# Test doubles and Mocks

**1**. What are similarities and differences between stubs and mocks?

They are both doubles of objects involved in dependencies that we want to enact and control. Stubs are a bit more limited in that they do not have a working implementation, so they do not allow for checking the interactions with the stubbed object.

**2**. Suppose you are gearing up to test a system that has multiple dependencies:

a) language specific libraries and utility methods

b) intra-object and intra-class dependencies

c) inter-class static dependencies

d) inheritance (including interfaces) and composition (including polymorphism)

e) file IO (input-output)

f) Excel spreadsheet dependencies

g) database (say with an SQL -structured query language - database) dependencies

h) UI (user interface) dependencies (either desktop app or web app); note that most languages have libraries for different UIs.

i) network dependencies (over different protocols, for example, HTTP and HTTPS for web services, FTP for file transfer, SMTP for email, etc)

j) other system dependencies

Which of the objects involved in these dependencies are better candidates to stub/mock and why?

Definitely no: a-d (because interacting with the class themselves would not be any different than interacting with their doubles)
Maybe or maybe not: e-j (probably only if there are slow processes, or that interact with outside infrastructure or are hard to simulate etc)

**3**. What are the steps of implementing a mock for an object that interacts with a database (both querying for information and saving to the database or updating the database; you can think of the calls to save to the database/ update the database as similar to sending an invoice via a web service)?

```
1. Create a way to inject the mocked dependency (for example, via the constructor)
2. Persist the mocked dependency (for example, by saving it as a field in the
entity you want to test)
3. In your test:
    a. Generate data (often hard-coded) to mimic the data in the database
    b. Specify what data needs to be provided when certain method calls are
    issued (using <when> in Mockito)
    c. Check that certain methods are called and with expected arguements
    (using <verify> in Mockito)
    d. Any additional asserts of the expected output or behavior
```

**4**. Next, suppose we want to save the history of the queries to the database - for each query the timestamp of when it was issued and its content and perform additional tests on the history of queries. What are some additional implementations we need to perform?

```
Like the previous problem, with some additional things:
1. Extend the query object to include the timestamp of when it was issued.
2. Use the ArguementCaptor class instead of mock method to generate the mock
(for additional functionality).
3. A method to convert between a simple query and a query with the
timestamp would also be useful.
```