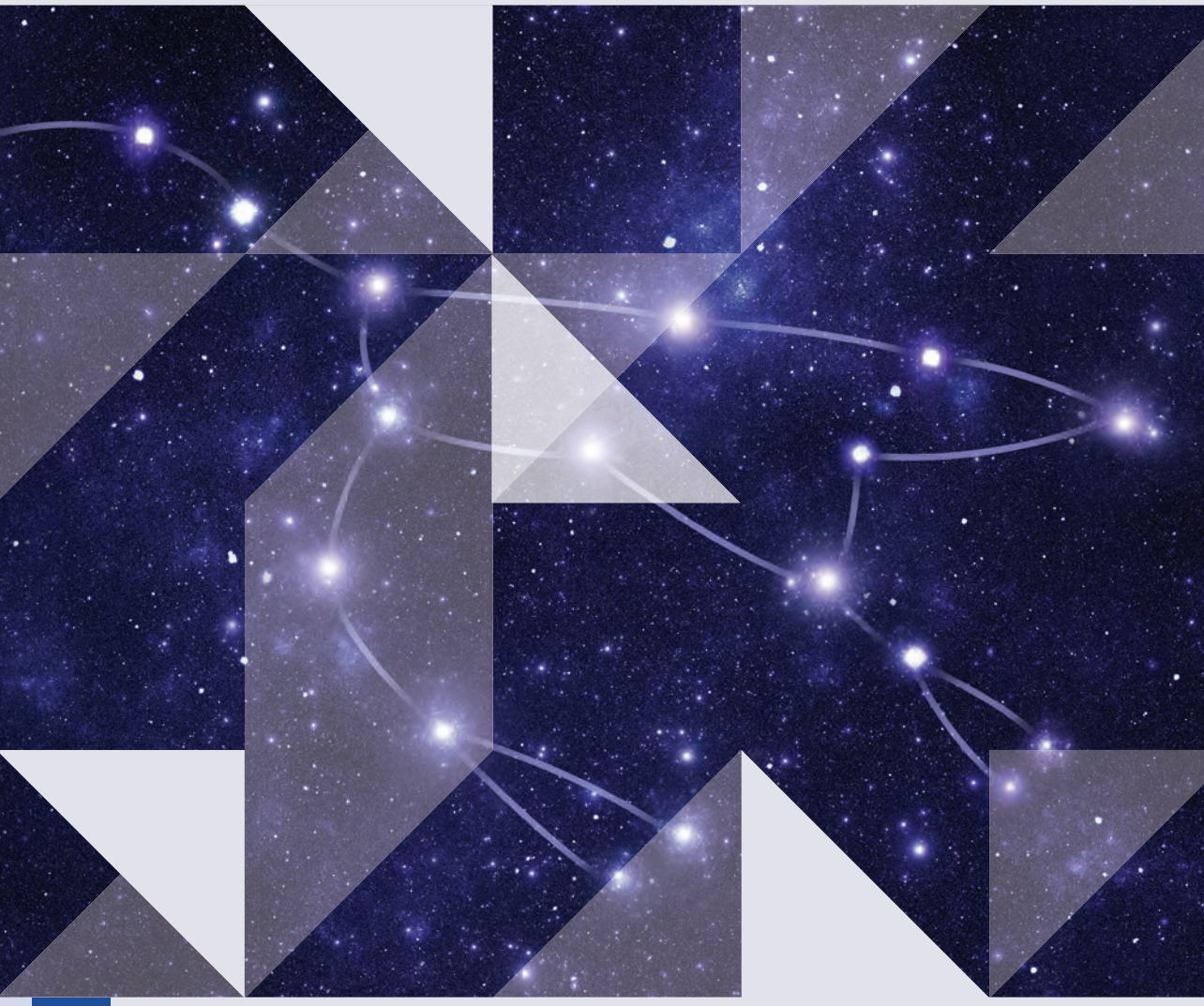
Leerboek Oracle PL/SQL

GILBERT RATTINK



ACADEMIC SERVICE

Derde druk

Leerboek Oracle PL/SQL

Derde druk

Gilbert Rattink

ACADEMIC SERVICE

Meer informatie over deze en andere uitgaven kunt u verkrijgen bij:
Sdu Klantenservice
Postbus 20014
2500 EA Den Haag
tel.: (070) 378 98 80
www.sdu.nl/service

1e druk, december 1998
2e druk, mei 2006
3e druk, januari 2012

Zetwerk: Redactiebureau Ron Heijer, Markelo
Omslagontwerp: Twin Media, Culemborg

ISBN 978 90 395 2661 3
NUR 123 / 995

Voorwoord

Dit boek vormt een uitgebreide kennismaking met de taal PL/SQL van Oracle. Deze taal combineert de kracht van de vierdegeneratietaal SQL met de procedurele elementen van een derdegeneratietaal. Met deze taal kunnen in een Oracle-database PL/SQL-objecten worden gemaakt, zoals procedures, functies, packages en database triggers. Bovendien wordt PL/SQL toegepast in Oracle-applicaties, zoals Oracle Application Express (APEX) en de klassieke Oracle-tools uit de Developer-familie (Forms, Reports en Graphics).

Het boek is in eerste instantie bedoeld voor het onderwijs op HBO- en academisch niveau. Het is bovendien geschikt voor zelfstudie of voor gebruik buiten het reguliere onderwijs. Het kan dienen als ondersteuning van een PL/SQL-cursus of voor het opfrissen van kennis indien er lange tijd niet meer met PL/SQL is gewerkt.

Als voorkennis wordt verondersteld dat de lezer de principes van een relationele database kent en een goede kennis heeft van Oracle SQL, zoals beschreven in bijvoorbeeld *Leerboek Oracle SQL (De Haan, 2004)*. We veronderstellen dat de lezer vertrouwd is met het programma SQL*Plus of met ontwikkeltools zoals SQL Developer of TOAD. Met deze interfaces kunnen op eenvoudige wijze SQL- en PL/SQL-programma's worden uitgevoerd. Hoewel kennis van een derdegeneratietaal het begrip van PL/SQL zal versnellen, is voorkennis van een dergelijke taal niet vereist.

In het boek is de theorie ruim voorzien van voorbeelden. In vrijwel alle hoofdstukken zijn paragrafen met opgaven opgenomen. De lezer kan aan de hand van deze opgaven testen of hij de theorie begrepen heeft. De uitwerkingen van de opgaven zijn samengevat in een bijlage.

Zoals de titel aangeeft, moet dit boek als leerboek worden beschouwd. Het probeert dan ook niet de mogelijkheden van de taal PL/SQL compleet te behandelen. Het boek zou dan in omvang flink toenemen en voor een cursus minder geschikt worden. Doel van het boek is de lezer in staat te stellen de meest relevante mogelijkheden te benutten die PL/SQL biedt in Oracle-applicaties en in de Oracle-database. De belangrijkste concepten en constructies komen in het boek aan de orde. Vervolgens is de lezer in staat zijn kennis aan de hand van de documentatie van Oracle te verdiepen. De volledige documentatie van SQL en PL/SQL is op het internet beschikbaar:
www.oracle.com/pls/db112/portal.all__books

Om de opgaven te kunnen maken en om met de voorbeelden uit het boek te kunnen experimenteren is het noodzakelijk dat de lezer beschikt over een computer waarop de volgende software is geïnstalleerd:
- Oracle11g Server
- SQL*Plus dan wel
- SQL Developer

Deze software kunt u eventueel downloaden van de officiële site van Oracle: www.oracle.com.

In de hoofdstukken worden achtereenvolgens de volgende onderwerpen besproken.

In hoofdstuk 1 wordt de noodzaak van het bestaan van een taal als PL/SQL toegelicht. Waarom voldoet de op zichzelf bijzonder krachtige taal SQL, die de basis vormt voor de communicatie met een relationele database, soms niet? En waarom zijn soms de in derdegeneratietalen gebruikelijke constructies als iteratie en voorwaardelijke uitvoering nodig?

In hoofdstuk 2 komt de basissyntaxis van PL/SQL aan de orde. De structuur van een PL/SQL-blok wordt behandeld en de manier waarop variabelen worden gedeclareerd, gevuld en uitgelezen. Vervolgens komt aan de orde welke operatoren en functies kunnen worden gebruikt voor het uitvoeren van berekeningen. Na de syntaxis van procedurele constructies wordt aangegeven hoe gegevens uit een database in een PL/SQL-programma worden overgeheveld en hoe vanuit PL/SQL-programma's tabellen kunnen worden bewerkt.

In hoofdstuk 3 wordt beschreven hoe in een PL/SQL-programma runtime-fouten oftewel exceptions worden onderschept. De voorgedefinieerde Oracle-exceptions worden behandeld, evenals de manier waarop eigen exceptions kunnen worden gedefinieerd.

Hoofdstuk 4 gaat in op het definiëren van en het werken met samengestelde datatypen. Daarbij wordt duidelijk gemaakt hoe recordstructuren en arraystructuren worden gedefinieerd en hoe variabelen op basis van die structuren worden gedeclareerd. Oracle hanteert overigens niet de term *array* maar *collection*. In verband met de uitgebreide mogelijkheden die collections in verschillende varianten bieden, gaat het hoofdstuk vooral over collections. In dat kader komt ook aan de orde hoe het mechanisme van bulk collect en bulk DML werkt. Het biedt ons onder andere de mogelijkheid een belangrijke eigenschap van SQL te omzeilen. SQL is gebaseerd op de verzamelingenleer. SQL-commando's werken op verzamelingen en elementen daarvan zijn in principe ongeordend. Rij-voor-rijverwerking is dus per definitie niet mogelijk met SQL. Via collections kan rij-voor-rijverwerking wel worden toegepast.

Hoofdstuk 5 beschrijft hoe procedures en functies worden geschreven. Het gaat daarbij om de syntaxis van procedures en functies zoals ze in een anoniem PL/SQL-blok kunnen worden gebruikt of in een applicaties zoals Oracle Developer, Oracle Application Express en andere.

In hoofdstuk 6 komen cursors aan de orde. Een cursor is een stukje geheugen dat voor de verwerking van SQL-commando's wordt gebruikt. Met een cursor wordt het onder andere mogelijk select-opdrachten centraal te definiëren, om er vervolgens op verschillende plekken in de programmatuur gebruik van te maken.

In hoofdstuk 7 wordt behandeld hoe procedures en functies in de Oracle-database als object kunnen worden opgeslagen. Bovendien worden de redenen toegelicht om tot het maken van dergelijke objecten over te gaan. Zo spelen performanceoverwegingen, onderhoudbaarheid van applicaties en beveiliging een belangrijke rol.

Hoofdstuk 8 beschrijft wat packages zijn en hoe en waarom die in een Oracle-database aangemaakt worden. Een package is een pakketje met verschillende procedures en functies die logisch gezien bij elkaar horen. Het gebruik van packages voegt nieuwe mogelijkheden toe boven op de mogelijkheden van procedures en functies die in hoofdstuk 7 behandeld zijn. Er is ook een korte toelichting op packages die met Oracle worden meegeleverd.

In hoofdstuk 9 komen tot slot de database triggers aan de orde. Dit zijn objecten met PL/SQL-code die in de database kunnen worden gemaakt. Een database trigger is een programma dat automatisch wordt uitgevoerd als een tabel of view wordt bewerkt. Met database triggers kunnen complexe business rules worden bewaakt en kan gedetailleerde log-informatie worden onderhouden.

Er zijn drie bijlagen toegevoegd. In bijlage A vindt u een overzicht van de tabellen die in de voorbeelden van het boek en in de opgaven worden gebruikt. Het zijn dezelfde tabellen die in *Leerboek Oracle SQL* (De Haan, 2004) worden gebruikt. Bijlage B bevat uitwerkingen van de opgaven uit de diverse hoofdstukken van het boek. In bijlage C gaan we in op een paar aspecten van PL/SQL die in lagere versies dan versie 11 net wat anders (of helemaal niet) werken.

Bijlage A treft u achter in het boek aan. Bijlagen B en C vindt u op de website www.academicservice.nl. Daar vindt u ook scripts met de uitwerkingen van de opgaven, eventueel aanvullende informatie en scripts waarmee u de tabellen kunt aanmaken die bij de voorbeelden en bij de opgaven worden gebruikt.

In de beschrijving van de syntaxis van PL/SQL hanteren we de volgende conventies:

`niet cursief`	Dit is tekst die letterlijk moet worden ingevoerd.
`cursief`	Dit is variabele tekst, die door de programmeur moet worden ingevuld.
( )	Ronde haken moeten letterlijk worden getypt.
[ ]	Rechte haken geven aan dat het omsloten gedeelte optioneel is.
{ | }	Accolades omsluiten twee of meer opties waaruit er een gekozen moet worden; de verticale streep scheidt de keuzemogelijkheden.

Tot slot wil ik de directeuren van Transfer Solutions, Gerard Hilte en Agnes Snellers, bedanken voor het feit dat er weinig privé-uren in het boek zijn gaan zitten. Ook wil ik een aantal collega's bedanken, ondanks wier opmerkingen dit boek uiteindelijk toch heeft kunnen verschijnen: Dorine Allan, Roger Derksen, Erik van Elderen, Marlies Morel en Eric Valk. Wim Grommen bedank ik vooral voor zijn redigeerwerk bij de tweede druk.

Bij de tweede druk
De opzet van deze tweede druk verschilt niet van die van de eerste druk. De grote verschillen zijn vooral inhoudelijk van aard. Met de komst van Oracle10g (en die van 9i daarvoor) is een groot aantal nieuwe mogelijkheden voor de PL/SQL-programmeur beschikbaar gekomen. Hoewel dit boek een leerboek is en geen reference guide wil zijn, ontkomen we er toch niet aan een groot deel van die nieuwe mogelijkheden te bespreken. Maar omdat het in de meeste gevallen tot veel programmeergemak leidt, hebben we daar geen enkele moeite mee. Nieuwe mogelijkheden die we in dit boek behandelen, zijn onder andere: errorfuncties, dynamisch SQL met execute immediate, nested tables, varrays, bulk collect, bulk DML, multiset-operatoren, save exceptions, autonome transacties, compilatiewaarschuwingen en instead of triggers.

De structuur van het boek is vrijwel niet aangepast ten opzichte van de eerste druk. De hoofdstukken over cursors en samengestelde datatypen zijn van plaats verwisseld. Dat heeft te maken met het feit dat het belang van collections flink is toegenomen en dat collections nu toepassingsmogelijkheden kennen die vroeger alleen met cursors konden worden gebouwd.

