

Collection classes



Collections

Basic operations of collections

- add objects to collection
- remove objects from collection
- search for an object in collection
- retrieve object from collection
- iterate through collection

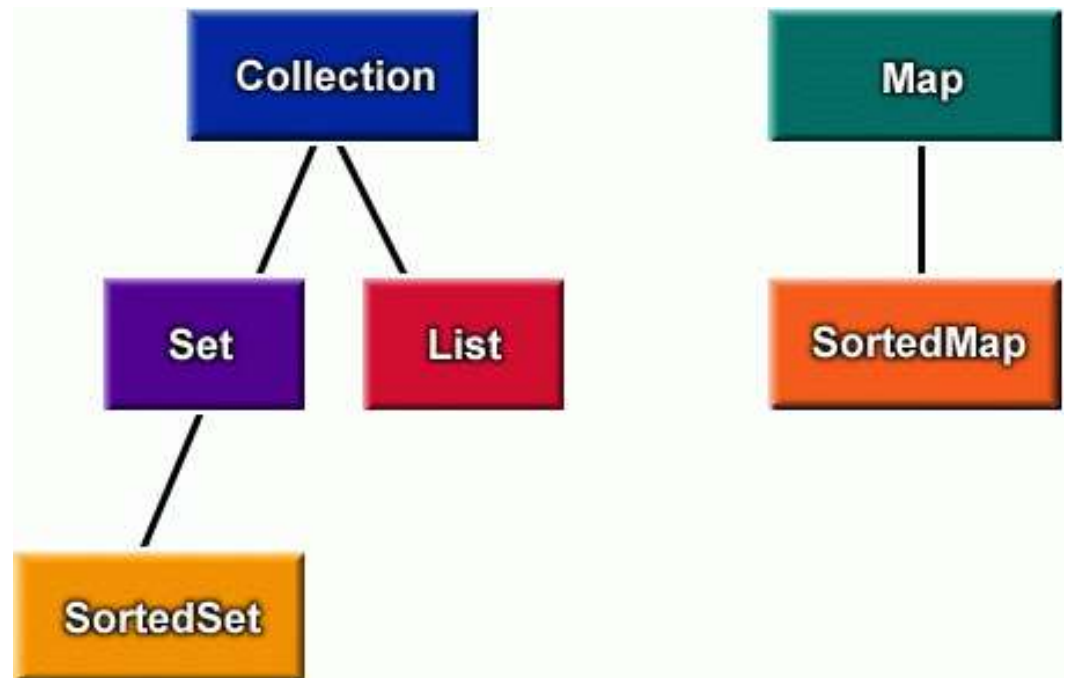
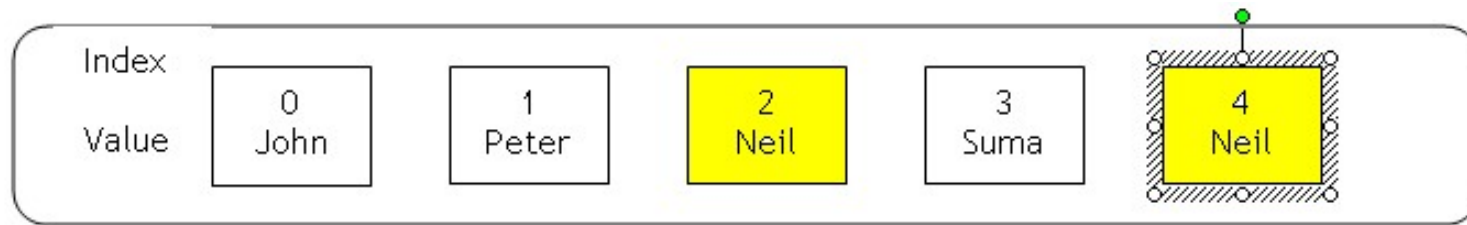
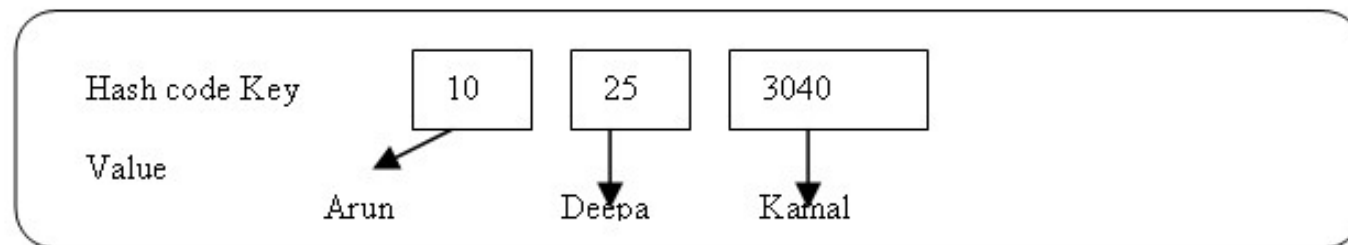
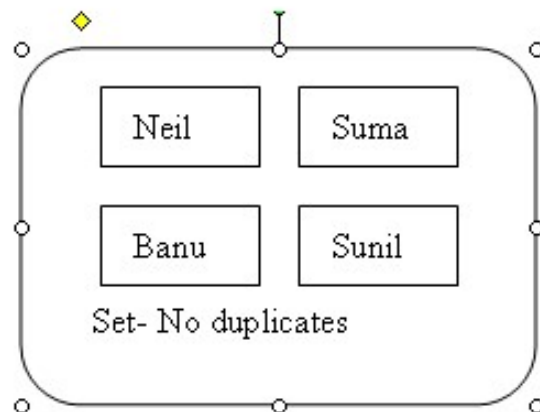


Illustration of List, Set and Map



List - Duplicates allowed



Hash Map - Key generated from RollNo

Collections

interface java.util.Collection

- root interface in the collections hierarchy
- extended by List, Set, Queue
- methods

boolean add(Object)

boolean remove(Object)

boolean addAll(Collection)

boolean removeAll(Collection)

Object[] toArray()

boolean contains(Object)

Iterator iterator()

int size()

Collections

interface java.util.List

- represents ordered collection
- allows duplicates
- implemented by ArrayList, Vector, LinkedList
- additional methods

void add(int index, Object)

Object set(int index, Object)

boolean remove(int index)

boolean addAll(int index, Collection)

Object get(int index)

ListIterator listIterator()

Collections

interface java.util.Set

represents unordered collection

duplicates not allowed

extended by SortedSet

Collections

interface java.util.Map

- represents key-value pairs
- Not part of Collection hierarchy
- extended by SortedMap
- implemented by Hashtable, HashMap
- methods

boolean containsKey(Object key)

boolean containsValue(Object value)

Object get(Object key)

Set keySet()

Collection values()

Object put(Object key, Object value)

Object remove(Object key)

Collections

Classes

- **ArrayList** : Resizable array implementation of **List**
Implements **RandomAccess** interface
- **Vector** : Same as **ArrayList** but threadsafe (legacy class)
- **LinkedList** : Linked list implementation of **List**
provides add, remove at beginning or end
- **HashSet** : Unsorted, unordered implementation of **Set**
uses hashCode()
- **LinkedHashSet** : ordered version of **HashSet**
- **TreeSet** : implementation of **SortedSet** (elements sorted)
- **HashMap** : unsorted **Map** implementation
- **Hashtable** : same as **HashMap** but threadsafe (legacy class)
- **TreeMap** : implementation of **SortedMap**

ArrayList

- Is an implementation of the `List` interface
 - The list automatically grows if elements exceed initial size.
- Has a numeric index
 - Elements are accessed by index.
 - Elements can be inserted based on index.
 - Elements can be overwritten.
- Allows duplicate items

```
List partList = new ArrayList();
partList.add(new Integer(1111));
partList.add(new Integer(2222));
partList.add(new Integer(3333));
partList.add(new Integer(4444)); // ArrayList auto grows
System.out.println("First Part: " +
                    partList.get(0)); // First item
partList.add(0, new Integer(5555)); // Insert at position 1
```

TreeSet: Implementation of Set

```
public class SetExample {  
    public static void main(String[] args){  
        Set set = new TreeSet();  
  
        set.add("one");  
        set.add("two");  
        set.add("three");  
        set.add("three"); // not added, only unique  
  
        Iterator itr= set.iterator();  
        while(itr.hasNext()){  
            System.out.println("Item: " + itr.next());  
        }  
    }  
}
```

Collections

java.util.Iterator interface

Used to iterate through all the elements of the collection

Methods

boolean hasNext()

Object next()

void remove()

Enhanced for loop

- Iterating over collections looks cluttered

```
ArrayList lst = .....;  
Iterator i = lst.iterator();  
While( i.hasNext() )  
    System.out.println(i.next());
```

- Using enhanced for loop we can do the same thing as

```
ArrayList lst = .....;  
for (Object t: lst) )  
    System.out.println(t);
```

TreeMap: Implementation of Map

```
public class MapExample {  
    public static void main(String[] args){  
        Map partList = new TreeMap();  
        partList.put("S001", "Blue Polo Shirt");  
        partList.put("S002", "Black Polo Shirt");  
        partList.put("H001", "Duke Hat");  
  
        partList.put("S002", "Black T-Shirt"); // Overwrite value  
        Set keys = partList.keySet();  
  
        System.out.println("=== Part List ===");  
        for (Object key:keys){  
            System.out.println("Part#: " + key + " " +  
                               partList.get(key));  
        }  
    }  
}
```

Generics

- Provide flexible type safety to your code
- Move many common errors from run time to compile time
- Provide cleaner, easier-to-write code
- Reduce the need for casting with collections
- Are used heavily in the Java Collections API

