Once you’ve got MongoDB installed and started (a process that should only take a few minutes), you can [connect to the MongoDB shell](http://docs.mongodb.org/manual/tutorial/getting-started/#connect-to-a-database). Most of the MongoDB technical documentation is written for the shell, so it’s always useful to know how to access it, and how use it to troubleshoot problems or prototype solutions.

When you’ve connected, you should see something like

MongoDB shell version: 2.6.3

connecting to: test

> \_

Since you’re in the console, let’s take it for a spin. Firstly we’ll have a [look at all the databases](http://docs.mongodb.org/manual/tutorial/getting-started/#select-a-database) that are there right now:

> show dbs

Assuming this is a clean installation, there shouldn’t be much to see:

> show dbs

admin (empty)

local 0.078GB

That’s great, but [as you can see there’s loads of documentation](http://docs.mongodb.org/manual/tutorial/getting-started/) on how to play with MongoDB from the shell. The shell is a really great environment for trying out queries and looking at things from the point-of-view of the server. However, I promised you Java, so we’re going to step away from the shell and get on with connecting via Java.

**Getting started with Java**

First, you’re going to want to set up your project/IDE to use the [MongoDB Java Driver](http://docs.mongodb.org/ecosystem/tutorial/getting-started-with-java-driver/#getting-started-with-java-driver). These days IDEs tend to pick up the correct dependencies through your [Gradle](http://www.gradle.org/) or [Maven](http://maven.apache.org/) configuration, so I’m just going to cover configuring these.

At the time of writing, the latest version of the Java driver is 2.12.3 - this is designed to work with the MongoDB 2.6 series.

**Gradle**

You’ll need to add the following to your dependencies in build.gradle:

compile 'org.mongodb:mongo-java-driver:2.12.3'

**Maven**

For maven, you’ll want:

<dependencies>

<dependency>

<groupId>org.mongodb</groupId>

<artifactId>mongo-java-driver</artifactId>

<version>2.12.3</version>

</dependency>

</dependencies>

Alternatively, if you’re really old-school and like maintaining your dependencies the hard way, you can always [download the JAR file](http://central.maven.org/maven2/org/mongodb/mongo-java-driver/).

If you don’t already have a project that you want to try with MongoDB, I’ve created a [series of unit tests on github](https://github.com/trishagee/mongodb-getting-started) which you can use to get a feel for working with MongoDB and Java.

**Connecting via Java**

Assuming you’ve resolved your dependencies and you’ve set up your project, you’re ready to connect to MongoDB from your Java application.

Since MongoDB is a document database, you might not be surprised to learn that you don’t connect to it via traditional SQL/relational DB methods like JDBC. But it’s simple all the same:

MongoClient mongoClient = new MongoClient(new MongoClientURI("mongodb://localhost:27017"));

Where I’ve put mongodb://localhost:27017, you’ll want to put the address of where you’ve installed MongoDB. There’s more detailed information on how to create the correct URI, including how to connect to a [Replica Set](http://docs.mongodb.org/manual/replication/), in the [MongoClientURI documentation](http://api.mongodb.org/java/2.12/com/mongodb/MongoURI.html).

If you’re connecting to a local instance on the default port, you can simply use:

MongoClient mongoClient = new MongoClient();

Note that this does throw a checked Exception, UnknownHostException. You’ll either have to catch this or declare it, depending upon what your policy is for exception handling.

The MongoClient is your route in to MongoDB, from this you’ll get your database and collections to work with (more on this later). Your instance of MongoClient (e.g. mongoClient above) will ordinarily be a singleton in your application. However, if you need to connect via different credentials (different user names and passwords) you’ll want a MongoClient per set of credentials.

It is important to limit the number of MongoClient instances in your application, hence why we suggest a singleton - the MongoClient is effectively the connection pool, so for every new MongoClient, you are opening a new pool. Using a single MongoClient (and optionally [configuring its settings](http://api.mongodb.org/java/2.12/com/mongodb/MongoClientOptions.Builder.html)) will allow the driver to correctly manage your connections to the server. This MongoClient singleton is [safe to be used by multiple threads](http://docs.mongodb.org/ecosystem/drivers/java-concurrency/).

One final thing you need to be aware of: you want your application to shut down the connections to MongoDB when it finishes running. Always make sure your application or web server calls MongoClient.close() when it shuts down.

Try out connecting to MongoDB by getting the test in [Exercise1ConnectingTest](https://github.com/trishagee/mongodb-getting-started/blob/master/src/test/java/com/mechanitis/mongodb/gettingstarted/Exercise1ConnectingTest.java) to pass.

**Where are my tables?**

MongoDB doesn’t have tables, rows, columns, joins etc. There are some new concepts to learn when you’re using it, but nothing too challenging.

While you still have the concept of a [database](http://docs.mongodb.org/manual/reference/glossary/#term-database), the [documents](http://docs.mongodb.org/manual/reference/glossary/#term-document) (which we’ll cover in more detail later) are stored in [collections](http://docs.mongodb.org/manual/reference/glossary/#term-collection), rather than your database being made up of tables of data. But it can be helpful to think of documents like rows and collections like tables in a traditional database. And collections can have [indexes](http://docs.mongodb.org/manual/core/indexes-introduction/) like you’d expect.

**Selecting Databases and Collections**

You’re going to want to define which databases and collections you’re using in your Java application. If you remember, a few sections ago we used the MongoDB shell to show the databases in your MongoDB instance, and you had an admin and a local.

Creating and getting a database or collection is extremely easy in MongoDB:

DB database = mongoClient.getDB("TheDatabaseName");

You can replace "TheDatabaseName" with whatever the name of your database is. If the database doesn’t already exist, it will be created automatically the first time you insert anything into it, so there’s no need for null checks or exception handling on the off-chance the database doesn’t exist.