Bij de derde druk
De opzet van deze derde druk is niet gewijzigd ten opzichte van die van de tweede druk. De verschillen zijn inhoudelijk van aard. De inhoud is afgestemd op de mogelijkheden die PL/SQL vanaf versie 11g van Oracle biedt. Nieuw is bijlage C waarin enkele verschillen tussen Oracle 11g en lagere versies worden beschreven. Dat leek ons handig voor ontwikkelaars die te maken krijgen met een Oracle-versie die lager is dan 11g. Ook is het handig voor ontwikkelaars die wel met 11g werken, maar die te maken krijgen met uit lagere versies gemigreerde PL/SQL-code en mogelijk met constructies die in 11g beter door andere constructies vervangen kunnen worden.

Bij de derde druk is een bedankje op zijn plaats voor de nieuwe directie van Transfer Solutions. Harm Bodewes, Albert Leenders en Jurgen Duijster stelden mij wat kantooruren beschikbaar zodat ik deze derde druk van het Leerboek snel kon afronden.

Reacties op dit boek zijn van harte welkom. Zend deze naar de uitgever of via e-mail naar de auteur: leerboek__PLSQL@gimonet.nl.

Gilbert Rattink, september 2011

Inhoud

De bijlagen B en C zijn beschikbaar via de pagina bij dit boek op de website www.academicservice.nl.

Hoofdstuk 1

PL/SQL: introductie

Met PL/SQL (Procedural Language/Structured Query Language) wordt de taal aangeduid waarmee het mogelijk is de taal SQL, een niet-procedurele vierde-generatietaal (4GL), uit te breiden met elementen uit een procedurele derde-generatietaal (3GL). Hiermee kan een aantal tekortkomingen van de taal SQL worden opgeheven. PL/SQL-technologie kan zowel in de Oracle Server als in de tools (Oracle Application Express, Developer) worden gebruikt. Dit eerste hoofdstuk beschrijft waarom een taal als PL/SQL naast de taal SQL nodig is als er met een relationele database wordt gewerkt.

1.1 SQL: de mogelijkheden en de onmogelijkheden

1.1.1 Mogelijkheden

Als er met een Oracle-database, of met welke andere relationele database dan ook, wordt gewerkt, is de taal SQL bijzonder belangrijk. SQL, een afkorting voor Structured Query Language, is de enige taal waarmee gegevens in een relationele database kunnen worden bewerkt of uit een relationele database worden opgehaald. Bovendien is SQL nodig om een relationele database te maken en te beheren. Met andere woorden, zonder SQL kunnen we niet met een relationele database aan de slag. SQL wordt een *vierdegeneratietaal* (4GL) genoemd, omdat zij declaratief is. Dat houdt in dat we met SQL eerder definiëren wat we willen en niet zozeer hoe de computer tot een resultaat moet komen. Stel dat we een systeem voor personeelszaken hebben gebouwd. In tabellen wordt informatie bijgehouden over medewerkers en over de afdelingen waar zij werken. Het systeem is met Oracle gebouwd en is dus een relationele database. Willen we een overzicht hebben van alle namen van medewerkers en het nummer van de afdeling waar zij werken, dan ziet het bijbehorende SQL-programma er als volgt uit:

```
select med.naam
,      med.afd
from medewerkers med
```

In het voorbeeld wordt niet gespecificeerd hoe de tabel met medewerkerinformatie moet worden benaderd. Het geeft alleen aan wat we willen zien. Het programma ziet er heel eenvoudig uit en is gemakkelijk te lezen. Zelfs iemand die geen SQL maar wel Engels beheerst, kan een goed vermoeden hebben over wat in het programma wordt opgevraagd. Zelfs een wat ingewikkelder programma oogt nog steeds eenvoudig. De volgende SQL-opdracht berekent het gemiddelde salaris dat per afdeling wordt verdiend:

```
select med.afd
,      avg(med.maandsal)
from   medewerkers med
group  by med.afd
```

Ook hier wordt niet gespecificeerd hoe de tabel moet worden benaderd of hoe het eindresultaat moet worden berekend.

De taal SQL is gebaseerd op de theorie van de verzamelingenleer. SQL wordt dan ook wel *setgeoriënteerd* genoemd. Een select-opdracht levert een verzameling gegevens op. Ook voor het wijzigen van gegevens is SQL juist ontwikkeld om verzamelingen van rijen te bewerken. De volgende SQL-opdracht zorgt ervoor dat de verzameling van medewerkers uit afdeling 10 een salarisverhoging van 5 procent krijgt:

```
update medewerkers med
set    med.maandsal = med.maandsal * 1.05
where  med.afd = 10
```

Ook dit commando ziet er eenvoudig uit. Merk op dat in de where-regel alleen wordt aangegeven welke rijen uit de tabel in de te bewerken verzameling vallen. Deze verzameling wordt bewerkt. De volgorde waarin de medewerkers uit deze verzameling worden bewerkt, is irrelevant. Dit is volledig conform de theorie, aangezien het concept volgorde in de verzamelingenleer niet bestaat.

1.1.2 Onmogelijkheden

Ondanks de kracht van SQL zijn er toch wat tekortkomingen te constateren. In veel andere programmeertalen kunnen variabelen worden gebruikt waaraan waarden kunnen worden toegekend die op een later tijdstip kunnen worden uitgelezen of overschreven. In SQL kunnen geen variabelen worden gedeclareerd. Wel kent SQL de constructie van een sub-query, waardoor in enkele gevallen variabelen niet nodig zijn. De sub-query kan het gemis van variabelen in SQL echter niet altijd goedmaken. We zijn dan genoodzaakt tussenresultaten zelf op te schrijven of te onthouden.

Een tweede tekortkoming van SQL is dat de kracht van de verzamelingenleer soms ook kan tegenwerken. Stel dat we de medewerkers van ons bedrijf in een bepaalde volgorde een salarisverhoging willen geven; SQL laat ons dan in de steek. We hebben bijvoorbeeld een totaalbedrag van € 2.000 te besteden en elke medewerker ontvangt daar eenzelfde deel van. Nu moeten de medewerkers in volgorde van leeftijd worden bewerkt: de oudsten profiteren het eerst. Als het totaalbedrag is bereikt, kan het zijn dat enkele jongere medewerkers niet hebben kunnen profiteren van de salarisverhoging. Er is nu geen SQL-opdracht te bedenken waarmee deze bewerking kan worden doorgevoerd. Het volgordeprincipe is SQL immers vreemd.

Er is ook nog een derde tekortkoming van SQL. De taal is declaratief en niet procedureel. Dat omschreven we eerder als de kracht van SQL. In sommige situaties willen we echter toch een probleem procedureel oplossen. In SQL ontbreken procedurele constructies, die in andere programmeertalen gebruikelijk zijn. Een voorbeeld daarvan is de lusconstructie waarmee een reeks bewerkingen een aantal malen achtereen kan worden uitgevoerd. Als we alle medewerkers net zo lang vijf procent salarisverhoging willen geven tot het totaal uit te keren bedrag aan salarissen een grensbedrag overschrijdt, laat

SQL ons in de steek. SQL beschikt niet over een lusconstructie die een update-opdracht herhaaldelijk uitvoert totdat een bepaalde stopconditie wordt bereikt.

Kort samengevat is het met SQL niet mogelijk om:
- resultaten van SQL-commando's te gebruiken binnen een volgend commando;
- elementen uit een verzameling in een bepaalde volgorde te bewerken (rij-voor-rij-verwerking);
- procedurele problemen procedureel aan te pakken.

Het is duidelijk dat procedurele elementen uit een derdegeneratietaal (3GL) een welkome aanvulling zouden zijn op de kracht van een vierdegeneratietaal (4GL). Oracle levert dan ook zogeheten precompilers, zoals Pro*C en Pro*Pascal. Hiermee wordt het mogelijk SQL-commando's op te nemen in 3GL-programma's. Dat wordt *embedded SQL* genoemd. De complete kracht van een derdegeneratietaal wordt hierdoor gecombineerd met de complete kracht van een vierdegeneratietaal. Een nadeel van applicaties die met precompilers zijn gebouwd, is de platformafhankelijkheid. Een applicatie die op een UNIX-platform is ontwikkeld, draait niet direct op bijvoorbeeld Windows. Met andere woorden: de code van de applicatie is niet portabel. Daarom heeft Oracle zelf een platformonafhankelijke derdegeneratietaal ontwikkeld die portabel is en volledig geïntegreerd met de Oracle Server: PL/SQL.

1.2 PL/SQL

De letters PL in de afkorting PL/SQL staan voor Procedural Language. Met PL/SQL kan een procedurele schil om SQL worden aangelegd waarmee de tekortkomingen van SQL, die in de vorige paragraaf zijn genoemd, worden omzeild. Om PL/SQL te kunnen verwerken, is een zogeheten PL/SQL-motor (engine) nodig. Deze engine is in de Oracle Server ingebouwd, maar er is ook een engine aanwezig in de Oracle tools. Daardoor kunnen we de taal PL/SQL in de database gebruiken, maar ook in applicaties van de klassieke Oracle Developer-tools, zoals Forms en Reports. In dit boek komen deze Oracle-tools en het gebruik van PL/SQL in die tools echter niet aan de orde. De syntaxis zoals die wordt behandeld, is algemeen en dus ook van toepassing bij de tools. In dit boek maken we gebruik van de engine in de database. We gaan er van uit dat u daarbij gebruikmaakt van SQL*Plus, Oracle SQL Developer of enig andere ontwikkeltool waarmee u een Oracle-database kunt benaderen. In de laatste drie hoofdstukken lichten we speciale toepassingen van PL/SQL in de database toe.

Hier volgt een overzicht van de uitbreidingen die PL/SQL aan SQL toevoegt.

Variabelen
In PL/SQL kunnen variabelen en constanten worden gedeclareerd. Aan deze variabelen kunnen waarden worden toegekend die eventueel via een select-opdracht uit de database worden opgehaald. Naar deze variabelen kan vervolgens in andere SQL-commando's worden gerefereerd. Variabelen bestaan in verschillende soorten, variërend van

eenvoudig (bijvoorbeeld tekstvariabelen) tot complexe recordstructuren of variabelen om verzamelingen in op te slaan (collections).

Programmabesturing
Net als andere derdegeneratietalen bevat PL/SQL mogelijkheden om iteratie (een lusconstructie oftewel *loop*) en voorwaardelijke uitvoering (if-then-else-constructie) te programmeren. Ook is er een mogelijkheid om met goto-structuren te werken. Via een lus zou een van de problemen uit de vorige paragraaf kunnen worden opgelost. Daar moest een salarisverhoging net zolang worden doorgevoerd tot een maximum salaristotaal werd bereikt.

Cursors
Een cursor is een stuk geheugen dat door Oracle wordt gebruikt voor de verwerking van SQL-commando's. Met PL/SQL kunnen we voor SELECT-opdrachten zelf cursors declareren. Dat stelt ons in staat op een centrale plaats query's te definiëren en die op verschillende plaatsen in de programmatuur aan te roepen. Ook is een cursor een van de constructies waarmee we rij-voor-rijverwerking kunnen implementeren.

Attributen
PL/SQL is in hoge mate geïntegreerd met de Oracle-database. Zo kunnen we bijvoorbeeld bij de declaratie van variabelen via attributen gebruikmaken van datatypen of datastructuren die in de database zijn gedefinieerd.

Modulariteit
Modulair werken houdt in dat programma's worden opgedeeld in kleinere eenheden (modules) die geschikt zijn voor hergebruik. De onderhoudbaarheid van een applicatie wordt door een modulaire aanpak vergroot. In PL/SQL kunnen modulaire programmaonderdelen worden gemaakt in de vorm van procedures, functies, packages en cursors.

Afschermen van broncode
Met PL/SQL kunnen zogeheten *packages* worden geschreven. Door packages te gebruiken kunnen we bepaalde onderdelen van de broncode van procedures en functies voor gebruikers afschermen. Gebruikers kunnen zo bijvoorbeeld niet zien wat een procedure regel voor regel doet. Documentatie en tips voor het gebruik van een procedure kunnen daarbij wel zichtbaar worden gehouden.

Supplied packages
Oracle levert bij zijn database een groot aantal packages mee. Dat zijn pakketjes met voorgebakken programmatuur: bouwstenen die in een PL/SQL-programma's kunnen worden gebruikt. Daarmee kunnen PL/SQL-applicaties aanmerkelijk worden verrijkt. U bespaart tijd en moeite om zelf dergelijke functionaliteit te schrijven.

Foutafhandeling
Via zogeheten exception handlers is het mogelijk in een PL/SQL-programma runtimefouten te onderscheppen. Daardoor kunnen algemene foutmeldingen door specifiekere en gebruikersvriendelijke worden vervangen. Ook kunnen we er via exception handlers voor zorgen dat in foutsituaties eventueel alternatieve acties worden uitgevoerd.

Hoofdstuk 2

PL/SQL: basis

In dit hoofdstuk komen de basiseigenschappen van PL/SQL aan de orde. We gaan in op de blokstructuur van een PL/SQL-programma en beschrijven hoe variabelen worden gedeclareerd en van waarden worden voorzien. Verder komt de syntaxis van procedurele constructies zoals iteratie en voorwaardelijke uitvoering aan de orde. Ook wordt aandacht besteed aan het overbrengen van gegevens uit tabellen naar variabelen in een PL/SQL-programma en aan het sturen van datamanipulatiecommando's vanuit een PL/SQL-programma naar tabellen.

2.1 PL/SQL-blokstructuur

PL/SQL-programma's zijn opgebouwd uit blokken, die zelf ook weer uit blokken opgebouwd kunnen zijn. Dit wordt 'het nesten van blokken' genoemd. Waarom u blokken in andere blokken zou willen nesten, komt in een later hoofdstuk aan de orde. Voorlopig beperken we ons tot het gebruik van slechts een enkel blok en bekijken we de onderdelen waaruit een blok is opgebouwd.

Een PL/SQL-blok bestaat uit drie delen en wordt afgesloten met het sleutelwoord END:

declaraties	Dit deel is optioneel en wordt ingeleid door het woord DECLARE. In dit deel worden variabelen, constanten, procedures, functies, packages, cursors, exceptions en datatypen gedeclareerd.
uitvoerbare code	Dit deel is verplicht en wordt ingeleid met het woord BEGIN. Dit deel, ook wel de *body* genoemd, bevat SQL- en PL/SQL-commando's.
exception handler	Dit deel is optioneel en wordt ingeleid met het woord EXCEPTION. Het bevat code waarmee foutsituaties worden afgehandeld.

