

# MongoDB C# Driver Tutorial

---

Copyright © 2010 10gen Inc. All Rights Reserved | Licensed under Creative Commons  
<http://creativecommons.org/licenses/by-nc-sa/3.0>

## Draft version (dated 2010-09-30)

This document is a draft version. While we feel that the information provided here is quite accurate it is possible it might change in minor ways as we assimilate user feedback and continue implementation of the C# Driver.

## Introduction

This tutorial is an introduction to the 10gen supported C# Driver for MongoDB. It assumes some familiarity with MongoDB and therefore focuses mainly on how to access MongoDB using C#. It is divided into two parts: The C# Driver, and the BSON Library. The C# Driver is built on top of the BSON Library, which is designed to be usable separately from the C# Driver.

In Part 1 we will cover the main classes of the C# Driver: `MongoServer`, `MongoDatabase`, `MongoCollection`, `MongoCursor`, `MongoGridFS`, `MongoGridFSFileInfo` and `SafeMode`.

In Part 2 we will cover the main classes of the BSON Library: `BsonType`, `BsonValue` (and its subclasses), `BsonElement`, `BsonDocument` and `BsonArray`.

## Part 1: The C# Driver

Part 1 of this tutorial is organized in a top down fashion, so you may sometimes have to peek ahead if something isn't making sense right away. You might even want to read Part 2 before Part 1 if you are totally unfamiliar with BSON concepts such as documents and elements.

## References and namespaces

In order to use the C# Driver from your program you must add references to the following two DLLs:

- MongoDB.BsonLibrary.dll
- MongoDB.CSharpDriver.dll

You also should add the following using statements to your source files:

```
using MongoDB.BsonLibrary;  
using MongoDB.CSharpDriver;
```

With very few exceptions the names of classes that you will be using are prefixed with either “Bson” if they are part of the BsonLibrary or “Mongo” if they are part of the C# Driver. This is done to minimize the chances of getting name collisions when you add the two using statements to your program. Some classes that are expected to be used in method arguments (mainly enums and flags) have shorter names that don’t use either of the prefixes.

We like to use C#’s var statement to declare our variables as it leads to shorter, and we feel, more readable code. Visual Studio makes it easy to see a variable’s type if you need to by either hovering the mouse pointer over the variable or by using Intellisense. However, when reading this document you don’t have that capability, so in this document, rather than writing:

```
var server = MongoServer.Create(connectionString);  
var test = server["test"];  
var books = test["books"];
```

as we recommend, we will instead write:

```
MongoServer server = MongoServer.Create(connectionString);  
MongoDatabase test = server["test"];  
MongoCollection<BsonDocument> books = test["books"];
```

so that you can see exactly what types are being used.

## Thread safety

Only a few of the C# Driver classes are thread safe. Among them: MongoServer, MongoDBase, MongoCollection and MongoGridFS. Common classes you will use a lot that are not thread safe include MongoCursor and all the classes from the BsonLibrary (except BsonSymbolTable). A class is not thread safe unless specifically documented as being thread safe.

All static properties and methods of all classes are thread safe.

## MongoServer class

This class serves as the root object for working with MongoDB. An instance of this class on the client represents a MongoDB server that you wish to communicate with. While this class does have public constructors, the recommended way to get an instance of this class is to use the Create factory method.

Each instance of MongoServer maintains a pool of connections to the server. These connections are shared among all the calls to the server. One of the few reasons you might want to call the constructor for MongoServer directly instead of calling the Create factory method is if you want to maintain a separate connection pool for some operations.

Instances of this class are thread safe.

### Connection String

The easiest way to connect to MongoDB is to use a connection string. The standard MongoDB connection string format is a URL in the following format:

```
mongodb://[username:password@]hostname[:port][/database]
```

The username and password should only be present if you are using authentication on the MongoDB server. These credentials will apply to a single database if the database name is present, otherwise they will be the default credentials for all databases. To authenticate against the admin database append "(admin)" to the username.

The port number is optional and defaults to 27017.

If the database name is present then this connection string can also be used with the Create method of MongoDBDatabase. The Create method of MongoServer ignores the database name if present (other than to determine whether the credentials apply to a single database or are the default credentials for all databases).

To connect to a replica set specify the seed list by providing multiple hostnames separated by commas. For example:

```
mongodb://server1,server2:27017,server2:27018
```

This connection string specifies a seed list consisting of three servers (two of which are on the same machine but on different port numbers).

The C# Driver is able to connect to a replica set even if the seed list is incomplete. It will find the primary server even if the primary is not in the seed list as long as at least one of the secondary servers on the seed list responds (the response will contain the full replica set and the name of the current primary).

### Create factory method

The best way to get an instance of MongoServer is to use the Create factory method. This method will return the same instance of MongoServer whenever the same connection string is used, so you don't have to worry about creating a whole bunch of instances if you call Create more than once.

Furthermore, if you are only working with one database you may find it easier to skip calling this method altogether and call the Create factory method of MongoDBDatabase instead.

To connect to MongoDB on localhost you would write code like:

```
string connectionString = "mongodb://localhost";  
MongoServer server = MongoServer.Create(connectionString);
```

Or perhaps:

```
MongoServer server = MongoServer.Create();
```

since connecting to localhost is the default.

The Create method in `MongoServer` ignores the database name if present. However, if the database name is omitted but credentials are present, then `MongoServer` will assume that these credentials are to be used with all databases and will store them in `MongoServer`'s `Credentials` property and use them as the default credentials whenever `GetDatabase` is called without credentials. That makes it real easy to use the same credentials with all databases.

The Create method uses the seed list as the identity of the server when determining if there is already an existing instance of `MongoServer` to return (not the actual members of the replica set as that is not known until after the connection is established and is subject to change).

### SafeMode property

This property represents the default `SafeMode` for this server. It will be inherited by any instances of `MongoDatabase` that are returned by this server instance, but a database's default `SafeMode` can be changed independently of the server's after that. The `SafeMode` class is described further on.

### GetDatabase method

From an instance of `MongoServer` you can get instances of objects representing the databases on that server using the `GetDatabase` method. To get a database object you would write:

```
MongoDatabase test = server.GetDatabase("test");
```

or

```
MongoDatabase test = server["test"];
```

The two forms are completely equivalent. If you are using authentication you have to write slightly different code, as in:

```
MongoCredentials credentials =  
    new MongoCredentials("username", "password");  
MongoDatabase test = server.GetDatabase("test", credentials);
```

or

```
MongoDatabase test = server["test", credentials];
```

`MongoServer` maintains a table of `MongoDatabase` instances by database/credential combinations, and each time you ask for the same database/credentials combination you get the same instance of `MongoDatabase` back, so you don't need to worry about unneeded duplicate instances coming into existence.

## MongoDatabase class

This class represents a database on a MongoDB server. Unless you are using authentication, you would normally have only one instance of this class for each database you want to work with. If you are using authentication then you will end up with an instance of this class for every database/credentials combination you want to work with.

Instance of this class are thread safe.

However, if you are writing a multi-threaded program and doing a sequence of related operations (which is common!), you probably want to be using RequestStart/RequestDone to ensure that related operations on a thread all occur on the same connection to the server.

## Create factory method

Normally you get instances of MongoDatabase by calling GetDatabase on the server instance, but MongoDatabase also has a Create factory method which takes a URL as a parameter. In this case though, the database name is required, and an exception will be thrown if it is missing from the URL. If you choose to use the Create factory method in MongoDatabase you do not have to call the Create factory method in MongoServer (although it will be called for you and the resulting server object is available to you if needed in the new database instance's Server property).

To create an instance of MongoDatabase using the Create factory method you would write:

```
string connectionString = "mongodb://localhost/test";
MongoDatabase test = MongoDatabase.Create(connectionString);
MongoServer server = test.Server; // if needed
```

Note that this is completely equivalent to the following code:

```
string connectionString = "mongodb://localhost/test";
MongoServer server = MongoServer.Create(connectionString);
MongoDatabase test = server.GetDatabase("test");
```

The first form can be advantageous if you don't need immediate access to the server object and also because the database name (and credentials if present) come from the URL and are not hardcoded.

## Server property

This property allows you to navigate back from a MongoDatabase object to the MongoServer object that it belongs to.

## Credentials property

This property allows you to examine the credentials associated with this instance of a MongoDatabase. You cannot change the credentials. If you want to work with different credentials you have to go back to the server object and get a new database object with the new credentials. This is because an instance of

MongoDatabase is normally shared by many parts of your code, and changing the credentials could have unintended side effects.

