Université Internationale de Rabat

THE INNOVATIVE UNIVERSITY

Advanced Databases

Pr. Hasna EL HAJI
Pr. Oussama EL HAJJAMY

September 2025

Brief Review: Database Fundamentals

Database (DB)

A collection of data stored together, without unnecessary redundancy, to serve multiple applications.

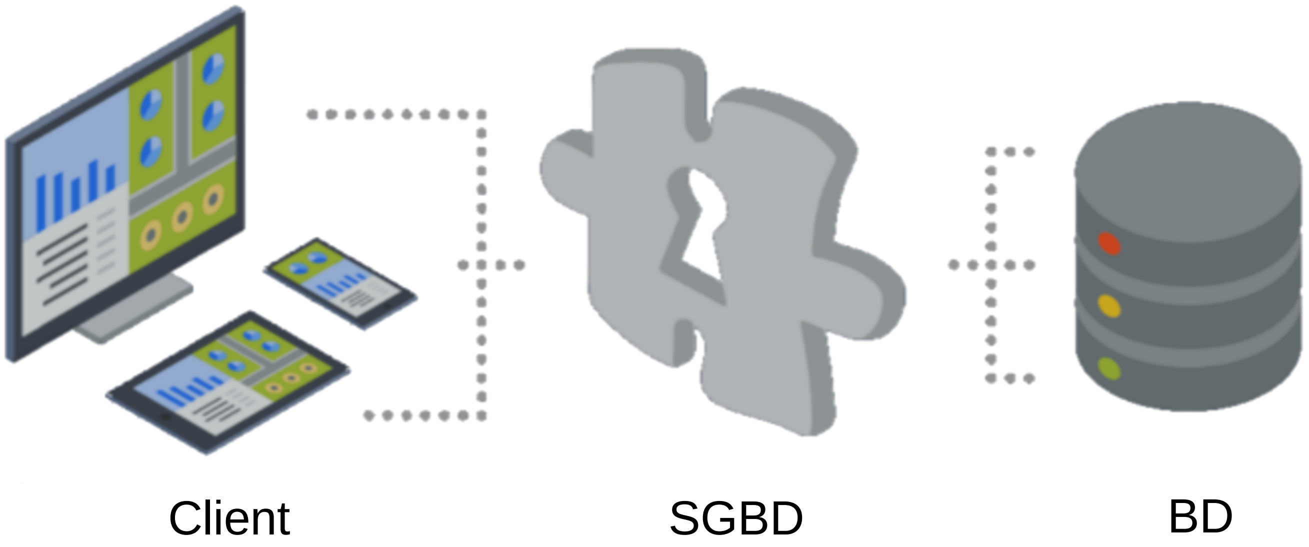
- The data is organized to remain independent of the programs that access it.
- It is structured to enable various operations, including reading, deletion, updating, sorting, and comparison, among others.

Brief Review: Database Fundamentals

Database Management System (DBMS)

The software that allows interaction with a database.

All operations on a database are made possible through the DBMS, which defines, manipulates, and controls the data.



Brief Review: Database Fundamentals

Relational Database Management System (RDBMS)

Various types of DBMS exist; however, the relational model (RDBMS) has long been established as the standard.

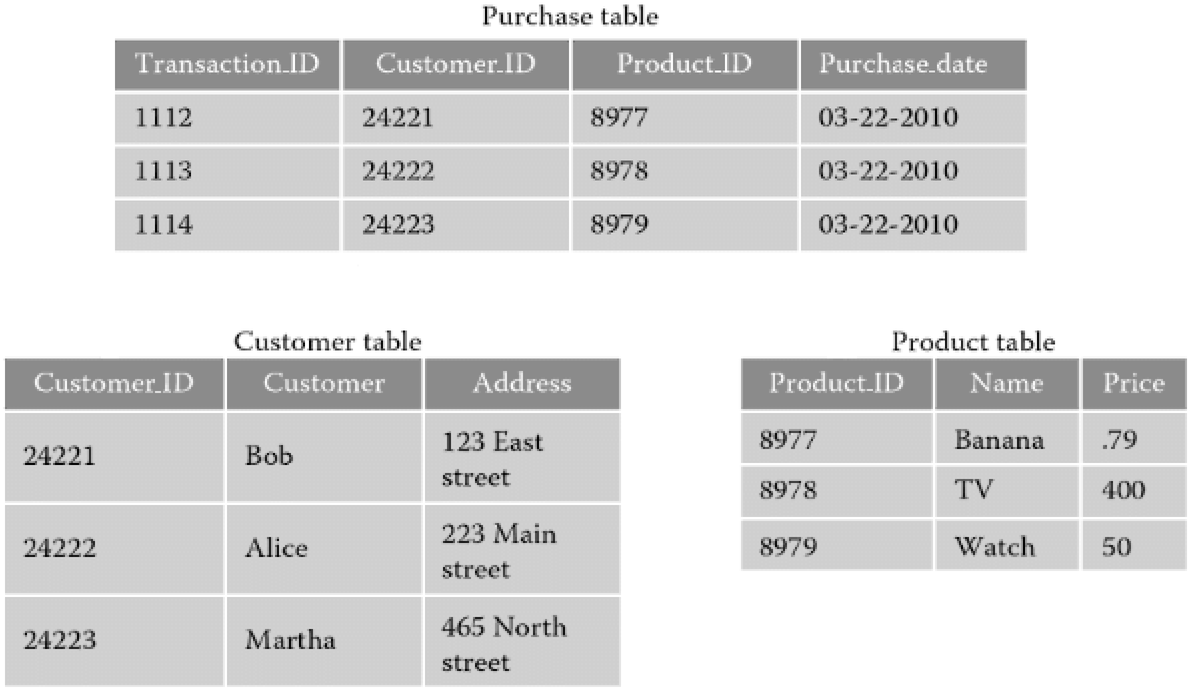
A variety of relational database software products are used on the market (e.g., Access, Oracle, SQL Server, PostgreSQL, Sybase, MySQL, DB2, etc.).

A DBMS facilitates data management through an intuitive and straightforward tabular representation.

Brief Review: Database Fundamentals

Table

A **table** in the relational model is a **relation** composed of **attributes**, with each attribute drawing its values from a defined **domain**.



Purchase table

Transaction_ID	Customer_ID	Product_ID	Purchase_date
1112	24221	8977	03-22-2010
1113	24222	8978	03-22-2010
1114	24223	8979	03-22-2010

Customer table

Customer_ID	Customer	Address
24221	Bob	123 East street
24222	Alice	223 Main street
24223	Martha	465 North street

Product table

Product_ID	Name	Price
8977	Banana	.79
8978	TV	400
8979	Watch	50

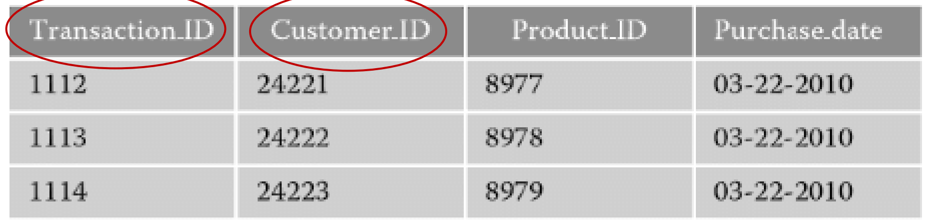
Brief Review: Database Fundamentals

Attributes

An attribute is a named column of a table that specifies the data that can appear in that column.

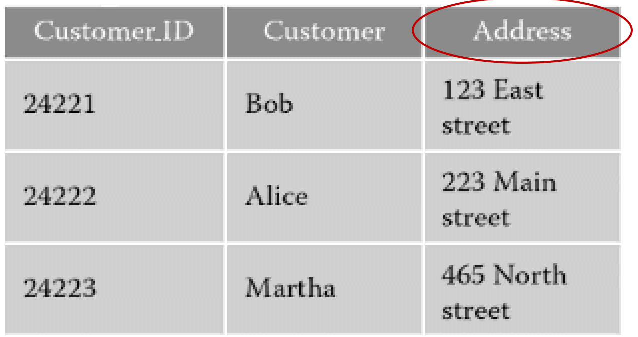
An attribute is defined by a name and a domain.

Purchase table



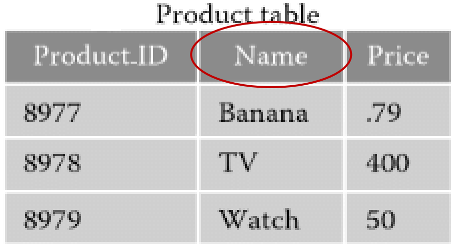
Transaction_ID	Customer_ID	Product_ID	Purchase_date
1112	24221	8977	03-22-2010
1113	24222	8978	03-22-2010
1114	24223	8979	03-22-2010

Customer table



Customer_ID	Customer	Address
24221	Bob	123 East street
24222	Alice	223 Main street
24223	Martha	465 North street

Product table



Product_ID	Name	Price
8977	Banana	.79
8978	TV	400
8979	Watch	50

Brief Review: Database Fundamentals

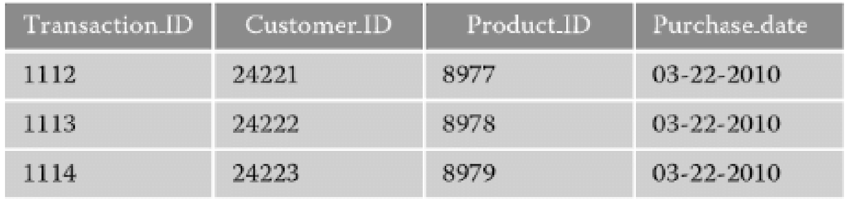
Domain

The domain of an attribute is the allowed set of values and rules for that attribute.

A domain combines data type, format, range, nullability, and any business rules that make values valid and meaningful.

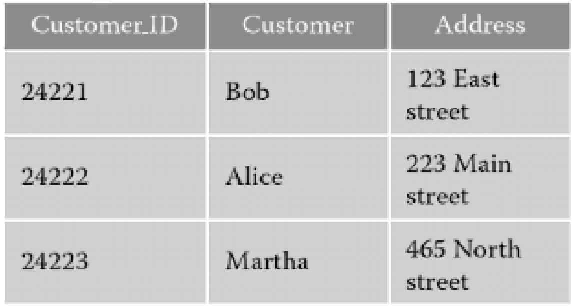
Examples

Purchase table



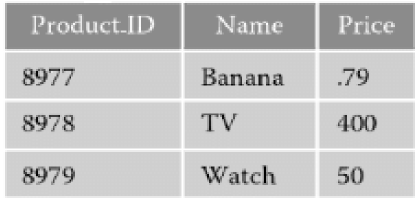
Transaction_ID	Customer_ID	Product_ID	Purchase_date
1112	24221	8977	03-22-2010
1113	24222	8978	03-22-2010
1114	24223	8979	03-22-2010

Customer table



Customer_ID	Customer	Address
24221	Bob	123 East street
24222	Alice	223 Main street
24223	Martha	465 North street

Product table



Product_ID	Name	Price
8977	Banana	.79
8978	TV	400
8979	Watch	50

Attribute	Domain
CustomerID	INT, NOT NULL, UNIQUE
Costumer	VARCHAR(200), NOT NULL
Price	REAL, CHECK (Price BETWEEN 18 AND 65)

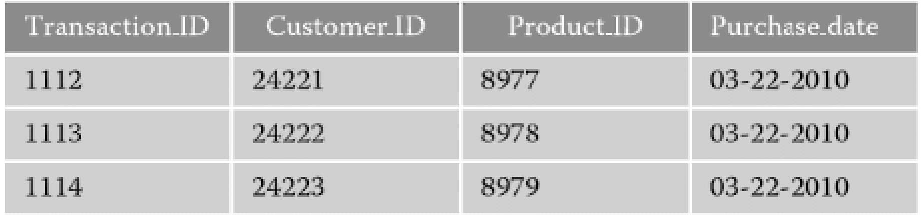
Brief Review: Database Fundamentals

Relation Schema

A relation schema R, denoted as R(A_1, A_2, …, A_n), is a set of attributes R = {A_1, A_2, …, A_n}.

- Each attribute A_i is associated with a dom
- A relation schema describes a relation.

Purchase table

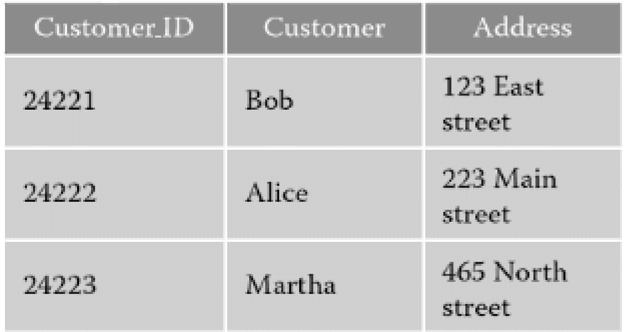


Transaction_ID	Customer_ID	Product_ID	Purchase_date
1112	24221	8977	03-22-2010
1113	24222	8978	03-22-2010
1114	24223	8979	03-22-2010

Example

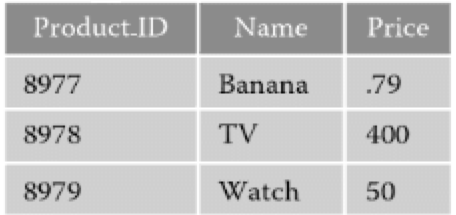
Customer(Customer_ID, Customer, Address)

Customer table



Customer_ID	Customer	Address
24221	Bob	123 East street
24222	Alice	223 Main street
24223	Martha	465 North street

Product table



Product_ID	Name	Price
8977	Banana	.79
8978	TV	400
8979	Watch	50

Brief Review: Database Fundamentals

N-tuple

In the relational model, a n-tuple represents a single, complete row in a relation (table), containing a value for each attribute.

Example

For the table **Customer(Customer_ID, Customer, Address)**, a possible n-tuple is:

(24221, 'Bob', '123 East Street')

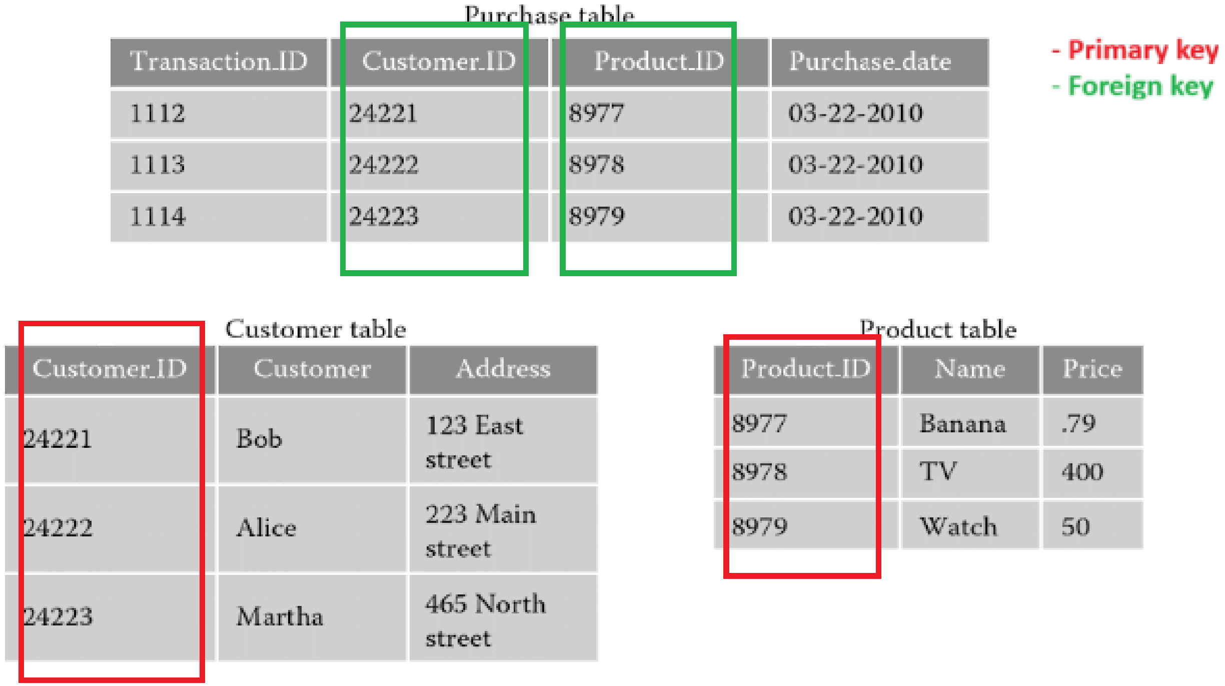
Brief Review: Database Fundamentals

Primary Key and Foreign Key

A **primary key** is a minimal set of one or more attributes whose values uniquely identify each tuple (row) in a relation and cannot contain null values.

A **foreign key** refers to a key originating from another table.

The concept of a foreign key is employed to establish the semantic relationship between two tables



Purchase table

Transaction_ID	Customer_ID	Product_ID	Purchase_date
1112	24221	8977	03-22-2010
1113	24222	8978	03-22-2010
1114	24223	8979	03-22-2010

- Primary key
- Foreign key

Customer table

Customer_ID	Customer	Address
24221	Bob	123 East street
24222	Alice	223 Main street
24223	Martha	465 North street

Product table

Product_ID	Name	Price
8977	Banana	.79
8978	TV	400
8979	Watch	50

Functional Dependencies

- Introduction

ProfMod						
ProfNum	ProfName	Salary	ModNum	ModName	Description	HL
1003	Tahiri	40000	570	JAVA	JAVA programming	48
1003	Tahiri	40000	580	XML	Elements of XML	32
…	…	…	…	…	…	…

(NumProf, NumMod) is the primary key of the relation ProfMod.

Insertion Anomaly: The addition of certain information is only possible if other related information is already present.

In the example above, details about a module can only be entered in association with a professor.

Functional Dependencies

- Introduction

ProfMod						
ProfNum	ProfName	Salary	ModNum	ModName	Description	HL
1003	Tahiri	40000	570	JAVA	JAVA programming	48
1003	Tahiri	40000	580	XML	Elements of XML	32
…	…	…	…	…	…	…

(NumProf, NumMod) is the primary key of the relation ProfMod.

Deletion Anomaly: The removal of certain information causes the loss of other related information.

In the example above, deleting the professor 'Tahiri' results in the deletion of the details of the modules he teaches.

Functional Dependencies

- Introduction

ProfMod						
ProfNum	ProfName	Salary	ModNum	ModName	Description	HL
1003	Tahiri	40000	570	JAVA	JAVA programming	48
1003	Tahiri	40000	580	XML	Elements of XML	32
…	…	…	…	…	…	…

(NumProf, NumMod) is the primary key of the relation ProfMod.

Update Anomaly: Modifying a single piece of information requires changes in multiple rows.

In the example above, updating the salary of professor 'Tahiri' must be performed in all rows where he appears.

Functional Dependencies

- Introduction

- Need to decompose the ProfMod relation to avoid the previous issues.

- The design of relations must take into account the dependencies among the various attributes.

Functional Dependencies

- Definition

Given a relation R, there is a functional dependency from X in R to Y in R if and only if: for every instance of R, if two tuples (rows) of R have the same values for the attributes in X, then they also have the same values for the attributes in Y.

It is written as: $X \rightarrow Y$

It is read as:

- There exists a functional dependency from X to Y.
- X determines Y.
- Y functionally depends on X.

Functional Dependencies

- Example

Country	
CountryID	CountryName
212	Morocco
33	France
34	Spain
212	Morocco
39	Italy
33	France
…	…

Two records with the same CountryID have the same CountryName.

$\forall$ x, y $\in$ Country :

x. CountryID = y. CountryID $\Rightarrow$ x. CountryName = y. CountryName

idPays → NomPays

Functional Dependencies

- Example

ProfMod						
ProfNum	ProfName	Salary	ModNum	ModName	Description	HL
1003	Tahiri	40000	570	JAVA	JAVA programming	48
1025	Younoussi	40000	420	Calculus	Functions	32
1003	Tahiri	40000	580	XML	Elements of XML	32
…	…	…	…	…	…	…

R : ProfMod

U = {ProfNum, ProfName, Salary, ModNum, ModName, Description, HL}

On a : {ProfNum, ModNum} → U

ProfNum → ProfName, Salary

ModNum → ModName, Description, HL

Functional Dependencies

- Purpose

➢ A functional dependency X → Y means that the value(s) of X uniquely determine the value(s) of Y.

➢ These dependencies help in identifying candidate keys.

➢ A candidate key of a relation is a minimal subset of attributes that functionally determines all attributes in the relation.

Elementary Functional Dependencies

- Definition
  - A functional dependency $X \rightarrow Y$ is said to be **elementary** if Y depends on the entire set X, and not just on a subset of X.
  - By definition, functional dependencies involving only two attributes ($X \rightarrow Y$) are always elementary.

Elementary Functional Dependencies

- Examples

- ProductRef → ProductName is elementary (involves only two attributes).
- (InvoiceNum, ProductRef) → QuantityOrdered is elementary (neither the product reference alone nor the invoice number alone is sufficient to determine the quantity).
- (InvoiceNum, ProductRef) → ProductName is not elementary, since the product reference alone is sufficient to determine the product name.

Invoice			
ProductRef	**ProductName**	**InvoiceNum**	**QuantityOrdered**
P001	Stylo Bleu	F1001	10
P002	Cahier A4	F1001	5
P001	Stylo Bleu	F1002	7
P003	Gomme	F1002	12

Direct Functional Dependencies

- Definition

A functional dependency $X \rightarrow Y$ is said to be **direct** if there does not exist an attribute Z that would create a transitive functional dependency $X \rightarrow Z \rightarrow Y$.

Direct Functional Dependencies

- Examples
- (InvoiceNum, ProductRef) → QuantityOrdered is a direct dependency, since the ordered quantity depends directly on the invoice–product combination.
  (InvoiceNum) → QuantityOrdered is not valid, because an invoice can contain multiple products.
  (ProductRef) → QuantityOrdered is not valid, because the same product can have different quantities in different invoices.

ProductInvoice			
InvoiceNum	**ProductRef**	**QuantityOrdered**	**UnitPrice**
F001	P001	5	20
F001	P002	2	15
F002	P001	1	20

Armstrong's Axioms

Armstrong's Axioms are a set of inference rules used to derive all functional dependencies in a relational database. The three basic rules are:

- Reflexivity

  $Y \subseteq X \Rightarrow X \rightarrow Y$

  Any set of attributes functionally determines its own subset.

  In particular, $X \rightarrow X$ is always true

- Augmentation

  $X \rightarrow Y \Rightarrow XZ \rightarrow YZ$

  Attributes can be added to both sides of a dependency without violating it.

- Transitivity

  $X \rightarrow Y$ and $Y \rightarrow Z \Rightarrow X \rightarrow Z$

  A dependency can be inferred through a chain of dependencies.

Additional rules

There are also additional derived rules (like Union, Pseudo-Transitivity, Reduction and Decomposition) that can be obtained from Armstrong's basic axioms:

- Union

  $X \rightarrow Y$ and $X \rightarrow Z \Rightarrow X \rightarrow YZ$

- Pseudo-Transitivity

  $X \rightarrow Y$ and $YW \rightarrow Z \Rightarrow XW \rightarrow Z$

- Reduction

  $X \rightarrow Y$ and $Z \subseteq Y \Rightarrow X \rightarrow Z$

- Decomposition

  $X \rightarrow YZ \Rightarrow X \rightarrow Y$ and $X \rightarrow Z$

Closure of a set of attributes

- Definition

The closure of a subset X of U, denoted by X^+, is defined as the set of all attributes A in U such that X → A.

- Purpose

The closure is used to determine all attributes functionally dependent on X and to test whether X is a candidate key.

Closure of a set of attributes

- Example

Relation: Employee(EID, EName, PNO, PName, Location, Duration)

Step 1: Determine Functional Dependencies (FDs)

FD = {

EID → EName,

PNO → {PName, Location},

{EID, PNO} → Duration

}

- The Employee ID (EID) uniquely determines the Employee Name (EName).
- In other words, for each employee, their ID corresponds to exactly one name.

- The Project Number (PNO) uniquely determines the Project Name (PName) and its Location.
- This means that each project number is associated with exactly one project name and one location.

- The combination of Employee ID (EID) and Project Number (PNO) determines the Duration of the employee’s work on that project.
- This implies that to know how long an employee worked on a project, you need both their ID and the project number.

Step 2: Find the closure of subsets (taking each FD and adding the attributes it determines)

EID$^+$ = {EID, EName}

PNO$^+$ = {PNO, PName, Location}

{EID, PNO}$^+$ = {EID, PNO, EName, PName, Location, Duration}

Superkey

- Definition

A superkey of a relation is a set of attributes that uniquely identifies the tuples in the relation.

No two distinct tuples have the same values for all attributes in the superkey.

X is said to be a superkey of R if X+ = U

Candidate key

- Definition

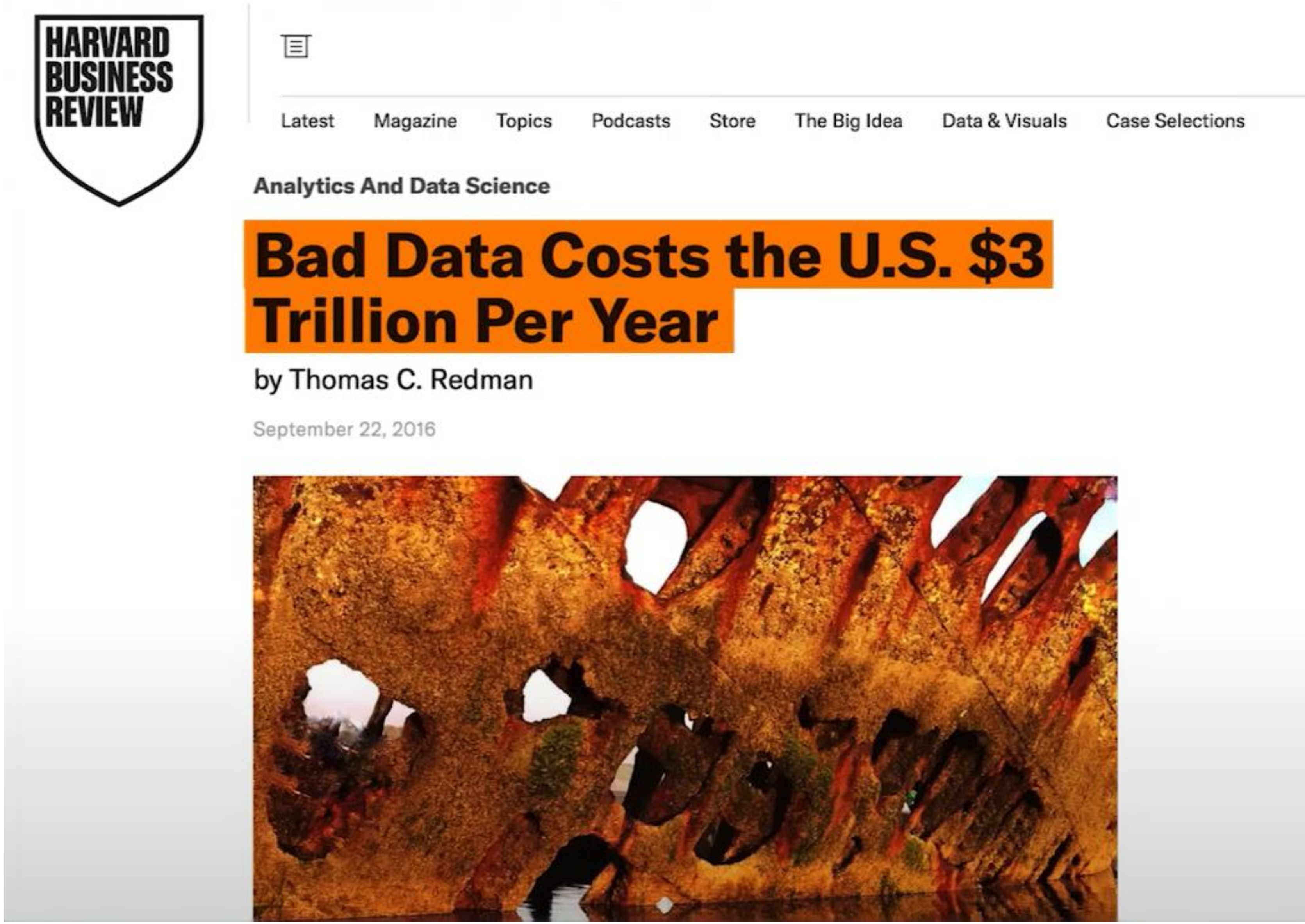
A candidate key is any superkey of a relation that is minimal (i.e., it ceases to be a superkey if any attribute is removed).

A relation can therefore have multiple candidate keys.

X is said to be a candidate key of R if $X^+ = U$ and there is no proper subset $Y \subset X$ such that $Y^+ = U$

Note: The primary key of R is a unique key selected from among the candidate keys.

Data Normalization



Analytics And Data Science

Bad Data Costs the U.S. $3 Trillion Per Year

by Thomas C. Redman

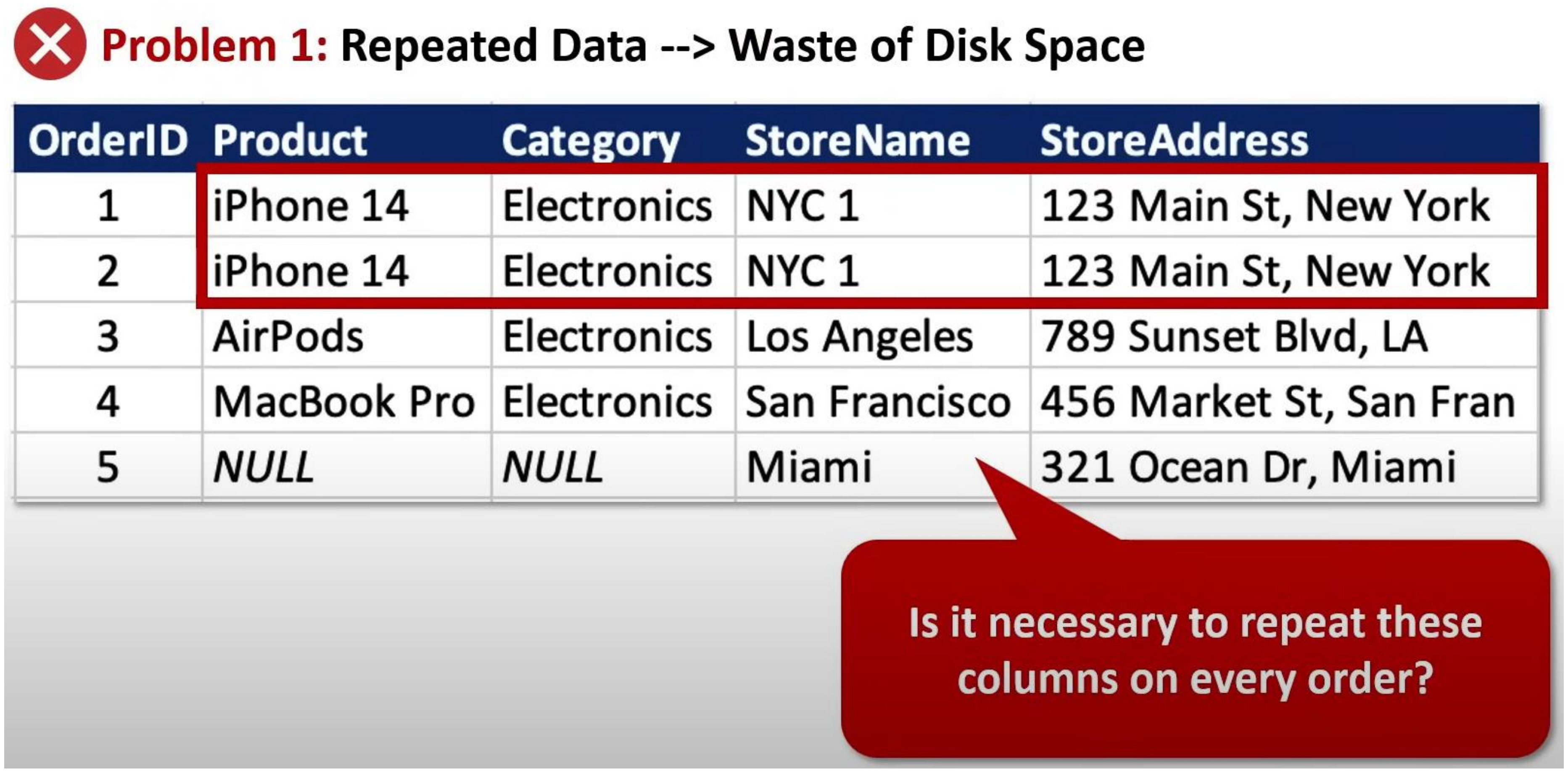
September 22, 2016

Data Normalization

- What is normalization?
  - Normalization is based on functional dependencies, which help us break a big set of data into smaller, well-organized tables.
  - Normalization involves multiple normal forms.
  - Each normal form is a step that reduces repetition and keeps the data cleaner and more consistent.

