

A Brief Introduction to Binding.

Dr. Tom Johnston

In its most universal sense, binding is the assignment of content to form. In the context of a language, it is the assignment of semantics to syntax. But IT professionals usually think of binding in the more specific sense of the assignment of values to the variables manipulated by a computer program.

Binding Values to Variables.

To illustrate, let's say that we have a cursor-controlled loop through a table of customers. The schema for that table is shown in Figure 1.

A schema, in the specific sense in which I will use the expression, is a named template of typed variables. A schema for a relational table is what is created with a CREATE TABLE statement, and it defines the *syntax* of a table.

Customer

cust- nbr	cust-nm	cust-status	q1- purch
char(7)	char(30)	char(3)

Figure 1. The Customer Table Schema.

A brief note on how to read these graphical representations of schemas. The top row contains the column names for the table. The left-to-right sequence of columns corresponds to the top-to-bottom sequence of columns in a CREATE TABLE statement. Following standard convention, I place the primary key column or columns first. In addition, primary keys columns will always be shown in boldface. Foreign keys will be shown in italics.

The second row contains the data types and lengths (henceforth, “data types” by itself will be taken to include length, when length can be specified) of the columns.

Neither of these rows, obviously, is a row in the sense of being an instance of a relational table. These are just rows of a graphical representation of a relational table schema. However, the third row can be thought of as a “viewer”, in which “real” rows, i.e. instances of the table, appear. If I can illustrate my points one sample row at a time, I will use this “viewer” format. If it takes multiple rows, then the concept of a viewer goes away. In that case, the third row of this graphic of a relational table is then the first real row of some result set; the next row of this graphic is the second row of the result set, and so on.

As our cursor scrolls through each row of the table, a different set of values appears in the viewer. Let’s say that we have just moved the cursor to some row, and the following set of values pops up:

Customer

cust-nbr	cust-nm	cust-status	q1-purch
char(7)	char(30)	char(3)
CHI-305	Jones	vip	\$386.04

Figure 2. A row of the Customer Table.

At this point – *but not before* – the value “CHI-305” is bound to the variable cust-nbr. The value “Jones” is bound to the variable cust-nm. And so on. At this point, if code were to display the current value of the variable cust-nm on the screen, it would display “Jones”.

When it’s time to move on, we bump the cursor, and get the next customer. Let’s say what pops up in the viewer this time is:

Customer

cust- nbr	cust-nm	cust-status	q1- purch
char(7)	char(30)	char(3)
GA-338	Smith	occ	\$705.85

Figure 3. Another Row of the Customer Table.

Now it is the value “GA-338” which is bound to the variable `cust-nbr`, and the value “Smith” which is bound to the variable `cust-nm`.

These variables are said to be “late-bound” to their values. The specific form of late-binding we see here is often called “run-time binding” because values are assigned to these variables only when the program is actually running.

The important question about binding is cost. (Indeed, the important question about *any* aspect of IT development and maintenance is cost.¹) And a critically important question that we should always ask ourselves is: how much does it cost to *change* the binding of variables?

Well, in this example, how much does it cost to execute the program which bumps a cursor through the rows of a Customer table? For all practical purposes, it costs *nothing*. As the variables are unbound from one set of values and bound to the next set, unbound from one customer row and bound to the next, the programmer doesn’t have to do anything.

Now let’s consider a program which calculates a gift certificate to be offered to each customer, based on how much the customer has purchased in the most recently-completed quarter. That total, for the first quarter, is contained in the `q1-purch` column. Management decides that the discount will be five percent of total purchases for the quarter.

¹ More precisely, of course, the important thing, the bottom-line thing, is the difference between cost and value provided, the “margin” so to speak. But if the objective has already been decided on, i.e. if management has already decided that a new system will be built, the focus then shifts to cost. This is because, with the decision to build having been made, the only way in which IT can improve the margin is to lower the cost.

Inside this program, there is a variable called `cust-gift-pct-rate`, type `Decimal(3,2)`, and the variable is assigned the value “1.05”. Note that this value is assigned in the program itself. It is written in the source code, and compiled into the object code. We often say in such cases that the value is *hardcoded* into the program.

This value could have been read from an external data source. If it had been, it would have been *run-time-bound* to the program. But having been written into the source code, it is said to be “early-bound”, and in particular *compile-time-bound*, to that program. It is compile-time bound because in order to change it, we would have to change the source code and recompile the program.

In this case, hardcoding, or compile-time-binding, is not a bad idea. Gift certificates are offered quarterly, but the discount rate – in this case, five percent – applies to all customers, and (let us suppose) is good for all four quarters. So this program won’t have to be re-written and re-compiled until next year. And the re-write, if needed, will involve only a one-line change, to the line which assigns the value to `cust-gift-pct-rate`.