### SafeMode property

This property represents the default SafeMode for this database. It will be inherited by any instances of MongoClient that are returned by this database instance, but a collection's default SafeMode can be changed independently of the database's after that. The SafeMode class is described further on.

### GridFS property

This property gives you access to the GridFS object associated with this database. See the description of the MongoGridFS class below. The MongoGridFS class has methods like Upload, Download and Find which allow you to interact with the GridFS file system.

### GetCollection method

From an instance of MongoDatabase you can get instances of objects representing the collections on that database using the GetCollection method. To get a collection object you would write:

```
MongoCollection<BsonDocument> books =  
    database.GetCollection("books");
```

or

```
MongoCollection<BsonDocument> books = database["books"];
```

The two forms are completely equivalent.

MongoDatabase maintains a table of collections that have been used so far, and each time you ask for the same collection you get the same instance back, so you don't need to worry about unneeded duplicate instances coming into existence.

### GetCollection<TDefaultDocument> method

One of the features of collections in MongoDB is that they are schema-free. However, it is also not unusual for a collection to hold primarily one type of document. We call this the default document type. Often the default document type will be BsonDocument, but you can specify a different default document type for a collection by using the GetCollection<TDefaultDocument> method. For example:

```
MongoCollection<Employee> employees =  
    Database.GetCollection<Employee>("employees");
```

There is no indexer equivalent because C# does not allow type parameters on indexers.

Even though a collection has a default document type you can always query for documents of any other type by using the Find<TQuery, TDocument> method instead of Find<TQuery> (Find<TQuery> is just a convenient shortcut for Find<TQuery, TDefaultDocument>).

The initial version of the C# Driver only supports working with documents of type `BsonDocument` (because BSON serialization/deserialization support is not yet implemented for other types).

### DropCollection method

This method can be used to drop a collection from a database. `MongoCollection` also has a method called `RemoveAll` that drops the data but leaves the collection (and any indexes) in place. To use this method write:

```
BsonDocument result = database.DropCollection("books");
```

`DropCollection` is actually a wrapper for a database command. All methods that wrap a command return the command result, which in some cases may have information of interest to you.

### RequestStart/RequestDone methods

It often happens that a thread does a series of database operations that are related. More importantly, a thread sometimes does a read that is dependent on earlier writes. If the read happens to be done on a different connection than the write sometimes the results will not yet be available. A thread can indicate that it is doing a series of related operations by using `RequestStart` and that it is done with the series of related operations by calling `RequestDone`. For example:

```
database.RequestStart();  
// do a series of operations on database  
database.RequestDone();
```

There is actually a slight problem with this example: if an exception is thrown while doing the operations on the database `RequestDone` will never be called. To prevent that you could put the call to `RequestDone` in a finally block, or even easier, take advantage of the fact that `RequestStart` returns an instance of a helper object that implements `IDisposable`, and use the C# using statement to guarantee that `RequestDone` is called automatically. In this case, you write:

```
using (database.RequestStart()) {  
    // do a series of operations on database  
}
```

Note that in this case you do NOT call `RequestDone` yourself, that happens automatically when you leave the scope of the using statement.

`RequestStart` increments a counter and `RequestDone` decrements it, so you can nest calls to `RequestStart` and `RequestDone` and the request isn't actually over until the counter reaches zero again. This helps when you are implementing methods that need to call `RequestStart/RequestDone` but are in turn called by code that has itself called `RequestStart`. An example of this is the `Upload` method in `MongoGridFS`, which calls `RequestStart/RequestDone`. Because of the nesting behavior of `RequestStart` it is perfectly fine to call `Upload` from code that has itself called `RequestStart`.

## MongoCollection<TDefaultDocument> class

Instances of this class represent a collection in a MongoDB database. You get an instance of a `MongoCollection` by calling the `GetCollection` or `GetCollection<TDefaultDocument>` methods of `MongoDatabase`. The type parameter `TDefaultDocument` is the default document type for this collection (which will be `BsonDocument` if you don't specify otherwise).

Instances of this class are thread safe.

## Database property

This property allows you to navigate back from a `MongoCollection` object to the `MongoDatabase` object that it belongs to.

## SafeMode property

This property represents the default `SafeMode` for this collection. It will be the default `SafeMode` for operations performed on this collection, but many methods provide a way to override the `SafeMode` for just that one operation. The `SafeMode` class is described further on.

## Insert<TDocument> method

Before we can retrieve any information from a collection we need to put it there. The `Insert` method is one way to insert documents into a collection (but see also the `Save` method and the `upsert` versions of the `Update` method). For example:

```
BsonDocument book = new BsonDocument {
    { "author", "Ernest Hemingway" },
    { "title", "For Whom the Bell Tolls" }
};
books.Insert(book);
```

The insert statement could also have been written as:

```
books.Insert<BsonDocument>(book);
```

but this is not necessary as the compiler can infer the type `TDocument` from the argument. Note that `TDocument` must be a type that can be serialized to a BSON document. The initial version of the C# Driver only knows how to serialize instances of the `BsonDocument` class.

Use the `InsertBatch` method when you want to insert more than one document at once. For example:

```
BsonDocument[] batch = {
    new BsonDocument {
        { "author", "Kurt Vonnegut" },
        { "title", "Cat's Cradle" }
    },
    new BsonDocument {
        { "author", "Kurt Vonnegut" },
```



```
        { "title", "Slaughterhouse-Five" }  
    }  
};  
books.InsertBatch(batch);
```

The advantage of inserting multiple documents at once is that it minimizes the number of network transmissions. All the documents being inserted are transmitted to the server in one message (unless the total size of the serialized documents exceeds 16MB, in which case they will be transmitted in multiple messages each holding as many documents as possible).

### Find<TQuery> method

The most common operation you will perform against a collection is to query it. This is done with the several variations of the Find method. In this tutorial we only show the simplest forms, but there are other forms that let you use a JavaScript where clause instead of a query object and/or specify the fields you want returned.

Here's some sample code to read back the documents we just inserted into the books collection:

```
BsonDocument query = new BsonDocument {  
    { "author", "Kurt Vonnegut" }  
};  
MongoCursor<BsonDocument> cursor = books.Find(query);  
foreach (BsonDocument book in cursor) {  
    Console.WriteLine(  
        "{0} by {1}",  
        book["title"].AsString,  
        book["author"].AsString  
    );  
}
```

If the document being returned is not of the default document type, then use a version of Find that lets you specify the document type:

```
MongoCursor<Employee> cursor =  
    employees.Find<BsonDocument, Employee>(query);
```

There are many forms of query objects that are supported, but they are not specific to the C# Driver, and we don't have space to describe them here. An empty query object retrieves all documents, and the C# Driver allows null as equivalent to an empty query object.

There is much more to say about MongoCursor, but there is a whole section below about using cursors so look there for more details.

## FindAll method

This method is just a shortcut for `FindAll<TDefaultDocument>` and retrieves all documents from a collection. For example:

```

MongoCursor<BsonDocument> cursor = books.FindAll();
foreach (BsonDocument book in cursor) {
    Console.WriteLine(
        "{0} by {1}",
        book["title"].AsString,
        book["author"].AsString
    );
}

```

To query for a document that is not of the default document type use `FindAll<TDocument>`.

## FindOne<TQuery> method

If you know there is only one matching document or if you only want the first matching document you can use the `FindOne` method. For example:

```

BsonDocument query = new BsonDocument {
    { "author", "Kurt Vonnegut" }
};
BsonDocument book = books.FindOne(query);

```

`FindOne` returns either the first matching document it finds or C# null if there is none. If there is more than one matching document it is not specified which one will be returned.

`FindOne<TQuery>` is actually a shortcut for:

```
FindOne<TQuery, TDefaultDocument>(query);
```

which in turn is a shortcut for:

```
Find<TQuery, TDefaultDocument>(query).Limit(1).FirstOrDefault();
```

which is useful to know if you think there may be multiple matching documents and want to control which one is returned. See the example for the `FirstOrDefault` method of `MongoCursor` below.

To query for a document that is not of the default document type use `FindOne<TQuery, TDocument>`.