Hier volgt een voorbeeld van een eenvoudig PL/SQL-blok:

```
DECLARE
   v_kwadraat number;
BEGIN
   v_kwadraat := 3 * 3;
   dbms_output.put_line( v_kwadraat);
EXCEPTION
   when program_error
   then raise_application_error (-20000, 'Interne fout opgetreden');
END;
```

In dit programma wordt een variabele gedeclareerd waarin een getal kan worden opgeslagen. Vervolgens wordt aan die variabele het resultaat van de berekening 3*3 toegekend. En tot slot wordt de inhoud van de variabele naar het scherm weggeschreven. In het foutafhandelingsgedeelte wordt geregeld dat we, als er een interne fout optreedt, een melding krijgen met de tekst “Interne fout opgetreden”.

Een PL/SQL-blok moet op zijn minst één uitvoerbaar commando bevatten. Elk uitvoerbaar commando moet met een puntkomma (;) worden afgesloten. De puntkomma heeft dus in PL/SQL een andere betekenis dan de puntkomma die u in SQL*Plus achter een SQL-commando plaatst. In SQL*Plus sluit de puntkomma de opbouw van de SQL*Plus-buffer af. Vervolgens wordt het SQL-commando uitgevoerd. In PL/SQL markeert de puntkomma alleen maar het einde van een PL/SQL-commando. Ook het eind van het blok wordt met een puntkomma gemarkeerd. De laatste puntkomma zorgt er in PL/SQL niet voor dat een programma wordt uitgevoerd. Om vanuit SQL*Plus het PL/SQL-blok als geheel uit te voeren kan het SQL*Plus-commando RUN (of de slash) worden gebruikt. Als u met SQL Developer werkt, dan plaatst u de cursor ergens in het PL/SQL-blok en drukt u op de button of de functietoets F9 om het statement uit te voeren.

2.2 Variabelen declareren

In het declaratiedeel van een PL/SQL-blok kunnen we verschillende soorten zogeheten *identifiers* declareren: variabelen, constanten, procedures, functies, packages, cursors en exceptions. In deze paragraaf behandelen we de declaratie van variabelen en constanten. De overige identifiers komen in latere hoofdstukken aan de orde.

In het declaratiegedeelte van het PL/SQL-blok kunnen zowel variabelen als constanten worden gedeclareerd. Buiten het desbetreffende blok zijn de variabelen niet benaderbaar. Variabelen en constanten worden gebruikt om gegevens op te slaan, zodat die verderop in een programma weer gebruikt kunnen worden. De syntaxis voor declaratie van variabelen is als volgt:

```
variabelennaam [CONSTANT] datatype [NOT NULL] [ { := | DEFAULT } expressie ] ;
```

Expressies komen in het volgende hoofdstuk aan de orde.

Hier volgen enkele voorbeelden van de declaratie van variabelen:

```
DECLARE
   v_pi          constant number(8,7)          := 3.1415927;
   v_account               number(11) not null  := 5128876;

   v_bonus                 number               default 0;
   v_naam1                 varchar2(20)         default 'Joris';
   v_naam2                 v_naam1%type         default 'Corneel';
```

```
   v_maandsal             medewerkers.maandsal%type;
   v_gisteren             date                 default sysdate-1;
   v_max_aantal  constant pls_integer          := 7500;
BEGIN
 ...
END;
```

Bij de declaratie van een variabele moeten op zijn minst een naam en een datatype worden gespecificeerd. De naam, die op enkele gereserveerde woorden na vrij gekozen mag worden, mag niet langer zijn dan veertig tekens. Het is verstandig zinvolle namen te gebruiken en namen als var1 en var2 te vermijden. In de voorbeelden begint de naam van een variabele altijd met het prefix v__ . Dit prefix fungeert als afkorting voor *variabele*. We duiden met v__ aan dat de variabele een zogeheten scalaire (enkelvoudige) variabele is. Het is verstandig voor andere soorten variabelen (samengestelde datatypen) en voor cursors en exceptions (die ook gedeclareerd moeten worden en die verderop in het boek behandeld worden) andere prefixen te gebruiken. Deze manier van naamgeving berust overigens op conventie en is niet verplicht. Het bevordert wel in hoge mate de leesbaarheid en de onderhoudbaarheid van de programmatuur, omdat u aan de naam van een constructie kunt zien met welk soort constructie u te maken hebt.

2.2.1 Datatypen

Het datatype van een variabele kan een van de datatypen zijn die we ook bij de definitie van kolommen in de database kunnen gebruiken. Daarnaast is in PL/SQL een aantal extra datatypen beschikbaar. De zogeheten scalaire (enkelvoudige) datatypen zijn in vier categorieën onder te verdelen, afhankelijk van de soort waarden die erin moeten worden opgeslagen:

- logische waarden (booleans);
- datumwaarden;
- numerieke waarden;
- alfanumerieke waarden.

Naast enkelvoudige of scalaire datatypen bestaan er ook samengestelde of complexe datatypen. Deze komen in hoofdstuk 4 aan de orde. Het gaat dan om de volgende datatypen:

- record (in andere programmeertalen ook wel structure genoemd);
- collection (in andere programmeertalen is *array* de gebruikelijke term).

Op de mogelijkheid eigen scalaire datatypen te definiëren op basis van bestaande datatypen, zogeheten user defined subtypes, gaan we in dit boek niet nader in. Ook is het hier niet de bedoeling een compleet overzicht van alle datatypen te geven. We behandelen de belangrijkste en verwijzen voor een compleet overzicht naar de Oracle-documentatie.

Logische waarden

Logische waarden, ook wel booleans genoemd, worden in derdegeneratietalen gebruikt om de besturing van een programma te beïnvloeden, bijvoorbeeld bij loops of if-then-

constructies. We kunnen een boolean zien als het resultaat van een vergelijking of een bewering. In tweewaardige logica, waar de meeste derdegeneratietalen op gebaseerd zijn, kan een boolean twee waarden hebben:
- TRUE
- FALSE

Een bewering is in de tweewaardige logica waar (true) of niet waar (false). In een relationele omgeving wordt met driewaardige logica gewerkt. Hierin kan een bewering waar of niet waar zijn, maar is het ook mogelijk dat het niet bekend is of de bewering waar of niet waar is. In deze derde situatie heeft de boolean de waarde UNK (unknown). Meestal wordt deze waarde gerepresenteerd door NULL.

Om de waarde UNK toe te lichten kunnen we de bewering: 'Ernie is op 15 september 1965 geboren' wat nader bekijken. Als we niet weten op welke dag Ernie geboren is, kunnen we niet zeggen dat de bewering waar is, maar kunnen we ook niet zeggen dat de bewering niet waar is. Kortom, het is onbekend of de bewering (on)waar is. Booleans kunnen in een Oracle-omgeving daarom de volgende waarden aannemen:
- TRUE
- FALSE
- NULL

Hierbij staat NULL voor de waarde UNK. Een programmeur moet goed op de hoogte zijn van de gevolgen als een boolean tot UNK evalueert. Enkele consequenties van de driewaardige logica komen in een volgende paragraaf aan de orde.

Datumwaarden
Voor datums is het datatype date beschikbaar. Net zoals het datatype date in de database bevat een date-variabele in een PL/SQL-programma behalve informatie over de datum ook informatie over de tijd.

Numerieke waarden
We bespreken een aantal datatypen voor numerieke waarden. Het eerstgenoemde datatype (number) is ook in de database beschikbaar, waar het voor kolomdefinities wordt gebruikt, het tweede (pls__integer) is alleen in PL/SQL beschikbaar.

number [*(precisie, schaal)*]
Dit type wordt gebruikt voor getallen van willekeurige grootte en heeft een variabele lengte. Zowel precisie als schaal kunnen worden gespecificeerd. Precisie staat voor het totaal aantal cijfers, terwijl de schaal aangeeft op hoeveel posities achter (of bij negatieve schaal voor) de decimale punt wordt afgerond. Het maximum voor precisie is 38 cijfers en beslaat het bereik van 10^{-129} tot 9.99^{125}. De schaal kan variëren van -84 tot 127. De standaardwaarde voor schaal is 0. Er is een aantal synoniemen beschikbaar voor het datatype number. Deze synoniemen zijn: dec, decimal, double precision, int, integer, numeric, real en smallint.

We zouden vanwege de compatibiliteit met andere databases dan Oracle-databases of vanwege de leesbaarheid voor een van deze synoniemen kunnen kiezen.

Voorbeelden:

```
v_x number(3,2);    3.346 wordt opgeslagen als 3.35
v_y number(4,-3);   3456 wordt opgeslagen als 3000
v_z number(5);      1234.667 wordt opgeslagen als 1235
```

Als we geen precisie en schaal definiëren, passen er veertig cijfers in een variabele.

pls_integer
Met het eerder beschreven datatype number kunnen numerieke waarden nauwkeurig en efficiënt worden opgeslagen. Dit komt doordat number-waarden in decimaal formaat worden opgeslagen. Daardoor zijn rekenkundige bewerkingen op die waarden echter niet direct mogelijk. Als er met numbers gerekend wordt, moet Oracle dit intern eerst naar een binaire waarde converteren. Als we in een PL/SQL-programma sneller willen rekenen met gehele getallen, kunnen we beter het datatype pls_integer gebruiken. Dit datatype kan gehele getallen bevatten die liggen tussen -2147483648 en +2147483647. Van dit datatype zijn overigens twee subtypen beschikbaar: *natural* en *positive*. Het datatype natural kan de natuurlijke getallen (van 0 tot 2147483647) bevatten. Het datatype positive kan positieve getallen bevatten (van 1 tot 2147483647). Er bestaat overigens een datatype dat heel erg op pls_integer lijkt: binary_integer. Het is echter verstandig binary_integer niet te gebruiken, en wel uit performanceoverwegingen en omdat het gedrag van binary_integer verschillend kan zijn over verschillende platforms en Oracle-versies heen.

Alfanumerieke waarden
Voor alfanumerieke waarden zijn de volgende datatypen beschikbaar.

varchar2 (*lengte* [BYTE | CHAR])
Dit datatype wordt gebruikt om alfanumerieke gegevens met variabele lengte op te slaan. Het maximum voor *lengte* is 32767 bytes. Het is overigens verplicht een maximale lengte te specificeren, waarbij u kunt kiezen om dat in termen van bytes of characters (tekens) te doen. Ook als u voor characters kiest, wordt intern overigens de maximale lengte in bytes uitgedrukt.

Pas op
Een databasekolom van het datatype varchar2 kan slechts 4000 bytes breed zijn. Een PL/SQL-variabele van datatype varchar2 kan dus meer gegevens bevatten dan er in een varchar2-kolom past. Houd daar rekening mee als u de inhoud van een dergelijke variabele naar een databasekolom wegschrijft.

char [(*lengte* [BYTE | CHAR])]
Dit datatype wordt gebruikt voor het opslaan van alfanumerieke gegevens met een vaste lengte. Indien noodzakelijk wordt de inhoud van de variabele aan de rechterkant met spaties aangevuld. Als er geen lengte wordt opgegeven, komt dit overeen met char(1); het maximum is 32767 bytes.

Pas op
Vergelijkt u in een programma de inhoud van een varchar2-variabele met de inhoud van een char-variabele, houd dan rekening met het feit dat de tekst in de char-variabele aan de rechterkant aangevuld is met spaties, die u misschien niet ziet, maar die er wel zijn. Het is om problemen voor te zijn dan ook aan te raden om voor tekstvariabelen alleen het datatype varchar2 te gebruiken en nooit char.

Overige waarden
We noemen thans nog een aantal datatypen die niet in de vorige categorieën vallen.

BLOB *en* CLOB
Deze afkortingen staan voor Binary en Character Large OBjects. Variabelen van dit datatype kunnen zeer grote hoeveelheden data bevatten. BLOB's zijn er voor binaire data en CLOB's voor tekstdata. De maximale inhoud van een LOB-variabele is 8 tot 128 terabytes, afhankelijk van de databaseblokgrootte van uw database.

Het kan overigens zijn dat u in oudere programmatuur nog de datatypes LONG en LONG RAW tegenkomt, die oorspronkelijk bedoeld waren voor tekstvariabelen met veel gegevens, respectievelijk variabelen met veel binaire data. Oracle raadt aan om die datatypes niet meer te gebruiken en ze eventueel door BLOB's of CLOB's te vervangen.

raw(*lengte*)
Dit datatype wordt gebruikt om raw data in op te slaan die niet door PL/SQL moeten worden geïnterpreteerd. Een variabele van het datatype raw heeft een maximumlengte van 32.767 bytes, terwijl de maximumbreedte van een raw-kolom in de database 2000 bytes is.

rowid
Dit datatype wordt gebruikt om de waarde van de pseudo-column rowid van tabellen in op te slaan.

Het attribuut %TYPE
PL/SQL-variabelen worden vaak gebruikt om waarden uit de database op te slaan. Als een variabele bijvoorbeeld het maandsalaris van de directeur gaat bevatten, moet het datatype van die variabele gelijk zijn aan het datatype van de kolom maandsal in de tabel medewerkers: number(6,2). Als in de loop van de tijd het datatype van de kolom maandsal echter veranderd moet worden, omdat de salarissen bijzonder hoog worden of omdat het bedrijf naar een exotisch land verhuist en de salarissen in al even exotische valuta

moeten worden uitgedrukt, zouden ook alle variabelen in alle PL/SQL-programma's waarin het maandsalaris voorkomt, moeten worden aangepast. Dit probleem treedt ook op als het datatype van een numerieke kolom in de database wordt aangepast omdat er ook alfanumerieke tekens in de kolom opgenomen gaan worden. Dit onderhoudsprobleem kan worden voorkomen als bij de declaratie van variabelen voor het datatype wordt verwezen naar het datatype van een kolom in de database. Maak dan gebruik van het suffix %TYPE.

Met %TYPE kan bij de declaratie van variabelen worden verwezen:
- naar het datatype van een eerder gedeclareerde variabele;
- of naar het datatype van een kolom in de database.

```
v_maandsal     medewerkers.maandsal%type default 0;
v_oud_maandsal v_maandsal%type;
v_afdnaam      afdelingen.naam%type;
v_sofi_nr      admin.taxes.sofi_nr%type;
```

De variabele v_maandsal neemt het datatype over van de kolom maandsal in de tabel medewerkers (van de huidige gebruiker). De variabele v_oud_maandsal neemt het datatype over van de daarvoor gedeclareerde variabele v_maandsal. Het is ook mogelijk te verwijzen naar tabellen van andere gebruikers dan de huidige. De variabele v_sofi_nr neemt bijvoorbeeld het datatype over van de kolom sofi_nr in de tabel taxes in het databaseschema van admin.

Tip
Declareer het datatype voor variabelen die hun waarden direct uit een databasekolom krijgen altijd met het attribuut %TYPE.

Pas op
Als u gebruikmaakt van %TYPE, worden alleen het datatype en de precisie overgenomen. Eventuele defaultwaarden van de databasekolom worden niet overgenomen.
Als u bij de declaratie van een variabele doorverwijst naar het datatype van variabelen die not null zijn, of die als constante gedeclareerd zijn, dan worden die restricties niet overgenomen. In het vorige voorbeeld is de initiële waarde van `v_oud_maandsal` dus niet gedefinieerd.

