**🔹 Code Explanation**

{

"Name": "sachin",

"address": {

"city": "Bandra",

"country": "IND",

"State": "Mumbai"

},

"PhoneNumbers": ["111-222-333", "999-888-777-66"]

}

**✅ Breaking it down:**

* "Name": "sachin"
  + This is a **key-value pair**.
  + Key = "Name"
  + Value = "sachin"
* "address": { ... }
  + The value of "address" is another JSON **object** (nested document).
  + Inside it, we have:
    - "city": "Bandra"
    - "country": "IND"
    - "State": "Mumbai"

👉 This shows **hierarchical/nested data** storage, which SQL finds difficult but MongoDB handles easily.

* "PhoneNumbers": ["111-222-333", "999-888-777-66"]
  + The value here is a **JSON array** (a list of values).
  + It contains multiple phone numbers for the same person.
  + In SQL, you’d usually need another table for multiple phone numbers, but in MongoDB, you can store them inside the same document.

**🔹 How MongoDB Stores This**

* This entire block is **one document** inside a MongoDB **collection** (similar to a table in SQL).
* A collection can have many such documents, each representing one person.
* Example: A Users collection could have documents for Sachin, Rahul, Priya, etc.

**🔹 Why This is Powerful**

* **Flexible schema** → You can add new fields anytime without altering the database structure.  
  (e.g., tomorrow you can add "Email": "sachin@gmail.com" without any issue).
* **Nested data** → Can store complex, real-world information in a single document.
* **Arrays supported** → Multiple values like phone numbers, addresses, tags can be stored easily.

Eg2:

{

"emp\_details": [

{

"emp\_name": "sachin",

"email": "sachin@mi.com",

"job\_profile": "batsman"

},

{

"emp\_name": "dravid",

"email": "dravid@rcb.com",

"job\_profile": "keeper"

},

{

"emp\_name": "dhoni",

"email": "dhoni@csk.com",

"job\_profile": "captain"

}

]

}

**🔹 Explanation**

* "emp\_details": [ ... ]
  + Here "emp\_details" is a **key**.
  + Its value is an **array** (notice the square brackets [ ]).
  + Inside the array, we have **multiple employee records**, each written as an object {...}.
* Each employee object contains:
  + "emp\_name" → Employee’s name
  + "email" → Employee’s email ID
  + "job\_profile" → Their role/job title

For example:

* Sachin → Batsman
* Dravid → Keeper
* Dhoni → Captain

**🔹 Why this is useful in MongoDB**

👉 Instead of creating multiple tables like in SQL (Employees, JobProfiles, etc.), MongoDB allows you to keep everything **together in one nested structure**.

* **Array of Objects** → Makes it easy to store multiple employees under "emp\_details".
* **Flexible schema** → Tomorrow, you can add "team": "India" for only Dhoni, without touching Sachin’s or Dravid’s records.

**🔹 Equivalent SQL Representation**

If you wanted the same thing in SQL, you’d likely have a table like this:

| **emp\_name** | **email** | **job\_profile** |
| --- | --- | --- |
| sachin | sachin@mi.com | batsman |
| dravid | dravid@rcb.com | keeper |
| dhoni | dhoni@csk.com | captain |

👉 But notice:

1. SQL uses **rows & columns**.
2. MongoDB uses **documents with nested arrays**, which is closer to real-world JSON-like data.

**MongoDB Shell Commands Explained (Updated)**

**1. Add <mongodb\_home>\bin folder to the Path environment variable**

* The **bin folder** contains MongoDB’s important files (mongo.exe for shell, mongod.exe for server).
* Adding this folder to the **Path variable** lets you run MongoDB commands from any folder in Command Prompt without typing the full path.

👉 Example Path:

C:\Program Files\MongoDB\Server\4.4\bin

**2. Open MongoDB Shell**

* Open **Command Prompt** and type:

mongo

* This starts the **Mongo shell**, which is like a chat window to interact with MongoDB.

**3. List All Databases**

* To see existing databases, type:

show dbs

👉 Output might look like:

admin 0.000GB

config 0.000GB

local 0.000GB

These are the default system databases that MongoDB creates automatically.

**4. Create a New Database and Insert Data**

Now let’s actually create our own database and put some data inside it.

**Step 4.1 → Create (or Switch to) a New Database**

use studentDB

* This tells MongoDB:  
  “I want to work with a database called **studentDB**.”
* If it already exists → MongoDB switches to it.
* If it doesn’t exist → MongoDB prepares it, but it will only become *real* once we insert some data.

👉 Output:

switched to db studentDB

**Step 4.2 → Insert Data into a Collection**

Now we add a collection (like a table in SQL) and insert one record (called a *document* in MongoDB).

db.students.insertOne({name: "Rahul", age: 21, course: "Java"})

Let’s break it down:

* db → Refers to the current database (studentDB).
* students → The collection name (like a table). If it doesn’t exist, MongoDB will create it automatically.
* insertOne() → Command to insert **one document (record)**.
* {name: "Rahul", age: 21, course: "Java"} → The actual data in JSON format (key-value pairs).

