# COMP9120

Week 8: Introduction to Database Application Development

Semester 1, 2022

Today's content will not be covered in exam or quiz. It is only for assignment 2.

Dr Mohammad Polash
School of Computer Science

THE UNIVERSITY OF
SYDNEY

**Contents developed by Dr Lijun Chang**

*I would like to acknowledge the Traditional Owners of Australia and recognise their continuing connection to land, water and culture. I am currently on the land of the Gadigal people of the Eora nation and pay my respects to their Elders, past, present and emerging.*

*I further acknowledge the Traditional Owners of the country on which you are on and pay respects to their Elders, past, present and future.*

› **Introduction to Database Application Development**

› **A Very Short Introduction to Java and Python**

› **DB Application Development in Java and Python**

› **Error Handling in Java and Python**

› **Security and SQL Injection**

› **Stored Procedure**

› *Interactive SQL*:  SQL statements input from terminal;  DBMS outputs to screen

  - Inadequate for most uses

    - It may be necessary to process the data before output

    - Amount of data returned not known in advance

    - SQL has limited expressive power

› *Non-interactive SQL*:  SQL statements are included in a database application program written in a host language, like C++, Java, Python, PHP

› *Client-side* vs. *Server-side* application development

  - Client-side: database application program

  - Server-side: stored procedures and triggers

centralized system

| presentation services | application services | API | DBMS |

user module

› *Presentation Services* - displays forms, handles flow of information to/from screen

› *Application Services* - implements user request, interacts with DBMS

› *DBMS* handles user requests – retrieves and returns results of queries or handles user requests for inserts/updates/deletes of data.

› Examples: Any application with integrated DB

- MS Access systems

- SQLite, esp. Smartphone apps

R&G §7.5

THE UNIVERSITY OF SYDNEY

client machines

database server machine

presentation services | application services

●
●
●

presentation services | application services

DBMS

communication / network

R&G § 7.5

application / web server machine

database server machine

client machines

presentation server

presentation server

application server

DBMS

communication (IPC or network)