Simple Store Class Without Generics

```
public class StoreString {  
    private String message;  
  
    public void store(String message){  
        this.message = message;  
    }  
  
    public String get(){  
        return this.message;  
    }  
}
```

```
public class StoreShirt {  
    private Shirt shirt;  
  
    public void store(Shirt shirt){  
        this.shirt = shirt;  
    }  
  
    public Shirt get(){  
        return this.shirt;  
    }  
}
```

Generic Store Class

```
public class StoreAny <T>{  
  
    private T t;  
  
    public void store(T t){  
        this.t = t;  
    }  
  
    public T get(){  
        return this.t;  
    }  
}
```


Generics in Action

Compare the type-restricted objects to their generic alternatives.

```
public static void main(String args[]){  
    StoreString myMessage = new StoreString(); // Type  
    StoreShirt myShirt = new StoreShirt();      // Type
```

```
//Generics
```

```
StoreAny<String> message = new StoreAny<String>();  
StoreAny<Shirt> shirt = new StoreAny<Shirt>();
```

```
myMessage.store("Save this for me"); // Type  
message.store("Save this for me"); // Generic  
shirt.store("Hello"); // compilation error
```

```
}
```

Generic Collections

- Generic collections used to hold homogeneous data
- They are type safe
- No typecasting required while extracting elements

```
List<Emp> list = new ArrayList<Emp>();  
list.add(new Emp());  
Emp e = list.get(0);
```

```
HashMap<String, Mammal> map =  
    new HashMap<String, Mammal>();  
map.put("wombat", new Mammal("wombat"));  
Mammal w = map.get("wombat");
```

Ordering Collections

- The `Comparable` and `Comparator` interfaces are used to sort collections.
 - Both are implemented by using generics.
- Using the `Comparable` interface:
 - Overrides the `compareTo` method
 - Provides only one sort option
- The `Comparator` interface:
 - Is implemented by using the `compare` method
 - Enables you to create multiple `Comparator` classes
 - Enables you to create and use numerous sorting options

Comparable: Example

```
public class Student implements Comparable<Student>{
    private String name;
    private long id = 0;
    private double gpa = 0.0;

    public Student(String name, long id, double gpa){
        // Additional code here
    }

    // getters and setters

    public int compareTo(Student s){
        if(this.id < s.id)
            return -1;
        if (this.id > s.id)
            return 1;
        return 0;
    }
}
```

Comparable : Example

```
public class TestComparable {  
    public static void main(String[] args){  
        Set<Student> studentList = new TreeSet<Student>();  
  
        studentList.add(new Student("Thomas Jefferson", 1111, 3.8));  
        studentList.add(new Student("John Adams", 2222, 3.9));  
        studentList.add(new Student("George Washington", 3333, 3.4));  
  
        for(Student student:studentList){  
            System.out.println(student);  
        }  
    }  
}
```

Comparator - Example

```
class IdComp implements Comparator<Student> {  
    public int compare(Student a, Student b){  
        return a.getId () - b.getId( );  
    }  
}
```

```
class NameComp implements Comparator<Student> {  
    public int compare(Student a, Student b){  
        return a.getName().compareTo( b.getName( ) );  
    }  
}
```

.....

```
TreeSet <Student> ts1 = new TreeSet<Student> ( new IdComp() ); //sorted on id  
TreeSet <Student> ts2 = new TreeSet<Student> ( new NameComp() ); // sorted on name
```

Collections class

- Collections class is used exclusively with static methods that operate on or return collections
- provides some convenience methods that are highly useful in working with Java collections
- Some of the methods of Collections:
 - **static <T> int binarySearch(List, <T> T key)** - Searches the list for the specified object using the binary search algorithm.
 - **static<T> void copy(List <T> dest, List <T> src)** - Copies all of the elements from one list into another
 - **static<T> void sort(List <T> list)** - Sorts the list into ascending order, according to the **natural ordering** of its elements
 - **static<T> void sort(List <T> list , Comparator<T> c)** - Sorts the list according to the order induced by the specified comparator
 - **static void swap(List <T> list , int i, int j)** - Swaps the elements at the specified positions in the list

SQL

Relating Multiple Tables

- Each row of data in a table is uniquely identified by a primary key.
- You can logically relate data from multiple tables using foreign keys.

Table name: EMPLOYEES

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	DEPARTMENT_ID
100	Steven	King	90
101	Neena	Kochhar	90
102	Lex	De Haan	90
103	Alexander	Hunold	60
104	Bruce	Ernst	60
107	Diana	Lorentz	60
124	Kevin	Mourgos	50
141	Trenna	Rajs	50
142	Curtis	Davies	50

Primary key

Foreign key

Table name: DEPARTMENTS

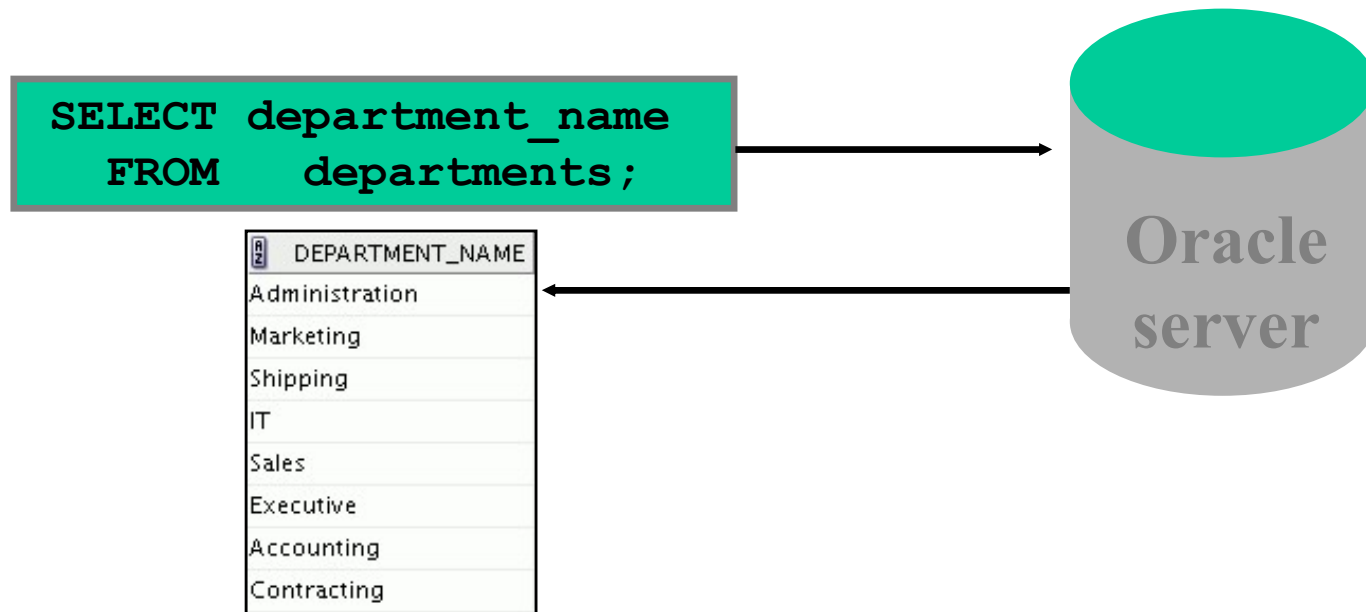
DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
50	Shipping	124	1500
60	IT	103	1400
80	Sales	149	2500
90	Executive	100	1700
110	Accounting	205	1700
190	Contracting	(null)	1700

Primary key

Using SQL to Query Your Database

- **Structured query language (SQL) is:**

- **The ANSI standard language for operating relational databases**
- **Efficient, easy to learn, and use**
- **Functionally complete (With SQL, you can define, retrieve, and manipulate data in the tables.)**



SQL Statements

- SELECT
- INSERT
- UPDATE
- DELETE
- MERGE

Data manipulation language (DML)

- CREATE
- ALTER
- DROP
- RENAME
- TRUNCATE
- COMMENT

Data definition language (DDL)

- GRANT
- REVOKE

Data control language (DCL)

- COMMIT
- ROLLBACK
- SAVEPOINT

Transaction control

CREATE TABLE Statement

- You specify:
 - The table name
 - The column name, column data type, and column size

```
CREATE TABLE tableName (  
    column datatype [, ...]  
    [Constraint specification  
);
```

- To see the table structure
 - Describe *tableName*

Creating Tables

- Create the table:

```
CREATE TABLE ord  
  ( ID DECIMAL(3),  
    quantity DECIMAL(3)  
  );
```

Data Types

Java DB, Derby	Format
SMALLINT	
INTEGER	
DECIMAL(p,s) or NUMERIC(p,s)	
FLOAT	
FLOAT	
SMALLINT	
SMALLINT	
VARCHAR	Single quotes
CHAR(1)	Single quotes
DATE	yyyy-mm-dd, mm/dd/yyyy, dd.mm.yyyy
TIME	hh:mm[:ss] , hh.mm[:ss]
TIMESTAMP	yyyy-mm-dd hh:mm:ss[.nnnnnn]

Including Constraints

Constraints enforce rules at the table level.

Constraints prevent the deletion of a table if there are dependencies.

The following constraint types are valid:

NOT NULL

UNIQUE

PRIMARY KEY

FOREIGN KEY

CHECK

CREATE TABLE: Example

```
CREATE TABLE customers

( customer_id          DECIMAL(6)  primary key
, cust_first_name      VARCHAR(20)
  CONSTRAINT fname_nn NOT NULL
, cust_last_name       VARCHAR(20)
  CONSTRAINT lname_nn NOT NULL
, cust_address         varchar(50)
, city_code            char(2)
, language             VARCHAR(3)
, territory            VARCHAR(30)
, credit_limit         NUMERIC(9,2)
, cust_email           VARCHAR(30)
, account_mgr_id       NUMERIC(6)
,   CONSTRAINT ck_credit_limit
                      CHECK (credit_limit <= 5000)
,   CONSTRAINT city_fk
  foreign key(city_code)
    references city(city_code)
) ;
```


Basic SELECT Statement




```
SELECT * | { [DISTINCT] column | expression [alias] , ... }  
FROM      table;
```

SELECT identifies
the columns to be
displayed.

