



# Scala CodeCount™

## Counting Standard

*University of Southern California*

**Center for Systems and Software Engineering**

December , 2016

**Revision Sheet**

Date	Version	Revision Description	Author
2/18/2016	1.0	Original Release	Derek Lengenfelder

# Table of Contents

No.	Contents	Page No.
1.0	<a href="#">Definitions</a>	4
1.1	<a href="#">Overview</a>	4
1.2	<a href="#">SLOC</a>	4
1.3	<a href="#">Physical SLOC</a>	4
1.4	<a href="#">Logical SLOC</a>	4
1.5	<a href="#">Data declaration line</a>	4
1.6	<a href="#">Compiler directive</a>	4
1.7	<a href="#">Executable keywords</a>	5
1.8	<a href="#">Blank line</a>	5
1.9	<a href="#">Comment line</a>	5
1.10	<a href="#">Executable line of code</a>	5
2.0	<a href="#">Checklist for source statement counts</a>	6
3.0	<a href="#">Examples of logical SLOC counting</a>	7
3.1	<a href="#">Executable Lines</a>	7
3.1.1	<a href="#">Selection Statements</a>	7
3.1.2	<a href="#">Iteration Statements</a>	8
3.1.3	<a href="#">Jump Statements</a>	8
3.1.4	<a href="#">Expression Statements</a>	9
3.1.5	<a href="#">Block Statements</a>	9
3.2	<a href="#">Declaration lines</a>	9
3.3	<a href="#">Compiler directives</a>	9
4.0	<a href="#">Cyclomatic Complexity</a>	10

# 1. Definitions

- 1.1. **Overview** – Unified Code Count (UCC) has a Procedural Programming perspective when looking at languages and source files, rather than an Object Oriented or function Programming point of view that Scala supports. Although this difference may cause a “disconnect” from a Scala developer’s point of view, it is hope that the metrics gathered by UCC will be of use to the Scala community.
- 1.2. **SLOC** – Source Lines of Code is a unit used to measure the size of software program. SLOC counts the program source code based on a certain set of rules. SLOC is a key input for estimating project effort and is also used to calculate productivity and other measurements.
- 1.3. **Physical SLOC** – One physical SLOC is corresponding to one line starting with the first character and ending by a carriage return or an end-of-file marker of the same line, and which excludes the blank and comment line.
- 1.4. **Logical SLOC** – Lines of code intended to measure “statements”, which normally terminate by a semicolon (C/C++, Java, C#) or a carriage return (VB, Assembly), etc. Logical SLOC are not sensitive to format and style conventions, but they are language-dependent. A Scala interpreter/compiler infers line endings without requiring the source code to have very many semicolons. A semicolon is needed to separate 2 or more logical source statements on the same physical line, but a line with only 1 logical statement does not require a semicolon. Hence, UCC gives an approximation for logical SLOC that might not exactly match what is expected.
- 1.5. **Data declaration line or data line** – A line that contains declaration of data and used by an assembler or compiler to interpret other elements of the program.

The following table lists the Scala keywords that denote data declaration lines:

abstract	Array	boolean	Byte	Char
Class	Double	Extends	Float	HashMap
HashSet	implements	int	LinkedHashMap	LinkedList
Long	object	override	private	protected
sealed	Short	static	String	TreeMap
Val	var	Vector		

**Table 1 Data Declaration Types**

- 1.6. **Compiler Directives** – A statement that tells the compiler how to compile a program, but not what to compile.

The following table lists the Scala keywords that denote data declaration lines:

package	import
---------	--------

**Table 2 Compiler Directives**

- 1.7. **Executable Keywords** – Scala keywords are reserved with predefined characteristics as far as syntax and meanings (semantic or otherwise) to enable various Scala language specific features.

The following table lists the Scala executable keywords:

break	case	catch	def	do
else	finally	for	if	match
new	return	super	this	throw
try	while			

**Table 3 Executable Keywords**

- 1.8. **Blank Line** – A physical line of code, which contains any number of white space characters (spaces, tabs, form feed, carriage return, line feed, or their derivatives).
- 1.9. **Comment Line** – A comment is defined as a string of zero or more characters that follow language-specific comment delimiter.