2.2.2 Variabelen met een defaultwaarde

Variabelen worden in het declaratiedeel gedeclareerd en kunnen vervolgens in de body van het programma worden gebruikt. Op het moment van declareren krijgen variabelen de waarde NULL. In de body kan die waarde direct worden overschreven door er een nieuwe waarde aan toe te kennen. Maar het is mogelijk direct tijdens de declaratie al een waarde aan een variabele toe te kennen. Vooral omdat de initiële waarde NULL is en deze waarde soms aanleiding kan zijn tot programmeerfouten, is het in de regel een goede

programmeertechniek de initiële waarde bij de declaratie al op iets anders te zetten dan NULL. Daartoe kan het sleutelwoord DEFAULT worden gebruikt.

Voorbeeld:

```
DECLARE
   v_teller1 number;
   v_teller2 number default 0;
BEGIN
   v_teller1 := v_teller1 + 1;
   v_teller2 := v_teller2 + 1;
END;
```

In dit voorbeeld heeft v__teller1 aan het begin van de body de waarde NULL en heeft v__teller2 de initiële waarde 0. Als verderop in het programma beide tellers met 1 worden verhoogd, zal v__teller2 de waarde 1 hebben gekregen, terwijl v__teller1 nog steeds NULL is. Als u 1 bij 0 optelt, is het resultaat immers 1, maar iets onbekends plus 1 blijft even onbekend.

Overigens kan in plaats van het sleutelwoord DEFAULT ook het algemene toekenningsteken := worden gebruikt. Dit teken kunnen we zowel in het declaratiedeel als in de body en de exception handler van een PL/SQL-programma toepassen om aan variabelen een waarde toe te kennen. DEFAULT kan alleen in het declaratiedeel worden toegepast.

Pas op
Als u bij de declaratie van een variabele met NOT NULL wilt forceren dat de variabele nooit de null-waarde kan aannemen of als u met CONSTANT wilt forceren dat de waarde van de variabele in het programma nooit veranderd wordt, bent u verplicht een defaultwaarde toe te kennen.

2.3 Operatoren

Programma's zijn in de regel bedoeld om bewerkingen uit te voeren. Zo worden er berekeningen gedaan, worden er variabelen met elkaar vergeleken, worden woorden achter elkaar geplakt, enzovoort. Om dergelijke bewerkingen te kunnen uitvoeren, zijn operatoren en functies nodig. De operatoren en functies die we in PL/SQL gebruiken, zijn vrijwel dezelfde operatoren en functies die in SQL kunnen worden gebruikt. Voor een uitvoerige behandeling verwijzen we dan ook naar *Leerboek Oracle SQL* en de officiële Oracle-documentatie. Volledigheidshalve vatten we hier het een en ander samen.

Operatoren worden gebruikt om expressies te bouwen. Een expressie is een uitdrukking waarin variabelen, constanten en/of operatoren op een bepaalde manier met elkaar worden gecombineerd. De eenvoudigste expressie is overigens een constante. Een operator is een bewerking; datgene waarop de bewerking wordt uitgevoerd, heet de operand.

Voorbeelden van expressies:

```
3 + 4
v_plaats = 'UTRECHT'
v_comm IS NULL
v_voorl || ' ' || v_naam
12 * v_maandsal > 20000 and v_comm >= 100
999
```

In het eerste voorbeeld zijn 3 en 4 de operanden en is + de operator. In het tweede voorbeeld zijn v_plaats en 'UTRECHT' de operanden en is = de operator. Het derde voorbeeld is bijzonder, omdat de expressie bestaat uit één operand en één operator: v_comm is de operand en de operator is IS NULL. Het vierde en vijfde voorbeeld zijn wat complexer, omdat de operanden in principe weer expressies zijn die uit operatoren en operanden zijn samengesteld. Het laatste voorbeeld is juist eenvoudig, omdat de expressie uit enkel een constante bestaat. Operatoren kunnen in verschillende categorieën worden onderverdeeld.

2.3.1 Rekenkundige operatoren

In PL/SQL kunnen de volgende rekenkundige operatoren worden gebruikt:

Operator	Betekenis	Voorbeeld
**	machtsverheffen	2**3
+, -	teken (positief of negatief)	+2, -3
*, /	vermenigvuldigen, delen	2*3, 12/5
+, -	optellen, aftrekken	3+8, 3-8

De operator voor machtsverheffen (**) is overigens niet in SQL maar wel in PL/SQL beschikbaar.

We herinneren u eraan dat in SQL en PL/SQL niet alleen met getallen kan worden gerekend. In SQL en in PL/SQL kan ook bij een datum een getal worden opgeteld en kunnen twee datums van elkaar worden afgetrokken. Zo levert sysdate + 7 de datum op die een week na vandaag ligt. En als u van sysdate uw geboortedatum aftrekt, ziet u hoeveel dagen u al op de wereld bent.

2.3.2 Alfanumerieke operatoren

Eigenlijk is de meervoudsvorm operatoren voor alfanumerieke operanden niet van toepassing. Er is slecht één alfanumerieke operator: het concatenatieteken.

Operator	Betekenis	Voorbeeld
||	concateneren (plakken)	'a' || 'b' null || 'a'

Pas op
Bij een expressie geldt in de regel dat deze tot NULL evalueert als een van de operanden de NULL-waarde heeft. Zo is NULL plus 1 net zo NULL als NULL plus 1000. De concatenatieoperator is hierop een uitzondering: NULL || 'aap' evalueert tot 'aap'en niet tot NULL.

2.3.3 Vergelijkingsoperatoren

Operatoren die een expressie vergelijken met een andere expressie worden vergelijkingsopera toren genoemd. Een vergelijking is altijd een bewering die tot TRUE, FALSE of UNK evalueert. De vergelijkingsoperatoren zijn dezelfde die ook in SQL beschikbaar zijn. Deze kunnen worden onderverdeeld in de volgende categorieën.

- Relationele operatoren

Operator	Betekenis
=	is gelijk aan
<>, !=, ^=	is niet gelijk aan
<	is kleiner dan
>	is groter dan
<=	is kleiner dan of gelijk aan
>=	is groter dan of gelijk aan

- IS NULL
  Deze operator is bijzonder, omdat hiermee niet twee expressies met elkaar worden vergeleken. De operator werkt namelijk maar op één operand. Het resultaat is TRUE als de operand NULL is en FALSE als de operand niet NULL is.

Pas op
Gebruik om te testen of een expressie NULL is nooit het isgelijkteken (=). De test `v_variable = null` levert nooit TRUE op, ook niet als `v_variable` de waarde NULL heeft. Zelfs `null = null` levert niet TRUE op. Dat klinkt wellicht merkwaardig, maar is wel correct. Immers, u weet nooit of iets dat onbekend (UNK, NULL) is, gelijk is aan iets anders dat onbekend is. Met andere woorden: met A IS NULL test u of A onbekend is, terwijl u met A = NULL test of A gelijk is aan iets dat niet bekend is.

- LIKE
  Met deze operator kan worden getest of een alfanumerieke reeks aan een patroon voldoet. In het patroon kunnen zogeheten wildcards worden gebruikt:
  – De underscore (het liggende streepje: _) representeert precies één willekeurig teken (bijvoorbeeld v_naam like 'JANS_EN').

  - Het procentteken (%) representeert nul of meer willekeurige tekens (bijvoorbeeld v_naam like 'S%').
- BETWEEN
  Met deze operator kunt u nagaan of de waarde van een variabele binnen een bepaald bereik ligt. Bijvoorbeeld: `50 between 45 and 100` levert TRUE op. Ook `45 between 45 and 100` heeft TRUE als resultaat. Merk dus op dat de grenswaarden bij het bereik meegerekend worden. Ook bij alfanumerieke en datumvariabelen kan deze operator worden gebruikt.
- IN
  Met deze operator kunt u nagaan of een waarde in een verzameling voorkomt. Zo levert `3 in (1, 2, 3, 4, 5)` TRUE op, terwijl `3 in (2, 5, 6)` FALSE als resultaat heeft.

Pas op

Pas op met het gebruik van de operator NOT als in de verzameling de NULL-waarde voorkomt. Zo levert `3 not in (4, 5, 6, NULL)` UNK op en niet TRUE, zoals u misschien zou denken. Het is immers niet met zekerheid te zeggen dat 3 niet in de gegeven verzameling voorkomt als van een van de elementen uit de verzameling onbekend (UNK, NULL) is.

2.3.4 Logische operatoren

Er zijn drie operatoren waarmee logische expressies, oftewel boolean expressies, gebouwd kunnen worden. Deze operatoren werken op operanden die TRUE, FALSE of UNK/NULL opleveren, en het resultaat is een expressie die zelf ook weer TRUE, FALSE of UNK/NULL oplevert. Deze operatoren zijn:

Operator	Betekenis
AND	logische 'en'
OR	logische 'of' (inclusief)
NOT	logische ontkenning

Zoals al eerder werd opgemerkt, hebben we in een relationele omgeving met driewaardige logica te maken. Dit houdt in dat het resultaat van een bewering onbekend (UNK) kan zijn. Het is voor een programmeur van belang dat hij of zij weet wat het resultaat is van een logische expressie als een operand in de expressie UNK oplevert. Met behulp van zogeheten waarheidstabellen kunnen logische expressies tot TRUE, FALSE of NULL worden gereduceerd. De NULL-waarde staat voor onbekend (UNK) en betekent: FALSE noch TRUE.

In de volgende tabel leest u dat de logische expressie NOT TRUE evalueert tot FALSE, dat NOT FALSE evalueert tot TRUE en dat NOT UNK evalueert tot UNK.

Waarheidstabel voor NOT:

NOT	TRUE	FALSE	UNK
	FALSE	TRUE	UNK

De volgende tabel beschrijft de operator AND. In de linkerkolom staan de mogelijke waarden van logische expressie A, in de bovenste rij de waarden van logische expressie B en in de corresponderende cellen staat de waarde van de expressie A AND B.

Waarheidstabel voor AND:

AND	TRUE	FALSE	UNK
TRUE	TRUE	FALSE	UNK
FALSE	FALSE	FALSE	FALSE
UNK	UNK	FALSE	UNK

De volgende tabel leest u op dezelfde manier als de tabel voor de operator AND.

Waarheidstabel voor OR:

OR	TRUE	FALSE	UNK
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNK
UNK	TRUE	UNK	UNK

2.3.5 Prioriteitsregels voor operatoren

In complexe exp ressies gelden de volgende prioriteitsregels, dus als het ware de uitgebreide regel van Meneer Van Dale Wacht Op Antwoord.

Prioriteit	Operator	Betekenis
1	**	machtsverheffen
2	+,-	teken
3	*,/	vermenigvuldigen, delen
4	+, -, ||	optellen, aftrekken, plakken
5	=, !=, <>, ^=, <=, >=, IS NULL, LIKE, BETWEEN, IN	vergelijkingen
6	NOT	logische ontkenning
7	AND	logische 'en'
8	OR	logische 'of'

2.3.6 Case-expressie

Met operatoren zijn we in staat om een expressie te maken. We noemen voor de volledigheid in dit kader ook nog de case-expressie die in de taal SQL beschikbaar is. We kunnen deze expressie ook in PL/SQL gebruiken. Zo kunnen we de inhoud van een variabele vullen, afhankelijk van een bepaalde voorwaarde. In onderstaand voorbeeld wordt afhankelijk van de inhoud van de variabele v_functie een waarde toegekend aan de variabele v_functiecode.

```
v_functiecode    := case v_functie
                       when 'DIRECTEUR' then 1
                       when 'MANAGER'   then 2
                       else                  3
                    end;
```

We hebben hier de case-expressie beschreven, maar we komen case in dit hoofdstuk nog een keer tegen in de paragraaf over voorwaardelijke uitvoering. We hebben daar niet te maken met de case-expressie, maar met het case-statement.

2.4 Functies

In SQL kunnen bewerkingen worden uitgevoerd met behulp van functies. Deze functies kunnen in een aantal categorieën worden onderverdeeld:

- numerieke functies;
- alfanumerieke functies;
- groepsfuncties;
- datumfuncties;
- conversiefuncties;
- overige functies.

Een beschrijving van de functies vindt u in *Leerboek Oracle SQL* en in de Oracle-documentatie. Vrijwel al deze functies zijn ook in PL/SQL beschikbaar. We volstaan hier met het geven van een aantal voorbeelden:

```
v_lengte      := length( v_naam );
v_hoogste_sal := greatest( v_maandsal1 , v_maandsal2 );
v_jaarsal     := ( 12 * v_maandsal ) + nvl( v_comm, 0 );
v_rest        := mod( v_getal , 2 );
v_naam        := upper( v_naam );
v_datum_char  := to_char( sysdate , 'dd-mm-yyyy' );
v_datum       := to_date( '15021961' , 'ddmmyyyy' );
```

Er zijn enkele functies die niet direct in PL/SQL kunnen worden gebruikt. Het gaat daarbij om:

- alle groepsfuncties;
- de functie DECODE.

De volgende constructies worden dus *niet* geaccepteerd:

```
v_maximum      := max( v_maandsal );
v_som          := sum( v_comm );
v_functiecode  := decode( v_functie
                        , 'DIRECTEUR', 1
                        , 'MANAGER' , 2
                        , 3);
```

Het spreekt voor zich dat groepsfuncties op deze manier niet kunnen worden gebruikt. Wat zou bijvoorbeeld het resultaat van het eerste voorbeeld moeten zijn? Groepsfuncties werken per definitie op verzamelingen van gegevens en niet op één expressie.

Ook de DECODE-functie mag niet worden gebruikt bij het toekennen van waarden aan een variabele. Het is een krachtige functie die al lang in Oracle bestaat en veel wordt gebruikt in SQL-statements. Het is niet echt een gemis dat deze functie niet op deze manier in PL/SQL kan worden gebruikt. De DECODE-functie maakt het in feite mogelijk in SQL een soort IF-THEN-constructie te bouwen. Maar daarvoor kunnen we in SQL (en dus ook in PL/SQL) de case-expressie gebruiken. Dit hebben we in paragraaf 2.3.6 beschreven.

Tip
In PL/SQL-programma's kunnen complete select-, insert-, update- en delete-commando's worden opgenomen. In deze commando's kan de complete syntaxis van SQL worden toegepast. Daar kunnen dan ook alle functies worden gebruikt, dus ook de DECODE-functie en de groepsfuncties.

