[Debug your code with System.debug!](http://www.sfdc99.com/2014/02/22/debug-your-code-with-system-debug/)

System.debug() simply “prints” the value of anything you put inside it!

trigger **CountFriends** on Contact(before insert) {

for (Contact **c** : Trigger.new) {

// We often use System.debug to see the values of variables

System.debug('David has a new friend!');

System.debug('Hi ' + **c**.FirstName + ' ' + **c**.LastName + '!');

}

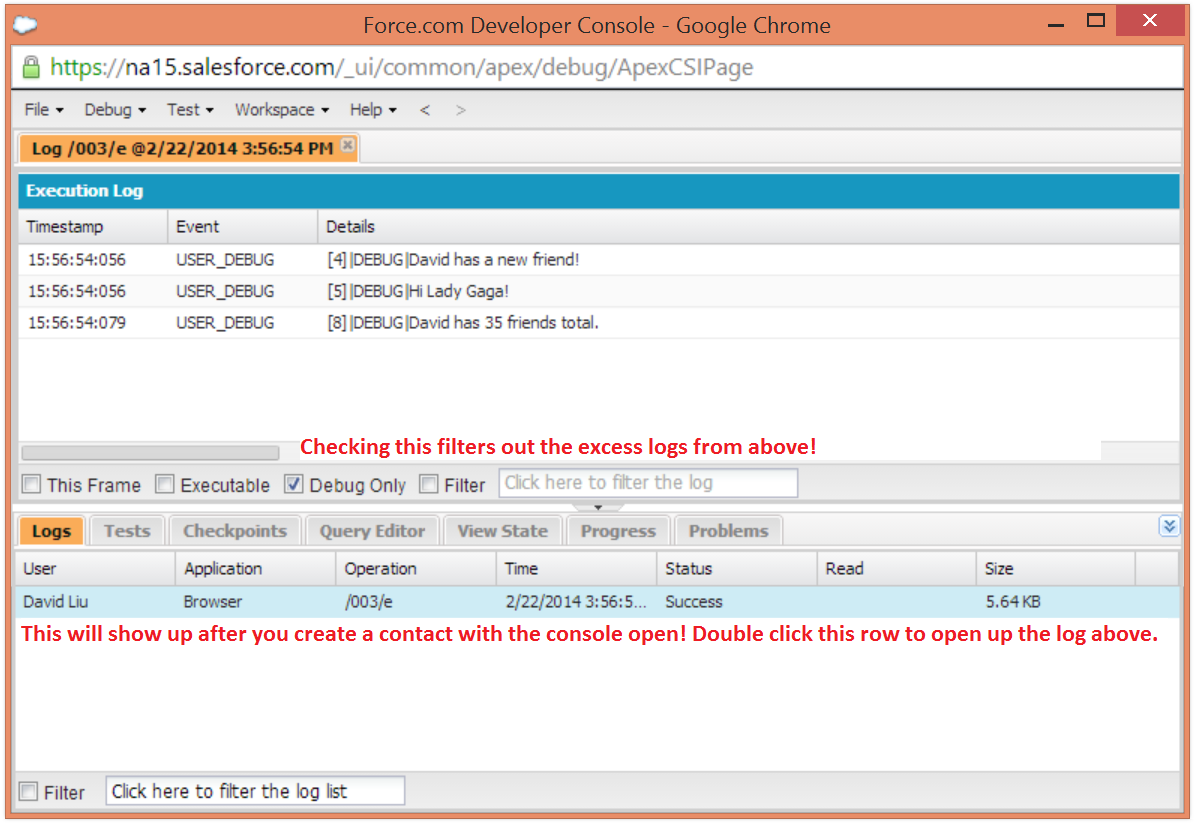
List<Contact> **total** = **[SELECT Id FROM Contact]**;

System.debug('David has ' + **total**.size() + ' friends total.');

}

To see the output of your System.debug, you need to:

1. Write your trigger (no need for a test class yet!)
2. Open up the [Developer Console](https://help.salesforce.com/HTViewHelpDoc?id=code_system_log.htm):  
   – Click **Your Name >> Developer Console** on the top right of any Salesforce page  
   – You must have the “View All Data” permission to access this!
3. Do something in Salesforce that will make your trigger run!
4. Open the log for your latest action, then filter to show “Debug Only”

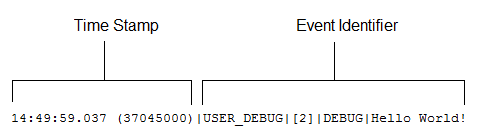
Here’s what the above steps will look like with our example:  


Units of code include, but are not limited to, the following:

* Triggers
* Workflow invocations and time-based workflow
* Validation rules
* Approval processes
* Apex lead convert
* @future method invocations
* Web service invocations
* executeAnonymous calls
* Visualforce property accesses on Apex controllers
* Visualforce actions on Apex controllers
* Execution of the batch Apex start and finish methods, and each execution of the execute method
* Execution of the Apex System.Schedule execute method
* Incoming email handling

Log Lines

Log lines are included inside units of code and indicate which code or rules are being executed. Log lines can also be messages written to the debug log. For example:

****

Log lines are made up of a set of fields, delimited by a pipe (|). The format is:

* timestamp: Consists of the time when the event occurred and a value between parentheses. The time is in the user’s time zone and in the format HH:mm:ss.SSS. The value in parentheses represents the time elapsed in nanoseconds since the start of the request. The elapsed time value is excluded from logs reviewed in the Developer Console when you use the Execution Log view. However, you can see the elapsed time when you use the Raw Log view. To open the Raw Log view, from the Developer Console’s Logs tab, right-click the name of a log and select **Open Raw Log**.
* event identifier: Specifies the event that triggered the debug log entry (such as SAVEPOINT\_RESET or VALIDATION\_RULE).

