Things, Names, And Identifiers

Charles Wetherell
Oracle Corporation

October 24, 2019

Abstract

Databases are full of things: tables, sequences, columns, views, PL/SQL units, what have you. Things have names and are manipulated by mentioning the names. The programming languages SQL and PL/SQL use identifiers, not names. Questions show many programmers are confused about the difference. This note describes the relationships between things, names, and identifiers. Once the programming rules are absorbed, developers can write code faster and with less heartburn.

1 A Coding Question

> Programming is the art of
> algorithm design and the craft
> of debugging errant code.
>
> Ellen Ullman

Imagine a PL/SQL program where it is necessary to select the values of a single column from a table where both the column and the table are to be specified at execution. The `EXECUTE IMMEDIATE` statement constructs and executes SQL statements "on the fly". Also imagine the application already has a library procedure `Runit` for this chore. Figure 1.1 shows a little bit of the procedure and five possible calls. Presuming that goal is to select the column `Salary` from table `emp`, which of these calls will work and why? What are the problems with the calls which don't work?

Listing 1.1: A Program Snippet

```
1  procedure RunIt(TableID varchar2, ColumnID varchar2) is
2  ...
3  begin
4  ...
5  execute immediate 'select ' || ColumnID || ' from ' || TableID into MyVal;
6  ...
7  end;
8
9  RunIt(emp,      Salary);       -- First call
10 RunIt('emp',    'Salary');     -- Second call
11 RunIt('emp',    '"Salary"');   -- Third call
12 RunIt("emp",    '"Salary"');   -- Fourth call
13 RunIt('"EMP"', '"Salary"');    -- Fifth call
```

This note provides a way to think about these questions and – yes – the answers to the quiz are provided later – but don't peek! Before the answers, a journey through näive philosophy and some programming language technique will provide background for understanding. Theory and practice merge and illuminate the answers. Even if the theorizing slips away from memory, the rules that govern names and identifiers are

clearly stated and should remain ingrained so that one class of programming errors occurs no more.

2 Things And Signifiers

> Kid: What do monsters eat?
> Dad: Things.
> Kid: What do monsters drink?
> Dad: Coke.
> Kid: Why?
> (Cue jingle) Things go better
> with Coca-Cola,
> Things go better with Coke!
>
> 1960's advertising

A database is full of things. The main things are tables, but there are many others: views, columns, sequences, data types, PL/SQL units, on and on. Databases exist so data can be stored and manipulated and retrieved. For example, it may be interesting to find the total salary of each employee last year and to have that report sorted by department number first and employee name second. Probably the data is stored in several tables, one for employee personnel information, one for the departmental organization, and one for salary payment transaction records. Assume every employee has a Social Security number and that each employee record contains that number. Matching records from the three tables using Social Security numbers builds the report. SQL programmers recognize this as a basic database operation.

How are the necessary tables, columns, and so on accessed? Informally, tables and columns were described in terms of their properties. This is not enough; what might be a departmental organization table to me might be a personnel assignments table to you. To avoid ambiguity, database things are referenced by their names. For example, the table of all employees might be named `Employees`, the table of department information named `DEPT_ORG`, and the salary table named `salary`.[1] The Social Security number columns might all be named `SS_ID`. To operate on the things, the operations mention their names. One property of names can already be seen: the same name might apply to different things in different contexts.

[1] Names with mixed cases are unusual; they appear here as prelude to explanations which will come later.

Another question arises. How are the operations on these things specified? Typically, a SQL statement or a PL/SQL program drives an operation. A SQL statement for salary report might start like this:

```
select "Employee".surname, sum(salary.amount), ...
from "Employee", salary, DEPT_ORG
where "Employee".ss_id = dept_org.ss_id ...
```

The SQL statement is a program text and mentions of database elements appear in it. But why is the departmental organization table mentioned once as `DEPT_ORG` and once as `dept_org`? Why is the employee information table mentioned as `"Employee"` with those double quote marks `"` wrapped around the text? Why is the Social Security column name written `ss_id` rather than `SS_ID` as it was before?