## Save<TDocument> method

The `Save` method is a combination of `Insert` and `Update`. It examines the document provided to it and if the document is lacking an `"_id"` element it assumes it is a new document and calls `Insert` on it. If the document has an `"_id"` element it assumes it is probably an existing document and calls `Update` on it but

sets the Upsert flag just in case so the document will be inserted if it doesn't already exist. For example, you could correct an error in the title of a book using:

```
BsonDocument query = new BsonDocument {
    { "author", "Kurt Vonnegut" }
    { "title", "Cats Craddle" }
};
BsonDocument book = books.FindOne(query);
if (book != null) {
    book["title"] = "Cat's Cradle";
    books.Save(book);
}
```

Note that we check the return value of FindOne to verify that it actually found a matching document.

### Update<TQuery, TUpdate> method

The Update method is used to update existing documents (but see the Upsert flag below). The code sample shown in the Save method could also have been written as:

```
BsonDocument query = new BsonDocument {
    { "author", "Kurt Vonnegut" }
    { "title", "Cats Craddle" }
};
BsonDocument update = new BsonDocument {
    { "$set", new BsonDocument("title", "Cat's Cradle") }
};
BsonDocument updatedBoook = books.Update(query, update);
```

The Update method supports many different kinds of update documents, but again, we don't have space in this tutorial to describe them. The above example used the "\$set" update modifier to update the value of one element of the matching document.

There is also an overload of Update that allows you to specify one or more UpdateFlags. The values are Upsert and Multi. Normally an Update only affects one document, but using UpdateFlags.Multi you can request that all matching documents be updated. Also, by specifying UpdateFlags.Upsert you can request that the update document be inserted if a matching document can't be found (note that the Upsert feature requires that the update document be a complete document and not just an update operation as in the sample above).

### Count method

Sometimes you just need to know how many documents a collection has. There are two versions of the Count method, the first counts all documents, and the second counts matching documents. For example:

```
BsonDocument query = new BsonDocument {
```

```
        { "author", "Ernest Hemingway" }  
    };  
    int total = books.Count();  
    int booksByHemingway = books.Count(query);
```

### Remove method

This method can be used to remove documents from a collection. For example, to remove all of Hemingway's books from the collection, we write:

```
BsonDocument query = new BsonDocument {  
    { "author", "Ernest Hemingway" }  
};  
books.Remove(query);
```

If we only wanted to remove one book by Hemingway we would write:

```
BsonDocument query = new BsonDocument {  
    { "author", "Ernest Hemingway" }  
};  
books.Remove(query, RemoveFlags.Single);
```

although we wouldn't have control over which book exactly was removed; but it would be one by Hemingway.

### RemoveAll method

If you want to remove all the documents in a collection, use:

```
books.RemoveAll();
```

Use with caution! RemoveAll is just a shortcut for Remove(null). See also the DropCollection method in MongoDB, although this method differs from DropCollection in that only the data is removed, not the collection itself or any indexes.

### CreateIndex method

This method is used to create an index on a collection. In its simplest form you simply provide a list of the element names to be indexed. There are more complex forms that use a parameter to specify the details of the index to be created, but they are beyond the scope of this tutorial.

To create an index for the books collection on author, we would write:

```
books.CreateIndex("author");
```

Note that MongoDB is forgiving if you try to create an index that already exists, but to reduce the number of times the server is asked to create the same index use the EnsureIndex method instead.

## EnsureIndex method

You will normally call this method instead of `CreateIndex`. The driver keeps track of which indexes it has created and only calls `CreateIndex` once for each index (the driver doesn't know whether the index actually exists on the server, so it must call `CreateIndex` at least once to make sure the index exists). Calling `EnsureIndex` is just like calling `CreateIndex`:

```
books.EnsureIndex("author");
```

`EnsureIndex` also can't tell whether the index has been deleted on the server since it was created. You can make `EnsureIndex` forget about any indexes it knows about by calling `ResetIndexCache`. The next time `EnsureIndex` is called it will call `CreateIndex` once again (after that the index will be in the index cache again and `CreateIndex` will not be called anymore for that index).

## Distinct method

This method is used to find all the distinct values for a given key. For example, to find all the authors in the books collection you would write:

```
IEnumerable<BsonValue> authors = books.Distinct("author");
foreach (string author in authors) {
    Console.WriteLine(author.AsString);
}
```

There is also an overload of the `Distinct` method that takes a query, so you can find the distinct values for a given key of just those documents that match the query.

## FindAndModify method

Text missing.

## Group method

Text missing.

## MapReduce method

Text missing.

## MongoCursor<TQuery, TDocument> class

The `Find` method (and its variations) don't return the actual results of the find. Instead they return a cursor that can be enumerated to retrieve the results of the find. The query isn't actually sent to the server until we first attempt to enumerate the cursor, which means not only that the `Find` methods don't communicate with the server (they just return a cursor) but that we can control the results by altering the cursor in useful ways before enumerating it.

In C# the most convenient way to enumerate a cursor is to use the `foreach` statement. However, if necessary you can call a cursor's `GetEnumerator` method and work with the enumerator directly. You

can enumerate a cursor as many times as you wish. Each time you call GetEnumerator (usually indirectly through the foreach statement) a new query is sent to the server.

Once you enumerate a cursor it becomes frozen and no further changes can be made to it.

Instances of this class are NOT thread safe. However, once the cursor is frozen it becomes thread safe for enumeration (keeping in mind that each time GetEnumerator is called a separate query will be sent to the server).

### Collection property

You can use this property to navigate back from a cursor to the collection it is querying.

### Methods that modify the cursor before execution

Methods in this category are used to modify the cursor in some way before it is enumerated to control the results returned by the cursor. Note that all these methods return the cursor itself allowing them to be chained together (a “fluent interface”).

### Skip method

This method controls how many documents the server should skip over before returning results. It is often useful in pagination, although be aware that large values for Skip can become very inefficient. For example, suppose we had many books by Isaac Asimov and wanted to skip the first 5:

```
BsonDocument query = new BsonDocument {
    { "author", "Isaac Asimov" }
};
MongoCursor<BsonDocument> cursor = books.Find(query);
foreach (BsonDocument book in cursor.Skip(5)) {
    Console.WriteLine(book["title"].AsString);
}
```

### Limit method

The Limit method lets us control how many documents are returned by the cursor. This example is similar to the previous one, and tells the cursor to return at most 10 matching documents:

```
BsonDocument query = new BsonDocument {
    { "author", "Isaac Asimov" }
};
MongoCursor<BsonDocument> cursor = books.Find(query);
foreach (BsonDocument book in cursor.Limit(10)) {
    Console.WriteLine(book["title"].AsString);
}
```

The limit is enforced client side, and it is possible that the server might return more documents than needed, in which case the extra ones will be ignored. The enumerator sets some values in the query message it sends to the server to tell it how many documents to send back, but these are sometimes

only treated as hints by the server. For small limits you can use a negative value to let the server know to return all the results in a single message (the actual limit enforced client side will be the absolute value of the negative number).

### Sort method

Often we wish the results to be returned in a particular order. The Sort method allows us to tell the server what order to return the results in. In its simplest form we simply provide the names of the keys we want the results sorted on, as in:

```
MongoCursor<BsonDocument> cursor = books.FindAll();
foreach (BsonDocument book in cursor.Sort("author", "title")) {
    Console.WriteLine(
        "{0}: {1}",
        book["author"].AsString,
        book["title"].AsString
    );
}
```

Other overloads of the sort method let you control the sort order of each key individually.

### Other cursor modification methods

There are more cursor modification methods available that won't be described here, but their names alone suggest their use: AddOption, Batch, Fields, Hint, and Snapshot.

### Methods that trigger enumeration

The following methods all trigger enumeration of the cursor. This means that all modifications to the cursor using the above methods must be done first, before the cursor is enumerated and becomes frozen.

### Count method

If you don't need the individual results but just want to know the count of matching documents you can use this method. This method is equivalent to calling Count on the collection. Note that this method ignores any Skip or Limit options that have been set on the cursor (but see the Size method).

### Size method

This method is similar to the Count method, but honors any Skip or Limit options that have been set on the cursor.

### GetEnumerator method (or foreach statement)

This method returns an `IEnumerator<TDocument>` that can be used to enumerate the results. Usually you won't call this method yourself, but will instead use the C# foreach statement to enumerate the results. Note that the actual communication with the server doesn't occur until the first time the MoveNext method is called on the enumerator, but the cursor does become frozen as soon as GetEnumerator is called.