Also includes additional information logged with that event, such as the method name or the line and character number where the code was executed. If a line number can’t be located, [EXTERNAL] is logged instead. For example, [EXTERNAL] is logged for built-in Apex classes or code that’s in a managed package.

For some events (CODE\_UNIT\_STARTED, CODE\_UNIT\_FINISHED, VF\_APEX\_CALL\_START, VF\_APEX\_CALL\_END, CONSTRUCTOR\_ENTRY, and CONSTRUCTOR\_EXIT), the end of the event identifier includes a pipe (|) followed by a typeRef for an Apex class or trigger.

For a trigger, the typeRef begins with the SFDC trigger prefix \_\_sfdc\_trigger/. For example, \_\_sfdc\_trigger/YourTriggerName or \_\_sfdc\_trigger/YourNamespace/YourTriggerName.

For a class, the typeRef uses the format YourClass, YourClass$YourInnerClass or YourNamespace/YourClass$YourInnerClass.

More Log Data

In addition, the log contains the following information.

* Cumulative resource usage is logged at the end of many code units. Among these code units are triggers, executeAnonymous, batch Apex message processing, @future methods, Apex test methods, Apex web service methods, and Apex lead convert.
* Cumulative profiling information is logged once at the end of the transaction and contains information about DML invocations, expensive queries, and so on. “Expensive” queries use resources heavily.

The following is an example debug log.

|  |  |  |  |
| --- | --- | --- | --- |
| 01 | 37.0 APEX\_CODE,FINEST;APEX\_PROFILING,INFO;CALLOUT,INFO;DB,INFO;SYSTEM,DEBUG; | | |
| 02 | | VALIDATION,INFO;VISUALFORCE,INFO;WORKFLOW,INFO |

|  |  |  |
| --- | --- | --- |
| 03 | | Execute Anonymous: System.debug('Hello World!'); |
| 04 | 16:06:58.18 (18043585)|USER\_INFO|[EXTERNAL]|005D0000001bYPN|devuser@example.org| | | |

|  |  |
| --- | --- |
| 05 | Pacific Standard Time|GMT-08:00 |
| 06 | 16:06:58.18 (18348659)|EXECUTION\_STARTED | |

|  |  |  |
| --- | --- | --- |
| 07 | 16:06:58.18 (18383790)|CODE\_UNIT\_STARTED|[EXTERNAL]|execute\_anonymous\_apex | |
| 08 | 16:06:58.18 (23822880)|HEAP\_ALLOCATE|[72]|Bytes:3 |

|  |  |
| --- | --- |
| 09 | 16:06:58.18 (24271272)|HEAP\_ALLOCATE|[77]|Bytes:152 |
| 10 | 16:06:58.18 (24691098)|HEAP\_ALLOCATE|[342]|Bytes:408 | |

|  |  |  |
| --- | --- | --- |
| 11 | 16:06:58.18 (25306695)|HEAP\_ALLOCATE|[355]|Bytes:408 | |
| 12 | 16:06:58.18 (25787912)|HEAP\_ALLOCATE|[467]|Bytes:48 |

|  |  |
| --- | --- |
| 13 | 16:06:58.18 (26415871)|HEAP\_ALLOCATE|[139]|Bytes:6 |
| 14 | 16:06:58.18 (26979574)|HEAP\_ALLOCATE|[EXTERNAL]|Bytes:1 | |

|  |  |
| --- | --- |
| 15 | 16:06:58.18 (27384663)|STATEMENT\_EXECUTE|[1] |
| 16 | 16:06:58.18 (27414067)|STATEMENT\_EXECUTE|[1] |

|  |  |
| --- | --- |
| 17 | 16:06:58.18 (27458836)|HEAP\_ALLOCATE|[1]|Bytes:12 |
| 18 | 16:06:58.18 (27612700)|HEAP\_ALLOCATE|[50]|Bytes:5 |

|  |  |
| --- | --- |
| 19 | 16:06:58.18 (27768171)|HEAP\_ALLOCATE|[56]|Bytes:5 |
| 20 | 16:06:58.18 (27877126)|HEAP\_ALLOCATE|[64]|Bytes:7 |

|  |  |  |
| --- | --- | --- |
| 21 | 16:06:58.18 (49244886)|USER\_DEBUG|[1]|DEBUG|Hello World! | |
| 22 | 16:06:58.49 (49590539)|CUMULATIVE\_LIMIT\_USAGE |

