



# ***Mejores Practicas***

## **PLSQL**





# *El proceso de desarrollo 1*

- Establecer estándares y guías de trabajo antes de escribir código
  - Escogencia de las herramientas de trabajo
  - Metodología para escribir código SQL en la aplicación
  - Arquitectura para manejo de errores:
  - Procedimiento para revisión y prueba al software





## *El proceso de desarrollo 2*

- Solicitar ayuda si se está en problemas por mas de 30 minutos
- Efectuar revisiones al código fuente escrito por otros
- Verificar el estricto cumplimiento de los estándares
- Generar código siempre que sea posible y/o conveniente
- Establecer procedimientos formales para pruebas al software
- Contar con personal externo para pruebas funcionales






# *Estilo De Codificación Y Convenciones*

- Código escrito con un formato consistente y legible, fácil de mantener
- Metodologías para nombrar objetos
  - Identificar el alcance de una variable dentro de nombre:  
g\_XX para variables globales  
l\_XX para variables locales.
  - Usar sufijos para todo tipo de variables (tipos de datos, registros, colecciones, etc).
  - Mayúsculas para las palabras reservadas.
  - Si la variable o la rutina tiene mas de una palabra, separarlas mediante “\_”.
  - No mezclar mayúsculas y minúsculas dentro del nombre
  - Agrupar los tipos de variables para facilidad de lectura: Types en un lugar, Records en otro, etc.






# *Estandarizar los encabezados de programas*

- Todo programa debe tener
  - Versión, Autor y Copyright
  - Ubicación: ¿Dónde está almacenado el programa?
  - Descripción: breve reseña de la funcionalidad del programa
  - Parámetros: Si la rutina posee parámetros, describir c/u de ellos
  - Modificaciones: Reseñar la fecha, autor del cambio y descripción del cambio implementado, ordenados descendientemente por fecha.





## *Finalizar cada sentencia END con el nombre del Módulo.*

- Toda sentencia END, debe estar seguida por el módulo o rutina que está cerrando; esto facilita la lectura de los programas, en especial cuando se tienen varios bloques anidados.

- Ejemplo

```
Procedure antifraude(linea in number, restriccion in  
    number, onoff in number)
```

```
Begin
```

```
---
```

```
End; --antifraude
```





# *Nomenclatura*

## *Objetos de base de datos*

### CHECK

Ch\_<validación>

- **Ejemplo**

Ch\_sexo

Ch\_valida\_fecha

Ch\_estado





# *Nomenclatura*

## *Objetos de base de datos*

### PRIMARY

PK\_<tabla>

Se usará el mismo nombre de la tabla sobre la cual se esta definiendo

- **Ejemplo**

Pk\_linea

Pk\_transacc





# *Nomenclatura*

## *Objetos de base de datos*

### FOREIGN

FK\_<detalle>\_<maestro>\_#

Se usará el mismo nombre de la tabla sobre la cual se esta definiendo

\_# Solo se usa cuando entre las dos tablas exista mas de una relación

- **Ejemplo**

FK\_saldo\_linecuen

FK\_linea\_tipoact

FK\_linea\_cliente\_1 --Dueño

FK\_linea\_cliente\_2 --Inquilino





# *Nomenclatura*

## *Objetos de base de datos*

### INDEX

IN\_<tabla>\_<columnas>

Se usará el mismo nombre de la tabla donde se esté definiendo el índice; para las columnas se seguirá el mismo estándar de 8 caracteres hasta donde sea posible, dada la multiplicidad de columnas que un índice podría tener

- **Ejemplo**

IN\_linea\_tipoacti

IN\_tarifa\_tarifa





# *Nomenclatura*

## *Objetos de base de datos*

### TRIGGER

TR\_<tabla>\_<funcionalidad>

Se usará el mismo nombre de la tabla donde está definido el trigger; la funcionalidad será para la cual esa creado el mismo