Pas op
Net als in SQL wordt in PL/SQL impliciete datatypeconversie toegepast. En net als in SQL is het raadzaam niet op de impliciete conversie te vertrouwen, maar expliciet met behulp van de conversiefuncties te converteren. Impliciete datatypeconversie kan van negatieve invloed zijn op de performance en kan bij nieuwe softwarereleases of bij toepassing op een andere databaseversie of met andere sessie-instellingen anders uitwerken.

2.5 Debug-meldingen

Tijdens het ontwikkelen van PL/SQL-programmatuur is het soms nodig debug-meldingen naar het scherm te schrijven, zodat de werking van het programma getest kan worden. In ontwikkelomgevingen als SQL*Plus en SQL Developer kunnen we dat regelen met behulp van de procedure PUT__LINE. Bij het installeren van Oracle wordt een aantal packages meegeleverd met daarin waardevolle procedures en functies die in PL/SQL kunnen worden gebruikt. Aan packages is een apart hoofdstuk gewijd. Voor

dit moment is het voldoende om te weten dat een package een pakketje met logisch bij elkaar horende PL/SQL-code is. Een van die packages heet DBMS__OUTPUT en bevat een aantal procedures en functies om gegevens naar het scherm te schrijven, waaronder de procedure PUT__LINE. De syntaxis van deze procedure is als volgt:

```
DBMS_OUTPUT.PUT_LINE( expressie ) ;
```

Als we een procedure uit een package aanroepen, moeten we de naam van het package vooraf laten gaan aan de naam van de procedure. Package-naam en procedurenaam worden van elkaar gescheiden door een punt.

Overigens verschijnen deze debug-meldingen alleen maar in de ontwikkelomgeving indien de volgende setting is ingesteld:

```
SET SERVEROUTPUT ON
```

Voor een uitgebreide bespreking van het package DBMS__OUTPUT verwijzen we naar de Oracle-documentatie.

Zo ontvangt de user SCOTT met het volgende programma een vriendelijke groet:

```
SQL> BEGIN
2 dbms_output.put_line( 'Hallo '|| user || ', een prettige ' || to_char( sysdate, 'day' ));
3 END;
4 /
Hallo SCOTT, een prettige sunday

PL/SQL procedure successfully completed.
```

Deze debug-meldingen worden pas naar het scherm geschreven als het PL/SQL-programma is afgelopen. Tot dat moment worden de meldingen in een interne buffer geplaatst. De buffergrootte kunt u in SQL*Plus bepalen. Hiervoor dient bij het aanzetten van de serveroutput ook een 'size' te worden opgegeven.

```
set serveroutput on size [ n | UNLIMITED ]
```

Merk op dat aan de procedure PUT__LINE slechts één argument kan worden meegegeven. Als u de waarde van meer dan één variabele op het scherm wilt zien, moet elke variabele apart met dbms__output.put__line naar het scherm worden geschreven. Als alternatief kan ook één aanroep van dbms__output.put__line volstaan, als de verschillende variabelen maar aan elkaar geplakt worden, waardoor er in feite weer sprake is van één werkelijk argument.

2.6 Programmabesturing

In derdegeneratietalen bestaan constructies waarmee een programma bestuurd kan worden. Soms moeten bepaalde commando's alleen onder bepaalde voorwaarden worden uitgevoerd. Daarvoor kennen de meeste programmeertalen een zogeheten *if-then*-constructie en een *case*-constructie. In andere gevallen moeten bepaalde commando's een aantal malen achter elkaar worden uitgevoerd. Daarvoor bestaat in de verschillende programmeertalen een zogeheten *loop*. Een andere constructie is *goto*. In het algemeen wordt het gebruik van deze laatste constructie afgeraden, vanwege de onoverzichtelijke, en daarmee slecht onderhoudbare code die dan kan ontstaan. Ook in PL/SQL kunnen we deze drie constructies toepassen. We beschrijven ze in deze paragraaf.

2.6.1 Voorwaardelijke uitvoering

Net als in derdegeneratietalen bestaat in PL/SQL de mogelijkheid bepaalde code onder voorwaarden uit te voeren. We hebben daarvoor twee constructies tot onze beschikking: de zogeheten if-then-else-constructie en de case-constructie.

2.6.1.1 If then else

De syntaxis voor de if-then-else-constructie is als volgt:

```
IF logische_expressie1
THEN
   commandoreeks1;
[
ELSIF logische_expressie2
THEN
   commandoreeks2;
]
[ELSE
  commandoreeks3;
  ]
END IF;
```

Als *logische__expressie1* TRUE is, wordt *commandoreeks1* uitgevoerd. Als *logische__expressie1* niet TRUE oplevert en er een ELSIF-tak is, wordt *logische__expressie2* geëvalueerd. Levert dat TRUE op, dan wordt *commandoreeks2* uitgevoerd. Er kunnen eventueel meer ELSIF-constructies worden opgenomen. Als een logische expressie niet TRUE oplevert, wordt de bijbehorende ELSIF of anders de ELSE-tak uitgevoerd.

Een paar voorbeelden:

```
if v_opslag = 'JA'
then
   v_sal := v_sal + 100;
end if;
```

Of, met een else-tak:

```
if v_extra_opslag = 'JA'
then
   v_sal := v_sal + 100 +50;
else
   v_sal := v_sal + 100;
end if;
```

Of, met een elsif-tak:

```
if v_extra_opslag = 'JA'
then
   v_sal := v_sal + 100 +50;
elsif v_sal < 2000
then
   v_sal := v_sal + 100;
else
   v_sal := v_sal + 80;
end if;
```

Tip

In plaats van een ELSIF-THEN-ELSE-constructie te gebruiken, kunnen ook complete IF-THEN-ELSE-constructies worden genest. De ELSIF-variant geniet echter de voorkeur, omdat die minder code vergt, gemakkelijker leesbaar en beter onderhoudbaar is. Vergelijk bijvoorbeeld de volgende twee constructies:

```
v_leeftijd := months_between( sysdate , v_gbdatum )/12;
if v_leeftijd > 60
then
   v_categorie := 'oudere';
elsif v_leeftijd > 35
then
   v_categorie := 'oudere jongere';
else
   v_categorie := 'jongere';
end if;
```

Deze constructie is overzichtelijker dan de volgende:

```
v_leeftijd := months_between( sysdate , v_gbdatum )/12;
if v_leeftijd > 60
then
   v_categorie := 'oudere';
else
   if v_leeftijd > 35
```

```
            then
               v_categorie := 'oudere jongere';
            else
               v_categorie := 'jongere';
            end if;
         end if;
```

2.6.1.2 Case

We hebben in dit hoofdstuk de case-expressie al behandeld. Daarmee konden we de voorwaardelijke toekenning van waarden aan een variabele regelen. Er bestaat ook een case-statement (case-constructie) waarmee we voorwaardelijke uitvoering van meerdere regels PL/SQL kunnen regelen. De toepassing van deze case-constructie is te vergelijken met die van de if-then-constructie. Het voorbeeld uit de vorige paragraaf zouden we ook als volgt kunnen programmeren:

```
v_leeftijd := months_between( sysdate , v_gbdatum )/12;
case
  when  v_leeftijd > 60
  then
    v_categorie := 'oudere';
  when v_leeftijd > 35
  then
   v_categorie := 'oudere jongere';
  else
   v_categorie := 'jongere';
end case;
```

Er zijn twee varianten van de case-constructie beschikbaar: de search case en de simple case. Hieronder volgt de syntaxis, om te beginnen de search case:

```
CASE
  WHEN logische_expressie1
  THEN
     commandoreeks1;
  [
  WHEN logische_expressie2
  THEN
     commandoreeks2;
  ]
  [ELSE
    commandoreeks3;
    ]
END CASE;
```

Merk op dat de case-constructie niet wordt afgesloten met END (zoals de case-expressie uit paragraaf 2.3.6), maar met END CASE (vergelijkbaar met END IF).

De logische expressies van de case-constructie worden van boven naar beneden geëvalueerd. Zodra de expressie van een when-regel true is, wordt de bijbehorende then-tak uitgevoerd, waarna de case-constructie wordt verlaten. De case-constructie heeft dus veel weg van de if-then-constructie, maar er is een groot verschil. Als er geen else-tak is opgenomen, en geen van de logische expressies uit de case evalueert tot true, dan stopt het programma met een foutmelding: er moet altijd een match in de case-constructie worden gevonden. De if-constructie zou in deze situatie geen fout hebben opgeleverd en het programma zou gewoon verder lopen.

Naast de search case bestaat er ook een simple case. Hieronder volgt de syntaxis:

```
CASE variabele
  WHEN expressie1
  THEN
     commandoreeks1;
  [
  WHEN expressie2
  THEN
     commandoreeks2;
  ]
  [ELSE
    commandoreeks3;
    ]
END CASE;
```

Achter het sleutelwoord CASE staat een variabele. De inhoud van deze variabele wordt vergeleken met de when-regels. Als er een match is, dan wordt de bijbehorende then-tak uitgevoerd en de constructie afgesloten. Ook hier geldt dat er ergens een match moet zijn, omdat het programma anders met een fout wordt afgebroken.

Een voorbeeld:

```
case v_extra_opslag
  when 'JA'
  then
     v_sal := v_sal + 100 +50;
  else
     v_sal := v_sal + 100;
end case;
```

2.6.2 Iteratie

Met behulp van loop-constructies kan iteratie, het herhalen van bepaalde code, worden geregeld zonder dat code dubbel moet worden geprogrammeerd. PL/SQL kent drie soorten loops:

- de eenvoudige loop;
- de while-loop;
- de for-loop.

Eenvoudige loop

De eenvoudigste en meest algemene loop heeft de volgende syntaxis:

```
LOOP
   commandoreeks;
   [EXIT [WHEN logische_expressie] ;]
END LOOP;
```

In principe is dit een oneindige loop, waarbij alles tussen LOOP en END LOOP wordt herhaald. De EXIT-regel is syntactisch gezien optioneel (er verschijnt geen foutmelding als deze regel in het programma ontbreekt), maar in de praktijk zult u de loop altijd aan de hand van een stopconditie met EXIT moeten verlaten. Doet u dat niet, dan hebt u een oneindige loop geschreven. Dat heeft als vervelend effect dat uw programma niet meer vanzelf stopt en gaat vastlopen.

Als *logische_expressie* TRUE wordt, wordt de code die binnen de loop achter het EXIT-commando staat, niet meer uitgevoerd en vervolgt het programma zijn weg bij het eerste commando achter END LOOP.

Het EXIT-commando kan binnen een loop ook in de then-tak van een if-then-else-constructie worden opgenomen. Maar EXIT WHEN *logische_expressie* is een elegantere en beter leesbare constructie, die de voorkeur geniet.

Het is vrijdag en iemand wil voor de planning van zijn ADV-dagen graag een overzicht hebben van de datums van de eerste tien vrijdagen die na de systeemdatum liggen. Daarvoor schrijft hij het volgende programma:

```
DECLARE
   v_teller number default 0;
   v_sysdate date default sysdate;
BEGIN
   LOOP
      v_teller := v_teller + 1 ;
      dbms_output.put_line( v_sysdate + ( v_teller * 7) );
      exit when v_teller >= 10;
    END LOOP;
END;
```

De teller wordt binnen de loop telkens met 1 opgehoogd. Vervolgens wordt bij de systeemdatum telkens een aantal weken opgeteld, waarbij het aantal afhankelijk is van de teller. De datum die dan ontstaat, wordt afgedrukt. Als de loop tienmaal is doorlopen, wordt hij verlaten.

While-loop

Een tweede soort loop is de while-loop. De belangrijkste eigenschap hiervan is dat aan het begin van de loop de voorwaarde (een logische expressie) wordt geëvalueerd waaronder de loop nog een keer doorlopen moet worden. Dit maakt de while-loop overzichtelijk en gemakkelijk leesbaar.

De syntaxis is:

```
WHILE logische_expressie
LOOP
   commandoreeks;
END LOOP;
```

Het voorbeeld waarin de eerstvolgende tien ADV-dagen worden afgedrukt, ziet er met deze loop als volgt uit:

```
DECLARE
   v_teller number default 0;
   v_sysdate date default sysdate;
BEGIN
   WHILE v_teller < 10 LOOP
      v_teller := v_teller + 1 ;
      dbms_output.put_line( v_sysdate + ( v_teller * 7) );
   END LOOP;
END;
```

Tip

Sommige derdegeneratietalen kennen ook een until-loop. Daar wordt code herhaald totdat een logische expressie een keer TRUE wordt. Hoewel PL/SQL dit soort loops niet kent, is het vrij eenvoudig een vergelijkbare constructie te bouwen:

```
LOOP
   ...
   EXIT WHEN logische_expressie;
END LOOP;
```

For-loop

Deze derde soort loop is vooral handig als van tevoren bekend is dat de loop een vast aantal malen herhaald moet worden. De eenvoudige loop en de while-loop kunnen daar

ook voor worden gebruikt, maar dan moet de programmeur zelf tellers declareren en bijwerken om de loop op het juiste moment te kunnen verlaten. De for-loop is speciaal voor dit soort problemen bestemd en de toepassing ervan bevordert de leesbaarheid en onderhoudbaarheid van het programma.

De syntaxis is:

```
FOR teller IN [REVERSE] ondergrens..bovengrens
LOOP
   commandoreeks ;
END LOOP;
```

De variabele *teller* (ook wel de loop-index genoemd) is een variabele die impliciet door het gebruik van de loop wordt gedeclareerd en die bovendien alleen binnen de loop gelezen kan worden. Buiten de loop bestaat die variabele niet meer. De onder- en bovengrens worden bij het binnengaan van de loop geëvalueerd. Dat wil zeggen dat het aanpassen van de grenzen binnen de loop geen effect heeft op het aantal keren dat de loop doorlopen wordt.

Elke keer als de loop doorlopen is, wordt de loop-index automatisch met 1 opgehoogd (of in de reverse-variant met 1 verlaagd).

Hier volgt nog eens het voorbeeld van de vrijdagen, gebouwd met behulp van de for-loop:

```
DECLARE
   v_sysdate date default sysdate;
BEGIN
   FOR i_teller in 1 .. 10 LOOP
      dbms_output.put_line( v_sysdate + ( i_teller * 7 ) );
   END LOOP;
END;
```

Merk op dat we voor de naam van de indexvariabele een ander prefix hebben gebruikt dan voor 'gewone' variabelen: een i__ in plaats van een v__. Dit is gedaan om aan de naam van die variabele te kunnen herkennen dat deze zich anders gedraagt dan een variabele die gewoon in het declaratiegedeelte is gedeclareerd. De variabele die we het prefix i__ hebben gegeven, heeft immers een beperkter bereik: hij is alleen binnen een loop beschikbaar. Buiten de loop bestaat hij niet en u kunt er dan ook niet aan refereren. U krijgt een syntactische foutmelding als u dat toch zou proberen. Als de laatste waarde van de indexvariabele om de een of andere reden moet worden onthouden, moet de waarde van de teller worden toegekend aan een 'gewone' variabele voordat de loop wordt verlaten.