|  |  |  |
| --- | --- | --- |
| 23 | 16:06:58.49 (49590539)|LIMIT\_USAGE\_FOR\_NS|(default)| | |
| 24 | Number of SOQL queries: 0 out of 100 |

|  |  |  |
| --- | --- | --- |
| 25 | Number of query rows: 0 out of 50000 | |
| 26 | Number of SOSL queries: 0 out of 20 |

|  |  |  |
| --- | --- | --- |
| 27 | Number of DML statements: 0 out of 150 | |
| 28 | Number of DML rows: 0 out of 10000 |

|  |  |
| --- | --- |
| 29 | Maximum CPU time: 0 out of 10000 |
| 30 | Maximum heap size: 0 out of 6000000 | |

|  |  |
| --- | --- |
| 31 | Number of callouts: 0 out of 100 |
| 32 | Number of Email Invocations: 0 out of 10 | |

|  |  |
| --- | --- |
| 33 | Number of future calls: 0 out of 50 |
| 34 | Number of queueable jobs added to the queue: 0 out of 50 | |

|  |  |  |
| --- | --- | --- |
| 35 | Number of Mobile Apex push calls: 0 out of 10 | |
| 36 |  |

|  |  |  |
| --- | --- | --- |
| 37 | 16:06:58.49 (49590539)|CUMULATIVE\_LIMIT\_USAGE\_END | |
| 38 |  |

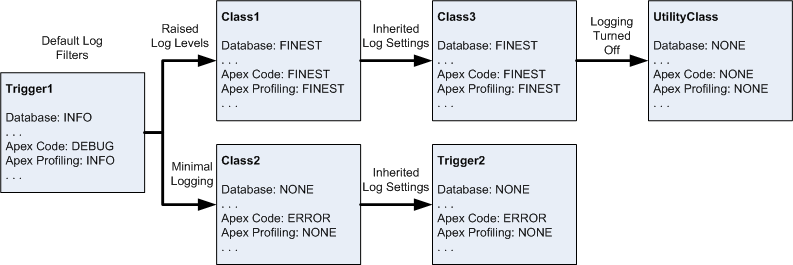
|  |  |  |
| --- | --- | --- |
| 39 | 16:06:58.18 (52417923)|CODE\_UNIT\_FINISHED|execute\_anonymous\_apex | |
| 40 | 16:06:58.18 (54114689)|EXECUTION\_FINISHED |

## Setting Debug Log Filters for Apex Classes and Triggers

Debug log filtering provides a mechanism for fine-tuning the log verbosity at the trigger and class level. This is especially helpful when debugging Apex logic. For example, to evaluate the output of a complex process, you can raise the log verbosity for a given class while turning off logging for other classes or triggers within a single request.

When you override the debug log levels for a class or trigger, these debug levels also apply to the class methods that your class or trigger calls and the triggers that get executed as a result. All class methods and triggers in the execution path inherit the debug log settings from their caller, unless they have these settings overridden.

The following diagram illustrates overriding debug log levels at the class and trigger level. For this scenario, suppose Class1is causing some issues that you would like to take a closer look at. To this end, the debug log levels of Class1 are raised to the finest granularity. Class3 doesn't override these log levels, and therefore inherits the granular log filters of Class1. However, UtilityClass has already been tested and is known to work properly, so it has its log filters turned off. Similarly, Class2 isn't in the code path that causes a problem, therefore it has its logging minimized to log only errors for the Apex Code category. Trigger2 inherits these log settings from Class2.

**Fine-tuning debug logging for classes and triggers**

The following is a pseudo-code example that the diagram is based on.

1. Trigger1 calls a method of Class1 and another method of Class2. For example:

|  |  |  |
| --- | --- | --- |
| 1 | trigger Trigger1 on Account (before insert) { | |
| 2 | Class1.someMethod(); |

|  |  |  |
| --- | --- | --- |
| 3 | Class2.anotherMethod(); | |
| 4 | } |

1. Class1 calls a method of Class3, which in turn calls a method of a utility class. For example:

|  |  |
| --- | --- |
| 01 | public class Class1 { |
| 02 | public static void someMethod() { | |

|  |  |  |
| --- | --- | --- |
| 03 | Class3.thirdMethod(); | |
| 04 | } |

|  |  |  |
| --- | --- | --- |
| 05 | } | |
| 06 |  |

|  |  |
| --- | --- |
| 07 | public class Class3 { |
| 08 | public static void thirdMethod() { | |

|  |  |  |
| --- | --- | --- |
| 09 | UtilityClass.doSomething(); | |
| 10 | } |

|  |  |
| --- | --- |
| 11 | } |

1. Class2 causes a trigger, Trigger2, to be executed. For example:

|  |  |
| --- | --- |
| 1 | public class Class2 { |
| 2 | public static void anotherMethod() { | |

|  |  |  |
| --- | --- | --- |
| 3 | // Some code that causes Trigger2 to be fired. | |
| 4 | } |

|  |  |
| --- | --- |
| 5 | } |