Scala comment delimiters are “//” for a single line (until the end of the line) and “/\*” for a possible multiline comment. A whole comment line may span one line and does not contain any compilable source code. An embedded comment can co-exist with compilable source code on the same physical line. Banners and empty comments are treated as types of comments. Scala allows nesting of multiline comments to any arbitrary depth, for example:

```
/* This is an outer block comment
/* this is an inner block comment */
now ending the outer block */
```

- 1.10. **Executable Line of code** – A line that contains software instruction executed during runtime and on which a breakpoint can be set in a debugging tool. An instruction can be stated in a simple or compound form.
- An executable line of code may contain the following program control statements:
    - Selection statements (if, ? operator, switch)
    - Iteration statements (for, while, do-while)
    - Empty statements (one or more “;”)
    - Jump statements (return, goto, break, continue, exit function)
    - Expression statements (function calls, assignment statements, operations, etc.)
    - Block statements
  - An executable line of code may not contain the following statements:
    - Compiler directives
    - Data declaration (data) lines
    - Whole line comments, including empty comments and banners
    - Blank lines

## 2. Checklist for source statement counts

<u>PHYSICAL SLOC COUNTING RULES</u>			
MEASUREMENT UNIT	ORDER OF PRECEDENCE	PHYSICAL SLOC	COMMENTS
<b>Executable Lines</b>	1	One per line	Defined in 1.10 and Keywords in 1.7
<b>Non-executable Lines</b>			
Declaration (Data) Lines	2	One per line	Defined in 1.5
Compiler Directives	3	One per line	Defined in 1.6
Comments			Defined in 1.9
On their own lines	4	Not Included	
Embedded	5	Not Included	
Banners	6	Not included	
Empty Comments	7	Not included	
Blank lines	8	Not Included	Defined in 1.8

<u>LOGICAL SLOC COUNTING RULES</u>				
NO.	STRUCTURE	ORDER OF PRECEDENCE	LOGICAL SLOC RULES	COMMENTS
R01	"for", "while", "match" or "if" statement	1	Count once	"while" is an independent statement.
R02	do {...} while (...); statement	2	Count once	Braces {...} and semicolon ; used with this statement are not counted.
R03	Statements ending by a semicolon	3	Count once per statement, including empty statement	Semicolons within "for" statement are not counted. Semicolons used with R01 and R02 are not counted.
R04	Block delimiters, braces {...}	4	Count once per pair of braces {...}, except where a closing brace is followed by a semicolon, i.e. }; or an opening brace comes after a keyword "else".	Braces used with R01 and R02 are not counted. Function definition is counted since it is followed by {...}.
R05	Compiler directive	5	Count once per directive	

### 3. Examples

#### EXECUTABLE LINES

#### SELECTION Statements

##### ESS1 – if-else if-else and nested if statements

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
<pre>if (&lt;boolean expression&gt;) {     &lt;statements&gt;; } else if (&lt;boolean expression&gt;) {     &lt;statements&gt;; } ... else {     &lt;statements&gt;; }</pre>	<pre>if (x == 0) {     println ("zero"); } else if (x &gt; 0) {     println ("positive"); } else {     println ("negative"); }</pre>	<pre>1 0 1 0 1 0 1 0 0 1 0</pre>
NOTE: complexity is not considered, i.e. multiple "&&" or "  " as part of the expression.	<pre>if ((x != 0) &amp;&amp; (x &gt; 0)) {     println (s"\$x"); }</pre>	<pre>1 0 1 0</pre>

##### ESS2 – switch and nested switch statements

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
<pre>&lt;matchvar&gt; match {     case &lt;value&gt; =&gt; &lt;expression&gt;     case &lt;value&gt; =&gt; &lt;expression&gt;     case &lt;value&gt; =&gt; &lt;expression&gt;     ...     case &lt;value&gt; =&gt; &lt;expression&gt;     case &lt;value&gt; =&gt; &lt;expression&gt;     case &lt;value&gt; =&gt; &lt;expression&gt; }</pre>	<pre>i match {     case 1 =&gt; println("January")     case 2 =&gt; println("February")     case 3 =&gt; println("March")     case 4 =&gt; println("April")     case 5 =&gt; println("May")     case 6 =&gt; println("June")     case 7 =&gt; println("July")     case 8 =&gt; println("August")     case 9 =&gt; println("September")     case 10 =&gt; println("October")     case 11 =&gt; println("November")     case 12 =&gt; println("December")     // catch default with a variable     case whoa =&gt;         println("Unexpected case: " +             whoa.toString) }</pre>	<pre>1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0</pre>