**Presentation Tier**

**Middle Tier**

**Data Management Tier**

R&G §7.5
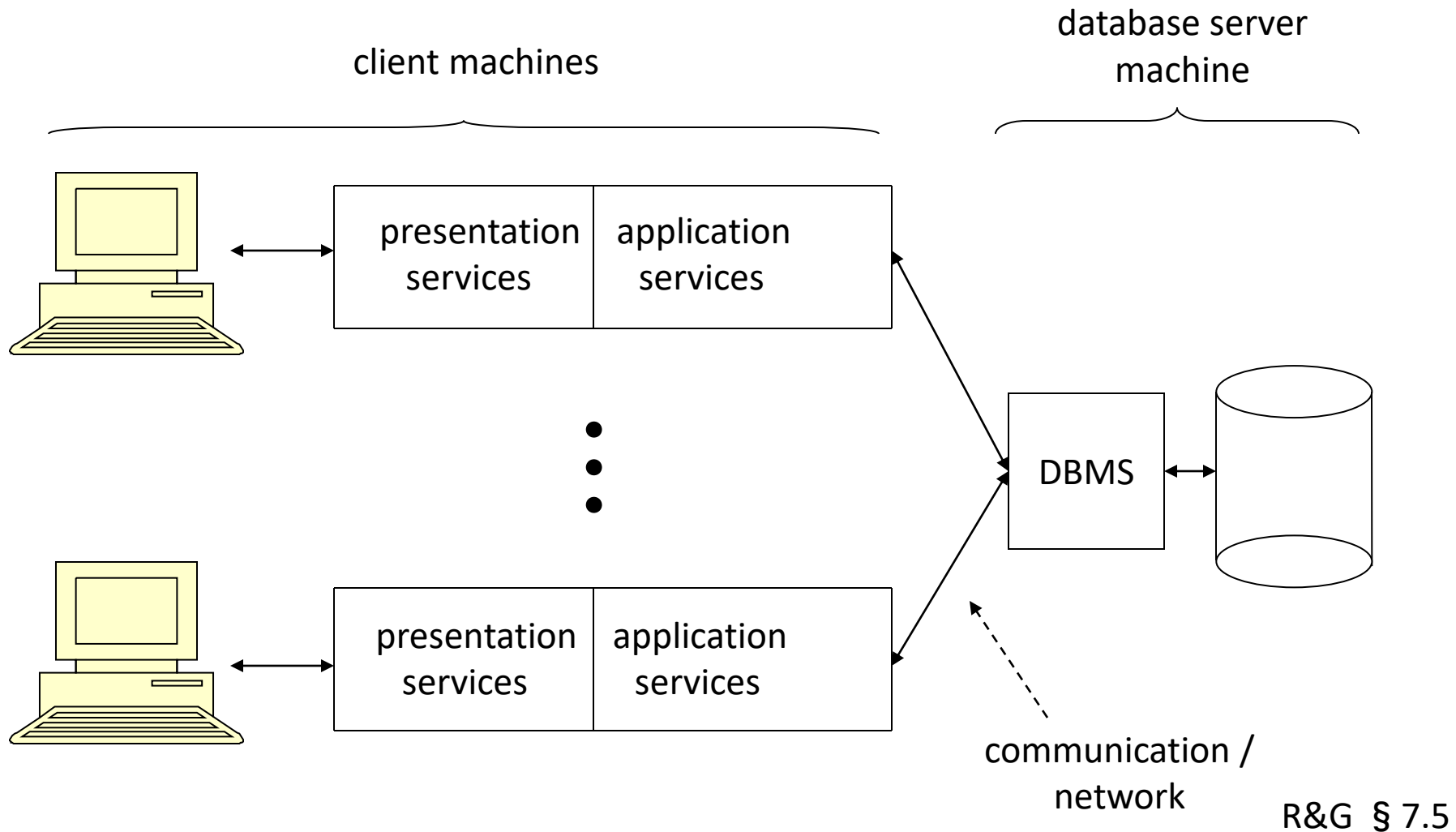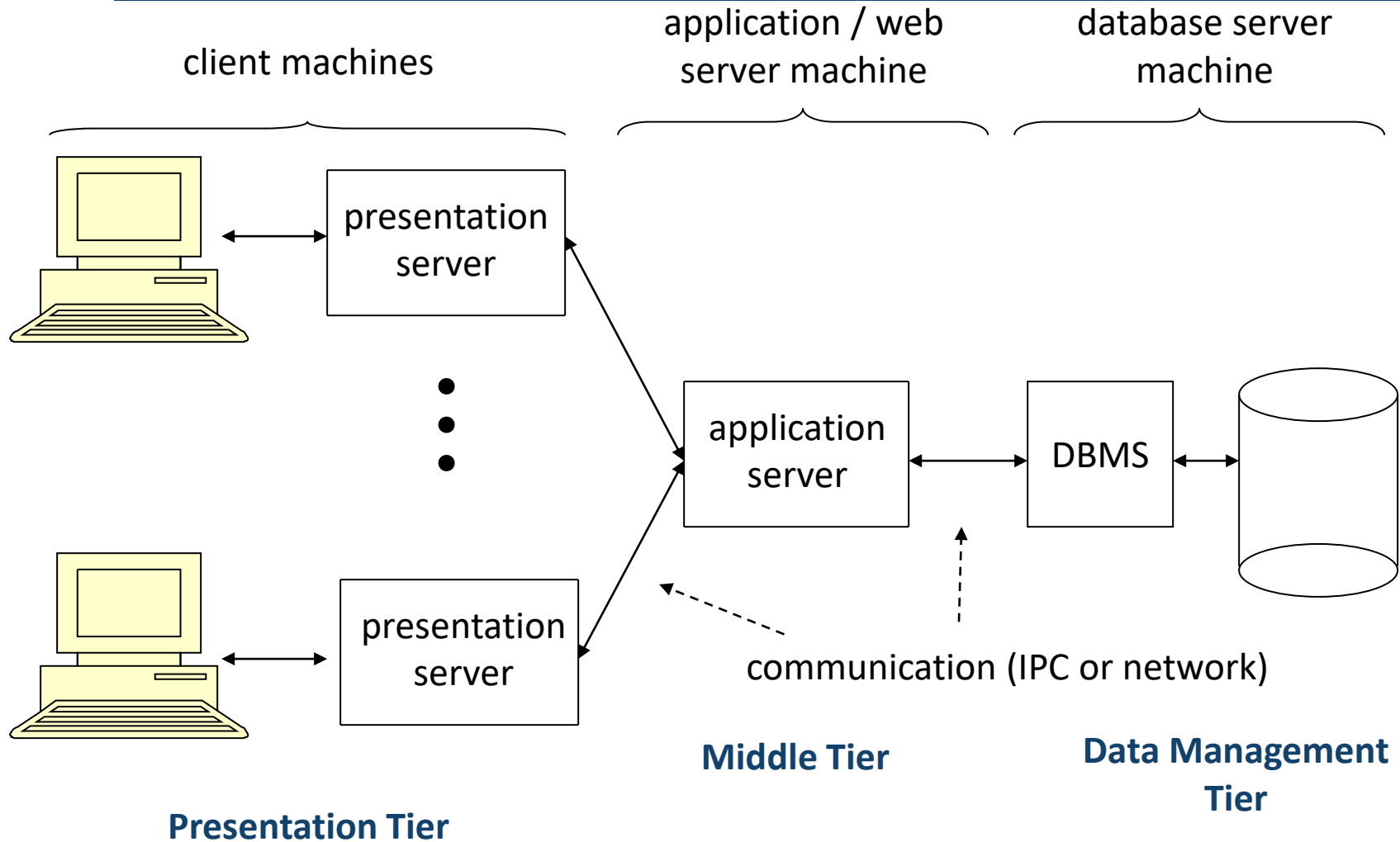
› SQL commands can be called from within a *host language* (e.g., C++, Java, PHP, Python) program

- SQL statements can refer to host variables

- Must include a statement to *connect* to the right database

```
int userInput = takeUserInput();

SELECT *
FROM student
WHERE sid = userInput
```

› Two main integration approaches

- Statement-level interface (SLI)

  - Embed SQL in the host language (Embedded SQL in C, SQLJ)

- Call-level interface (CLI)

  - Create special API to call SQL commands (ODBC, JDBC, PHP-PDO, etc.)

  - SQL statements are passed as arguments to host language (library) procedures / APIs

R&G § 6.1

› **Introduction to Database Application Development**

› **A Very Short Introduction to Java and Python**

› **DB Application Development in Java and Python**

› **Error Handling in Java and Python**

› **Security and SQL Injection**

› **Stored Procedure**

› OOP is a programming paradigm based on the concept of "objects", which may contain data (in the form of *variables*, often known as *attributes*) and code (in the form of *methods*).

- object's methods can access and often modify the attributes of the object with which they are associated

- Objects are instances of *classes*, which typically also determine their type.

```java
public class Employee {
        String name;
        int age;
        double salary;
        public Employee(String name) {
                this.name = name;
        }
        public void setAge(int empAge) {
                age = empAge;
        }
        public void setSalary(double empSalary) {
                salary = empSalary;
        }
        public void printEmployee() {
                System.out.println("Name:"+ name );
                System.out.println("Age:" + age );
                System.out.println("Salary:" + salary);
        }
}
```

› Java requires you to *compile* (translate) your program into a form that the machine understands.