Data Normalization

- Why do we need to normalize data?



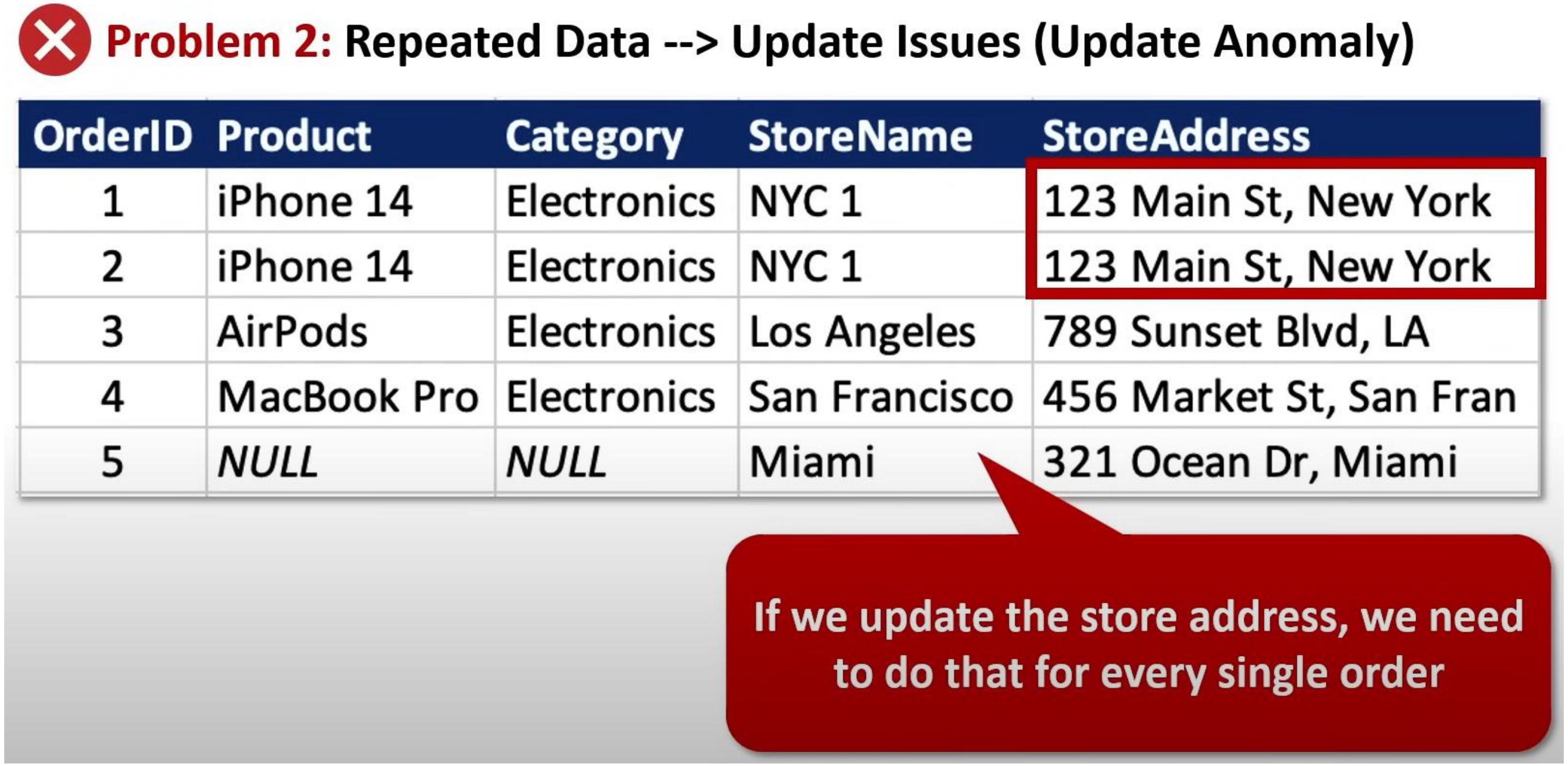
☒ **Problem 1: Repeated Data --> Waste of Disk Space**

OrderID	Product	Category	StoreName	StoreAddress
1	iPhone 14	Electronics	NYC 1	123 Main St, New York
2	iPhone 14	Electronics	NYC 1	123 Main St, New York
3	AirPods	Electronics	Los Angeles	789 Sunset Blvd, LA
4	MacBook Pro	Electronics	San Francisco	456 Market St, San Fran
5	*NULL*	*NULL*	Miami	321 Ocean Dr, Miami

Is it necessary to repeat these columns on every order?

Data Normalization

- Why do we need to normalize data?



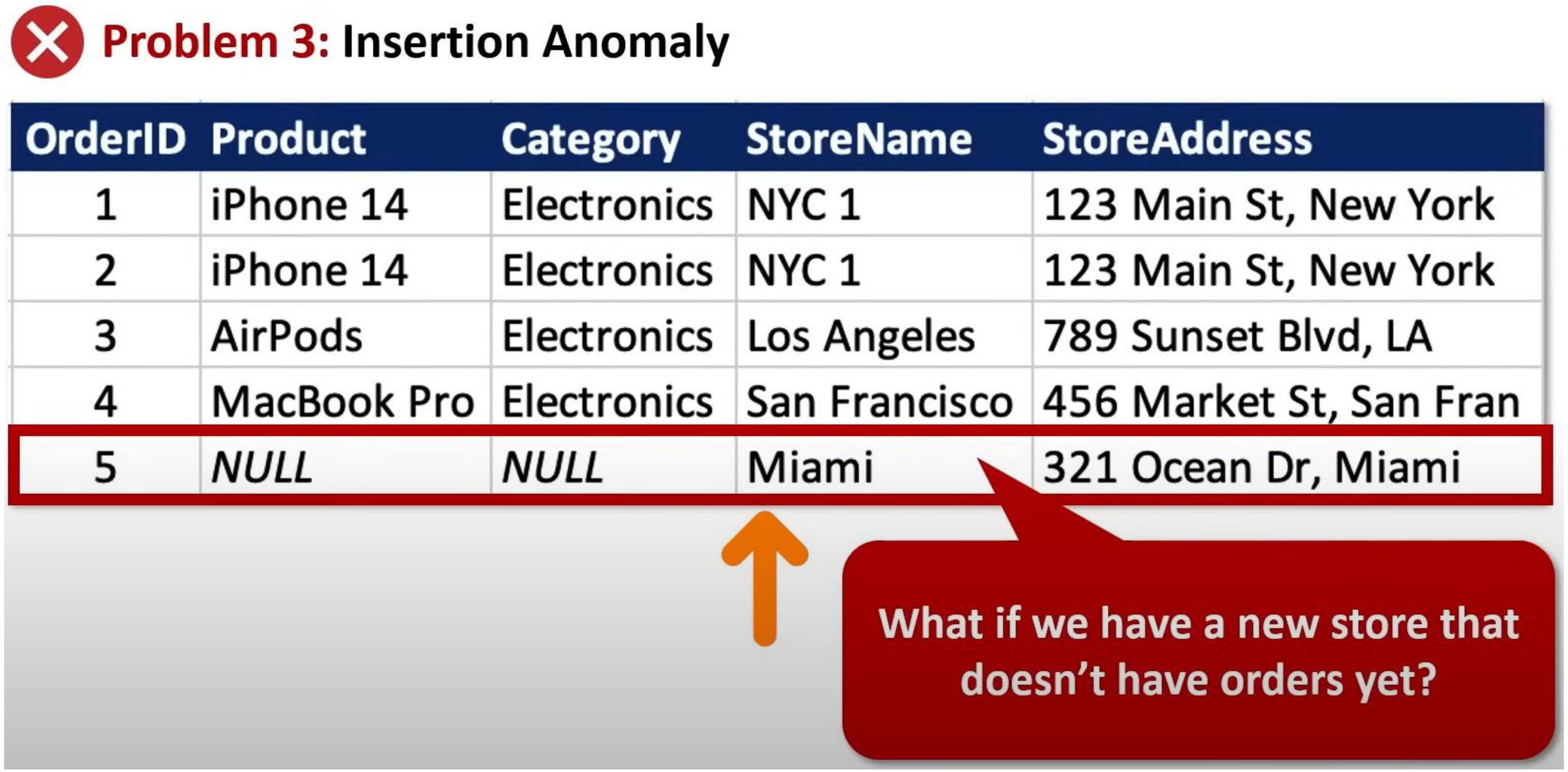
❌ Problem 2: Repeated Data --> Update Issues (Update Anomaly)

OrderID	Product	Category	StoreName	StoreAddress
1	iPhone 14	Electronics	NYC 1	123 Main St, New York
2	iPhone 14	Electronics	NYC 1	123 Main St, New York
3	AirPods	Electronics	Los Angeles	789 Sunset Blvd, LA
4	MacBook Pro	Electronics	San Francisco	456 Market St, San Fran
5	*NULL*	*NULL*	Miami	321 Ocean Dr, Miami

If we update the store address, we need to do that for every single order

Data Normalization

- Why do we need to normalize data?



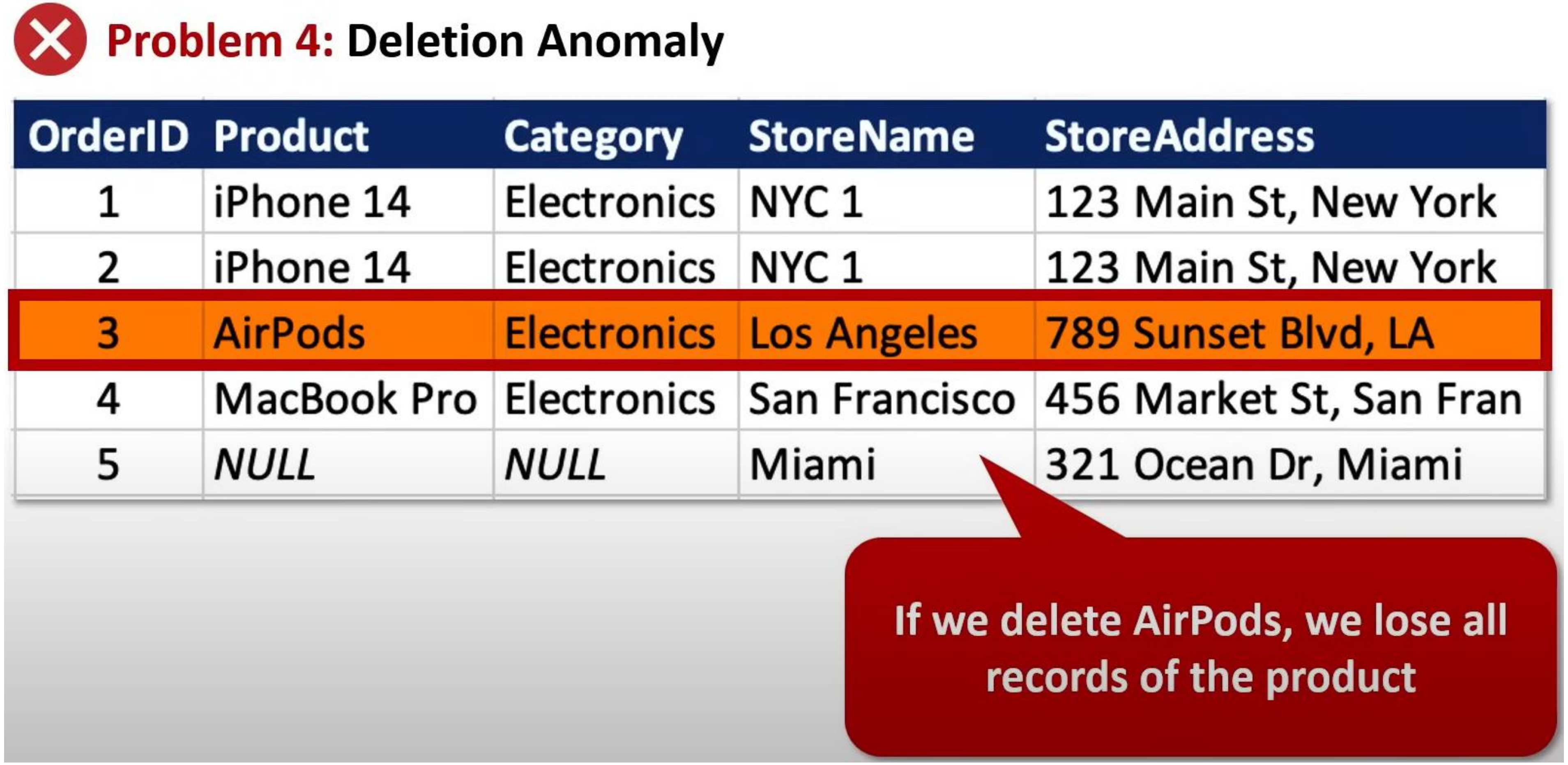
Problem 3: Insertion Anomaly

OrderID	Product	Category	StoreName	StoreAddress
1	iPhone 14	Electronics	NYC 1	123 Main St, New York
2	iPhone 14	Electronics	NYC 1	123 Main St, New York
3	AirPods	Electronics	Los Angeles	789 Sunset Blvd, LA
4	MacBook Pro	Electronics	San Francisco	456 Market St, San Fran
5	*NULL*	*NULL*	Miami	321 Ocean Dr, Miami

What if we have a new store that doesn't have orders yet?

Data Normalization

- Why do we need to normalize data?

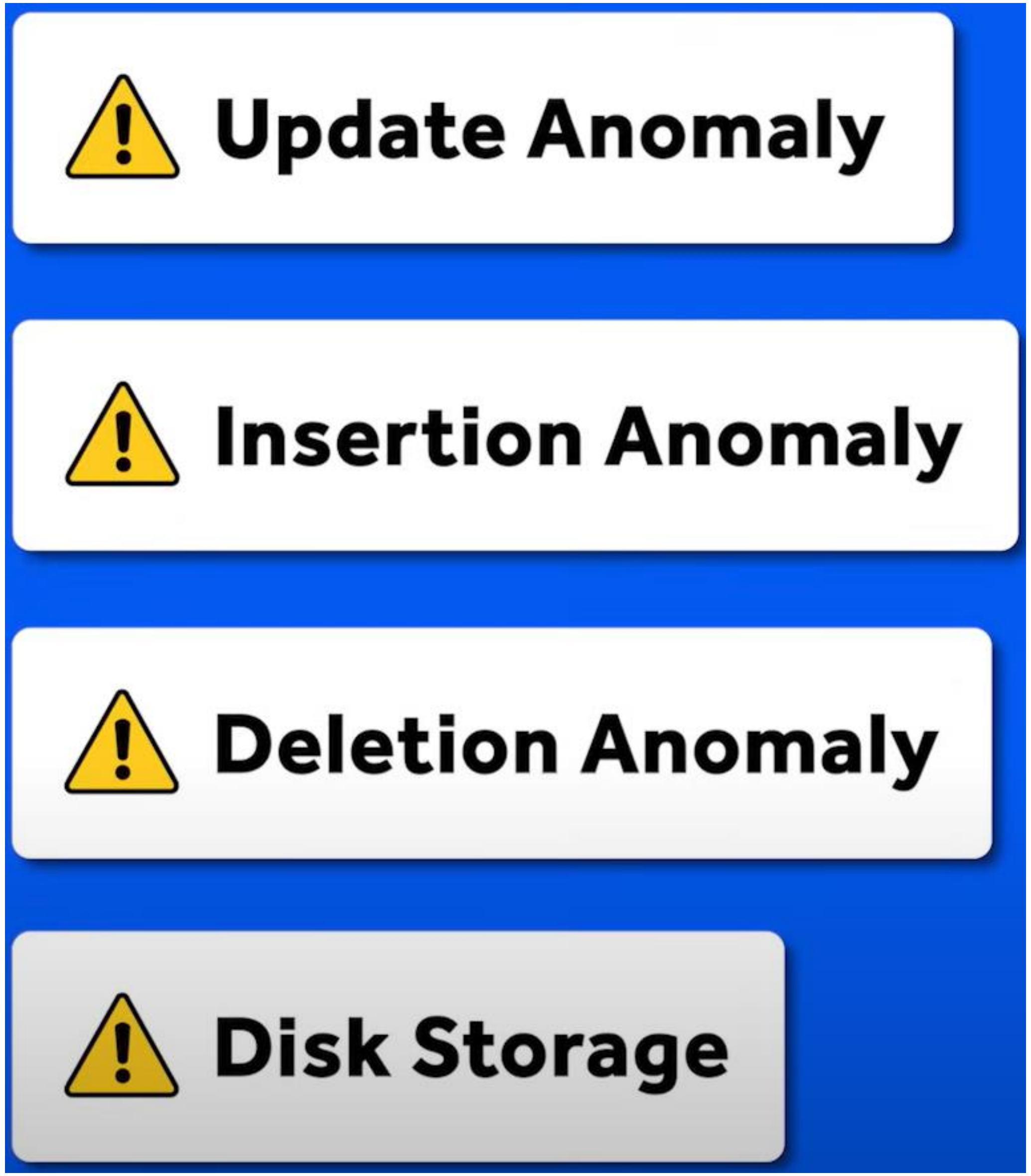


Problem 4: Deletion Anomaly

OrderID	Product	Category	StoreName	StoreAddress
1	iPhone 14	Electronics	NYC 1	123 Main St, New York
2	iPhone 14	Electronics	NYC 1	123 Main St, New York
3	AirPods	Electronics	Los Angeles	789 Sunset Blvd, LA
4	MacBook Pro	Electronics	San Francisco	456 Market St, San Fran
5	*NULL*	*NULL*	Miami	321 Ocean Dr, Miami

If we delete AirPods, we lose all records of the product

Why data normalization?



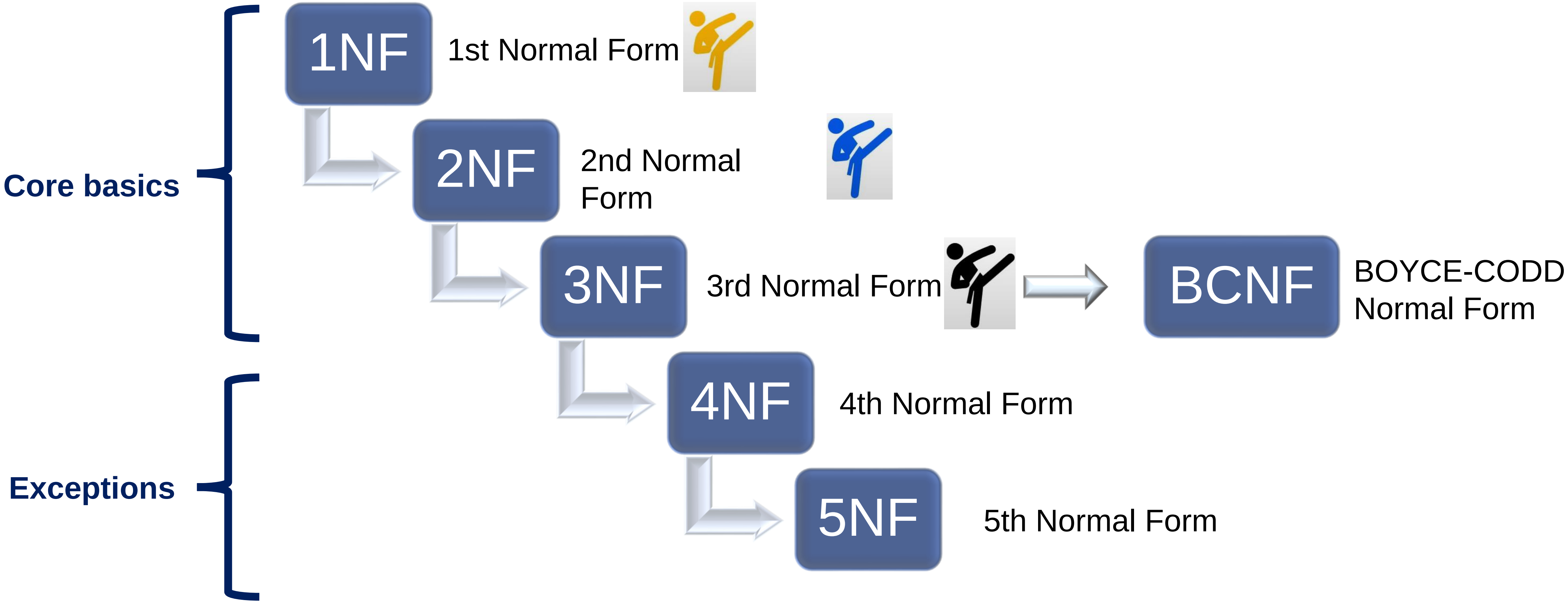
- ⚠ Update Anomaly
- ⚠ Insertion Anomaly
- ⚠ Deletion Anomaly
- ⚠ Disk Storage

Normal Forms

- Intuition
    - Normal forms are step-by-step design rules that guide in structuring a relational database, so data is consistent, non-redundant, and easy to maintain.
    - They could be seen as “levels of cleanliness,” where each level solves a specific set of problems.

Normal Forms

- 5 rules to normalize data



Core basics

1NF 1st Normal Form

2NF 2nd Normal Form

3NF 3rd Normal Form

BCNF BOYCE-CODD Normal Form

Exceptions

4NF 4th Normal Form

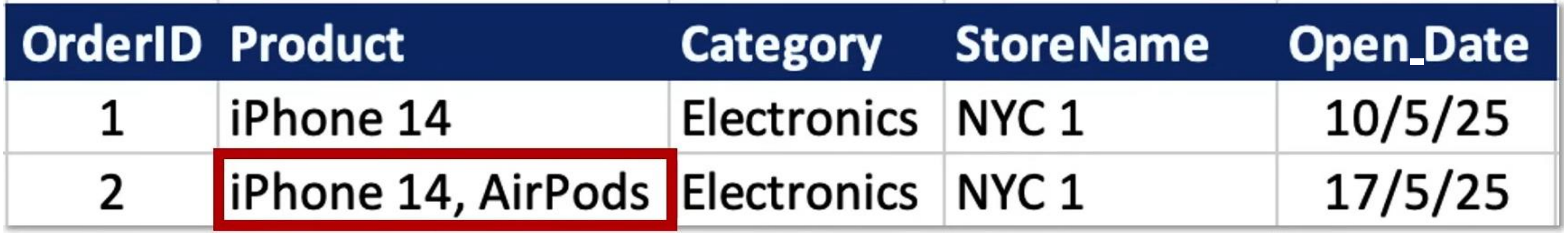
5NF 5th Normal Form

1st Normal Form

- Definition

A table is in First Normal form 1NF if it has no multi-valued attributes and each row is uniquely identifiable.

- Example



OrderID	Product	Category	StoreName	Open_Date
1	iPhone 14	Electronics	NYC 1	10/5/25
2	iPhone 14, AirPods	Electronics	NYC 1	17/5/25

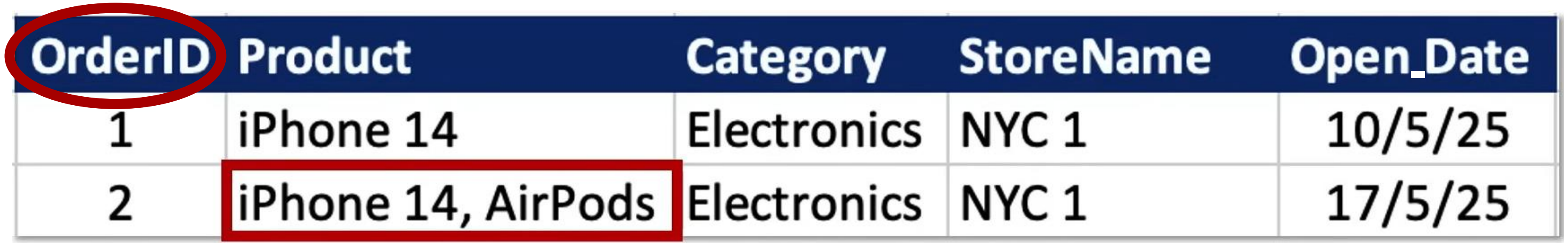
Multi-valued cell.

- How to bring a table into First Normal Form (1NF)?
  - Identify multi-valued cells and break them into separate rows.
  - If the existing key no longer ensures uniqueness, define a new key (possibly composite).

1st Normal Form

- Converting the example to 1NF

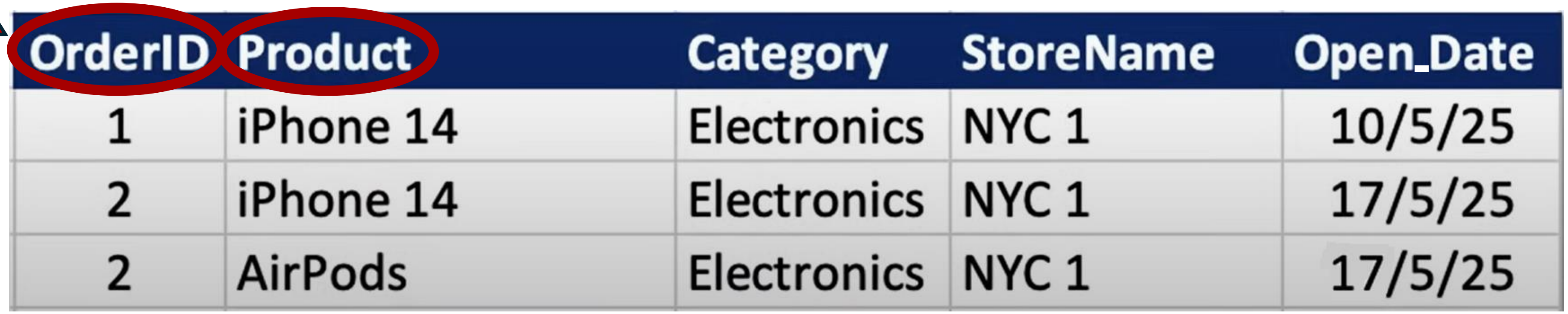
PK



OrderID	Product	Category	StoreName	Open_Date
1	iPhone 14	Electronics	NYC 1	10/5/25
2	iPhone 14, AirPods	Electronics	NYC 1	17/5/25

The primary key of the new table becomes the composite key (OrderID, Product), which ensures that each row is unique.

PK



OrderID	Product	Category	StoreName	Open_Date
1	iPhone 14	Electronics	NYC 1	10/5/25
2	iPhone 14	Electronics	NYC 1	17/5/25
2	AirPods	Electronics	NYC 1	17/5/25

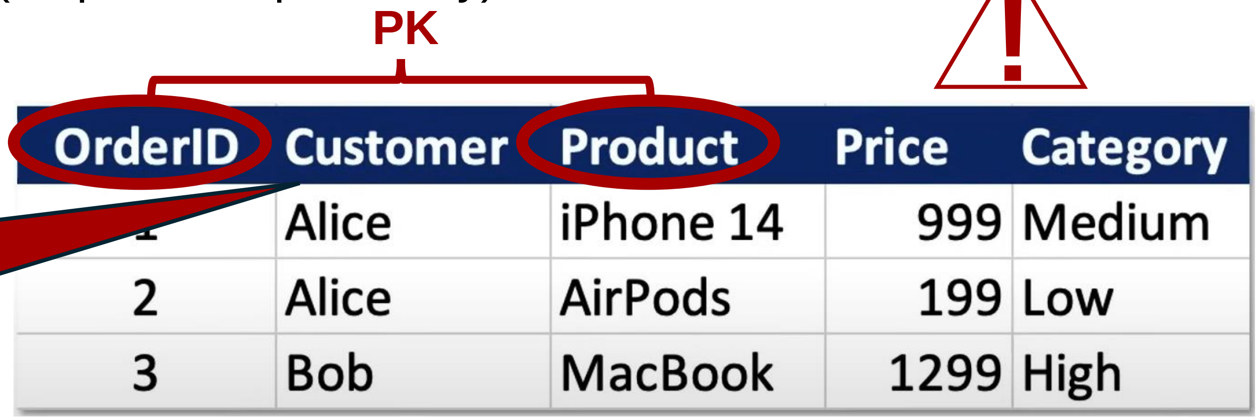
PK: Primary Key

2nd Normal Form

- Definition

A table is in Second Normal Form 2NF if it is in 1NF and every non-key attribute depends on the whole primary key (no partial dependency).

- Example



OrderID	Customer	Product	Price	Category
1	Alice	iPhone 14	999	Medium
2	Alice	AirPods	199	Low
3	Bob	MacBook	1299	High

Customer depends solely on the OrderID attribute, which is only a subset of the composite key (OrderID, Product). Therefore, Customer should be moved to a separate table.

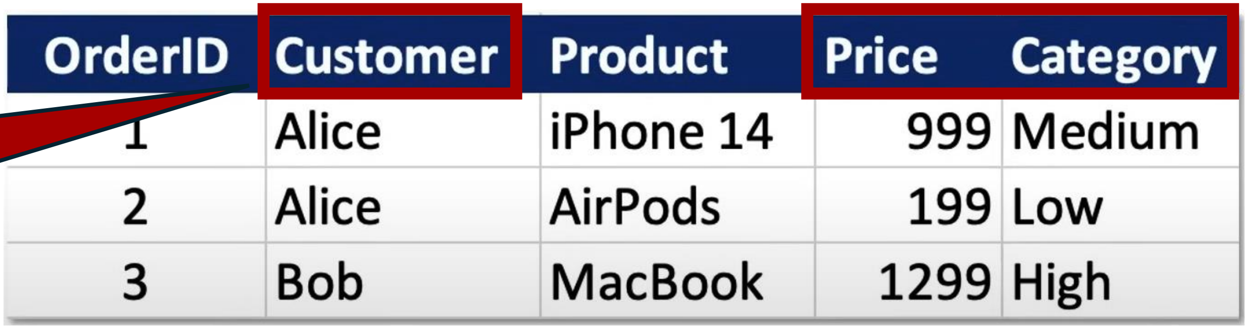
Price and Category depend solely on the Product attribute, which is only a subset of the composite key (OrderID, Product). Therefore, these attributes should be moved to a separate table.

- How to bring a table into Second Normal Form (2NF)?
    - The table must already have atomic values and a valid primary key (1NF).
    - Create new tables for attributes that depend only on a subset of the composite key.

2nd Normal Form

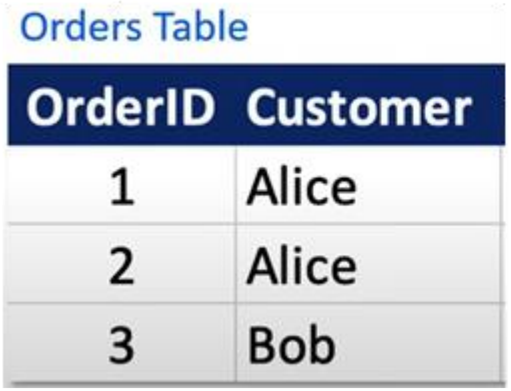
- Converting the example to 2NF

Customer depends solely on the OrderID attribute, which is only a subset of the composite key (OrderID, Product). Therefore, Customer should be moved to a separate table.