## FirstOrDefault method

Sometimes you only want or expect one result, in which case you can use this method, which is more convenient than enumerating the cursor. In simple cases you might prefer to use the FindOne method in MongoClient, but the FirstOrDefault method on the cursor lets you provide interesting options on the cursor before retrieving the one document. For example, suppose you wanted to find the first book Hemingway published:

```
BsonDocument query = new BsonDocument {
    { "author", "Ernest Hemingway" }
};
MongoCursor<BsonDocument> cursor = books.Find(query);
cursor.Sort("publicationDate");
BsonDocument firstBook = cursor.FirstOrDefault();
```

Because cursor modification methods support a fluent interface, you could also chain the calls together as in:

```
BsonDocument query = new BsonDocument {
    { "author", "Ernest Hemingway" }
};
BsonDocument firstBook = books.Find(query)
    .Sort("publicationDate")
    .FirstOrDefault();
```

FirstOrDefault is actually an extension method for IEnumerable<T> which can be used with a MongoCursor because MongoCursor<TQuery, TDocument> implements IEnumerable<TDocument>. Because of this you can use any IEnumerable<T> extension method with a MongoCursor.

## ToArray method

This IEnumerable<T> extension method enumerates the cursor for you and returns an array of T. For example:

```
BsonDocument query = new BsonDocument {
    { "author", "Ernest Hemingway" }
};
BsonDocument[] books = books.Find(query).ToArray();
```

If there are no matching documents the return value will be an empty array, not C# null.

## ToList method

This IEnumerable<T> extension method enumerates the cursor for you and returns a List<T>. For example:

```
BsonDocument query = new BsonDocument {
    { "author", "Ernest Hemingway" }
```



```
};  
List<BsonDocument> books = books.Find(query).ToList();
```

If there are no matching documents the return value will be an empty List, not C# null.

### **IEnumerable<T> extension methods that trigger enumeration**

Because `MongoCursor<TQuery, TDocument>` implements `IEnumerable<TDocument>` all of the extension methods defined for `IEnumerable<T>` can be used with a `MongoCursor`, and most of them trigger enumeration of the cursor. Following is a list of some of the more common methods:

- `First`
- `FirstOrDefault`
- `Last`
- `LastOrDefault`
- `Single`
- `SingleOrDefault`
- `ToArray`
- `Tolist`

Keep in mind that all extension methods that “look” at the sequence will of course enumerate the cursor. Another point to make is that some of the extension methods for `IEnumerable<T>` have the same name as methods in `MongoCursor<TQuery, TDocument>`, and in that case the methods defined in `MongoCursor<TQuery, TDocument>` take precedence (unless you cast your cursor to `IEnumerable<TDocument>` first).

### **Consuming a cursor safely**

When enumeration is triggered on a cursor a connection is acquired from the connection pool and is not released back to the connection pool until it is no longer needed. It is important to make sure that the connection gets returned to the connection pool by ensuring that the enumerator gets a chance to release the connection. The implementation of a `MongoCursor` enumerator always releases the connection as soon as it can (possibly even before enumeration of the results is complete) to minimize the chances of the connection being lost. If the enumerator is not properly disposed your program will not crash, but the connection will not be returned to the connection pool and will be lost (and will eventually be closed when it is garbage collected) which will result in your program opening more connections than necessary.

If a `MongoCursor` enumerator is consumed all the way to the end you can be sure the connection was released back to the connection pool (in fact, it will have been returned as soon as a reply to the Query or GetMore message indicated there would be no additional results).

The key to making sure that the connection is returned to the connection pool in all circumstances is to guarantee that the `Dispose` method of the enumerator returned by `GetEnumerator` is called. This is only really a concern if you are consuming the enumerator manually. All of the extension methods for

`IEnumerable<T>` call `Dispose` on the enumerator. The C# `foreach` statement is also guaranteed to call `Dispose` on the enumerator, so if you stick to `IEnumerable<T>` extension methods and the `foreach` statement you will be fine.

### Explain method

This method is different from any of the other cursor methods because it doesn't actually return the results of the query. Instead, the server returns information about how the query would have been executed. This information can be useful when trying to figure out why a query is not performing well. To use this method you would write:

```
BsonDocument query = new BsonDocument {  
    { "author", "Ernest Hemingway" }  
};  
MongoCursor<BsonDocument> cursor = books.Find(query);  
BsonDocument explanation = cursor.Explain();
```

### MongoGridFS class

This class is the C# Driver implementation of the GridFS specification for storing files in a MongoDB database. You normally get an instance of this class by using the `GridFS` property of `MongoDatabase`.

Instances of this class are thread safe. However, if you plan to modify the `Settings` for this instance, you should do so before the multiple threads start accessing an instance.

### Database property

This property allows you to navigate back from the instance of `MongoGridFS` to the database it is associated with.

### Settings property

This property gives you access to the `Settings` for this instance. You can change the `Root` name of the GridFS collection (default `"fs"`) and the `DefaultChunkSize` for new files (default 256KB). Note that the default chunk size only applies to new files. Any files previously created with a different chunk size will continue to be processed using the existing chunk size. It is not possible to change the chunk size of a file once it has been created.

### Upload method

This method uploads a file from the client computer to the database. It always creates a new file in the GridFS file system, so if you upload the same file more than once there will be multiple versions of the file in the database. To use this method write:

```
MongoGridFS gridFS = database.GridFS;  
MongoGridFSFileInfo fileInfo = gridFS.Upload("volcano.jpg");
```

The return value of Upload is information about the file that was just uploaded. You can ignore the return value if you wish. Upload will throw an exception if it fails.

There are other overloads of Upload that let you specify different local and remote filenames or to upload directly from a Stream instead of from a file.

### Download method

This method downloads a file from the database to the client computer. If the file already exists on the client computer it will be overwritten. To use this method write:

```
MongoGridFS gridFS = database.GridFS;  
gridFS.Download("volcano.jpg");
```

As mentioned before, there may be multiple versions of the same filename in the database. Download by default downloads the most recent version (as defined by the "uploadDate" element of the file metadata). You can download a different version by providing an integer version parameter. The values for a version number can be:

1	The first version
2	The second version
n	The nth version
-1	The newest version
-2	The second newest version
-n	The nth newest version
0	Any version (not very useful unless there's only one)

So for example, to retrieve the second newest version of our file you would write:

```
MongoGridFS gridFS = database.GridFS;  
gridFS.Download("volcano.jpg", -2); // second newest version
```

Other overloads of the Download method let you specify different local and remote filenames or to download directly to a Stream instead of a file.

### Find method

The Find method queries the GridFS file system for information about a file. If more than one file matches, then information for all of them is returned (even when providing a specific filename there may be multiple matches if the same filename has been uploaded more than once). For example:

```
MongoGridFS gridFS = database.GridFS;  
IEnumerable<MongoGridFSFileInfo> files =  
    gridFS.Find("volcano.jpg");
```

If there are no matches the list will be empty (not C# null).

### FindOne method

If you expect there to only be one matching filename or are willing to specify a version number then it is more convenient to use the FindOne method. For example:

```
MongoGridFS gridFS = database.GridFS;
MongoGridFSFileInfo file = gridFS.FindOne("volcano.jpg");
```

Or to get information about the second version of the file:

```
MongoGridFS gridFS = database.GridFS;
MongoGridFSFileInfo file = gridFS.FindOne("volcano.jpg", 2);
```

In both cases, if there is no matching file C# null is returned.

### Exists method

If you just want to know if the file exists but don't need any of the metadata about the file you can use this method. For example:

```
MongoGridFS gridFS = database.GridFS;
if (gridFS.Exists("volcano.jpg")) {
    // now you know the file exists
}
```

### Delete method

This method deletes matching files from the GridFS file system. If more than one file matches, they will all be deleted (see the sample below for how to delete just one version of a file). For example:

```
MongoGridFS gridFS = database.GridFS;
gridFS.Delete("volcano.jpg");
```

If you only want to delete the second newest version write something like:

```
MongoGridFS gridFS = database.GridFS;
MongoGridFSFileInfo fileInfo = gridFS.FindOne("volcano.jpg", -2);
if (fileInfo != null) {
    gridFS.DeleteById(fileInfo.Id); // delete by _id
}
```

### MongoGridFSFileInfo class

This class represents information about a file stored in the GridFS file system. This class is designed to be as similar as possible to .NET's FileInfo class.

### ChunkSize property

The chunk size used when this file was created.

### GridFS property

You can use this property to navigate from a `MongoGridFSFileInro` instance back to the `MongoGridFS` object it belongs to.

### Id property

This property has the value of the “\_id” element for this file. This value is assigned when the file is first uploaded and is guaranteed to be unique. For files uploaded using the C# Driver this will be an `ObjectId`, although it is possible you will occasionally encounter files uploaded with different drivers where the Id might be of a different type.