*compile*            *execute*

| source code Hello.java | byte code Hello.class | output |
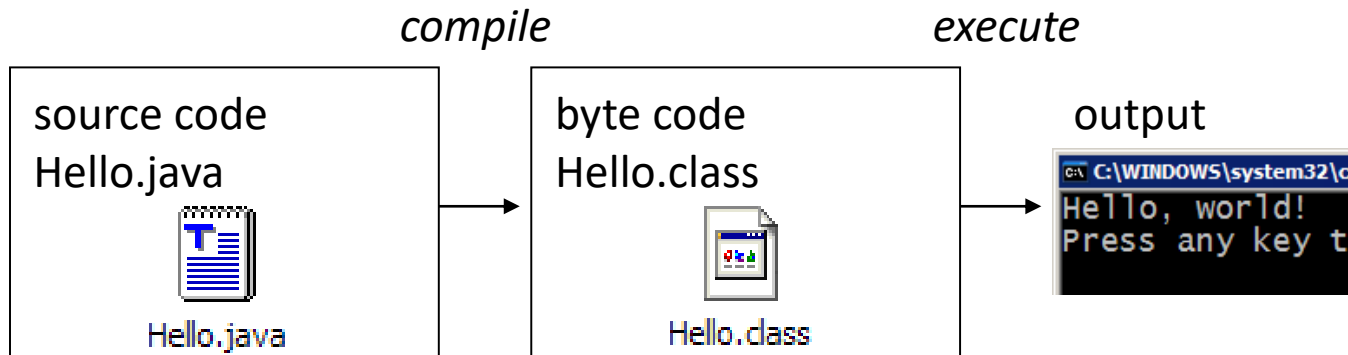
› Python is instead directly *interpreted* into machine instructions.

*interpret*

| source code Hello.py | output |
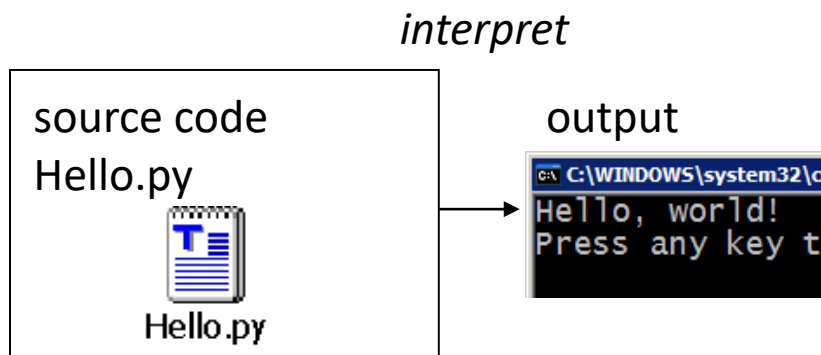
Hello.java

```
class Hello {
        public static void main(String[] args) {
                System.out.println("Hello World !!!");
    }
}
```

javac Hello.java                                    *(compilation creates Hello.class )*

java Hello                                              *(Execution on the local JVM)*

› Everything is in a class

› In the generated byte code, one file for each public class

› In the runnable public class:

- **public static void main(String**[] args)

- Execution starts from the **main** function

Hello.py

```
def printHelloWorld():
        print("Hello World !!!")

printHelloworld()
```

› python Hello.py

› Not everything in a class

› Execution starts from the first statement at *indentation* level 0 of Hello.py

```java
boolean myFunction(int myValue, String myText) {
  boolean valueToReturn = false;
  try {
      // Do something with input parameters
      // ** Interact with database **
      // Set return value
      System.out.println("Try Block");
  } catch (Exception e) {
      // Handle errors
      System.out.println("Exception Block");
  } finally {
      // Cleanup
      System.out.println("Finally Block");
  }
  return valueToReturn;
}
```

```python
def myFunction(myValue, myText):
    valueToReturn = False
    try:
        # Do something with input parameters
        # ** Interact with database **
        # Set return value
        print("Try block")
    except:
        # Handle errors
        print("Except Block")
    finally:
        # Cleanups
        print("Finally Block")
    return valueToReturn
```

› **Introduction to Database Application Development**

› **A Very Short Introduction to Java and Python**

› **DB Application Development in Java and Python**

› **Error Handling in Java and Python**

› **Security and SQL Injection**

› **Stored Procedure**

› JDBC is a **Java API** for communicating with database systems supporting SQL

› JDBC supports a variety of features for querying and updating data, and for retrieving query results

› Model for communicating with the database:

- Acquire a connection

- Create a "Statement" object

- Execute queries using the Statement object to send queries and fetch results

- Exception mechanism to handle errors
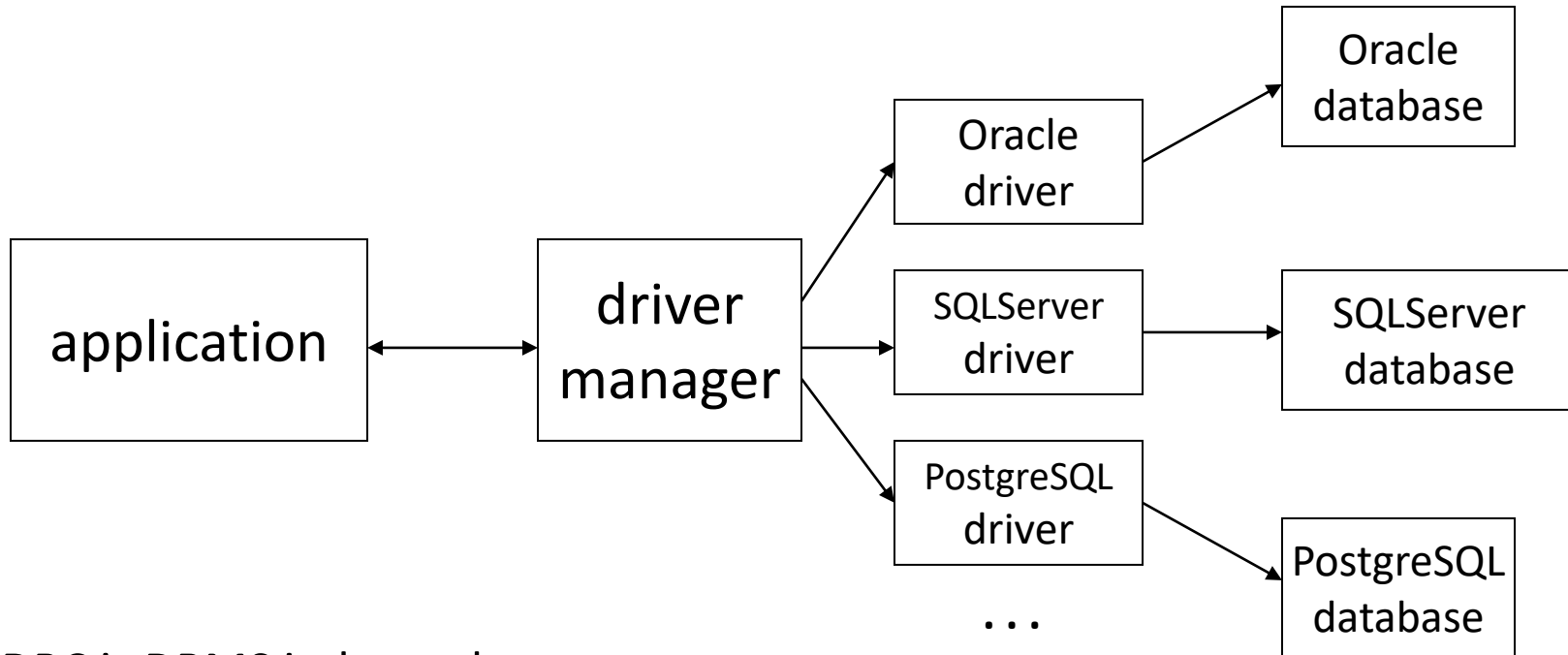
> JDBC is DBMS independent

- JDBC functions are generic

- DriverManager allows to connect to specific driver

  - Even to different databases from the same program

› A session with a data source is started through the creation of a **Connection** object

- example with PostgreSQL

        **PGSimpleDataSource** source = **new PGSimpleDataSource**();

        source.**setServerName**(myHost);

        source.**setDatabaseName**(myDB);

        source.**setUser**(userid);

        source.**setPassword**(passwd);

        **Connection** conn = source.**getConnection**();

- Always release resource with conn.**close**();

› SQL operations are conducted using **java.sql.Statement**

- Constructed from a Connection object:

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

- Execute a SQL query:

    stmt.**executeQuery**("**SELECT** … ");

- Execute a DML statement (INSERT/UPDATE/DELETE)

    stmt.**executeUpdate**("**INSERT INTO** … ");

- Release resource with stmt.**close**();


› Two other ways (will be covered later)

- *PreparedStatement* (semi-static SQL statements)

- *CallableStatement* (stored procedures)

› stmt.**executeQuery** returns data, encapsulated in a **ResultSet** object (a cursor)

```
ResultSet rs = stmt.executeQuery(sql);
while(rs.next()) {

   //Iterate through records
  // process the data

    rs.getString("name");
}
rs.close() // Release resources
```

› A **ResultSet** can be a very powerful cursor:

- **previous**(): moves one row back

- **absolute**(**int** num): moves to the row with the specified number

- **relative**(**int** num): moves forward or backward

- **first**() and **last**(): jump to ends

- **wasNull**():  dealing with NULL values

| SQL Type | Java class | ResultSet get method |
|---|---|---|
| BIT | Boolean | getBoolean() |
| CHAR | String | getString() |
| VARCHAR | String | getString() |
| DOUBLE | Double | getDouble() |
| FLOAT | Double | getDouble() |
| INTEGER | Integer | getInt() |
| REAL | Double | getFloat() |
| DATE | java.sql.Date | getDate() |
| TIME | java.sql.Time | getTime() |
| TIMESTAMP | java.sql.TimeStamp | getTimestamp() |

For python matching example: http://initd.org/psycopg/docs/usage.html#adaptation-of-python-values-to-sql-types

› **Python's API (DB-API)** for communicating with database systems supporting SQL

- Specific *modules* for each db engine (eg: Oracle, Postgres, IBM DB2, etc) provide an implementation for common DB-API functionality

› Model for communicating with the database:

- Acquire a connection

- Create a "cursor" object

- Execute queries using the cursor object to send queries and fetch results

- Exception mechanism to handle errors

```python
import psycopg2

try:
    # fetch connection object to connect to the database
    conn = psycopg2.connect(database="postgres",user="test",password="secret", host="host")

    # fetch cursor prepare to query the database
    curs = conn.cursor()

    # execute a SQL query
    curs.execute("SELECT name
              FROM Student NATURAL JOIN Enrolled
              WHERE uos_code = 'COMP9120'")

    # can loop through the resultset
    for result in curs:
        print (" student: " + result[0])

    # for illustrating close methods – calling close() on cursor and connection
    # objects will release their associated resources.
    curs.close()
    conn.close()

except Exception as e: # error handling
    print("SQL error: unable to connect to database or execute query")
    print(e)
```

› **execute**(operation[,parameters])

- Execute a query or a command with the sql provided as the operation argument;

- the sql query parameter values provided in the parameters argument.