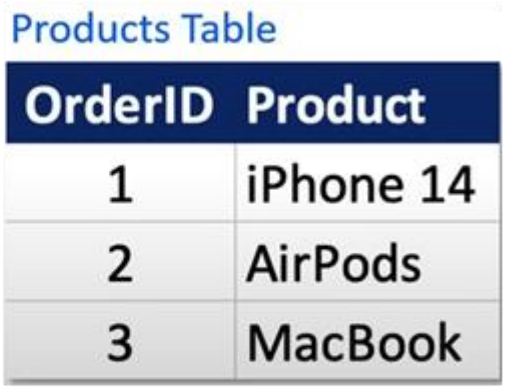
OrderID	Customer	Product	Price	Category
1	Alice	iPhone 14	999	Medium
2	Alice	AirPods	199	Low
3	Bob	MacBook	1299	High

Price and Category depend solely on the Product attribute, which is only a subset of the composite key (OrderID, Product). Therefore, these attributes should be moved to a separate table.



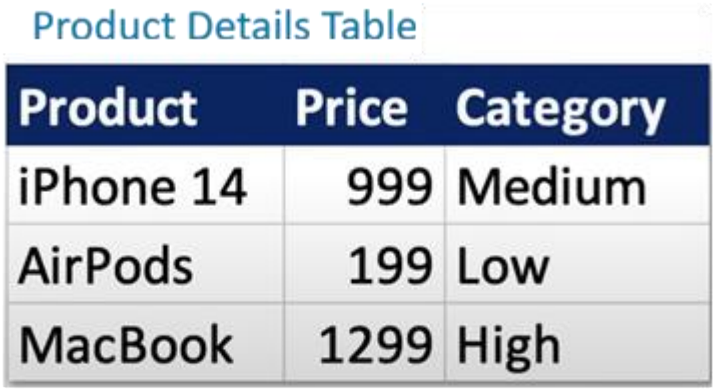
Orders Table

OrderID	Customer
1	Alice
2	Alice
3	Bob



Products Table

OrderID	Product
1	iPhone 14
2	AirPods
3	MacBook



Product Details Table

Product	Price	Category
iPhone 14	999	Medium
AirPods	199	Low
MacBook	1299	High

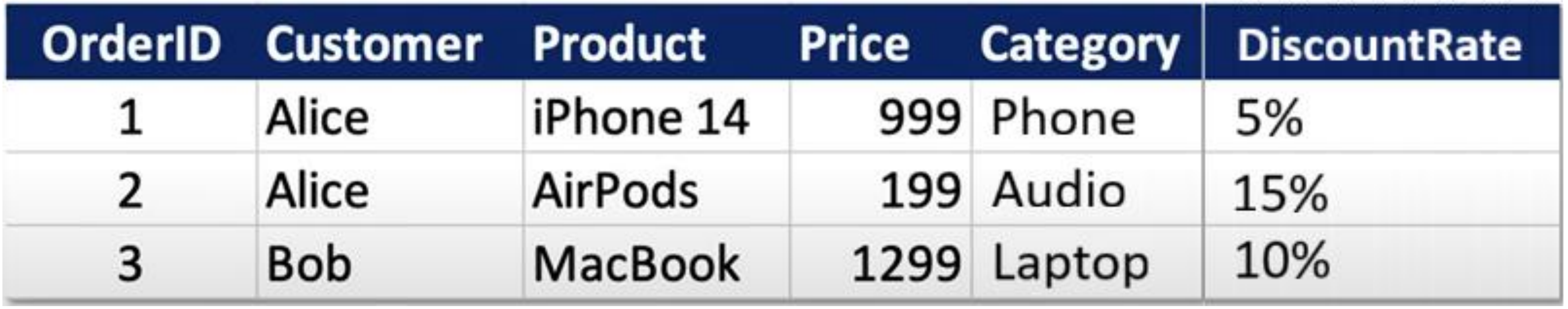
3rd Normal Form

- Definition

A table is in Third Normal Form 3NF if it is in 2NF and every non-key attribute must depend only on the primary key, and not on another non-key attribute (no transitive dependencies).

- Examp

The table is not in 2NF.



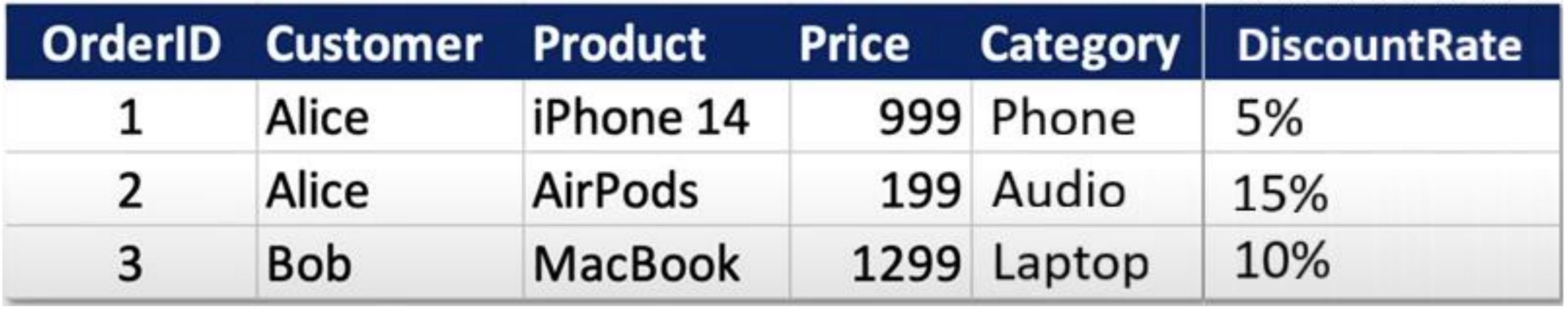
OrderID	Customer	Product	Price	Category	DiscountRate
1	Alice	iPhone 14	999	Phone	5%
2	Alice	AirPods	199	Audio	15%
3	Bob	MacBook	1299	Laptop	10%

There is a transitive dependency between Category and DiscountRate

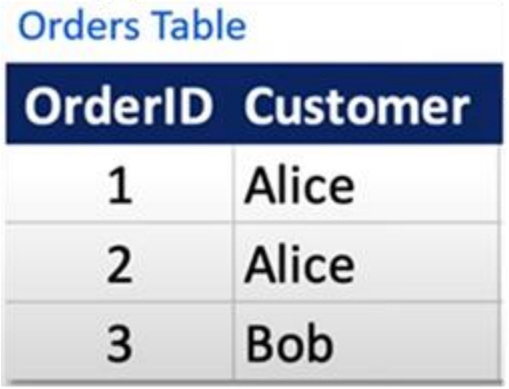
- How to bring a table into Third Normal Form (3NF)?
  - Verify 2NF: ensure all non-key attributes depend on the whole primary key.
  - Find transitive dependencies: look for attributes that depend on another non-key attribute rather than directly on the key.
  - Decompose: move those attributes into a new table.

3rd Normal Form

- Converting the example to 3NF

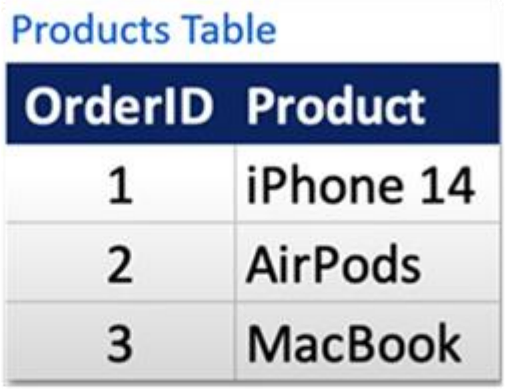


OrderID	Customer	Product	Price	Category	DiscountRate
1	Alice	iPhone 14	999	Phone	5%
2	Alice	AirPods	199	Audio	15%
3	Bob	MacBook	1299	Laptop	10%



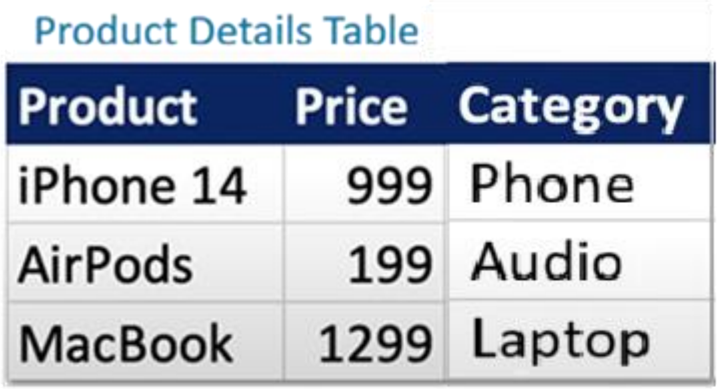
Orders Table

OrderID	Customer
1	Alice
2	Alice
3	Bob



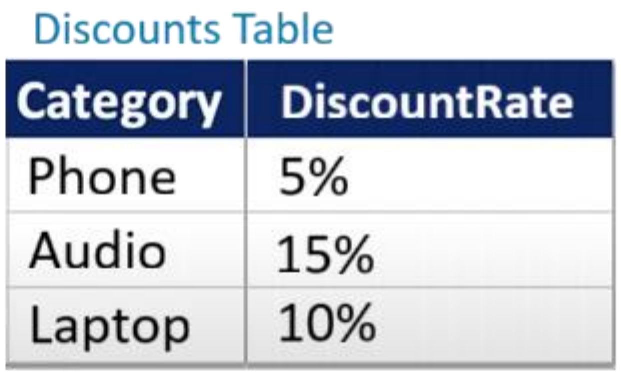
Products Table

OrderID	Product
1	iPhone 14
2	AirPods
3	MacBook



Product Details Table

Product	Price	Category
iPhone 14	999	Phone
AirPods	199	Audio
MacBook	1299	Laptop



Discounts Table

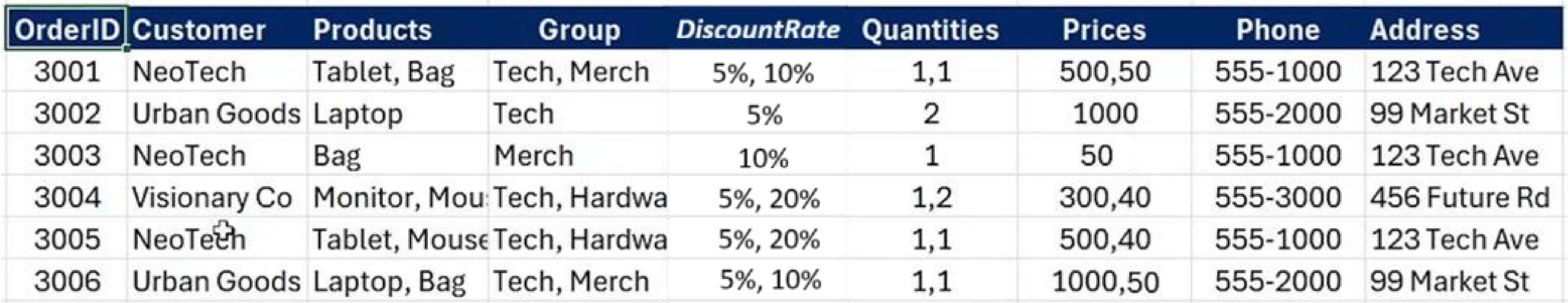
Category	DiscountRate
Phone	5%
Audio	15%
Laptop	10%

We Convert the table to 2NF first then 3NF.
Now each non-key attribute depends only on the key of its own table.

Boyce–Codd, 4th and 5th Normal Forms

- **Boyce–Codd** is a stricter version of 3NF
- It handles certain tricky cases where 3NF still allows anomalies.
- The **4th form** handles Multi-valued dependencies.
- For example, a professor can teach many courses and speak many languages so, we have to keep those relationships in separate tables.
- The **5th form** ensures that a table is divided into smaller tables so that all data can be recombined without repeating or contradicting anything

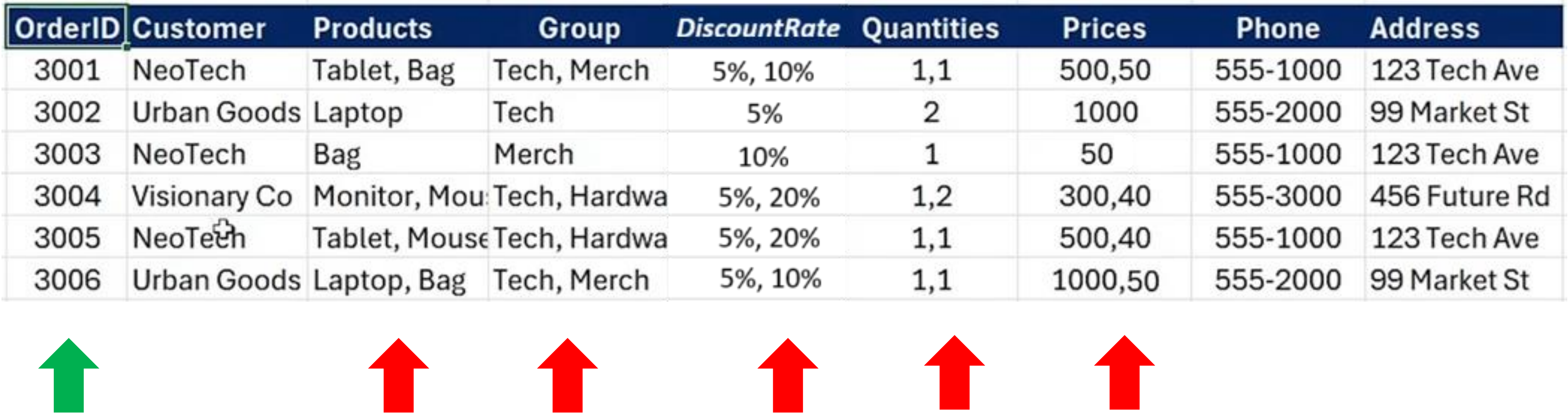
Practical Example



OrderID	Customer	Products	Group	*DiscountRate*	Quantities	Prices	Phone	Address
3001	NeoTech	Tablet, Bag	Tech, Merch	5%, 10%	1,1	500,50	555-1000	123 Tech Ave
3002	Urban Goods	Laptop	Tech	5%	2	1000	555-2000	99 Market St
3003	NeoTech	Bag	Merch	10%	1	50	555-1000	123 Tech Ave
3004	Visionary Co	Monitor, Mou	Tech, Hardwa	5%, 20%	1,2	300,40	555-3000	456 Future Rd
3005	NeoTech	Tablet, Mouse	Tech, Hardwa	5%, 20%	1,1	500,40	555-1000	123 Tech Ave
3006	Urban Goods	Laptop, Bag	Tech, Merch	5%, 10%	1,1	1000,50	555-2000	99 Market St

Practical Example

Is the table in 1NF?

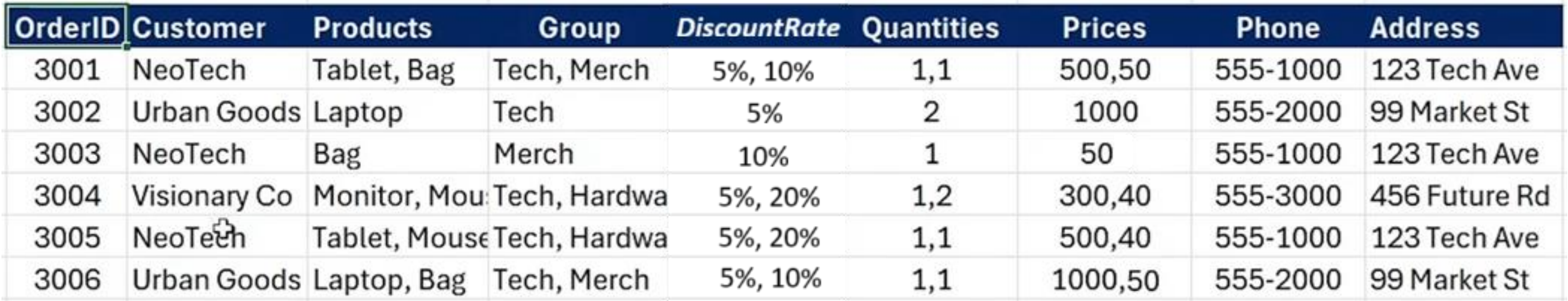


OrderID	Customer	Products	Group	*DiscountRate*	Quantities	Prices	Phone	Address
3001	NeoTech	Tablet, Bag	Tech, Merch	5%, 10%	1,1	500,50	555-1000	123 Tech Ave
3002	Urban Goods	Laptop	Tech	5%	2	1000	555-2000	99 Market St
3003	NeoTech	Bag	Merch	10%	1	50	555-1000	123 Tech Ave
3004	Visionary Co	Monitor, Mou	Tech, Hardwa	5%, 20%	1,2	300,40	555-3000	456 Future Rd
3005	NeoTech	Tablet, Mouse	Tech, Hardwa	5%, 20%	1,1	500,40	555-1000	123 Tech Ave
3006	Urban Goods	Laptop, Bag	Tech, Merch	5%, 10%	1,1	1000,50	555-2000	99 Market St

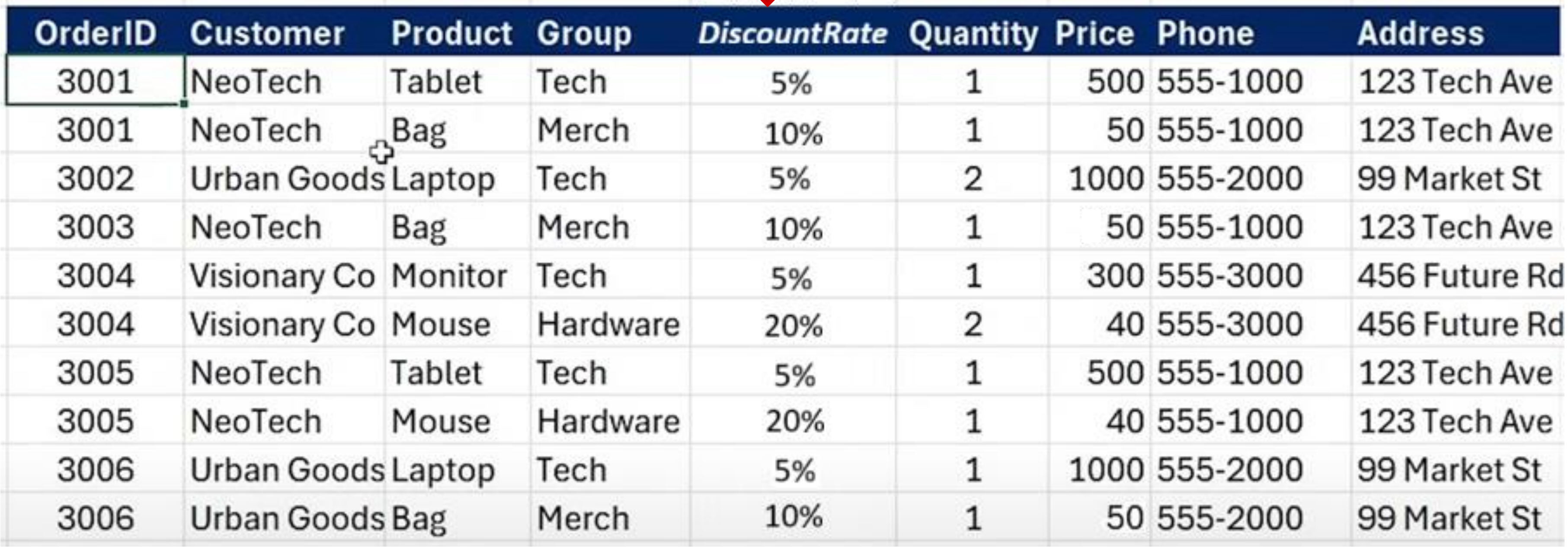
- The table is not in 1NF.
- Each row is unique.
- But some cells hold multiple values.

Practical Example

Is the table in 1NF?



OrderID	Customer	Products	Group	DiscountRate	Quantities	Prices	Phone	Address
3001	NeoTech	Tablet, Bag	Tech, Merch	5%, 10%	1,1	500,50	555-1000	123 Tech Ave
3002	Urban Goods	Laptop	Tech	5%	2	1000	555-2000	99 Market St
3003	NeoTech	Bag	Merch	10%	1	50	555-1000	123 Tech Ave
3004	Visionary Co	Monitor, Mou	Tech, Hardwa	5%, 20%	1,2	300,40	555-3000	456 Future Rd
3005	NeoTech	Tablet, Mouse	Tech, Hardwa	5%, 20%	1,1	500,40	555-1000	123 Tech Ave
3006	Urban Goods	Laptop, Bag	Tech, Merch	5%, 10%	1,1	1000,50	555-2000	99 Market St

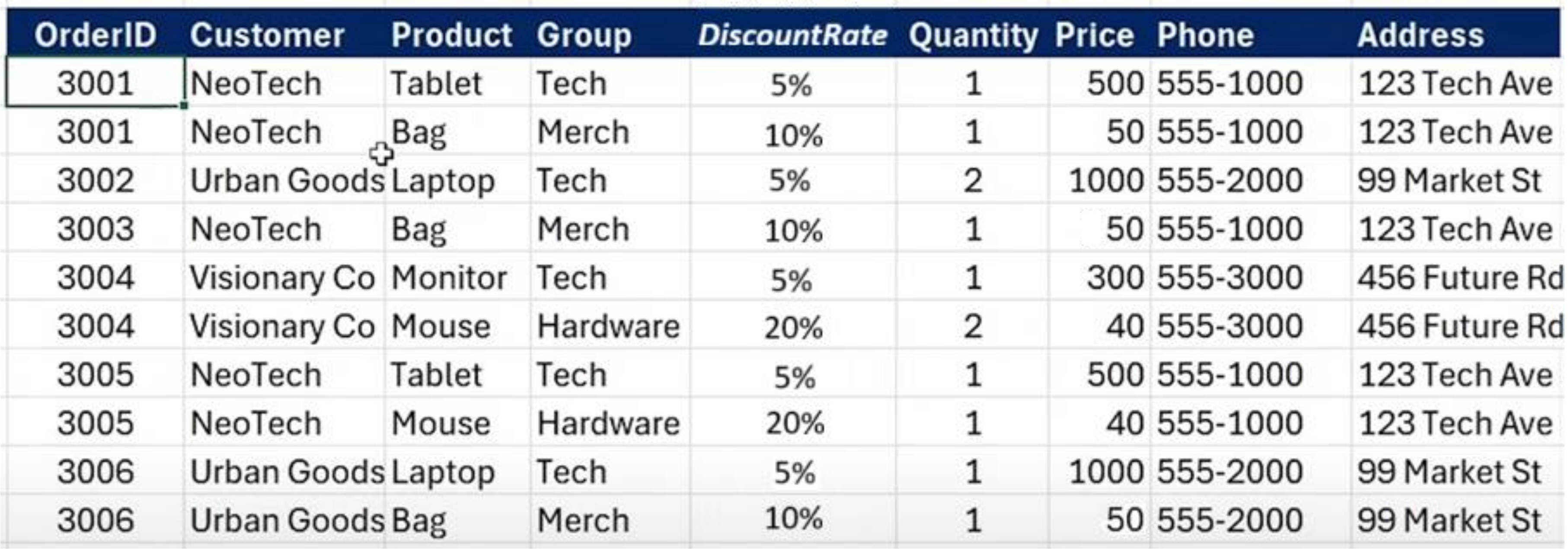


OrderID	Customer	Product	Group	DiscountRate	Quantity	Price	Phone	Address
3001	NeoTech	Tablet	Tech	5%	1	500	555-1000	123 Tech Ave
3001	NeoTech	Bag	Merch	10%	1	50	555-1000	123 Tech Ave
3002	Urban Goods	Laptop	Tech	5%	2	1000	555-2000	99 Market St
3003	NeoTech	Bag	Merch	10%	1	50	555-1000	123 Tech Ave
3004	Visionary Co	Monitor	Tech	5%	1	300	555-3000	456 Future Rd
3004	Visionary Co	Mouse	Hardware	20%	2	40	555-3000	456 Future Rd
3005	NeoTech	Tablet	Tech	5%	1	500	555-1000	123 Tech Ave
3005	NeoTech	Mouse	Hardware	20%	1	40	555-1000	123 Tech Ave
3006	Urban Goods	Laptop	Tech	5%	1	1000	555-2000	99 Market St
3006	Urban Goods	Bag	Merch	10%	1	50	555-2000	99 Market St

Now, every cell stores only one atomic value.

Practical Example

Is the table in 1NF?

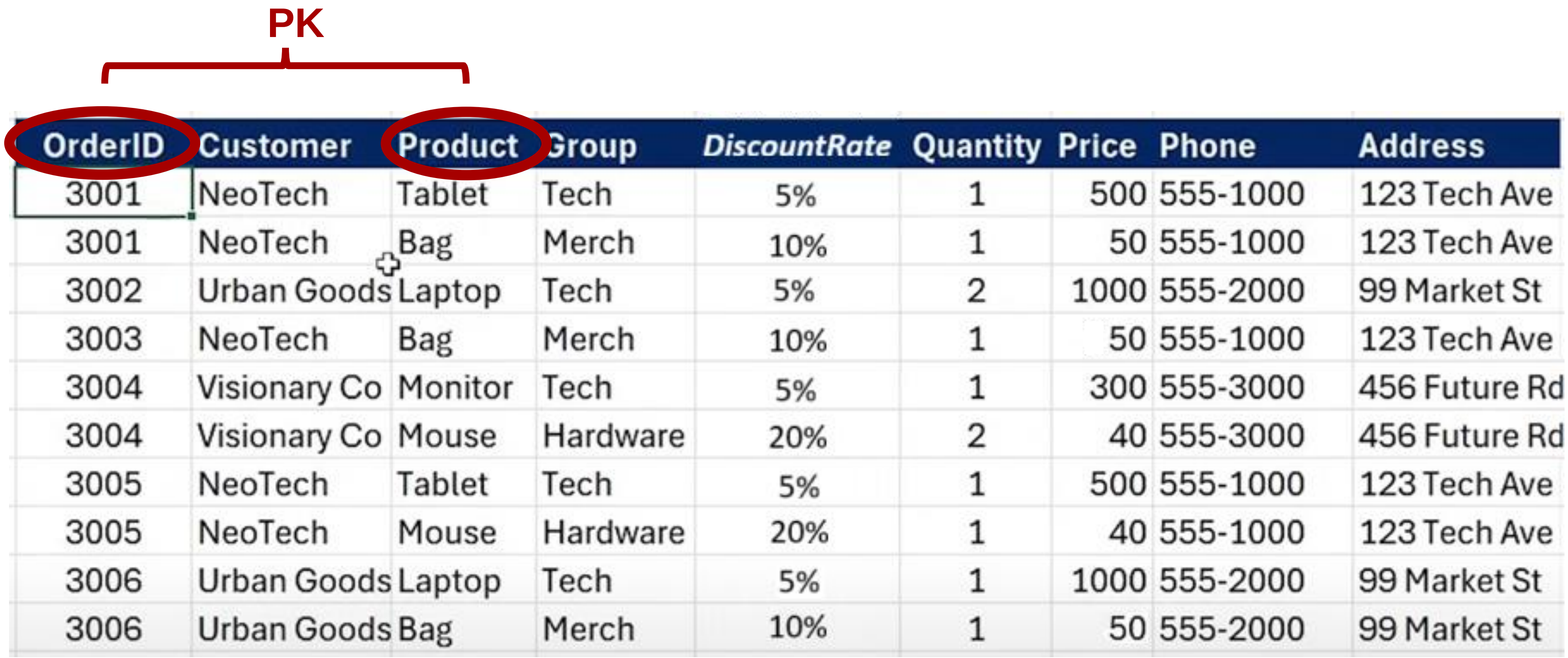


OrderID	Customer	Product	Group	*DiscountRate*	Quantity	Price	Phone	Address
3001	NeoTech	Tablet	Tech	5%	1	500	555-1000	123 Tech Ave
3001	NeoTech	Bag	Merch	10%	1	50	555-1000	123 Tech Ave
3002	Urban Goods	Laptop	Tech	5%	2	1000	555-2000	99 Market St
3003	NeoTech	Bag	Merch	10%	1	50	555-1000	123 Tech Ave
3004	Visionary Co	Monitor	Tech	5%	1	300	555-3000	456 Future Rd
3004	Visionary Co	Mouse	Hardware	20%	2	40	555-3000	456 Future Rd
3005	NeoTech	Tablet	Tech	5%	1	500	555-1000	123 Tech Ave
3005	NeoTech	Mouse	Hardware	20%	1	40	555-1000	123 Tech Ave
3006	Urban Goods	Laptop	Tech	5%	1	1000	555-2000	99 Market St
3006	Urban Goods	Bag	Merch	10%	1	50	555-2000	99 Market St

- Problem: Primary key violation!!
- Solution: Consider a composite key (OrderID, Product)

Practical Example

Is the table in 1NF?



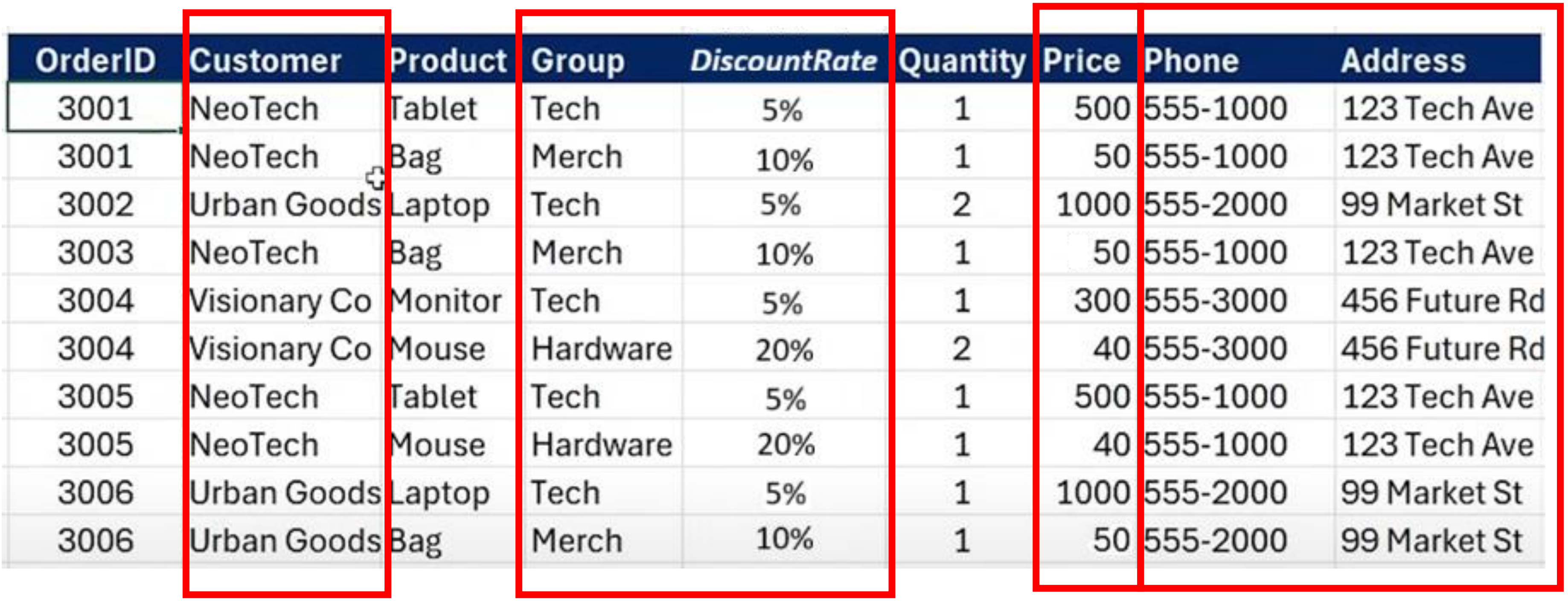
PK

OrderID	Customer	Product	Group	DiscountRate	Quantity	Price	Phone	Address
3001	NeoTech	Tablet	Tech	5%	1	500	555-1000	123 Tech Ave
3001	NeoTech	Bag	Merch	10%	1	50	555-1000	123 Tech Ave
3002	Urban Goods	Laptop	Tech	5%	2	1000	555-2000	99 Market St
3003	NeoTech	Bag	Merch	10%	1	50	555-1000	123 Tech Ave
3004	Visionary Co	Monitor	Tech	5%	1	300	555-3000	456 Future Rd
3004	Visionary Co	Mouse	Hardware	20%	2	40	555-3000	456 Future Rd
3005	NeoTech	Tablet	Tech	5%	1	500	555-1000	123 Tech Ave
3005	NeoTech	Mouse	Hardware	20%	1	40	555-1000	123 Tech Ave
3006	Urban Goods	Laptop	Tech	5%	1	1000	555-2000	99 Market St
3006	Urban Goods	Bag	Merch	10%	1	50	555-2000	99 Market St

Now the table is in 1NF.

