24.7.25

MONGODB ASSIGNMENT

DATA TYPES

1. String

Query:

db.students.insertOne({ name: "Alice" })

db.students.find({ name: "Alice" })

Output:

```
test> db.students.insertOne({ name: "Alice" })
... db.students.find({ name: "Alice" })
[ { _id: ObjectId('6881d9fca7747b0e0eeec4a9'), name: 'Alice' } ]
test>
```

Explanation: String is the most common data type in MongoDB. It's used to store text. MongoDB stores it in UTF-8 format.

---

2. Integer

Query:

db.students.insertOne({ name: "Bob", age: 21 })

db.students.find({ name: "Bob" })

Output:

```
test> db.students.insertOne({ name: "Bob", age: 21 })
... db.students.find({ name: "Bob" })
[ { _id: ObjectId('6881da2fa7747b0e0eeec4aa'), name: 'Bob', age: 21 } ]
test>
```

Explanation: Integer is used to store whole numbers like age, count, etc. MongoDB supports 32-bit and 64-bit integers.

---

3. Double

Query:

db.students.insertOne({ name: "Charlie", marks: 89.5 })

db.students.find({ name: "Charlie" })

Output:

```
test> db.students.insertOne({ name: "Charlie", marks: 89.5 })
... db.students.find({ name: "Charlie" })
[
  {
    _id: ObjectId('6881da5ca7747b0e0eeec4ab'),
    name: 'Charlie',
    marks: 89.5
  }
]
test>
```

Explanation: Double is a data type for storing floating-point numbers. Useful for storing decimals like marks, prices, etc.

---

4. Boolean

Query:

db.students.insertOne({ name: "David", passed: true })

db.students.find({ name: "David" })

Output:

```
test> db.students.insertOne({ name: "David", passed: true })
... db.students.find({ name: "David" })
[
  {
    _id: ObjectId('6881da86a7747b0e0eeec4ac'),
    name: 'David',
    passed: true
  }
]
```

Explanation: Boolean holds either true or false. Useful for flags like pass/fail, yes/no, active/inactive.

---

5. Null

Query:

db.students.insertOne({ name: "Eve", phone: null })

db.students.find({ name: "Eve" })

Output:

```
test> db.students.insertOne({ name: "Eve", phone: null })
... db.students.find({ name: "Eve" })
[
  {
    _id: ObjectId('6881dab0a7747b0e0eeec4ad'),
    name: 'Eve',
    phone: null
  }
```

Explanation: Null represents missing or unknown data. Use when value is intentionally not set.

---

6. Array

Query:

db.students.insertOne({ name: "Frank", skills: ["Java", "Python"] })

db.students.find({ name: "Frank" })

Output:

```
test> db.students.insertOne({ name: "Frank", skills: ["Java", "Python"] })
... db.students.find({ name: "Frank" })
[
  {
    _id: ObjectId('6881daeda7747b0e0eeec4ae'),
    name: 'Frank',
    skills: [ 'Java', 'Python' ]
  }
]
```

Explanation: Array holds multiple values. Can store strings, numbers, or even documents. Very flexible.

---

7. Object / Embedded Document

Query:

db.students.insertOne({

name: "Grace",

address: { city: "Chennai", pin: 600001 }

})

db.students.find({ name: "Grace" })

Output:

```
test> db.students.insertOne({
...   name: "Grace",
...   address: { city: "Chennai", pin: 600001 }
... })
... db.students.find({ name: "Grace" })
[
  {
    _id: ObjectId('6881db12a7747b0e0eeec4af'),
    name: 'Grace',
    address: { city: 'Chennai', pin: 600001 }
  }
]
```

Explanation: Embedded documents allow nesting. Helpful for storing related structured data together.

---

8. ObjectId

Query:

db.students.insertOne({ name: "Hannah" })

db.students.find({ name: "Hannah" })

Output:

```
test> db.students.insertOne({ name: "Hannah" })
... db.students.find({ name: "Hannah" })
[ { _id: ObjectId('6881db51a7747b0e0eeec4b0'), name: 'Hannah' } ]
test>
```

Explanation: MongoDB automatically creates a unique ObjectId for each document unless you specify your own unique _id.

