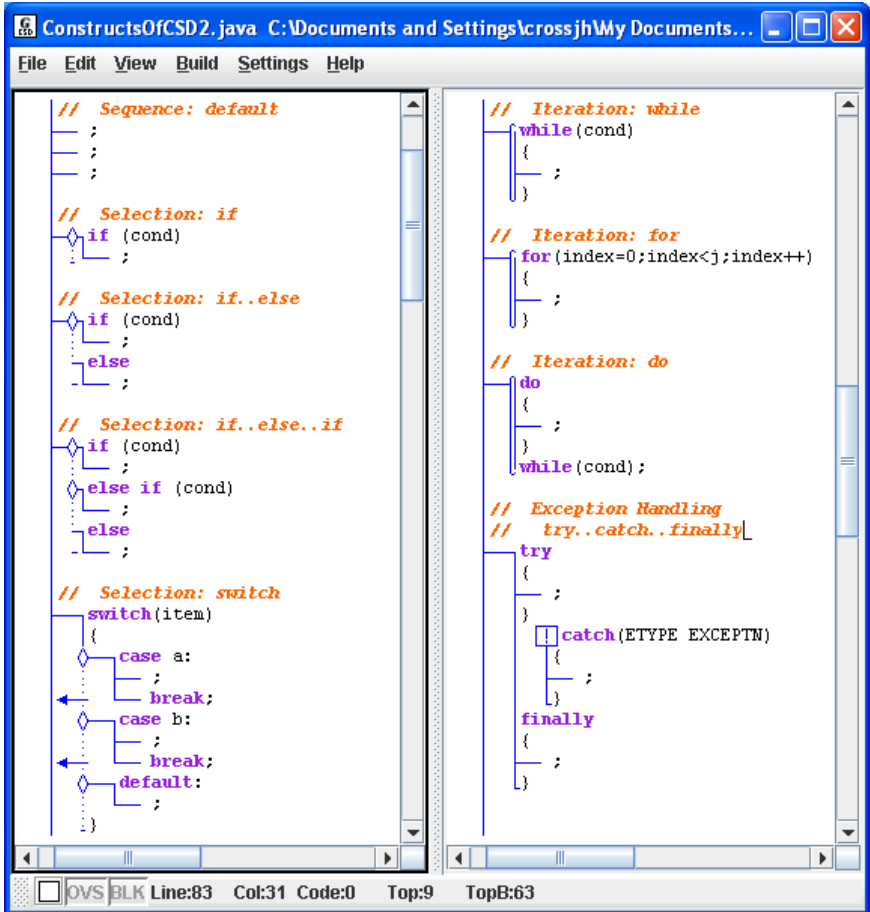



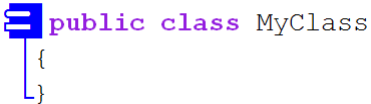
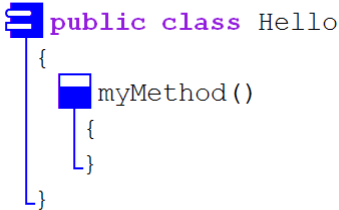
Figure 5-3. CSD Control Constructs generated in CSD Window

5.5 Hints on Working with the CSD

The CSD is generated based on the source code text in the CSD window. When you click **View > Generate CSD** (or press **F2**), jGRASP parses the source code based on a grammar or syntax that is slightly more forgiving than the Java compiler. If your program will compile successfully, the CSD should generate successfully as well. However, the CSD may generate successfully even if your program will not compile. Your program may be syntactically correct, but not necessarily semantically correct. For the most part, CSD generation is based on the syntax of your program only.

Enter code in syntactically correct chunks - To reap the most benefit from using the CSD when entering a program, you should take care to enter code in syntactically correct chunks, and then regenerate the CSD often. If an error is reported, it should be fixed before you move on. If the error message from the *generate* step is not sufficient to understand the problem, compile your program and you will get a more complete error message.

“**Growing a program**” is described in the table below. Although the program being “grown” does nothing useful, it is both syntactically and semantically correct. More importantly, it illustrates the incremental steps that should be used to write your programs. After the code is entered in each step, click the Generate CSD button  or press F2 to generate the CSD.

Step	Code to Enter	After CSD is generated
1	<pre>public class MyClass { }</pre>	
2	<pre>public class MyClass { myMethod() { } }</pre>	

3	<pre> public class MyClass { myMethod() { while (true) { ; } } } </pre>	
---	---	--

5.6 Reading Source Code with the CSD

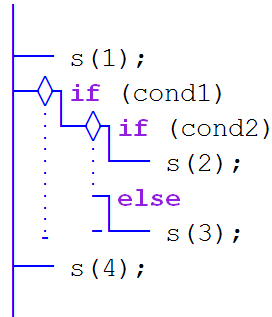
The CSD notation for each of the control constructs has been carefully designed to aid in reading and scanning source code. While the notation is meant to be intuitive, there are several reading strategies worth pointing out, especially for deeply nested code.