### Length property

The total length of the file in bytes.

### MD5 property

The MD5 hash of the file as computed at the server when it was uploaded.

### Name property

The remote filename of the file in the GridFS file system.

### UploadDate property

The `DateTime` when this file was uploaded. This `DateTime` is always in UTC.

### Methods to be implemented soon

This class also will implement methods that return a Stream-like object (a subclass of `Stream`) that allow you to read and write to GridFS files the same way you do local files (a possible alternative to using the Upload/Download methods, although probably not as fast). These methods all return an instance of `MongoGridFSStream`, a subclass of `Stream` that supports Read, Write and Seek. The methods that return a `MongoGridFSStream` (or a `StreamWriter` wrapped around it) are: `AppendText`, `Create`, `CreateText`, `Open`, `OpenRead`, `OpenText`, and `OpenWrite`.

### SafeMode class

Several times earlier in this tutorial we have mentioned `SafeMode`. There are various levels of `SafeMode`, and this class is used to represent those levels. `SafeMode` applies only to operations that don't already return a value (so it doesn't apply to queries or commands). It applies to the following methods of `MongoCollection`: `Insert`, `Remove`, `Save` and `Update`. It also applies to a `MongoGridFS` object as a whole, so all GridFS operations are performed at the same `SafeMode` level.

The gist of `SafeMode` is that after an `Insert`, `Remove`, `Save` or `Update` message is sent to the server it is followed by a `GetLastError` command so that the C# Driver can verify that the operation succeeded. In addition, when using replica sets there are `SafeModes` that let us specify how many replications we

want to wait to have completed before `GetLastError` returns. The standard way to access `SafeMode` values is to use static properties and methods of the class, as in:

```
SafeMode.False  
SafeMode.True  
SafeMode.WaitForReplications(n);
```

The value of “n” includes the primary, so to wait for at least one slave to have completed the replication we would use a value of “2”.

We also mentioned that `SafeMode` is inherited from server to database to collection and finally to operation. This inheritance happens at the moment the corresponding object is created but can be modified independently thereafter. So for example, if we want `SafeMode` to be on in general (the default is off for performance reasons) but we have one collection that we want it to be turned off for we can write:

```
MongoServer server = MongoServer.Create(connectionString);  
server.SafeMode = SafeMode.True; // default to SafeMode.True  
MongoDatabase test = server["test"]; // inherits SafeMode.True  
MongoCollection log = test["log"]; // inherits SafeMode.True  
log.SafeMode = SafeMode.False; // turn off SafeMode for log  
BsonDocument importantMessage = new BsonDocument {  
    // contents of important message  
};  
log.Insert(importantMessage, SafeMode.True); // override SafeMode
```

The last line of code illustrates that `SafeMode` can be overridden at the individual operation level.

## Part 2: The BSON Library

The remainder of this tutorial is about the BSON Library. This library implements serialization and deserialization of data in the BSON format (we won't cover that) and also provides an in-memory representation of a BSON document (which we will cover). The important classes are `BsonType`, `BsonValue` (and its subclasses), `BsonElement`, `BsonDocument` and `BsonArray`.

Unlike Part 1, which had a top down structure, in Part 2 we will examine the BSON Library bottom up, starting with types and values and working up towards elements, documents and arrays.

### BsonType enumeration

Every value in BSON has a type which is one of the fixed set of types BSON supports. The following enumeration lists all of the data types BSON supports:

```
public enum BsonType {  
    Double = 0x01,
```

```

String = 0x02,
Document = 0x03,
Array = 0x04,
Binary = 0x05,
ObjectId = 0x07,
Boolean = 0x08,
DateTime = 0x09,
Null = 0x0a,
RegularExpression = 0x0b,
JavaScript = 0x0d,
Symbol = 0x0e,
JavaScriptWithScope = 0x0f,
Int32 = 0x10,
Timestamp = 0x11,
Int64 = 0x12,
MinKey = 0xff,
MaxKey = 0x7f
}

```

We will examine these types in more detail as we discuss BsonValues.

### BsonValue class

This is an abstract class that represents the concept of a typed BSON value. There is a subclass of BsonValue for every value in the BsonType enumeration. BsonValue defines a number of properties and methods that are common to all BSON types (or in some cases to convert from BsonValue to some its subclasses or to .NET types).

Because BsonValue is an abstract class you cannot construct instances of this class, you can only construct instances of its concrete subclasses. On the other hand, you will often see variables, parameters and return values of this type.

### Creating BsonValues

There are several ways to create instances of BsonValues (all described later):

1. Use the static Create method of BsonValue
2. Use the static Create method of one of the BsonValue subclasses
3. Take advantage of the implicit conversions from .NET types to BsonValue
4. Call the public constructor (if available) of a BsonValue subclass

All of the BsonValue subclasses have a static Create method. We encourage you to use the static Create methods so that subclasses can handle special cases as required (often returning an instance of a pre-created object for commonly used values to avoid creating unnecessary objects). In some cases there is no public constructor and you have to use the static Create method.

## BsonType property

This property can be used to determine the actual type of a `BsonValue`. You could of course use the actual class of the object to do the same thing, but it is often more convenient (and faster) to use this property. The following example shows several approaches:

```
BsonValue value;
if (value.BsonType == BsonType.Int32) {
    // we know value is an instance of BsonInt32
}
if (value is BsonInt32) {
    // another way to tell that value is a BsonInt32
}
if (value.IsInt32) {
    // the easiest way to tell that value is a BsonInt32
}
```

See also the `Is[Type]` properties for the most convenient way to test if a value is of a particular type.

## As[Type] properties

`BsonValue` defines a number of properties that can be used to downcast a `BsonValue` to a concrete type. The result of the `As[Type]` properties will either be a subclass of `BsonValue` or when appropriate a .NET primitive value. The full list of `As[Type]` properties is:

```
AsBoolean (bool)
AsBsonArray
AsBsonBinaryData
AsBsonDocument
AsBsonJavaScript // also works if BsonType == JavaScriptWithScope
AsBsonJavaScriptWithScope
AsBsonMaxKey
AsBsonMinKey
AsBsonNull
AsBsonRegularExpression
AsBsonSymbol
AsBsonTimestamp
AsByteArray
AsDateTime (DateTime)
AsDouble (double)
AsGuid (Guid)
AsInt32 (int)
AsInt64 (long)
AsObjectId
AsString (string)
```



The properties that return .NET primitive types are the ones that have a .NET type in parenthesis after them in the above list.

All of these properties throw an `InvalidCastException` if the actual type of the `BsonValue` is not the required one. If you're not sure of the type of a `BsonValue`, you can test it beforehand using the `Is[Type]` properties described below.

`AsBsonJavaScript` is a special case because it succeeds even if the actual type of the value is `BsonJavaScriptWithScope` because `BsonJavaScriptWithScope` is a subclass of `BsonJavaScript`.

See also the more limited set of `To[Type]` properties which can convert between some types.

### Is[Type] properties

`BsonValue` defines a number of properties that can be used to query if an instance of `BsonValue` is of a particular type. These are conveniences and let you write more compact code. For example:

```
if (value.IsString) {  
    // we know value is an instance of BsonString  
    // we also know value.AsString will not fail  
}
```

The full list of `Is[Type]` properties is:

```
IsBoolean  
IsBsonArray  
IsBsonBinaryData  
IsBsonDocument  
IsBsonJavaScript  
IsBsonJavaScriptWithCode  
IsBsonMaxKey  
IsBsonMinKey  
IsBsonNull  
IsBsonRegularExpression  
IsBsonSymbol  
IsBsonTimestamp  
IsDateTime  
IsDouble  
IsGuid  
IsInt32  
IsInt64  
IsObjectId  
IsString
```

### To[Type] methods

`BsonValue` provides a few methods that can do limited conversion between types. These conversions that for the most part are safe and aren't expected to fail. The `To[Type]` methods are:

```
ToBoolean  
ToDouble  
ToInt32  
ToInt64
```

The `ToBoolean` method never fails. It uses the JavaScript definition of truthiness, where `false`, `0`, `0.0`, `Nan`, `BsonNull` and `""` are false, and everything else is true (including the string `"false"`!).

The `ToBoolean` method is particularly useful when the documents you are processing have inconsistent ways of recording true/false values. For example:

```
if (employee["ismanager"].ToBoolean()) {  
    // we know the employee is a manager  
    // even if value is sometimes true and sometimes 1  
}
```

will work even if the documents use a mixture of types to represent true/false values. In contrast, the `AsBoolean` property will work **ONLY** if the values are recorded using strictly `BsonBoolean` values.