👉 Output:

{

"acknowledged" : true,

"insertedId" : ObjectId("64bdf3a22f0c...")

}

This means MongoDB successfully stored the data and gave it a unique ID.

**Step 4.3 → Insert Multiple Records**

You can also insert multiple records in one go:

db.students.insertMany([

{name: "Anita", age: 22, course: "Python"},

{name: "Vikram", age: 23, course: "Spring Boot"},

{name: "Meena", age: 20, course: "MongoDB"}

])

* insertMany() → Lets you insert multiple documents at once.
* The square brackets [] hold multiple JSON objects {}. Each object is one record.

👉 Output:

{

"acknowledged" : true,

"insertedIds" : [

ObjectId("64bdf3a22f0c..."),

ObjectId("64bdf3a22f0d..."),

ObjectId("64bdf3a22f0e...")

]

}

MongoDB confirms all records were stored and shows their unique IDs.

**Step 4.4 → Verify the Data**

To see what’s inside the collection:

db.students.find()

👉 Output:

{ "\_id" : ObjectId("..."), "name" : "Rahul", "age" : 21, "course" : "Java" }

{ "\_id" : ObjectId("..."), "name" : "Anita", "age" : 22, "course" : "Python" }

{ "\_id" : ObjectId("..."), "name" : "Vikram", "age" : 23, "course" : "Spring Boot" }

{ "\_id" : ObjectId("..."), "name" : "Meena", "age" : 20, "course" : "MongoDB" }

* find() shows all documents in the collection.
* Notice: \_id field is automatically created by MongoDB to uniquely identify each document.

✅ So now, we:

1. Created our own DB (studentDB).
2. Created a collection (students).
3. Inserted single & multiple records.
4. Verified data using find().

The command used is:

db.students.find().pretty()

**✅ Explanation of the command:**

1. **db** → Refers to the current database you are using.  
   Example: If you had done use fsDB, then db means *fsDB*.
2. **customer** → This is the **collection** name (similar to a table in SQL).  
   It stores multiple documents (rows in SQL terms).
3. **find()** → This is used to **fetch all documents** from the collection.  
   By default, it shows data in a raw one-line JSON format (hard to read).
4. **pretty()** → This makes the output **formatted and easy to read** (well-spaced JSON).  
   Without pretty(), you would see compressed JSON.

**✅ Output shown in your example:**

{

"\_id" : ObjectId("643f5667caab5d65806450fc"),

"cno" : 10,

"cname" : "sachin",

"cadd" : "mi",

"avg" : 54.5

}

{

"\_id" : ObjectId("643f56c8caab5d65806450fd"),

"cno" : 7,

"cname" : "dhoni",

"cadd" : "CSK",

"avg" : 53

}

{

"\_id" : ObjectId("643f56d8caab5d65806450fe"),

"cno" : 18,

"cname" : "kohli",

"cadd" : "RCB",

"avg" : 54

}

**✅ Breaking down the fields in each document:**

* **\_id** : A unique value automatically generated by MongoDB (called ObjectId).
  + It ensures each document is uniquely identifiable.
  + If you don’t provide it while inserting data, MongoDB creates it by default.
* **cno** : A custom field (player number in this case).
* **cname** : The name of the player (sachin, dhoni, kohli).
* **cadd** : The address/team of the player (MI, CSK, RCB).
* **avg** : The batting average of the player (numeric value).

👉 So, this step basically means:  
*"Show me all the records/documents stored in the customer collection, but display them in a neat JSON format that’s easy to read."*

**find().pretty()**

* Command:
* db.customer.find().pretty()
* What it does:  
  Same as find(), it fetches all documents.  
  The only difference is: it **formats the output neatly** with proper indentation and line breaks.
* How the output looks:  
  This is **human-friendly** and easier to read.

Example:

{

"\_id" : ObjectId("643f5667caab5d65806450fc"),

"cno" : 10,

"cname" : "sachin",

"cadd" : "mi",

"avg" : 54.5

}

{

"\_id" : ObjectId("643f56c8caab5d65806450fd"),

"cno" : 7,

"cname" : "dhoni",

"cadd" : "CSK",

"avg" : 53

}

**✅ Key Difference**

| **Aspect** | **find()** | **find().pretty()** |
| --- | --- | --- |
| Output format | Compact, single-line JSON | Formatted, indented JSON |
| Human readability | Harder to read | Easier to read |
| Use case | Fast viewing, machine processing | Debugging, analysis by developers |

For mongodb lectures

<https://www.mongodb.com/docs/manual/introduction/>

db.students.insertOne(

{ cno: 9, cname: 'lara', mobileNo: [9997776655, 676676554] }

)

🔍 Explanation:

db.customer

Here, db means the current database you are working with.

customer is the collection name (like a table in SQL).

insertOne({...})

This command is used to insert one document into the collection.