FROM identifies
the table
containing those
columns.



Selecting All Columns

```
SELECT *  
FROM inventories ;
```

	 PRODUCT_ID	 WAREHOUSE_ID	 QUANTITY_ON_HAND
1	3108	8	122
2	3110	8	123
3	3112	8	123
4	3117	8	124
5	3124	8	125
6	3127	8	125
7	3129	8	126
8	3134	8	149
9	3139	8	150
10	3140	8	150
11	3143	8	151

Selecting Specific Columns

```
SELECT product_id, quantity_on_hand  
FROM inventories ;
```

	 PRODUCT_ID	 QUANTITY_ON_HAND
1	3108	122
2	3110	123
3	3112	123
4	3117	124
5	3124	125
6	3127	125
7	3129	126
8	3134	149
9	3139	150
10	3140	150
11	3143	151




Arithmetic Expressions

- Create expressions with number and date data by using arithmetic operators.

Operator	Description
+	Add
-	Subtract
*	Multiply
/	Divide

Using Arithmetic Operators

```
SELECT product_id, quantity_on_hand, quantity_on_hand+200  
FROM inventories ;
```

	 PRODUCT_ID	 QUANTITY_ON_HAND	 QUANTITY_ON_HAND+200
1	3108	122	322
2	3110	123	323
3	3112	123	323
4	3117	124	324
5	3124	125	325
6	3127	125	325
7	3129	126	326
8	3134	149	349
9	3139	150	350
10	3140	150	350
11	3143	151	351

What is a NULL value ?

What is a NULL value ?

If a row does not have an entry for a particular column, that value is said to be NULL..

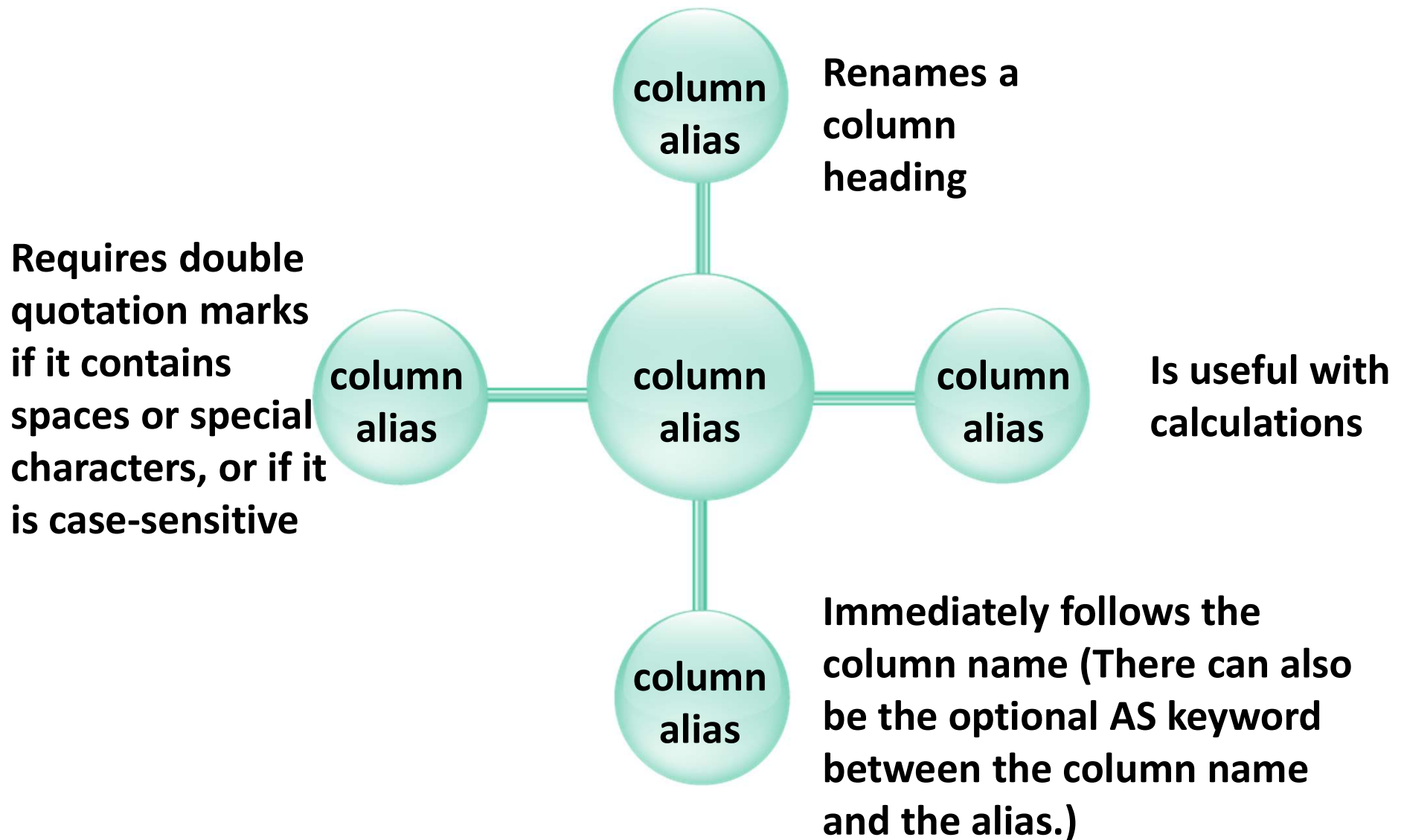
What is a NULL value ?

It is the absence of any character, zero, blank space etc.

What is a NULL value ?



Arithmetic operations on a NULL value always return a NULL value.

Defining a Column Alias28





Using Column Aliases

```
SELECT product_id AS Product , quantity_on_hand Quantity  
FROM inventories ;
```

	 PRODUCT	 QUANTITY
1	3108	122
2	3110	123
3	3112	123
4	3117	124

• • •

```
SELECT order_id "Order " , order_date) "Date of Order"  
FROM orders ;
```

	 Order	 Date of Order
1	2458	17-AUG-99
2	2397	20-NOV-99
3	2454	03-OCT-99
4	2354	15-JUL-00

• • •

Concatenation Operator

- A concatenation operator:
 - Links columns or character strings to other columns
 - Is represented by two vertical bars (||)
 - Creates a resultant column that is a character expression

```
SELECT first_name || last_name AS "NAME"  
FROM customers ;
```

	NAME
1	EllenAbel
2	SundarAnde
3	MozheAtkinson
4	DavidAustin
5	HermannBaer
6	ShelliBaida
7	AmitBanda
8	ElizabethBates


Limiting Rows Using a Selection

EMPLOYEES

	EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
1	200	Whalen	AD_ASST	10
2	201	Hartstein	MK_MAN	20
3	202	Fay	MK_REP	20
4	205	Higgins	AC_MGR	110
5	206	Gietz	AC_ACCOUNT	110

...

**“retrieve all
employees in
department 90”**



	EMPLOYEE_ID	LAST_NAME	JOB_ID	DEPARTMENT_ID
1	100	King	AD_PRES	90
2	101	Kochhar	AD_VP	90
3	102	De Haan	AD_VP	90

Limiting the Rows That Are Selected




- Restrict the rows that are returned by using the :
- **WHERE** clause

```
SELECT * | { [DISTINCT] column | expression [alias] , ... }  
FROM    table  
[WHERE condition(s) ] ;
```

- The **WHERE** clause follows the **FROM** clause.

Using the WHERE Clause

```
SELECT order_id, order_date, order_status  
FROM orders  
WHERE order_status = 1 ;
```

	 ORDER_ID	 ORDER_DATE	 ORDER_STATUS
1	2397	20-NOV-99 04.11.54.696211000 AM	1
2	2454	03-OCT-99 05.19.34.678340000 AM	1
3	2421	13-MAR-99 09.23.54.562432000 AM	1
4	2431	14-SEP-98 06.33.04.763452000 PM	1
5	2439	31-AUG-99 09.49.37.811132000 PM	1
6	2444	28-JUL-99 01.52.27.462632000 AM	1

Comparison Operators

Operator	Meaning
=	Equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
<>	Not equal to
BETWEEN ...AND...	Between two values (inclusive)
IN (set)	Match any of a list of values
LIKE	Match a character pattern
IS NULL	Is a null value

Using Comparison Operators

```
SELECT order_id, order_date
FROM orders
WHERE order_id <= 2400 ;
```

	ORDER_ID	ORDER_DATE
1	2354	15-JUL-00 05.48.23.234567000 AM
2	2355	26-JAN-98 10.52.51.962632000 PM
3	2356	26-JAN-00 10.52.41.934562000 PM
4	2357	09-JAN-98 09.49.44.123456000 AM
5	2358	09-JAN-00 06.33.12.654278000 AM
6	2359	09-JAN-98 11.04.13.112233000 AM

• • •

Range Conditions Using the BETWEEN Operator

- Use the **BETWEEN** operator to display rows based on a range of values:

```
SELECT product_id, quantity_on_hand  
FROM inventories  
WHERE product_id BETWEEN 3100 AND 3108;
```



Lower limit



Upper limit

	PRODUCT_ID	QUANTITY_ON_HAND
1	3108	122
2	3108	110
3	3108	194
4	3108	170
5	3108	146

Membership Condition Using the IN Operator

- Use the IN operator to test for values in a list:

```
SELECT order_id, order_mode, order_status  
FROM orders  
WHERE order_id IN (2458, 2397, 2454) ;
```

	ORDER_ID	ORDER_MODE	ORDER_STATUS
1	2397	direct	1
2	2454	direct	1
3	2458	direct	0

Pattern Matching Using the LIKE Operator

Use the LIKE operator to perform wildcard searches of valid search string values.

