CHAPTER 9
Numbers

Where would we be without numbers? While those of us who are math-challenged might prefer a text-only view of the world, the reality is that much of the data in any database is numeric. How much inventory do we have? How much money do we owe? At what rate is our business growing? These are just some of the questions that we expect to answer using numbers from databases.

When working with numbers in PL/SQL, you need to have at least a passing familiarity with the following:

- The numeric datatypes at your disposal. It also helps to know in what situations they are best used.
- How to convert between numbers and their textual representations. How else do you expect to get those numbers into and out of your database?
- PL/SQL's rich library of built-in numeric functions. After all, you don't want to reinvent the wheel.

Each of these topics is discussed in this chapter. I'll begin by looking at the datatypes themselves.

Numeric Datatypes

Like the Oracle database, PL/SQL offers a variety of numeric datatypes to suit different purposes:

NUMBER
: A true decimal datatype that is ideal for working with monetary amounts. NUMBER is also the only one of PL/SQL's numeric types to be implemented in a completely platform-independent fashion. Anything you do with NUMBERs should work the same regardless of the underlying hardware.

PLS_INTEGER and BINARY_INTEGER
: Integer datatypes conforming to your hardware's underlying integer representation. Arithmetic is performed using your hardware's native machine instructions. You cannot store values of these types in the database.

SIMPLE_INTEGER
: Introduced with Oracle Database 11*g*. Has the same range as BINARY_INTEGER, but does not allow for NULLs and does not raise an exception if an overflow occurs. The SIMPLE_INTEGER datatype results in significantly faster execution times for natively compiled code.

BINARY_FLOAT and BINARY_DOUBLE
: Single- and double-precision IEEE-754 binary floating-point types. I don't recommend these types for monetary amounts. They are useful, however, when you need fast floating-point arithmetic.

SIMPLE_FLOAT and SIMPLE_DOUBLE
: Introduced with Oracle Database 11*g*. Have the same range as BINARY_FLOAT and BINARY_DOUBLE, but do not allow for NULLs, do not raise an exception if an overflow occurs, and do not support special literals or predicates such as BINARY_FLOAT_MIN_NORMAL, IS NAN, or IS NOT INFINITE. These SIMPLE datatypes result in significantly faster execution times for natively compiled code.

In practice, you may encounter other numeric types, such as FLOAT, INTEGER, and DECIMAL. These are really nothing more than alternate names for the core numeric types just listed. I'll talk about these alternate names in "Numeric Subtypes" on page 256.

The NUMBER Type

The NUMBER datatype is by far the most common numeric datatype you'll encounter in the world of Oracle and PL/SQL programming. Use it to store integer, fixed-point, or floating-point numbers of just about any size. Prior to Oracle Database 10*g*, NUMBER was the only numeric datatype supported directly by the Oracle database engine (later versions also support BINARY_FLOAT and BINARY_DOUBLE). NUMBER is implemented in a platform-independent manner, and arithmetic on NUMBER values yields the same result no matter what hardware platform you run on.

The simplest way to declare a NUMBER variable is simply to specify the keyword NUMBER:

```
DECLARE
   x NUMBER;
```

Such a declaration results in a floating-point NUMBER. The Oracle database will allocate space for up to the maximum of 40 digits, and the decimal point will float to best accommodate whatever value you assign to the variable. NUMBER variables can hold values as small as 10^{-130} (1.0E – 130) and as large as $10^{126} - 1$ (1.0E126 – 1). Values smaller

than 10^{-130} will get rounded down to 0, and calculations resulting in values larger than or equal to 10^{126} will be undefined, causing runtime problems but not raising an exception. This range of values is demonstrated by the following code block:

```
DECLARE
   tiny_nbr NUMBER := 1e-130;
   test_nbr NUMBER;
   --                        11111111112222222222333333333334
   --                        12345678901234567890123456789012345678890
   big_nbr  NUMBER :=         9.99999999999999999999999999999999999999e125;
   --                        11111111112222222222333333333344444444
   --                        12345678901234567890123456789012345678890123456
   fmt_nbr VARCHAR2(50) := '9.99999999999999999999999999999999999999EEEE';
BEGIN
   DBMS_OUTPUT.PUT_LINE('tiny_nbr       =' || TO_CHAR(tiny_nbr, '9.9999EEEE'));
   -- NUMBERs that are too small round down to zero
   test_nbr := tiny_nbr / 1.0001;
   DBMS_OUTPUT.PUT_LINE('tiny made smaller =' || TO_CHAR(test_nbr, fmt_nbr));
   -- NUMBERs that are too large throw an error
   DBMS_OUTPUT.PUT_LINE('big_nbr           =' || TO_CHAR(big_nbr, fmt_nbr));
   test_nbr := big_nbr * 1.0001;        -- too big
   DBMS_OUTPUT.PUT_LINE('big made bigger   =' || TO_CHAR(test_nbr, fmt_nbr));
END;
```

The output from this block is:

```
tiny_nbr          = 1.0000E-130
tiny made smaller =  .000000000000000000000000000000000000000E+00
big_nbr           = 9.99999999999999999999999999999999999999900E+125
big made bigger   =##################################################
```

If you try to explicitly assign a number that is too large to your NUMBER variable, you'll raise a *PLS-00569: numeric overflow or underflow* exception. But if you assign calculation results that exceed the largest legal value, no exception is raised. If your application really needs to work with such large numbers, you will have to code validation routines that anticipate out-of-range values, or consider using BINARY_DOUBLE, which can be compared to BINARY_DOUBLE_INFINITY. Using binary datatypes has rounding implications, so be sure to read the sections on binary datatypes later in this chapter. For most applications, these rounding errors will probably cause you to choose the NUMBER datatype.

Often, when you declare a variable of type NUMBER, you will want to constrain its *precision* and *scale*, as follows:

```
NUMBER (precision, scale)
```

Such a declaration results in a fixed-point number. The *precision* is the total number of significant digits in the number. The *scale* dictates the number of digits to the right (positive scale) or left (negative scale) of the decimal point, and also affects the point at which rounding occurs. Both the *precision* and the *scale* values must be literal integer

values; you cannot use variables or constants in the declaration. Legal values for *precision* range from 1 to 38, and legal values for *scale* range from −84 to 127.

When declaring fixed-point numbers, the value for *scale* is usually less than the value for *precision*. For example, you might declare a variable holding a monetary amount as NUMBER(9,2), which allows values up to and including 9,999,999.99. Figure 9-1 shows how to interpret such a declaration.

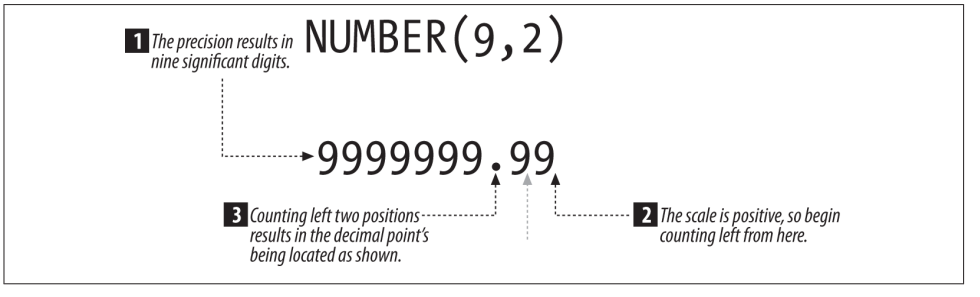

Figure 9-1. A typical fixed-point NUMBER declaration

As this figure illustrates, a declaration of NUMBER(9,2) results in a fixed-point number consisting of seven digits to the left of the decimal point and two digits to the right of the decimal point. Values stored in the variable will be rounded to a maximum of two decimal places, as shown in Table 9-1.

Table 9-1. Rounding of NUMBER(9,2) values

Original value	Rounded value that is actually stored
1,234.56	1,234.56
1,234,567.984623	1,234,567.98
1,234,567.985623	1,234,567.99
1,234,567.995623	1,234,568.00
10,000,000.00	Results in an ORA-06502: PL/SQL: numeric or value error exception, because the precision is too large for the variable
−10,000,000.00	Same error as for 10,000,000.00

The last two values in the table result in an exception because they require more significant digits to represent than the variable can handle. Values in the tens of millions require at least eight significant digits to the left of the decimal point. You can't round such values to fit into only seven digits, so you get overflow errors.

Things get more interesting when you declare a variable with a *scale* that exceeds the variable's *precision* or when you use a negative value for *scale*. Figure 9-2 illustrates the effect of a *scale* exceeding a variable's *precision*.