- **Ejemplo**

Tr\_linea\_estadistica

Tr\_linea\_ultimodi





# *Nomenclatura*

## *Objetos de base de datos*

### VISTAS

VI\_<nombre>

Para el nombre se usará el mismo estándar de las tablas

- **Ejemplo**

Vi\_linecuen Vista de línea cuenta

Vi\_repofall Vista de reportes de falla





# *Nomenclatura*

## *Objetos de base de datos*

### SECUENCIAS

SE\_<nombre>

Para el nombre se usará el mismo estándar de las tablas

- **Ejemplo**

Se\_estufact      Secuencia de estudio factibilidad

Se\_numerepo      Secuencia de numero de reportes





# *Elementos de programación*

Objeto

Estándar

**PROCEDURE**

?

**FUNCTION**

?

**PACKAGE**

PK\_<nombre>

- Utilizar verbos para procedimientos
  - » Ins\_XXX: si la rutina inserta registros
  - » Get\_XXX: si la rutina obtiene algo
  - » Del\_XXX: si la rutina elimina registros
  - » Upd\_XXX: si la rutina elimina registros
  - » Chk\_XXX: si la rutina evalúa alguna condición
- Utilizar sustantivos para las funciones.



# *Indentación*

- Utilizar técnicas de indentación para claridad del código , enfatizando en la relación del código con el bloques al que pertenece.
- Utilizar siempre dos espacios, no utilizar tabulador para ello y mantener bajo el mismo nivel tanto las sentencias que inician, como las que terminan un bloque dado
- En lo posible utilizar formateadores de código





# *Incluir Comentarios en el Código*

## *Fuente*

- Toda rutina debe tener comentarios que faciliten el conocimiento de la funcionalidad
- Deben agregar valor al programa en donde se incluyen.
- Ejemplo:
  - Un comentario sin sentido podría ser este:  
--Si los meses de atraso son mayores que el parámetro DCX  
IF mes\_atraso > get\_param('DCX') then ...
  - Aplicando una buena técnica para comentarios, el mismo código sería:  
--Si se cumple la condición para desconectar ...  
IF mes\_atraso > get\_param('DCX') then ...







# *Nombres de los Archivos Fuente*

- Un archivo fuente por objeto (procedure, function, package).
- El nombre del archivo será el mismo nombre de la rutina que representa para facilidad de ubicación en el disco duro.
- Crear un árbol dentro de la carpeta del proyecto, que clasifique en subdirectorios los diferentes objetos





# *Variables y Estructuras de Datos*

- NUMBER: especificar su precisión, pues si no se hace Oracle asigna un espacio de 38 dígitos de precisión y por tanto se desperdicia memoria.
- CHAR y VARCHAR: no utilizarlos.
- VARCHAR2: definir su precisión para la sarta que desea manipularse.
- INTEGER: para manejo de valores enteros.
- PLS\_INTEGER: es el formato mas eficiente para manejo de enteros.





# *Masiva utilización de %TYPE y %ROWTYPE*

- Permite a la aplicación ser flexible en cuanto a cambios en el diseño de las tablas y hace posible que los desarrolladores separen la construcción de los programas, de la estructura de los datos a manipular





# *Declarar Constantes cuando sea necesario*

- Evita que errores involuntarios alteren el valor de una variable.

- Ejemplo

DECLARE

x NUMBER CONSTANT := 10;





# *No inicializar variables en su declaración*

DECLARE

F date := fecha\_proceso\_empresa;

Debería describirse usando:

DECLARE

F date;

BEGIN

F := fecha\_proceso\_empresa;

EXCEPTION

<Rutina de manejo de errores>

END;



# *Reemplazar expresiones complejas por booleanos y funciones*

- **Ejemplo:**

```
IF cod_est_linea in ('IN','DE','SU')  
  AND cod_srv in ('LT','RD','PB','RP')  
  AND seq_troncal = 0 THEN
```

Debería describirse así, creando la función “es\_linea\_servicio”:

```
IF es_linea_servicio (cod_Srv, cod_est_linea, seq_troncal) THEN
```

Facilita la lectura, oculta la complejidad de la evaluación y con seguridad que esta función, puede requerirse en otro componente del sistema.