The `ToDouble`, `ToInt32` and `ToInt64` methods won't fail if converting between the numeric types (although the value may be truncated!). If converting from a string they will succeed if the string can be parsed as a number of the target type, otherwise they throw an exception.

### Create methods

While you cannot create instances of `BsonValue`, the `BsonValue` class has a static method called `Create` that you can use to create concrete instances of `BsonValue`. The type of the resulting `BsonValue` is determined at runtime based on the actual type of the value supplied. An exception is thrown if the value provided cannot be mapped to a `BsonValue`. For example:

```
BsonValue value = BsonValue.Create(1); // creates a BsonInt32  
BsonValue value = BsonValue.Create(1.0); // creates a BsonDouble  
BsonValue value = BsonValue.Create("abc"); // a BsonString
```

If you want to choose the type of the `BsonValue`, use the static `Create` method of one of the subclasses of `BsonValue` instead:

```
BsonDouble value = BsonDouble.Create(1);  
BsonJavaScript value = BsonJavaScript.Create("this.a == 4");  
BsonSymbol value = BsonSymbol.Create("+");
```

In the first case a `BsonDouble` will be created and the int value `1` will be converted to a double value `1.0`. In the second example a `BsonJavaScript` will be created and the Code value will be `"this.a == 4"`. In the third case a `BsonSymbol` will be created and `"+"` will be looked up in the `BsonSymbolTable`. If the value supplied cannot be converted to the type required by the `BsonValue` subclass an exception is thrown.

## Implicit conversions from .NET types to BsonValue

The BsonValue class also defines a number of implicit conversions to BsonValue. This makes it extremely easy to create BsonValues. Here are some examples:

```
BsonValue value = 1; // converts 1 to a BsonInt32
BsonValue value = 1.0; // converts 1.0 to a BsonDouble
BsonValue value = "abc"; // converts "abc" to a BsonString
```

These implicit conversions are specially handy when calling functions that take a BsonValue as a parameter, allowing you to use primitive .NET values as arguments. An important advantage of these implicit conversions is that the mapping from the .NET type to the matching BsonType happens at compile time, so the creation of BsonValues is extremely fast. There is also very little overhead due to the fact that we are creating an instance of BsonValue, because if we weren't using BsonValue and were using object instead all the primitive .NET values would have to be boxed. In essence, BsonValue is just our way of boxing .NET types (but our way we get to tag the value with a BsonType and we get to define a lot of helpful properties and methods).

The following implicit conversions are defined:

.NET type	BsonType
bool	Boolean
byte[]	Binary (subtype Binary)
DateTime	DateTime
double	Double
Guid	Binary (subtype Uuid)
int	Int32
long	Int64
Regex	RegularExpression
string	String

The list of implicit conversions is rather short on purpose. We have only defined implicit conversions in the cases where there is an exact match between the .NET type and the corresponding BsonType and in which there can be no loss of information in the conversion.

## Explicit conversions from BsonValue to .NET types

As an alternative to the As[Type] properties the BsonValue class also defines a set of explicit conversions to some .NET primitive types. For example, the following two statements are equivalent:

```
int age = person["age"].AsInt32;
int age = (int) person["age"];
```

The set of explicit conversions provided is the exact opposite of the set of implicit conversions provided, so the table in the previous section can be read from right to left to find the explicit conversions that are available.

You can use whichever you prefer, but the As[Type] methods are often more readable. Consider:

```
string zip = person["address"].AsBsonDocument["zip"].AsString;  
string zip = (string) ((BsonDocument) person["address"])["zip"];
```

In the second form you better get all those parenthesis just right!

### Equals and GetHashCode methods (and operator == and !=)

The BsonValue classes all override Equals and GetHashCode so instances of BsonValue are safe to use as Dictionary keys.

### BsonValue subclasses

We mentioned earlier that there is a subclass of BsonValue for every BsonType. Here is the full list of BsonValue subclasses:

```
BsonArray  
BsonBinaryData  
BsonBoolean  
BsonDateTime  
BsonDocument  
BsonDouble  
BsonInt32  
BsonInt64  
BsonJavaScript  
BsonJavaScriptWithScope  
BsonMaxKey  
BsonMinKey  
BsonObjectId  
BsonRegularExpression  
BsonString  
BsonSymbol  
BsonTimestamp
```

We will discuss each of these subclasses briefly, leaving BsonDocument and BsonArray for the end because they are the most complicated.

### BsonBinaryData class

This class holds values of BSON type Binary. It has the following properties and methods (in addition to those inherited from BsonValue):

```
BsonBinaryData(byte[] bytes)  
BsonBinaryData(byte[] bytes, BsonBinarySubType subType)  
BsonBinaryData(Guid guid)
```

```
byte[] Bytes
BsonBinarySubType SubType
implicit conversion from byte[]
implicit conversion from Guid
static BsonBinaryData Create(byte[] bytes)
static BsonBinaryData Create(byte[] bytes,
    BsonBinarySubType subType)
static BsonBinaryData Create(Guid guid)
static BsonBinaryData Create(object value)
Guid ToGuid()
```

### BsonBoolean class

This class holds values of BSON type Boolean. It has the following properties and methods (in addition to those inherited from BsonValue):

```
BsonBoolean(bool value)
static BsonBoolean False
static BsonBoolean True
object RawValue
bool Value
implicit conversion from bool
static BsonBoolean Create(bool value)
static BsonBoolean Create(object value)
```

This class has a few pre-created instances (False, True). You can use them yourself, and the Create method returns them when appropriate.

### BsonDateTime class

This class holds values of BSON type DateTime. It has the following properties and methods (in addition to those inherited from BsonValue):

```
BsonDateTime(DateTime value)
object RawValue
DateTime Value
implicit conversion from DateTime
static BsonDateTime Create(DateTime value)
static BsonDateTime Create(object value)
```

Note that BSON DateTime values are always in UTC.

## BsonDouble class

This class holds values of BSON type Double. It has the following properties and methods (in addition to those inherited from BsonValue):

```
BsonDouble(double value)
object RawValue
double Value
implicit conversion from double
static BsonDouble Create(double value)
static BsonDouble Create(object value)
```

## BsonInt32 class

This class holds values of BSON type Int32. It has the following properties and methods (in addition to those inherited from BsonValue):

```
BsonInt32(int value)
static BsonInt32 MinusOne
static BsonInt32 Zero
static BsonInt32 One
static BsonInt32 Two
static BsonInt32 Three
object RawValue
int Value
implicit conversion from int
static BsonInt32 Create(int value)
static BsonInt32 Create(object value)
```

This class has a few pre-created instances (MinusOne, Zero, One, Two, Three). The static Create method will return them when appropriate.

## BsonInt64 class

This class holds values of BSON type Int64. It has the following properties and methods (in addition to those inherited from BsonValue):

```
BsonInt64(long value)
object RawValue
long Value
implicit conversion from long
static BsonInt64 Create(long value)
static BsonInt64 Create(object value)
```

## BsonJavaScript class

This class holds values of BSON type JavaScript. It has the following properties and methods (in addition to those inherited from BsonValue):

```

BsonJavaScript(string code)
object RawValue
string Code
implicit conversion from string
static BsonBinaryData Create(object value)
static BsonJavaScript Create(string code)

```

## BsonJavaScriptWithScope class

This class holds values of BSON type JavaScriptScope. It has the following properties and methods (in addition to those inherited from BsonValue):

```

BsonJavaScriptWithScope(string code, BsonDocument scope)
string Code // inherited from BsonJavaScript
BsonDocument Scope
static BsonJavaScriptWithScope Create(object value)
static BsonJavaScriptWithScope Create(string code,
    BsonDocument scope)

```

Note that BsonJavaScriptWithScope is a subclass of BsonJavaScript.