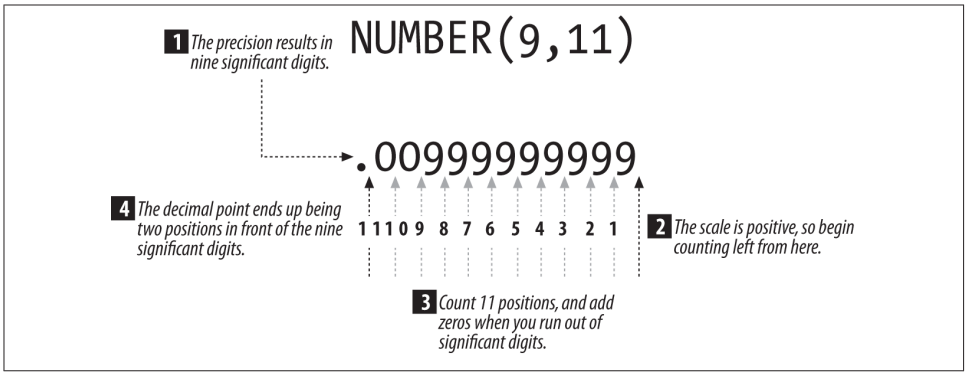

Figure 9-2. The effect of scale exceeding precision

The variable illustrated in this figure has the same number of significant digits as the variable in Figure 9-1, but those significant digits are used differently. Because the *scale* is 11, those nine significant digits can represent only absolute values less than 0.01. Values are rounded to the nearest hundred-billionth. Table 9-2 shows the results of storing some carefully chosen example values into a NUMBER(9,11) variable.

Table 9-2. Rounding of NUMBER(9,11) values

Original value	Rounded value that is actually stored
0.00123456789	0.00123456789
0.000000000005	0.00000000001
0.000000000004	0.00000000000
0.01	Too large a number for the variable; requires a significant digit in the hundredths position; results in an ORA-06502 error
–0.01	Same as for 0.01

Negative *scale* values extend the decimal point out to the right, in the opposite direction of the positive scale. Figure 9-3 illustrates a variable declared NUMBER(9,–11).

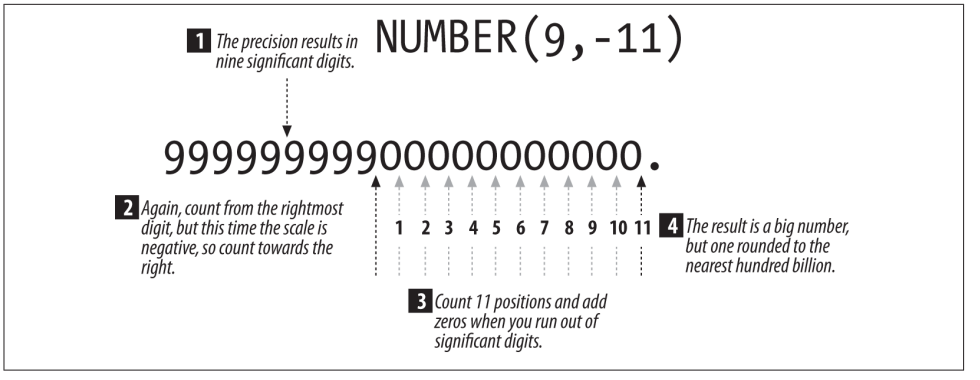

Figure 9-3. The effect of negative scale

Again I've used nine significant digits, but look where the decimal point is now! Rather than small values down to the hundred-billionth, the smallest value I can now represent precisely is 100 billion. Values less than 100 billion are rounded up or down to the nearest 100 billion, as illustrated in Table 9-3.

Table 9-3. Rounding of NUMBER(9,–11) values

Original value	Rounded value that is actually stored
50,000,000,000.123	100,000,000,000
49,999,999,999.999	0
150,000,975,230,001	150,000,000,000,000
100,000,000,000,000,000,000 or 1×10^{20}	Too large a number for the variable; requires a significant digit in the hundred-quintillions position; results in an ORA-06502 error
–100,000,000,000,000,000,000 or -1×10^{20}	Also results in an ORA-06502 error

As Figure 9-3 and Table 9-3 illustrate, negative scales allow me to represent some very large numbers, but at the cost of precision in the less significant digits. Any absolute value less than 50 trillion is rounded to zero when stored in a NUMBER(9,–11) variable.

When declaring NUMBER variables using *precision* and *scale*, bear in mind that *scale* is optional and defaults to zero. For example, the following declarations are equivalent:

```
x NUMBER(9,0);
x NUMBER(9);
```

Both of these declarations result in integer variables (i.e., zero digits past the decimal point) containing nine significant digits. The range of integer values that can be represented using nine significant digits is –999,999,999 through 999,999,999.

When used for fixed-point values, the range of NUMBER is constrained by the values that you are allowed to specify for *precision* and *scale*, as demonstrated in the following code block:

```
DECLARE
   low_nbr  NUMBER(38,127);
   high_nbr NUMBER(38,-84);
BEGIN
   /* 127 is largest scale, so begin with 1 and move
      decimal point 127 places to the left. Easy. */
   low_nbr := 1E-127;
   DBMS_OUTPUT.PUT_LINE('low_nbr = ' || low_nbr);

   /* -84 is smallest scale value. Add 37 to normalize
      the scientific notation, and we get E+121. */
   high_nbr := 9.99999999999999999999999999999999999999E+121;
   DBMS_OUTPUT.PUT_LINE('high_nbr = ' || high_nbr);
END;
```

The output is:

```
low_nbr =
1.00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000E-127
high_nbr =
9.99999999999999999999999999999999999900000000000000000000000000000000000000000000000000000000000000
000000000000000E+121
```

As before, low_nbr represents the low end of the positive range and high_nbr the high end. One difference is that when working with fixed-point numbers, you are limited to 38 significant digits.

Given the wide range and versatility of the NUMBER datatype, it's no wonder that it's so widely used. Using simply NUMBER in your declarations, you can represent floating-point values. By constraining those numbers using *precision* and *scale*, you can represent fixed-point decimal numbers. By setting *scale* to zero or omitting it entirely, you can represent integer values. One datatype covers all the bases.

The PLS_INTEGER Type

The PLS_INTEGER datatype stores signed integers in the range –2,147,483,648 through 2,147,483,647. Values are represented using your hardware platform's native integer format.

Following is an example of some PLS_INTEGER declarations:

```
DECLARE
   loop_counter PLS_INTEGER;
   days_in_standard_year CONSTANT PLS_INTEGER := 365;
   emp_vacation_days PLS_INTEGER DEFAULT 14;
```

The PLS_INTEGER datatype was designed for speed. Prior to Oracle Database 10*g*, PLS_INTEGER was the only integer datatype that used native machine arithmetic. All other numeric datatypes used the C language arithmetic library used with the NUMBER datatype. When you perform arithmetic using PLS_INTEGER values, the Oracle soft-

ware uses native machine arithmetic. As a result, it's faster to manipulate PLS_INTEGER values than it is to manipulate integers in the NUMBER datatype. Because PLS_INTEGER values are integers, you generally won't run into any compatibility issues as you move from one hardware platform to the next.

I recommend that you consider using PLS_INTEGER whenever you're faced with intensive integer arithmetic. Bear in mind, however, that if your use of PLS_INTEGER results in frequent conversions to and from the NUMBER type, you may be better off using NUMBER to begin with. You'll gain the greatest efficiency when you use PLS_INTEGER for integer arithmetic (and for loop counters) in cases where you can avoid multiple conversions to and from the NUMBER type. When this datatype is used in integer arithmetic, the resulting values are rounded to whole numbers, as shown in this example:

```
DECLARE
 int1 PLS_INTEGER;
 int2 PLS_INTEGER;
 int3 PLS_INTEGER;
  nbr  NUMBER;
BEGIN
  int1 := 100;
  int2 := 49;
  int3 := int2/int1;
  nbr  := int2/int1;
  DBMS_OUTPUT.PUT_LINE('integer 49/100 =' || TO_CHAR(int3));
  DBMS_OUTPUT.PUT_LINE('number  49/100 =' || TO_CHAR(nbr));
  int2 := 50;
  int3 := int2/int1;
  nbr  := int2/int1;
  DBMS_OUTPUT.PUT_LINE('integer 50/100 =' || TO_CHAR(int3));
  DBMS_OUTPUT.PUT_LINE('number  50/100 =' || TO_CHAR(nbr));
END;
```

This gives the following output:

```
integer 49/100 =0
number  49/100 =.49
integer 50/100 =1
number  50/100 =.5
```

If the resultant value of integer arithmetic is out of the range of valid values (−2,147,483,648 through 2,147,483,647), you will encounter an *ORA-01426: numeric overflow* error.

The BINARY_INTEGER Type

The BINARY_INTEGER datatype also allows you to store signed integers in a binary format. The semantics of this datatype changed in Oracle Database 10*g* Release 1. Beginning with that release, BINARY_INTEGER is equivalent to PLS_INTEGER. In Ora-

cle9*i* Database Release 2 and earlier releases, BINARY_INTEGER differed from PLS_INTEGER in that Oracle implemented it using platform-independent library code.