Search conditions can contain either literal characters or numbers:

- % denotes zero or many characters.
- _ denotes one character.

```
SELECT first_name  
FROM employees  
WHERE first_name LIKE 'S%' ;
```

Combining Wildcard Characters

- You can combine the two wildcard characters (% , _) with literal characters for pattern matching:

```
SELECT last_name  
FROM employees  
WHERE last_name LIKE '_o%' ;
```

	LAST_NAME
1	Kochhar
2	Lorentz
3	Mourgos

- You can use the ESCAPE identifier to search for the actual % and _ symbols.

Using the NULL Conditions

Test for nulls with the IS NULL operator.

```
SELECT  order_ID, order_status, sales_rep_id  
FROM    orders  
WHERE   sales_rep_id IS NULL;
```

	ORDER_ID	ORDER_STATUS	SALES_REP_ID
1	2355	8	(null)
2	2356	5	(null)
3	2359	9	(null)
4	2361	8	(null)
5	2362	4	(null)
6	2363	0	(null)




Defining Conditions Using the Logical Operators

Operator	Meaning
AND	Returns TRUE if <i>both</i> component conditions are true
OR	Returns TRUE if <i>either</i> component condition is true
NOT	Returns TRUE if the condition is false

Using the AND Operator

AND requires both the component conditions to be true:

```
SELECT  order_mode, order_status, customer_id  
FROM    orders  
WHERE   order_mode = 'direct '  
AND     customer_id = 103;
```

	 ORDER_MODE	 ORDER_STATUS	 CUSTOMER_ID
1	direct	1	103
2	direct	4	103

Using the OR Operator

OR requires either component condition to be true:

```
SELECT  order_id, order_status, order_total  
FROM    orders  
WHERE   order_status = 0  
        OR order_total >= 100000 ;
```

	ORDER_ID	ORDER_STATUS	ORDER_TOTAL
1	2458	0	70647.34
2	2354	0	46257
3	2434	8	242458.25
4	2361	8	120131.3
5	2363	0	10082.3
6	2367	10	144054.8
7	2369	0	11097.4
8	2375	2	103834.4
9	2385	4	295892
10	2388	4	282694.3
11	2399	0	25270.3

Using the NOT Operator

```
SELECT order_id, order_status, order_total  
FROM orders  
WHERE order_status  
      NOT IN (0,1,2,3) ;
```

	ORDER_ID	ORDER_STATUS	ORDER_TOTAL
1	2357	5	59872.4
2	2394	5	21863
3	2435	6	62303
4	2455	7	14087.5
5	2379	8	17848.2
6	2396	8	34930
7	2434	8	242458.25
8	2436	8	6394.8
9	2446	8	93570.57
10	2447	8	33893.6
11	2432	10	10523

Using the ORDER BY Clause

Sort the retrieved rows with the **ORDER BY** clause:

- **ASC:** Ascending order, default
- **DESC:** Descending order

The **ORDER BY** clause comes last in the **SELECT** statement:

```
SELECT order_id, order_date, order_status  
FROM orders  
ORDER BY order_date ;
```

	ORDER_ID	ORDER_DATE	ORDER_STATUS
1	2442	27-JUL-90 11.52.59.662632000 PM	9
2	2445	28-JUL-90 03.04.38.362632000 AM	8
3	2418	21-MAR-96 05.48.21.862632000 AM	4
4	2357	09-JAN-98 09.49.44.123456000 AM	5

Sorting

Sorting in descending order:

```
SELECT order_id, round(order_date), order_status  
FROM orders  
ORDER BY order_date desc;
```

Sorting by column alias:

```
SELECT order_id, round(order_date), order_status "Order Status"  
FROM orders  
ORDER BY order_date desc ;
```

Group Functions

Group functions operate on sets of rows to give one result per group.

EMPLOYEES

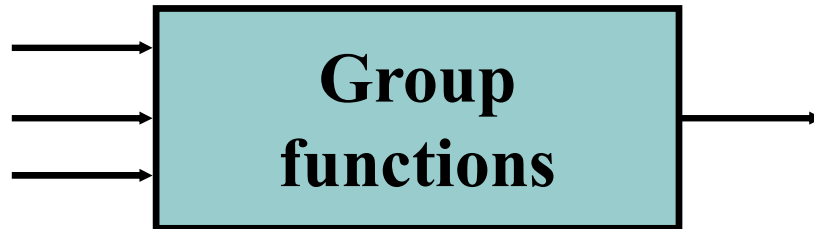
	DEPARTMENT_ID	SALARY
1	10	4400
2	20	13000
3	20	6000
4	110	12000
5	110	8300
6	90	24000
7	90	17000
8	90	17000
9	60	9000
10	60	6000
...		
18	80	11000
19	80	8600
20	(null)	7000

**Maximum
salary in
EMPLOYEES
table**

MAX(SALARY)
24000

Types of Group Functions

- AVG
- COUNT
- MAX
- MIN
- SUM



Group Functions: Syntax

```
SELECT group_function(column), ...  
FROM   table  
[WHERE condition]  
[ORDER BY column];
```

Using the AVG and SUM Functions

You can use AVG and SUM for numeric data.

You can use MIN and MAX for numeric, character, and date data types.

```
SELECT  AVG(order_total), MAX(order_total),  
        MIN (order_total), SUM( order_total)  
FROM    orders;
```

	AVG(ORDER_TOTAL)	MAX(ORDER_TOTAL)	MIN(ORDER_TOTAL)	SUM(ORDER_TOTAL)
1	44628.44125	295892	5451	3570275.3

Using the COUNT Function

- **COUNT(*)** returns the number of rows in a table:

```
SELECT count(*)  
FROM inventories  
WHERE warehouse_id = 8;
```

	AZ	COUNT(*)
1		186

- **COUNT(expr)** returns the number of rows with non-null values for *expr*:

```
SELECT COUNT(sales_rep_id)  
FROM orders  
WHERE order_status <=3;
```

	AZ	COUNT(SALES_REP_ID)
1		19

Creating Groups of Data

EMPLOYEES

	DEPARTMENT_ID	SALARY
1	10	4400
2	20	13000
3	20	6000
4	50	2500
5	50	2600
6	50	3100
7	50	3500
8	50	5800
9	60	9000
10	60	6000
11	60	4200
12	80	11000
13	80	8600

...

18	110	8300
19	110	12000
20	(null)	7000

4400

9500

3500

6400

10033

Average salary in
the
EMPLOYEES table

	DEPARTMENT_ID	AVG(SALARY)
1	(null)	7000
2	20	9500
3	90	19333.333333333333...
4	110	10150
5	50	3500
6	80	10033.333333333333...
7	10	4400
8	60	6400

Creating Groups of Data: GROUP BY Clause Syntax

- You can divide rows in a table into smaller groups by using the **GROUP BY** clause.

```
SELECT      column, group_function(column)
FROM        table
[WHERE      condition]
[GROUP BY  group_by_expression]
[ORDER BY  column];
```


Using the GROUP BY Clause

- **All the columns in the SELECT list that are not in group functions must be in the GROUP BY clause.**

```
SELECT warehouse_id, AVG(quantity_on_hand)
FROM inventories
GROUP BY warehouse_id ;
```

[illegible]

Using the GROUP BY Clause

- The GROUP BY column does not have to be in the SELECT list.

```
SELECT  AVG(order_total)
FROM    orders
GROUP BY order_status ;
```

[illegible]

Illegal Queries Using Group Functions

- Any column or expression in the SELECT list that is not an aggregate function must be in the GROUP BY clause:

```
SELECT department_id, COUNT(last_name)
FROM employees;
```

ORA-00937: not a single-group group function
00937. 00000 - "not a single-group group function"

A GROUP BY clause must be added to count the last names for each department_id.

```
SELECT department_id, job_id, COUNT(last_name)
FROM employees
GROUP BY department_id;
```

ORA-00979: not a GROUP BY expression
00979. 00000 - "not a GROUP BY expression"

Either add job_id in the GROUP BY or remove the job_id column from the SELECT list.

Restricting Group Results

EMPLOYEES

	DEPARTMENT_ID	SALARY
1	10	4400
2	20	13000
3	20	6000
4	50	2500
5	50	2600
6	50	3100
7	50	3500
8	50	5800
9	60	9000
10	60	6000
11	60	4200
12	80	11000
13	80	8600
...		
18	110	8300
19	110	12000
20	(null)	7000

The maximum salary
per department when
it is

	DEPARTMENT_ID	MAX(SALARY)
1	20	13000
2	90	24000
3	110	12000
4	80	11000

Restricting Group Results with the HAVING Clause

When you use the HAVING clause, the server restricts groups as follows:

Rows are grouped.

The group function is applied.

Groups matching the HAVING clause are displayed.

```
SELECT      column, group_function
FROM        table
[WHERE      condition]
[GROUP BY  group_by_expression]
[HAVING     group_condition]
[ORDER BY  column] ;
```

Using the HAVING Clause

```
SELECT warehouse_id, AVG(quantity_on_hand)
FROM inventories
GROUP BY warehouse_id
HAVING MAX (quantity_on_hand) > 130 ;
```

[illegible]

Using the HAVING Clause

```
SELECT    job_id, SUM(salary) PAYROLL
FROM      employees
WHERE     job_id NOT LIKE '%REP%'
GROUP BY  job_id
HAVING    SUM(salary) > 13000
ORDER BY  SUM(salary);
```

	 JOB_ID	 PAYROLL
1	IT_PROG	19200
2	AD_PREP	24000
3	AD_VP	34000


Obtaining Data from Multiple Tables

EMPLOYEES

	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
1	200	Whalen	10
2	201	Hartstein	20
3	202	Fay	20
...			
18	174	Abel	80
19	176	Taylor	80
20	178	Grant	(null)

DEPARTMENTS

	DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
1	10	Administration	1700
2	20	Marketing	1800
3	50	Shipping	1500
4	60	IT	1400
5	80	Sales	2500
6	90	Executive	1700
7	110	Accounting	1700
8	190	Contracting	1700



	EMPLOYEE_ID	DEPARTMENT_ID	DEPARTMENT_NAME
1	200	10	Administration
2	201	20	Marketing
3	202	20	Marketing
4	124	50	Shipping
...			
18	205	110	Accounting
19	206	110	Accounting

Creating Joins with the ON Clause

The join condition for the natural join is basically an equijoin of all columns with the same name.