The short answer to all these questions is that program texts contain identifiers and not names. Just as names reference things, identifiers reference names. The point of this paper is to understand the connections between things, names, and identifiers.

Things are the stuff of the world. Everything about us and everything we use is a thing. A database reflects many things.

- The employees whose information is in the database.
- The HR forms that encapsulate each employee's company life – pieces of paper with ink marks on them – each form and each ink mark itself a thing.
- The item each group of ink marks represents – for example, a salary.
- The datum – a number or text – that abstracts an item.
- The database tables that record this data.
- The particular data values inserted in the tables to one employee.

This list could go on. How is a particular thing singled out and how are groups of things discussed?

Linguistics and philosophy together provide an answer. Consider this sentence:

> The postman delivered three letters to John Smith at 123 Maple.

We understand some facts about the real world from this sentence.

- A person (*postman*) who delivers mail acted in the past.
- The action involved 3 objects (*the letters*).
- There was a specific number (*3*) involved.
- Another particular person (*John Smith*) received the letters. The name *John Smith* gives us the knowledge that a specific person was intended.
- The delivery took place at a building at a particular location (*123 Maple*). This is not the proper building name, but rather a geographical locator that allows the house to be found.

What this sentence did not include was any direct sensation that would allow us to observe the mail delivery. The sentence does not include any people, any envelopes, any houses. How is it possible to be certain what the sentence is describing?

Linguistics says that the nouns in a sentence are *references* to things, not the things themselves. The words used to signal the reference have an arbitrary connection to the things being discussed. For example, the *postman* in English might be the *mailman* or the *mail carrier*. In German, the *Postbote*, in French *facteur* or *factrice*, in Croatian

poštar, and so on. The connection of *123 Maple* to a particular building is obviously arbitrary. Both the naming of the street itself and the ways the numbers are attached to buildings are conventional and open to change. Similarly, John Smith might have been named *Jonathan Smith* or *Johann Schmidt* or even *Jon Smith Baring-Gould-Postlethwaite* and would still be the same person.

Philosophy tells how to understand these references. The noun *postman* is a signifier and the particular person who was delivering the mail on that day is the *signified*. There is a arbitrarily defined link between the signifier and the signified. In any particular language, the common nouns link to common objects and we all learn the links as part of learning the language. Proper nouns are more arbitrary. We have to be introduced to John Smith the first time we meet him; there is no indication on the person standing before us of the proper noun used to name him.[2]

There are some rules for signifier/signified relation.

- It must stay stable for any one interaction, whether a single word or long conversation such as a program.
- A signified thing may have more than one signifier. For example, the postman has different signifiers in different languages and at least three different signifiers in ordinary English. The particular postman has a given name – perhaps *Robert Roe* – and many noun phrases describe him – *star center on the high school team.*
- A signifier can signify more than one thing. For example, the address *123 Maple* probably occurs dozens of times in the United States. There is a place in New Jersey where one *River Road* crosses another and distinct *River Road*. Common nouns also can signify different things; *tree* can apply to both an oak outside the window and a data structure inside a program. *Table* has the same possibility; a SQL statement about a table could be written on a table.
- A signifier can link to a signifed thing which is in turn the signifier for a further signified thing. For example, *You should invest in ORCL* uses the stock symbol *ORCL* to signify the corporate name *Oracle Corporation* which in turn signifies the real world entity that is the (notice the lower case, common noun usage) corporation that builds databases and other computer-y things.

You may not have seen these rules written out before, but you have used them in both technical work and ordinary life. For example, mathematicians and computer scientists take great care to define terms exactly; an *ideal* is not something to believe in and work for – it is a particular kind of group. A *group* is not a collection of friends to have lunch with, but a precise abstraction of some common ideas from algebra. Similarly a *tree* or a *table* is not made of wood and a *column* does not hold up the office building. These *terms of art* are specialized signifiers when used in specialized contexts. A legal contract may have language like *the moving party hereinafter known as the plaintiff* or *the estate means henceforth all real and personal properties.* These phrases establish links between signifier and signified for the purpose of one document.

An old (funny?) linguistics joke illustrates signifiers. Someone says

> There are six letters in Boston.