Curiously, the package STANDARD looks like it constrains the BINARY_INTEGER type to the values –2,147,483,647 through 2,147,483,647, but I have encountered no exceptions assigning values from –2,147,483,648 through 2,147,483,647, which is a slightly larger range on the negative side:

```
subtype BINARY_INTEGER is INTEGER range '-2147483647'..2147483647;
```

I don't recommend using BINARY_INTEGER for new work. The only reason to use BINARY_INTEGER for new work is if you need your code to run on releases of Oracle prior to 7.3 (before PLS_INTEGER was introduced). I hope you're not running anything that old!

The SIMPLE_INTEGER Type

The SIMPLE_INTEGER datatype was introduced in Oracle Database 11*g*. This datatype is a performance-enhanced version of PLS_INTEGER with a few caveats. The SIMPLE_INTEGER datatype has the same range of values as PLS_INTEGER (–2,147,483,648 through 2,147,483,647), but it does not support NULL values or check for overflow conditions. So, you may be wondering why you would want to use this seemingly defective clone of PLS_INTEGER. Well, if you compile your code natively and your situation is such that your variable will never be NULL and will never overflow, then the SIMPLE_INTEGER type will scream with better performance. Consider this example:

```
/* File on web: simple_integer_demo.sql */
-- First create a compute-intensive procedure using PLS_INTEGER
CREATE OR REPLACE PROCEDURE pls_test (iterations IN PLS_INTEGER)
AS
   int1      PLS_INTEGER := 1;
   int2      PLS_INTEGER := 2;
   begints   timestamp;
   endts     timestamp;
BEGIN
   begints := SYSTIMESTAMP;

   FOR cnt IN 1 .. iterations
   LOOP
      int1 := int1 + int2 * cnt;
   END LOOP;

   endts := SYSTIMESTAMP;
   DBMS_OUTPUT.put_line(   iterations
                        || ' iterations had run time of:'
                        || TO_CHAR (endts - begints));
END;
/
```

```
-- Next, create the same procedure using SIMPLE_INTEGER
CREATE OR REPLACE PROCEDURE simple_test (iterations IN SIMPLE_INTEGER)
AS
   int1      SIMPLE_INTEGER := 1;
   int2      SIMPLE_INTEGER := 2;
   begints   timestamp;
   endts     timestamp;
BEGIN
   begints := SYSTIMESTAMP;

   FOR cnt IN 1 .. iterations
   LOOP
      int1 := int1 + int2 * cnt;
   END LOOP;

   endts := SYSTIMESTAMP;
   DBMS_OUTPUT.put_line(   iterations
                        || ' iterations had run time of:'
                        || TO_CHAR (endts - begints));
END;
/

-- first recompile the procedures to interpreted
ALTER PROCEDURE pls_test COMPILE PLSQL_CODE_TYPE=INTERPRETED;
/

ALTER PROCEDURE simple_test COMPILE PLSQL_CODE_TYPE=INTERPRETED
/

-- compare the run times
BEGIN pls_test(123456789); END;
/
123456789 iterations had run time of:+000000000 00:00:06.375000000

BEGIN simple_test(123456789); END;
/
123456789 iterations had run time of:+000000000 00:00:06.000000000

-- recompile to native code
ALTER PROCEDURE pls_test COMPILE PLSQL_CODE_TYPE=NATIVE
/

ALTER PROCEDURE simple_test COMPILE PLSQL_CODE_TYPE= NATIVE
/

-- compare the run times
BEGIN pls_test(123456789); END;
/
123456789 iterations had run time of:+000000000 00:00:03.703000000

BEGIN simple_test(123456789); END;
```

```
/
123456789 iterations had run time of:+000000000 00:00:01.203000000
```

You can see from this example that SIMPLE_INTEGER gave a slight performance edge with interpreted code (6% in this test on a Microsoft Windows server). Both PLS_INTEGER and SIMPLE_INTEGER are faster when compiled natively, but the native SIMPLE_INTEGER was over 300% faster than the native PLS_INTEGER! As a learning exercise, try this test with a NUMBER type also—I found SIMPLE_INTEGER over 1,000% faster than NUMBER. On a Linux server running Oracle Database 11*g* Release 2, I measured similarly large performance differences using SIMPLE_INTEGER (often several hundred percent faster than alternative numeric types).

The BINARY_FLOAT and BINARY_DOUBLE Types

Oracle Database 10*g* introduced two new floating-point types: BINARY_FLOAT and BINARY_DOUBLE. These types conform to the single- and double-precision floating-point types defined in the IEEE-754 floating-point standard. They are implemented by both PL/SQL and the database engine itself, so you can use them in table definitions as well as in your PL/SQL code. Table 9-4 compares these new types to the venerable NUMBER type.

Table 9-4. Comparison of floating-point types

Characteristic	BINARY_FLOAT	BINARY_DOUBLE	NUMBER
Maximum absolute value	3.40282347E+38F	1.7976931348623157E+308	9.999...999E+121 (38 9s total)
Minimum absolute value	1.17549435E–38F	2.2250748585072014E–308	1.0E–127
Number of bytes used for the value	4 (32 bits)	8 (64 bits)	Varies from 1 to 20
Number of length bytes	0	0	1
Representation	Binary, IEEE-754	Binary, IEEE-754	Decimal
Literal suffix	f	d	None

To write literals of these new types, you apply a suffix—either *f* or *d*, depending on whether you want your literal to be interpreted as a BINARY_FLOAT or as a BINARY_DOUBLE. For example:

```
DECLARE
  my_binary_float  BINARY_FLOAT  := .95f;
  my_binary_double BINARY_DOUBLE := .95d;
  my_number        NUMBER        := .95;
```

There are also some special literals you can use when working with the IEEE-754 floating-point types. The following are supported by both PL/SQL and SQL:

BINARY_FLOAT_NAN, BINARY_DOUBLE_NAN
: Represent "not a number" in single and double precision, respectively.

BINARY_FLOAT_INFINITY, BINARY_DOUBLE_INFINITY
: Represent infinity in single and double precision, respectively.

This next batch of literals are supported *only* by PL/SQL:

BINARY_FLOAT_MIN_NORMAL, BINARY_FLOAT_MAX_NORMAL
: Define the normal range of values you should plan on storing in single- and double-precision variables, respectively.

BINARY_FLOAT_MIN_SUBNORMAL, BINARY_FLOAT_MAX_SUBNORMAL
: Define what is referred to as the *subnormal* range of values. Subnormal values are a part of the IEEE-754 standard that's designed to reduce problems caused by underflow to zero.

Finally, there are some predicates to use with these datatypes:

IS NAN, IS NOT NAN
: Determine whether or not an IEEE-754 value is *not a number*.

IS INFINITE, IS NOT INFINITE
: Determine whether or not an IEEE-754 value represents infinity.

It's *very* important to understand that these BINARY types are indeed binary. I do not recommend them for any situation in which exact decimal representation is critical. The following code block illustrates why, for example, I would not use the new binary types to represent monetary values:

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(0.95f); -- BINARY_FLOAT
    DBMS_OUTPUT.PUT_LINE(0.95d); -- BINARY_DOUBLE
    DBMS_OUTPUT.PUT_LINE(0.95);  -- NUMBER
 END;
```

This example gives us:

```
9.49999988E-001
9.4999999999999996E-001
.95
```

Just as some fractions, such as 1/3, are not possible to represent precisely as decimal numbers, you'll often encounter cases where decimal numbers cannot be represented precisely as binary values. The decimal value 0.95 is just one such case. When dealing with money, use NUMBER.

Be careful when mixing floating-point types in comparisons. For example:

```
BEGIN
   IF 0.95f = 0.95d
   THEN
      DBMS_OUTPUT.PUT_LINE('TRUE');
   ELSE
      DBMS_OUTPUT.PUT_LINE('FALSE');
   END IF;

   IF ABS(0.95f - 0.95d) < 0.000001d
   THEN
      DBMS_OUTPUT.PUT_LINE('TRUE');
   ELSE
      DBMS_OUTPUT.PUT_LINE('FALSE');
   END IF;
END;
```

which results in:

```
FALSE
TRUE
```

This output of FALSE and TRUE, respectively, illustrates the kind of subtle problem you can run into when representing decimal values in binary form. The BINARY_DOUBLE representation of 0.95 has more digits than the BINARY_FLOAT version, and thus the two values do not compare as equal. The second comparison is TRUE because, to compensate for the fact that 0.95 cannot be represented precisely in binary, we arbitrarily accept the two values being compared as equal whenever the magnitude of their difference is less than one one-millionth.

When would you want to use the IEEE-754 types? One reason to use them is for performance, and another is for conformance to IEEE standards. If you are performing extensive numeric computations, you may see a significant increase in performance from using the IEEE-754 types. I ran the following code block, which reports the time needed to compute the area of 500,000 circles and to compute 5,000,000 sines. Both tasks are performed twice, once using BINARY_DOUBLE and once using NUMBER:

```
/* File on web: binary_performance.sql */
DECLARE
   bd BINARY_DOUBLE;
   bd_area BINARY_DOUBLE;
   bd_sine BINARY_DOUBLE;
   nm NUMBER;
   nm_area NUMBER;
   nm_sine NUMBER;
   pi_bd BINARY_DOUBLE := 3.1415926536d;
   pi_nm NUMBER := 3.1415926536;
   bd_begin TIMESTAMP(9);
   bd_end TIMESTAMP(9);
```

```
   bd_wall_time INTERVAL DAY TO SECOND(9);
   nm_begin TIMESTAMP(9);
   nm_end TIMESTAMP(9);
   nm_wall_time INTERVAL DAY TO SECOND(9);
BEGIN
   -- Compute area 5,000,000 times using binary doubles
   bd_begin := SYSTIMESTAMP;
   bd := 1d;
   LOOP
      bd_area := bd * bd * pi_bd;
      bd := bd + 1d;
      EXIT WHEN bd > 5000000;
   END LOOP;
   bd_end := SYSTIMESTAMP;

   -- Compute area 5,000,000 times using NUMBERs
   nm_begin := SYSTIMESTAMP;
   nm := 1;
   LOOP
      nm_area := nm * nm * 2 * pi_nm;
      nm := nm + 1;
      EXIT WHEN nm > 5000000;
   END LOOP;
   nm_end := SYSTIMESTAMP;

   -- Compute and display elapsed wall-clock time
   bd_wall_time := bd_end - bd_begin;
   nm_wall_time := nm_end - nm_begin;
   DBMS_OUTPUT.PUT_LINE('BINARY_DOUBLE area = ' || bd_wall_time);
   DBMS_OUTPUT.PUT_LINE('NUMBER        area = ' || nm_wall_time);

   -- Compute sine 5,000,000 times using binary doubles
   bd_begin := SYSTIMESTAMP;
   bd := 1d;
   LOOP
      bd_sine := sin(bd);
      bd := bd + 1d;
      EXIT WHEN bd > 5000000;
   END LOOP;
   bd_end := SYSTIMESTAMP;

   -- Compute sine 5,000,000 times using NUMBERs
   nm_begin := SYSTIMESTAMP;
   nm := 1;
   LOOP
      nm_sine := sin(nm);
      nm := nm + 1;
      EXIT WHEN nm > 5000000;
   END LOOP;
   nm_end := SYSTIMESTAMP;

   -- Compute and display elapsed wall-clock time for sine
```

```
    bd_wall_time := bd_end - bd_begin;
    nm_wall_time := nm_end - nm_begin;
    DBMS_OUTPUT.PUT_LINE('BINARY_DOUBLE sine = ' || bd_wall_time);
    DBMS_OUTPUT.PUT_LINE('NUMBER        sine = ' || nm_wall_time);
END;
```

My results, which were reasonably consistent over multiple runs, looked like this:

```
BINARY_DOUBLE area = +00 00:00:02.792692000
NUMBER        area = +00 00:00:08.942327000
BINARY_DOUBLE sine = +00 00:00:04.149930000
NUMBER        sine = +00 00:07:37.596783000
```

Be careful with benchmarks, including those I've just shown! As this example illustrates, the range of possible performance gains from using an IEEE-754 type over NUMBER is quite vast. Using BINARY_DOUBLE, you can compute the area of a circle 5 million times in approximately 40% of the time it takes when using NUMBER. If you decide to compute sine 5 million times, however, you can get that done in 0.9% of the time. The gain you get in a given situation depends on the computations involved. The message to take away here is *not* that IEEE-754 types will get things done a fixed percentage faster than NUMBER. Rather, it is that the potential performance improvement from using IEEE-754 types instead of NUMBER is well worth considering and investigating when you're performing extensive calculations.

There are, however, a few areas in which Oracle's implementation of binary floating-point types do not conform perfectly to the IEEE-754 standard. For example, Oracle coerces −0 to +0, whereas the IEEE-754 standard does not call for that behavior. If conformance is important to your application, check the section on "Datatypes" in Oracle's *SQL Reference* manual for the precise details on how and when Oracle diverges from the IEEE-754 standard.

Mixing the Floating-Point Types

Oracle enforces an order of precedence on the implicit conversion of floating-point types. From highest to lowest priority, that precedence is BINARY_DOUBLE, BINARY_FLOAT, and NUMBER. When you write an expression containing a mix of these types, the database attempts to convert all values in the expression to the highest precedence type found in the expression. For example, if you mix BINARY_FLOAT and NUMBER, Oracle first converts all values to BINARY_FLOAT.

If you don't want the database to perform these implicit conversions, you should use the functions TO_NUMBER, TO_BINARY_FLOAT, and TO_BINARY_DOUBLE. For example:

```
DECLARE
   nbr NUMBER := 0.95;
   bf BINARY_FLOAT := 2;
   nbr1 NUMBER;
```

```
    nbr2 NUMBER;
BEGIN
    -- Default precedence, promote to binary_float
    nbr1 := nbr * bf;

    -- Demote BINARY_FLOAT to NUMBER instead
    nbr2 := nbr * TO_NUMBER(bf);

    DBMS_OUTPUT.PUT_LINE(nbr1);
    DBMS_OUTPUT.PUT_LINE(nbr2);
END;
```

This results in:

```
1.89999998
1.9
```

To avoid ambiguity and possible errors involving implicit conversions, I recommend explicit conversions, such as with the functions TO_NUMBER, TO_BINARY_FLOAT, and TO_BINARY_DOUBLE.

The SIMPLE_FLOAT and SIMPLE_DOUBLE Types

The SIMPLE_FLOAT and SIMPLE_DOUBLE datatypes were introduced in Oracle Database 11*g*. These datatypes are performance-enhanced versions of the BINARY_FLOAT and BINARY_DOUBLE datatypes—but they have even more caveats than the SIMPLE_INTEGER type. The SIMPLE_FLOAT and SIMPLE_DOUBLE datatypes have the same range of values as BINARY_FLOAT and BINARY_DOUBLE, but they do not support NULL values, the special IEEE literals (BINARY_FLOAT_NAN, BINARY_DOUBLE_INFINITY, etc.), or the special IEEE predicates (IS NAN, IS INFINITY, etc.). They also do not check for overflow conditions. Like the SIMPLE_INTEGER type, though, under the right conditions these speedy cousins will make your code much faster when compiled natively.

Numeric Subtypes

Oracle also provides several *numeric subtypes*. Most of the time, these subtypes are simply alternate names for the basic types I have just discussed. These alternate names offer compatibility with ISO SQL, SQL/DS, and DB2 datatypes. They usually have the same range of legal values as their base types, but some offer additional functionality by restricting values to a subset of those supported by their base types. These subtypes are described in Table 9-5.

Table 9-5. Predefined numeric subtypes

Subtype	Compatibility	Corresponding Oracle datatype/notes
DEC (*precision*, *scale*)	ANSI	NUMBER (*precision*, *scale*)
DECIMAL (*precision*, *scale*)	IBM	NUMBER (*precision*, *scale*)
DOUBLE PRECISION	ANSI	NUMBER, with 126 binary digits of precision
FLOAT	ANSI, IBM	NUMBER, with 126 binary digits of precision
FLOAT (*binary_precision*)	ANSI, IBM	NUMBER, with a *binary_precision* of up to 126 (the default)
INT	ANSI	NUMBER(38)
INTEGER	ANSI, IBM	NUMBER(38)
NATURAL	N/A	PLS_INTEGER,[a] but allows only nonnegative values (0 and higher)
NATURALN	N/A	Same as NATURAL, but with the additional restriction of never being NULL
NUMERIC (*precision*, *scale*)	ANSI	NUMBER (*precision*, *scale*)
POSITIVE	N/A	PLS_INTEGER, but allows only positive values (1 and higher)
POSITIVEN	N/A	Same as POSITIVE, but with the additional restriction of never being NULL
REAL	ANSI	NUMBER, with 63 binary digits of precision
SIGNTYPE	N/A	PLS_INTEGER, limited to the values −1, 0, and 1
SMALLINT	ANSI, IBM	NUMBER(38)

[a] BINARY_INTEGER prior to Oracle Database 10*g*.

The NUMERIC, DECIMAL, and DEC datatypes can declare only fixed-point numbers. DOUBLE PRECISION and REAL are equivalent to NUMBER. FLOAT allows floating decimal points with binary precisions that range from 63 to 126 bits. I don't find it all that useful to define a number's precision in terms of bits rather than digits, though. I also don't find much use for the ISO/IBM-compatible subtypes, and I don't believe you will either.

The subtypes that I sometimes find useful are the PLS_INTEGER subtypes. NATURAL and POSITIVE are both subtypes of PLS_INTEGER. These subtypes constrain the values you can store in a variable, and their use can make a program more self-documenting. For example, if you have a variable whose values must always be nonnegative, you can declare that variable to be NATURAL (0 and higher) or POSITIVE (1 and higher), improving the self-documenting aspect of your code.

Number Conversions

Computers work with numbers best when those numbers are in some kind of binary format. We humans, on the other hand, prefer to see our numbers in the form of character strings containing digits, commas, and other punctuation. PL/SQL allows you to convert numbers back and forth between human- and machine-readable form. You'll usually perform such conversions using the TO_CHAR and TO_NUMBER functions.

When working with the IEEE-754 binary floating-point types, use TO_BINARY_FLOAT and TO_BINARY_DOUBLE. To simplify the discussion that follows, I'll generally refer only to TO_NUMBER. Please assume that any unqualified references to TO_NUMBER also apply to the TO_BINARY_FLOAT and TO_BINARY_DOUBLE functions.

The TO_NUMBER Function

The TO_NUMBER function explicitly converts both fixed- and variable-length strings as well as IEEE-754 floating-point types to the NUMBER datatype using an optional format mask. Use TO_NUMBER whenever you need to convert character string representations of numbers into their corresponding numeric values. Invoke TO_NUMBER as follows:

```
TO_NUMBER(string [,format [,nls_params]])
```

where:

string
: Is a string or BINARY_DOUBLE expression containing the representation of a number.

When using TO_BINARY_FLOAT and TO_BINARY_DOUBLE, you may use the strings 'INF' and '-INF' to represent positive and negative infinity. You may also use 'NaN' to represent "not a number." These special strings are case insensitive.

format
: Is an optional format mask that specifies how TO_NUMBER should interpret the character representation of the number contained in the first parameter if it is a string expression.

nls_params
: Is an optional string specifying various National Language Support (NLS) parameter values. You can use this to override your current session-level NLS parameter settings.

Using TO_NUMBER with no format

In many straightforward cases, you can use TO_NUMBER to convert strings to numbers without specifying any format string at all. For example, all of the following conversions work just fine:

```
DECLARE
   a NUMBER;
```

```
   b NUMBER;
   c NUMBER;
   d NUMBER;
   e BINARY_FLOAT;
   f BINARY_DOUBLE;
   g BINARY_DOUBLE;

   n1 VARCHAR2(20) := '-123456.78';
   n2 VARCHAR2(20) := '+123456.78';
BEGIN
   a := TO_NUMBER('123.45');
   b := TO_NUMBER(n1);
   c := TO_NUMBER(n2);
   d := TO_NUMBER('1.25E2');
   e := TO_BINARY_FLOAT('123.45');
   f := TO_BINARY_DOUBLE('inf');
   g := TO_BINARY_DOUBLE('NAN');
END;
```

Generally, you should be able to use TO_NUMBER without specifying a format when the following conditions apply:

- Your number is represented using only digits and a single decimal point.
- Any sign is leading, and must be either minus (–) or plus (+). If no sign is present, the number is assumed to be positive.
- Scientific notation is used—for example, 1.25E2.

If your character strings don't meet these criteria or if you need to round values to a specific number of decimal digits, then you need to invoke TO_NUMBER with a format model.

Using TO_NUMBER with a format model

Using TO_NUMBER with a format model enables you to deal with a much wider range of numeric representations than TO_NUMBER would otherwise recognize. Table B-1 (in Appendix B) gives a complete list of all supported number format model elements. For example, you can specify the locations of group separators and the currency symbol:

```
a := TO_NUMBER('$123,456.78','L999G999D99');
```

You don't necessarily need to specify the exact number of digits in your format model. TO_NUMBER is forgiving in this respect, as long as your model contains more digits than are in your actual value. For example, the following will work:

```
a := TO_NUMBER('$123,456.78','L999G999G999D99');
```

However, if you have more digits to the left or to the right of the decimal point than your format allows, the conversion will fail with an *ORA-06502: PL/SQL: numeric or*

value error. The first of the following conversions will fail because the string contains 10 digits to the left of the decimal, while the format calls for only 9. The second conversion will fail because there are too many digits to the right of the decimal point:

```
a := TO_NUMBER('$1234,567,890.78','L999G999G999D99');
a := TO_NUMBER('$234,567,890.789','L999G999G999D99');
```

You can force leading zeros using the 0 format element:

```
a := TO_NUMBER('001,234','000G000');
```

You can recognize angle-bracketed numbers as negative numbers using the PR element:

```
a := TO_NUMBER('<123.45>','999D99PR');
```

However, not all format elements can be used to convert strings to numbers. Some elements, such as RN for Roman numerals, are output only. The following attempt to convert the Roman numeral representation of a value to a number will fail:

```
a := TO_NUMBER('cxxiii','rn');
```

EEEE is another output-only format, but that's OK because you don't need it to convert values that are correctly represented in scientific notation. You can simply do:

```
a := TO_NUMBER('1.23456E-24');
```

Passing NLS settings to TO_NUMBER

Many of the number format model elements listed in Table B-1 ultimately derive their meaning from one of the NLS parameters. For example, the G element represents the numeric group separator, which is the second character in the NLS_NUMERIC_CHARACTERS setting in effect when the conversion takes place. You can view current NLS parameter settings by querying the NLS_SESSION_PARAMETERS view:

```
SQL> SELECT * FROM nls_session_parameters;

PARAMETER                VALUE
------------------------ ---------------
NLS_LANGUAGE             AMERICAN
NLS_TERRITORY            AMERICA
NLS_CURRENCY             $
NLS_ISO_CURRENCY         AMERICA
NLS_NUMERIC_CHARACTERS   .,
NLS_CALENDAR             GREGORIAN
NLS_DATE_FORMAT          DD-MON-RR
```

Some NLS parameter settings are by default dependent on others. For example, set NLS_TERRITORY to AMERICA, and Oracle defaults NLS_NUMERIC_CHARACTERS TO '.,'. If you need to, you can then override the NLS_NUMERIC_CHARACTERS setting (using an ALTER SESSION command, for example).

On rare occasions, you may want to override specific NLS parameter settings for a single call to TO_NUMBER. In the following example, I invoke TO_NUMBER and specify NLS settings corresponding to NLS_TERRITORY=FRANCE:

```
a := TO_NUMBER('F123.456,78','L999G999D99',
               'NLS_NUMERIC_CHARACTERS='',.'''
               || ' NLS_CURRENCY=''F'''
               || ' NLS_ISO_CURRENCY=FRANCE');
```

Because my NLS parameter string is so long, I've broken it up into three separate strings concatenated together so that the example fits nicely on the page. Note my doubling of quote characters. The setting I want for NLS_NUMERIC_CHARACTERS is:

```
NLS_NUMERIC_CHARACTERS=',.'
```

I need to embed this setting into my NLS parameter string, and to embed quotes within a string I must double them, so I end up with:

```
'NLS_NUMERIC_CHARACTERS='',.'''
```

The three NLS parameters set in this example are the only three you can set via TO_NUMBER. I don't know why that is. It certainly would be much more convenient if you could simply do the following:

```
a := TO_NUMBER('F123.456,78','L999G999D99','NLS_TERRITORY=FRANCE');
```

But unfortunately, NLS_TERRITORY is not something you can set via a call to TO_NUMBER. You are limited to specifying NLS_NUMERIC_CHARACTERS, NLS_CURRENCY, and NLS_ISO_CURRENCY.

For detailed information on setting the various NLS parameters, see Oracle's *Globalization Support Guide*.

Avoid using the third argument to TO_NUMBER; I believe it's better to rely on session settings to drive the way in which PL/SQL interprets format model elements such as L, G, and D. Instead of your having to hardcode such information throughout your programs, session settings can be controlled by the user outside the bounds of your code.

The TO_CHAR Function

The TO_CHAR function is the converse of TO_NUMBER, and converts numbers to their character representations. Using an optional format mask, you can be quite specific about the form those character representations take. Invoke TO_CHAR as follows:

```
TO_CHAR(number [,format [,nls_params]])
```

where:

number
: Is a number that you want to represent in character form. This number may be any of PL/SQL's numeric types: NUMBER, PLS_INTEGER, BINARY_INTEGER, BINARY_FLOAT, BINARY_DOUBLE, SIMPLE_INTEGER, SIMPLE_FLOAT, and SIMPLE_DOUBLE.

format
: Is an optional format mask that specifies how TO_CHAR should present the number in character form.

nls_params
: Is an optional string specifying various NLS parameter values. You can use this to override your current session-level NLS parameter settings.

If you want your results to be in the national character set, you can use TO_NCHAR in place of TO_CHAR. In that case, be certain you provide your number format string in the national character set as well. Otherwise, you may receive output consisting of all number signs: #.

Using TO_CHAR with no format

As with TO_NUMBER, you can invoke TO_CHAR without specifying a format mask:

```
DECLARE
   b VARCHAR2(30);
BEGIN
   b := TO_CHAR(123456789.01);
   DBMS_OUTPUT.PUT_LINE(b);
END;
```

The output is:

```
123456789.01
```

Unlike the situation with TO_NUMBER, you aren't likely to find this use of TO_CHAR very useful. At the very least, you may want to format your numeric output with group separators to make it more readable.

Using TO_CHAR with a format model

When converting numbers to their character string equivalents, you'll most often invoke TO_CHAR with a format model. For example, you can output a monetary amount as follows:

```
DECLARE
   b VARCHAR2(30);
BEGIN
   b := TO_CHAR(123456789.01,'L999G999G999D99');
```

```
   DBMS_OUTPUT.PUT_LINE(b);
END;
```

The output (in the United States) is:

```
$123,456,789.01
```

The format model elements in Table B-1 (in Appendix B) give you a lot of flexibility, and you should experiment with them to learn the finer points of how they work. The following example specifies that leading zeros be maintained, but the B format element is used to force any zero values to blanks. Notice that the B element precedes the number elements (the 0s) but follows the currency indicator (the L):

```
DECLARE
   b VARCHAR2(30);
   c VARCHAR2(30);
BEGIN
   b := TO_CHAR(123.01,'LB000G000G009D99');
   DBMS_OUTPUT.PUT_LINE(b);

   c := TO_CHAR(0,'LB000G000G009D99');
   DBMS_OUTPUT.PUT_LINE(c);
END;
```

The output is:

```
$000,000,123.01
```

You see only one line of output from this example, and that's from the first conversion. The second conversion involves a zero value, and the B format element causes TO_CHAR to return that value as a blank string, even though the format otherwise specifies that leading zeros be returned. As an experiment, try this same example on your system, but leave off the B.

Not all combinations of format elements are possible. For example, you can't use LRN to place a currency symbol in front of a value expressed in Roman numerals. Oracle doesn't document every such nuance. It takes some experience and some experimenting to get a feel for what's possible and what's not.

The V format element

The V format element is unusual enough to warrant a special explanation. The V element allows you to scale a value, and its operation is best explained through an illustration, which you'll find in Figure 9-4.

Why would you ever need such functionality? Look no further than the stock market for an example. The standard trading unit for stocks is 100 shares, and stock sales are sometimes reported in terms of the number of 100-share units sold. Thus, a sales figure of 123 actually represents 123 units of 100 shares, or 12,300 shares. The following ex-

ample shows how V can be used to scale a value such as 123 in recognition of the fact that it really represents 100s:

```
DECLARE
   shares_sold NUMBER := 123;
BEGIN
   DBMS_OUTPUT.PUT_LINE(
      TO_CHAR(shares_sold,'999G9V99')
   );
END;
```

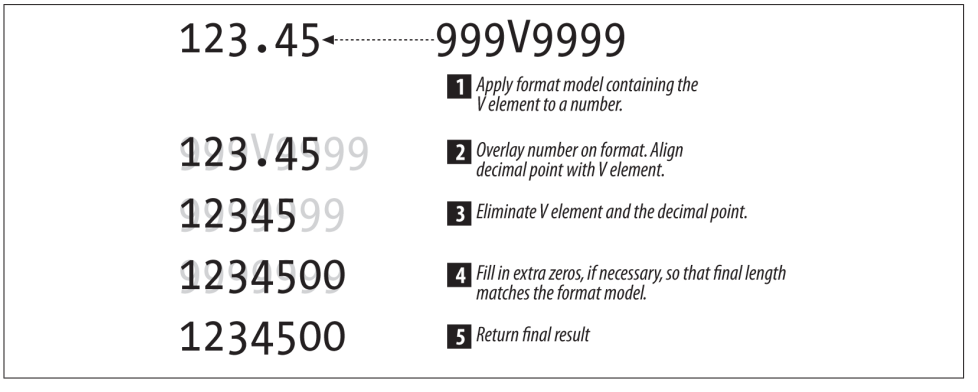

Figure 9-4. The V number format element

The output is:

```
12,300
```

Notice that the format model in this example includes the G element to specify the location of the group separator (the comma) in the displayed number. You can specify group separators only to the left of the V element, not to the right. This is unfortunate. Consider the following perfectly reasonable format model:

```
TO_CHAR(123.45,'9G99V9G999');
```

You would hope to get the result formatted as 1,234,500. However, the G to the right of the V is invalid. You can use 9G99V9999 to get a result of 1,234500, or you can use 999V9999 to get a result of 1234500. Neither result is as readable as you would no doubt like it to be.

You probably won't use the V element very often, but it's worth knowing about this bit of interesting functionality.

Rounding when converting numbers to character strings

When converting character strings to numbers, you'll get an error any time you have more digits to the left or right of the decimal point than the format model allows. When

converting numbers to characters, however, you'll get an error only if the number requires more digits to the left of the decimal point than the format model allows. If you specify fewer decimal digits (i.e., digits to the right of the decimal point) in your format model than the number requires, the number will be rounded so that the fractional portion fits your model.

When a conversion fails because the model doesn't specify enough digits to the left of the decimal point, TO_CHAR returns a string of number signs (#). For example, the following conversion fails because 123 doesn't fit into two digits:

```
SQL> DECLARE
  2    b VARCHAR2(30);BEGIN
  3    b := TO_CHAR(123.4567,'99.99');
  4    DBMS_OUTPUT.PUT_LINE(b);END;

######
```

It's perfectly OK, however, for your model not to include enough digits to cover the fractional portion of a value. In such cases, rounding occurs. For example:

```
SQL> BEGIN
  2     DBMS_OUTPUT.PUT_LINE(TO_CHAR(123.4567,'999.99'));
  3     DBMS_OUTPUT.PUT_LINE(TO_CHAR(123.4567,'999'));END;

123.46
123
```

Digits 5 and higher are rounded up, which is why 123.4567 is rounded up to 123.46. Digits less than 5 are rounded down, so 123.4xxx will always be rounded down to 123.

Dealing with spaces when converting numbers to character strings

A reasonably common problem encountered when converting numbers to character strings is that TO_CHAR always leaves room for the minus sign, even when numbers are positive. By default, TO_CHAR will leave one space in front of a number for use by a potential minus sign (–):

```
DECLARE
   b VARCHAR2(30);
   c VARCHAR2(30);
BEGIN
   b := TO_CHAR(-123.4,'999.99');
   c := TO_CHAR(123.4,'999.99');
   DBMS_OUTPUT.PUT_LINE(':' || b || ' ' || TO_CHAR(LENGTH(b)));
   DBMS_OUTPUT.PUT_LINE(':' || c || ' ' || TO_CHAR(LENGTH(c)));
END;
```

The output is:

```
:-123.40 7
: 123.40 7
```

Notice that both converted values have the same length, seven characters, even though the positive number requires only six characters when displayed in character form. That leading space can be a big help if you are trying to get columns of numbers to line up. However, it can be a bit of a pain if for some reason you need a compact number with no spaces whatsoever.

Use the PR element, and your positive numbers will have one leading space and one trailing space to accommodate the potential enclosing angle brackets. Spaces will be left to accommodate whatever sign indicator you choose in your format model.

There are a couple of approaches you can take if you really need your numbers converted to characters without leading or trailing spaces. One approach is to use the TM format model element to get the "text minimum" representation of a number:

```
DECLARE
   b VARCHAR2(30);
   c VARCHAR2(30);
BEGIN
   b := TO_CHAR(-123.4,'TM9');
   c := TO_CHAR(123.4,'TM9');
   DBMS_OUTPUT.PUT_LINE(':' || b || ' ' || TO_CHAR(LENGTH(b)));
   DBMS_OUTPUT.PUT_LINE(':' || c || ' ' || TO_CHAR(LENGTH(c)));
END;
```

The output is:

```
:-123.4 6
:123.4 5
```

The TM approach works, but doesn't allow you to specify any other formatting information. You can't, for example, specify TM999.99 in order to get a fixed two-decimal digit. If you need to specify other formatting information or if TM is not available in your release of PL/SQL, you'll need to trim the results of the conversion:

```
DECLARE
   b VARCHAR2(30);
   c VARCHAR2(30);
BEGIN
   b := LTRIM(TO_CHAR(-123.4,'999.99'));
   c := LTRIM(TO_CHAR(123.4,'999.99'));
   DBMS_OUTPUT.PUT_LINE(':' || b || ' ' || TO_CHAR(LENGTH(b)));
   DBMS_OUTPUT.PUT_LINE(':' || c || ' ' || TO_CHAR(LENGTH(c)));
END;
```

The output is:

```
:-123.40 7
:123.40 6
```

Here I've used LTRIM to remove any potential leading spaces, and I've successfully preserved our fixed two digits to the right of the decimal point. Use RTRIM if you are placing the sign to the right of the number (e.g., via the MI element) or TRIM if you are using something like PR that affects both sides of the number.

Passing NLS settings to TO_CHAR

As with TO_NUMBER, you have the option of passing a string of NLS parameter settings to TO_CHAR. For example:

```
BEGIN
   DBMS_OUTPUT.PUT_LINE(
      TO_CHAR(123456.78,'999G999D99','NLS_NUMERIC_CHARACTERS='',.''')
   );
END;
```

The output is:

```
123.456,78
```

The three NLS parameters you can set this way are NLS_NUMERIC_CHARACTERS, NLS_CURRENCY, and NLS_ISO_CURRENCY. See "Passing NLS settings to TO_NUMBER" on page 260 for an example of all three being set at once.

The CAST Function

The CAST function is used to convert numbers to strings and vice versa. The general format of the CAST function is as follows:

```
CAST (expression AS datatype)
```

The following example shows CAST being used first to convert a NUMBER to a VARCHAR2 string, and then to convert the characters in a VARCHAR2 string into their corresponding numeric value:

```
DECLARE
   a NUMBER := -123.45;
   a1 VARCHAR2(30);
   b VARCHAR2(30) := '-123.45';
   b1 NUMBER;
   b2 BINARY_FLOAT;
   b3 BINARY_DOUBLE;
BEGIN
   a1 := CAST (a AS VARCHAR2);
   b1 := CAST (b AS NUMBER);
   b2 := CAST (b AS BINARY_FLOAT);
   b3 := CAST (b AS BINARY_DOUBLE);
   DBMS_OUTPUT.PUT_LINE(a1);
   DBMS_OUTPUT.PUT_LINE(b1);
   DBMS_OUTPUT.PUT_LINE(b2);
```

```
   DBMS_OUTPUT.PUT_LINE(b3);
END;
```

The output is:

```
-123.45
-123.45
-1.23449997E+002
-1.2345E+002
```

CAST has the disadvantage of not supporting the use of number format models. An advantage of CAST, however, is that it is part of the ISO SQL standard, whereas the TO_CHAR and TO_NUMBER functions are not. If writing 100% ANSI-compliant code is important to you, you should investigate the use of CAST. Otherwise, I recommend using the traditional TO_NUMBER and TO_CHAR functions.

Because PL/SQL is not part of the ISO standard, it is by definition not possible to write 100% ISO-compliant PL/SQL code, so CAST seems to bring no real benefit to PL/SQL number conversions. CAST can, however, be used in the effort to write 100% ISO-compliant SQL statements (such as SELECT, INSERT, etc.).

Implicit Conversions

A final method of handling conversions between numbers and strings is to just leave it all to PL/SQL. Such conversions are referred to as *implicit conversions*, because you don't explicitly specify them in your code. Following are some straightforward implicit conversions that will work just fine:

```
DECLARE
   a NUMBER;
   b VARCHAR2(30);
BEGIN
   a := '-123.45';
   b := -123.45;
...
```

As I mentioned in Chapter 7, I have several problems with implicit conversions. I'm a strong believer in maintaining control over my code, and when you use an implicit conversion you are giving up some of that control. You should always know when conversions are taking place, and the best way to do that is to code them explicitly. Don't just let them happen. If you rely on implicit conversions, you lose track of when conversions are occurring, and your code is less efficient as a result. Explicit conversions also make your intent clear to other programmers, making your code more self-documenting and easier to understand.

Another problem with implicit conversions is that while they may work just fine (or seem to) in simple cases, sometimes they can be ambiguous. Consider the following:

```
DECLARE
   a NUMBER;
BEGIN
   a := '123.400' || 999;
```

What value will the variable *a* hold when this code executes? It all depends on how PL/SQL evaluates the expression on the right side of the assignment operator. If PL/SQL begins by converting the string to a number, you'll get the following result:

```
a := '123.400' || 999;
a := 123.4 || 999;
a := '123.4' || '999';
a := '123.4999';
a := 123.4999;
```

On the other hand, if PL/SQL begins by converting the number to a string, you'll get the following result:

```
a := '123.400' || 999;
a := '123.400' || '999';
a := '123.400999';
a := 123.400999;
```

Which is it? Do you know? Even if you *do* know, do you really want to leave future programmers guessing and scratching their heads when they look at your code? It would be much clearer, and therefore better, to write the conversion explicitly:

```
a := TO_NUMBER('123.400' || TO_CHAR(999));
```

This expression, by the way, represents how the database will evaluate the original example. Isn't it much easier to understand at a glance now that I've expressed the conversions explicitly?

Beware of Implicit Conversions!

In "The BINARY_FLOAT and BINARY_DOUBLE Types" on page 251, I showed some code (*binary_performance.sql*) that I used to compare the performance of BINARY_DOUBLE and NUMBER. When I first wrote that test, I coded the loops to compute area as follows:

```
DECLARE
   bd BINARY_DOUBLE;
  ...
BEGIN
   ...
   FOR bd IN 1..1000000 LOOP

     bd_area := bd**2 * pi_bd;
  END LOOP;
  ...
```

I was dumbfounded when my results initially showed that computations involving NUMBER were much faster than those involving BINARY_DOUBLE. I couldn't understand this, as I "knew" that the BINARY_DOUBLE arithmetic was all done in hardware, and therefore it should have been faster than NUMBER.

What I failed to discern, until someone at Oracle Corporation pointed out my blunder, was that my FOR loop (just shown) resulted in the implicit declaration of a PLS_INTEGER loop variable named bd. This new declaration of bd had a scope encompassing the loop block, and masked my declaration of bd as a BINARY_DOUBLE. Further, I wrote the constant value as 2, rather than as 2d, thereby making it a NUMBER. Thus, bd was first implicitly converted to a NUMBER, then raised to the power of 2, and the resulting NUMBER then had to be implicitly converted again into a BINARY_DOUBLE in order to be multiplied by pi_bd. No wonder my results were so poor! Such are the dangers inherent in implicit conversions.

Numeric Operators

PL/SQL implements several operators that are useful when working with numbers. The operators that can be used with numbers are shown in Table 9-6, in order of precedence. The operators with lower precedence evaluate first, while those with a higher precedence evaluate later. For full details on a particular operator, consult Oracle's *SQL Reference* manual.

Table 9-6. Numeric operators and precedence

Operator	Operation	Precedence
**	Exponentiation	1
+	Identity	2
–	Negation	2
*	Multiplication	3
/	Division	3
+	Addition	4
–	Subtraction	4
=	Equality	5
<	Less than	5
>	Greater than	5
<=	Less than or equal to	5
>=	Greater than or equal to	5
<>, !=, ~=, ^=	Not equal	5
IS NULL	Nullity	5
BETWEEN	Inclusive range	5

Operator	Operation	Precedence
NOT	Logical negation	6
AND	Conjunction	7
OR	Inclusion	8

Numeric Functions

PL/SQL implements several functions that are useful when you're working with numbers. You've already seen the conversion functions TO_CHAR, TO_NUMBER, TO_BINARY_FLOAT, and TO_BINARY_DOUBLE. The next few subsections briefly describe several other useful functions. For full details on a particular function, consult Oracle's *SQL Reference* manual.

Rounding and Truncation Functions

There are four different numeric functions that perform rounding and truncation actions: CEIL, FLOOR, ROUND, and TRUNC. It is easy to get confused about which to use in a particular situation. Table 9-7 compares these functions, and Figure 9-5 illustrates their use for different values and decimal place rounding.

Table 9-7. Comparison of functions that perform rounding and truncation actions

Function	Summary
CEIL	Returns the smallest integer that is greater than or equal to the specified value. This integer is the "ceiling" over your value.
FLOOR	Returns the largest integer that is less than or equal to the specified value. This integer is the "floor" under your value.
ROUND	Performs rounding on a number. You can round with a positive number of decimal places (the number of digits to the right of the decimal point) and also with a negative number of decimal places (the number of digits to the left of the decimal point).
TRUNC	Truncates a number to the specified number of decimal places. TRUNC simply discards all values beyond the number of decimal places provided in the call.

	1.75	1.3	55.56	55.56	10	Input
Function	0	0	1	-1	2	Number of decimal places
ROUND	2	1	55.6	60	10	
TRUNC	1	1	55.5	50	10	
FLOOR	1	1	55	55	10	
CEIL	2	2	56	56	10	

Figure 9-5. Impact of rounding and truncation functions

Trigonometric Functions

Many trigonometric functions are available from PL/SQL. When using them, be aware that all angles are expressed in radians, not in degrees. You can convert between radians and degrees as follows:

```
radians = pi * degrees / 180 -- From degrees to radians
degrees = radians * 180 / pi -- From radians to degrees
```

PL/SQL does not implement a function for π (pi) itself. However, you can obtain the value for π through the following call:

```
ACOS(-1)
```

The inverse cosine (ACOS) of –1 is defined as exactly π. Of course, because π is a never-ending decimal number, you always have to work with an approximation. Use the ROUND function if you want to round the results of ACOS(–1) to a specific number of decimal places.

Numeric Function Quick Reference

The following list briefly describes each of PL/SQL's built-in numeric functions. Where applicable, functions are overloaded for different numeric types. For example:

ABS
: Is overloaded for BINARY_DOUBLE, BINARY_FLOAT, NUMBER, SIMPLE_INTEGER, SIMPLE_FLOAT, SIMPLE_DOUBLE, and PLS_INTEGER, because you can take the absolute value of both floating-point and integer values

BITAND
: Is overloaded for PLS_INTEGER and INTEGER (a subtype of NUMBER), because the function is designed to AND only integer values

CEIL
: Is overloaded for BINARY_DOUBLE, BINARY_FLOAT, and NUMBER, because CEIL is a function that doesn't really apply to integers

To check what types a given function is overloaded for, DESCRIBE the built-in package SYS.STANDARD, like this:

```
SQL> DESCRIBE SYS.STANDARD ...full output trimmed for brevity...