<p>Reading Sequence</p> <p>The visualization of sequential control flow is as follows. After statement s(1) is executed, the next statement is found by scanning down and to the left along the solid CSD stem. While this seems trivial, its importance becomes clearer with the <i>if</i> statement and deeper nesting.</p>	
<p>Reading Selection</p> <p>Now combining the <i>sequence</i> with <i>selection (if.. else)</i>, after s(1), we enter the <i>if</i> statement marked by the diamond. If the condition is <i>true</i>, we follow the solid line to s(2). After s(2), we read down and to the left (passing through the dotted line) until we reach the next statement on the vertical stem</p>	

which is s(4). If the condition is *false*, we read down the dotted line (the false path) to the *else* and then on to s(3). After s(3), again we read down and to the left until we reach the next statement on the stem which is s(4).

Reading Selection with Nesting

As above, after s(1), we enter the if statement and if cond1 and cond2 are true, we follow the solid lines to s(2). After s(2), we read down and to the left (passing through both dotted lines) until we reach the next statement on the stem which is s(4). If cond1 is false, we read down the dotted line (the false path) to s(4). If cond2 is false, we read down the dotted line to the else and then on to s(3). After s(3), again we read down and to the left until we reach to the next statement on the stem which is s(4).



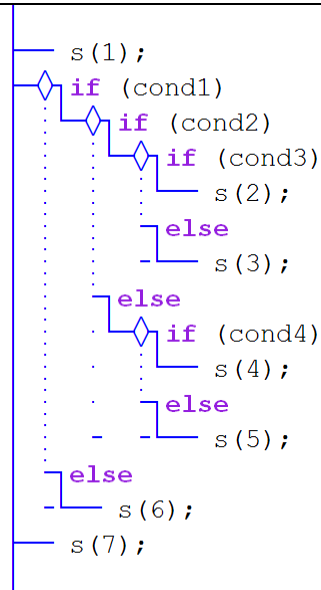
Reading Selection with Even Deeper Nesting

If cond1, cond2, and cond3 are true, we follow the solid lines to s(2). Using the strategy above, we immediately see the next statement to be executed will be s(7).

If cond1 is true but cond2 is false, we can easily follow the flow to either s(4) or s(5) depending on cond4.

If s(4) is executed, we can see immediately that s(7) follows.

In fact, from any statement, regardless of the level of nesting, the CSD makes it easy to see which statement is executed next.



Reading without the CSD

It should be clear from the code at right that following the flow of control without the CSD is somewhat more difficult.

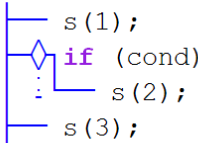
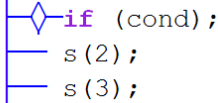
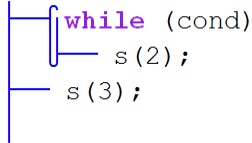
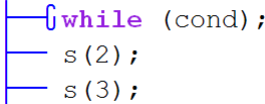
For example, after s(3) is executed, s(7) is next. With the CSD in the previous example, the reader can tell this at a glance. However, without the CSD, the reader may have to read and reread to ensure that he/she is seeing the indentation correctly.

While this is a simple example, as the nesting becomes deeper, the CSD becomes even more useful.

In addition to saving time in the reading process, the CSD aids in interpreting the source code correctly, as seen in the examples that follow.

```
s(1);  
if (cond1)  
    if (cond2)  
        if (cond3)  
            s(2);  
        else  
            s(3);  
    else  
        if (cond4)  
            s(4);  
        else  
            s(5);  
else  
    s(6);  
s(7);
```

Reading Correctly with the CSD

<p>Consider the fragment at right with $s(1)$ and $s(2)$ in the body of the <i>if</i> statement.</p> <p>After the CSD is generated, the reader can see how the compiler will interpret the code, and add the missing braces.</p>	<pre>s(1); if (cond) s(2); s(3);</pre> 
<p>Here is another common mistake made obvious by the CSD.</p> <p>The semi-colon after the condition was almost certainly unintended. However, the CSD shows what is there rather than what was intended.</p>	<pre>if (cond); s(2); s(3);</pre> 
<p>Similarly, the CSD provides the correct interpretation of the <i>while</i> statement.</p> <p>Missing braces . . .</p>	<pre>while (cond) s(2); s(3);</pre> 
<p>Similarly, the CSD provides the correct interpretation of the <i>while</i> statement.</p> <p>Unintended semi-colon . . .</p>	<pre>while (cond); s(2); s(3);</pre> 

As a final example of reading source code with the CSD, consider the following program, which is shown with and without the CSD. *FinallyTest* illustrates control flow when a *break*, *continue*, and *return* are used within *try* blocks that each have a *finally* clause. Although the flow of control may seem somewhat counterintuitive, the CSD should make it easier to interpret this source code correctly. First read the source code without the CSD. Recall that by definition, the *finally* clause is always executed not matter how the *try* block is exited.