Use the ON clause to specify arbitrary conditions or specify columns to join.

The join condition is separated from other search conditions.

The ON clause makes code easy to understand.

Retrieving Records with the ON Clause

```
SELECT e.order_status, e.customer_id, e.order_id,  
       d.order_id, d.quantity  
FROM   orders e JOIN order_items d  
ON     (e.order_id = d.order_id) ;
```

	ORDER_STATUS	CUSTOMER_ID	ORDER_ID	ORDER_ID_1	QUANTITY
1	8	104	2355	2355	200
2	5	105	2356	2356	38
3	5	108	2357	2357	140
4	2	105	2358	2358	9
5	9	106	2359	2359	1
6	8	108	2361	2361	180
7	4	109	2362	2362	200
8	0	144	2363	2363	9
9	4	145	2364	2364	6

INNER Versus OUTER Joins

A join between tables that returns rows on exact match is inner join

A join between two tables that returns the results of the INNER join as well as the unmatched rows from the left (or right) table is called a left (or right) OUTER join.

LEFT OUTER JOIN

```
SELECT e.last_name, e.department_id,  
       d.department_name  
FROM   employees e LEFT OUTER JOIN departments d  
ON     (e.department_id = d.department_id) ;
```

	A Z LAST_NAME	A Z DEPARTMENT_ID	A Z DEPARTMENT_NAME
1	Whalen	10	Administration
2	Fay	20	Marketing
3	Hartstein	20	Marketing
4	Vargas	50	Shipping
5	Matos	50	Shipping

...

16	Kochhar	90	Executive
17	King	90	Executive
18	Gietz	110	Accounting
19	Higgins	110	Accounting
20	Grant	(null)	(null)

RIGHT OUTER JOIN

```
SELECT e.last_name, d.department_id,  
       d.department_name  
FROM   employees e RIGHT OUTER JOIN departments d  
ON     (e.department_id = d.department_id) ;
```

	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
1	Whalen	10	Administration
2	Hartstein	20	Marketing
3	Fay	20	Marketing
4	Davies	50	Shipping
5	Vargas	50	Shipping
6	Rajs	50	Shipping
7	Mourgos	50	Shipping
8	Matos	50	Shipping

...

18	Higgins	110	Accounting
19	Gietz	110	Accounting
20	(null)	190	Contracting

FULL OUTER JOIN

```
SELECT e.last_name, d.department_id,  
       d.department_name  
FROM   employees e FULL OUTER JOIN departments d  
ON     (e.department_id = d.department_id) ;
```

	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
1	Whalen	10	Administration
2	Hartstein	20	Marketing
3	Fay	20	Marketing
4	Higgins	110	Accounting

...

17	Zlotkey	80	Sales
18	Abel	80	Sales
19	Taylor	80	Sales
20	Grant	(null)	(null)
21	(null)	190	Contracting

Using a Subquery to Solve a Problem

Who has a credit limit than Charlie Pacino's?

Main query:



Which customers have credit limit greater than Charlie Pacino's credit limit?

Subquery:



What is Charlie Pacino's credit limit?




Subquery Syntax

```
SELECT  select_list
FROM    table
WHERE   expr operator
        (SELECT    select_list
         FROM      table);
```

- The subquery (inner query) executes *before* the main query (outer query).
- The result of the subquery is used by the main query.

Using a Subquery

```
SELECT cust_first_name || cust_last_name
FROM customers
WHERE credit_limit <
      (Select credit_limit
       FROM customers
       WHERE cust_first_name = 'Charlie'
       AND cust_last_name = 'Pacino');
```

	 CUSTOMER_ID	 UNIT_PRICE	 WAREHOUSE_ID
1	105	199.1	9
2	105	199.1	2
3	105	199.1	4
4	105	199.1	6
5	105	199.1	8
6	105	226.6	9
7	105	226.6	2

Data Manipulation Language

A DML statement is executed when you:

- **Add new rows to a table**
- **Modify existing rows in a table**
- **Remove existing rows from a table**

***A transaction* consists of a collection of DML statements that form a logical unit of work.**

Adding a New Row to a Table

DEPARTMENTS

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	10	Administration	200	1700
2	20	Marketing	201	1800
3	50	Shipping	124	1500
4	60	IT	103	1400
5	80	Sales	149	2500
6	90	Executive	100	1700
7	110	Accounting	205	1700
8	190	Contracting	(null)	1700

70 Public Relations	100	1700
---------------------	-----	------

**New
row**

**Insert new row
into the
DEPARTMENTS table.**

	DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
1	70	Public Relations	100	1700
2	10	Administration	200	1700
3	20	Marketing	201	1800
4	50	Shipping	124	1500
5	60	IT	103	1400
6	80	Sales	149	2500
7	90	Executive	100	1700
8	110	Accounting	205	1700
9	190	Contracting	(null)	1700

INSERT Statement Syntax

- Add new rows to a table by using the INSERT statement:

```
INSERT INTO table [(column [, column...])]  
VALUES          (value [, value...]);
```

- With this syntax, only one row is inserted at a time.

Inserting New Rows

Insert a new row containing values for each column.

List values in the default order of the columns in the table.

Optionally, list the columns in the INSERT clause.

```
INSERT INTO order_items (order_id,  
line_item_id, product_id, unit_price, quantity)  
VALUES (2355, 1, 3108, 46, 200) ;
```

Enclose character and date values within single quotation marks.

Changing Data in a Table

EMPLOYEES

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY	MANAGER_ID	COMMISSION_PCT	DEPARTMENT_ID
100	Steven	King	24000	(null)	(null)	90
101	Neena	Kochhar	17000	100	(null)	90
102	Lex	De Haan	17000	100	(null)	90
103	Alexander	Hunold	9000	102	(null)	60
104	Bruce	Ernst	6000	103	(null)	60
107	Diana	Lorentz	4200	103	(null)	60
124	Kevin	Mourgos	5800	100	(null)	50

Update rows in the EMPLOYEES table: →

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY	MANAGER_ID	COMMISSION_PCT	DEPARTMENT_ID
100	Steven	King	24000	(null)	(null)	90
101	Neena	Kochhar	17000	100	(null)	90
102	Lex	De Haan	17000	100	(null)	90
103	Alexander	Hunold	9000	102	(null)	80
104	Bruce	Ernst	6000	103	(null)	80
107	Diana	Lorentz	4200	103	(null)	80
124	Kevin	Mourgos	5800	100	(null)	50

UPDATE Statement Syntax

- **Modify existing values in a table with the UPDATE statement:**

```
UPDATE      table  
SET         column = value [, column = value, ...]  
[WHERE      condition];
```

- **Update more than one row at a time (if required).**

Updating Rows in a Table

- Values for a specific row or rows are modified if you specify the **WHERE** clause:

```
UPDATE inventories  
SET warehouse_id = 7  
WHERE product_id = 3108 ;
```

```
1 rows updated
```

- Values for all the rows in the table are modified if you omit the **WHERE** clause:

```
UPDATE inventories  
SET warehouse_id = 7 ;
```

- Specify **SET *column_name* = NULL** to update a column value to **NULL**.

DELETE Statement

- You can remove existing rows from a table by using the **DELETE** statement:

```
DELETE [FROM] table  
[WHERE condition];
```

Deleting Rows from a Table

- Specific rows are deleted if you specify the **WHERE** clause:

```
DELETE FROM runreport  
WHERE comments = ' Editing Report ' ;
```

