Object Types with Getters and Setters

by Michael McLaughlin

This article is for you when you know the basics about how you work Oracle's object types. It teaches you how to write effective getters, setters, comparators, and static methods. Please read my September 2014 "Object Types & Bodies Basic" article if you're not sure how to work with object types.

Getters access an object instance and return values from an instance variable. Along with getters, you have setters. Setters let you assign a new value to an instance variable. Formally, getters are accessor methods and setters are mutator methods. PL/SQL implements getters as flinkeunctions and setters as procedures. After all a PL/SQL procedure is like a function that returns a `void` data type in Java.

The "Object Types & Bodies Basic" article introduces a `people_obj` object type. This article extends the behavior of the `people_obj` type. *Extends* is a funny word because it can have different meanings in object-oriented programming. Here, extends means to add functionality.

The first things we'll add are getters and setters for all the attributes of the object instance. We need to add them to the object type and body because Oracle implements objects like it does packages. The object type defines the published functions and procedures. The object body implements the published functions and procedures.

Here's the new `people_obj` type with getters and setters:

```
SQL> CREATE OR REPLACE
  2    TYPE people_obj IS OBJECT
  3    ( people_id    NUMBER
  4    , first_name   VARCHAR2(20)
  5    , middle_name  VARCHAR2(20)
  6    , last_name    VARCHAR2(20)
  7    , CONSTRUCTOR FUNCTION people_obj RETURN SELF AS RESULT
  8    , CONSTRUCTOR FUNCTION people_obj
  9      ( first_name   VARCHAR2
 10      , middle_name  VARCHAR2 DEFAULT NULL
 11      , last_name    VARCHAR2 ) RETURN SELF AS RESULT
 12    , MEMBER FUNCTION get_people_id RETURN NUMBER
 13    , MEMBER FUNCTION get_first_name RETURN VARCHAR2
 14    , MEMBER FUNCTION get_middle_name RETURN VARCHAR2
 15    , MEMBER FUNCTION get_last_name RETURN VARCHAR2
 16    , MEMBER PROCEDURE set_first_name (pv_first_name VARCHAR2)
 17    , MEMBER PROCEDURE set_middle_name (pv_first_name VARCHAR2)
 18    , MEMBER PROCEDURE set_last_name (pv_first_name VARCHAR2))
 19  INSTANTIABLE NOT FINAL;
 20  /
```

The new getters and setters are on lines 12 through 18. The closing parenthesis for the list of attributes, functions, and procedures moves from line 11 to line 18. While there are four attributes in the `people_obj` type and four getters for those attributes, there are only three setters. The reason for the difference is simple. The `people_id` attribute is a unique identifier. You should never change the value of a unique identifier.