## BsonMaxKey class

This class holds the singleton value of BSON type MaxKey. It has the following properties and methods (in addition to those inherited from BsonValue):

```

static BsonMaxKey Singleton // can use Bson.MaxKey instead

```

Note that this class is implemented a singleton and that therefore only a single instance of it exists. You can access this singleton value in your code in two ways:

```

BsonMaxKey mk = BsonMaxKey.Singleton;
BsonMaxKey mk = Bson.MaxKey; // more convenient

```

## BsonMinKey class

This class holds the singleton value of BSON type MinKey. It has the following properties and methods (in addition to those inherited from BsonValue):

```

static BsonMinKey Singleton // can use Bson.MinKey instead

```

Note that this class is implemented a singleton and that therefore only a single instance of it exists. You can access this singleton value in your code in two ways:

```
BsonMinKey mk = BsonMinKey.Singleton;  
BsonMinKey mk = Bson.MinKey; // more convenient
```

### BsonNull class

This class holds the singleton value of BSON type Null. It has the following properties and methods (in addition to those inherited from BsonValue):

```
static BsonNull Singleton // can use Bson.Null instead
```

Note that this class is implemented a singleton and that therefore only a single instance of it exists. You can access this singleton value in your code in two ways:

```
BsonNull n = BsonNull.Singleton;  
BsonNull n = Bson.Null; // more convenient
```

Note also that a BsonElement can never have a C# null as its value, instead the BsonNull singleton is used to represent BSON null values.

### BsonObjectId class

This class holds values of BSON type ObjectId. It has the following properties and methods (in addition to those inherited from BsonValue):

```
BsonObjectId(byte[] bytes) // must be exactly 12 bytes  
BsonObjectId(int timestamp, long machinePidIncrement)  
BsonObjectId(ObjectId value)  
BsonObjectId(string value)  
int Timestamp // first 4 bytes of ObjectId  
long MachinePidIncrement // last 8 bytes of ObjectId  
int Machine  
int Pid  
int Increment  
object RawValue  
ObjectId Value  
DateTime CreationTime // derived from Timestamp  
implicit conversion from ObjectId  
static BsonObjectId Create(byte[] bytes)  
static BsonObjectId Create(int timestamp,  
    long machinePidIncrement)  
static BsonObjectId Create(object value)
```



```
static ObjectId Create(ObjectId value)
static ObjectId Create(string value)
static ObjectId GenerateNewId()
static ObjectId Parse(string value)
static bool TryParse(string value, out ObjectId objectId)
byte[] ToByteArray()
```

See also the `ObjectId` struct in the next section. The difference between `BsonObjectId` and `ObjectId` is that `BsonObjectId` is a subclass of `BsonValue` and can be the Value of a `BsonElement`, whereas `ObjectId` is a struct and contains the actual physical representation of an `ObjectId` (so the Value property of `BsonObjectId` is of type `ObjectId`). Many of the properties of `ObjectId` are echoed in `BsonObjectId` and simply call through to the underlying `ObjectId` properties.

### ObjectId struct

This struct holds the physical representation of an `ObjectId`. It has the following properties and methods:

```
ObjectId(byte[] bytes) // must be exactly 12 bytes
ObjectId(int timestamp, long machinePidIncrement)
ObjectId(string value)
int Timestamp // first 4 bytes of ObjectId
long MachinePidIncrement // last 8 bytes of ObjectId
int Machine
int Pid
int Increment
DateTime CreationTime // derived from Timestamp
static ObjectId GenerateNewId()
static ObjectId Parse(string value)
static bool TryParse(string value, out ObjectId objectId)
byte[] ToByteArray()
```

### BsonRegularExpression class

This class holds values of BSON type `RegularExpression`. It has the following properties and methods (in addition to those inherited from `BsonValue`):

```
BsonRegularExpression(string pattern) // options == ""
BsonRegularExpression(string pattern, string options)
BsonRegularExpression(Regex regex)
string Pattern
string Options
implicit conversion from Regex
implicit conversion from string
static BsonRegularExpression Create(object value)
static BsonRegularExpression Create(string pattern)
```

```
static BsonRegularExpression Create(string pattern,
    string options)
static BsonRegularExpression Create(Regex regex)
Regex ToRegex()
```

### BsonString class

This class holds values of BSON type String. It has the following properties and methods (in addition to those inherited from BsonValue):

```
BsonString (string value)
static BsonString Empty
object RawValue
string Value
implicit conversion from string
static BsonString Create(object value)
static BsonString Create(string value)
```

This class has one pre-created instance (Empty). You can use it yourself, and the Create method returns it when appropriate.

### BsonSymbol class

This class holds values of BSON type Symbol. It has the following properties and methods (in addition to those inherited from BsonValue):

```
string Name
implicit conversion from string
static BsonSymbol Create(object value)
static BsonSymbol Create(string name)
```

This class does NOT have a public constructor because BsonSymbols are guaranteed to be unique, so therefore you have to use the static Create method instead (which looks up the BsonSymbol in the BsonSymbolTable and creates a new instance of BsonSymbol only if necessary).

### BsonTimestamp class

This class holds values of BSON type Timestamp. It has the following properties and methods (in addition to those inherited from BsonValue):

```
BsonTimestamp(int timestamp, int increment)
BsonTimestamp(long value)
object RawValue
long Value
int Timestamp // high order 32 bits of Value
```

```
int Increment // low order 32 bits of Value
implicit conversion from long
static BsonTimestamp Create(int timestamp, int increment)
static BsonTimestamp Create(long value)
static BsonTimestamp Create(object value)
```

### BsonElement class

This class represents instances of a BSON element: a name/value pair. It is what BsonDocuments contain (unlike BsonArrays, which contain BsonValues). This is a very simple class containing the following methods and properties:

```
BsonElement(string name, BsonValue value)
string Name
BsonValue Value
static BsonElement Create(string name, BsonValue value)
static BsonElement Create(bool condition, string name,
    BsonValue value)
```

You probably won't have to explicitly create a BsonElement very often; more likely they will be created for you automatically when you call one of the Add methods of BsonDocument.

The static Create methods return C# null if value is C# null (or if condition is false).

Note that BsonElement implements Equals and GetHashCode, so BsonElements can be used as keys in Dictionaries.

### BsonDocument class

This class is the workhorse of the C# Driver. You will use it a lot! Because it is used so much the interface is a little bit complicated, in order to provide for the many common use cases. There are basically three ways to create and populate a BsonDocument:

1. Create a new document and call Add and Set methods
2. Create a new document and use C# collection initializer syntax
3. Create a new document and use the fluent interface style supported by Add and Set

The first is straightforward and easy to use, but the second, while requiring you to be familiar with collection initializer syntax, is much more readable and is the recommended way to create BsonDocuments.

### BsonDocument constructor

You can use the BsonDocument constructor to create an empty BsonDocument or as an easy way to create a BsonDocument with a single element. All overloads of BsonDocument (except two) simply call the matching Add method with the arguments provided.

To create an empty `BsonDocument` write:

```
BsonDocument document = new BsonDocument();
```

To create a document and populate it with one element write:

```
BsonDocument query = new BsonDocument("author", "Hemmingway");
```

By default `BsonDocument` does not allow duplicate element names, and if you attempt to Add an element with an existing name an exception is thrown. If you want to allow duplicate element names you have to use the following constructor:

```
BsonDocument document = new BsonDocument(true); // allow dups
document.Add("hobby", "hiking");
document.Add("hobby", "cycling"); // only accessible via index
BsonValue h1 = document["hobby"]; // returns "hiking"
BsonValue h2 = document[0]; // also returns "hiking"
BsonValue h3 = document[1]; // returns "cycling"
```

There are also overloads of the `BsonDocument` constructor that take a:

- `BsonElement`
- `IDictionary<string, object>`
- `IEnumerable<BsonElement>`
- `params BsonElement[]`

See the corresponding `Add` method for a description of each one.

### **BsonDocument constructor with collection initializer syntax**

This is the preferred way to create and populate `BsonDocuments` (with the exception of single element documents which are more easily created by passing a name and value to the constructor). The way C# collection initializer syntax works is that it lets you provide a list of values (which can be tuples) that are passed to a matching `Add` method. The compiler simply inserts calls to the matching `Add` methods.

Many of the `BsonDocuments` created in sample code you've seen so far were created using this syntax. For example:

```
BsonDocument book = new BsonDocument {
    { "author", "Ernest Hemingway" },
    { "title", "For Whom the Bell Tolls" }
};
```

is converted by the compiler to the following:

```
BsonDocument book = new BsonDocument();
book.Add("author", "Ernest Hemingway");
```

```
book.Add("title", "For Whom the Bell Tolls");
```

The two are completely equivalent, but the first is more elegant and readable because it mirrors the structure of the BsonDocument being created and is less verbose.

A common mistake when using collection initializer syntax is to forget one of the sets of braces. The reason this doesn't work is apparent when we see that the compiler will convert:

```
BsonDocument wrong = new BsonDocument { "name", "value" };
```

to:

```
BsonDocument wrong = new BsonDocument();  
wrong.Add("name");  
wrong.Add("value");
```

which results in a compile time error because there is no Add method with a matching signature (fortunately).

