What A Surprise! Or Not!

Charles Wetherell
Oracle Corporation

28 September 2015

Abstract

A procedure has a surprising outcome. Or does it? Once the PL/SQL rules are understood, there are fewer surprise in life.

1 Introduction

A correspondent recently sent the PL/SQL team a small code example and reported surprise at the program's behavior. Once the example was stripped of superfluities, the behavior was easily explained because of a simple rule every PL/SQL programmer should know. The rule and its application to the example are explained here in great – not to say, excruciating – detail for those PL/SQL programmers who enjoy unraveling mysteries. Some morals are drawn.

2 The Example: A Surprise?

Listing 2.1 shows the program. Briefly, the procedure creates a variable, initializes it, calls an inner procedure with that variable as an `in out` argument, and then, in an exception handler, reports the value of the variable. The correspondent was surprised at the value reported.

The reasoning went like this.

- On line 3, variable `Skittish` is initialized to `97`.
- On line 17, procedure `Changer` is called with `Skittish` as an actual argument associated with a formal parameter `t` that is marked `in out`.
- On line 7, the formal parameter `t` has its value changed by adding `42`. No change should have been made to the corresponding actual argument.
- On line 8, the procedure raises an exception and so does not return normally.
- When the print on line 20 is reached, it should print `97` because the actual argument should remain unchanged until the call on line 17 finishes normally.
- However, the correspondent observed that the value printed was `139` and was surprised.

Listing 2.1: The Surprising Procedure

```
1  procedure SurpriseMe is
2
3    Skittish pls_integer := 97;
4
5    procedure Changer(t in out pls_integer) is
6    begin
7      t := t + 42;
8      raise zero_divide;
9    end;
10
11   procedure Watcher is
12   begin
13     dbms_output.put_line(Skittish);
14   end;
15
16 begin
17   Changer(Skittish);
18   exception
19     when zero_divide then
20       Watcher();
21 end;
```

3 The Mystery Demystified

Is this surprise warranted?

If the surprise were sensible, there would be no point to this note. The astute reader will already have guessed that there must be an explanation that is *not* surprising. Indeed there is and here it is.

Our correspondent has the general principle correct that the actual arguments associated with formal parameters tagged `out` or `in out`[1] are not given their new values until the call to the subprogram ends. But there is another more specific rule about calls that went unnoticed.

> When a call terminates with an unhandled exception, the value of an actual argument associated with an `out` formal parameter becomes *undefined*.

And that's true for every single actual argument so associated in a particular call. This rule appears in the current PL/SQL manual immediately after the discussion of parameter modes.

[1] Strictly speaking, the notion of `out` formal parameter encompasses `in out` formal parameters for the purposes of this discussion. The rest of the note will just call them `out` parameters.

What does *undefined* mean? It means that there is no prediction about what the value will be. It does *not* mean that the value will be `null` although that is a possibility. It simply means that there is no way to predict in advance what value will appear. Moreover, there is no requirement that the value stay the same from one execution to another or from one day to another. To repeat (redundantly), there is no information available about the value; what you see is what you get.

It does not matter what caused the exception. Here, the procedure raised an exception explicitly. But it could have been raised implicitly; so long as it escapes the procedure to the caller, that is sufficient to trigger the rule. And the call itself might trigger the exception. For example, evaluation of an actual argument might have a problem and the subprogram might never even be called. It is also possible that moving `out` values back from the call to the actual arguments could raise an exception. No matter where the exception comes from, if it does happen, the actual arguments instantly become undefined.[2]

And that solves the (apparent) mystery. When the procedure `Watcher` looked at the actual argument `Skittish` in the exception handler, the value of `Skittish` was already undefined; it had become undefined at the moment the `zero_divide` exception hit the call. Once the value is undefined, it might be anything. In particular, `Skittish` could have the value `97`, the value `139`, the value `-183`, the value `null`, or any other value that a `pls_integer` variable may hold. And on the next execution, it could return yet a different value.

4 Some More And A Moral

As it happens, there is a reason why the output might be `139`; this didn't happen entirely by chance. The compiler is smart enough to notice that after the call to `Changer`, there are no references to `Skittish` that can see the defined value of `Skittish`. The reference through `Watcher` does not count because it is only looking at the value and the value is undefined at the moment `Watcher` is called. Once this fact was known, the compiler realized that `Skittish` can be passed as if `nocopy` had been specified. Thus the assignment (`:=`) in `Changer` reset variable `Skittish` directly; there was no copy made. On the other hand, if any real reference to `Skittish` occurs on some legal code path following the call, this optimization will not be applied.

The correspondent who noticed this mystery happened to compile the example at two different optimization levels. At one level, the compiler made this change; at the other, it didn't. But this pattern cannot be relied upon; the PL/SQL team is free to

[2]This rule also applies when the `nocopy` hint has been applied to some (or all) of the formal parameters. Undefined is undefined and there is no getting around it.

change the specifics of optimization at any moment, always presuming that program semantics are preserved. The pattern may also change depending on the specific text of the program at hand. Nonetheless, it is comforting to have an explication of the mysterious, especially when the mystery turns out to be mundane.

And now the moral. When PL/SQL says something is undefined, no useful reliance can be placed on the undefined item. There is no pattern, no necessary reasoning, no experiment, that can provide a useful explanation of the status of the undefined item. Take care to know when undefined items appear and do not rely on them.

Finally, I have written a paper *Freedom, Order, and PL/SQL Optimization* that discusses many similar topics. If you enjoyed this account of a mystery solved, you may also enjoy learning more about PL/SQL and its definition. The URL for the ZIP file is

`http://www.oracle.com/technetwork/database/features/plsql/codeorder-133512.zip`

You may also find the general Oracle PL/SQL page `oracle.com/plsql` worth exploring. You can find a link to the paper and much else there besides.