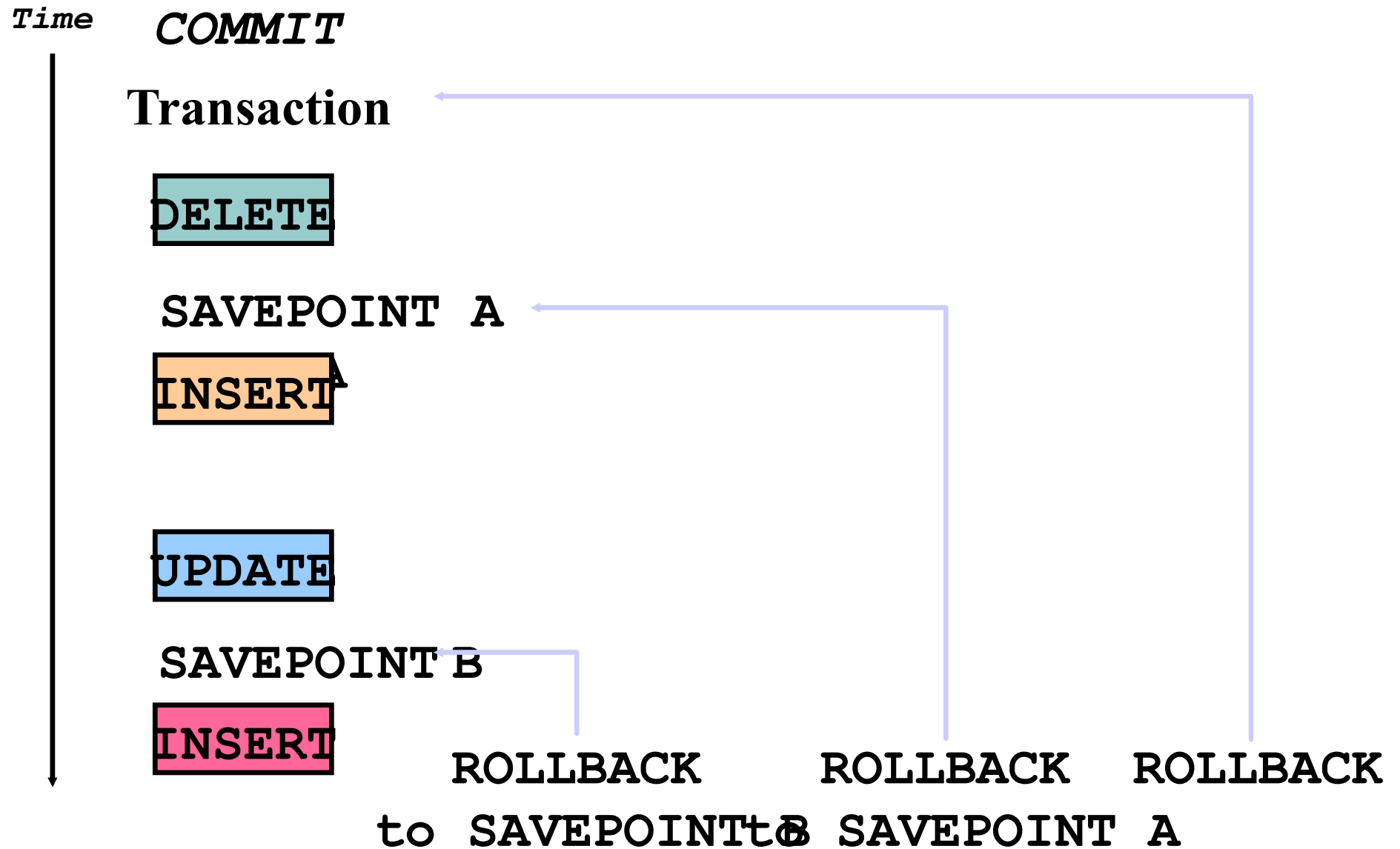
- All rows in the table are deleted if you omit the **WHERE** clause:

```
DELETE FROM copy_emp;
```

Database Transactions

- **A database transaction is defined as atomic activity that may have multiple DML statements**
- **COMMIT statement used to make changes permanent once all the DML statements are successful**
- **ROLLBACK statement is used to undo changes to the database if any of the DML statements fails**

Explicit Transaction Control Statements



Rolling Back Changes to a Marker

- Create a marker in the current transaction by using the **SAVEPOINT** statement.
- Roll back to that marker by using the **ROLLBACK TO SAVEPOINT** statement.

```
UPDATE . . .  
SAVEPOINT update_done;  
  
INSERT . . .  
ROLLBACK TO update_done;
```

Committing Data

- **Make the changes:**

```
DELETE FROM inventories  
WHERE product_id = 2458 ;
```

```
INSERT INTO Inventories  
VALUES (2670, 6, 159);
```

- **Commit the changes:**

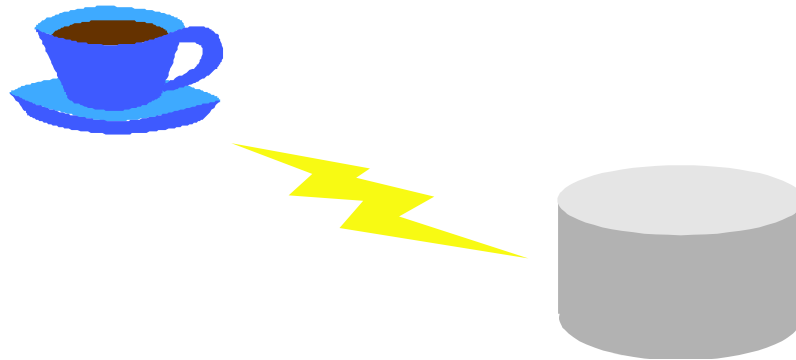
```
COMMIT ;
```

JDBC



Introduction to JDBC

- JDBC is a standard interface for connecting to relational databases from Java.
- The JDBC classes and interfaces are in the **java.sql** package.



java.sql package

Driver

DriverManager

DriverPropertyInfo

Connection

Statement

PreparedStatement

ResultSet

RowSet

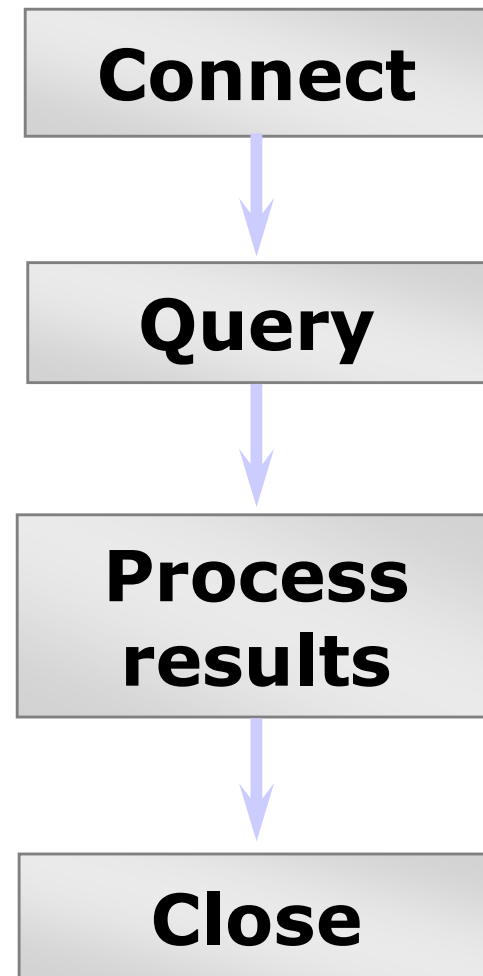
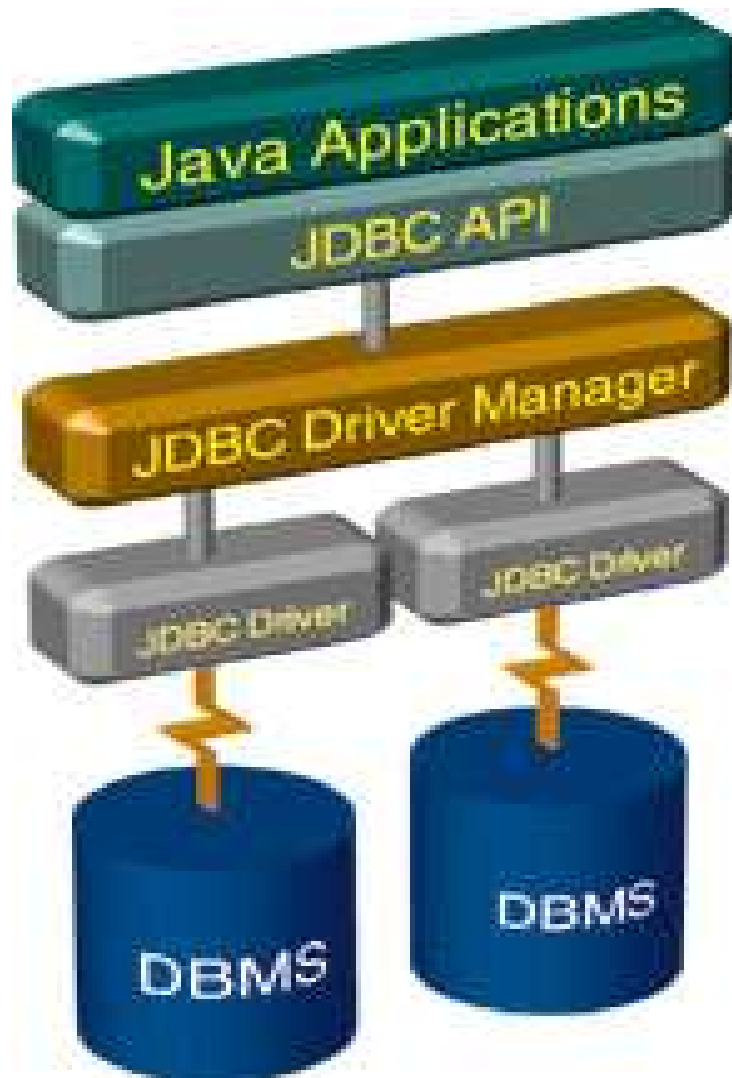
SQLException