## *Remover segmentos de código y variables no usadas*

- Limpiar variables declaradas y nunca usadas (Desperdicio de memoria).
- Eliminar código en comentario o que carece de funcionalidad (Fuente de preguntas o dudas en programadores).





## *Limpiar estructuras de datos al terminar el programa*

- Incluir rutinas de limpieza de variables, cursores y archivos manipulados.
- Debe ejecutarse si el programa termina bien o con error.
- Evita errores con cursores/archivos abiertos







# *Evitar las conversiones de datos implícitas*

Ejemplo:

```
DECLARE
```

```
X DATE := '01-abr-2004';
```

- El código no debe depender de características de la instalación, Regedit u otros parámetros.
- Usar las funciones de conversión: TO\_CHAR, TO\_NUMBER, TO\_DATE.



# *Reutilizar tipos de datos definidos por el usuario*

```
CREATE OR REPLACE PACKAGE vbles IS
```

```
    TYPE linea_r IS RECORD (  
        Linea linea.nro_linea%type,  
        Direccion linea.dir_instal%type );  
END vbles;
```

Cuando este tipo de datos se requiera en cualquier programa, su utilización se hará así:

```
DECLARE  
    Linea_reclamo vbles.linea_r;
```

Los TYPE pueden definirse también dentro de la Base de Datos

# *Utilize variables globales de manera cuidadosa (1)*

```
CREATE OR REPLACE FUNCTION
    incrementa_saldo (saldo NUMBER)
RETURN NUMBER IS
    l_tmp NUMBER(14,2);
BEGIN
    IF variables_globales.incrementa THEN
        l_tmp := saldo*1.16;
    Else
        l_tmp := saldo;
    END IF;

    RETURN l_tmp;
END;
```

# *Utilize variables globales de manera ciudadosa (2)*

CREATE OR REPLACE FUNCTION

    incrementa\_saldo (saldo NUMBER, incremento BOOLEAN)

    RETURN NUMBER IS

l\_tmp NUMBER(14,2);

BEGIN

IF incremento THEN

    l\_tmp := saldo\*1.16;

Else

    l\_tmp := saldo;


END IF;

RETURN l\_tmp;

END;



# *Control De Estructuras*

- Use ELSIF con cláusulas mutuamente exclusivas
  - Reemplace las sentencias IF con expresiones booleanas (IF es\_moroso THEN)
  - No declarar el índice de un ciclo FOR
  - Nunca abandonar un ciclo WHILE o LOOP mediante EXIT.
  - Mover expresiones constantes, fuera de los ciclos repetitivos
  - Utilizar bloques anónimos para asignar variables y recursos sólo cuando se requieren
- 



## 4. Manejo De Excepciones

- Definir un único procedimiento global para toda la aplicación, antes de escribirla
  - Procedimientos para manejar la mayoría de excepciones a nivel básico, como por ejemplo escribiendo en una bitácora de errores.
  - Una rutina para generar una parada mediante RAISE, evitando con ello la utilización de la sentencia RAISE\_APPLICATION\_ERROR.
  - Funciones para devolver el texto descriptivo de un error





# Sugerencias relativas a EXCEPTIONS

- *No generar excepciones para controlar el flujo del programa.*
- *No sobrecargar de errores una sentencia EXCEPTION: varias sentencias que pueden generar la misma excepción.*
- *Incluir siempre las excepciones que pueden ser anticipadas*
- *Ocultar el código de las excepciones bajo packages específicos.*
- *Utilizar con precaución la excepción WHEN OTHERS: siempre hay que visualizar SQLCODE y SQLERRM.*



# *Escritura del SQL*

- Siempre deben cumplirse las siguientes reglas:
  - Nunca debe repetirse una sentencia SQL en toda la aplicación.
  - Encapsular toda sentencia SQL dentro de packages (Llamados de primer nivel)
  - Escribir el código con la idea de que las estructuras cambiarán.
  - Siempre que se pueda, hacer uso de las mejoras introducidas por PLSQL para el llamado de sentencias.