› **executemany**(operation[,parameters])

- Can execute the same query (operation) multiple times, each time with a different parameter set

› **fetchone**()

- Fetch the next row of a query result set, or returns None if no more data is available

› **fetchmany**([size=cursor.arraysize])

- Return multiple rows from the result set for a given query, where one can specify the desired number of rows to return (where they are available)

› **fetchall**()

- fetch all remaining rows

› **close**()

- Release cursor resources

Reference: http://initd.org/psycopg/docs/cursor.html

› **Introduction to Database Application Development**

› **A Very Short Introduction to Java and Python**

› **DB Application Development in Java and Python**

› **Error Handling in Java and Python**

› **Security and SQL Injection**

› **Stored Procedure**

```python
import psycopg2

try:
    # fetch connection object to connect to the database
    conn = psycopg2.connect(database="postgres",user="test",password="secret", host="host")

    # fetch cursor prepare to query the database
    curs = conn.cursor()

    # execute a SQL query
    curs.execute("SELECT name
                  FROM Student NATURAL JOIN Enrolled
                  WHERE uos_code = 'COMP9120'")

    # can loop through the resultset
    for result in curs:
        print (" student: " + result[0])

    # for illustrating close methods – calling close() on cursor and connection
    # objects will release their associated resources.
    curs.close()
    conn.close()

except Exception as e: # error handling
    print("SQL error: unable to connect to database or execute query")
    print(e)
```

Bine ati venit in magazinul eSIMM'S.

INVALID SQL: 1016 : Can't op
SQL QUERY FAILURE: SELEC

INVALID SQL: 1016 : Can't op
SQL QUERY FAILURE: SELEC

INVALID SQL: 1016 : Can't op
SQL QUERY FAILURE: SELEC

INVALID SQL: 1016 : Can't op
SQL QUERY FAILURE: SELEC

INVALID SQL: 1016 : Can't op
SQL QUERY FAILURE: SELEC

INVALID SQL: 1016 : Can't op
SQL QUERY FAILURE: SELEC

INVALID SQL: 1016 : Can't op
SQL QUERY FAILURE: SELEC

INVALID SQL: 1016 : Can't op
SQL QUERY FAILURE: SELEC

INVALID SQL: 1016 : Can't op
SQL QUERY FAILURE: SELEC

INVALID SQL: 1016 : Can't op
SQL QUERY FAILURE: SELEC

**Association for Computing Machinery**
Advancing Computing as a Science & Profession

**ACM Order Rectification**

The web site you are accessing has experienced an unexpected error. Please contact the website administrator.

The following information is meant for the website developer for debugging purposes.

Error Occurred While Processing Request

Element ORDERID is undefined in URL.

The error occurred in **D:\wwwroot\Public\rectifyCC\rectifyCC.cfm: line 463**

```
461 :     WHERE a.order_id = b.order_id
462 :       AND a.order_id = c.order_id
463 :       AND a.order_id = '#URL.orderID#'
464 :   </CFQUERY>
465 :
```

Resources:

- Check the ColdFusion documentation to verify that you are using the correct syntax.
- Search the Knowledge Base to find a solution to your problem.

Browser    Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_5_8; en-us) AppleWebKit/531.9 (KHTML, like Gecko) Version/4.0.3 Safari/531.9

Remote
Address    129.78.220.7

Cauta:

› Most of java.sql can throw an **SQLException** if an error occurs.

- Catch and process with **catch**(**SQLException** e) { **…** }

- Use **getMessage**()or **getSQLState**() or **getErrorCode**() to identify problem

**Table 12. Class Code 23: Constraint Violation**

| SQLSTATE Value | Meaning |
|---|---|
| 23001 | The update or delete of a parent key is prevented by a RESTRICT update or delete rule. |
| 23502 | An insert or update value is null, but the column cannot contain null values. |
| 23503 | The insert or update value of a foreign key is invalid. |
| 23504 | The update or delete of a parent key is prevented by a NO ACTION update or delete rule. |
| 23505 | A violation of the constraint imposed by a unique index or a unique constraint occurred. |
| 23510 | A violation of a constraint on the use of the command imposed by the RLST table occurred. |
| 23511 | A parent row cannot be deleted, because the check constraint restricts the deletion. |
| 23512 | The check constraint cannot be added, because the table contains rows that do not satisfy the constraint definition. |
| 23513 | The resulting row of the INSERT or UPDATE does not conform to the check constraint definition. |
| 23514 | Check data processing has found constraint violations. |
| 23515 | The unique index could not be created or unique constraint added, because the table contains duplicate values of the specified key. |
| 23520 | The foreign key cannot be defined, because all of its values are not equal to a parent key of the parent table. |
| 23521 | The update of a catalog table violates an internal constraint. |

ftp://ftp.software.ibm.com/ps/products/db2/info/vr6/htm/db2m0/db2state.htm#HDRSTTMSG

› Most of java.sql can throw an **SQLException** if an error occurs.