Bijvoorbeeld:

```
DECLARE
   v_laatste_waarde number := 0;
   ...
BEGIN
   ...
   for i_teller in 1 .. 20 loop
      v_laatste_waarde := i_teller;
      ...
      exit when logische_expressie;
      ...
    end loop;
    dbms_output.put_line( v_laatste_waarde );
END;
```

Tip
Hoewel het hierboven niet expliciet vermeld staat, is in elke soort loop het gebruik van het exit-commando toegestaan.

Verderop in het boek komt nog een variant van de for-loop aan de orde: de zogeheten cursor-for-loop.

2.6.2.1 Het continue-statement

In de vorige paragraaf heeft u gezien hoe u een loop met het exit-statement kunt verlaten. Op het moment van een exit-commando wordt niet alleen de huidige iteratie van de loop afgebroken, maar zelfs de hele loop afgesloten. Er is nog een commando waarmee u de huidige iteratie kunt stoppen, maar waarmee niet de hele loop wordt verlaten: met het statement continue stopt de iteratie, maar gaat de loop verder bij de volgende iteratie. U kunt het dus gebruiken om de loop-constructie een lus over te laten slaan.

Syntactisch kunt u continue vergelijken met het exit-statement. Ook de continue-regel kunt u aanvullen met een when-regel. In dit voorbeeld wordt weer een lijst van datums afgedrukt, maar met het continue-statement slaan we nu de dagen in het weekend over. We tonen de continue in een for-loop maar ook in de eenvoudige en in de while-loop kunnen we de continue toepassen.

```
DECLARE
   v_sysdate date default sysdate;
   v_datum   date;
BEGIN
  FOR i_teller in 1 .. 10
  LOOP
    v_datum := v_sysdate + ( i_teller * 7 );
    if to_char( v_datum, 'd') in ('1','7')
```

```
   then
     continue;
   end if
   dbms_output.put_line( v_datum );
 END LOOP;
END;
```

Met de when-regel wordt de constructie nog wat eenvoudiger:

```
DECLARE
  v_sysdate date default sysdate;
  v_datum   date;
BEGIN
  FOR i_teller in 1 .. 10
  LOOP
    v_datum := v_sysdate + ( i_teller * 7 );
    continue when to_char( v_datum, 'd') in ('1','7');
    dbms_output.put_line( v_datum );
  END LOOP;
END;
```

2.6.3 Loop-labels

Vooral als er met geneste loops wordt gewerkt, kan het gebruik van zogeheten loop-labels de leesbaarheid en onderhoudbaarheid van een programma verhogen. Een loop-label is een identifier die tussen dubbele kleinerdan- en groterdantekens staat. Het label moet direct voor het begin van de loop-definitie staan. Deze labels dienen twee doeleinden. Ten eerste worden de leesbaarheid en onderhoudbaarheid vergroot, vooral als het om een omvangrijke loop gaat waarvan de code niet overzichtelijk op één pagina past. Ten tweede kan door het gebruik van loop-labels een reeks geneste loops in één keer worden verlaten.

De syntaxis is als volgt:

```
<<loop_label>>
LOOP
   ...;
END LOOP [ loop_label ];
```

Het volgende voorbeeld laat zien hoe bij geneste loops het label wordt gebruikt om niet alleen de huidige, maar ook de direct omvattende loop te verlaten.

```
<<buiten_loop>>
LOOP
   ...
   <<binnen_loop1>>
   LOOP
```

```
      ...
      <<binnen_loop2>>
      LOOP
         ...
         <<binnen_loop3>>
         LOOP
            ...
            exit binnen_loop2 when v_bool ;

         END LOOP binnen_loop3;
         ...
      END LOOP binnen_loop2; Programma gaat hier verder als v_bool TRUE wordt  ...

   END LOOP binnen_loop1;
   ...
END LOOP buiten_loop;
```

2.6.4 Goto

De syntaxis van PL/SQL laat het gebruik van labels en goto's toe. Met een goto-constructie kunnen we naar een willekeurige plek in het programma springen: naar voren, maar ook terug. We hebben daarvoor labels nodig die we op dezelfde manier definiëren als loop-labels. Met het volgende commando is het mogelijk naar zo'n label te springen:

```
GOTO label ;
```

Omdat het gebruik van goto-commando's in programmatuur vanwege de onderhoudbaarheid en leesbaarheid in het algemeen wordt afgeraden, gaan we hier niet verder in op deze mogelijkheid. In de regel zijn goto's in programma's overbodig, omdat ze altijd met behulp van loops, if-then-constructies en met toepassing van procedures herschreven kunnen worden.

2.7 Tabellen benaderen

In de voorbeelden die tot nu toe gebruikt zijn, zijn de tabellen buiten schot gebleven. Er zijn geen gegevens uit tabellen opgehaald en er zijn vanuit de voorbeeldprogramma's ook geen bewerkingen op tabellen uitgevoerd. In de praktijk zijn PL/SQL-programma's echter juist bedoeld om met gegevens uit een database te werken, om vervolgens bewerkingen op diezelfde database uit te voeren. In deze paragraaf komt aan de orde:

- hoe gegevens uit een database in PL/SQL-variabelen kunnen worden ondergebracht;
- hoe op basis van een PL/SQL-programma insert-, update- en delete-commando's naar een database kunnen worden gestuurd.

2.7.1 Gegevens uit tabellen ophalen

In veel gevallen moet een PL/SQL-programma werken met gegevens die afkomstig zijn uit een database. Die gegevens moeten worden toegekend aan PL/SQL-variabelen. Voor-

dat we kijken naar de methode om waarden uit tabellen aan variabelen toe te kennen, volgt hier een algemeen overzicht van de vijf manieren om een waarde aan een PL/SQL-variabele toe te kennen:

1. `DEFAULT`
   Gebruik van dit sleutelwoord is alleen toegestaan in het declaratiedeel van een PL/SQL-blok.
2. `:=`
   Dit is het algemene toekenningsteken dat in elk van de drie delen van een PL/SQL-blok gebruikt kan worden.
3. `select ... into`
   Met dit select-commando worden waarden die uit de database moeten worden opgehaald aan variabelen toegekend.
4. `fetch ... into`
   Hiermee worden waarden uit een cursor aan variabelen toegekend. In het hoofdstuk over cursors wordt dit commando nader behandeld.
5. `for recordvariabele in cursor_naam ... loop`
   Ook deze constructie wordt in het hoofdstuk over cursors besproken. Hiermee worden waarden uit een cursor aan variabelen toegekend.

Voor het ophalen van gegevens uit een database is in een PL/SQL-blok een select-commando nodig. De expressies die in de select-regel worden genoemd, moeten aan PL/SQL-variabelen worden toegekend. Daartoe wordt het commando uitgebreid, waarbij tussen de select-regel en de from-regel een into-regel wordt geplaatst. Deze into-regel noemt de PL/SQL-variabelen die door het select-commando worden gevuld.

De syntaxis is dus als volgt:

```
SELECT expressie
[,     expressie]
INTO   variabele
[,     variabele]
FROM tabel
...
```

De complete syntaxis van een select-commando kan worden toegepast, inclusief WHERE-regels, GROUP BY-regels en HAVING-regels.

Voorbeeld:

```
DECLARE
   v_maxsal number;
   v_salaris_stop boolean default false ;

BEGIN
   ...
```

```
   select max(med.maandsal)
   into v_maxsal
   from medewerkers med;
   ...
   if v_maxsal > 50000
   then
      v_salaris_stop := true;
   end if;
 ...;
END;
```

Restricties:

- Het select-commando moet precies één rij opleveren. Als het meer rijen oplevert, treedt er een runtime-fout op (een zogeheten exception). Dat is ook wel te begrijpen, want als met een select-into gepoogd wordt de variabele `v_maandsal` te vullen en het select-commando veertien maandsalarissen oplevert, welke zou dan in de variabele `v_maandsal` moeten worden ondergebracht? Ook als het select-commando geen enkele rij oplevert, treedt er een exception op. In een volgend hoofdstuk komt aan de orde hoe dergelijke exceptions met een zogeheten exception handler kunnen worden onderschept.
  Het is overigens op zichzelf wel mogelijk het resultaat van een query naar een variabele weg te schrijven als die query meer dan één rij oplevert. Daarvoor zijn collection-variabelen en BULK-bewerkingen nodig. Die komen in hoofdstuk 4 aan de orde.
- Omdat een Oracle-database geen datatype boolean kent, kunnen met deze constructie geen variabelen van het datatype boolean worden gevuld.
- Met deze constructie kan meer dan één variabele worden gevuld, maar er moeten wel altijd evenveel expressies in de select-lijst staan als er variabelen in de into-regel genoemd worden.

In de volgende voorbeelden ziet u foutmeldingen die verschijnen als het select-commando geen enkele rij of meer dan één rij oplevert:

```
DECLARE
   v_naam     medewerkers.naam%type;
   v_maandsal medewerkers.maandsal%type;
BEGIN
   select med.naam
   ,      med.maandsal
   into   v_naam
   ,      v_maandsal
   from   medewerkers med
   where  med.mnr = 007; -- deze medewerker komt niet voor

END;
```

```
DECLARE *
ERROR at line 1:
ORA-01403: no data found ORA-06512: at line 5


DECLARE
   v_naam     medewerkers.naam%type;
   v_maandsal medewerkers.maandsal%type;
BEGIN
   select med.naam
   ,      med.maandsal
   into   v_naam
   ,      v_maandsal
  from    medewerkers med; -- levert 14 rijen op

END;
DECLARE *
ERROR at line 1:
ORA-01422: exact fetch returns more than requested number of rows ORA-06512: at
line 5
```

Pas op

Stel, u hebt een PL/SQL-programma waarin met afzonderlijke select-commando's gegevens uit verschillende tabellen worden opgehaald om bijvoorbeeld statistische berekeningen uit te voeren. U wilt dan dat de tellingen een moment in de tijd weerspiegelen. Als de tabellen niet door anderen worden gemuteerd terwijl uw programma loopt, is er geen probleem. U kunt gerust achter elkaar een aantal tellingen uitvoeren. Maar als de tabellen wel gemuteerd worden, hebt u niet de garantie dat de tweede telling gebaseerd is op precies dezelfde database-inhoud als bij de eerste telling. U kunt deze zogeheten read consistency echter afdwingen, zonder dat u de database voor datamanipulatie op slot moet zetten. In een PL/SQL-programma kan namelijk het volgende SQL-commando worden toegepast:

```
SET TRANSACTION READ ONLY;
```

Hiermee wordt niet geforceerd dat andere gebruikers geen wijzigingen in de tabellen meer kunnen doorvoeren. Het is wel zo dat deze wijzigingen in uw sessie niet worden gezien. U creëert hiermee een momentopname die *read consistent* wordt genoemd. In deze read only-modus kunt u overigens zelf ook alleen maar select-commando's uitvoeren en geen datamanipulatie doen. Door een commit of rollback heft u de read only-status weer op.

2.7.2 Gegevens in tabellen wijzigen

Met een uitbreiding van de syntaxis van het select-commando (met de into-regel) is het mogelijk gegevens uit de database in een PL/SQL-programma onder te brengen. Omgekeerd is het ook mogelijk om vanuit een PL/SQL-programma gegevens in de database te wijzigen. Daartoe kunnen alle DML-commando's die de taal SQL rijk is zonder enige restrictie of aanpassing van de syntaxis worden ingezet. Zo kunnen insert-, update- en delete-commando's op dezelfde manier als in SQL worden gebruikt. Als extraatje komt er in PL/SQL bij dat PL/SQL-variabelen in de commando's kunnen worden opgenomen.

Overigens kunnen in PL/SQL-programma's ook transaction control-commando's worden gebruikt:

- `COMMIT [WORK]`
- `SAVEPOINT` *`savepoint_naam`*
- `ROLLBACK [TO` *`savepoint_naam`*`]`

Met deze commando's kunnen wijzigingen in tabellen definitief worden gemaakt, dan wel geheel of gedeeltelijk worden teruggedraaid. De werking van deze commando's is precies zoals in SQL is gedefinieerd. Voor een uitvoerige behandeling verwijzen we naar *Leerboek Oracle SQL*.

Hier volgt een voorbeeld van DML-commando's in een PL/SQL-blok:

```
DECLARE
   v_max_sal medewerkers.maandsal%type;
BEGIN
   select max(med.maandsal)
   into v_max_sal
   from medewerkers med;

   insert into stat_table (tekst , getal )
   values ('maximum maandsal is ',v_max_sal);

   delete from medewerkers
   where maandsal = v_max_sal;

   update afdelingen afd
   set afd.locatie = 'LEIDEN'
   where afd.naam = 'VERKOOP';

   commit;
END;
```

Bij de behandeling van het commando select-into werd de restrictie besproken dat het precies één rij moest ophalen. Als er geen rij werd gevonden, werd het programma met een foutmelding afgesloten. In een PL/SQL-programma geldt een dergelijke restrictie voor een DML-commando *niet*. Een DML-commando dat geen enkele rij bewerkt, heeft

geen foutmelding tot gevolg. Het is immers niet fout om een lege verzameling te bewerken.

Oracle ziet het niet als fout dat een DML-commando nul rijen bewerkt en forceert geen exception. Hoe kunt u dan wél achterhalen of een delete-commando effect heeft gehad of hoeveel rijen er zijn verwijderd? Deze informatie kan worden achterhaald via zogeheten cursorattributen, we die in paragraaf 2.7.4 beschrijven.

2.7.3 Rijen locken

In de praktijk komt het vaak voor dat u in een PL/SQL-programma gegevens uit een tabel ophaalt en naar variabelen wegschrijft, om vervolgens in het programma te rekenen met die gegevens. Als u dan tot slot op basis van die berekeningen wijzigingen in de database wilt doorvoeren, met name in de rij waarvan u gegevens hebt opgehaald, ligt er een mogelijk probleem. In principe is het mogelijk dat de rij in de tabel is gewijzigd tussen het moment waarop u de gegevens ophaalde en het moment dat u uw wijzigingen wilt wegschrijven. We kunnen immers met meerdere gebruikers op hetzelfde moment met dezelfde tabel aan de slag. Oracle regelt via zijn locking-mechanisme dat nooit twee of meer gebruikers op hetzelfde moment dezelfde rij kunnen muteren: degene die het eerst een rij bijwerkt of verwijdert, mag met die rij aan de slag en andere gebruikers moeten nu wachten totdat deze gebruiker zijn transactie met een commit of een rollback heeft afgerond. Zodra dat is gebeurd, kan de volgende gebruiker verder, omdat de lock door de commit of rollback is verdwenen en de rij is vrijgegeven. Dit mechanisme van row-level-locking, dat wordt ingeschakeld op het moment van muteren, is een mooi mechanisme. Bij het ophalen van gegevens worden geen locks geplaatst, zodat gebruikers elkaar niet in de weg zitten als ze toevallig dezelfde gegevens uit een tabel nodig hebben.