[2]Erwin's book has a good short description of *signifies/signfied*. Martin Erwin, *Once Upon an Algorithm: How Stories Explain Computing*, MIT Press, 2017,

Probably a false statement because likely there are thousands of envelopes delivered by snail mail every day in a large city like Boston. But we might have misheard

> There are six letters in "Boston".

which is undeniably and absolutely true[3]. What is the difference between the two versions? In the first, the word *Boston* is unquoted and so is a signifier for the city of Boston. In the second version, the word *"Boston"* is quoted and so signifies the English proper noun which is the name of that Massachusetts city. The proper noun certainly is written with six letters and this makes the statement true. The noun *letters* signifies one thing in the first sentence – envelopes with paper inside – and another in the second – elements of the Latin alphabet. Apparently linguists also enjoy very bad puns.

3 Platonic Programming

> After we came out of the church, we stood talking for some time together of Bishop Berkeley's ingenious sophistry to prove the non-existence of matter, and that every thing in the universe is merely ideal. I observed, that though we are satisfied his doctrine is not true, it is impossible to refute it. I never shall forget the alacrity with which Johnson answered, striking his foot with mighty force against a large stone, till he rebounded from it, "I refute it THUS."
>
> *The life of Samuel Johnson, Vol. 1*, James Boswell

Consider the integer 19. What, exactly, is it? Is it the digits 19 – in other words, the decimal notation for the integer? Perhaps it is `0x13`, `0b10011`, or `023` – all C

[3]This joke works in spoken English but gives itself away in written English.

notations. Or then again, perhaps 21_3, 12_{17}, or $211.11\ldots_e$?[4] How about *XIX* or *xviiii*? Or *nineteen* or *devetnaest* (Croatian) or *diecinueve* (Spanish) or *neunzehn* (German)? These cannot all be the integer also known as 19. Indeed, each reader will see a different version of what is written here, entirely new instances and configurations of ink or pixels. If any one of these is 19, what are all the others?

Socrates, in Plato's *The Allegory Of The Cave*, proposes a solution: what we perceive is not the "real" thing, only a flickering shadow. There are *ideal* things humans strive to apprehend, but only approximate apprehensions are available to us. We can appreciate finer and finer detail, but the ideal object is never completely understood. This may excessively obscure for ordinary objects like stones, roast beef sandwiches, and supersonic jet planes, but for mathematical objects, this ideal view makes considerable sense. No human will ever apprehend $\sqrt{2}$ as a fully written out decimal number. But we all believe $\sqrt{2}$ exists and we all think that use of $\sqrt{2}$ in a proof, say, refers to the same number as the root of the equation $x^2 = 2$. Although this is seldom mentioned, most mathematicians seem to be informal Platonists who believe there is a universe of eternal mathematical objects out there – mathematics is the physics of this wondrous universe.

This matters because the SQL databases and the PL/SQL programs which embed names and identifiers are themselves best thought of as mathematical objects, not as particular masses of bits on particular computers. Instead, those bits or text files or programming pads filled with handwritten "programs" are representations of underlying and ideal mathematical objects. Each of the things (identifiers and names in particular) will have both an abstract existence, permanent and universal, and a variety of specific representations in particular instances, versions, copies, and memories on many different computers, printouts, scratch pads, video screens, whiteboards, conversations, and brains. It isimportant to keep track of the distinction between the representation of something and the thing itself. Also remember that signifiers and

[4]Irrational numbers can be the base of a notation. When the base is e, integers will have infinite, non-repeating decimal expansions.

signified can form chains with some links which are representations and some which are ideal things.

4 Identifiers And Names

That which we call a rose
By any other word would smell
as sweet.

Romeo And Juliet, II, ii.
William Shakespeare

Listing 4.1 show a PL/SQL procedure. To be more precise, the text is one possible representation of the procedure. The Platonic view suggests there are many *representations* of this procedure; somewhere, the ideal procedure lives. This is not the procedure itself – it is the *text* of the procedure. This is one signifier among many – an infinite number – of source code for the signfied procedure. The particular ink marks or pixels read at the moment are themselves just signifiers of a text which signifies the procedure.