SQLWarning

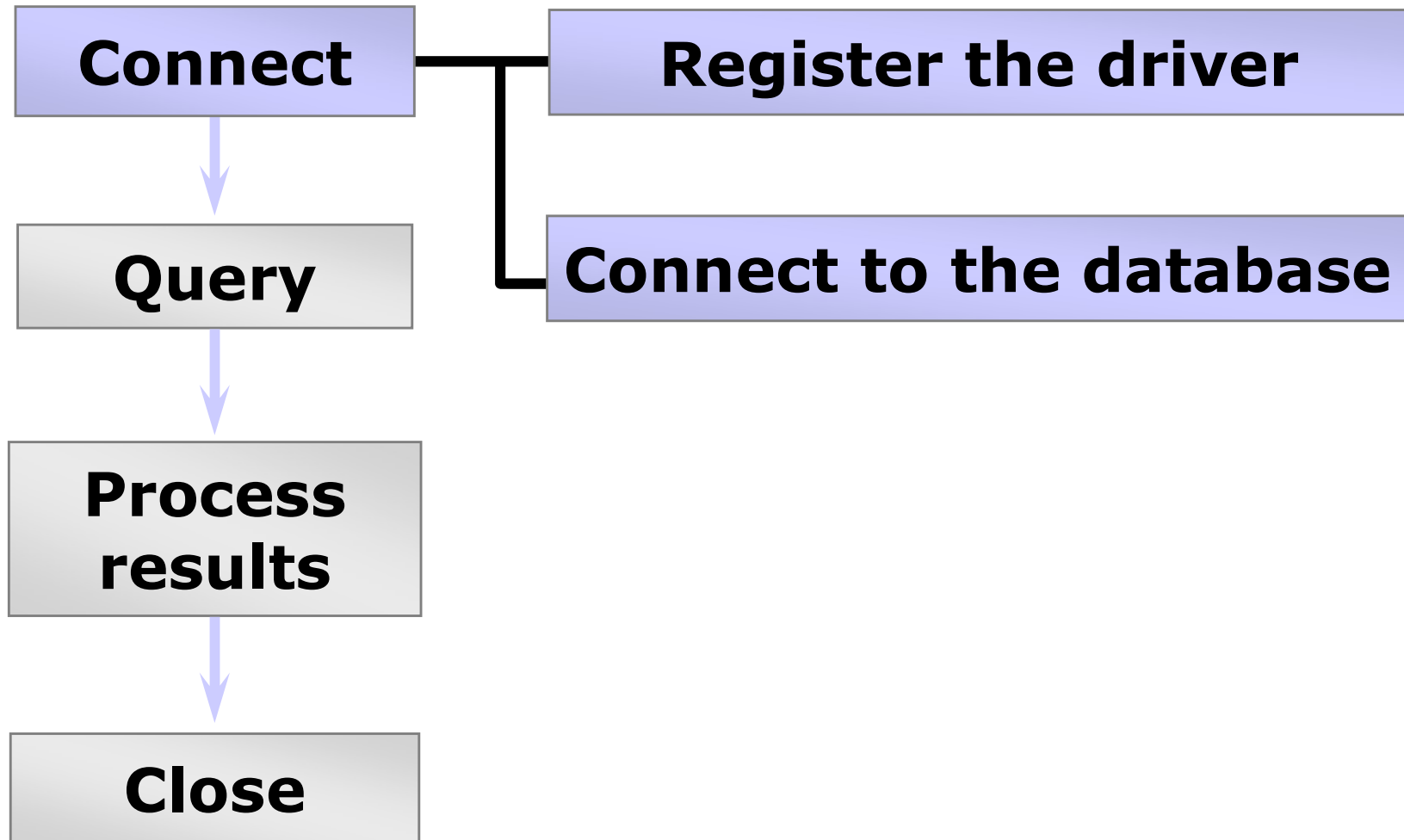
DatabaseMetaData

ResultSetMetaData

Architecture & Querying with JDBC

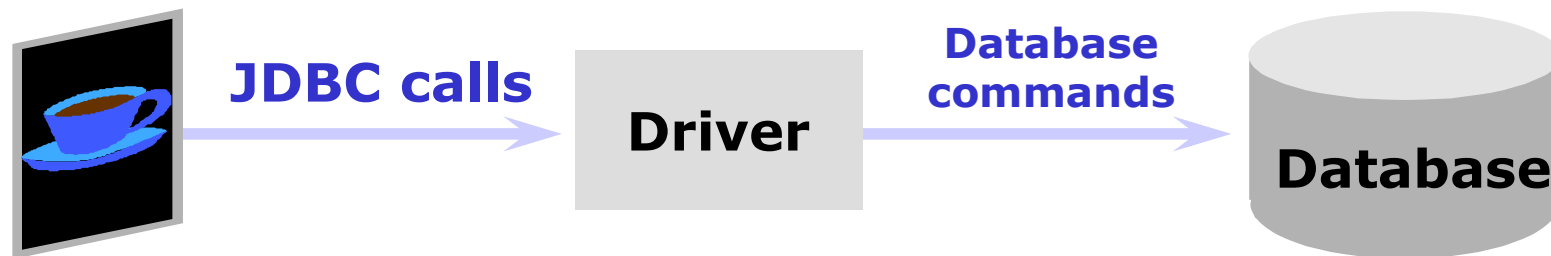


Step 1: Connect



Connect: A JDBC Driver

- A JDBC driver is an interpreter that translates JDBC method calls to vendor-specific database commands.
- JDBC driver Implements interfaces in `java.sql`



DriverManager class

All methods are static and the class does not have a constructor

Methods

Connection getConnection(**String url**)

Connection getConnection(**String url** , **String user** , **String password**)

Connection class

Methods

Statement createStatement()

PreparedStatement prepareStatement(**String sql**)

void close ()

DatabaseMetaData getMetaData()

void setAutoCommit(**boolean commit**) // default true

boolean getAutoCommit()

void commit()

void rollback()

Setting up database connection

```
Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" )
```

```
Connection con = DriverManager.getConnection( url )
```

URL format

jdbc: <sub protocol > : <subname related to database>

example

jdbc : odbc : student

jdbc : ids : //www.test.com:90/conn?dbtype=odbc&dsn=student

jdbc:derby://localhost:1527/ramanadb

Creating Connection

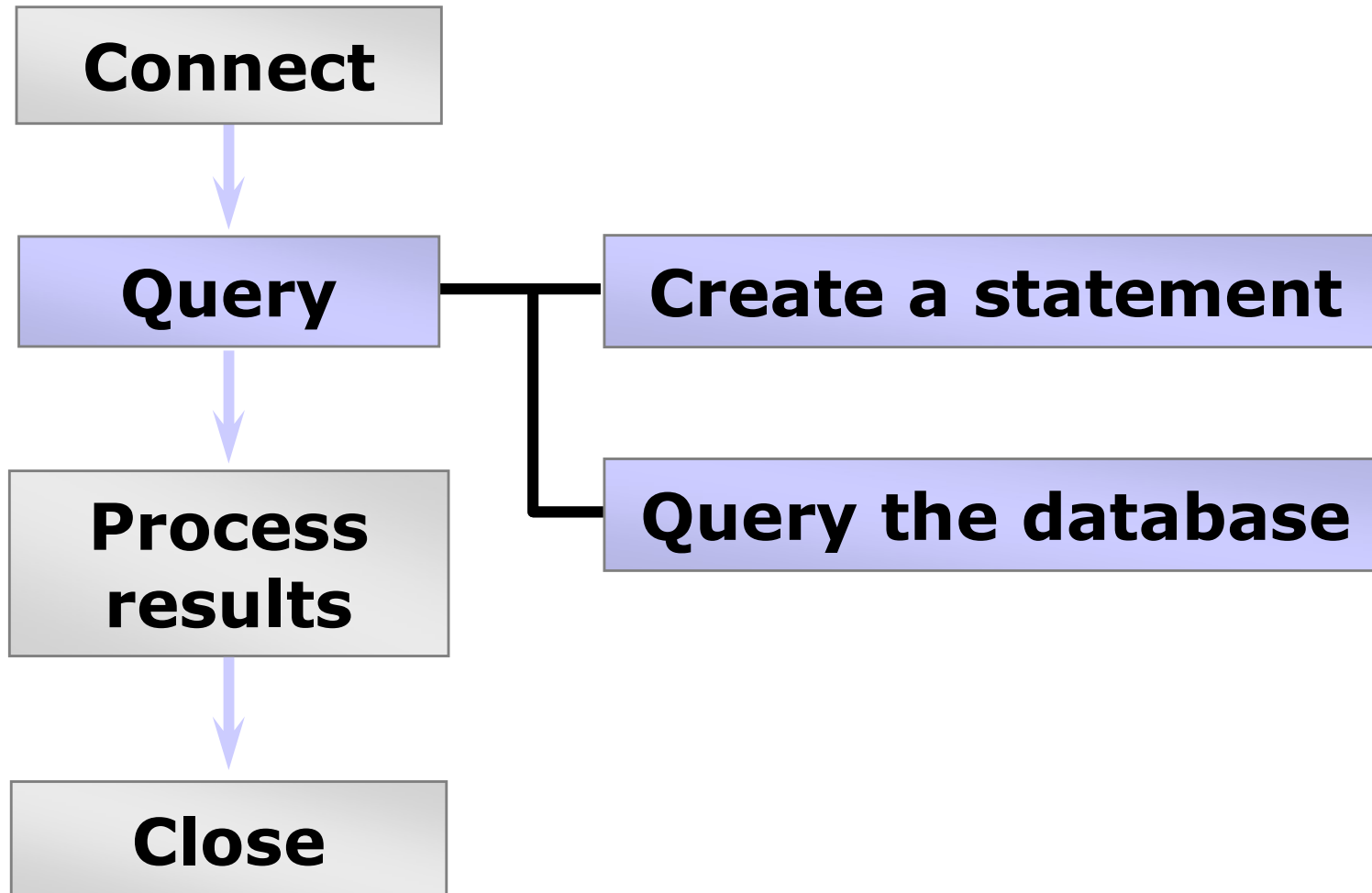
1. Register the driver (prior to java 5. Not required now)

```
Class c = Class.forName( " org.apache.derby.jdbc.ClientDriver ");
```

2. Connect to the database

```
Connection conn = DriverManager.getConnection(  
    "jdbc:derby:codejava/webdb ",  
    "user",  
    "pwd");
```


Step 2: Query



Statements

types of statements

- **Statement**
- **PreparedStatement**
- **CallableStatement**

All these are implemented as classes

Statement interface

Methods

ResultSet executeQuery(**String** query)

int executeUpdate(**String** sql)

boolean execute(**String** sql)

ResultSet getResultSet ()

int getUpdateCount()

Method	Returns	Used for
executeQuery(sqlString)	ResultSet	SELECT statement
executeUpdate(sqlString)	int (rows affected)	INSERT, UPDATE, DELETE, or a DDL
execute(sqlString)	boolean (true if there was a ResultSet)	Any SQL command or commands

Example code for Statement

```
Statement stmt ;
```

```
private void runStatement() throws SQLException {
```

```
    Class.forName ("jdbc.odbc.JdbcOdbcDriver");
```

```
    Connection con = DriverManager.getConnection("jdbc:odbc:dsn" );
```

```
    String sql = "select name , salary from emp where empno = 3010 " ;
```

```
    stmt = con.createStatement( ) ;
```

```
    ResultSet rs = stmt.executeQuery( sql ) ;
```

```
}
```