Getting the collection you want from the database is simple too:

DBCollection collection = database.getCollection("TheCollectionName");

Again, replacing "TheCollectionName" with whatever your collection is called.

If you’re playing along with the test code, you now know enough to get the tests  
in [Exercise2MongoClientTest](https://github.com/trishagee/mongodb-getting-started/blob/master/src/test/java/com/mechanitis/mongodb/gettingstarted/Exercise2MongoClientTest.java) to pass.

**An introduction to documents**

Something that is, hopefully, becoming clear to you as you work through the examples in this blog, is that MongoDB is different from the traditional relational databases you’ve used. As I’ve mentioned, there are collections, rather than tables, and documents, rather than rows and columns.

Documents are much more flexible than a traditional row, as you have a dynamic schema rather than an enforced one. You can evolve the document over time without incurring the cost of schema migrations and tedious update scripts. But I’m getting ahead of myself.

Although documents don’t look like the tables, columns and rows you’re used to, they should look familiar if you’ve done anything even remotely JSON-like. Here’s an example:

person = {

\_id: "jo",

name: "Jo Bloggs",

age: 34,

address: {

street: "123 Fake St",

city: "Faketon",

state: "MA",

zip: &#x201C;12345&#x201D;

}

books: [ 27464, 747854, ...]

}

There are a few interesting things to note:

1. Like JSON, documents are structures of name/value pairs, and the values can be one of a number of [primitive types](http://docs.mongodb.org/manual/reference/bson-types/), including Strings and various number types.
2. It also supports nested documents - in the example above, address is a subdocument inside the person document. Unlike a relational database, where you might store this in a separate table and provide a reference to it, in MongoDB if that data benefits from always being associated with its parent, you can embed it in its parent.
3. You can even store an array of values. The books field in the example above is an array of integers that might represent, for example, IDs of books the person has bought or borrowed.

You can find out more detailed information about Documents in [the documentation](http://docs.mongodb.org/manual/core/document/).

**Creating a document and saving it to the database**

In Java, if you wanted to create a document like the one above, you’d do something like:

List<Integer> books = Arrays.asList(27464, 747854);

DBObject person = new BasicDBObject("\_id", "jo")

.append("name", "Jo Bloggs")

.append("address", new BasicDBObject("street", "123 Fake St")

.append("city", "Faketon")

.append("state", "MA")

.append("zip", 12345))

.append("books", books);

At this point, it’s really easy to save it into your database:

MongoClient mongoClient = new MongoClient();

DB database = mongoClient.getDB("Examples");

DBCollection collection = database.getCollection("people");

>collection.insert(person);

Note that the first three lines are set-up, and you don’t need to re-initialize those every time.

Now if we look inside MongoDB, we can see that the database has been created:

> show dbs

Examples 0.078GB

admin (empty)

local 0.078GB\_

…and we can see the collection has been created as well:

> use Examples

switched to db Examples

> show collections

people

system.indexes

> \_

…finally, we can see the our person, “Jo”, was inserted:

> db.people.findOne()

{

"\_id" : "jo",

"name" : "Jo Bloggs",

"age": 34,

"address" : {

"street" : "123 Fake St",

"city" : "Faketon",

"state" : "MA",

"zip" : "12345"

},

"books" : [

27464,

747854

]

}

> \_

As a Java developer, you can see the similarities between the Document that’s stored in MongoDB, and your domain object. In your code, that person would probably be a Person class, with simple primitive fields, an array field, and an Address field.

So rather than building your DBObject manually like the above example, you’re more likely to be converting your domain object into a DBObject. It’s best not to have the MongoDB-specific DBObject class in your domain objects, so you might want to create a PersonAdaptor that converts your Person domain object to a DBObject:

public static final DBObject toDBObject(Person person) {

return new BasicDBObject("\_id", person.getId())

.append("name", person.getName())

.append("address", new BasicDBObject("street", person.getAddress().getStreet())

.append("city", person.getAddress().getTown())

.append("phone", person.getAddress().getPhone()))

.append("books", person.getBookIds());

}

As before, once you have the DBObject, you can save this into MongoDB:

collection.insert(PersonAdaptor.toDBObject(myPerson));

Now you’ve got all the basics to get the tests in [Exercise3InsertTest](https://github.com/trishagee/mongodb-getting-started/blob/master/src/test/java/com/mechanitis/mongodb/gettingstarted/Exercise3InsertTest.java) to pass.

**Getting documents back out again**

Now you’ve saved a Person to the database, and we’ve seen it in the database using the shell, you’re going to want to get it back out into your Java application. In this post, we’re going to cover the very basics of retrieving a document - in a later post we’ll cover more complex querying.

You’ll have guessed by the fact that MongoDB is a document database that we’re not going to be using SQL to query. Instead, we query by example, building up a document that looks like the document we’re looking for. So if we wanted to look for the person we saved into the database, “Jo Bloggs”, we remember that the \_id field had the value of “jo”, and we create a document that matches this shape:

DBObject query = new BasicDBObject("\_id", "jo");

DBCursor cursor = collection.find(query);

As you can see, the find method returns a cursor for the results. Since \_id needs to be unique, we know that if we look for a document with this ID, we will find only one document, and it will be the one we want:

DBObject jo = cursor.one();

Earlier we saw that documents are simply made up of name/value pairs, where the value can be anything from a simple String or primitive, to more complex types like arrays or subdocuments. Therefore in Java, we can more or less treat DBObject as a Map<String, Object>. So if we wanted to look at the fields of the document we got back from the database, we can get them with:

(String)cursor.one().get("name");

Note that you’ll need to cast the value to a String, as the compiler only knows that it’s an Object.