Good programmers instinctively (or sometimes consciously) try to minimize binding costs. They do this by hardcoding only those variables which will not change values very often and which, when they do, will be easy and inexpensive to change. Variables like `cust-gift-pct-rate`.

But suppose that the discount offered to customers was computed by a different program which took a percentage associated with each customer, calculated total sales for the quarter, and then computed the gift discount. In this case, it would be absurd to hardcode these discount percentage values in the program that calculates the discounts, because that would mean hardcoding an internal table that had one row for each customer and two columns – `cust-nbr` and `cust-gift-pct-rate`.

Hardcoding is so obviously wrong in this case that it is unlikely that it would ever occur to an experienced developer to do it. But that doesn’t mean that there isn’t an important principle at work. That principle is shown in Figure 4.

Early-bind only what changes infrequently, and can be changed at little cost.

Figure 4. Principle: The Binding-Time Principle.

The reason behind this principle is simple. It is that the earlier the binding, the more expensive it is to change.

So far, we have discussed binding in terms of binding values to variables. Before we go on, we need to recognize that this is only a special case of the more general issue of binding.

Binding Semantics to Syntax.

Binding is more than what happens when variables are assigned a value. Binding, in its most general sense, is the assignment of *content* to *form*, of *semantics* to *syntax*.² In the example just considered, the values were the content, while the variables (with their names and data types, and in their specific sequence) were the form; the values were the semantics, while the variables (with their names and data types, and in their specific sequence) were the syntax. The rows scrolling through the viewer were the semantics. The “viewer” itself was the syntax.³ As each new row popped up, the viewer changed its content. Semantics is content; syntax is structure.

² These different formulations of binding exist in decreasing order of generality. Binding, in the most general sense, is *the assignment of content to form*. Since the kind of content we in IT are interested in is structured information, we are familiar with a more specific way of thinking about binding, which is as *the assignment of information to the structures that contain it*. But this is just *the assignment of semantics to syntax*, in the specific form of database structures and processes. Finally, from the programmer/developer’s point of view, binding is *the assignment of values to variables*.

³ In my graphical convention for illustrating result sets, the top two rows provide the syntax for the viewer, which is the third row.

If there are other cases of semantics, syntax and binding which are relevant to our work as data modelers, what are they? Are there early and late binding issues that affect data modelers, and not just programmers? Yes, there are.

Binding and Data Models.

There are two binding time issues that directly affect data modelers. They are:

1. Business data used as primary and foreign keys.
2. Associative tables and embedded foreign keys as two different syntactical forms that express relationships.

In both cases, relational theory and relational DBMSs *early bind* semantics to syntax. And in both cases, this has caused an almost unimaginably great amount of unnecessary cost in the maintenance of information systems and their databases.

But before I explain how this is so, and what we can do about it, I need to warn you that the situation is somewhat like that of the emperor and his clothes. For it is generally accepted, among both data modelers and academic computer scientists, that one of the strengths of relational theory is that it does *not* early bind semantics to syntax. The point is seldom couched in this language, however. Instead, practitioners and computer scientists talk about *logical/physical data independence*.

And let me add here that this reference to a fairy tale by Hans Christian Andersen intends no disrespect to Dr. Codd. The mathematicization of data structures (as relations), permitting the manipulation of data by a well-defined formal language (predicate logic plus a few other operators, dressed up as SQL), was a Copernican Revolution in the management of data – and will prove itself, over time, to be just as significant, and just as enduring, as that 16th century revolution.

Binding Costs.

Early binding business requirements to database schemas or program code is the single most expensive mistake made in IT development. These mistakes

are made so frequently, I believe, because the focus in developing business applications is on development – not on the costs of maintaining and enhancing applications over their lifetimes – and because early binding does not contribute to the cost of development. Early binding only begins to hurt when we have to change these applications to accommodate changes in the business rules they implement.

Material for Further Development.

1. Binding ontologies to schemas.
Levels of abstraction. Late binding by means of (a) generalization and (b) interpretation provided by structural metadata.

2. Binding cardinalities to foreign keys.
Changing relationship cardinalities is changing the semantics of those relationships. But when the change is from or to a many-to-many relationship, associative tables must be removed and foreign keys embedded in one of the related tables, or vice versa. Such changes in semantics require extensive and usually costly changes to syntax.

3. Appropriately assigning semantics to components of information systems.
The distinction between volatile and stable semantics. Information system components to which semantics may be bound are: database schemas, procedural code, metadata tables, and the background knowledge of the system users.