In het geschetste PL/SQL-programma zou het echter wel fijn zijn geweest als bij het ophalen van de gegevens al een lock op de desbetreffende rij zou zijn geplaatst. We hadden dan de garantie gehad dat anderen de rij in de tabel niet kunnen aanpassen of verwijderen. En als wij dan zelf onze mutatie doorvoeren, zouden we zeker weten dat de gegevens in ons programma in overeenstemming zijn met de situatie in de database.

Welnu, het is mogelijk rijen in een tabel te locken op het moment dat we uit een tabel selecteren. We doen dat door het SELECT-statement uit te breiden met een extra regel: FOR UPDATE.

Rijen die aan de where-regel voldoen, zullen nu worden gelockt en ze blijven gelockt totdat in de Oracle-sessie een commit of rollback wordt uitgevoerd. De complete syntaxis voor deze for update luidt als volgt:

```
FOR UPDATE [ OF kolomnaam  [, kolomnaam ]]    [ WAIT [n] | NOWAIT]
```

We kijken eerst naar een eenvoudig voorbeeld:

```
DECLARE
   v_mnr      medewerkers.mnr%type;
   v_maandsal medewerkers.maandsal%type;
   v_comm     medewerkers.comm%type;
BEGIN
   -- gegevens ophalen en betreffende rij locken:
   select mnr
   ,      maandsal
   ,      comm
   into   v_mnr
   ,      v_maandsal
   ,      v_comm
   from   medewerkers
   where  naam = 'DE KONING'
   for update;


   -- hier zouden eventuele berekeningen
   -- kunnen staan waardoor v_maandsal
   -- wordt herberekend
   --  ...

   -- en tot slot de rij in de tabel bijwerken:
   update medewerkers
   set    maandsal = v_maandsal
   where  mnr = v_mnr;

END;
```

Merk op dat de rij in dit voorbeeld wordt gelockt op het moment dat de SELECT wordt uitgevoerd, maar dat de lock na afloop van het programma nog steeds bestaat: er is in het programma zelf nog geen commit of rollback uitgevoerd. Om de lock vrij te geven zal de transactie buiten het programma nog moeten worden afgesloten. Tot dat moment kunnen anderen de gegevens van DE KONING immers niet muteren.

Het kan natuurlijk voorkomen dat het niet lukt om de rij te locken die moet worden opgehaald. Er kunnen immers andere gebruikers zijn die op dat moment de rij aan het bewerken zijn en nog niet gecommit hebben. Dit zou betekenen dat ons programma even niet verder kan. We hebben dan een paar mogelijkheden:

1. We kunnen het programma onbeperkt laten wachten totdat de rij wordt vrijgegeven en voor ons beschikbaar is. Dit is het standaardgedrag en wordt geregeld door de default-optie WAIT.
2. We kunnen aangeven dat ons programma slechts even of helemaal niet moet wachten indien de rij al door een ander proces gelockt is. Dan moeten we het sleutelwoord

WAIT expliciet noemen en daarachter het aantal seconden vermelden dat ons programma maximaal mag wachten. Als die tijd is verstreken en ons proces nog steeds geen lock heeft kunnen leggen op de desbetreffende rij, wordt ons programma met een foutmelding afgebroken.

```
select maandsal
into   v_maandsal
from   medewerkers
where  naam = 'DE KONING'
for update wait 20;
```

Als de rij van DE KONING inderdaad al is gelockt, wacht het programma twintig seconden. Als de rij binnen die tijd vrijkomt, gaat ons programma ook verder. Maar als de rij na twintig seconden nog steeds gelockt is, stopt het programma met de volgende exception:

```
ORA-00054: resource busy and acquired with NOWAIT specified
```

We zagen bij de syntaxis overigens ook de optie NOWAIT. Dat is in feite hetzelfde als 0 seconden wachten, dus als WAIT 0.

We kijken nog even naar het effect van de FOR UPDATE als het SELECT-statement een join bevat. Als dat namelijk het geval is, worden standaard de rijen gelockt uit alle tabellen die in de join betrokken zijn. Maar we kunnen, als het standaardgedrag niet wenselijk is, regelen dat alleen rijen uit bepaalde tabellen worden gelockt. Daartoe wordt de for update-regel uitgebreid tot:

```
FOR UPDATE OF  tabelalias.kolomnaam
[,             tabelalias2.kolomnaam]
[,             ...                  ]
```

U specificeert vervolgens een kolom uit elke tabel waar u wel rijen gelockt wil hebben. In dit voorbeeld joinen we twee tabellen, maar locken we alleen een rij in een van de twee tabellen:

```
select med.naam
,      med.maandsal
,      afd.locatie
into   v_naam
,      v_maandsal
,      v_locatie
from   medewerkers med
,      afdelingen afd
where  med.afd = afd.anr
and    med.naam = 'DE KONING'
for update of med.mnr ;
```

In dit voorbeeld wordt geen enkele rij uit de tabel afdelingen gelockt, maar wel de rij van DE KONING uit de tabel medewerkers. Merk overigens op dat de regel

```
for update of med.mnr
```

misschien suggereert dat alleen de *kolom* mnr uit de tabel MEDEWERKERS wordt gelockt. Dit is niet het geval. Oracle kent row level locking maar geen column level locking. Dat hier toch een kolom genoemd moet worden, is een historisch overblijfsel uit de tijd dat Oracle van plan was column level locking te implementeren.

Tip
De FOR UPDATE-regel is geen PL/SQL-constructie, maar onderdeel van de syntaxis van het select-statement in SQL. U kunt dus ook buiten PL/SQL om met deze constructie tijdens het uitvoeren van een zoekopdracht rijen in een tabel locken.

2.7.4 Cursorattributen

Heel algemeen geldt dat voor de uitvoering van elk SQL-commando geheugenruimte, cursor genaamd, wordt gebruikt om het commando te verwerken en om wat informatie over de verwerking bij te houden. Het gedeelte van het geheugen dat hiervoor wordt gebruikt, heet *private SQL area*. De inhoud van deze private SQL area is toegankelijk. Het enige dat daarvoor nodig is, is dat er een naam aan wordt toegekend. In een volgend hoofdstuk wordt behandeld hoe u zelf cursors kunt definiëren en er een naam aan kunt toekennen. Het is niet mogelijk de naam van de cursor die Oracle voor DML-commando's gebruikt zelf te bepalen. Die cursor heeft al een voorgedefinieerde naam en heet *SQL*. Iedere keer dat er een nieuw SQL-commando wordt uitgevoerd, wordt de inhoud van de SQL-cursor overschreven met informatie over het meest recente commando. De informatie die via de cursor beschikbaar komt, heeft betrekking op het aantal rijen dat door het commando is bewerkt. Die informatie komt op twee manieren beschikbaar:

- via een getalletje: het aantal rijen dat bewerkt is;
- via boolean-variabelen die aangeven of er rijen zijn bewerkt (of niet).

Deze informatie komt beschikbaar via de cursorattributen:

Cursorattribuut	Waarde
%FOUND	Datatype boolean TRUE als een commando een of meer rijen heeft bewerkt; FALSE als een commando is uitgevoerd dat geen enkele rij heeft bewerkt; NULL als er nog geen SQL-commando is uitgevoerd.
%NOTFOUND	Datatype boolean Logische ontkenning van %FOUND.
%ROWCOUNT	Datatype number Een getal dat het aantal bewerkte rijen weergeeft, of NULL als er nog geen DML-commando is uitgevoerd.

Als een attribuut van een cursor wordt uitgevraagd, moet de naam van de cursor voor het attribuut worden geplaatst. Omdat de cursor die in deze paragraaf behandeld wordt de naam SQL heeft, moet de waarde van de attributen van deze cursor worden uitgevraagd met SQL%found, SQL%notfound en SQL%rowcount.

Voorbeeld:

```
DECLARE
  v_teller number;

BEGIN
   update medewerkers med
   set med.maandsal = med.maandsal + 100
   where med.functie = 'VERKOPER';

   if SQL%FOUND
   then
      v_teller :=SQL%ROWCOUNT;
      insert into log_tabel
      values(v_teller,' rijen bewerkt');
     commit work;
   end if;
  dbms_output.put_line('aantal =' || SQL%rowcount);
END;
```

De laatste regel van het programma schrijft de SQL%rowcount naar het scherm. Maar welk getal wordt er dan weggeschreven als er zes verkopers waren? En welk getal als er geen verkopers waren? Voor de beantwoording van die vraag is het relevant dat u zich realiseert dat de cursor-attributen betrekking hebben op het laatst uitgevoerde SQL-commando. Als er zes verkopers waren, dan regelt het programma dat er een insert op de log-tabel plaatsvindt. Het laatst uitgevoerde commando is dan de insert. De SQL%rowcount is dan dus 1, want er heeft 1 insert in de log-tabel plaatsgevonden. Maar als er geen verkopers waren, dan is de insert niet uitgevoerd en is de update het laatst uitgevoerde SQL-commando. Er zijn bij die update geen rijen bijgewerkt, dus de SQL%rowcount is 0!

Om problemen met de interpretatie van SQL%rowcount te voorkomen, is het verstandig om de rowcount altijd in een variabele op te vangen, direct na het SQL-commando. Door een goede naamgeving te gebruiken kunt u dan verderop in het programma altijd bepalen met welke rowcount u te maken heeft. Het bovenstaande voorbeeld zou er in de nettere vorm als volgt uit kunnen zien:

```
DECLARE
  v_teller_updates number;
  v_teller_inserts number;
```

```
BEGIN
   update medewerkers med
   set med.maandsal = med.maandsal + 100
   where med.functie = 'VERKOPER';
   v_teller_updates :=SQL%ROWCOUNT;

   if SQL%FOUND
   then
      insert into log_tabel
      values(v_teller,' rijen bewerkt');
      v_teller_inserts := SQL%rowcount;
     commit work;
   end if;
  dbms_output.put_line('aantal updates =' || v_teller_updates;
END;
```

Pas op

Het is niet mogelijk in SQL-commando's zelf te refereren aan cursorattributen. Waarom zou het insert-commando uit het vorige voorbeeld niet als volgt gedefinieerd kunnen worden?

```
insert into log_tabel values(SQL%ROWCOUNT,' rijen bewerkt');
```

De reden is dat de cursorattributen worden overschreven op het moment dat er een nieuw SQL-commando wordt uitgevoerd. De attributen kunnen dus ook pas worden benaderd nadat het commando is uitgevoerd en de verwerkingsinformatie bekend is. Daarom levert het bovenstaande commando een foutmelding op:

```
PLS-00229: Attribute expression within SQL expression PL/SQL: SQL
Statement ignored
```

2.8 Dynamisch SQL

We hebben nu gezien hoe u in een PL/SQL-blok gegevens uit de database ophaalt, hoe u gegevens in variabelen opslaat, hoe u met variabelen kunt rekenen en hoe u gegevens wegschrijft naar tabellen. Daarmee hebben we ook gezien wat geldige commando's in een PL/SQL-blok zijn, namelijk:

- PL-commando's: variabelen declareren, variabelen vullen, if-then-constructies, enzovoort;
- SELECT-statements: mits voorzien van de INTO-regel;
- DML-statements: inserts, updates en deletes;
- transactiestatements: rollback, savepoint en commit.

In dit lijstje komen de DDL-commando's, zoals create, drop, alter, grant en revoke, niet voor. Het is inderdaad niet mogelijk om vanuit een PL/SQL-blok bijvoorbeeld een create table-commando uit te voeren. Als we dat toch proberen, krijgen we een foutmelding:

```
SQL> BEGIN
  2     create table temptab
  3     (logtekst varchar2(100));
  4  END;
  5  /
   create table temptab
   *
ERROR at line 2:
ORA-06550: line 2, column 4:
PLS-00103: Encountered the symbol "CREATE" when expecting one of the following:
begin case declare exit for goto if loop mod null pragma
raise return select update while with <an identifier>
<a double-quoted delimited-identifier> <a bind variable> <<
close current delete fetch lock insert open rollback
savepoint set sql execute commit forall merge pipe
```

Het gaat dus bij het compileren van het programma al fout. En dat is jammer, want het is best denkbaar dat u voor een PL/SQL-programma bijvoorbeeld tijdelijk gebruik wilt maken van een tabel die er niet is. U wilt bijvoorbeeld de mogelijkheid hebben om aan het begin van een programma een tabel te maken en die in de loop van het programma te vullen met foutmeldingen. Maar als aan het eind van het programma blijkt dat de tabel leeg is, wilt u de tabel ook meteen weer droppen. Het voert te ver om in te gaan op de reden waarom deze commando's niet worden geaccepteerd. We kijken echter wel naar een constructie in PL/SQL waarmee het toch mogelijk wordt DDL-commando's uit te voeren. We maken dan gebruik van het volgende commando:

```
execute immediate commandostring ;
```

Dit commando verwacht dat *commandostring* een variabele of literal is van het datatype varchar2 en dat die tekst een willekeurig geldig SQL-commando bevat.

In ons voorbeeld om een tijdelijke tabel te maken zou het commando er als volgt uit kunnen zien:

```
SQL> BEGIN
  2     execute immediate 'create table temptab (logtekst varchar2(100))';
  3  END;
  4  /

PL/SQL procedure successfully completed.
```

Het is ook mogelijk het commando in een variabele te plaatsen en vervolgens aan die variabele te refereren:

```
DECLARE
   v_commando varchar2(200);
BEGIN
   v_commando := 'create table temptab (logtekst varchar2(100))';
   execute immediate v_commando;
END;
```

Het gebruik van execute immediate beperkt zich overigens niet tot DDL-commando's. Zoals gezegd, kunnen we er elk willekeurig SQL-commando mee laten uitvoeren. Een voorbeeld met een DML-commando:

```
DECLARE
   v_commando varchar2(200);
BEGIN
   v_commando := 'delete from medewerkers where mnr=7654';
   execute immediate v_commando;
END;
```

Maar ook SELECT-statements kunnen via execute immediate worden uitgevoerd:

```
SQL> DECLARE
  2     v_commando varchar2(200);
  3     v_naam     medewerkers.naam%type;
  4  BEGIN
  5     v_commando := 'select naam from medewerkers where mnr=7900';
  6     execute immediate v_commando into v_naam;
  7     dbms_output.put_line( v_naam );
  8  END;
  9  /
JANSEN

PL/SQL procedure successfully completed.
```

Let in dit voorbeeld op de plaats waar de INTO zich bevindt.