Listing 4.1: An Example Program

```
1 procedure Show_Salary is
2   the_sal pls_integer;
3 begin
4   select sal into "THE_SAL"from emp where empno = 23;
5   dbms_output.put_line(THE_SAL);
6 end;
```

This all seems pointlessly tedious and academic. Or is it? Philosophy and mathematics force examination of details to ensure no hidden assumptions or misunderstandings are smuggled into an argument. Now that the relationship of physical marks to text to program to procedure has been noticed, simply saying "the program text" is fine. If needed, the analysis microscope can turn the magnification higher.

Inside the text, we see some examples of *identifiers*. SQL and PL/SQL define identifiers the same way; the details can be found in the reference manuals. Here is a complete list of the identifiers found in the program text:

```
Show_Salary the_sal pls_integer sal "THE_SAL" emp empno dbms_output
put_line THE_SAL
```

Each identifier is a coherent fragment of the text. The technical name for such a fragment is *token* and corresponds roughly to anything that might be a word or punctuation in a natural language. Because this is not the procedure itself, these

identifier tokens are definitely *not* names. This is the first lesson: identifiers live in program texts and names live in the ideal SQL database.

An identifier signifies a database name so that the program text can specify operations that manipulate the various named things in the database. A ticket to Boston is expected to provide physical transportation on a physical device to the physical city of Boston, not to a six letter word. Similarly, the identifier `emp` is expected to provide access to a database table which has the data necessary for the procedure to execute. SQL and PL/SQL connect identifiers to names with a specific protocol.

There are two kinds of identifiers.

- *Ordinary* identifiers start with a letter and continue with letters, digits, or with the punctuation marks `$`, `_`, and `#`. For example, `put_line` is an ordinary identifier.
- *Quoted* identifiers begin with a double quote `"`, continue with any characters except `nul`, newline, or double quote, and end with a double quote. There must be at least one character inside the double quotes. For example, `"THE(sal)"` is a quoted identifier.

Ordinary identifiers are similar to those found in most programming languages. But quoted identifiers are the fundamental way to signify names in the Oracle universe. Here is the rule for quoted identifiers:

> The name signified by a quoted identifier is exactly the text surrounded by the double quotes in the identifier.

Notice that the name does not contain the double quotes. For example, `"ab_BDC!!..()fooBoo"` signifies the name `ab_BDC!!..()fooBoo`. The rule for ordinary identifiers is more complicated.

1. Convert all the alphabetic characters in the identifier to upper case. Leave other characters untouched.
2. Surround the resulting text with double quotes.
3. Apply the quoted identifier rule to find the signified name.

So the ordinary identifier `the_sal` transforms to the text `THE_SAL` and then to the quoted identifier `"THE_SAL"`. This then signifies the name `THE_SAL` which is a local variable in the PL/SQL procedure.

The effect of the ordinary identifier rule is that

- `the_sal`
- `THE_SAL`
- `"THE_SAL"`

are distinct identifiers – two ordinary, one quoted – but that they all signify the same name: `THE_SAL`. Similarly, the identifier `emp` could have been written as any one of the eight ordinary identifiers `emp`, `Emp`. `eMp`, `emP`, `EMp`. `EmP`, `eMP`, or `EMP` or as the quoted identifier `"EMP"` and they would all have signified the same database table name `EMP`. The quoted identifier `"emp"` signifies the name `emp` and there probably is no table with this name. It would not signify the table `EMP`.

Perhaps the most confusing aspect is that PL/SQL and SQL names are themselves represented as sequences of characters just as identifiers in a program text are. This

dual use of representation often catches us by surprise. However, we see it commonly enough: *deadbeef* may be an unappetizing menu item but `deadbeef` is a good hexidecimal value to put in unused memory when searching for memory access bugs. The same characters represent quite different things.

SQL and PL/SQL are often said to be "case insensitive" languages. That is not quite true. Rather, ordinary identifiers go through the three step dance so they signify names with only upper case alphabetic characters. Names – like quoted identifiers – may contain almost any characters. The "case doesn't matter" idea makes typing program text easier but is too simple an idea to explain all database behavior and naming.

