Compatibility Changes with Legacy `mongo` Shell

This page describes differences between `mongosh` and the legacy `mongo` shell. In addition to the alternatives listed here, you can use the mongocompat snippet to access to legacy `mongo` shell APIs. Snippets are an experimental feature, for more information, see Snippets.

```
snippet install mongocompat
```

Deprecated Methods

The following shell methods are deprecated in `mongosh`. Instead, use the methods listed in the **Alternative Resources** column.

Deprecated Method	Alternative Resources
`db.collection.copyTo()`	Aggregation stage: `$out`
`db.collection.count()`	• `db.collection.countDocuments()` • `db.collection.estimatedDocumentCount()`
`db.collection.insert()`	• `db.collection.insertOne()` • `db.collection.insertMany()` • `db.collection.bulkWrite()`
`db.collection.remove()`	• `db.collection.deleteOne()` • `db.collection.deleteMany()` • `db.collection.findOneAndDelete()` • `db.collection.bulkWrite()`
`db.collection.save()`	• `db.collection.insertOne()` • `db.collection.insertMany()` • `db.collection.updateOne()`

Deprecated Method	Alternative Resources
	• `db.collection.updateMany()` • `db.collection.findOneAndUpdate()`
`db.collection.update()`	• `db.collection.updateOne()` • `db.collection.updateMany()` • `db.collection.findOneAndUpdate()` • `db.collection.bulkWrite()`
`DBQuery.shellBatchSize`	• config.set("displayBatchSize", "<value>") • cursor.batchSize()
`Mongo.getSecondaryOk`	`Mongo.getReadPrefMode()`
`Mongo.isCausalConsistency`	`Session.getOptions()`
`Mongo.setSecondaryOk`	`Mongo.setReadPref()`
`rs.secondaryOk`	No longer required. See Read Operations on a Secondary Node.

Read Preference Behavior

Read Operations on a Secondary Node

When using the legacy mongo shell to connect directly to secondary replica set member, you must run `mongo.setReadPref()` to enable secondary reads.

When using `mongosh` to connect directly to a secondary replica set member, you can read from that member if you specify a read preference of either:

- `primaryPreferred`
- `secondary`
- `secondaryPreferred`

To specify a read preference, you can use either:

- The `readPreference` connection string option when connecting to the node.
- The `Mongo.setReadPref()` method.

When using `mongosh` to connect directly to a secondary replica set member, if your read preference is set to `primaryPreferred`, `secondary` or `secondaryPreferred` it is *not* required to run `rs.secondaryOk()`.

`show` Helper Methods

The following `show` helper methods always use a read preference of `primaryPreferred`, even when a different read preference has been specified for the operation:

- `show dbs`
- `show databases`
- `show collections`
- `show tables`

In the legacy `mongo` shell, these operations use the specified read preference.

Write Preference Behavior

Retryable writes are enabled by default in `mongosh`. Retryable writes were disabled by default in the legacy `mongo` shell. To disable retryable writes, use `--retryWrites=false`.

ObjectId Methods and Attributes

These ObjectId() methods work differently in `mongosh` than in the legacy `mongo` shell.

Method or Attribute	`mongo` Behavior	`mongosh` Behavior
`ObjectId.str`	Returns a hexadecimal string: `6419ccfce40afaf9317567b7`	Undefined (Not available)
`ObjectId.valueOf()`	Returns the value of `ObjectId.str`: `6419ccfce40afaf9317567b7`	Returns a formatted string: `ObjectId("6419ccfce40afaf9317567b7")`
`ObjectId.toString()`	Returns a formatted string: `ObjectId("6419ccfce40afaf9317567b7")`	Returns a hexadecimal formatted string: `6419ccfce40afaf9317567b7`

Numeric Values

The legacy `mongo` shell stored numerical values as `doubles` by default.
In `mongosh` numbers are stored as 32 bit integers, `Int32`, or else as `Double` if the value cannot be stored as an `Int32`.

MongoDB Shell continues to support the numeric types that are supported in `mongo` shell. However, the preferred types have been updated to better align with the MongoDB drivers. See mongosh Data Types for more information.

The preferred types for numeric variables are different in MongoDB Shell than the types suggested in the legacy `mongo` shell. The types in `mongosh` better align with the types used by the MongoDB Drivers.

`mongo` type	`mongosh` type
`NumberInt`	`Int32`
`NumberLong`	`Long`
`NumberDecimal`	`Decimal128`

WARNING

Data types may be stored inconsistently if you connect to the same collection using both `mongosh` and the legacy `mongo` shell.

TIP

See also:

For more information on managing types, refer to the schema validation overview.

`--eval` Behavior

`mongosh --eval` does not quote object keys in its ouptut.

To get output suitable for automated parsing, use `EJSON.stringify()`.

```
mongosh --quiet --host rs0/centos1104 --port 27500 \
--eval "EJSON.stringify(rs.status().members.map( \
m => ({'id':m._id, 'name':m.name, 'stateStr':m.stateStr})));" \
| jq
```

After parsing with `jq`, the output resembles this:

```
[
{
"id": 0,
"name": "centos1104:27500",
"stateStr": "PRIMARY"
},
{
"id": 1,
"name": "centos1104:27502",
"stateStr": "SECONDARY"
},
{
"id": 2,
"name": "centos1104:27503",
"stateStr": "SECONDARY"
}
]
```

NOTE

`EJSON` has built in formatting options which may eliminate the need for a parser like `jq`.

For example, the following code produces output that is formatted the same as above.

```
mongosh --quiet --host rs0/centos1104 --port 27500 \
--eval "EJSON.stringify( rs.status().members.map( \
({ _id, name, stateStr }) => ({ _id, name, stateStr })), null, 2);"
```

Limitations on Database Calls

The results of database queries cannot be passed inside the following contexts:

- Class constructor functions
- Non-async generator functions
- Callbacks to `.sort()` on an array

To access to the results of database calls, use async functions, async generator functions, or `.map()`.

Constructors

The following constructors do not work:

```
// This code will fail
class FindResults {
constructor() {
this.value = db.students.find();
}
}
// This code will fail
function listEntries() { return db.students.find(); }
class FindResults {
constructor() {
this.value = listEntries();
}
}
```

Use an `async` function instead:

```
class FindResults {
constructor() {
this.value = ( async() => {
return db.students.find();
} )();
}
}
```

NOTE

You can also create a method that performs a database operation inside a class as an alternative to working with asynchronous JavaScript.

```
class FindResults {
constructor() { }
init() { this.value = db.students.find(); }
}
```

To use this class, first construct a class instance then call the `.init()` method.

Generator Functions

The following generator functions do not work:

```
// This code will fail
function* FindResults() {
yield db.students.findMany();
}
// This code will fail
function listEntries() { return db.students.findMany(); }
function* findResults() {
yield listEntries();
}
```

Use an `async generator function` instead:

```
function listEntries() { return db.students.findMany(); }
async function* findResults() {
yield listEntries();
}
```

Array Sort

The following array sort does not work:

```
// This code will fail
db.getCollectionNames().sort( ( collectionOne, collectionTwo ) => {
return db[ collectionOne ].estimatedDocumentCount() - db[ collectionOne
].estimatedDocumentCount() )
} );
```

Use `.map()` instead.

```
db.getCollectionNames().map( collectionName => {
return { collectionName, size: db[ collectionName ].estimatedDocumentCount() };
} ).sort( ( collectionOne, collectionTwo ) => {
return collectionOne.size - collectionTwo.size;
} ).map( collection => collection.collectionName);
```

This approach to array sort is often more performant than the equivalent unsupported code.