# *Utilización de Variables*

- Especial atención a nombres de variables de PLSQL, cuando se usan dentro de sentencias SQL.
- Tienen preferencia, las columnas de las tablas.
  - Ejemplo:
  - DECLARE
  - Dpto NUMBER := 10;
  - BEGIN
    - » DELETE FROM emp
    - » WHERE dpto = dpto;



# Utilizar una única rutina para generar excepciones

- Todos los mensajes de error de la aplicación deben estar codificados en una tabla.
- No emplear códigos de error en los programas: definirlos en un package como CONSTANT

- Ejemplo

```
IF pak_ciclo.ciclo_procesado (p_ciclo) THEN  
    err.raise (err.ciclo_ya_procesado);  
END IF;
```





# Aislar el efecto de COMMIT/ROLLBACK mediante transacciones autónomas

- Usar *PRAGMA AUTONOMOUS\_TRANSACTION* antes del begin del procedimiento, para que se pueda dentro de él manejar el concepto de transacción, sin afectar la operación global que se efectúa.
- Util para rutinas de manejo de errores





## **Colocar toda consulta a una fila de una tabla, bajo una función**

- *Toda sentencia del tipo `SELECT .. INTO`, deberá estar definida dentro de una función que a su vez hará parte de un package.*
- *Dentro de estas rutinas se deberá hacer un manejo de todas las excepciones posibles incluyendo `NO_DATA_FOUND`, `TOO_MANY_ROWS` y `WHEN OTHERS`.*
- *Toda tabla deberá tener un package de primer nivel.*
- *Será desarrollada una interface para la generación automática de código de primer nivel para dar cumplimiento a esta recomendación.*





## Colocar toda consulta a una fila de una tabla, bajo una función

- Ejemplo:
- `X := pn_linea.direccion_instalacion(plinea);`
- `Y := pn_linea_cuenta.ciclo(plinea);`
- `Z := pn_linea.registro(plinea);`





## **Declarar dentro de packages, aquellos cursores que sean reutilizables**

- Todos aquellos cursores que puedan ser reutilizados en varios sitios del sistema, deben ser declarados en la especificación de un package, evitando con ello la duplicidad de código en su declaración.
- Nomenclatura a usar:  
<package>.<nombre cursor>





## **Recibir el resultado de un cursor mediante variables tipo RECORD**

- Mayor flexibilidad por cambios en las columnas
- Escasa declaración de variables.
- Adaptabilidad en cambios de la sentencia SELECT.





## Utilizar la función COUNT sólo para contar registros

- La función COUNT sólo debe usarse para responder a la pregunta ¿Cuántos registros cumplen cierta condición?
- No usarla para saber si existen registros o no!
- Para ello, utilizar cursores e indagar por el atributo % FOUND.





## No utilizar FOR .. LOOP para retornar una sola fila

- *El propósito primordial de todo ciclo repetitivo es, como su nombre lo indica, efectuar varias interacciones; toda sentencia que retorne una fila, será implementada utilizando SELECT .. INTO o cursores.*

- *Ejemplo:*

```
FOR i IN (SELECT nombre FROM empleado
          WHERE cedula = x) LOOP
    Y := i.nombre;
END LOOP;
```



## Parametrizar cursores explícitos

- Siempre que en la definición de un cursor, se haga referencia a variables, las mismas deberán ser pasadas al cursor mediante parámetros, facilitando con ello la lectura del mismo y permitiendo al programador conocer en detalle las relaciones que las variables juegan al interior de la sentencia SELECT.