It would have been possible to design the Oracle database so it did not allow all those "fancy" characters in names and so its names were case insensitive. But the outside world does not live by these rules. Other databases, other programming languages, other systems all have their own rules for how identifiers and names are constructed. It is far easier to allow more-or-less arbitrary names in the database at the cost of a few extra double quote characters when necessary than it is to devise some transliteration scheme. For day-to-day Oracle programming, ordinary identifiers work well with low effort. The more powerful quoted identifiers and names with arbitrary characters cater for the idiosyncrasies of other systems with which Oracle communicates.

5 Practical Programming

> Knowledge is of no value unless you put it into practice.
>
> Anton Chekhov

In ordinary discussion of SQL and PL/SQL programs, the identifier/name distinction seldom causes problems. If something is unclear, a question usually resolves any confusion. But there are two situations where ambiguity or misuse can cause problems

- In documentation.
- In programs which construct and execute other programs: for example, PL/SQL's `EXECUTE IMMEDIATE` statement.

Documentation often says something like "a table name may appear in a `WHERE` clause list". It is true that during execution, the SQL statement that has the `WHERE` clause may include the name of a table, but in the *program text*, an identifier must appear to signify the table name. The statement would be clearer had it read "an

identifier that specifies a table may appear ...". In particular, any syntax diagram that has the entry *name* should be rewritten with *name* replaced by *identifier*. It is impossible to write a name in program text; names only exist within the database. Only identifiers may be written in a program text.

Many applications create new program text to be executed sooner or later. The PL/SQL `EXECUTE IMMEDIATE` statement is expressly designed to create and execute SQL statements on the fly. PL/SQL subprograms have arguments that specify some task and then construct the needed SQL. The arguments may specify database objects such as tables and columns. Programmers are often confused about what is to be supplied as values for these arguments.

In Listing 5.1 (a repeat of Listing 1.1), the procedure `RunIt` presumably provides some information about a column in a table. The column and table are specified by the formal parameters `ColumnID` and `TableID`. If the table is named `EMP` and the column is named `Salary`, which of the calls will work properly?

First Call: The first call is unlikely to work. The actual arguments are the PL/SQL variables `emp` and `Salary`. Unless these are both typed `varchar2`, variable `emp` contains a string value such as `emp`, and `Salary` contains exactly the string value `"Salary"` – notice the value includes the double quotes – the call will fail. The string value in the first variable could also be `EMP` or `eMP` or any of the eight combinations of letters that make up an identifier for the name `EMP`.

The first call confuses PL/SQL identifiers with SQL identifiers. Also, the `EXECUTE IMMEDIATE` statement takes a text value for its operand, not identifiers. The example statement shown shows how the *values* of the actual arguments are used to construct the *text* of a statement to be executed; the PL/SQL identifiers themselves are *not* magically combined during the string construction to provide their own text representation as part of the final text. In PL/SQL, identifiers always provide their values.

Second Call: The second call almost works. It correctly provides text values in the form of string literals for placement in the skeleton `EXECUTE IMMEDIATE` statement. The first value is one of the eight possible identifiers for a table whose name is `EMP`. But the second value `'Salary'` is the text of an identifier for a column named `SALARY`, not `Salary`.

Listing 5.1: A Program Snippet

```
1  procedure RunIt(TableID varchar2, ColumnID varchar2) is
2  ...
3  begin
4  ...
5  execute immediate 'select ' || ColumnID || ' from ' || TableID into MyVal;
6  ...
7  end;
8
9  RunIt(emp,      Salary);       -- First call
10 RunIt('emp',    'Salary');     -- Second call
11 RunIt('emp',    '"Salary"');   -- Third call
12 RunIt("emp",    '"Salary"');   -- Fourth call
13 RunIt('"EMP"', '"Salary"');    -- Fifth call
```

A probable cause here is the difference between single quotes ' and double quotes " in SQL and PL/SQL. Single quotes delimit and define string literals. Double quotes similarly bound quoted identifiers. The two usages cannot be interchanged. The error might arise from a momentary lapse of attention or from a lack of understanding.