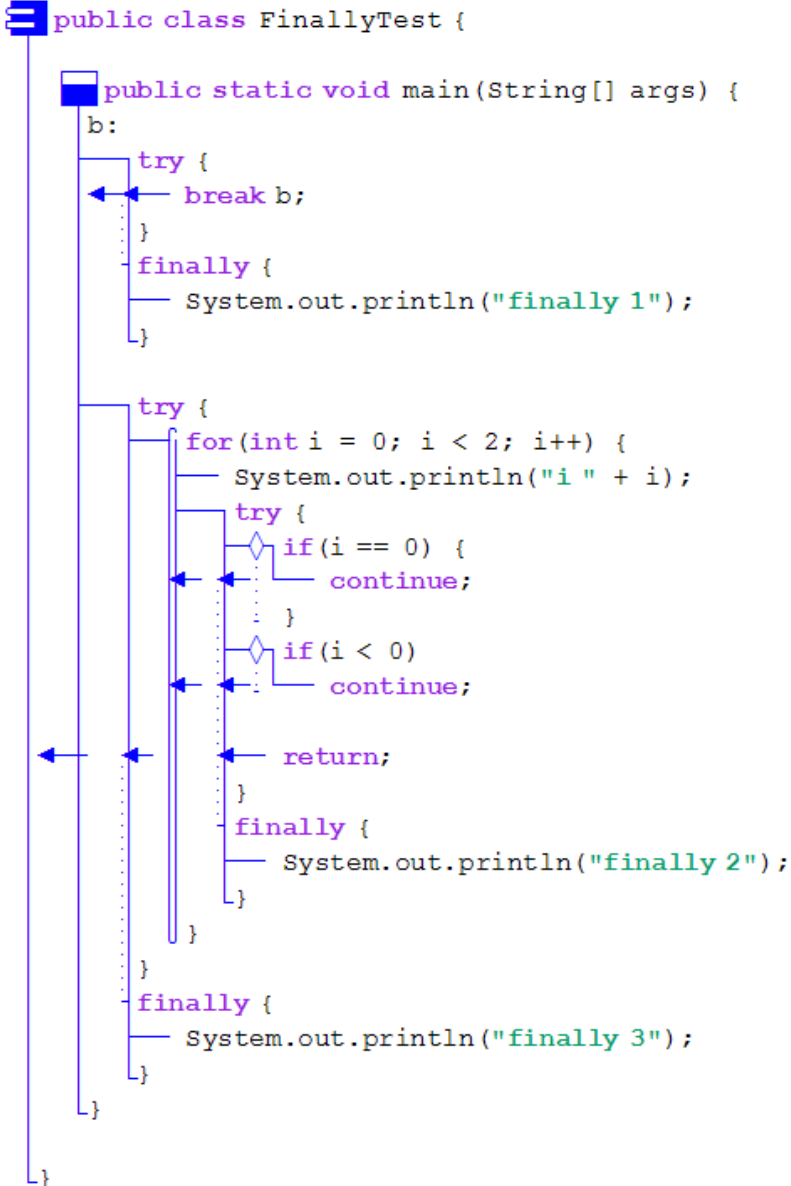
Try-Finally with *break*, *continue*, and *return* statements with no CSD

```
public class FinallyTest {

    public static void main(String[] args) {
    b:
        try {
            break b;
        }
        finally {
            System.out.println("finally 1");
        }

        try {
            for(int i = 0; i < 2; i++) {
                System.out.println("i " + i);
                try {
                    if(i == 0) {
                        continue;
                    }
                    if(i < 0)
                        continue;

                    return;
                }
                finally {
                    System.out.println("finally 2");
                }
            }
        }
        finally {
            System.out.println("finally 3");
        }
    }
};
```


Try-Finally with break, continue, and return statements with CSD

In our experience, this code is often misinterpreted when read without the CSD, but understood correctly when read with the CSD. Refer to the output if you need a hint. The output for *FinallyTest* is as follows:

```
finally 1
i 0
finally 2
i 1
finally 2
finally 3
```

5.7 References

[Cross 1998] J. H. Cross, S. Maghsoodloo, and T. D. Hendrix, "Control Structure Diagrams: Overview and Initial Evaluation," *Journal of Empirical Software Engineering*, Vol. 3, No. 2, 1998, 131-158.

[Hendrix 2002] T. D. Hendrix, J. H. Cross, S. Maghsoodloo, and K. H. Chang, "Empirically Evaluating Scaleable Software Visualizations: An Experimental Framework," *IEEE Transactions on Software Engineering*, Vol. 28, No. 5, May 2002, 463-477.