  - Catch and process with **catch**(**SQLException** e) { **...** }

  - Use **getMessage**()or **getSQLState**() or **getErrorCode**() to identify problem

› Sub-classes available to catch specific types e.g.:

  - **SQLTimeoutException**

  - **SQLIntegrityConstraintViolationException**

› **SQLWarning** is a subclass of **SQLException**; not as severe

  - conn.**getWarnings**()**;**

  - conn.**getNextWarning**();

  - conn.**clearWarnings**();

The following function may fail to work for a number of reasons.

```
void exampleEnrolment() {
  PGSimpleDataSource source = new PGSimpleDataSource();
  source.setServerName(myHost);
  source.setDatabaseName(myDB);
  source.setUser(userid);
  source.setPassword(passwd);
  Connection conn = source.getConnection();

  Statement stmt = conn.createStatement();
  stmt.executeUpdate("INSERT INTO Transcript VALUES
    (123,'COMP9120', 'S1', 2022,'HD')");

  conn.close();
}
```

May fail due to server/connection problems

› Query could time out
› Primary key violation
› Foreign key violation

```java
void exampleEnrolment() {
  Connection conn = null;
  try {
    conn = openConnection();
    Statement stmt = conn.createStatement();
    stmt.executeUpdate("INSERT INTO Transcript VALUES (123,'COMP9120', 'S1', 2022,'HD')");
  }
  catch (SQLIntegrityConstraintViolationException e) {
          System.err.println("Violated a constraint!");
  }
  catch (SQLTimeoutException e) {
          System.err.println("Operation timed out");
  }
  catch (SQLException e) {
          System.err.println("Other problem");
  }
finally {
    if (conn != null)
     try{conn.close();} catch(SQLException e) {//handle exception}
  }
}
```

› Error handling via normal exception mechanism of Python

- Errors and warnings are made available as Python exceptions

  - **Warning**      raised for warnings such as data truncation on insert, etc

  - **Error**        exception raised for various db-related errors

› psycopg API extension:

- Exception attributes for detailed SQL error codes and messages

- **pgerror**        string of the error message returned by backend

- **pgcode**         string with the **SQLSTATE** error code returned by backend

› Example:

```
try:
    psycopg2.connect(…)
except psycopg2.Error as e:
    print("Problem connecting to database:")
    print(e.pgerror)
    print(e.pgcode)
```

Demo purpose only. As said before:
please do not directly print SQL exceptions :)

```
StandardError
|__    Warning
|__    Error
        |__    InterfaceError
        |__    DatabaseError
                |__    DataError
                |__    OperationalError
                |       |__    psycopg2.extensions.QueryCanceledError
                |       |__    psycopg2.extensions.TransactionRollbackError
                |__    IntegrityError
                |__    InternalError
                |__    ProgrammingError
                |__    NotSupportedError
```

http://initd.org/psycopg/docs/module.html#exceptions

Also see: https://www.python.org/dev/peps/pep-0249

› **Introduction to Database Application Development**

› **A Very Short Introduction to Java and Python**

› **DB Application Development in Java and Python**

› **Error Handling in Java and Python**

› **Security and SQL Injection**

› **Stored Procedure**

| User Name: | ' or '1'='1 |
|---|---|
| Password: | ' or '1'='1 |

| Details: |
|---|
| |

http://www.open.edu/openlearn/ocw/mod/oucontent/view.php?id=48319&section=1.3

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(
 "SELECT * FROM Student WHERE
 name = '" + uName + "' AND Pass ='" + uPass + "'");
```

SELECT * FROM Student WHERE
Name ='John' AND Pass ='myPassword'

SELECT * FROM Student WHERE
Name ='' or '1'='1' AND Pass ='' or '1'='1'

Will return all students' details!!

| User Name: | ' or '1'='1 |
|---|---|
| Password: | ' or '1'='1 |

Details:

› How do we limit risks?

- Hide error messages that expose the internals

- Use salted hash passwords

- Prevent SQL injection with:

    - *Prepared Statements* in Java

    - *Anonymous or Named Parameters* in Python

› **java.sql.PreparedStatement** allows a statement to be executed with host variables after the query plan has been evaluated

› <u>Example</u>:

credit(**int** amount, **int** accountno) {
  stmt = conn.**prepareStatement**("**UPDATE** account
                                        **SET** balance = balance + ?
                                        **WHERE** account_number = ?");

  stmt.**setFloat**(1, amount);

  stmt.**setInt**(2, accountno);

  result = stmt.**executeQuery**();
}

› More flexible and <u>secure</u>

› As a rule, always use in preference to Statements

› What security issues can you identify in this function? How could they be fixed?

```
void getStudentAddress(int studID, String password) {
  Connection conn = openConnection();

  Statement stmt = conn.createStatement();
  ResultSet rs = stmt.executeQuery(
          "SELECT address FROM Student WHERE studId='"
          + studID + "' AND password = '" + password + "'");

  // Process results
  while (rs.next()){
    System.out.println(rs.getString("address"));
  }
  conn.close();
}
```

> Try:
> getStudentAddress(307088592, "' OR 1=1 -- ");

**SELECT** address **FROM** Student **WHERE** studId='307088592' **AND** password = '' **OR** 1=1 —-

› What security issues can you identify in this function? How could they be fixed?

```java
void getStudentAddress(int studID, String password) {
  Connection conn = openConnection();

  PreparedStatement stmt = conn.prepareStatement(
   "SELECT address FROM Student WHERE studId=? AND password=?");

  stmt.setInt(1, studID);
  stmt.setString(2, password);

  ResultSet rs = stmt.executeQuery();

  // Process results
  while (rs.next()){
    System.out.println(rs.getString("address"));
  }
  conn.close();
}
```

NEVER ever use Python string concatenation (+) or string parameter interpolation (%) to pass variables to a SQL query string!
=> otherwise your program is vulnerable to SQL Injection attacks

```
query="""SELECT E.studId FROM Enrolled E
        WHERE E.uosCode = """ + uosCode +
        " AND E.semester = " + semester

cursor.execute( query )
```

Two (safe) approaches for passing query parameters:
(because **execute**() will do any necessary escaping / conversions for parameter markers)

# 1. Anonymous Parameters