Next, lets implement the object body. I'm opting to show the complete object body because some readers may not check out the earlier article. Here's the `people_obj` body:

```
SQL> CREATE OR REPLACE
  2    TYPE BODY people_obj IS
  3
  4    /* Default constructor. */
  5    CONSTRUCTOR FUNCTION people_obj RETURN SELF AS RESULT IS
  6
  7    /* Set a counter variable using a sequence. */
  8    lv_people_obj_s  NUMBER := people_obj_s.NEXTVAL;
  9
 10    BEGIN
 11      /* Assign a sequence value to the instance. */
 12      self.people_id := lv_people_obj_s;
 13
 14      /* Return a constructed instance. */
 15      RETURN;
 16    END people_obj;
 17
 18    /* Override constructor. */
 19    CONSTRUCTOR FUNCTION people_obj
 20    ( first_name   VARCHAR2
 21    , middle_name  VARCHAR2 DEFAULT NULL
 22    , last_name    VARCHAR2 ) RETURN SELF AS RESULT IS
 23
 24      /* Create a empty default instance. */
 25      people  PEOPLE_OBJ := people_obj();
 26
 27    BEGIN
 28      /* Create the instance with the default constructor. */
 29      people.first_name := first_name;
 30      people.middle_name := middle_name;
 31      people.last_name := last_name;
 32
```

```
33      /* Assign a local instance this instance. */
34      self := people;
35
36      /* Return the current instance. */
37      RETURN;
38    END people_obj;
39
40    /* Get people ID attribute. */
41    MEMBER FUNCTION get_people_id RETURN NUMBER IS
42    BEGIN
43      RETURN self.people_id;
44    END get_people_id;
45
46    /* Get first name attribute. */
47    MEMBER FUNCTION get_first_name RETURN VARCHAR2 IS
48    BEGIN
49      RETURN self.first_name;
50    END get_first_name;
51
52    /* Get middle name attribute. */
53    MEMBER FUNCTION get_middle_name RETURN VARCHAR2 IS
54    BEGIN
55      RETURN self.middle_name;
56    END get_middle_name;
57
58    /* Get last name attribute. */
59    MEMBER FUNCTION get_last_name RETURN VARCHAR2 IS
60    BEGIN
61      RETURN self.last_name;
62    END get_last_name;
63
64    /* Set first name attribute. */
65    MEMBER PROCEDURE set_first_name
66    ( pv_first_name  VARCHAR2 ) IS
67    BEGIN
68      self.first_name := pv_first_name;
69    END set_first_name;
70
```

```
 71    /* Set middle name attribute. */
 72    MEMBER PROCEDURE set_middle_name
 73    ( pv_middle_name  VARCHAR2 ) IS
 74    BEGIN
 75      self.middle_name := pv_middle_name;
 76    END set_middle_name;
 77
 78    /* Set last name attribute. */
 79    MEMBER PROCEDURE set_last_name
 80    ( pv_last_name  VARCHAR2 ) IS
 81    BEGIN
 82      self.last_name := pv_last_name;
 83    END set_last_name;
 84  END;
 85  /
```

The `get_people_id` member function on lines 41-44 returns the unique identifier for the object instance. The `get_first_name` member function on lines 47-50 returns the `first_name` attribute. The `get_middle_name` member function on lines 53-56 returns the `middle_name` attribute. The `get_last_name` member function on lines 59-62 returns the `last_name` attribute. Each of these getters returns an instance attribute. The `self` reserved word identifies the current instance of the object type.

The `set_first_name` member procedure on lines 65-69 assigns a value to the `first_name` attribute. The `set_middle_name` procedure on lines 72-76 assigns a value to the `middle_name` attribute. The `set_last_name` member procedures on lines 79-83 assigns a value to the `last_name` attribute. The constructor functions create instances of the `people_obj` and return them to the calling scope. Each of these setters assigns a value to an instance attribute.

Comparative functions are limited to the `MAP` and `ORDER` member functions. The `MAP` function only works with the `CHAR`, `DATE`, `NUMBER`, or `VARCHAR2` data type. You could implement a `MAP` function against the `last_name` attribute but not the collection of the three variable length strings. You would implement an `ORDER` member function to compare the collection of strings.

You can define an equals `MAP` function in the `people_obj` object type like:

```
SQL> CREATE OR REPLACE
  2    TYPE people_obj IS OBJECT
  3    ( people_id    NUMBER
...
 19    , MAP MEMBER FUNCTION equals RETURN VARCHAR2)
 20   INSTANTIABLE NOT FINAL;
 21   /
```

After creating the people_obj object type, you can implement the following MAP function:

```
SQL> CREATE OR REPLACE
  2    TYPE BODY people_obj IS
...
 85    /* Implement an equals MAP function. */
 86    MAP MEMBER FUNCTION equals RETURN VARCHAR2 IS
 87    BEGIN
 88      RETURN self.last_name;
 89    END equals;
 90
 91  END;
 92  /
```

The MAP function is inadequate when you compare multiple attributes. You can implement an ORDER member function with the following syntax in the people_obj object type.

```
SQL> CREATE OR REPLACE
  2    TYPE people_obj IS OBJECT
  3    ( people_id    NUMBER
...
 19    , ORDER MEMBER FUNCTION equals
 20      (pv_people PEOPLE_OBJ) RETURN NUMBER)
 21   INSTANTIABLE NOT FINAL;
 22   /
```