FUNCTION CEIL RETURNS NUMBER
 Argument Name                  Type                    In/Out Default?
 ------------------------------ ----------------------- ------ --------
 N                              NUMBER                  IN
FUNCTION CEIL RETURNS BINARY_FLOAT
 Argument Name                  Type                    In/Out Default?
 ------------------------------ ----------------------- ------ --------
 F                              BINARY_FLOAT            IN
```

```
FUNCTION CEIL RETURNS BINARY_DOUBLE
 Argument Name                  Type                    In/Out Default?
 ------------------------------ ----------------------- ------ --------
 D                              BINARY_DOUBLE           IN
```

Almost all the functions in the following list are defined in the built-in package. SYS.STANDARD. BIN_TO_NUM is the one exception that I've noticed. For complete documentation of a given function, refer to Oracle's *SQL Reference* manual.

ABS(n)
: Returns the absolute value of *n*.

ACOS(n)
: Returns the inverse cosine of *n*, where *n* must be between −1 and 1. The value returned by ACOS is between 0 and π.

ASIN(n)
: Returns the inverse sine, where *n* must be between −1 and 1. The value returned by ASIN is between −π/2 and π/2.

ATAN(n)
: Returns the inverse tangent, where the number *n* must be between −infinity and infinity. The value returned by ATAN is between −π/2 and π/2.

ATAN2(n, m)
: Returns the inverse tangent of *n*/*m*, where the numbers *n* and *m* must be between −infinity and infinity. The value returned by ATAN is between −π and π. The result of ATAN2(*n*,*m*) is defined to be identical to ATAN(*n*/*m*).

BIN_TO_NUM(b1, b2, ... bn)
: Converts the bit vector represented by *b1* through *bn* into a number. Each of *b1* through *bn* must evaluate to either 0 or 1. For example, BIN_TO_NUM(1,1,0,0) yields 12.

BITAND(n, m)
: Performs a logical AND between *n* and *m*. For example, BITAND(12,4) yields 4, indicating that the value 12 (binary 1100) has the 4 bit set. Similarly, BITAND(12,8) yields 8, indicating that the 8 bit is also set.

  You'll find it easiest to work with BITAND if you confine yourself to positive integers. Values of type PLS_INTEGER, a good type to use in conjunction with BITAND, can store powers of two up to 2^{30}, giving you 30 bits to work with.

CEIL(n)
: Returns the smallest integer greater than or equal to *n*. For a comparison of CEIL with several other numeric functions, see Table 9-7 and Figure 9-5.

COS(n)

Returns the cosine of the angle *n*, which must be expressed in radians. If your angle is specified in degrees, then you should convert it to radians as described in "Trigonometric Functions" on page 272.

COSH(n)

Returns the hyperbolic cosine of *n*. If *n* is a real number, and *i* is the imaginary square root of −1, then the relationship between COS and COSH can be expressed as follows: COS (*i* * *n*) = COSH (*n*).

EXP(n)

Returns the value *e* raised to the *n*th power, where *n* is the input argument. The number *e* (approximately equal to 2.71828) is the base of the system of natural logarithms.

FLOOR(n)

Returns the largest integer that is less than or equal to *n*. For a comparison of FLOOR with several other numeric functions, see Table 9-7 and Figure 9-5.

GREATEST(n1, n2, ... n3)

Returns the largest number among the list of input numbers; for example, GREATEST (1, 0, −1, 20) yields 20.

LEAST(n1, n2, ... n3)

Returns the lowest number among the list of input numbers; for example, LEAST (1, 0, −1, 20) yields −1.

LN(n)

Returns the natural logarithm of *n*. The argument *n* must be greater than or equal to 0. If you pass LN a negative argument, you will receive the following error:

```
ORA-01428: argument '-1' is out of range
```

LOG(b, n)

Returns the base *b* logarithm of *n*. The argument *n* must be greater than or equal to 0. The base *b* must be greater than 1. If you pass LOG an argument that violates either of these rules, you will receive the following error:

```
ORA-01428: argument '-1' is out of range
```

MOD(n, m)

Returns the remainder of *n* divided by *m*. The remainder is computed using a formula equivalent to *n*–(*m**FLOOR(*n*/*m*)) when *n* and *m* are both positive or both negative, and *n*–(*m**CEIL(*n*/*m*)) when the signs of *n* and *m* differ. For example, MOD(10, 2.8) yields 1.6. If *m* is zero, then *n* is returned unchanged.

You can use MOD to determine quickly if a number is odd or even:

```
FUNCTION is_odd (num_in IN NUMBER) RETURN BOOLEAN
IS
BEGIN
   RETURN MOD (num_in, 2) = 1;
END;

FUNCTION is_even (num_in IN NUMBER) RETURN BOOLEAN
IS
BEGIN
   RETURN MOD (num_in, 2) = 0;
END;
```

NANVL(n, m)
: Returns *m* if *n* is NaN (not a number); otherwise, returns *n*. The value returned will be in the type of the argument with the highest numeric precedence: BINARY_DOUBLE, BINARY_FLOAT, or NUMBER, in that order.

POWER(n, m)
: Raises *n* to the power *m*. If *n* is negative, then *m* must be an integer. The following example uses POWER to calculate the range of valid values for a PLS_INTEGER variable ($-2^{31}-1$ through $2^{31}-1$):

```
POWER (-2, 31) - 1 .. POWER (2, 31) - 1
```

The result is:

```
-2147483648 .. 2147483647
```

REMAINDER(n, m)
: Returns the "remainder" of *n* divided by *m*. The remainder is defined as n–(m*ROUND(n/m). For example, REMAINDER(10, 2.8) yields –1.2. Compare with MOD.

ROUND(n)
: Returns *n* rounded to the nearest integer. For example:

```
ROUND (153.46) --> 153
```

ROUND(n, m)
: Returns *n* rounded to *m* decimal places. The value of *m* can be less than zero. A negative value for *m* directs ROUND to round digits to the left of the decimal point rather than to the right. Here are some examples:

```
ROUND (153.46, 1) --> 153.5
ROUND (153, -1) --> 150
```

For a comparison of ROUND with several other numeric functions, see Figure 9-5 and Table 9-7i.

SIGN(n)
: Returns either –1, 0, or +1, depending on whether *n* is less than zero, equal to zero, or greater than zero, respectively.

SIN(n)

Returns the sine of the specified angle, which must be expressed in radians. If your angle is specified in degrees, then you should convert it to radians as described in "Trigonometric Functions" on page 272.

SINH(n)

Returns the hyperbolic sine of *n*. If *n* is a real number, and *i* is the imaginary square root of –1, then the relationship between SIN and SINH can be expressed as follows: SIN (*i* * *n*) = *i* * SINH (*n*).

SQRT(n)

Returns the square root of *n*, which must be greater than or equal to 0. If *n* is negative, you will receive an error like the following:

```
ORA-01428: argument '-1' is out of range
```

TAN(n)

Returns the tangent of the angle *n*, which must be expressed in radians. If your angle is specified in degrees, then you should convert it to radians as described in "Trigonometric Functions" on page 272.

TANH(n)

Returns the hyperbolic tangent of *n*. If *n* is a real number, and *i* is the imaginary square root of –1, then the relationship between TAN and TANH can be expressed as follows: TAN (*i* * *n*) = *i* * TANH (*n*).

TRUNC(n)

Truncates *n* to an integer. For example, TRUNC(10.51) yields the result 10.

TRUNC(n, m)

Truncates *n* to *m* decimal places. For example, TRUNC(10.789, 2) yields 10.78.

The value of *m* can be less than zero. A negative value for this argument directs TRUNC to truncate or zero out digits to the left of the decimal point rather than to the right. For example, TRUNC(1264, –2) yields 1200.

For a comparison of TRUNC with several other numeric functions, see Table 9-7 and Figure 9-5.