Practical Example

Is the table in 2NF?



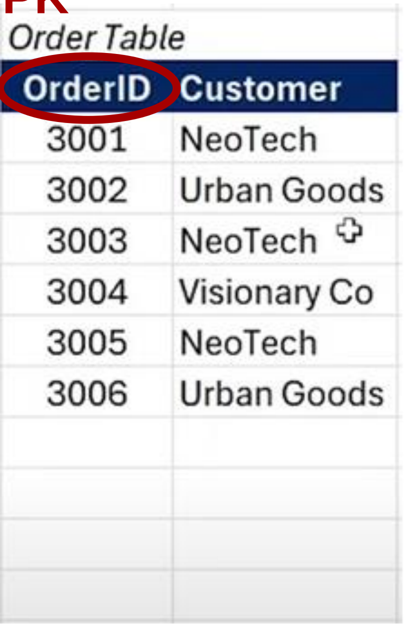
OrderID	Customer	Product	Group	DiscountRate	Quantity	Price	Phone	Address
3001	NeoTech	Tablet	Tech	5%	1	500	555-1000	123 Tech Ave
3001	NeoTech	Bag	Merch	10%	1	50	555-1000	123 Tech Ave
3002	Urban Goods	Laptop	Tech	5%	2	1000	555-2000	99 Market St
3003	NeoTech	Bag	Merch	10%	1	50	555-1000	123 Tech Ave
3004	Visionary Co	Monitor	Tech	5%	1	300	555-3000	456 Future Rd
3004	Visionary Co	Mouse	Hardware	20%	2	40	555-3000	456 Future Rd
3005	NeoTech	Tablet	Tech	5%	1	500	555-1000	123 Tech Ave
3005	NeoTech	Mouse	Hardware	20%	1	40	555-1000	123 Tech Ave
3006	Urban Goods	Laptop	Tech	5%	1	1000	555-2000	99 Market St
3006	Urban Goods	Bag	Merch	10%	1	50	555-2000	99 Market St

- The table is not in 2NF.
- Problem: Customer, Group, DiscountRate, Price, Phone and Address do not depend on the whole primary key (OrderID, Product)

Practical Example

Is the table in 2NF?

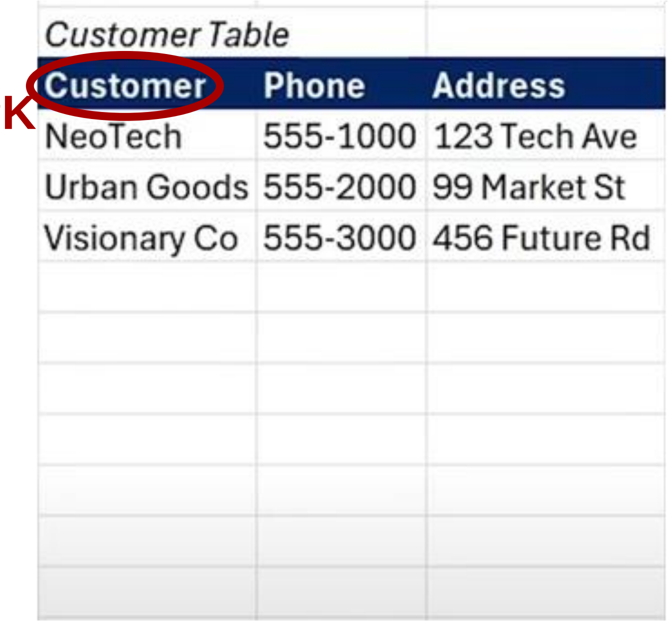
PK



Order Table

OrderID	Customer
3001	NeoTech
3002	Urban Goods
3003	NeoTech
3004	Visionary Co
3005	NeoTech
3006	Urban Goods

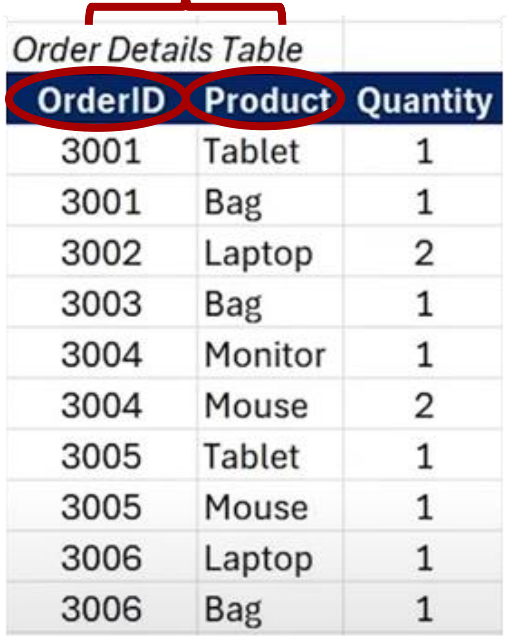
PK



Customer Table

Customer	Phone	Address
NeoTech	555-1000	123 Tech Ave
Urban Goods	555-2000	99 Market St
Visionary Co	555-3000	456 Future Rd

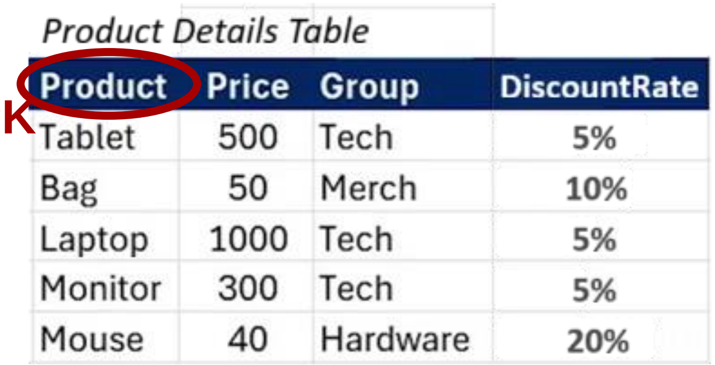
PK



Order Details Table

OrderID	Product	Quantity
3001	Tablet	1
3001	Bag	1
3002	Laptop	2
3003	Bag	1
3004	Monitor	1
3004	Mouse	2
3005	Tablet	1
3005	Mouse	1
3006	Laptop	1
3006	Bag	1

PK



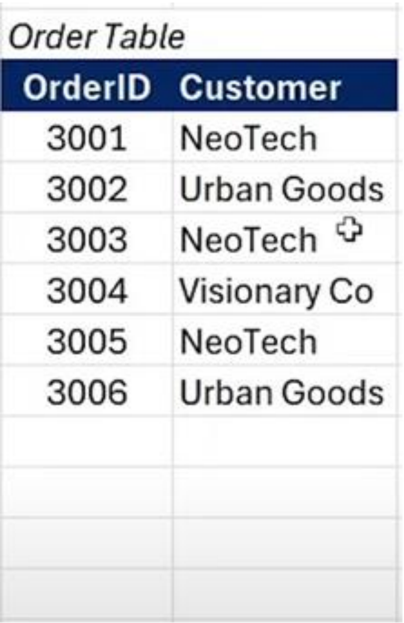
Product Details Table

Product	Price	Group	DiscountRate
Tablet	500	Tech	5%
Bag	50	Merch	10%
Laptop	1000	Tech	5%
Monitor	300	Tech	5%
Mouse	40	Hardware	20%

- Now all non-key columns are fully dependent on the entire primary key of each table
- In addition, the tables are already in 1NF
- Thus, the tables are in 2NF

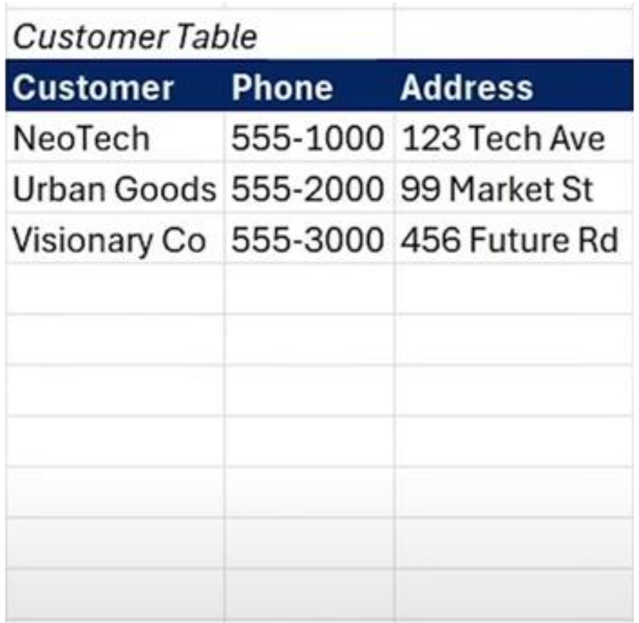
Practical Example

Is the table in 3NF?



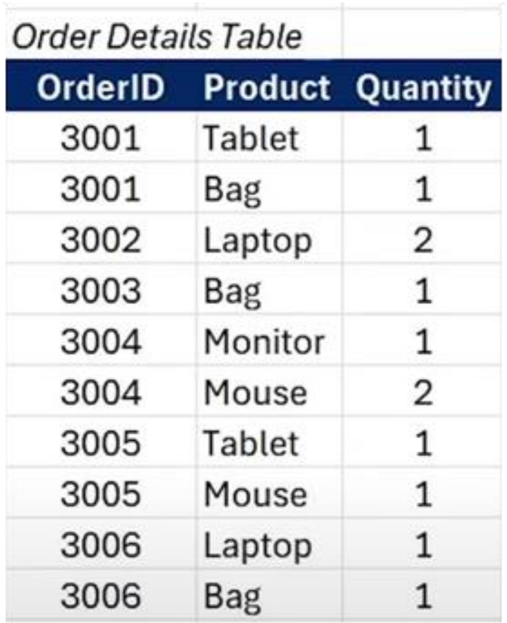
Order Table

OrderID	Customer
3001	NeoTech
3002	Urban Goods
3003	NeoTech
3004	Visionary Co
3005	NeoTech
3006	Urban Goods



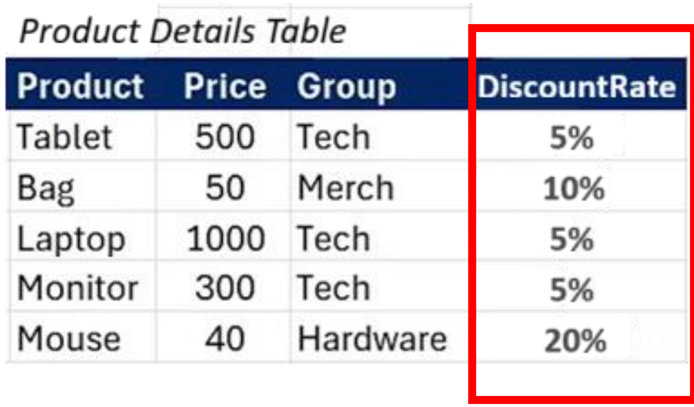
Customer Table

Customer	Phone	Address
NeoTech	555-1000	123 Tech Ave
Urban Goods	555-2000	99 Market St
Visionary Co	555-3000	456 Future Rd



Order Details Table

OrderID	Product	Quantity
3001	Tablet	1
3001	Bag	1
3002	Laptop	2
3003	Bag	1
3004	Monitor	1
3004	Mouse	2
3005	Tablet	1
3005	Mouse	1
3006	Laptop	1
3006	Bag	1



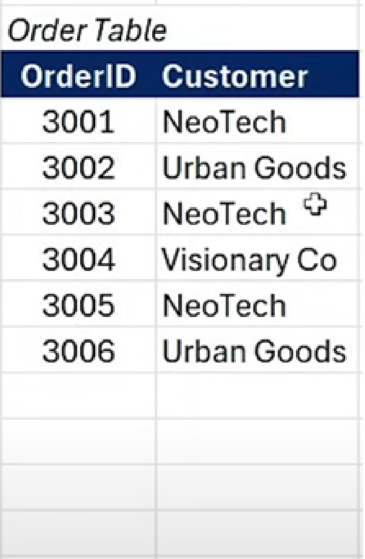
Product Details Table

Product	Price	Group	DiscountRate
Tablet	500	Tech	5%
Bag	50	Merch	10%
Laptop	1000	Tech	5%
Monitor	300	Tech	5%
Mouse	40	Hardware	20%

- Problem: The column ‘DiscountRate’ depends on a non-key column ‘Group’.

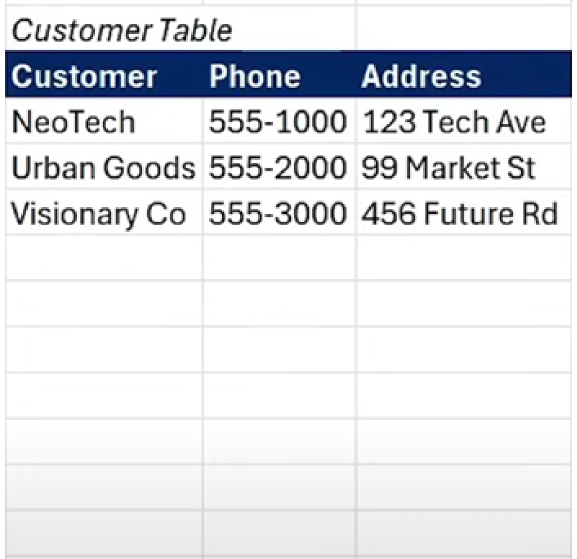
Practical Example

Is the table in 3NF?



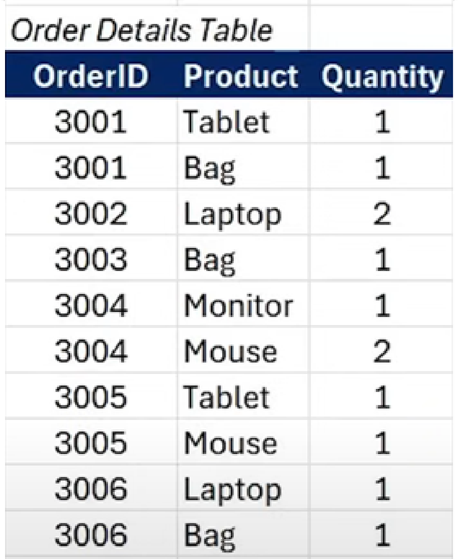
Order Table

OrderID	Customer
3001	NeoTech
3002	Urban Goods
3003	NeoTech
3004	Visionary Co
3005	NeoTech
3006	Urban Goods



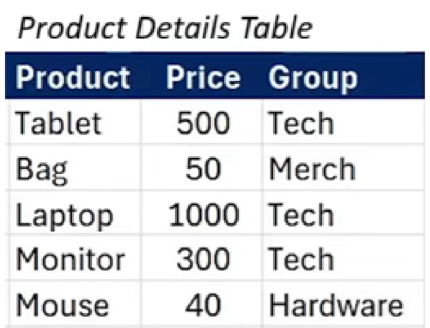
Customer Table

Customer	Phone	Address
NeoTech	555-1000	123 Tech Ave
Urban Goods	555-2000	99 Market St
Visionary Co	555-3000	456 Future Rd



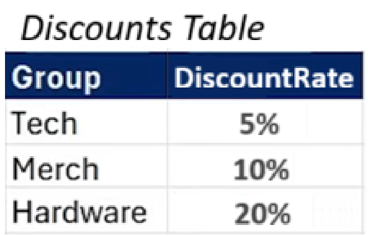
Order Details Table

OrderID	Product	Quantity
3001	Tablet	1
3001	Bag	1
3002	Laptop	2
3003	Bag	1
3004	Monitor	1
3004	Mouse	2
3005	Tablet	1
3005	Mouse	1
3006	Laptop	1
3006	Bag	1



Product Details Table

Product	Price	Group
Tablet	500	Tech
Bag	50	Merch
Laptop	1000	Tech
Monitor	300	Tech
Mouse	40	Hardware



Discounts Table

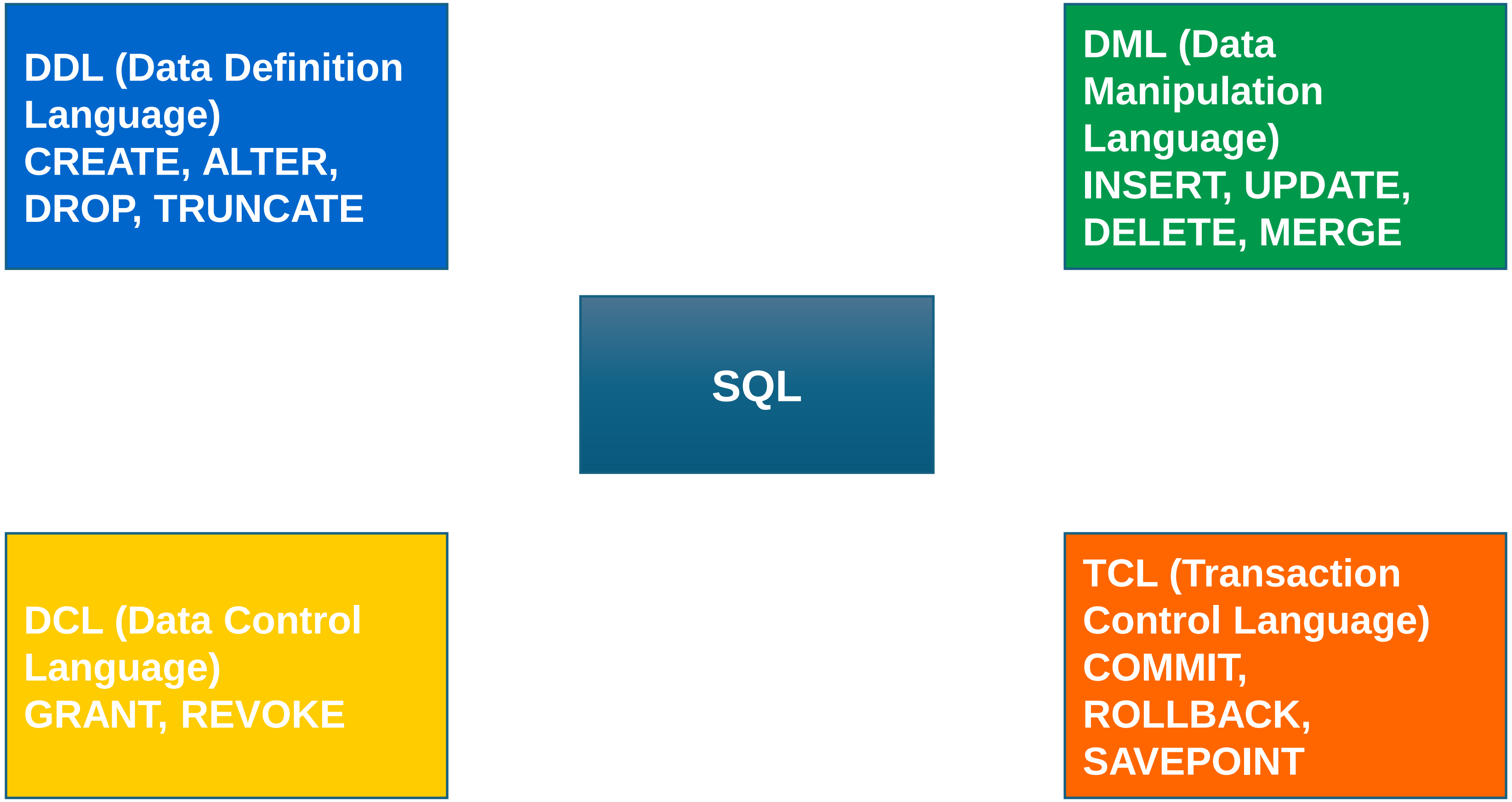
Group	DiscountRate
Tech	5%
Merch	10%
Hardware	20%

- All non-key columns depend only on the primary key (no transitive dependencies).
- The tables now are in 3NF form.

Back to Basics: SQL Commands Overview

- SQL Command Types: DDL, DML, DCL, TCL

SQL (Structured Query Language) is the standard language for data interaction in Relational Database Management Systems (RDBMS).



DDL (Data Definition Language)
CREATE, ALTER, DROP, TRUNCATE

DML (Data Manipulation Language)
INSERT, UPDATE, DELETE, MERGE

SQL

DCL (Data Control Language)
GRANT, REVOKE

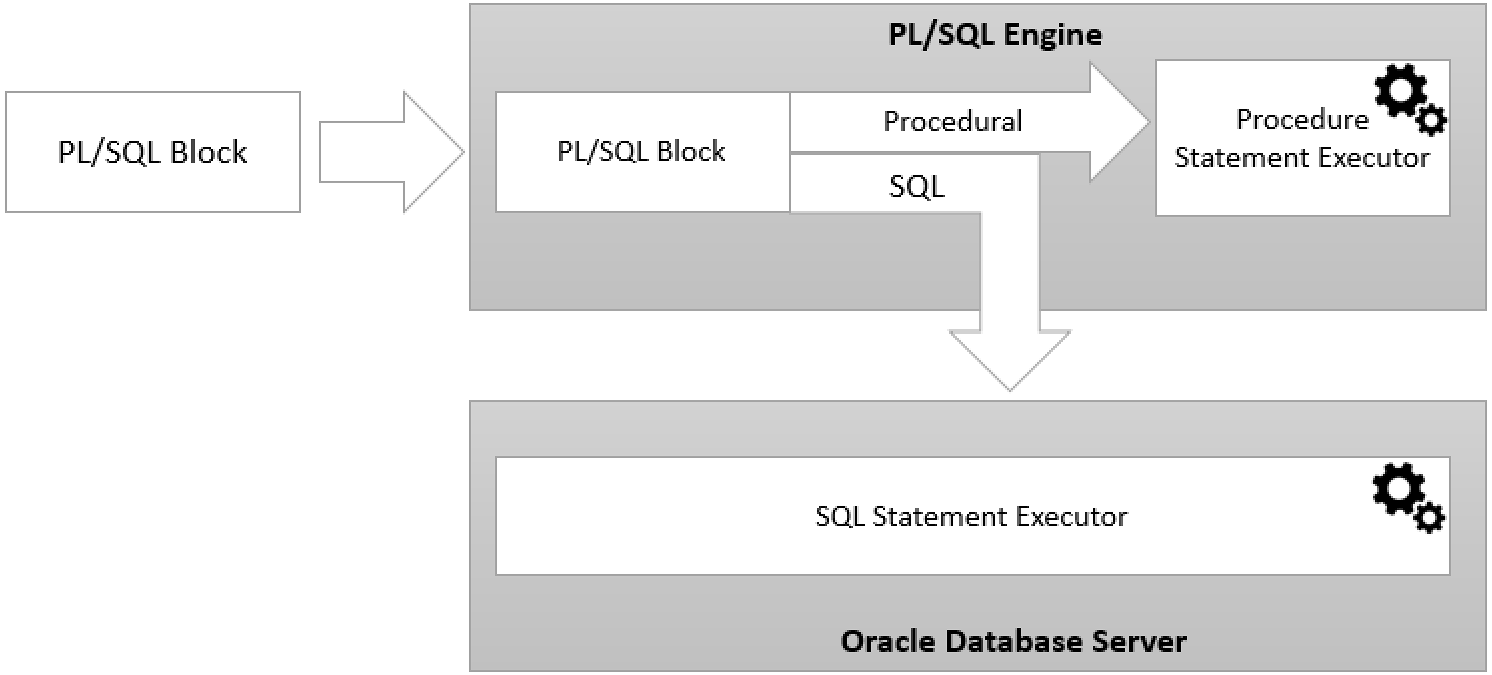
TCL (Transaction Control Language)
COMMIT, ROLLBACK, SAVEPOINT

PL/SQL

- What is PL/SQL?
  - PL/SQL stands for Procedural Language/Structured Query Language. PL/SQL is an extension to the SQL language, specifically designed for the Oracle Database.
  - While SQL is excellent for interacting with data in a relational database management system (RDBMS), it is not designed to make complex programs.
  - PL/SQL combines the power of a classical procedural programming language (loops, conditions, variables) with the data manipulation capabilities of SQL.

PL/SQL

- PL/SQL architecture



- The PL/SQL engine compiles PL/SQL code into bytecode and executes the executable code. It can only be installed in an Oracle Database server or an application development tool like Oracle Forms.

- Once you submit a PL/SQL block to the Oracle Database server, the PL/SQL engine collaborates with the SQL engine to compile and execute the code.

- PL/SQL engine runs the procedural elements while the SQL engine processes the SQL statements.

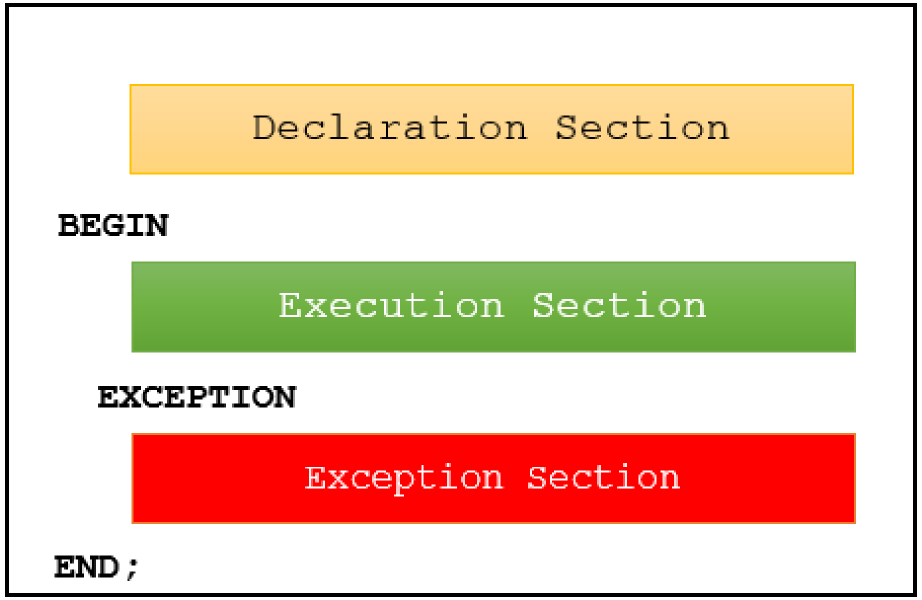
PL/SQL

- Block overview

  - PL/SQL is a block-structured language whose code is organized into blocks.
  - A block consists of three sections:
    1. Declaration Section
    2. Executable Section
    3. Exception-handling Section
  - The executable section is mandatory, while the declaration and exception-handling sections are optional.

PL/SQL

- Block overview



```
    Declaration Section
BEGIN
    Execution Section
  EXCEPTION
    Exception Section
END;
```

1) Declaration section

In the declaration section, you declare variables, allocate memory for cursors, and define data types.

2) Executable section

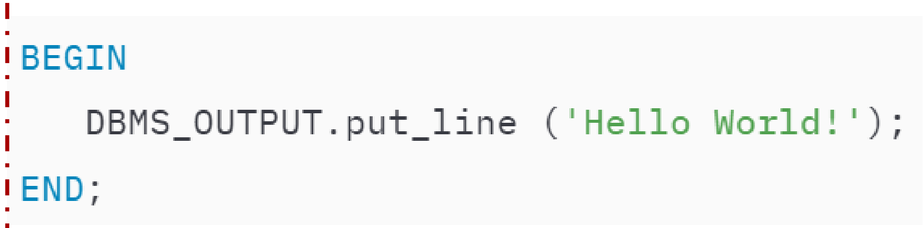
An executable section starts with the keyword BEGIN and ends with the keyword END. The executable section must have a least one executable statement, even if it is a NULL statement that does nothing.

3) Exception-handling section

An exception-handling section that starts with the keyword EXCEPTION. In the exception-handling section, you catch and handle exceptions raised by the code in the execution section.

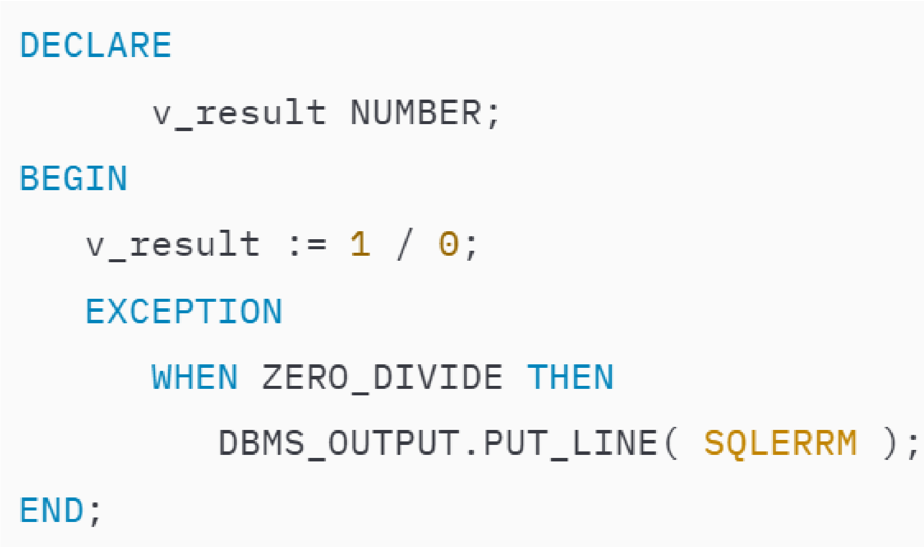
PL/SQL

- Block examples



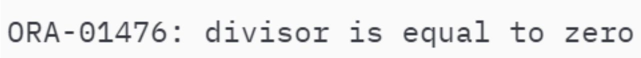
```
BEGIN
   DBMS_OUTPUT.put_line ('Hello World!');
END;
```

- The example on the right shows a simple PL/SQL block with one executable section.
- The executable section calls the **DMBS_OUTPUT.PUT_LINE** procedure to display the "Hello World!" message on the screen.
- **put_line** is a procedure from the **DBMS_OUTPUT** package that displays output on the screen.



```
DECLARE
      v_result NUMBER;
BEGIN
   v_result := 1 / 0;
   EXCEPTION
      WHEN ZERO_DIVIDE THEN
         DBMS_OUTPUT.PUT_LINE( SQLERRM );
END;
```

- The block example on the right adds an exception-handling section that catches ZERO_DIVIDE exception raised in the executable section and displays an error message.
- The error message is:



```
ORA-01476: divisor is equal to zero
```

PL/SQL

- Data Types
  - Each value in PL/SQL such as a constant, variable and parameter has a data type that determines the storage format, valid values, and allowed operations.
  - PL/SQL has two kinds of data types:
    - Scalar
    - Composite
  - **Scalar** types are data types that store single values, such as numbers, Boolean, characters, and datetime.
  - In contrast, **composite** types are data types that store multiple values, such as records and collections.