Third Call: The third call will succeed as intended. The first actual argument `'emp'` is the same as in the second call and it will correctly create an identifier which will signify the table named `EMP`. The second argument `'"Salary"'` is a string literal which has the correct text for the quoted identifier `"Salary"`. The single quotes on the outside create the string literal and the value of that literal includes the double quotes that for the quoted identifier. The use of both kinds of quotes on the same value is often a stumbling block for beginning PL/SQL programmers.

Fourth Call: The fourth call fails. The second argument is the same as the third call and will work properly. But the first argument `"emp"` is simply a PL/SQL quoted identifier. As with the first call, unless that identifier names a PL/SQL variable which happens to have a text value that is an appropriate argument to create the SQL name `EMP`, the call will fail.

Fifth Call: This call will work. The first argument is a string literal whose value is `"EMP"` – exactly what is needed to signify the SQL table `EMP`. And the second argument is unchanged from the third call so it will also work properly.

An observant reader might have noticed one more problem with the example. The formal parameter identifiers are `TableID` and `ColumnID`. Even though the associated type is `varchar2`, a programmer might be misled into thinking that an identifier is a possible value in PL/SQL. It isn't, of course: only the text of an identifier can be a value. Misnaming of formal parameters is a subtle and pernicious way to encourage programming mistakes. There are many instances of poor parameter naming choices throughout Oracle supplied library code and, no doubt, in application code as well. A better naming choice might have been `TableIDText` and `ColumnIDText` or perhaps `TableText` and `ColumnText`.

6 Finale: In Wonderland

> Morrissey writes wonderful
> song titles, but sadly he often
> forgets to write the song.
>
> Elvis Costello

The distinction between names and things and the distinction between signifiers and significand has occupied philosophers at least since Socrates. Lewis Carroll gives us a wonderful example.

> "You are sad", the Knight said in an anxious tone: "let me sing you a song to comfort you."
>
> "Is it very long?" Alice asked, for she had heard a good deal of poetry that day.
>
> "It's long," said the Knight, "but it's very, very beautiful. Everybody that hears me sing it - either it brings the tears into their eyes, or else -"
>
> "Or else what?" said Alice, for the Knight had made a sudden pause.
>
> "Or else it doesn't, you know. The name of the song is called 'Haddocks' Eyes'."
>
> "Oh, that's the name of the song, is it?" Alice said, trying to feel interested.
>
> "No, you don't understand," the Knight said, looking a little vexed. "That is what the name is called. The name really is 'The Aged Aged Man'."
>
> "Then I ought to have said 'That's what the song is called?'" Alice corrected herself.
>
> "No, you oughtn't: that's quite another thing! The song is called 'Ways And Means': but that's only what it's called, you know!"
>
> "Well, what is the song, then?" said Alice, who was by this time completely bewildered.
>
> "I was coming to that," the Knight said. "The song really is 'A-sitting On A Gate': and the tune's my own invention."

The White Knight lists the name of the name of the song as *Haddocks' Eyes*, the name of the song as *The Aged Aged Man*, what the song is called (another signifier) *Ways And Means*, and the song itself is *A-sitting On A Gate*. Alice was confused but Oracle programmers need not be. Things are not the same as their representations, names themselves can be things, names are not the same as the things they signify, and there can be chains of signifiers. We can hope Alice enjoyed the song as much as we enjoy our newfound skills with identifiers and names.

Postscript

After this essay was complete and ready for posting, I read Benjamin Dreyer's style guide *Dreyer's English: An Utterly Correct Guide to Clarity and Style*. In his section on commas, he reminds us that

> We were all thoroughly indoctrinated in grade school to precede or follow dialogue with a comma...

and one example is

> Atticus said dryly, "Do not let this inspire you to further glory, Jeremy."

Dreyer goes on to say

> . . . this rule does not apply in constructions in which dialogue is preceded or followed by some version of the word "to be". . .

His example is

> "Happy New Year" is a thing one ought to stop saying after January 8.

Dreyer's reasoning is

> . . . the phrase in question is less dialogue than a noun-in-quote-marks. . .

and this is exactly the point of the lame joke about Boston in Chapter 2 and that the White Knight is making to Alice.