- Ejemplo:

DECLARE

CURSOR morosos (pciclo in ciclo.cod\_ciclo%type) is

SELECT linea, saldo, atraso

FROM linea

WHERE cod\_ciclo = pciclo;




## Utilizar la cláusula RETURNING para recuperar información de filas modificadas.

- *Siempre que se requiera obtener alguna columna de un registro que acaba de ser insertado o modificado, se debe usar la cláusula RETURNING; con ello, se evita la inclusión de un SELECT adicional para obtener dicho valor.*

```
INSERT INTO solicitud VALUES (pk_solic, nro_solic, ...  
VALUES(seq_solic.nextval, nro_solic, ....);  
SELECT seq_solic.curr_val  
into solicitud_ingresada  
FROM dual;
```

```
INSERT INTO solicitud VALUES (pk_solic, nro_solic, ...  
VALUES(seq_solic.nextval, nro_solic, ....)  
RETURNING pk_solic INTO solicitud_ingresada;
```



# Utilizar las mejoras ofrecidas por PLSQL para manejo de consultas con múltiples registros retornados

- *Hacer uso de la sentencia BULK COLLECT dentro de la cláusula SELECT para traer en una sólo interacción todos los registros obtenidos de la consulta.*

- *Ejemplo:*

*SELECT codigo, nombre, sueldo*

*BULK COLLECT INTO v\_codigos, v\_nombres, v\_sueldos*

*FROM empleado;*





# Encapsular sentencias DML dentro de packages

- Nunca debe escribirse una sentencia DML dentro del código de la aplicación; por el contrario, estas sentencias deberán hacer parte de los paquetes de primer nivel que cada tabla de la aplicación deberá tener implementada.
- Esto permite que la aplicación se ejecute mas rápidamente dado que por ejemplo, todo INSERT en una tabla dada será siempre el mismo y una única versión del mismo estará presente en la SGA.





## Utilizar la sentencia **FORALL** para operaciones **DML** masivas

- Si se tienen datos almacenados en vectores o tablas PLSQL y se desea hacer con ellos un proceso de actualización de tablas, se recomienda usar la sentencia **FORALL** para optimizar su desempeño .
- La utilización del **FORALL**, minimiza la cantidad de cambios de contexto y optimiza dramáticamente estas operaciones **DML**.





## Si se utiliza SQL dinámico, siempre usar variables Bind

DECLARE

comando VARCHAR2(500);

periodo NUMBER(4);

pciclo NUMBER(2) := 2;

mi\_cursor ref cursor;

BEGIN

period := 200404;

comando := 'SELECT linea, saldo FROM lin' || periodo || ' WHERE cod\_ciclo  
= :x';

OPEN mi\_cursor FOR comando **USING** pciclo;





## 7. CONSTRUCCIÓN DE PROGRAMAS

- Encapsular las reglas de negocio, dentro de packages

- *Ejemplo:*

Reemplazar esto:

```
IF pcod_Srv in ('LT','PB','RM') AND  
  ptipo_sub_act not in ('O','M') AND  
  ptipo_Act != 'P' THEN  
  -- aplico subsidios y contribuciones
```

por esto:

```
IF aplica_subsidio_contribucion(plinea) THEN  
  -- aplico subsidios y contribuciones
```







## **Limitar el tamaño del código fuente a una página usando modularización.**

- Toda sección ejecutable de cualquier rutina, no deberá sobrepasar las 60 líneas de código y para ello se deberá hacer uso extensivo de las técnicas de modularización. Cuando el código fuente se estructura ordenadamente usando esta técnica, resulta un código mas fácil de leer y mantener.



# Utilizar en los llamados a procedimientos, los nombres de los parámetros

- En todo llamado a una rutina, función o procedimiento se procurará utilizar la notación de parámetros con su nombre para lograr un incremento en la comprensión en el código en especial cuando se tienen rutinas con gran cantidad de parámetros (PLSQL utiliza por default la notación posicional de parámetros).