PL/SQL

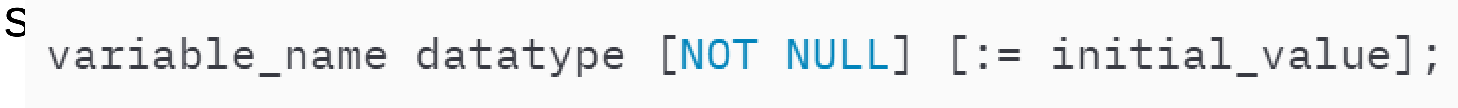
- Data Types
  - Commonly used scalar types in PL/SQL include:

Type	Description	Example
NUMBER	Numeric values (integer or decimal)	v_salary NUMBER;
VARCHAR2	Variable-length string	v_name VARCHAR2(50);
CHAR	Fixed-length string	v_code CHAR(5);
DATE	Date and time values	v_hire DATE;
BOOLEAN	TRUE / FALSE values	v_active BOOLEAN;

  - Other specialized scalar types also exist (PLS_INTEGER, NATURAL, POSITIVE, etc.)

PL/SQL

- Variables - Declaring variables
  - In PL/SQL, a variable is named storage location that stores a value of a particular data type.
  - The value of the variable may change through out the execution of the program.
  - Before using a variable, we must declare it in the declaration section of a block.
  - The s

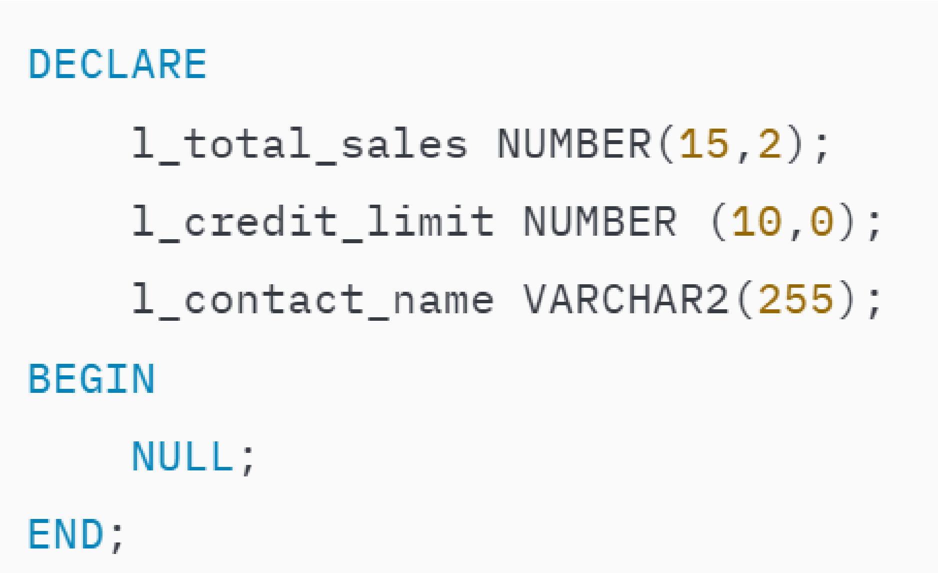


```
variable_name datatype [NOT NULL] [:= initial_value];
```

  - In this syntax:
    - First, we specify the name of the variable. The name of the variable should be as descriptive as possible, such as total_sales, credit_limit, and sales_revenue.
    - Second, we choose an appropriate data type for the variable. The data type depends on the kind of value that we want the variable to store, for example, number, character, Boolean, or datetime.

PL/SQL

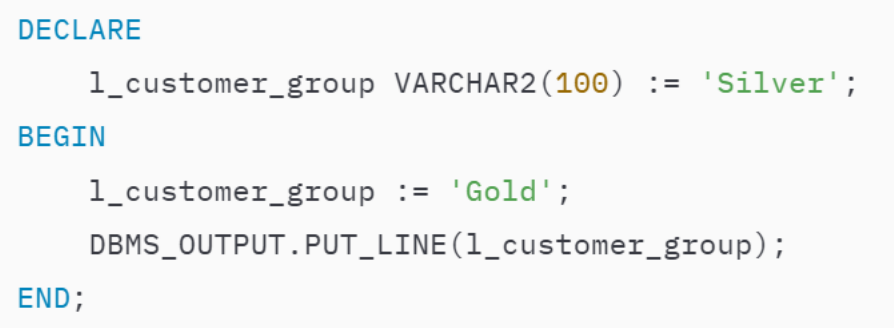
- Variables - Declaring variables
  - The following example declares three variables l_total_sales, l_credit_limit, and l_contact_name:



```
DECLARE
    l_total_sales NUMBER(15,2);
    l_credit_limit NUMBER (10,0);
    l_contact_name VARCHAR2(255);
BEGIN
    NULL;
END;
```

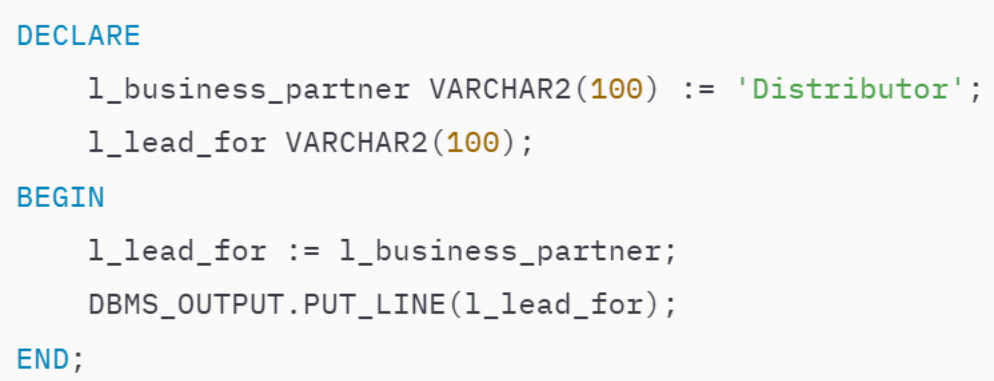
PL/SQL

- Variables - Variable assignments
  - To assign a value to a variable, we use the assignment operator (:=)
  - For example:



```
DECLARE
    l_customer_group VARCHAR2(100) := 'Silver';
BEGIN
    l_customer_group := 'Gold';
    DBMS_OUTPUT.PUT_LINE(l_customer_group);
END;
```

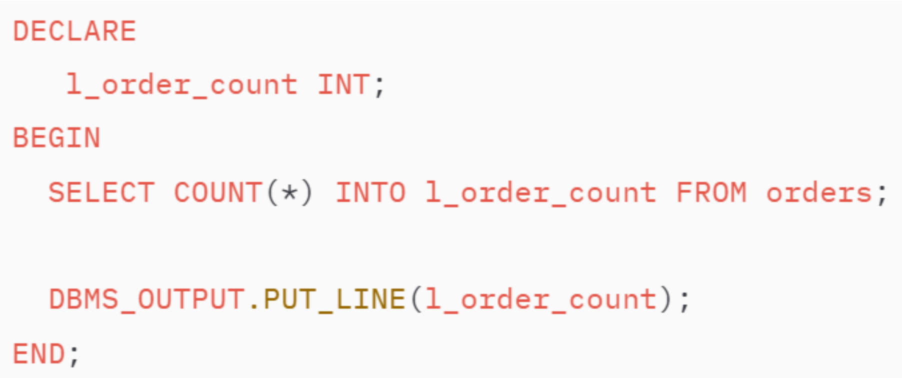
  - We can assign a value of a variable to another as shown in the following example:



```
DECLARE
    l_business_partner VARCHAR2(100) := 'Distributor';
    l_lead_for VARCHAR2(100);
BEGIN
    l_lead_for := l_business_partner;
    DBMS_OUTPUT.PUT_LINE(l_lead_for);
END;
```

PL/SQL

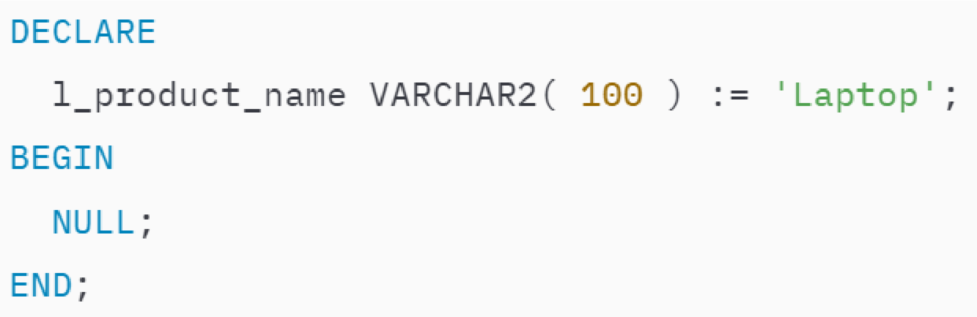
- Variables - Variable assignments
  - Additionally, we can select a value from a table and assign it to a variable using the SELECT INTO statement.
  - For example:



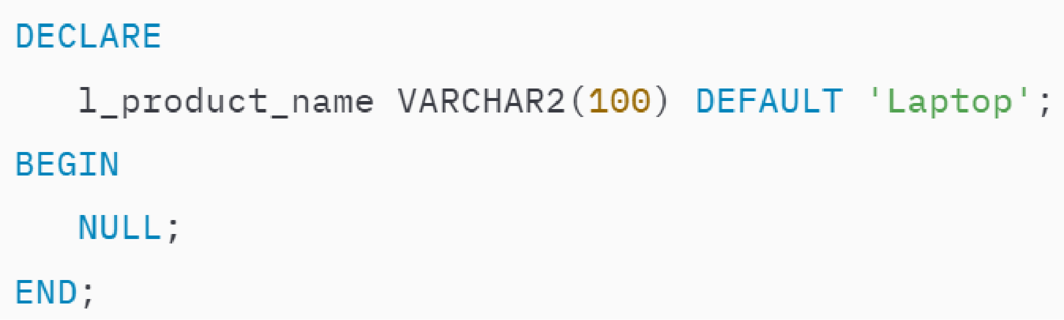
```
DECLARE
   l_order_count INT;
BEGIN
  SELECT COUNT(*) INTO l_order_count FROM orders;

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

PL/SQL

- Variables - Default values
  - PL/SQL allows us to set a default value for a variable at the declaration time.
  - To assign a default value to a variable, we use the assignment operator (:=) or the DEFAULT keyword.
  - The following examples declare a variable named l_product_name with an initial value 'Laptop':



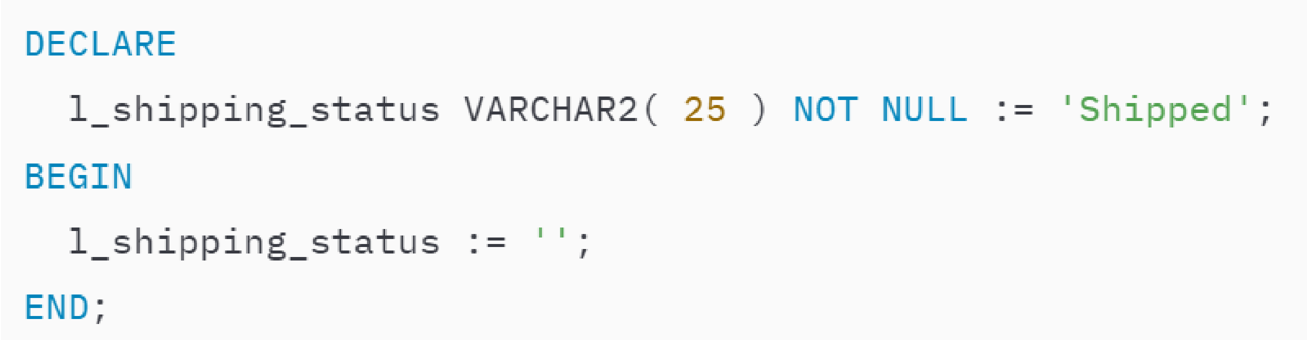
```
DECLARE
  l_product_name VARCHAR2( 100 ) := 'Laptop';
BEGIN
  NULL;
END;
```



```
DECLARE
   l_product_name VARCHAR2(100) DEFAULT 'Laptop';
BEGIN
   NULL;
END;
```

PL/SQL

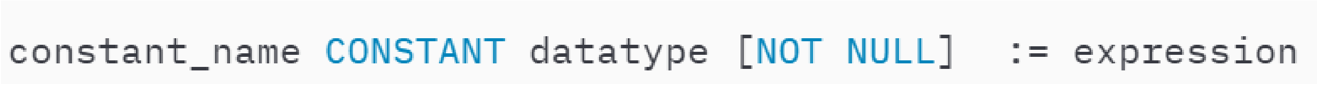
- Variables - NOT NULL constraint
  - If we impose the NOT NULL constraint on a value, then the variable cannot accept NULL. Additionally, a variable declared with the NOT NULL must be initialized with a non-null value.
  - Example:



```
DECLARE
  l_shipping_status VARCHAR2( 25 ) NOT NULL := 'Shipped';
BEGIN
  l_shipping_status := '';
END;
```

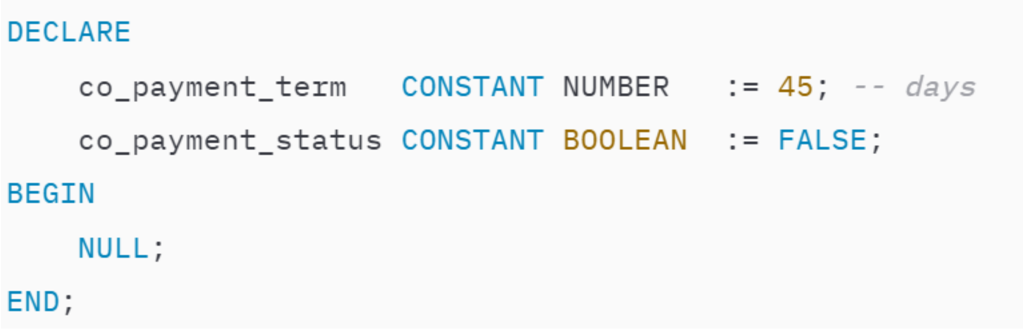
PL/SQL

- Variables – Constants
  - Unlike a variable, a constant holds a value that does not change throughout the program’s execution.
  - Constants make the code more readable.
  - To declare a constant, we specify the name, CONSTANT keyword, data type, and the default value. The following illustrates the syntax of declaring a constant:



```
constant_name CONSTANT datatype [NOT NULL]  := expression
```

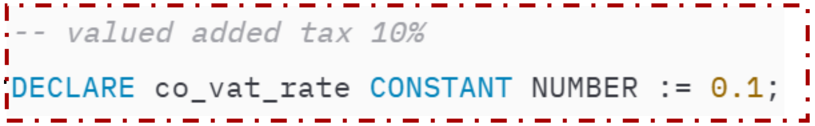
  - For example:



```
DECLARE
    co_payment_term   CONSTANT NUMBER   := 45; -- days
    co_payment_status CONSTANT BOOLEAN  := FALSE;
BEGIN
    NULL;
END;
```

PL/SQL

- Comments
  - PL/SQL comments allow to describe the purpose of a line or a block of PL/SQL code.
  - When compiling the PL/SQL code, the Oracle precompiler ignores comments. However, we should always use comments to help understand the code quickly later.
  - PL/SQL has types of comments:
    - Single-line comments
    - Multi-line comments
  - A single-line comment starts with a double hyphen (-

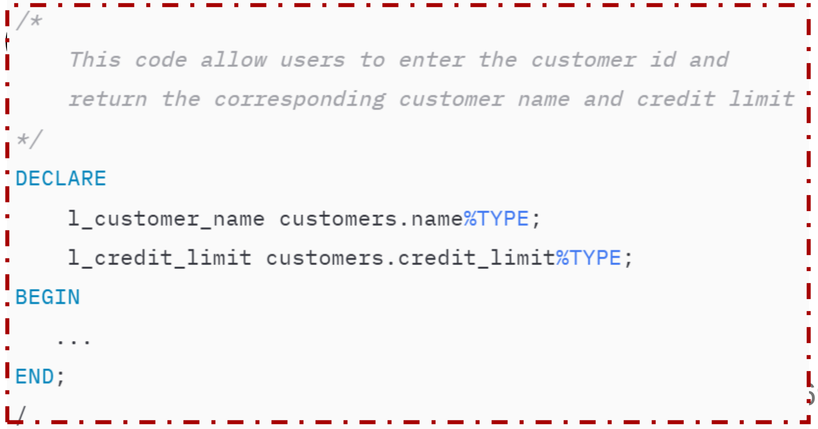


```
-- valued added tax 10%
DECLARE co_vat_rate CONSTANT NUMBER := 0.1;
```

  - A multi-line comment starts with a slash and asterisk

    and ends with an asterisk and slash (*/),

    and can span multiple lines:



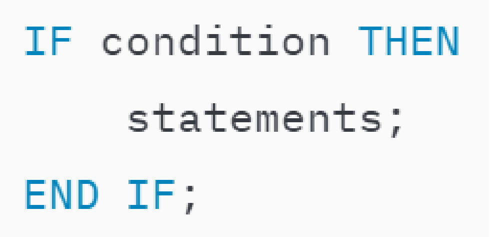
```
/*
    This code allow users to enter the customer id and
    return the corresponding customer name and credit limit
*/
DECLARE
    l_customer_name customers.name%TYPE;
    l_credit_limit customers.credit_limit%TYPE;
BEGIN
   ...
END;
/
```

PL/SQL

- IF Statement
  - The IF statement allows to either execute or skip a sequence of statements, depending on a condition. The IF statement has three forms:
    - IF THEN
    - IF THEN ELSE
    - IF THEN ELSIF

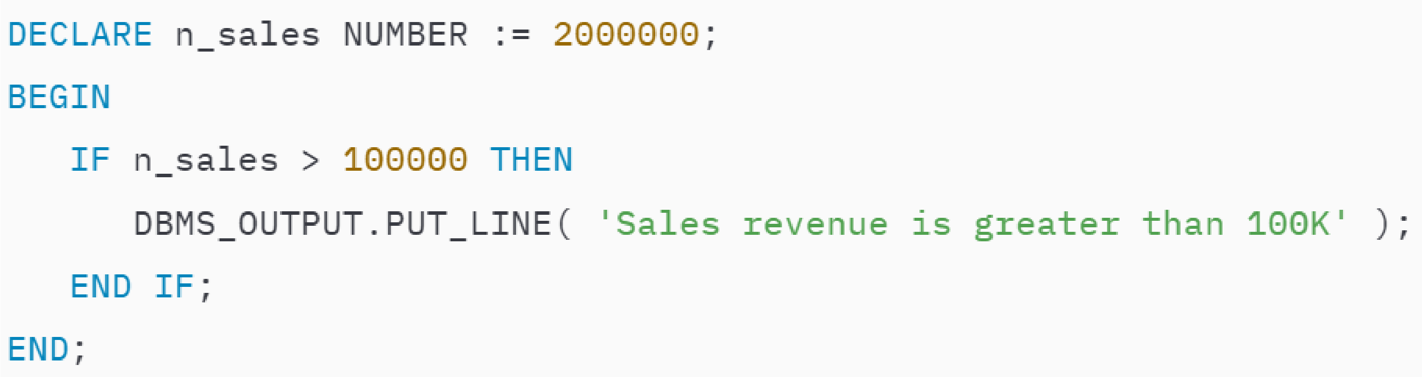
PL/SQL

- IF Statement
  - The following illustrates the syntax of the IF THEN statement:



```
IF condition THEN
    statements;
END IF;
```

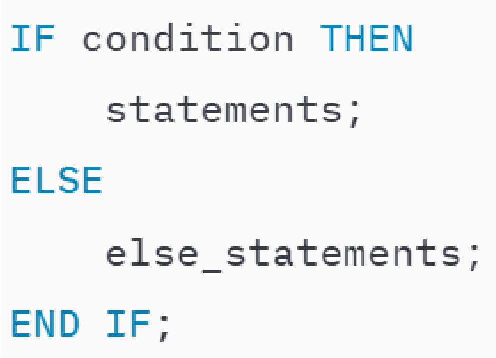
  - Example:



```
DECLARE n_sales NUMBER := 2000000;
BEGIN
   IF n_sales > 100000 THEN
      DBMS_OUTPUT.PUT_LINE( 'Sales revenue is greater than 100K' );
   END IF;
END;
```

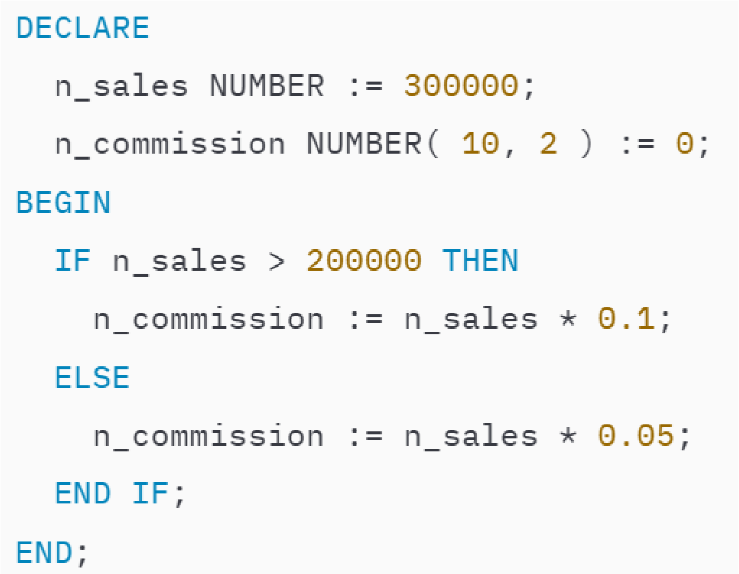
PL/SQL

- IF Statement
  - The IF THEN ELSE statement has the following structure:



```
IF condition THEN
    statements;
ELSE
    else_statements;
END IF;
```

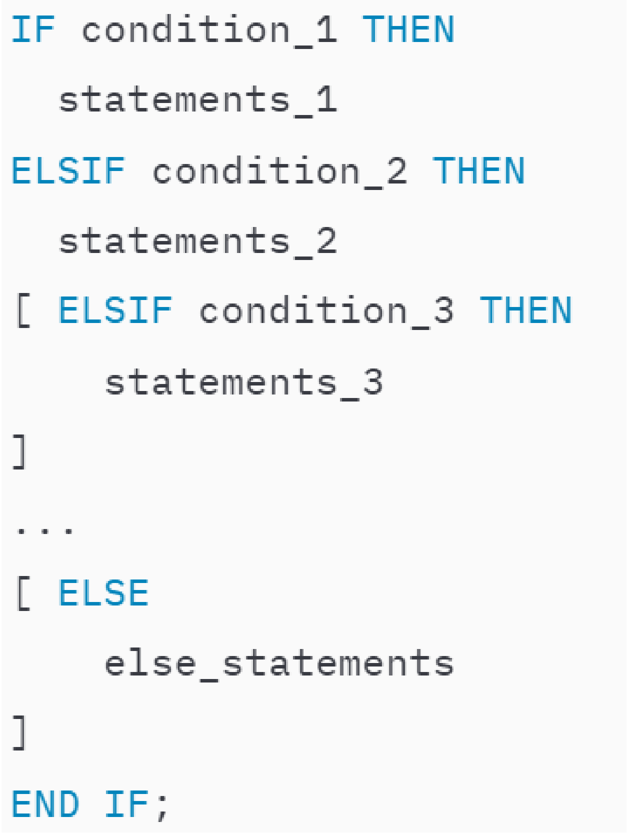
  - Example:



```
DECLARE
  n_sales NUMBER := 300000;
  n_commission NUMBER( 10, 2 ) := 0;
BEGIN
  IF n_sales > 200000 THEN
    n_commission := n_sales * 0.1;
  ELSE
    n_commission := n_sales * 0.05;
  END IF;
END;
```

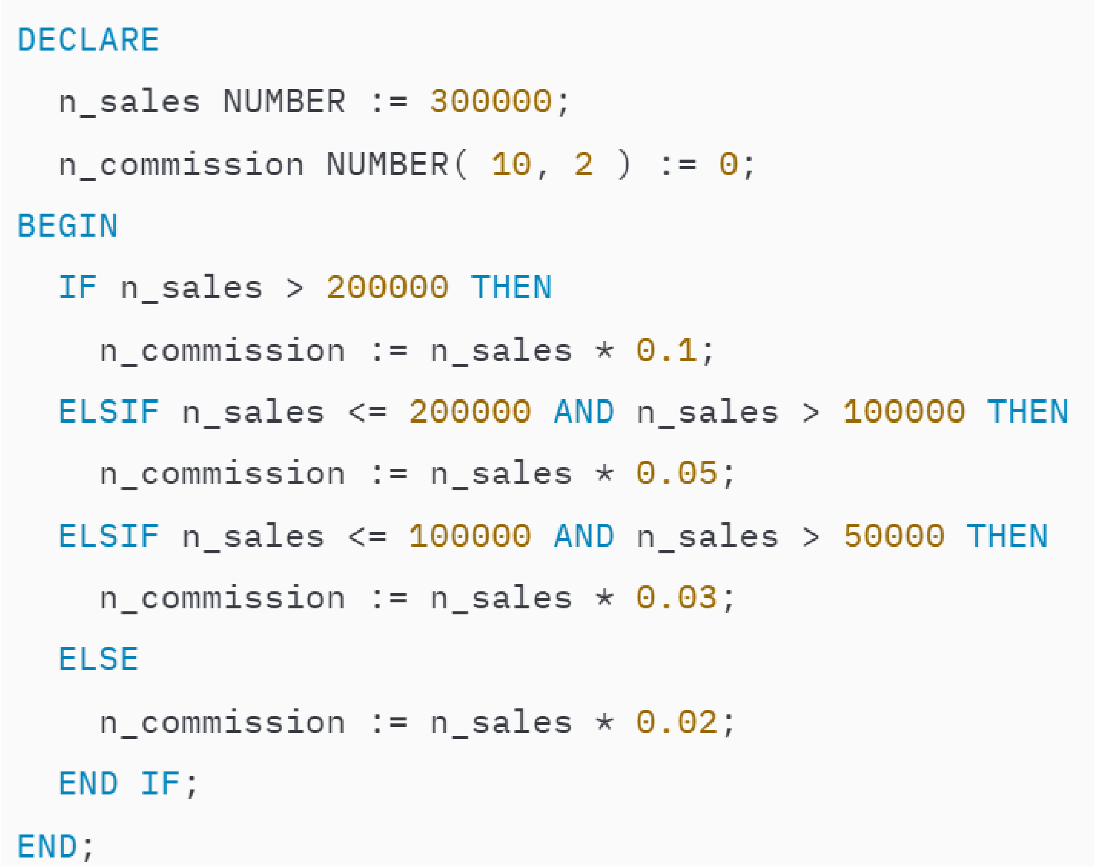
PL/SQL

- IF Statement
  - Here’s the syntax of the IF ELSIF statement:



```
IF condition_1 THEN
  statements_1
ELSIF condition_2 THEN
  statements_2
[ ELSIF condition_3 THEN
    statements_3
]
...
[ ELSE
    else_statements
]
END IF;
```

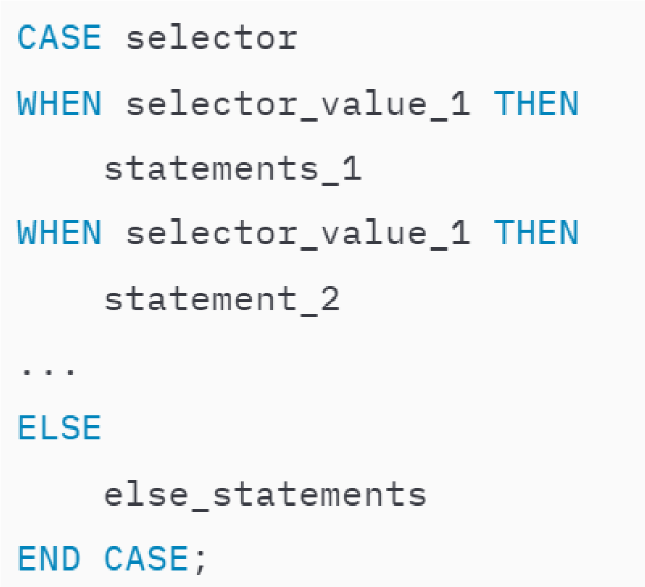
  - Example:



```
DECLARE
  n_sales NUMBER := 300000;
  n_commission NUMBER( 10, 2 ) := 0;
BEGIN
  IF n_sales > 200000 THEN
    n_commission := n_sales * 0.1;
  ELSIF n_sales <= 200000 AND n_sales > 100000 THEN
    n_commission := n_sales * 0.05;
  ELSIF n_sales <= 100000 AND n_sales > 50000 THEN
    n_commission := n_sales * 0.03;
  ELSE
    n_commission := n_sales * 0.02;
  END IF;
END;
```

PL/SQL

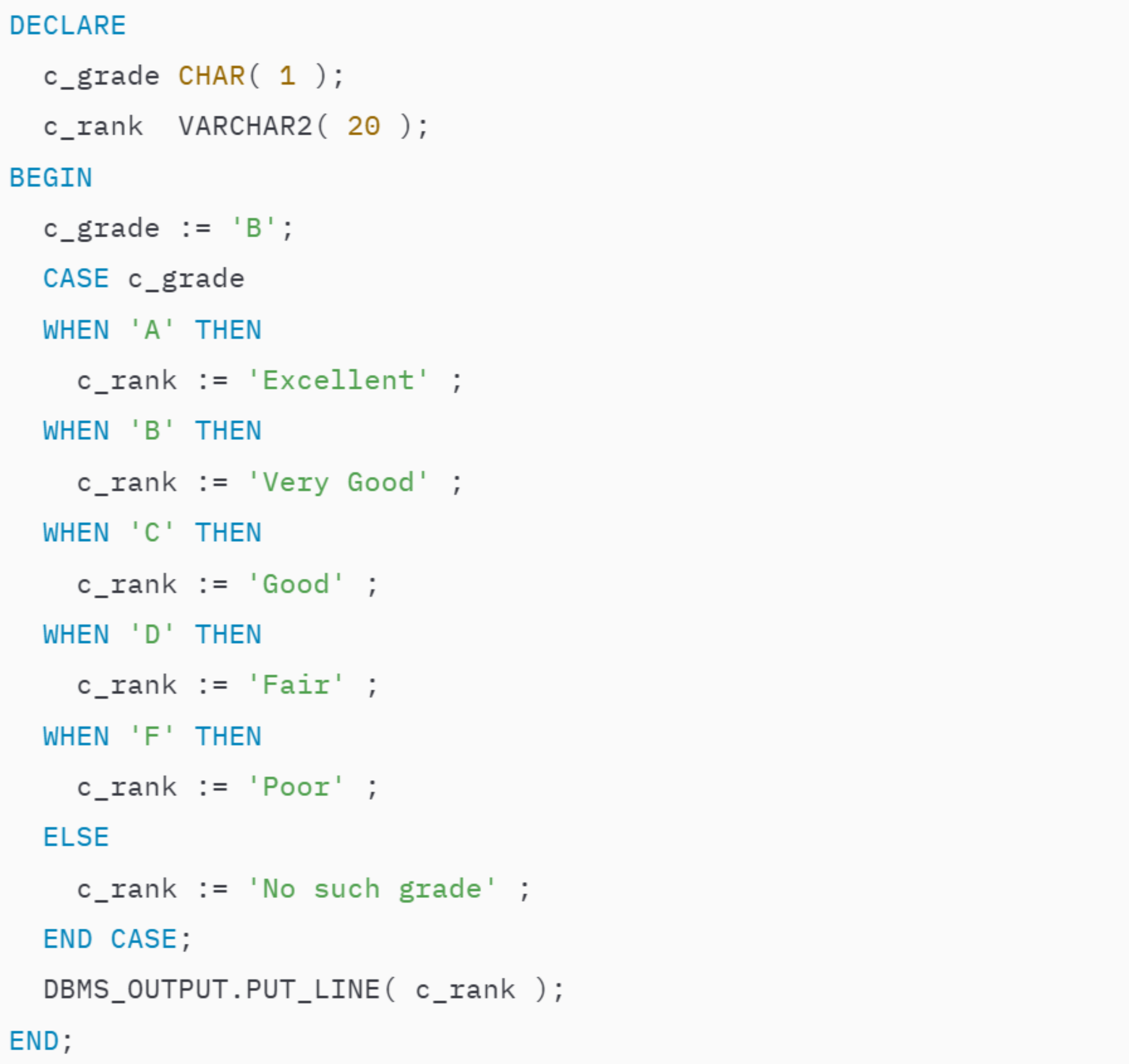
- CASE Statement
  - The CASE statement allows to choose one sequence of statements to execute out of many possible sequences.
  - A simple CASE statement evaluates a single expression and compares the result with some values.
  - The simple CASE statement has the following structure:



```
CASE selector
WHEN selector_value_1 THEN
    statements_1
WHEN selector_value_1 THEN
    statement_2
...
ELSE
    else_statements
END CASE;
```

PL/SQL

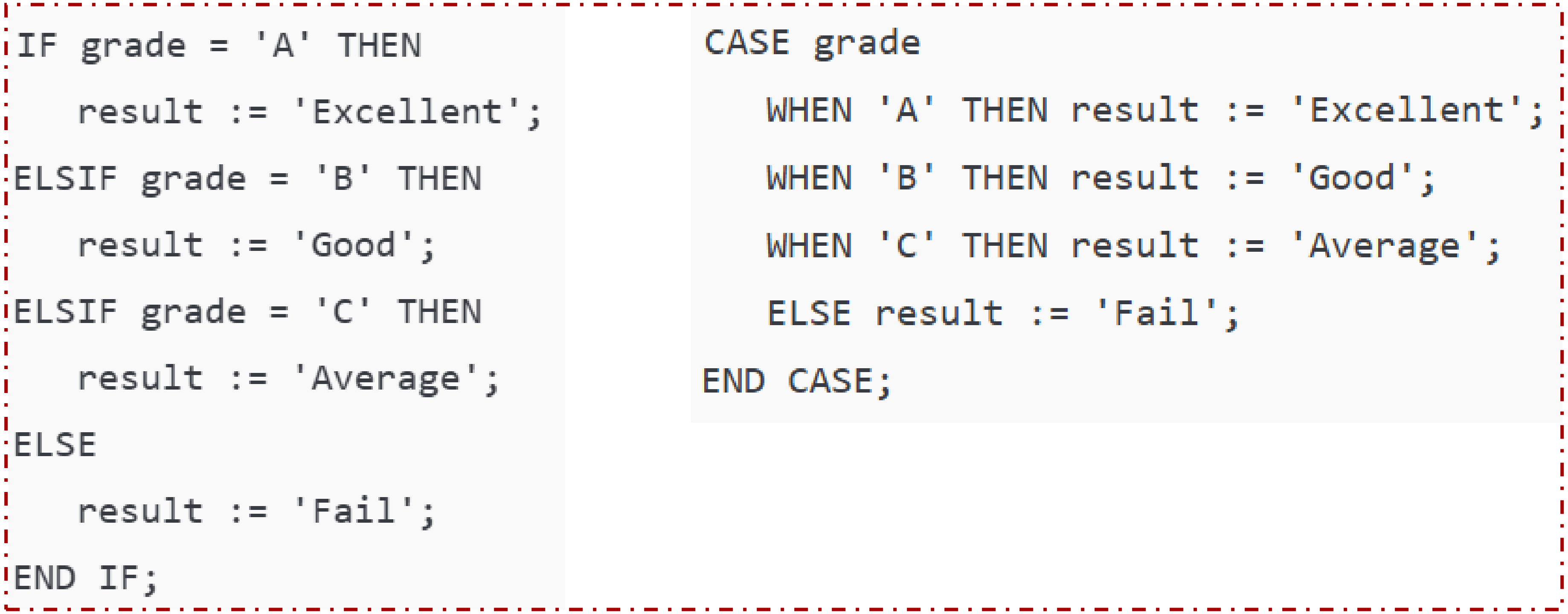
- CASE Statement
  - Example



```
DECLARE
  c_grade CHAR( 1 );
  c_rank  VARCHAR2( 20 );
BEGIN
  c_grade := 'B';
  CASE c_grade
  WHEN 'A' THEN
    c_rank := 'Excellent' ;
  WHEN 'B' THEN
    c_rank := 'Very Good' ;
  WHEN 'C' THEN
    c_rank := 'Good' ;
  WHEN 'D' THEN
    c_rank := 'Fair' ;
  WHEN 'F' THEN
    c_rank := 'Poor' ;
  ELSE
    c_rank := 'No such grade' ;
  END CASE;
  DBMS_OUTPUT.PUT_LINE( c_rank );
END;
```

PL/SQL

- IF Statement vs CASE Statement
  - It seems like IF…ELSIF can do everything CASE does. Technically, yes, but CASE has advantages in readability, conciseness, and SQL integration.



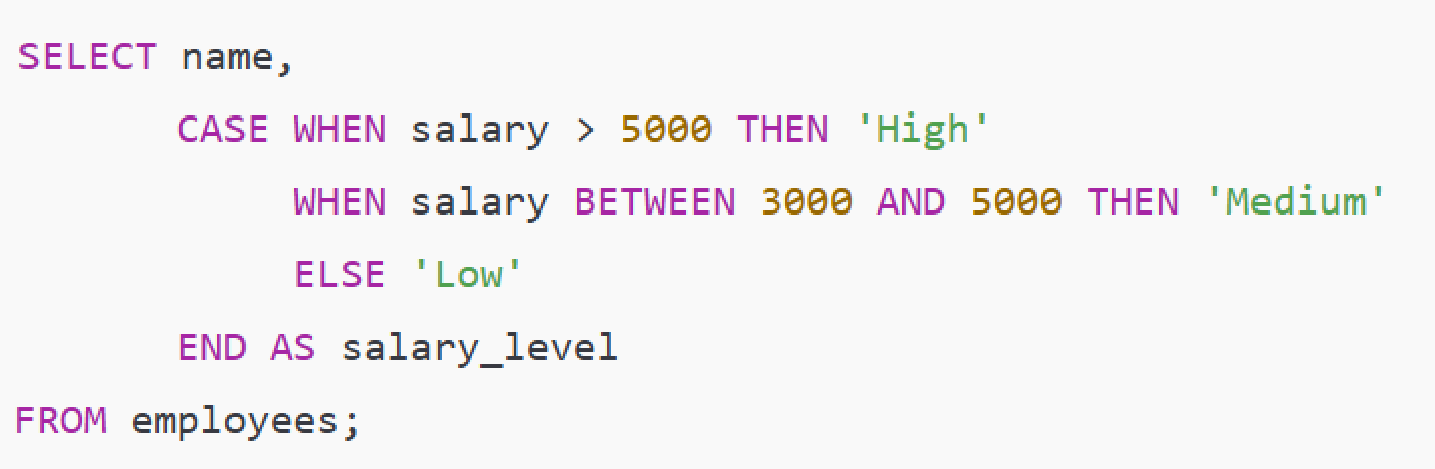
```
IF grade = 'A' THEN
   result := 'Excellent';
ELSIF grade = 'B' THEN
   result := 'Good';
ELSIF grade = 'C' THEN
   result := 'Average';
ELSE
   result := 'Fail';
END IF;
```

```
CASE grade
   WHEN 'A' THEN result := 'Excellent';
   WHEN 'B' THEN result := 'Good';
   WHEN 'C' THEN result := 'Average';
   ELSE result := 'Fail';
END CASE;
```

Example showing that CASE is more compact for multiple conditions

PL/SQL

- IF Statement vs CASE Statement
  - CASE can be embedded **inside SELECT, UPDATE, or ORDER BY**, while IF…ELSIF can only be used in PL/SQL procedural code.

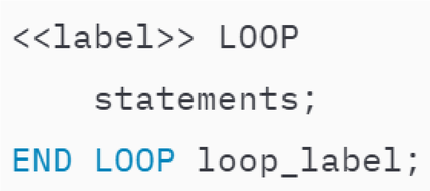


```
SELECT name,
       CASE WHEN salary > 5000 THEN 'High'
            WHEN salary BETWEEN 3000 AND 5000 THEN 'Medium'
            ELSE 'Low'
       END AS salary_level
FROM employees;
```

Example showing that CASE can be embedded inside SELECT

PL/SQL

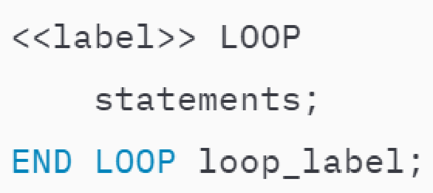
- LOOP statement
  - The LOOP statement is a control structure that repeatedly executes a code block until a specific condition is true or until we manually exit the loop.
  - Here’s the syntax of the LOOP statement:



```
<<label>> LOOP
    statements;
END LOOP loop_label;
```

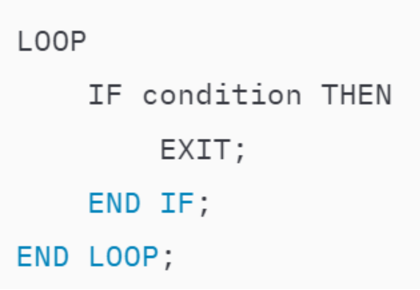
PL/SQL

- LOOP statement
  - The LOOP statement is a control structure that repeatedly executes a code block until a specific condition is true or until we manually exit the loop.
  - Here’s the syntax of the LOOP statement:



```
<<label>> LOOP
    statements;
END LOOP loop_label;
```

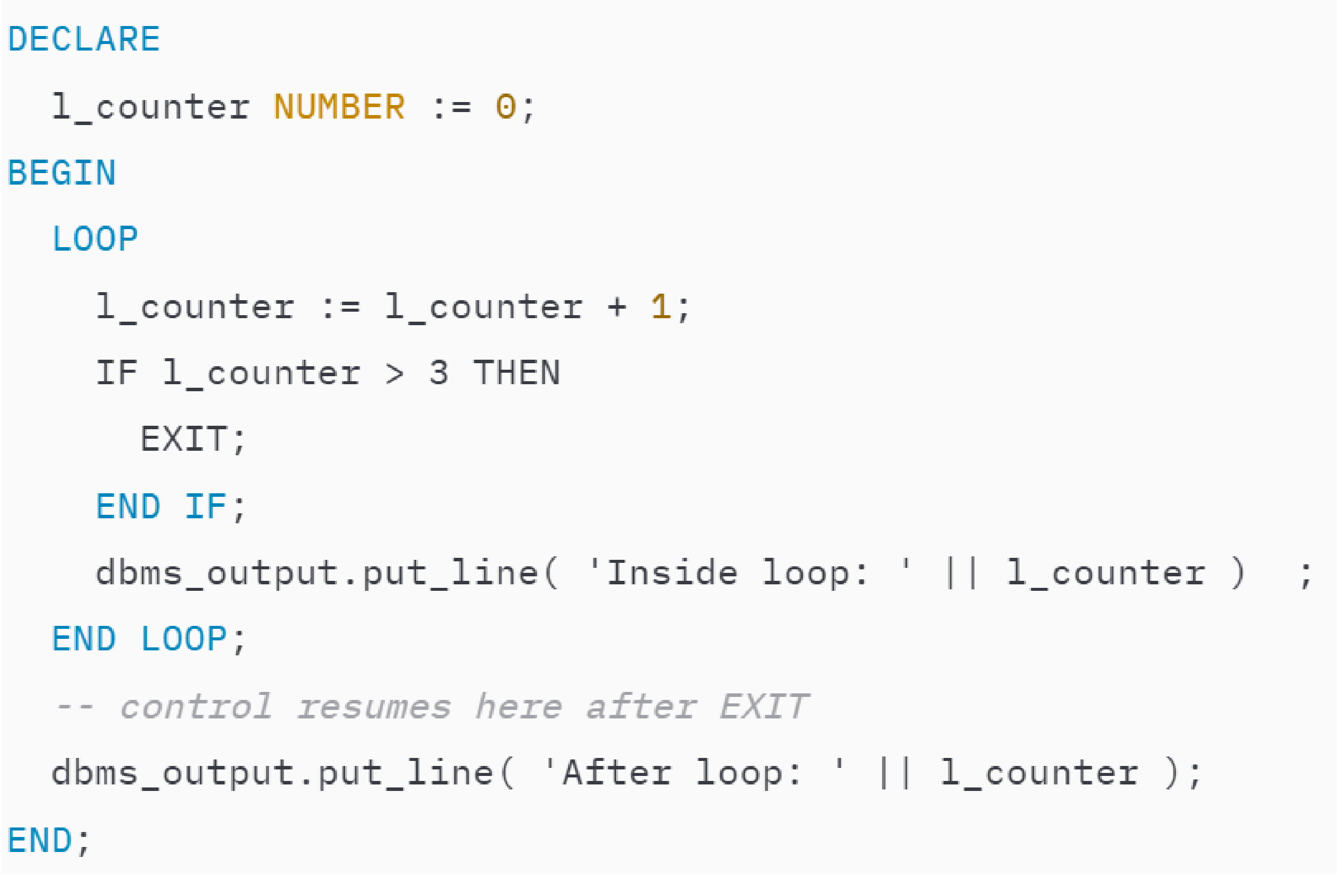
- EXIT statement
  - The EXIT statement allows to terminate the entire loop prematurely.
  - Typically, we use the EXIT statement with an IF statement to terminate a loop when a condition is true.



```
LOOP
    IF condition THEN
        EXIT;
    END IF;
END LOOP;
```

PL/SQL

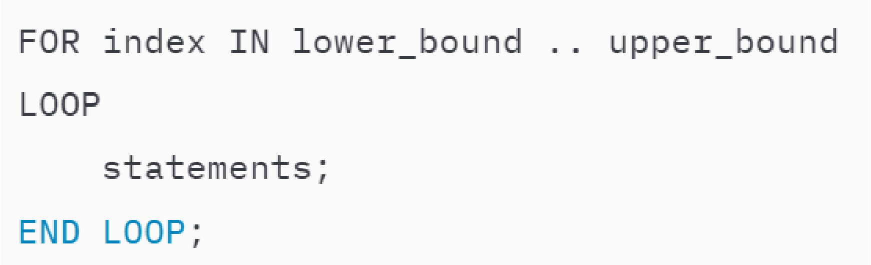
- LOOP statement
  - Example:



```
DECLARE
  l_counter NUMBER := 0;
BEGIN
  LOOP
    l_counter := l_counter + 1;
    IF l_counter > 3 THEN
      EXIT;
    END IF;
    dbms_output.put_line( 'Inside loop: ' || l_counter )  ;
  END LOOP;
  -- control resumes here after EXIT
  dbms_output.put_line( 'After loop: ' || l_counter );
END;
```

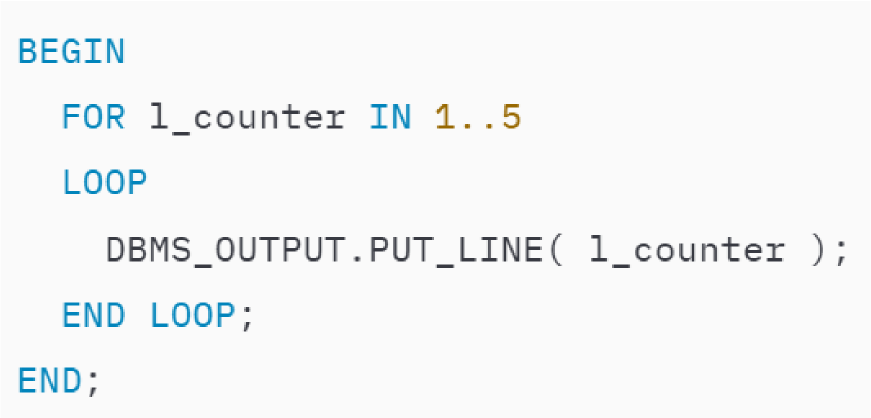
PL/SQL

- FOR LOOP statement
  - FOR LOOP executes a sequence of statements a specified number of times.
  - The FOR LOOP statement has the following structure:



```
FOR index IN lower_bound .. upper_bound
LOOP
    statements;
END LOOP;
```

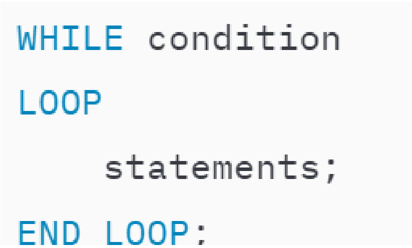
  - Example:



```
BEGIN
  FOR l_counter IN 1..5
  LOOP
    DBMS_OUTPUT.PUT_LINE( l_counter );
  END LOOP;
END;
```

PL/SQL

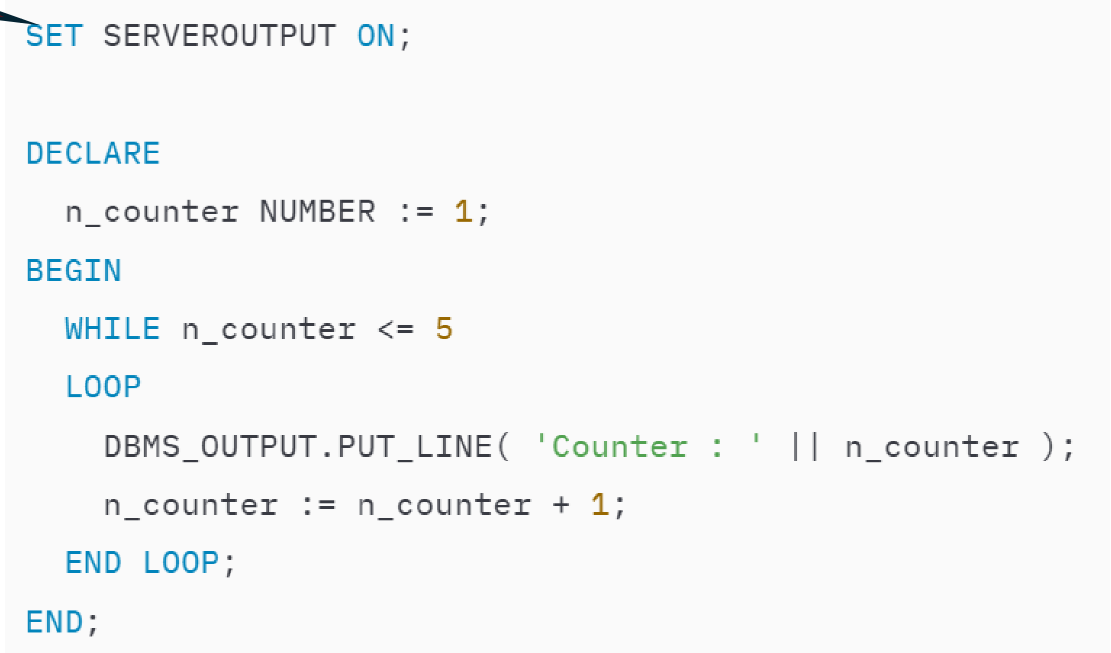
- WHILE loop statement
  - WHILE loop is a control structure that repeatedly executes a code block if a specific condition remains true.
  - Here's the syntax for the WHILE loop statement:



```
WHILE condition
LOOP
    statements;
END LOOP;
```

This command line enables the display of output from the DBMS_OUTPUT.PUT_LINE procedure on your screen.

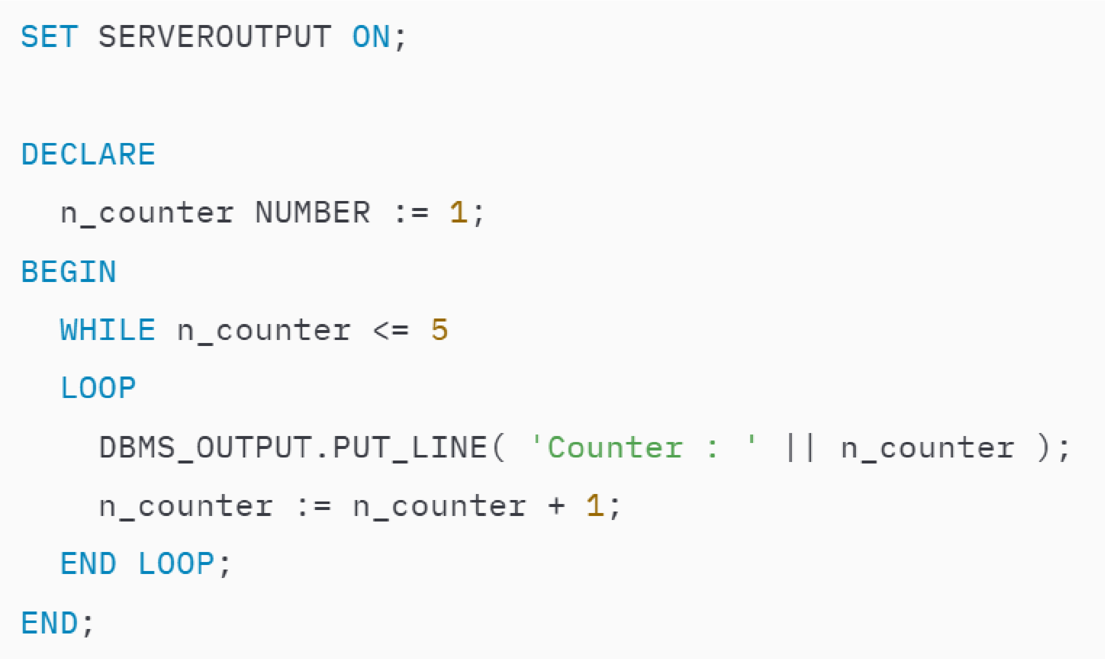
  - Example:



```
SET SERVEROUTPUT ON;

DECLARE
  n_counter NUMBER := 1;
BEGIN
  WHILE n_counter <= 5
  LOOP
    DBMS_OUTPUT.PUT_LINE( 'Counter : ' || n_counter );
    n_counter := n_counter + 1;
  END LOOP;
END;
```

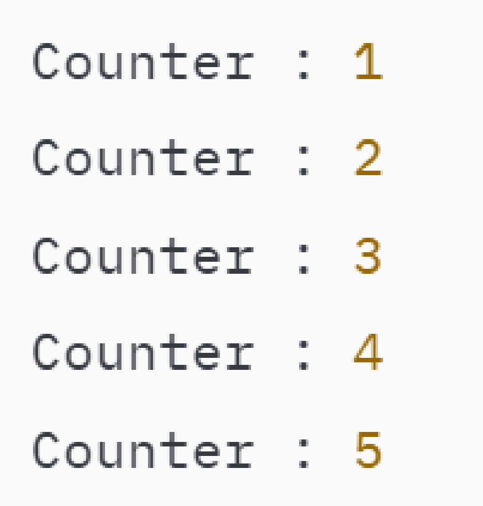
PL/SQL

- WHILE loop statement
  - Example:



```
SET SERVEROUTPUT ON;

DECLARE
  n_counter NUMBER := 1;
BEGIN
  WHILE n_counter <= 5
  LOOP
    DBMS_OUTPUT.PUT_LINE( 'Counter : ' || n_counter );
    n_counter := n_counter + 1;
  END LOOP;
END;
```

Output



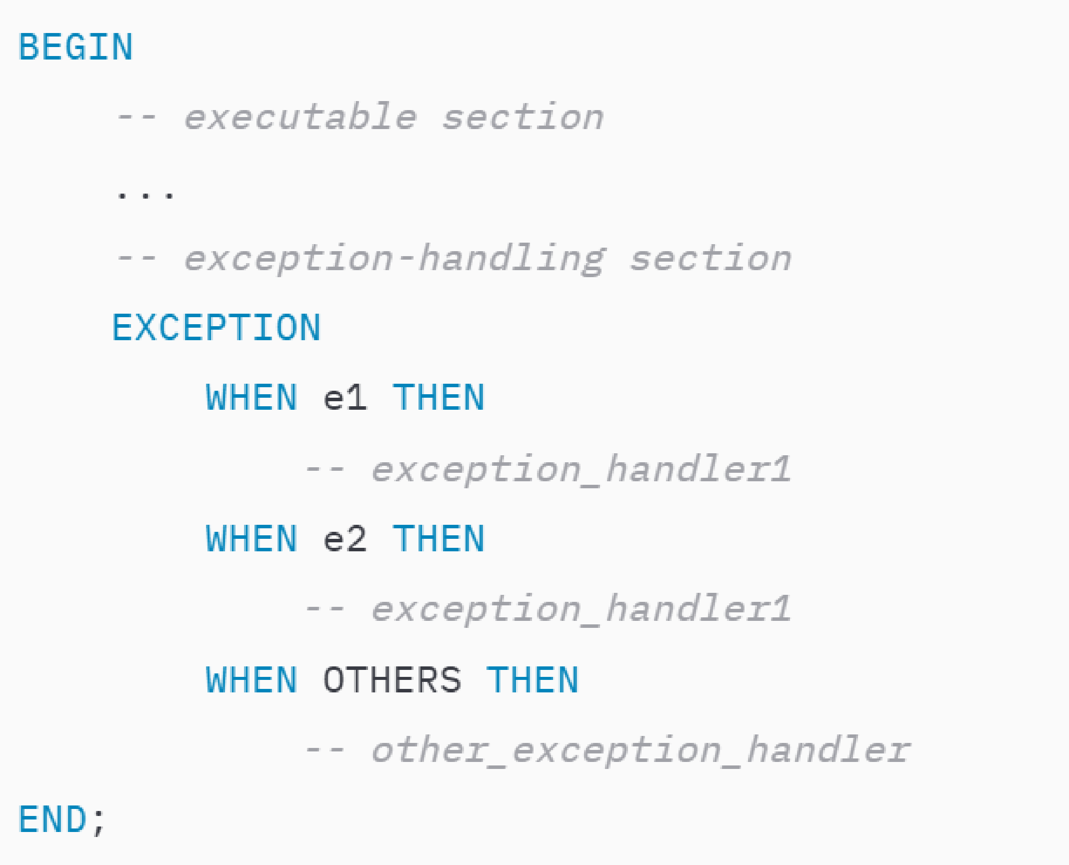
```
Counter : 1
Counter : 2
Counter : 3
Counter : 4
Counter : 5
```

PL/SQL

- PL/SQL Exceptions
  - PL/SQL treats all errors in an anonymous block, procedure, or function as exceptions. The exceptions can have different causes, such as:
    - Coding mistakes
    - Software bugs
    - Hardware failures
  - It is not possible to anticipate all potential exceptions. However, you can write code to handle exceptions to enable the program to continue running as usual.
  - The code that you write to handle exceptions is called an exception handler.
  - A PL/SQL block may have an exception-handling section with one or more exception handlers.

PL/SQL

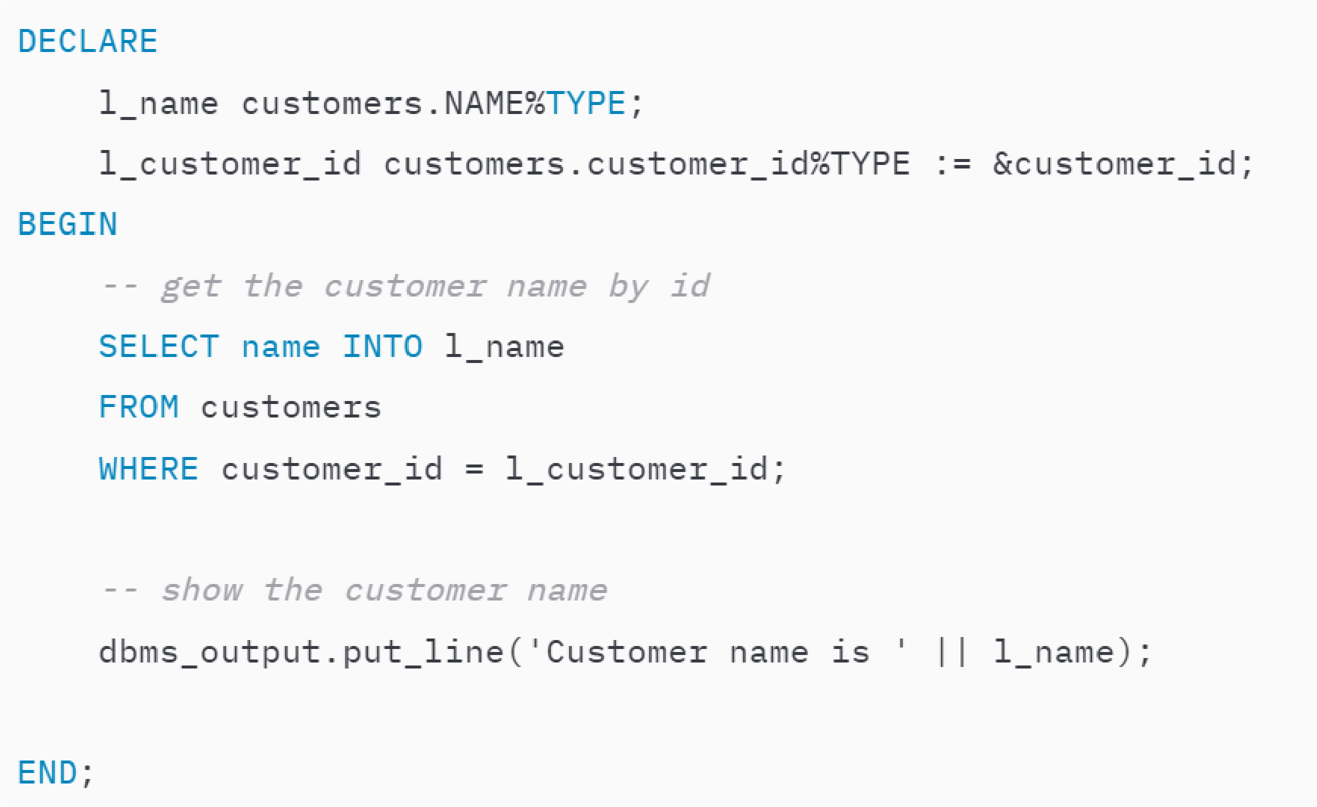
- PL/SQL Exceptions
  - Here’s the basic syntax of the exception-handling section:



```
BEGIN
    -- executable section
    ...
    -- exception-handling section
    EXCEPTION
        WHEN e1 THEN
            -- exception_handler1
        WHEN e2 THEN
            -- exception_handler1
        WHEN OTHERS THEN
            -- other_exception_handler
END;
```

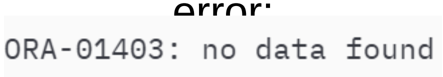
PL/SQL

- PL/SQL Exceptions
  - NO_DATA_FOUND exception example:



```
DECLARE
    l_name customers.NAME%TYPE;
    l_customer_id customers.customer_id%TYPE := &customer_id;
BEGIN
    -- get the customer name by id
    SELECT name INTO l_name
    FROM customers
    WHERE customer_id = l_customer_id;

    -- show the customer name
    dbms_output.put_line('Customer name is ' || l_name);