**ESS3 – try-catch blocks**

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
try { <statements> }	try { // Scala Code }	0 0 0 0
catch { case <label>: Exception => exceptionHandler() ... case <labelN>: ExceptionN => exceptionHandlerN() }	catch { case foo: FooException => handleFooException(foo) case bar: BarException => handleBarException(bar) case _: Throwable => println("Got some other kind of exception") }	0 0 1 1 1 1
finally { <statements> }	finally { println("Clean up") }	0 0 1 0

**ITERATION Statements****EIS1 – For Loop**

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
for (i <- <num1> until <num2>) { <statements> }	for (i <- 1 until 7) { println (s"\$i") }	1 0 1 0

**EIS2 – While Loop**

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
while (<boolean expression>) <statement>;	while (i < 7) { println (s"\$i") i += 1; }	1 0 1 1 0

**EIS3 – do-while Loop**

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
do { <statements>; } while (<boolean expression>;)	do { println (s"\$i") i += 1 } while (i != 7);	0 0 1 1 1

**JUMP Statements**

(are counted as they invoke action-pass to the next statement)

**EJS1 - return**

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
return expression	return i;	1



**EXPRESSION Statements****EES1 – function and procedure call**

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
<function_name>(<parameters>)	do_something(p1, p2)	1

**EES2 – assignment statement**

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
[val   var] <name>: <Type> =	var x = y;	1
<value>;	val name = "file1";	1
	var a = 1; var b = 2; var c = 3;	3

**BLOCK Statements****EBS1 – simple block (related statements treated as a unit)**

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
/* start of block */	/* start of block */	0
{	{	0
<statements>	println ("Hello World!");	1
}	}	0
/* end of block */	/* end of block */	0

**DECLARATION OR DATA LINES****DDL1 – function prototype, variable declaration**

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
<type> <name> ( < parameter_list>	private static void foo (int param);	1
);	double amount;	1
<type> <name>;	Iterator<String>	1
Class<T>		

**COMPILER DIRECTIVES****CDL1 – directive types**

GENERAL EXAMPLE	SPECIFIC EXAMPLE	SLOC COUNT
package <package_name>	package test	1
import <package_name>	import scala.math._	1

## 4. Cyclomatic Complexity

Cyclomatic complexity measures the number of linearly independent paths through a program. It is measured for each function, procedure, or method according to each specific program language. This metric indicates the risk of program complexity and also determines the number of independent tests required to verify program coverage.

The cyclomatic complexity is computed by counting the number of decisions plus one for the linear path. Decisions are determined by the number of conditional statements in a function. A function without any decisions would have a cyclomatic complexity of one. Each decision such as an if condition or a for loop adds one to the cyclomatic complexity.

The cyclomatic complexity metric  $v(G)$  was defined by Thomas McCabe. Several variations are commonly used but are not included in UCC. The modified cyclomatic complexity counts select blocks as a single decision rather than counting each case. The strict or extended cyclomatic complexity includes Boolean operators within conditional statements as additional decisions. Please see: [cyclomatic\\_complexity\\_standard.pdf](#) which has more details of different ways where specific cyclomatic complexity metrics are found and presented.

Cyclomatic Complexity	Risk Evaluation
1-10	A simple program, without much risk
11-20	More complex, moderate risk
21-50	Complex, high risk program
>50	Untestable program, very high risk

For Scala, the following table lists the conditional keywords used to compute cyclomatic complexity.

Scala statement	CC count	Rationale
if	+1	if adds a decision
else if	+1	else if adds a decision
else	0	decision is at the if statement
match/case	+1 per case	each case adds a decision - not the match
for	+1	for adds a decision at loop start
while	+1	while adds a decision at loop start or at the end of do loop
do	0	decision is at while statement - no decision at unconditional loop
try	0	decision is at catch statement
catch	+1	catch adds a decision