Ejemplo:

```
BEGIN
```

```
    pak_rm.liquidar_rm(2004,4,10,'S',FALSE);
```

```
END;
```

por esto:

```
BEGIN
```

```
    pak_rm.liquidar_rm (ano_proceso => 2004,  
                        mes_proceso => 4,  
                        ciclo => 10,  
                        aplicar_estadisticas => 'S',  
                        cronometro_tiempos => FALSE);
```

```
END;
```



## Evitar programas que tengan efectos secundarios no enunciados

- *El nombre de la rutina debe dar idea de lo que sucede en su interior.*
- *Ejemplo:*

*`X := empleado_te.nombre_empleado (pcedula);`*





# Toda función debe contener una sola cláusula RETURN

- Cambiar esto:

```
IF saldo > 0 THEN  
    RETURN TRUE;  
ELSE  
    RETURN FALSE;  
END IF;
```

por esto:

```
IF saldo > 0 THEN  
    x := TRUE;  
ELSE  
    x := FALSE;  
END IF;  
  
RETURN x;
```





# Toda función debe tener solamente parámetros de entrada

- *No cambiar el estilo estándar de las funciones.*
- *Si se requiere devolver varios parámetros:*
  - *Retornar un registro*
  - *Retornar una colección*
  - *Convertir la función en un procedure*





## Nunca retornar NULL desde funciones booleanas

- Una función booleana, debería únicamente retornar valores TRUE/FALSE, dado que si se llegara a retornar NULL, el programador que use la rutina debe tener en cuenta esta posibilidad en todo lugar en donde haga uso de la función .
- *Ejemplo:*  
*Tablas de Verdad*





## **Limitar el tamaño del código fuente de los triggers**

- Dado que los triggers son muy utilizados para implementar sobre ellos las reglas del negocio, es conveniente que el texto fuente que los compone resida en packages dedicados a las reglas del negocio (que hemos llamado en este documento, de segundo nivel), manteniendo como código del trigger, simplemente el llamado a estas rutinas.





## **Consolidar triggers que se ejecuten bajo las mismas condiciones**

- Ninguna tabla deberá tener dos o mas triggers disparados bajo la misma condición o evento (before/after insert(delete/update);
- No hay garantía del orden en que la base de datos los ejecutará.







## Llenar columnas derivadas, mediante triggers

- Desnormalizaciones
- Optimización de Rendimiento
- Exigencias a nivel de programación
- Siempre debe garantizarse sincronismo





## 8. Construcción de Packages

- Agrupar tipos de datos y funcionalidad de un componente software o proceso en packages.
- Construir siempre en primera instancia las especificaciones de los packages y luego el cuerpo.
- Utilizar la persistencia de variables de un package, para montar caché de datos a nivel local .
  - *Ejemplo: thisuser.name*
  - *Búsqueda de ciudad en Otros Cargos*





# Utilizar la técnica de Overloading para el llamado de rutinas dentro de packages

```
CREATE OR REPLACE PACKAGE ul IS
```

```
FUNCTION get_ciclo(plinea IN NUMBER, ptipo IN VARCHAR2);
```

```
FUNCTION get_ciclo(pseq_linea IN NUMBER);
```

```
END;
```





# Generar y manejar independientemente la especificación y el cuerpo de los packages

- Sin importar la herramienta de programación que sea utilizada, siempre deben ser almacenados en archivos separados tanto la especificación, como el cuerpo del programa; con el paso del tiempo, un muy alto porcentaje de cambios ocurre sobre el cuerpo del paquete y no sobre la especificación.
- *Disminuye Objetos Inválidos*





## **QUE HACEMOS AHORA??**

- Definir una estrategia de Implementación de los Estándares
- Qué hacemos con el código ya escrito?
- Cómo vamos a escribir el nuevo código?





# Estrategia

- Publicar el documento completo en la Intranet
- Publicar también esta presentación
- Generar un mail diario con una mejor práctica
- Generar listado de cambios en el software por semana
- Revisar con la persona encargada el cumplimiento de estándares
- Llevar una bitácora por persona
- Hacer esto por un tiempo prudencial
- ***Hacerlo Oficial: “No se compila si no satisface los estándares establecidos”***