---

9. Undefined

Query:

db.students.insertOne({ name: "Ian", duration: undefined })

db.students.find({ name: "Ian" })

Output:

```
test> db.students.insertOne({ name: "Ian", duration: undefined })
... db.students.find({ name: "Ian" })
[
  {
    _id: ObjectId('6881db80a7747b0e0eeec4b1'),
    name: 'Ian',
    duration: null
  }
]
```

Explanation: Deprecated. Rarely used now. null is preferred. If undefined is used, the field usually disappears.

---

10. Binary

Query:

db.files.insertOne({ name: "img", data: new BinData(0, "SGVsbG8=") })

db.files.find({ name: "img" })

Output:

```
est> db.files.insertOne({ name: "img", data: new BinData(0, "SGVsbG8=") })
.. db.files.find({ name: "img" })

{
  _id: ObjectId('6881dbaaa7747b0e0eeec4b2'),
  name: 'img',
  data: Binary.createFromBase64('SGVsbG8=', 0)
}
```

Explanation: Binary data is used for storing files like images or PDFs. It stores raw binary values.

---

11. Date

Query:

db.students.insertOne({ name: "Jack", joined: new Date() })

db.students.find({ name: "Jack" })

Output:

```
test> db.students.insertOne({ name: "Jack", joined: new Date() })
... db.students.find({ name: "Jack" })
[
  {
    _id: ObjectId('6881dbd5a7747b0e0eeec4b3'),
    name: 'Jack',
    joined: ISODate('2025-07-24T07:08:05.730Z')
  }
]
```

Explanation: Date is stored in UTC format. Useful for timestamps, logs, event tracking, etc.

---

12. MinKey / MaxKey

Query:

db.test.insertOne({ key: MinKey() })

db.test.insertOne({ key: MaxKey() })

db.test.find()

Output:

```
test> db.test.insertOne({ key: MinKey() })
... db.test.insertOne({ key: MaxKey() })
... db.test.find()
[
  { _id: ObjectId('6881dbf8a7747b0e0eeec4b4'), key: MinKey() },
  { _id: ObjectId('6881dbf8a7747b0e0eeec4b5'), key: MaxKey() }
]
test>
```

Explanation: MinKey is the smallest possible BSON value. MaxKey is the largest. Used internally in range queries.

---

13. Symbol

Query:

db.symbols.insertOne({ sym: new Symbol("alpha") })

db.symbols.find()

Output:

```
test> db.symbols.insertOne({ sym: "alpha" })
... db.symbols.find()
[ { _id: ObjectId('6881dc8ea7747b0e0eeec4b6'), sym: 'alpha' } ]
test>
```

Explanation: Symbol is similar to string. Rare and deprecated in most cases. Automatically shown as string in shell.

---

14. Regular Expression

Query:

db.students.find({ name: { $regex: /an/i } })

Output:

```
test> db.students.find({ name: { $regex: /an/i } })
[
  {
    _id: ObjectId('6881daeda7747b0e0eeec4ae'),
    name: 'Frank',
    skills: [ 'Java', 'Python' ]
  },
  { _id: ObjectId('6881db51a7747b0e0eeec4b0'), name: 'Hannah'
  {
    _id: ObjectId('6881db80a7747b0e0eeec4b1'),
    name: 'Ian',
    duration: null
  },
  { _id: ObjectId('6881dcb2a7747b0e0eeec4b7'), name: /an/i }
]
```

Explanation: Used for pattern matching. For example, find names that contain "an" (case insensitive).

---

15. JavaScript

Query:

db.scripts.insertOne({ code: function() { return "hi" } })

db.scripts.find()

Output:

```
test> db.scripts.insertOne({ code: function() { return "hi" } })
... db.scripts.find()
[
  {
    _id: ObjectId('6881dd70a7747b0e0eeec4b8'),
    code: Code('function() { return "hi" }')
  }
]
test>
```

Explanation: Allows storing JavaScript code inside documents. Rarely used in modern apps, used in system.js collection.

---

16. Timestamp

Query:

db.logs.insertOne({ event: "login", time: Timestamp() })

db.logs.find()

Output:

```
test> db.logs.insertOne({ event: "login", time: Timestamp() })
... db.logs.find()
[
  {
    _id: ObjectId('6881dd9ba7747b0e0eeec4b9'),
    event: 'login',
    time: Timestamp({ t: 1753341339, i: 1 })
  }
]
```

Explanation: Used for tracking changes to documents. Similar to Date but optimized for replication events.