Statement methods

1. Create an empty statement object

```
Statement stmt = conn.createStatement();
```

2. Execute the statement

```
ResultSet rset = stmt.executeQuery(statement);  
int count = stmt.executeUpdate(statement);  
boolean isquery = stmt.execute(statement);
```

Statement methods: **Examples**

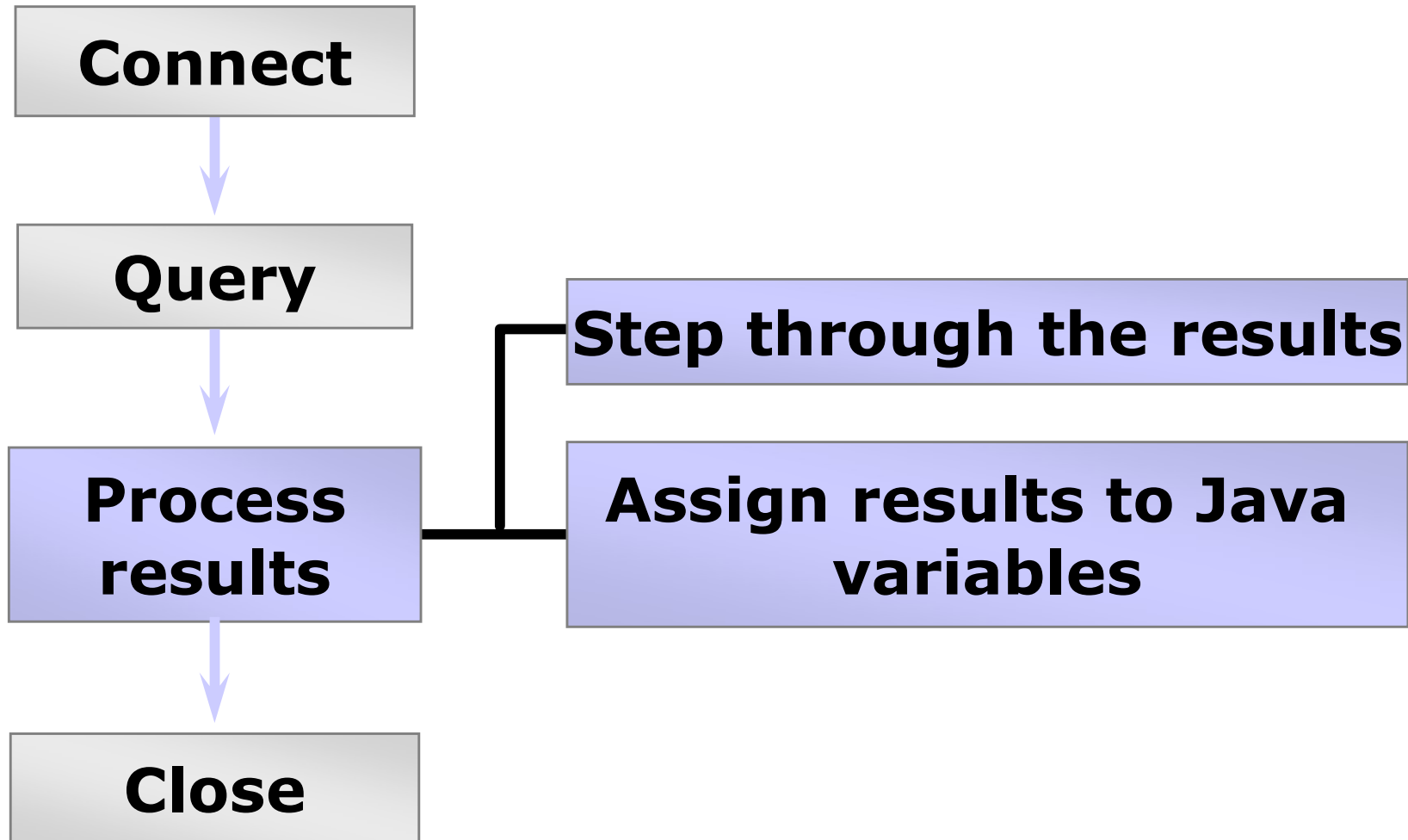
Execute a select statement

```
Statement stmt = conn.createStatement();  
ResultSet rset = stmt.executeQuery  
("select NAME, VERTICAL from STUDENT");
```

Execute a delete statement

```
Statement stmt = conn.createStatement();  
int rowcount = stmt.executeUpdate  
("delete from STUDENT where ID = 1000");
```

Step 3: Process the Results



Using a ResultSet Object

```
String query = "SELECT * FROM Employee";  
ResultSet rs = stmt.executeQuery(query);
```



ResultSet cursor →

The first `next()` method invocation returns `true`, and `rs` points to the first row of data.

`rs.next()`



110	Troy	Hammer	1965-03-31	102109.15
123	Michael	Walton	1986-08-25	93400.20
201	Thomas	Fitzpatrick	1961-09-22	75123.45
101	Abhijit	Gopali	1956-06-01	70000.00

`rs.next()`



`rs.next()`



`rs.next()`



`rs.next()`



No data

The last `next()` method invocation returns `false`, and the `rs` instance is now null.

ExecuteQuery() - example

```
1  package com.example.text;
2
3  import java.sql.DriverManager;
4  import java.sql.ResultSet;
5  import java.sql.SQLException;
6  import java.util.Date;
7
8  public class SimpleJDBCTest {
9
10     public static void main(String[] args) {
11         String url = "jdbc:oracle:thin@localhost:1521:xe";
12         String username = "hr";
13         String password = "hr";
14         String query = "SELECT * FROM Employee";
15         try {
16             1 Connection con =
17                 DriverManager.getConnection (url, username, password);
18                 Statement stmt = con.createStatement ();
19                 ResultSet rs = stmt.executeQuery (query) ;
```

ExecuteQuery() - example

```
19         while (rs.next()) {
20             int empID = rs.getInt("ID");
21             String first = rs.getString("FirstName");
22             String last = rs.getString("LastName");
23             Date birthDate = rs.getDate("BirthDate");
24             float salary = rs.getFloat("Salary");
25             System.out.println("Employee ID:   " + empID + "\n"
26                               + "Employee Name: " + first + " " + last + "\n"
27                               + "Birth Date:   " + birthDate + "\n"
28                               + "Salary:      " + salary);
29         } // end of while
30     } catch (SQLException e) {
31         System.out.println("SQL Exception: " + e);
32     } // end of try-with-resources
33 }
34 }
```

ExecuteUpdate() - example

```
1. public class InsertJDBCExample {
2.     public static void main(String[] args) {
3.         // Create the "url"
4.         // assume database server is running on the localhost
5.         String url = "jdbc:oracle:thin@localhost:1521:xe";
6.         String unm = "hr";
7.         String pwd = "hr";
8.     try {
9.         Connection con = DriverManager.getConnection(url, unm, pwd)
10.         Statement stmt = con.createStatement();
11.         String query = "INSERT INTO Employee VALUES (500, 'Jill',
12.             'Murray','1950-09-21', 150000)";
13.         if (stmt.executeUpdate(query) > 0) {
14.             System.out.println("A new Employee record is added");
15.         }
16.         String query1="select * from Employee";
17.         ResultSet rs = stmt.executeUpdate(query1);
18.         //code to display the rows
19.     }
```

PreparedStatement

- PreparedStatement is a subclass of Statement that allows you to pass arguments to a precompiled SQL statement.

```
double value = 100_000.00;  
String query = "SELECT * FROM Employee WHERE  
    Salary > ?";  
PreparedStatement pstmt =  
    con.prepareStatement(query);  
pstmt.setDouble(1, value);  
ResultSet rs = pstmt.executeQuery();
```

- PreparedStatement is useful when you want to execute a SQL statement multiple times.

PreparedStatement

Methods

ResultSet executeQuery()

int executeUpdate()

boolean execute()

ResultSet getResultSet ()

int getUpdateCount()

void setString(**int** parameterindex , String x)

void setBoolean(**int** parameterindex , boolean x)

void setInt(**int** parameterindex , int x)

void setFloat(**int** parameterindex , float x)

void setDate(**int** parameterindex , java.sql.Date x)

void clearParameters()

PreparedStatement: Setting Parameters

- In general, there is a **setXXX** method for each type in the Java programming language.
- **setXXX** arguments:
 - The first argument indicates which question mark placeholder is to be set.
 - The second argument indicates the replacement value.
- For example:

```
pStmt.setInt(1, 175);  
pStmt.setString(2, "Charles");
```

PreparedStatement: Example

```
PreparedStatement updateEmp;  
String updateString = "update Employee"  
    + "set SALARY= ? where EMP_NAME like ?";  
updateEmp = con.prepareStatement(updateString);  
int[] salary = {1750, 1500, 6000, 1550, 9050};  
String[] names = {"David", "Tom", "Nick", "Harry", "Mark"};  
for(int i=0;i<names.length;i++)  
{  
    updateEmp.setInt(1, salary[i]);  
    updateEmp.setString(2, names[i]);  
    updateEmp.executeUpdate();  
}
```

CallableStatement

- A **CallableStatement** object holds parameters for calling stored procedures.
- A callable statement can contain variables that you supply each time you execute the call.
- When the stored procedure returns, computed values (if any) are retrieved through the **CallableStatement** object.

Using Transactions

- The server-side driver does not support **autocommit** mode.
- With other drivers new connections are in **autocommit** mode and use
- **conn.setAutoCommit(false)** to turn **autocommit** off.
- To control transactions when you are not in **autocommit** mode:
 - **conn.commit()**: Commit a transaction
 - **conn.rollback()**: Roll back a transaction