The ORDER function is more complete than the MAP function. You can implement a last name, first name, and middle name ORDER function as follows:

```
SQL> CREATE OR REPLACE
  2    TYPE BODY people_obj IS
...
 85    /* Implement an equals MAP function. */
 86    ORDER MEMBER FUNCTION equals
 87    (pv_people PEOPLE_OBJ) RETURN NUMBER IS
 88    BEGIN
 89      IF NVL(self.last_name,'A') > NVL(pv_people.last_name,'A') THEN
 90        RETURN 1;
 91      ELSIF NVL(self.last_name,'A') = NVL(pv_people.last_name,'A') AND
 92          NVL(self.first_name,'A') > NVL(pv_people.first_name,'A') THEN
 93        RETURN 1;
```

```
 94      ELSIF NVL(self.last_name,'A') = NVL(pv_people.last_name,'A') AND
 95          NVL(self.first_name,'A') = NVL(pv_people.first_name,'A') AND
 96          NVL(self.middle_name,'A') > NVL(pv_people.middle_name,'A') THEN
 97        RETURN 1;
 98      ELSE
 99        RETURN 0;
100      END IF;
101    END equals;
102  END;
103  /
```

The equals ORDER function on lines 86 through 101 checks for a three conditions. First, it checks whether the instance's `last_name` is greater than the parameter object's `last_name`. Second, it checks whether the last names are equal and the instance's `first_name` is greater than the parameter object's `first_name`. Finally, it checks whether the last and first names are equal and the `middle_name` is greater than the parameter object's `middle_name` value.

Unfortunately, it's hard to test this comparison without adding a `to_string` function. The `to_string` function prints the formatted name. You can add the `to_string` function to the object type like so:

```
SQL> CREATE OR REPLACE
  2    TYPE people_obj IS OBJECT
  3    ( people_id    NUMBER
...
 19    , MAP MEMBER FUNCTION equals RETURN VARCHAR2
 21    , MEMBER FUNCTION to_string RETURN VARCHAR2)
 20   INSTANTIABLE NOT FINAL;
 21   /
```

Line 21 shows the declaration of the `to_string` function, and the following code snippet shows you the implementation of the `to_string` function:

```
SQL> CREATE OR REPLACE
  2    TYPE BODY people_obj IS
...
103    /* Create a to_string function. */
104    MEMBER FUNCTION to_string RETURN VARCHAR2 IS
105    BEGIN
106      RETURN self.last_name || ', ' || self.first_name || ' ' ||
107             self.middle_name;
108    END to_string;
109
```

```
110  END;
111  /
```

After assembling all the parts, we can test whether the ORDER comparative function works. The following anonymous block program declares a `people_list` collection that holds instances of the `people_obj` object type.

```
SQL> DECLARE
  2    /* Declare an object type. */
  3    TYPE people_list IS TABLE OF people_obj;
  4
  5    /* Declare three object types. */
  6    lv_obj1  PEOPLE_OBJ := people_obj('Fred',NULL,'Maher');
  7    lv_obj2  PEOPLE_OBJ := people_obj('John',NULL,'Fedele');
  8    lv_obj3  PEOPLE_OBJ := people_obj('James',NULL,'Fedele');
  9    lv_obj4  PEOPLE_OBJ := people_obj('James','Xavier','Fedele');
 10
 11    /* Declare a list of the object type. */
 12    lv_objs PEOPLE_LIST := people_list( lv_obj1, lv_obj2
 13                                      , lv_obj3, lv_obj4);
 14
 15    /* Swap A and B. */
 16    PROCEDURE swap
 17    ( a IN OUT PEOPLE_OBJ
 18    , b IN OUT PEOPLE_OBJ ) IS
 19      /* Declare a third variable. */
 20      c PEOPLE_OBJ;
 21    BEGIN
 22      /* Swap values. */
 23      c := b;
 24      b := a;
 25      a := c;
 26    END swap;
 27
 28  BEGIN
 29    /* Nested loop comparison. */
 30    FOR i IN 1..lv_objs.COUNT LOOP
 31      FOR j IN 1..lv_objs.COUNT LOOP
 32        IF lv_objs(i).equals(lv_objs(j)) = 0  THEN
 33          swap(lv_objs(i), lv_objs(j));
```

```
 34        END IF;
 35      END LOOP;
 36    END LOOP;
 37
 38    /* Print the reordered list. */
 39    FOR i IN 1..lv_objs.COUNT LOOP
 40      dbms_output.put_line(lv_objs(i).to_string());
 41    END LOOP;
 42  END;
 43  /
```