---

17. Decimal128

Query:

db.products.insertOne({ name: "Gold", price: NumberDecimal("99999.99") })

db.products.find()

Output:

Explanation: Decimal128 supports high-precision decimals. Ideal for financial or scientific values.

RELATIONS

1. One-to-One: Embedded Address

   DATABASE : use gfg

   INSERT:

   db.student.insertOne({
   StudentName: "GeeksA",
   StudentId: "g_f_g_1209",
   Branch: "CSE",
   PermanentAddress: {
   permaAddress: "5th Cross, Sector 1",
   City: "Delhi",
   PinCode: 202333
   }
   })

   A. Query to view Permanent Address:

   db.student.find({ StudentName: "GeeksA" }, {
   "PermanentAddress.permaAddress": 1 }).pretty()

```
gfg> db.student.find({ StudentName: "GeeksA" }, { "PermanentAddress.permaAddress": 1 }).pretty()
[
  {
    _id: ObjectId('6881fcd606c5acdaa0eec4a9'),
    PermanentAddress: { permaAddress: '5th Cross, Sector 1' }
  },
  { _id: ObjectId('6881fce806c5acdaa0eec4aa') }
]
```

2. One-to-Many: Multiple Addresses (Array of Embedded Docs)

DATABASE :use gfg

INSERT:

```
db.student.insertOne({
  StudentName: "GeeksA",
  StudentId: "g_f_g_1209",
  Branch: "CSE",
  Address: [
    {
      Type: "Permanent",
      AddressLine: "5th Cross, Sector 1",
      City: "Delhi",
      PinCode: 202333
    },
    {
      Type: "Current",
      AddressLine: "Flat No 302, Palm Towers",
      City: "Mumbai",
      PinCode: 334509
    }
  ]
})
```

B. Query to fetch both addresses:

```
db.student.find(
  { StudentName: "GeeksA" },
  { "Address.Type": 1, "Address.AddressLine": 1 }
).pretty()
```

```
gfg> db.student.find(
...   { StudentName: "GeeksA" },
...   { "Address.Type": 1, "Address.AddressLine": 1 }
... ).pretty()
[
  { _id: ObjectId('6881fcd606c5acdaa0eec4a9') },
  {
    _id: ObjectId('6881fce806c5acdaa0eec4aa'),
    Address: [
      { Type: 'Permanent', AddressLine: '5th Cross, Sector 1' },
      { Type: 'Current', AddressLine: 'Flat No 302, Palm Towers' }
    ]
```

3. One-to-Many: Document Reference (classes stored separately)

```
use gfg

// Class 1 Document
db.classes.insertOne({
  TeacherId: "g_f_g_1209",
  ClassId: "C_123",
  ClassName: "GeeksA",
  StudentCount: 23,
  Subject: "Science"
})

// Class 2 Document
db.classes.insertOne({
  TeacherId: "g_f_g_1209",
  ClassId: "C_234",
  ClassName: "GeeksB",
```

```
  StudentCount: 33,
  Subject: "Maths"
})

// Teacher Document referencing class IDs
db.teacher.insertOne({
  teacherName: "Sunita",
  TeacherId: "g_f_g_1209",
  classIds: ["C_123", "C_234"]
})
```

C. Query to get teacher's class references:

```
db.teacher.findOne({ teacherName: "Sunita" }, { classIds: 1 })
```

```
gfg> db.teacher.findOne({ teacherName: "Sunita" }, { classIds: 1 })
{
  _id: ObjectId('6881fd2706c5acdaa0eec4ad'),
  classIds: [ 'C_123', 'C_234' ]
}
```

D. Then fetch classes manually using $in:

```
db.classes.find({ ClassId: { $in: ["C_123", "C_234"] } }).pretty()
```

```
gfg> db.classes.find({ ClassId: { $in: ["C_123", "C_234"] } }).pretty()
[
  {
    _id: ObjectId('6881fd2706c5acdaa0eec4ab'),
    TeacherId: 'g_f_g_1209',
    ClassId: 'C_123',
    ClassName: 'GeeksA',
    StudentCount: 23,
    Subject: 'Science'
  },
  {
    _id: ObjectId('6881fd2706c5acdaa0eec4ac'),
    TeacherId: 'g_f_g_1209',
    ClassId: 'C_234',
    ClassName: 'GeeksB',
    StudentCount: 33,
    Subject: 'Maths'
  }
]
```