Execute immediate is een krachtige constructie. En dat is niet alleen zo omdat we daarmee een restrictie van PL/SQL kunnen omzeilen (namelijk dat we geen DDL-commando's mogen uitvoeren in een PL/SQL-blok). Het krachtige zit hem ook in het feit dat we nu een SQL-statement dynamisch kunnen opbouwen. Daarmee wordt bedoeld dat we tijdens het runnen van een programma een SQL-statement kunnen laten opbouwen en uitvoeren en dat we dus een programma een SQL-statement kunnen laten samenstellen. We zullen daar verderop in het boek nog voorbeelden van zien. Maar om een idee te geven van hoe zoiets werkt, volgt hier een voorbeeld. Dit programma zoekt uit welke

tabel het meest recent is aangemaakt en toont vervolgens de naam van die tabel en het aantal rijen in die tabel:

```
SQL>  DECLARE
 2     v_tabel  varchar2(35);
 3     v_query  varchar2(100);
 4     v_aantal pls_integer;
 5  BEGIN
 6     select object_name
 7     into   v_tabel
 8     from   user_objects
 9     where  object_type = 'TABLE'
10     and    created     = (select max(created)
11                           from user_objects
12                           where object_type = 'TABLE');
13     v_query := 'select count(*) from ' || v_tabel;
14     execute immediate  v_query into v_aantal;
15     dbms_output.put_line('Tabel : ' || v_tabel);
16     dbms_output.put_line('#rijen: ' || v_aantal);
17* END;
18  /
Tabel : TEMPTAB
#rijen: 0

PL/SQL procedure successfully completed.
```

Merk op dat dit programma zonder dynamisch SQL en dus zonder execute immediate niet kan worden gemaakt. U zou kunnen denken dat het volgende programma ook zou moeten werken:

```
 1  DECLARE
 2     v_tabel  varchar2(35);
 3     v_aantal pls_integer;
 4  BEGIN
 5     select object_name
 6     into   v_tabel
 7     from   user_objects
 8     where  object_type = 'TABLE'
 9     and    created     = (select max(created)
10                           from user_objects
11                           where object_type = 'TABLE');
12     select count(*)
13     into   v_aantal
14     from   v_tabel;
15     dbms_output.put_line('Tabel : ' || v_tabel);
16     dbms_output.put_line('#rijen: ' || v_aantal);
17* END;
```

In dit voorbeeld vullen we de variabele v_tabel en gebruiken we die variabele direct in het SELECT-statement. In de vorige paragraaf hebben we toch immers ook aan PL-variabelen in SQL-statements gerefereerd? Toch gaat het in dit geval niet goed als we het programma uitvoeren:

```
   from   v_tabel;
          *
ERROR at line 14:
ORA-06550: line 14, column 11:
PL/SQL: ORA-00942: table or view does not exist
ORA-06550: line 12, column 4:
PL/SQL: SQL Statement ignored
```

Zoals u ziet, gaat het nu mis, omdat Oracle op zoek gaat naar een tabel die als naam V_TABEL heeft. En die is er niet. Blijkbaar kunnen we in SQL-statements refereren aan PL-variabelen, maar met uitzondering van de FROM-regel. Gelukkig biedt execute immediate ons de mogelijkheid dit soort programma's toch te bouwen.

Tot slot nog enkele opmerkingen met betrekking tot execute immediate:

- U bent er zelf verantwoordelijk voor dat de commandostring een geldig SQL-commando is. Daarmee bedoelen we het volgende: een PL/SQL-programma wordt in twee stappen uitgevoerd. Eerst wordt het programma gecompileerd. Daarbij voert Oracle onder andere een controle op de syntaxis uit. Daarna wordt het programma uitgevoerd. Als u execute immediate gebruikt, wordt bij het controleren van de syntaxis alleen gecontroleerd of u achter execute immediate iets van het datatype tekst hebt staan. Er wordt niet gecontroleerd of dat een geldig SQL-commando is. Dat laatste ontdekt u pas als het programma wordt uitgevoerd. Op dit moment is dat verschil tussen compileren en uitvoeren nog niet merkbaar. Maar dat verschil is wel merkbaar als we met stored PL/SQL-objecten werken. In praktijk is het dan mogelijk programma's te bouwen waarbij het compileren slaagt, maar waarbij een foutmelding pas optreedt als een eindgebruiker met de door u gebouwde procedure aan de slag gaat. We gaan hier verder niet op in. Meer informatie over stored PL/SQL-objecten treft u aan in hoofdstuk 7.
- Gebruik nooit een execute immediate als een 'direct' SQL-commando ook mogelijk is. U profiteert er dan namelijk wel van dat bij het compileren al wordt gecontroleerd of het SQL-statement aan de syntaxis van SQL voldoet.
- Als u een DDL-commando naar de database stuurt, heeft zo'n commando altijd een impliciete commit tot gevolg. Daarbij maakt het niet uit of dat DDL-commando direct of via een omweg met execute immediate wordt uitgevoerd. In het eerste voorbeeld uit deze paragraaf zouden openstaande wijzigingen nu definitief zijn vastgelegd.

2.9 Commentaar toevoegen

Als u uw eigen programmacode schrijft, dan weet u in de regel goed wat u aan het doen bent op het moment dat u de code schrijft. Maar als u uw code moet aanpassen en u heeft die code al een tijdje niet meer gezien, dan kan het best zijn dat u goed moet lezen en puzzelen voordat u weer doorheeft hoe u uw programma ooit bedoeld had. En als u code van een andere programmeur leest, dan is het misschien nog veel moeilijker om te achterhalen wat die met zijn code bedoelde.

U kunt op veel manieren zorgen dat code (ook later weer) goed te interpreteren is. Dat begint bij een goede naamgeving. Als u variabelen gebruikt met de namen v1, v2, v3, in plaats van iets zinvols als v_maandsalaris, v_afdelingshoofd, v_opslag, dan bent u in de programmeerfase sneller klaar met typen. Maar u krijgt later onherroepelijk problemen als u ergens in de code variabele v2 tegenkomt en moet achterhalen wat daar nu eigenlijk in bewaard werd.

Ook als we later in dit boek procedures en functies gaan schrijven, al dan niet met parameters, dan benadrukken we het belang van een zinvolle naamgeving.

Maar met mooie naamgeving alleen komt u er niet altijd. Dan kan het verstandig zijn om commentaar toe te voegen aan de broncode. Er zijn twee manieren om in PL/SQL commentaar aan te duiden.

- Met `/*` duidt u het begin en met `*/` het einde van commentaar aan. U plaatst deze tekens om de tekst die als commentaar dient. Deze tekst kan over meerdere regels worden verspreid.
- Een alternatief is het gebruik van twee mintekens (`--`). Alles wat in het programma op dezelfde regel achter `--` staat, wordt als commentaar beschouwd.

Zeker bij grote en complexe programma's is het bijvoorbeeld verstandig om bovenin toe te lichten wat de bedoeling van het programma is, wie het wanneer heeft gemaakt en wat eventuele restricties zijn. Ook de wijzigingshistorie van een programma kan op deze manier in het programma zelf worden vastgelegd.

```
DECLARE
  /* Programma ter opschoning van log-regels
     12-12-2011 G.Rattink in kader van Project AX332
     N.B. Bevat dynamisch SQL met DDL-statements
          en zal dus impliciete commits uitvoeren
  */
BEGIN
 …
END;
```

Of

```
DECLARE
  -- Programma ter opschoning van log-regels
  -- 12-12-2011 G.Rattink in kader van Project AX332
  -- N.B. Bevat dynamisch SQL met DDL-statements
  --      en zal dus impliciete commits uitvoeren
BEGIN
 …
END;
```

Gebruik commentaarregels overigens alleen om zinvol commentaar toe te voegen. Programmacode die voor zichzelf spreekt, laat u zonder commentaar bestaan. Overbodig commentaar zoals in het voorbeeld hieronder zorgt er eerder voor dat code wolliger en onoverzichtelijk wordt…

```
DECLARE
  v_systeemdatum date;
BEGIN
  -- systeemdatum vastleggen
  v_systeemdatum := sysdate;
  …
END;
```

U kunt met deze commentaartekens bij het testen van uw programmatuur ook programmacode tijdelijk uitschakelen. Het is overigens verstandig om in de uiteindelijke productieversie van uw programma geen 'uitgecommentarieerde' programmacode meer te hebben staan.

2.10 Oefeningen

1. Schrijf een PL/SQL-programma en declareer twee variabelen. Schrijf in de ene uw Oracle-gebruikersnaam en in de andere de systeemdatum weg. Schrijf vervolgens de inhoud van beide variabelen naar het scherm.

2. Schrijf een programma waarin een tekstvariabele (`v_naam`), drie numerieke variabelen (`v_dagen`, `v_maanden`, `v_jaren`) en twee datumvariabelen (`v_vandaag` en `v_gebdatum`) zijn gedeclareerd. Ken tijdens de declaratie uw naam toe aan de tekstvariabele. Ken de systeemdatum en uw geboortedatum toe aan de datumvariabelen. Laat het programma berekenen wat uw leeftijd is, uitgedrukt in dagen.
   Bereken ook wat uw leeftijd is, uitgedrukt in maanden en eenmaal uitgedrukt in jaren. Schrijf het resultaat van de berekeningen weg naar de drie numerieke variabelen. Schrijf tot slot de inhoud van alle variabelen naar het scherm.

3. Schrijf drie programma's die de tafel van zes naar het scherm schrijven. Maak één keer gebruik van de eenvoudige loop, één keer van de while-loop en één keer van de numerieke for-loop. Welke loop kiest u in de praktijk?

4. Maak een programma dat uitrekent hoe vaak u tot nu toe in een weekend jarig was. Schrijf dat aantal naar het scherm. U kunt de SQL-functie to_char gebruiken om uit een datum de dagnaam of het dagnummer zichtbaar te maken.

5. Een verzekeringsmaatschappij kent twee soorten reisverzekeringen: tijdelijke en doorlopende. Voor tijdelijke reisverzekeringen gelden tarieven per dag; voor de doorlopende gelden tarieven per jaar. De tarieven zijn als volgt:
   - Voor een doorlopende verzekering betaalt men 8 euro aan poliskosten. Verder geldt per volwassene een bedrag van 50 euro en per kind 20 euro.
   - Voor een tijdelijke reisverzekering betaalt men ook 8 euro aan poliskosten. Maar vervolgens geldt per dag een tarief van 1,25 euro per volwassene en 0,75 euro per kind.
     a. Maak een programma dat uitrekent wat een tijdelijke en wat een doorlopende reisverzekering kost voor een gezin dat uit twee ouders en een kind bestaat en dat zeven dagen op vakantie gaat.
     b. Breid het programma uit en bereken hoeveel dagen het gezin op vakantie moet gaan voordat de doorlopende verzekering voordeliger wordt.

De volgende opgaven hebben betrekking op de demotabellen zoals die in bijlage A zijn beschreven. Met de scripts dropcase.sql en crecase.sql kunt u de tabellen droppen en maken. Als u de oorspronkelijke inhoud van de tabellen wilt terughalen, gebruikt u het script vulcase.sql. Deze scripts treft u aan op de meegeleverde cd-rom.

6. Schrijf een PL/SQL-programma dat een bedrag van € 8.000 gelijkmatig over alle medewerkers verdeelt.

7. Schrijf een PL/SQL-programma dat alle medewerkers een salarisverhoging van 10 procent geeft. Daarbij mag het totaal aan uit te keren maandsalarissen de grens van € 55.000 niet overschrijden. Het programma moet de bewerking dus zo vaak als mogelijk herhalen, maar de grens van € 55.000 mag niet worden overschreden.

8. Schrijf een PL/SQL-programma dat uitrekent wat (voor de werkgever) voordeliger is: iedereen 10% salarisverhoging of iedereen 80 euro erbij. Zorg ervoor dat de voordeligste verhoging ook wordt doorgevoerd.

9. Pas het programma van opgave 8 aan. Laat het programma een tabel met de naam LOG_TABEL maken als die nog niet bestaat. De tabel moet de volgende structuur hebben:

```
Name                          Null?    Type
----------------------------- -------- ---------------
GETAL                                  NUMBER
TEKST                                  VARCHAR2(200)
DATUM                                  DATE
```

Laat het programma op het eind de nieuwe som van de salarissen naar de log_tabel schrijven.

Register

Oracle-leerboeken van Academic Service
Deze derde druk van het *Leerboek Oracle PL/SQL* maakt deel uit van de reeks Oracle-leerboeken van Academic Service. Architectuur, beheer, ontwerp en data-warehousing zijn aspecten die in deze boeken aan de orde komen. De boeken zijn gericht op opleidingen in het hoger en wetenschappelijk onderwijs waar de Oracle softwareomgeving wordt onderwezen.

Over dit boek
Dit boek gaat uitgebreid in op de programmeertaal PL/SQL. Deze taal is een toevoeging aan de niet-procedurele taal SQL, die als basistaal geldt om relationele databases te benaderen. Met PL/SQL wordt het mogelijk om de kracht en eenvoud van de vierdegeneratietaal SQL te combineren met procedurele elementen die in derdegeneratietalen beschikbaar zijn. Kennis van deze taal is onmisbaar voor iedereen die bij het ontwerpen en bouwen van een Oracle-applicatie betrokken is.
Het eerste gedeelte van het *Leerboek Oracle PL/SQL* behandelt vooral de syntaxis van de taal. Daarbij wordt ingegaan op de structuur van een PL/SQL-programma, het declareren van variabelen, het definiëren van subprogramma's, het afhandelen van foutsituaties en het gebruik van cursors. Het tweede deel van het boek gaat in op de toepassing van PL/SQL in de Oracle-database. De meeste hoofdstukken zijn voorzien van opgaven en via de website is online materiaal beschikbaar, o.a. uitwerkingen van de oefeningen en scripts om de oefentabellen aan te maken.
Leerboek Oracle PL/SQL is in eerste instantie bedoeld voor het onderwijs op hbo- en academisch niveau, maar is ook geschikt voor gebruik in practicumsituaties en bij zelfstudie.

Over de auteur
Drs. Gilbert Rattink is senior consultant en trainer bij Transfer Solutions bv. Daarvoor was hij enige jaren werkzaam bij Oracle Nederland. Ook was hij als wetenschappelijk medewerker verbonden aan het Centrum voor Lexicale Informatie en betrokken bij de bouw van een omvangrijke Oracle-database met woordgegevens

ISBN 978 90 395 2661 3
NUR 123/995
9 789039 526613
www.academicservice.nl