The `people_obj` instances on lines 6 through 9 are out of order in the starting collection. The local `swap` procedure reorders them on lines 30 through 36. You would see the following output from the preceding anonymous block:

```
Fedele, James
Fedele, James Xavier
Fedele, John
Maher, Fred
```

All of our work in this paper so far shows you how to work with implementing functions and procedures in instances of object types. PL/SQL object types support `MEMBER` functions and procedures to work with object instances. PL/SQL object types also support `STATIC` functions and procedures. You use `STATIC` functions and procedures when you want to write and call a module in an object type that works like a function or procedure in a package.

You can call a `STATIC` function or procedure without creating an instance of an object. Creating an instance of the object type is a key use of `STATIC` functions. This approach is very much like how Oracle implements temporary `BLOB` and `CLOB` columns.

Here's the snippet of additional code required in the `people_obj` object type:

```
SQL> CREATE OR REPLACE
  2    TYPE people_obj IS OBJECT
  3    ( people_id    NUMBER
...
 21    , MEMBER FUNCTION to_string RETURN VARCHAR2
 22    , STATIC FUNCTION get_people_obj
 23      ( pv_people_id NUMBER) RETURN people_obj)
 20   INSTANTIABLE NOT FINAL;
 21   /
```

The `get_people_obj` function is a `STATIC` function and it takes a single number to return a name. It accomplishes this by using a parameterized cursor. You would implement the `get_people_obj` function like so:

```
SQL> CREATE OR REPLACE
  2    TYPE BODY people_obj IS
...
109    /* Create a get_people_obj function. */
110    STATIC FUNCTION get_people_obj
111    ( pv_people_id NUMBER ) RETURN PEOPLE_OBJ IS
112
113      /* Implement a cursor. */
114      CURSOR get_people_obj
115      ( cv_people_id NUMBER ) IS
116      SELECT   first_name
117      ,        middle_name
118      ,        last_name
119      FROM     contact
120      WHERE    contact_id = cv_people_id;
121
122      /* Create a cursor variable. */
123      lv_contact get_people_obj%ROWTYPE;
124
125      /* Create a temporary instance of people_obj. */
126      lv_people_obj  PEOPLE_OBJ;
127    BEGIN
128      /* Open, fetch and close cursor. */
129      OPEN get_people_obj(pv_people_id);
130      FETCH get_people_obj INTO lv_contact;
131      lv_people_obj := people_obj( first_name => lv_contact.first_name
132                                 , middle_name => lv_contact.middle_name
133                                 , last_name => lv_contact.last_name);
134      CLOSE get_people_obj;
135      RETURN lv_people_obj;
136    END get_people_obj;
137
138  END;
139  /
```

The `get_people_obj` function takes a single numeric parameter. The numeric parameter passes the primary key value for the contact table. Then, the STATIC function returns an instance of the `people_obj` object type. It accomplishes that feat by using the numeric value as a lookup key in the contact table, as you can see in the `get_people_obj`

cursor on lines 114 through 120. The `STATIC` method opens, fetches a single row, and closes on lines 129 through 135.

Now you can call the `get_people_obj` function in a query and return an instance of `people_obj`. You can also use the `to_string` method to view the output, as follows:

```
SQL> SELECT   people_obj.get_people_obj(1003).to_string()
  2  FROM     dual;
```

It prints:

```
PEOPLE_OBJ.GET_PEOPLE_OBJ(1003).TO_STRING()
-------------------------------------------

Vizquel, Oscar
```

This article has shown you how to write effective getters, setters, comparators, and static methods. It also has shown how to test and work with Oracle object types and bodies.