```
studid = 12345
cursor.execute(
    "SELECT name FROM Student WHERE sid=%s",
    (studid) )
```
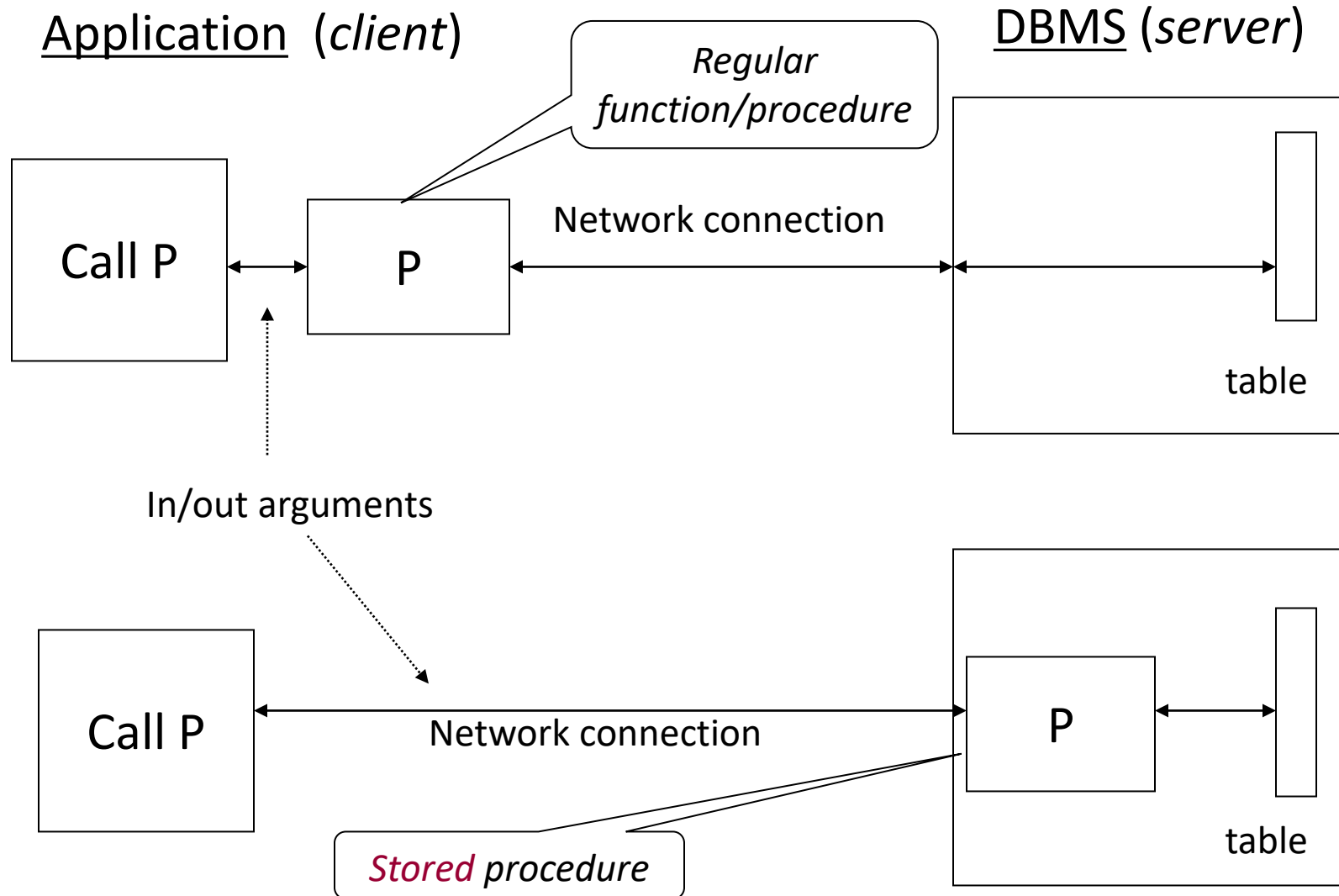
# 2. Named Parameters

```
studid = 12345
cursor.execute(
    "SELECT name FROM Student WHERE sid=%(sid)s",
    {'sid': studid} )
```

› **Introduction to Database Application Development**

› **A Very Short Introduction to Java and Python**

› **DB Application Development in Java and Python**

› **Error Handling in Java and Python**

› **Security and SQL Injection**

› **Stored Procedure**

› Run logic within the database server

- Included as schema element (stored in DBMS)

- Invoked by the application

› Pros:

- Additional abstraction layer
(programmers do not need to know the schema)

- Reduced data transfer

› Cons:

- What if you wanted to switch DBMS?

- rewrite all logic?

Application (*client*)

DBMS (*server*)

*Regular function/procedure*

Call P

P

Network connection

table

In/out arguments

Call P

Network connection

P

table

*Stored procedure*

› All major database systems provide extensions of SQL to a simple, general purpose language

- PostgreSQL: PL/pgSQL, Oracle: PL/SQL (syntax differs!!!)

› Procedure Declarations (with SQL/PSM)

**CREATE PROCEDURE** name ( parameter1,…, parameterN )
   *local variable declarations*
   *procedure code;*

› Stored Procedures can have parameters

- of a valid SQL type (parameter types must match)

- three different modes

  - **IN**  arguments to procedure

  - **OUT**  return values

  - **INOUT**  combination of **IN** and **OUT**

› Stored Procedures have full access to SQL, plus extensions:

- Local variables, loops, if-then-else conditions

```
CREATE FUNCTION RateStudent(studId INTEGER, uos VARCHAR) RETURNS CHAR AS $$
DECLARE
        grade       CHAR;
        marks       INTEGER;
BEGIN

        SELECT SUM(mark) INTO marks
        FROM Assessment
        WHERE sid=studId AND uosCode=uos;