END;
```

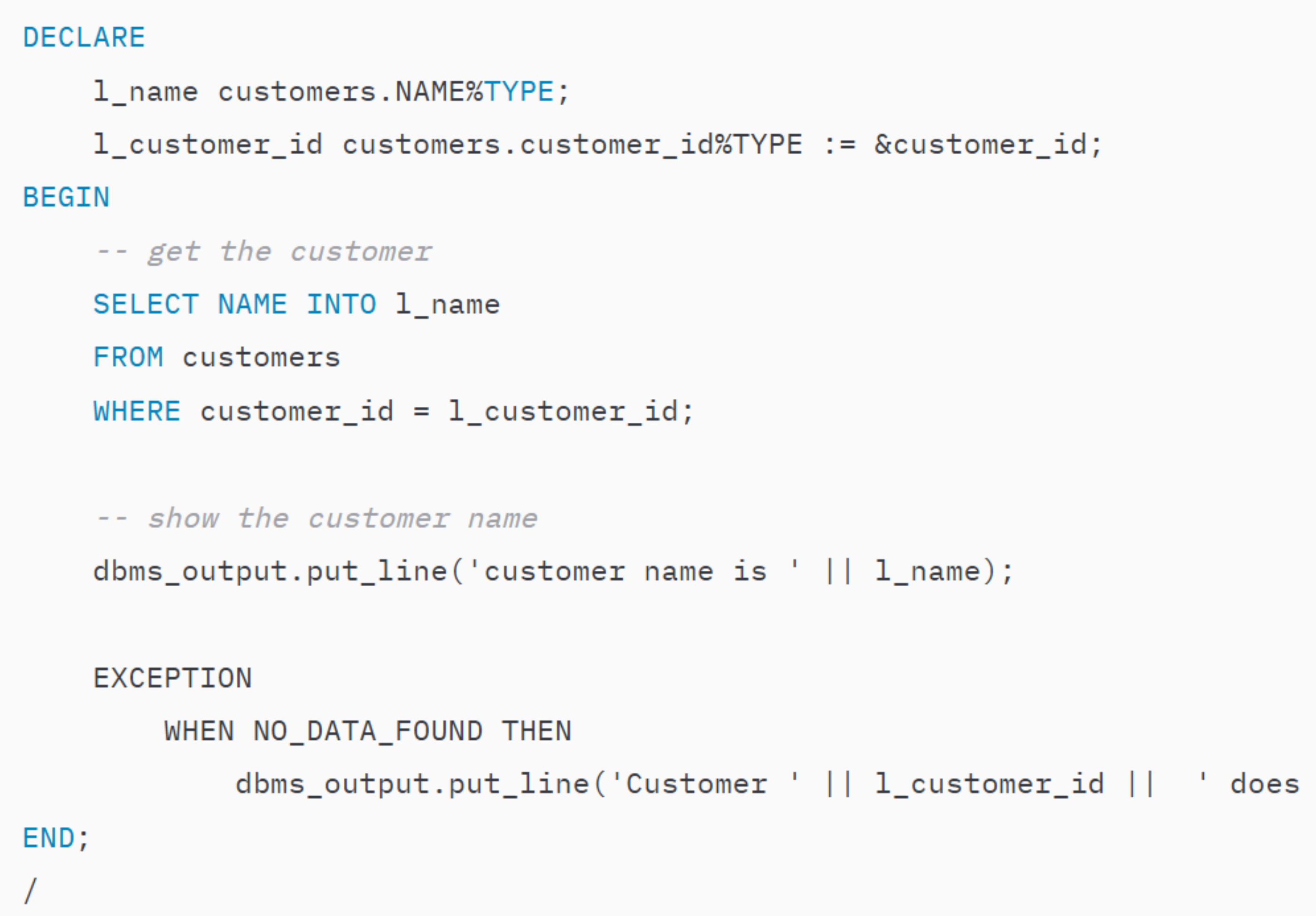
If you execute the block and enter the customer id as 0, Oracle will issue the following error:



```
ORA-01403: no data found
```

PL/SQL

- PL/SQL Exceptions
  - NO_DATA_FOUND exception-handling example:



```
DECLARE
    l_name customers.NAME%TYPE;
    l_customer_id customers.customer_id%TYPE := &customer_id;
BEGIN
    -- get the customer
    SELECT NAME INTO l_name
    FROM customers
    WHERE customer_id = l_customer_id;

    -- show the customer name
    dbms_output.put_line('customer name is ' || l_name);

    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            dbms_output.put_line('Customer ' || l_customer_id ||  ' does not exist');
END;
/
```

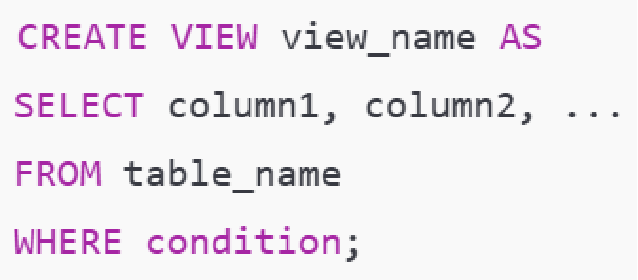
PL/SQL

- Views

  - A view in SQL is a virtual table, it does not store data itself, but displays data from one or more tables through a saved SQL query.

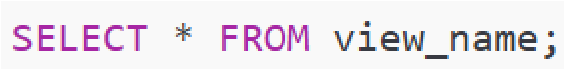
  - A view could be seen as a window or a shortcut to complex queries.

  - Syntax:



```
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

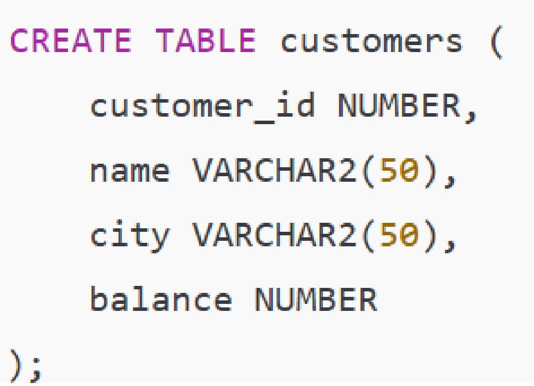
  - Once created, we can use the view just like a regular table:



```
SELECT * FROM view_name;
```

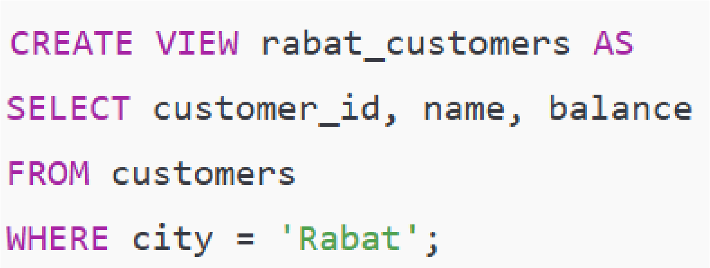
PL/SQL

- Views
  - Example:
    - Suppose we have a table:



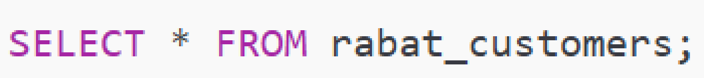
```
CREATE TABLE customers (
    customer_id NUMBER,
    name VARCHAR2(50),
    city VARCHAR2(50),
    balance NUMBER
);
```

- If we often need to see customers from Rabat only, we can create a view:



```
CREATE VIEW rabat_customers AS
SELECT customer_id, name, balance
FROM customers
WHERE city = 'Rabat';
```

Now we can simply query:



```
SELECT * FROM rabat_customers;
```

PL/SQL

- Views
  - Why use views?
    - Simplify complex queries
    - Improve security (hide certain columns)
    - Make code easier to read and maintain

PL/SQL

- Cursors
  - A cursor is a pointer to rows returned by a query.
  - PL/SQL has two types of cursors:
    - Implicit cursors.
    - Explicit cursors.

PL/SQL

- Cursors - Implicit cursors

  - Oracle automatically creates implicit cursor for SELECT INTO, INSERT, UPDATE, or DELETE. We don’t declare it ourselves.

  - In other words, Oracle automatically uses an implicit cursor to fetch that one row.

PL/SQL

- Cursors - Explicit cursors

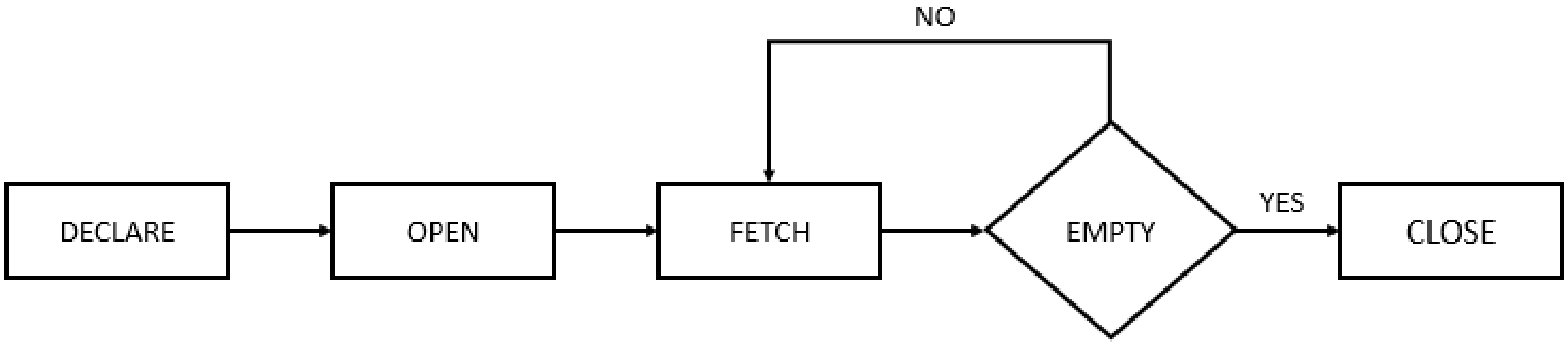
    - An explicit cursor is a SELECT statement declared explicitly in the declaration section of the current block.

    - For an explicit cursor, we have control over its execution cycle from OPEN, FETCH, and CLOSE.

    - Oracle defines an execution cycle that executes an SQL statement and associates a cursor with it.

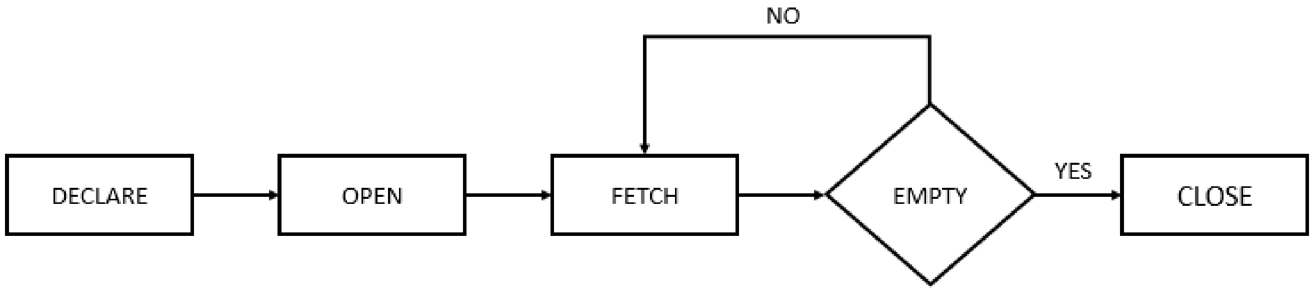
PL/SQL

- Cursors - Explicit cursors
  - The following illustration shows the execution cycle of an explicit cursor:

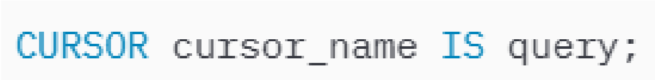


PL/SQL

- Cursors - Explicit cursors



- **Declare a cursor**:
  - Before using an explicit cursor, we must declare it in the declaration section of a block as follows:

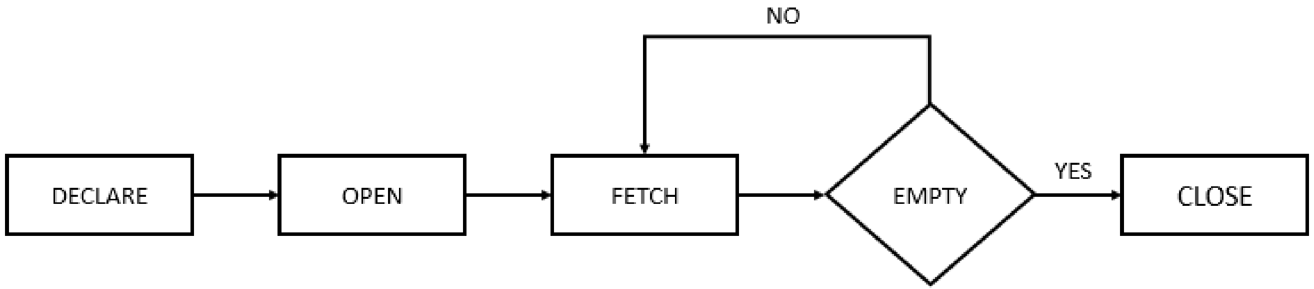


```
CURSOR cursor_name IS query;
```

  - In this syntax:
    - First, we specify the name of the cursor after the CURSOR keyword.
    - Second, we define a query to fetch data after the IS keyword.

PL/SQL

- Cursors - Explicit cursors



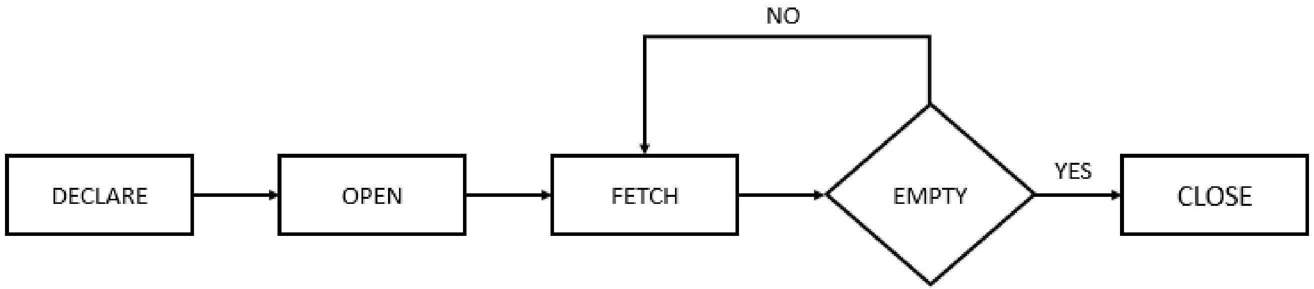
  - **Open a cursor**:
    - Before starting to fetch rows from the cursor, we must open it. To open a cursor, we use the following syntax:

```
OPEN cursor_name;
```

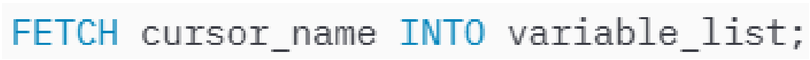
    - In this syntax, cursor_name is the name of the cursor we declare in the declaration section.
    - When we open a cursor, Oracle parses the query, binds variables, and executes the associated SQL statement.

PL/SQL

- Cursors - Explicit cursors



- **Fetch from a cursor**:
  - The FETCH statement places the contents of the current row into variables. The syntax of FETCH statement is as follows:

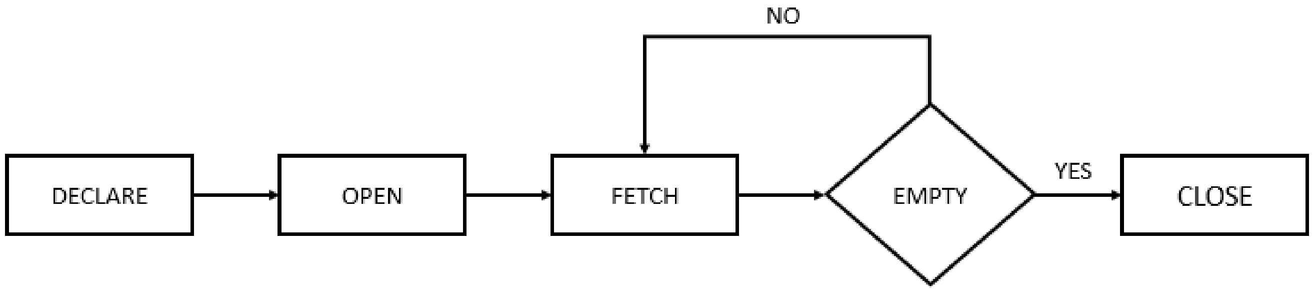


```
FETCH cursor_name INTO variable_list;
```

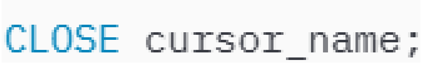
  - To retrieve all rows in a result set, we must fetch each row until the last one.

PL/SQL

- Cursors - Explicit cursors



- **Closing a cursor**:
  - After fetching all rows, we need to close the cursor with the CLOSE statement:



```
CLOSE cursor_name;
```

  - Closing a cursor instructs Oracle to release allocated memory at an appropriate time.

PL/SQL

- Cursors - Explicit cursors
  - **Explicit Cursor Attributes**:
    - A cursor has four attributes, which we can reference in the following format:

```
cursor_name%attribute
```

where cursor_name is the name of the explicit cursor.

- The four cursor attributes are:
  - **%ISOPEN**
  - **%FOUND**
  - **%NOTFOUND**
  - **%ROWCOUNT**

PL/SQL

- Cursors - Explicit cursors
  - **Explicit Cursor Attributes**:
    - **%ISOPEN**
      - This attribute is TRUE if the cursor is open or FALSE if it is not.

PL/SQL

- Cursors - Explicit cursors
  - **Explicit Cursor Attributes**:
    - **%FOUND**
      - This attribute has four values:
        - NULL before the first fetch.
        - TRUE if a record was fetched successfully.
        - FALSE if no row is returned.
        - INVALID_CURSOR if the cursor is not opened.

PL/SQL

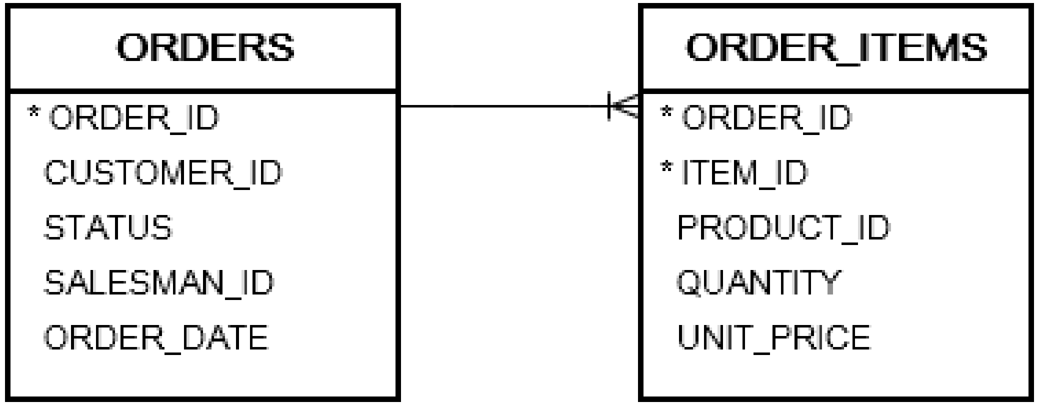
- Cursors - Explicit cursors
  - **Explicit Cursor Attributes**:
    - **%NOTFOUND**
      - This attribute has four values:
        - NULL before the first fetch.
        - FALSE if a record was fetched successfully.
        - TRUE if no row is returned.
        - INVALID_CURSOR if the cursor is not opened.

PL/SQL

- Cursors - Explicit cursors
  - **Explicit Cursor Attributes**:
    - **%ROWCOUNT**
      - The %ROWCOUNT attribute returns the number of rows fetched from the cursor. If the cursor is not opened, this attribute returns INVALID_CURSOR.

PL/SQL

- Cursors - Example
  - Consider two tables ‘orders’ and ’order_items’.



ORDERS

* ORDER_ID
CUSTOMER_ID
STATUS
SALESMAN_ID
ORDER_DATE

ORDER_ITEMS

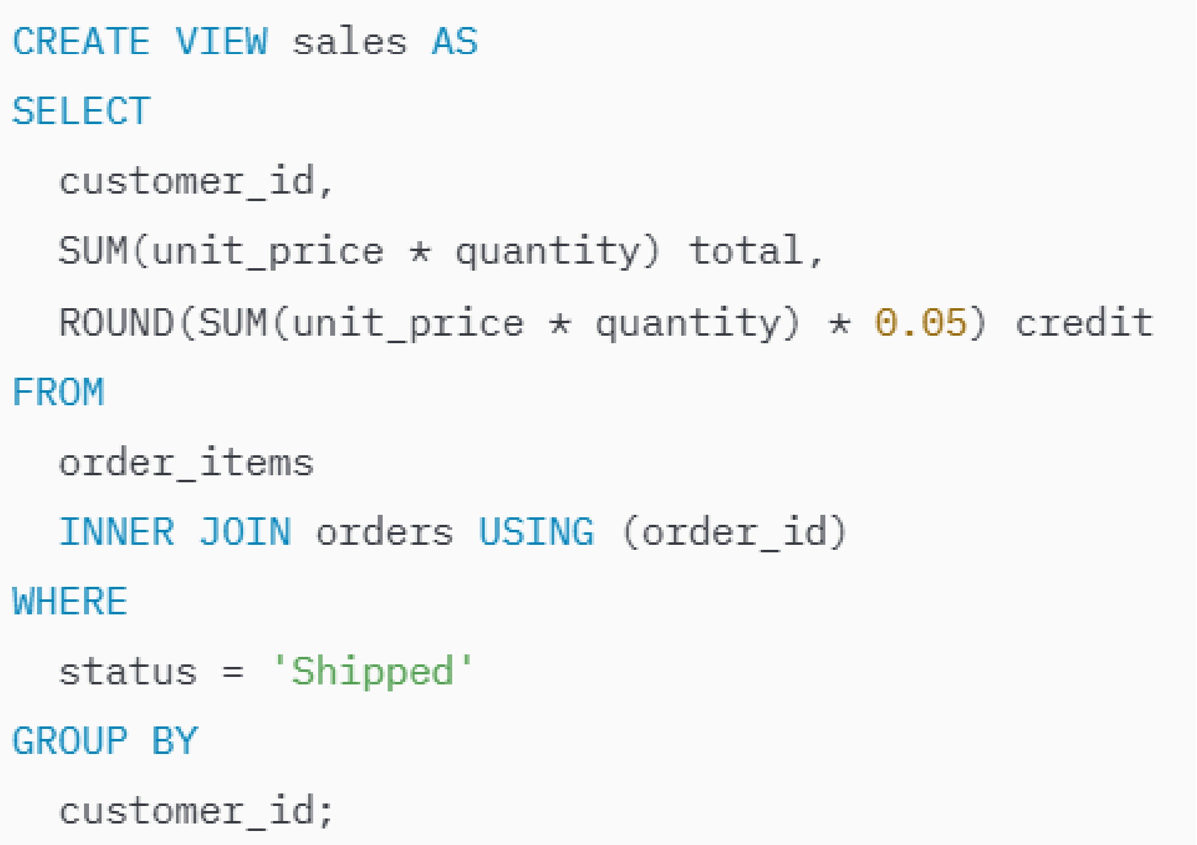
* ORDER_ID
* ITEM_ID
PRODUCT_ID
QUANTITY
UNIT_PRICE

PL/SQL

- Cursors - Example

  - The following statement creates a view

  that returns the sales revenues by customers:

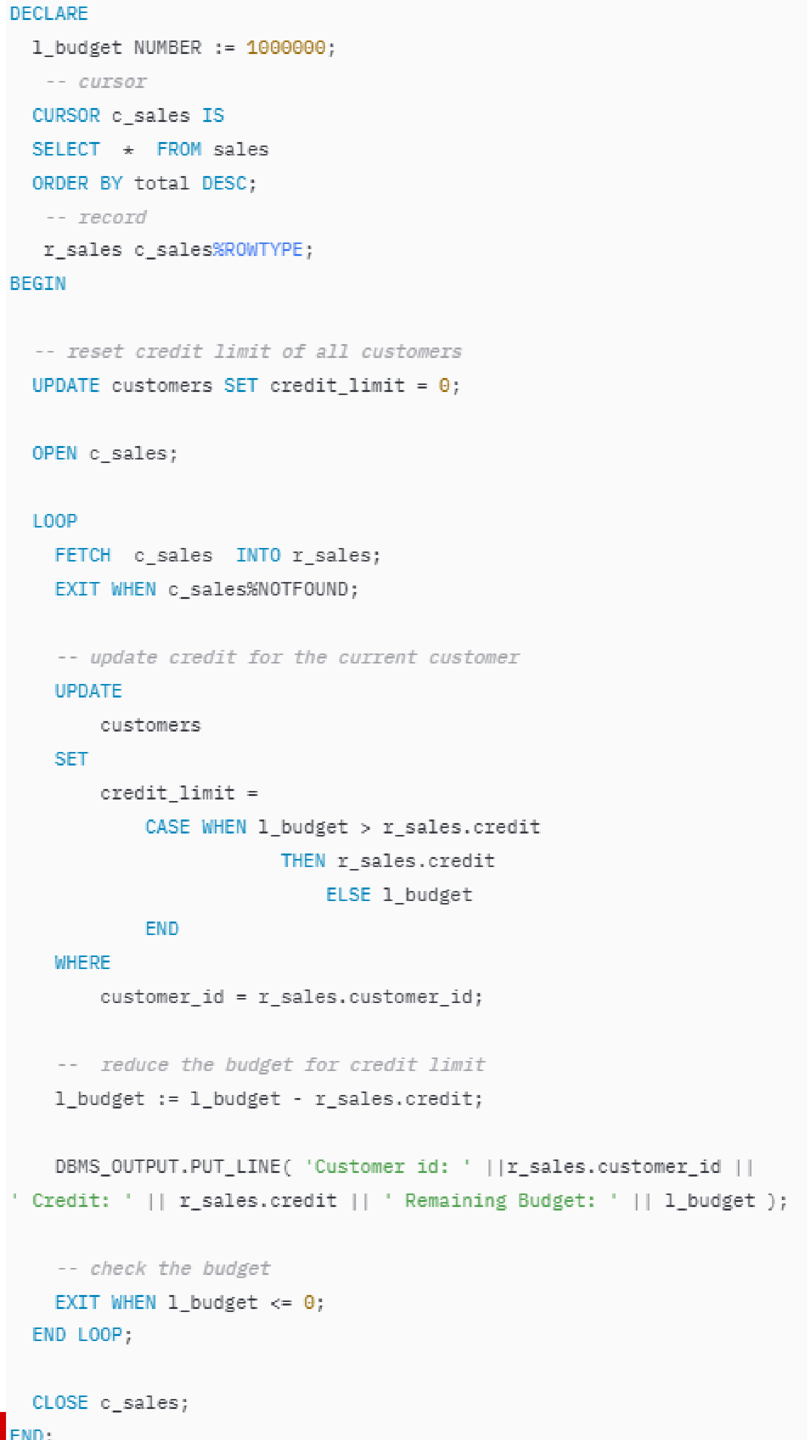


```
CREATE VIEW sales AS
SELECT
  customer_id,
  SUM(unit_price * quantity) total,
  ROUND(SUM(unit_price * quantity) * 0.05) credit
FROM
  order_items
  INNER JOIN orders USING (order_id)
WHERE
  status = 'Shipped'
GROUP BY
  customer_id;
```

- The values of the credit column are 5% of the total sales revenues.
- Suppose we need to develop an anonymous block that:
  - Reset the credit limits of all customers to zero.
  - Fetch customers sorted by sales in descending order and give them new credit limits from a budget of 1 million.

PL/SQL

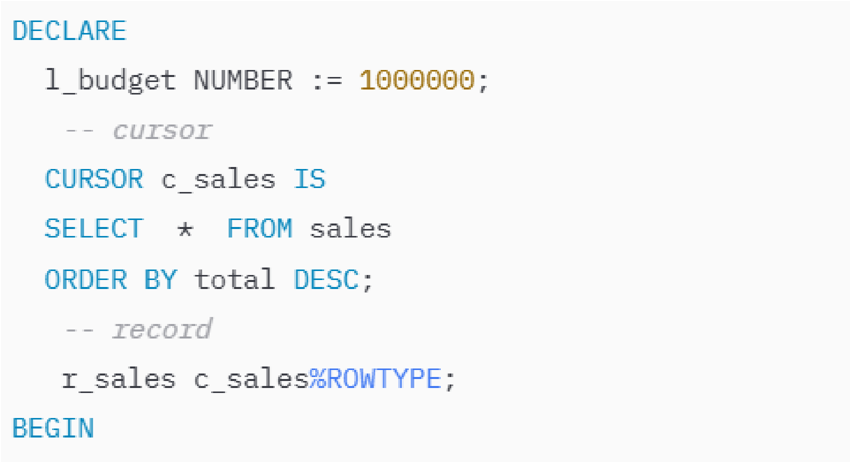
- Cursors - Example
  - The right-hand side block illustrates the logic.



```
DECLARE
  l_budget NUMBER := 1000000;
   -- cursor
  CURSOR c_sales IS
  SELECT  *  FROM sales
  ORDER BY total DESC;
   -- record
   r_sales c_sales%ROWTYPE;
BEGIN

  -- reset credit limit of all customers
  UPDATE customers SET credit_limit = 0;

  OPEN c_sales;

  LOOP
    FETCH  c_sales  INTO r_sales;
    EXIT WHEN c_sales%NOTFOUND;

    -- update credit for the current customer
    UPDATE
        customers
    SET
        credit_limit =
            CASE WHEN l_budget > r_sales.credit
                        THEN r_sales.credit
                            ELSE l_budget
            END
    WHERE
        customer_id = r_sales.customer_id;

    --  reduce the budget for credit limit
    l_budget := l_budget - r_sales.credit;

    DBMS_OUTPUT.PUT_LINE( 'Customer id: ' ||r_sales.customer_id ||
' Credit: ' || r_sales.credit || ' Remaining Budget: ' || l_budget );

    -- check the budget
    EXIT WHEN l_budget <= 0;
  END LOOP;

  CLOSE c_sales;
END;
```

PL/SQL

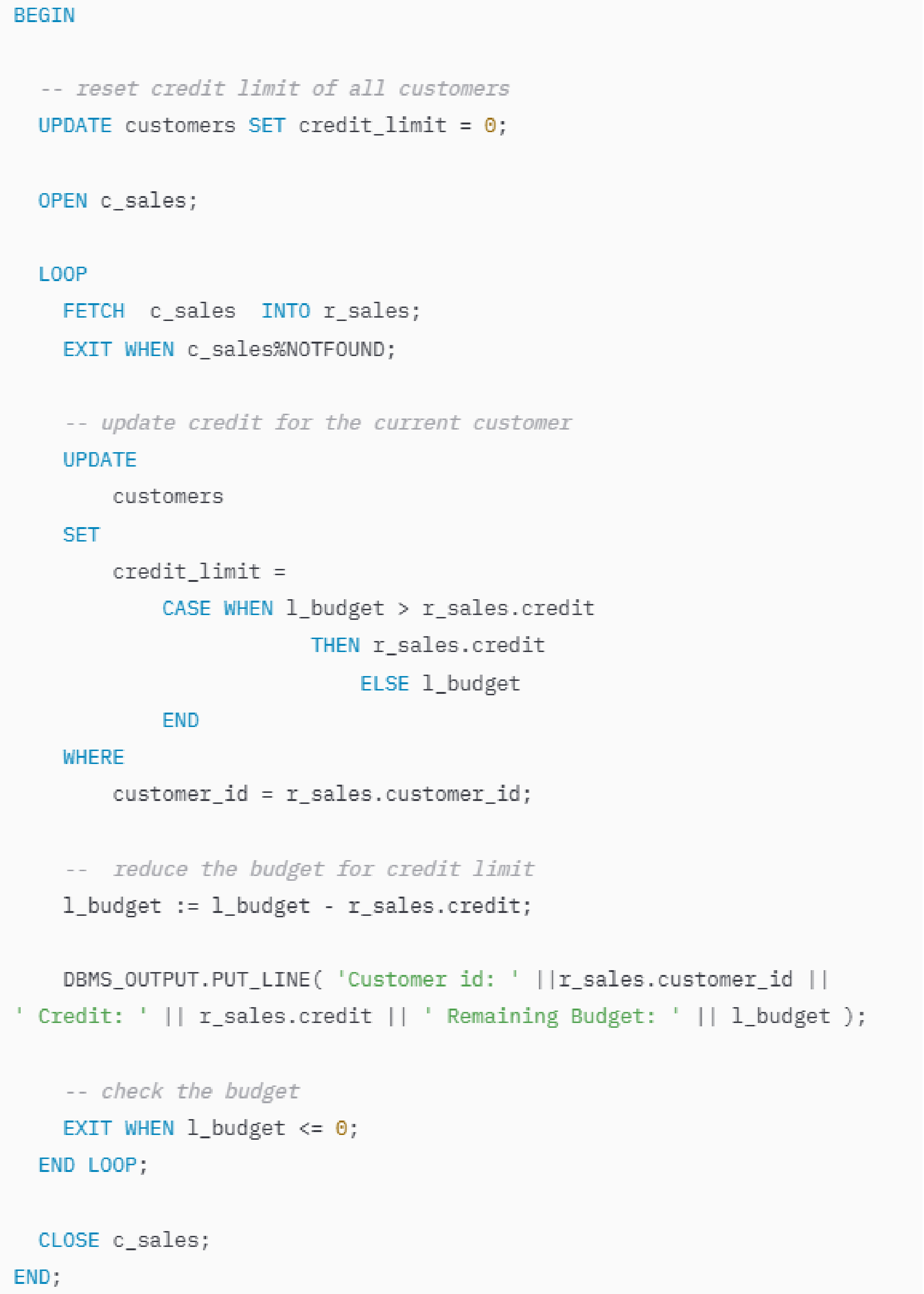
- Cursors - Example
  - In the declaration section, we declare three variables.
    - The first one is l_budget whose initial value is 1,000,000.
    - The second variable is an explicit cursor variable named c_sales whose SELECT statement retrieves data from the sales view.
    - The third variable is a cursor-based record named c_sales.



```
DECLARE
  l_budget NUMBER := 1000000;
   -- cursor
  CURSOR c_sales IS
  SELECT  *  FROM sales
  ORDER BY total DESC;
   -- record
   r_sales c_sales%ROWTYPE;
BEGIN
```

PL/SQL

- Cursors - Example
  - In the execution section, we perform the following:
  - First, reset the credit limits of all customers to zero using an UPDATE statement.
  - Second, open the c_sales cursor.
  - Third, fetch each row from the cursor. We updated the credit limit and reduced the budget in each loop iteration. The loop terminates when no row is fetched, or the budget is exhausted.
  - Finally, close the cursor.