A document = one record (in JSON format).

Inside { ... }

cno: 9 → A field called cno with value 9 (like a column-value pair in SQL).

cname: 'lara' → A field called cname with value lara.

mobileNo: [9997776655, 676676554] → This is the important part.

Here, mobileNo is a field.

Its value is an array containing two numbers.

Arrays allow us to store multiple values for one field (something SQL struggles with).

Result / Output

{

"acknowledged" : true,

"insertedId" : ObjectId("643f5bc7caab5d6580645101")

}

"acknowledged" : true → MongoDB confirms the insert was successful.

"insertedId" → MongoDB automatically generates a unique \_id (here ObjectId("643f5bc7...")) for the new document.

This \_id works like the primary key in SQL.

📦 Final Document Stored:

{

"\_id": ObjectId("643f5bc7caab5d6580645101"),

"cno": 9,

"cname": "lara",

"mobileNo": [9997776655, 676676554]

}

To find docs of a collection with condition

db.students.find({cname:'lara'}).pretty()

🔍 Explanation

db.customer

db = the current database you’re working with.

customer = the collection (similar to a table in SQL).

find({...})

find() is used to search and return documents from the collection.

Inside {...} we write the condition (similar to the WHERE clause in SQL).

Here:

{ cname: 'lara' }

means → return all documents where the field cname has the value "lara".

pretty()

This makes the output formatted and readable (with line breaks and indentation).

Without .pretty(), MongoDB will show everything in one line, which is harder to read.

📦 Output Document

{

"\_id" : ObjectId("643f5bc7caab5d6580645101"),

"cno" : 9,

"cname" : "lara",

"mobileNo" : [

9997776655,

676676554

]

}

✅ Explanation of the Result

\_id → auto-generated unique ID for this document (works like primary key).

cno → customer number 9.

cname → the name is "lara".

mobileNo → an array with two mobile numbers stored inside.

📊 SQL Equivalent

If you think in SQL terms, the same query would look like:

SELECT \* FROM customer WHERE cname = 'lara';

⚡ So in short:

find({condition}) → works like WHERE in SQL.

pretty() → just makes the output human-friendly.

**🔹 1. Using remove()**

db.customer.remove({cno:63})

* **db.customer** → select the customer collection.
* **remove({...})** → deletes documents that match the given condition.
* {cno:63} → condition → delete all documents where the field cno equals 63.

⚠️ Note: remove() can delete **multiple documents** if more than one match is found.

**🔹 2. Using deleteOne()**

db.customer.deleteOne({cno:45})

* **deleteOne({...})** → deletes **only one document** that matches the condition.
* {cno:45} → condition → delete one document where cno = 45.
* Even if multiple documents have cno:45, only the **first match** will be deleted.

✅ The result:

{ "acknowledged" : true, "deletedCount" : 1 }

* "acknowledged" : true → means MongoDB received and executed the delete request.
* "deletedCount" : 1 → shows **how many documents were deleted**. Here, 1.

**⚡ Extra Knowledge**

MongoDB also has:

1. **deleteMany()**
2. db.customer.deleteMany({cname: "lara"})

→ deletes **all documents** where cname = lara.

1. **Delete All Documents in a Collection**
2. db.customer.deleteMany({})

→ deletes everything inside the collection, but keeps the collection itself.

**📊 SQL Equivalent**

* remove({cno:63}) or deleteOne({cno:45}) in MongoDB ≈

DELETE FROM customer WHERE cno = 63;

DELETE FROM customer WHERE cno = 45 LIMIT 1;

**🔹 1. Dropping a Collection**

db.customer.drop()

* **db.customer** → points to the customer collection in the current database.
* **.drop()** → completely removes the entire collection (all documents + collection structure).

✅ Output:

true

This means the collection was successfully dropped.

**🔹 2. Checking Collections**

show collections

* This command lists all the collections present in the current database.
* After dropping, customer collection will no longer appear.

**🔹 3. Checking Databases**

show dbs

* Lists all logical databases available in MongoDB.
* Notice that MYDB still appears even though we dropped the customer collection.

👉 Why?  
Because a **database name remains** in the list until it becomes completely empty (no collections, no data). Once all collections are deleted, and you restart MongoDB, the empty DB might disappear from the list.

**⚡ Key Difference**

* **Deleting a Document** (with deleteOne, deleteMany, remove) → only removes some rows (records) inside the collection.
* **Dropping a Collection** (db.collection.drop()) → removes the **entire table/collection**, not just the data.
* **Dropping a Database** (db.dropDatabase()) → removes the **whole database** with all its collections.

👉 Example in SQL Terms:

* DELETE FROM customer WHERE cno=63; → delete some rows (like deleteOne, deleteMany).
* DROP TABLE customer; → same as db.customer.drop().
* DROP DATABASE MYDB; → same as db.dropDatabase().