        IF          marks>=85THEN grade := 'H';
        ELSIF       marks>=75THEN grade := 'D';
        ELSIF       marks>=65THEN grade := 'C';
        ELSIF       marks>=50THEN grade := 'P';
        ELSE                      grade := 'F';
        END IF;

        RETURN grade;
END; $$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION RateStudent_INOUT(IN studId INTEGER, IN uos VARCHAR, OUT result CHAR) AS $$
DECLARE
        grade       CHAR;
        marks       INTEGER;
BEGIN

        SELECT SUM(mark) INTO marks
        FROM Assessment
        WHERE sid=studId AND uosCode=uos;

        IF          marks>=85THEN grade := 'H';
        ELSIF       marks>=75THEN grade := 'D';
        ELSIF       marks>=65THEN grade := 'C';
        ELSIF       marks>=50THEN grade := 'P';
        ELSE                      grade := 'F';
        END IF;

        result := grade;
END; $$ LANGUAGE plpgsql;
```

› Use **java.sql.CallableStatement** subclass of Statement

› Calling a stored procedure with return value:

```
CallableStatement call = conn.prepareCall(”{? = call RateStudent(?, ?)}");
call.registerOutParameter(1, Types.CHAR);
call.setInt(2, 101);
call.setString(3, "COMP9120");
call.execute();

String result = call.getString(1);
```

› Calling a stored procedure with IN/OUT parameter :

```
CallableStatement call = conn.prepareCall(”{call RateStudent_INOUT(?, ?, ?)}");
call.setInt(1, 101);
call.setString(2, "COMP9120");
call.registerOutParameter(3, Types.CHAR);
call.execute();

String result = call.getString(3);
```

› **Cursor** objects have an explicit **callproc**()method

- **cursor.callproc**() makes the OUT parameters available as resultset

› Calling a stored procedure with return value:

```
curs.callproc("RateStudent", [101, "comp9120"])
output = curs.fetchone()


result = output[0];
```

› Calling a stored procedure with IN/OUT parameter :

```
curs.callproc("RateStudent_INOUT", [101, "comp9120"])
output = curs.fetchone()


result = output[0];
```

› The following function performs several queries and updates, incurring several network round trips. Rewrite as a stored procedure and change the function to call this.

```java
void enrolStudent(Connection conn, int studID, String uos, String sem, int year)
{
  PreparedStatement stmt = conn.prepareStatement("INSERT INTO Transcript
                                                 VALUES (?, ?, ?, ?, null)");
  stmt.setInt(1, studID);
  stmt.setString(2, uos);
  stmt.setString(3, sem);
  stmt.setInt(4, year);
  stmt.executeUpdate();
  stmt.close();

  stmt = conn.prepareStatement("UPDATE UoSOffering SET enrollment=enrollment+1
                               WHERE UoSCode=? AND Semester=? AND Year=?");
  stmt.setString(1, uos);
  stmt.setString(2, sem);
  stmt.setInt(3, year);
  stmt.executeUpdate();
  stmt.close();
}
```

```sql
CREATE OR REPLACE FUNCTION ENROLSTUDENT ( sid INTEGER, uos VARCHAR,
                                          sem VARCHAR,  yr  INTEGER) AS $$

BEGIN
 INSERT INTO Transcript VALUES (sid, uos, sem, yr, null);
 UPDATE UoSOffering SET enrollment=enrollment+1
                    WHERE uoSCode=uos AND semester=sem AND year=yr;
END; $$ LANGUAGE plpgsql;
```

```java
void enrolStudent(Connection conn, int studID, String uos,
                                   String sem, int year) {
 CallableStatement stmt = conn.prepareCall("{call ENROLSTUDENT (?, ?, ?, ?)}");
 stmt.setInt(1, studID);
 stmt.setString(2, uos);
 stmt.setString(3, sem);
 stmt.setInt(4, year);
 stmt.executeUpdate();
 stmt.close();
}
```

› Some Keywords

- Cursors

- Prepared Statement

- Stored Procedure

- SQL Injection

› Understanding (within a DB application API)

- Error handling

› Skills

- Write application code (e.g. Java functions with JDBC) to interact with a database

- Identify and avoid major security flaws in client code

- Write stored procedures and call from client code

› Ramakrishnan/Gehrke (3rd edition - the 'Cow' book)

 - Chapter 6; 7.5

› Kifer/Bernstein/Lewis (2nd edition)

 - Chapter 8

› Ullman/Widom (3rd edition of 'First Course in Database Systems')

 - Chapter 9  (*covers Stored Procedures, ESQL, CLI, JDBC and PHP*)

Further Documentation:

 - Java JDBC reference: http://docs.oracle.com/javase/7/docs/api/java/sql/package-summary.html

 - **PostgreSQL JDBC Documentation:** https://jdbc.postgresql.org/documentation/head/index.html

 - Python DB-API: https://www.python.org/dev/peps/pep-0249/

 - A Postgres Python DB-API Adapter: http://initd.org/psycopg/docs/

   - Connection pooling: http://initd.org/psycopg/docs/pool.html

 - Database PL/pgSQL Language Reference: https://www.postgresql.org/docs/9.5/static/plpgsql.html

› Java & Python Language References:

- Java

    - https://www.ibm.com/developerworks/learn/java/intro-to-java-course/index.html

    - https://www.udemy.com/java-tutorial/

- Python

    - https://www.learnpython.org/

    - https://docs.python.org/2/tutorial/

› Schema Normalization

- Functional Dependencies

- Schema Normal Forms

- Schema Normalization

› Readings:

- **Ramakrishnan/Gehrke (Cow book), Chapter 19**

- Kifer/Bernstein/Lewis book, Chapter 6

- Ullman/Widom, Chapter 3 (up-to 3.5)

Enjoy your break!