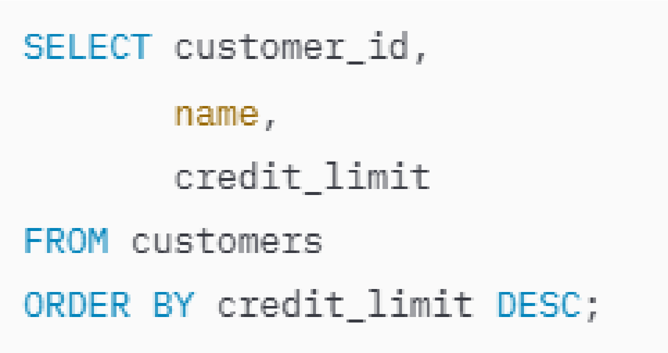


```
BEGIN

  -- reset credit limit of all customers
  UPDATE customers SET credit_limit = 0;

  OPEN c_sales;

  LOOP
    FETCH  c_sales  INTO r_sales;
    EXIT WHEN c_sales%NOTFOUND;

    -- update credit for the current customer
    UPDATE
        customers
    SET
        credit_limit =
            CASE WHEN l_budget > r_sales.credit
                        THEN r_sales.credit
                            ELSE l_budget
            END
    WHERE
        customer_id = r_sales.customer_id;

    --  reduce the budget for credit limit
    l_budget := l_budget - r_sales.credit;

    DBMS_OUTPUT.PUT_LINE( 'Customer id: ' ||r_sales.customer_id ||
' Credit: ' || r_sales.credit || ' Remaining Budget: ' || l_budget );

    -- check the budget
    EXIT WHEN l_budget <= 0;
  END LOOP;

  CLOSE c_sales;
END;
```

PL/SQL

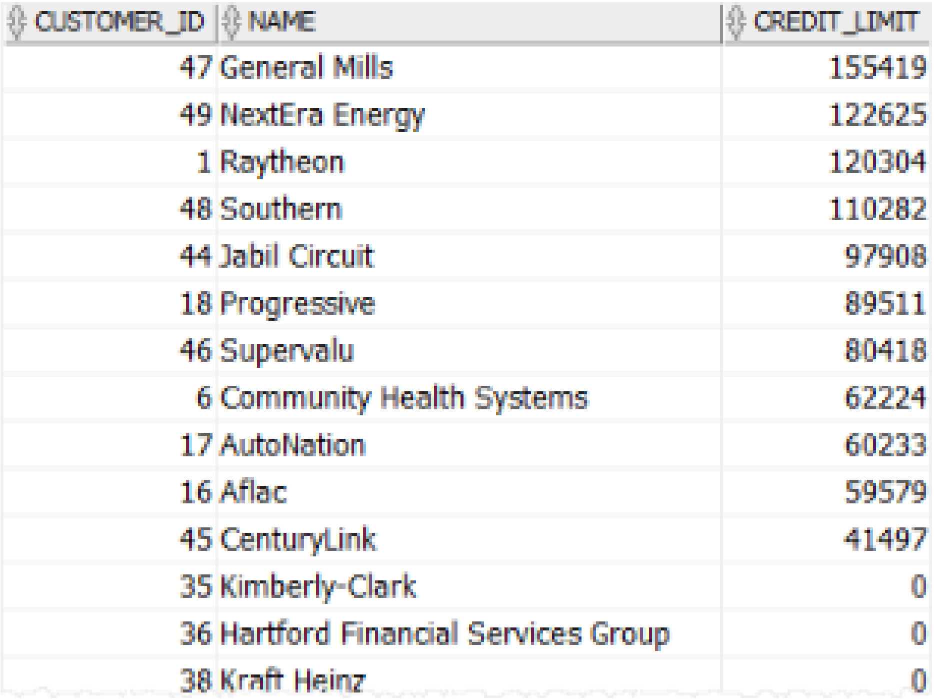
- Cursors - Example

  - The following query retrieves data from the customers table to verify the update:



```
SELECT customer_id,
       name,
       credit_limit
FROM customers
ORDER BY credit_limit DESC;
```

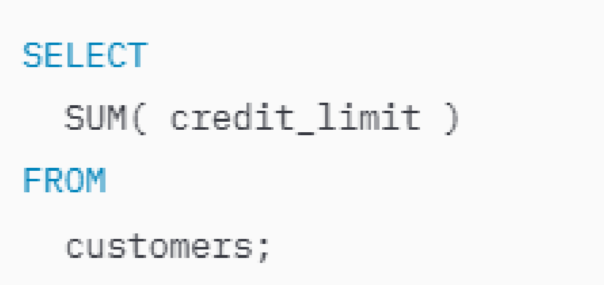
- Result:



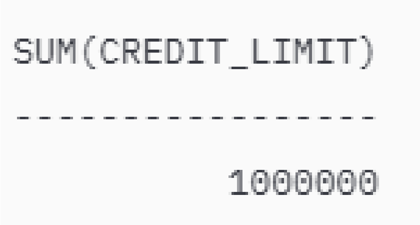
CUSTOMER_ID	NAME	CREDIT_LIMIT
47	General Mills	155419
49	NextEra Energy	122625
1	Raytheon	120304
48	Southern	110282
44	Jabil Circuit	97908
18	Progressive	89511
46	Supervalu	80418
6	Community Health Systems	62224
17	AutoNation	60233
16	Aflac	59579
45	CenturyLink	41497
35	Kimberly-Clark	0
36	Hartford Financial Services Group	0
38	Kraft Heinz	0

PL/SQL

- Cursors - Example
  - The output indicates that only the first few customers have credit limits. If we sum up all credit limits, the total should be 1 million as shown follows:



```
SELECT
  SUM( credit_limit )
FROM
  customers;
```



```
SUM(CREDIT_LIMIT)
-----------------
          1000000
```

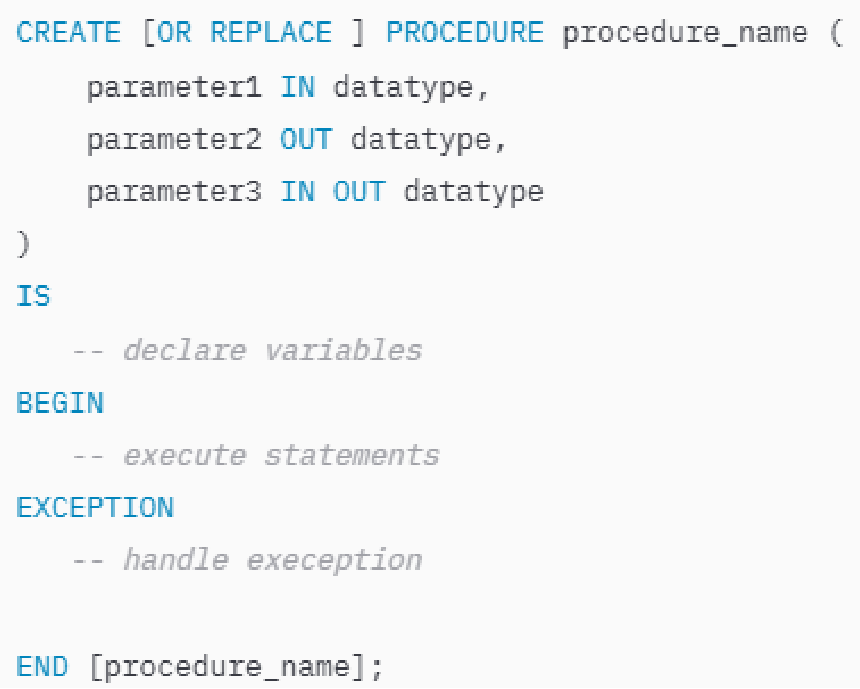
PL/SQL

- Procedures

  - In PL/SQL, a procedure is a named block that performs a specific task. Unlike an anonymous block, Oracle stores the procedure in the database, and we can execute it repeatedly.

  - Typically, we create a procedure to encapsulate a reusable code block.

  - Here’s the syntax for creating a procedure:

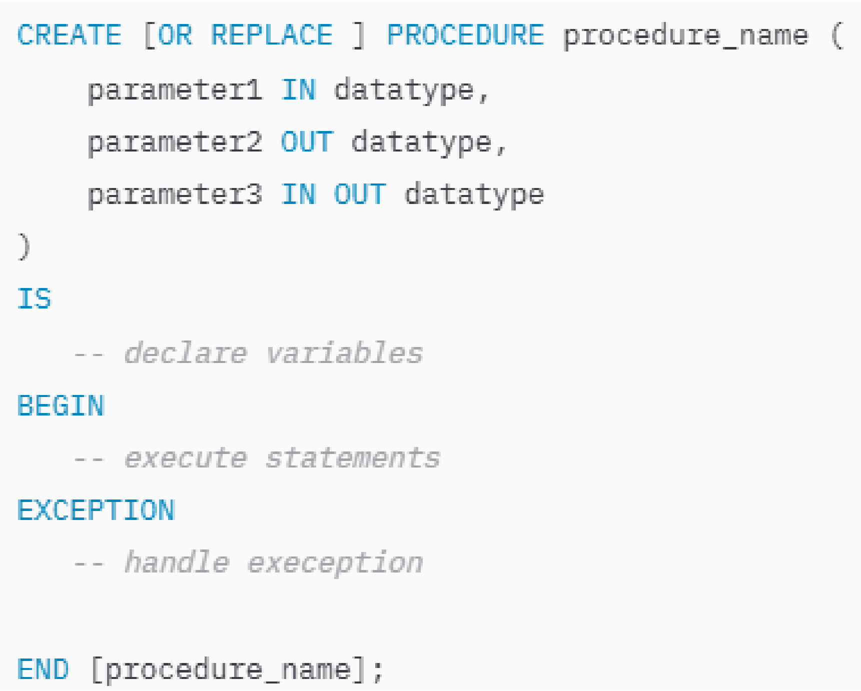


```
CREATE [OR REPLACE ] PROCEDURE procedure_name (
    parameter1 IN datatype,
    parameter2 OUT datatype,
    parameter3 IN OUT datatype
)
IS
   -- declare variables
BEGIN
   -- execute statements
EXCEPTION
   -- handle exeception

END [procedure_name];
```

The OR REPLACE option allows us to update an existing procedure with the new code without dropping it.

PL/SQL

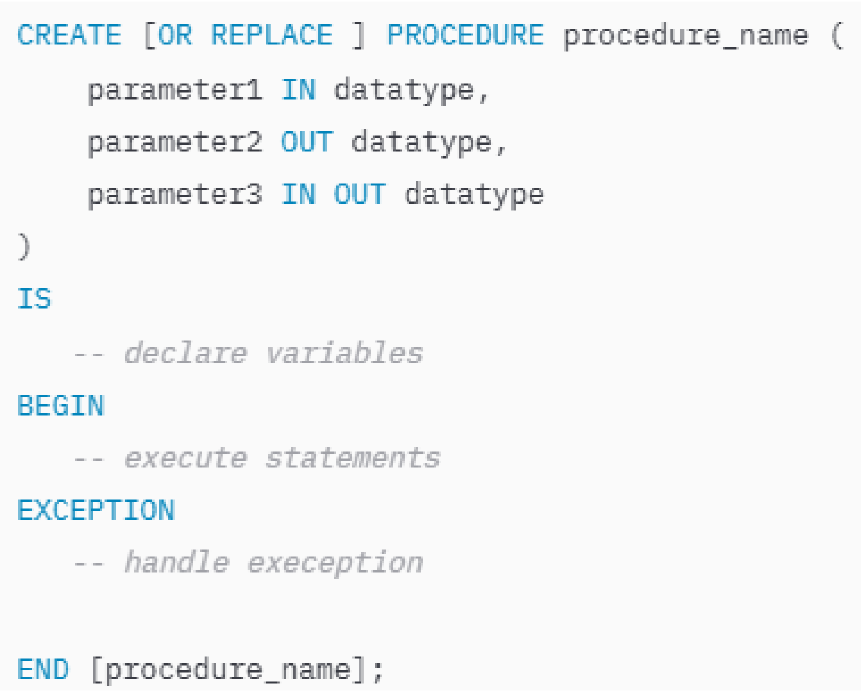
- Procedures - Header
  - A procedure begins with a header that specifies its name and an optional parameter list.
  - Each parameter can be in either IN, OUT, or INOUT mode.
  - The parameter mode determines if the procedure can access its initial value or modify its value:
    - **IN** parameter is read-only. We can reference its value within a procedure, but not modify it. If we don’t specify a parameter mode explicitly, PL/SQL automatically uses IN as the default mode.
    - **OUT** parameter is writable. It means you can assign a value to an OUT parameter and return it to the program called the procedure. Note that a procedure ignores the value that we provide for an OUT parameter.
    - **INOUT** parameter is both readable and writable. This means the procedure can access its initial value and also modify it.



```
CREATE [OR REPLACE ] PROCEDURE procedure_name (
    parameter1 IN datatype,
    parameter2 OUT datatype,
    parameter3 IN OUT datatype
)
IS
   -- declare variables
BEGIN
   -- execute statements
EXCEPTION
   -- handle exeception

END [procedure_name];
```

PL/SQL

- Procedures - Body
  - Similar to an anonymous block, the procedure body has three parts:
    - Declaration: This section is where we can declare variables, constants, cursors, and other elements. Unlike an anonymous block, a declaration section of a procedure does not start with the DECLARE keyword.
    - Execution: This section contains the core logic of the procedure, consisting of one or more statements that perform specific tasks. It can even be as simple as a NULL statement.
    - Exception handler: This section contains code for handling and responding to errors.

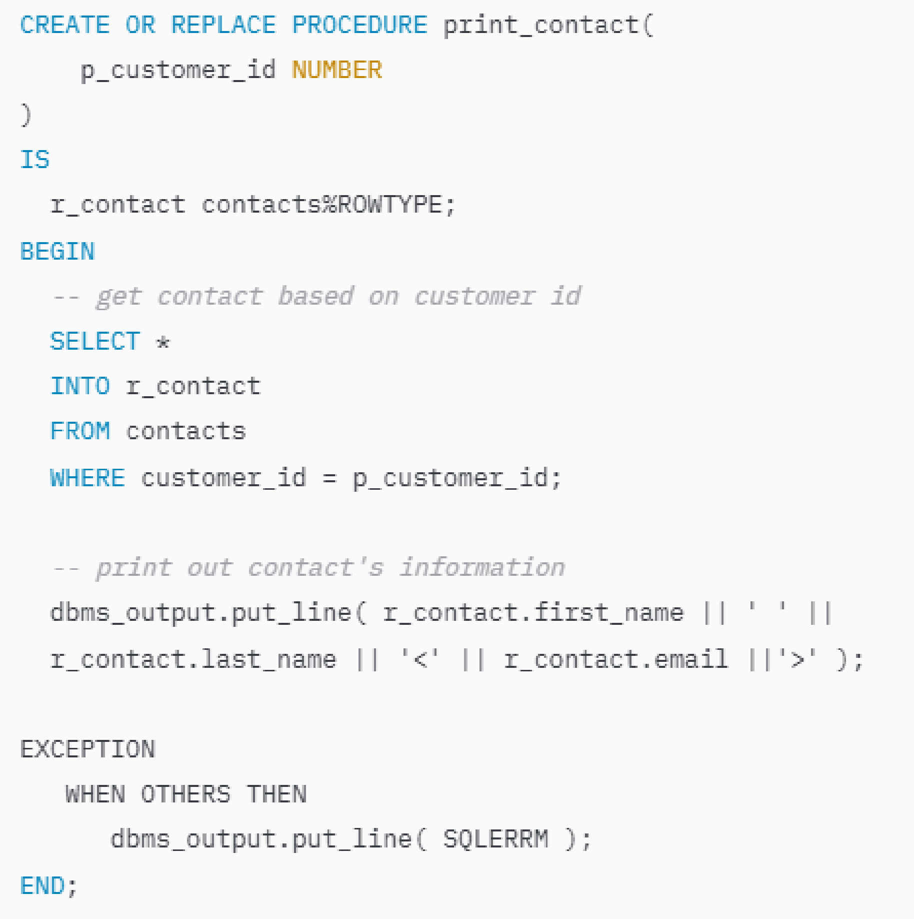


```
CREATE [OR REPLACE ] PROCEDURE procedure_name (
    parameter1 IN datatype,
    parameter2 OUT datatype,
    parameter3 IN OUT datatype
)
IS
   -- declare variables
BEGIN
   -- execute statements
EXCEPTION
   -- handle exeception

END [procedure_name];
```

PL/SQL

- Procedures - Example

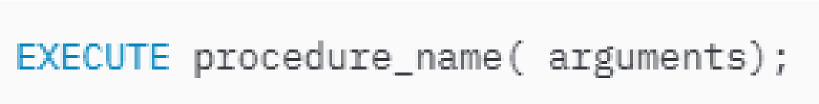
  - The following procedure accepts a customer id and displays the customer’s contact information, including first name, last name, and email:



```
CREATE OR REPLACE PROCEDURE print_contact(
    p_customer_id NUMBER
)
IS
  r_contact contacts%ROWTYPE;
BEGIN
  -- get contact based on customer id
  SELECT *
  INTO r_contact
  FROM contacts
  WHERE customer_id = p_customer_id;

  -- print out contact's information
  dbms_output.put_line( r_contact.first_name || ' ' ||
  r_contact.last_name || '<' || r_contact.email ||'>' );

EXCEPTION
   WHEN OTHERS THEN
      dbms_output.put_line( SQLERRM );
END;
```

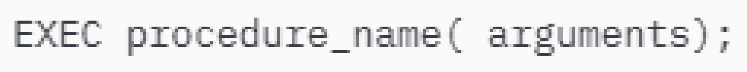
PL/SQL

- Procedures – Executing a procedure
  - The following shows the syntax for executing a procedure:



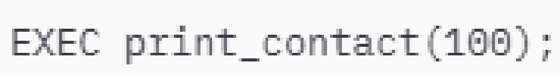
```
EXECUTE procedure_name( arguments);
```

  - Or



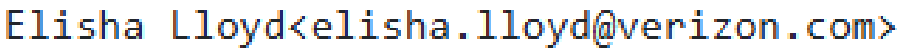
```
EXEC procedure_name( arguments);
```

  - For example, to execute the print_contact procedure that prints the contact information of customer id 100, we use the following statement:



```
EXEC print_contact(100);
```

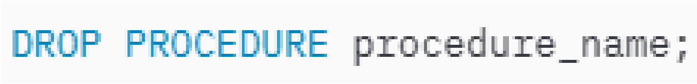
  - Output:



```
Elisha Lloyd<elisha.lloyd@verizon.com>
```

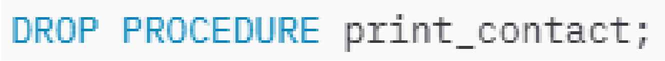
PL/SQL

- Procedures – Removing a procedure
  - To delete a procedure, we use the DROP PROCEDURE statement:



```
DROP PROCEDURE procedure_name;
```

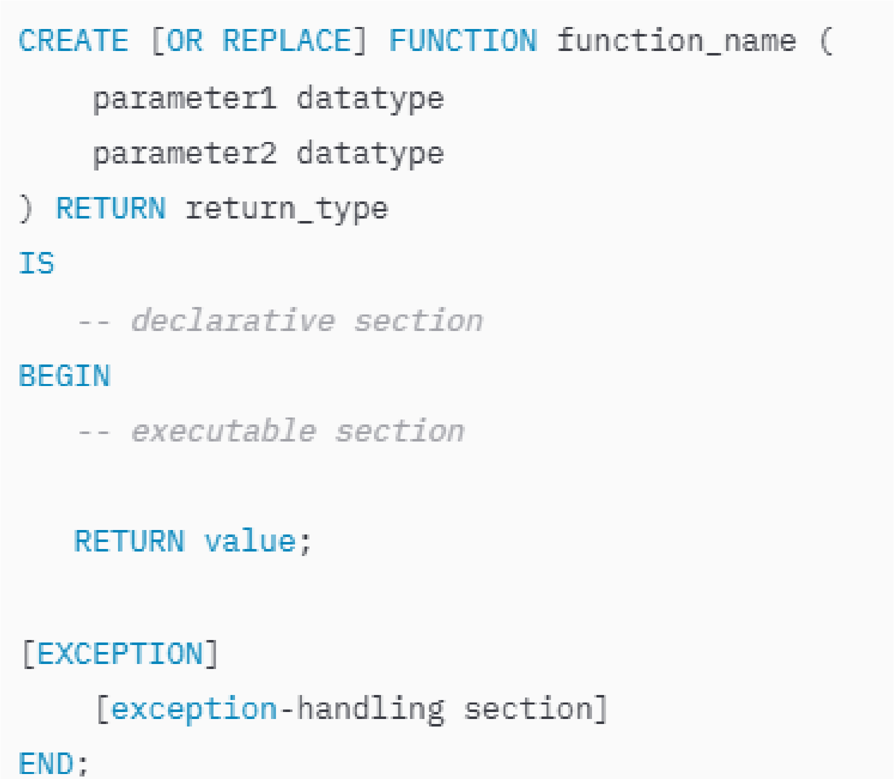
  - For example, the following statement drops the print_contact procedure :



```
DROP PROCEDURE print_contact;
```

PL/SQL

- Functions
  - In PL/SQL, a function is a reusable code block that performs a specific task and returns a single value.
  - Here’s the syntax for creating a function:



```
CREATE [OR REPLACE] FUNCTION function_name (
    parameter1 datatype
    parameter2 datatype
) RETURN return_type
IS
   -- declarative section
BEGIN
   -- executable section

   RETURN value;

[EXCEPTION]
    [exception-handling section]
END;
```

PL/SQL

- Functions – Header and Body

  - A function consists of a header and body:

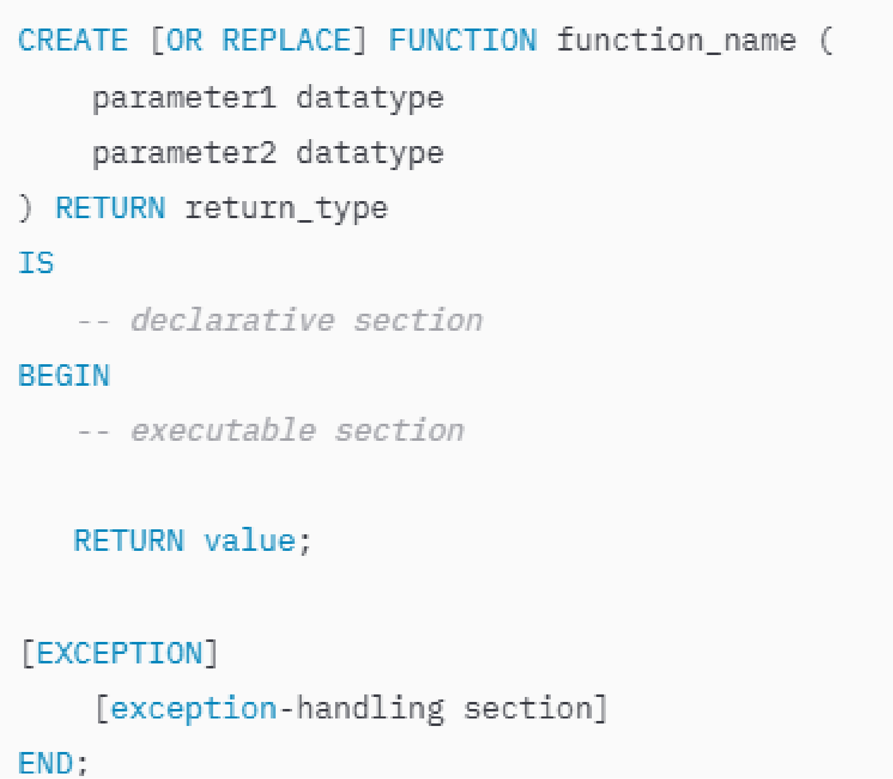
  - Function header consists of a function name and a RETURN clause specifying the returned value's datatype. Each parameter of the function can be either in the IN, OUT, or INOUT mode.

  - The function body is the same as the procedure's body, which has three sections: declaration, execution, and exception handler.

    - Declaration section is where we declare variables, constants, cursors, and user-defined types.

    - Execution section is where we place the executable statements. It's between the BEGIN and END keywords. Unlike a procedure, we must have at least one RETURN statement in the execution section.

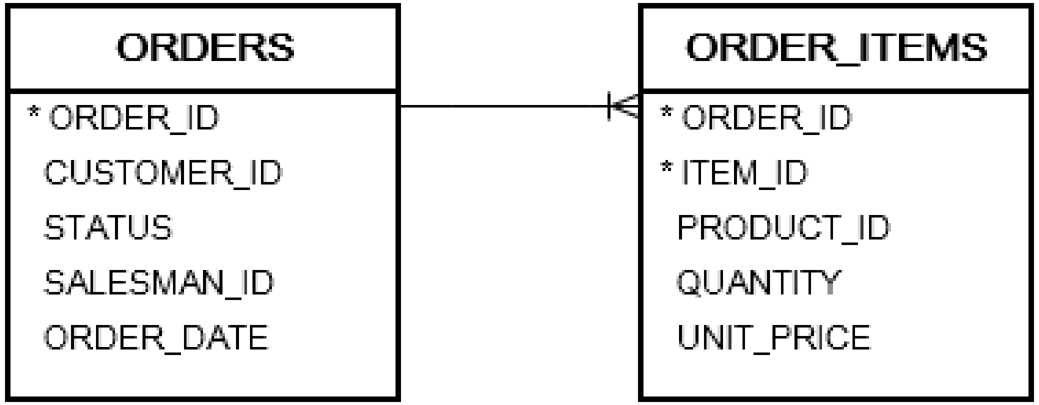
    - Exception-handling section is where we put the exception handler code.



```
CREATE [OR REPLACE] FUNCTION function_name (
    parameter1 datatype
    parameter2 datatype
) RETURN return_type
IS
   -- declarative section
BEGIN
   -- executable section

   RETURN value;

[EXCEPTION]
    [exception-handling section]
END;
```

PL/SQL

- Functions – Example
  - We’ll use the ‘orders’ and ‘order_items’ tables:



ORDERS

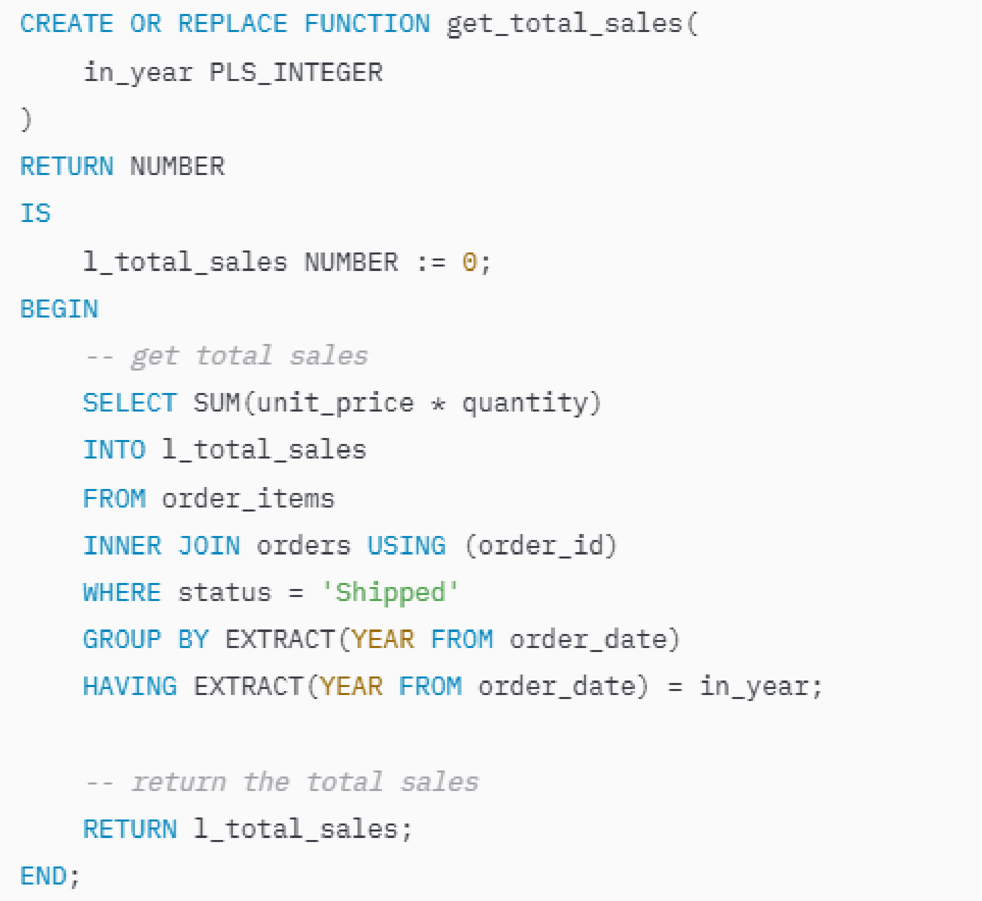
* ORDER_ID
CUSTOMER_ID
STATUS
SALESMAN_ID
ORDER_DATE

ORDER_ITEMS

* ORDER_ID
* ITEM_ID
PRODUCT_ID
QUANTITY
UNIT_PRICE

PL/SQL

- Functions – Example
  - The following example creates a function that calculates total sales by year:



```
CREATE OR REPLACE FUNCTION get_total_sales(
    in_year PLS_INTEGER
)
RETURN NUMBER
IS
    l_total_sales NUMBER := 0;
BEGIN
    -- get total sales
    SELECT SUM(unit_price * quantity)
    INTO l_total_sales
    FROM order_items
    INNER JOIN orders USING (order_id)
    WHERE status = 'Shipped'
    GROUP BY EXTRACT(YEAR FROM order_date)
    HAVING EXTRACT(YEAR FROM order_date) = in_year;

    -- return the total sales
    RETURN l_total_sales;
END;
```