### BsonDocument constructor with fluent interface Add methods

Another way to construct a BsonDocument is to use the fluent interface style Add methods to populate the document. For example:

```
BsonDocument book = new BsonDocument()  
    .Add("author", "Ernest Hemingway")  
    .Add("title", "For Whom the Bell Tolls");
```

We don't recommend this style because we feel C# collection initializer syntax is superior, but if you are porting code from another driver you may find this style useful.

### AllowDuplicateNames property

Determines whether this document is allowed to contain duplicate names or not. The BSON standard specifies that duplicate names are not allowed, but in some cases it might be useful (for example, when mapping an XML document to a BsonDocument).

### Count property

This property returns the number of elements that this document contains.

### Elements property

This property returns an IEnumerable<BsonElement> value that can be used to enumerate over the elements of the document. You can also enumerate over the elements in a BsonDocument directly (without using the Elements property) because BsonDocument implements IEnumerable<BsonElement>. See also the Names, Values and RawValues properties.

### Names property

This property returns an `IEnumerable<string>` value that can be used to enumerate over the names of the elements of the document.

### RawValues property

This property returns an `IEnumerable<object>` value that can be used to enumerate over the values of the elements of the document as raw objects (not all `BsonValues` have a raw value, only `Boolean`, `DateTime`, `Double`, `Int32`, `Int64`, `String` and `Timestamp` do). The sequence returned will contain `C# null` for any `BsonValues` of types that don't have a `RawValue`.

### Values property

This property returns an `IEnumerable<BsonValue>` value that can be used to enumerate over the values of the elements of the document as `BsonValues`.

### Add methods

There are many overloads of the `Add` method. Note that in all cases `C# null` values are ignored, so you don't have to put if statements around your calls to `Add` to check if your value is null (assuming you want it to be ignored). When an `Add` method is passed a `C# null` value it simply does nothing.

#### Add(BsonElement) method

This is the base `Add` method. All other `Add` methods end up calling this one. This method behaves slightly differently depending on whether duplicate element names are allowed or not. If duplicate names are not allowed and an element with the same name exists then an exception is thrown, otherwise it is added to the end of the collection and can only be accessed by index (accessing by name will return the existing element, not the new one).

#### Add(IEnumerable<BsonElement>) method

This method simply calls `Add(BsonElement)` for each element passed in.

#### Add(name, value) method

This method creates and adds a `BsonElement` to the document if the value supplied is not `C# null`. For example:

```
BsonDocument book = new BsonDocument();
book.Add("author", "Ernest Hemingway");
```

Since this `Add` method passes its arguments straight through to the `BsonElement` constructor you can refer to the discussion below about creating `BsonElements` and how the `.NET` value provided is mapped to a `BsonValue`.

It is important to remember that the `Add` method does nothing if the value supplied is a `C# null`. If you want a `BSON Null` to be stored for a missing value you have to supply the `BSON Null` value yourself. A convenient way to do that is using `C#'s` null coalescing operator:

```
BsonDocument book = new BsonDocument {  
    { "author", author ?? Bson.Null }  
};
```

### Add(condition, name, value) method

This method creates and adds a new BsonElement to the document if the condition is true and the value supplied is not C# null. This method exists to support conditionally adding elements to a document when using C# collection initializer syntax. For example:

```
BsonDocument book = new BsonDocument {  
    { "author", "Ernest Hemingway" },  
    { "title", "For Whom the Bell Tolls" },  
    { havePublicationDate, "publicationDate", publicationDate }  
};
```

This is only really necessary for struct values (because they can't be C# null). For example, suppose you only wanted to add the "subtitle" element if it was not C# null. We could simply write:

```
BsonDocument book = new BsonDocument {  
    { "author", "Ernest Hemingway" },  
    { "title", "For Whom the Bell Tolls" },  
    { "subtitle", subTitle }  
};
```

If the value of the subTitle is C# null the "subtitle" element will not be added to the document.

### Add(IEnumerable<BsonElement>) method

Each element will be added to this BsonDocument. Note that the elements are not cloned.

Since BsonDocument implements IEnumerable<BsonElement>, you can pass an instance of BsonDocument to this method. However, since the the elements are not cloned, the same BsonElements are in both BsonDocuments. This means that if you change the value of the BsonElement in one document it will change in the other as well. If you don't want this behavior then you must clone the elements before adding them to the document. For example:

```
BsonDocument order = new BsonDocument();  
BsonDocument items; // passed in from somewhere  
order.Add(items); // elements are now in both documents
```

or if you don't want to share the elements between the two documents use one of:

```
order.Add((BsonDocument) items.Clone());  
order.Add((BsonDocument) items.DeepClone());
```

depending on whether you need shallow or deep cloning.

The default behavior is to not clone because cloning is expensive and this way we default to the most efficient behavior and you get to decide when the cost of cloning is necessary.

### Add(IDictionary<string, object>) method

This method adds new elements to a BsonDocument based on the contents of a dictionary. Each key in the dictionary becomes the name of the new element, and each corresponding value become the value of the new element. As always, this method calls Add(BsonElement) with each new element, so duplicate names are handled as described in Add(BsonElement).

### Clear method

This method removes all elements from the document.

### Contains method

This method tests whether the document contains an element of the given name.

### ContainsValue method

This method tests whether the document contains an element with the given value.

### GetElement methods

Normally you will use GetValue, or even more commonly the corresponding indexers. In the rare cases where you might need to get the BsonElements themselves uses the two overloads of GetElement (one if by index and two if by name).

### GetValue methods and indexers

There are three forms of the GetValue method: the first takes a numeric index, the second takes the name of the element, and the third also takes a default value to be returned if there is no element of that name. In all cases the return value is of type BsonValue. There is also a matching indexer for each of the GetValue methods. For example:

```
BsonDocument book = books.FindOne();
string author = book.GetValue("author").AsString;
DateTime publicationDate =
    book.GetValue("publicationDate").AsDateTime;
int pages = book.GetValue("pages").AsInt32;
int pages = book.GetValue("pages", -1).AsInt32; // default -1
```

or using the indexers, which is the recommend way:

```
BsonDocument book = books.FindOne();
string author = book["author"].AsString;
DateTime publicationDate = book["publicationDate"].AsDateTime;
int pages = book["pages"].AsInt32;
int pages = book["pages", -1].AsInt32; // default -1
```



Since BsonValue also supports explicit casts to corresponding .NET types, this example could also have been written as:

```
BsonDocument book = books.FindOne();
string author = (string) book["author"];
DateTime publicationDate = (DateTime) book["publicationDate"];
int pages = (int) book["pages"];
int pages = (int) book["pages", -1]; // default -1
```

Which of these ways you use is a matter of personal choice. We recommend the second way (using indexers and As[Type] properties), as we feel it is the most readable (and doesn't require the extra parenthesis sometimes required by type casting (which can be specially confusing when they are nested (which they often are))). Compare the following two statements:

```
// person is a BsonDocument
BsonDocument firstBorn =
    person["children"].AsBsonArray[0].AsDocument;
BsonDocument firstBorn =
    (BsonDocument) ((BsonArray) person["children"])[0];
```

The first is somewhat easier to read from left to right. The same sequence of events is happening in both cases:

1. The "children" element of the person document is retrieved (a BsonValue)
2. The BsonValue is cast to a BsonArray
3. The first element of the children array is retrieved (a BsonValue)
4. The BsonValue is cast to a BsonDocument

The Get methods that have a name parameter throw an exception if an element with that name is not found and no default value was supplied. See the TryGetValue method if you don't want an exception.

### Merge method

This method allows you to merge elements from one BsonDocument into another. Each element from the source document is tested to see if the target document already has an element with that name. If the name already exists, the element is skipped, if not it is added to the target document.

### Remove method

This method is used to remove an element from a BsonDocument. If the document allows duplicate names then a call to Remove will remove all elements with that name.

### RemoveAt method

This method is used to remove an element from a BsonDocument using its index position.

## Set(index, value) method

While most of the time you will access elements by name, occasionally you will access them by index. When using this method the document must already have an element at that index or an exception will be thrown. There is an equivalent indexer. For example:

```
book.Set(0, "Ernest Hemingway"); // assume element 0 is author
book[0] = "Ernest Hemingway"; // shorter and better
```

## Set(name, value) method

If an element of this name exists, its value is replaced with the new value, otherwise a new element with this name and value are added. For example:

```
BsonDocument book = new BsonDocument();
book.Set("author", "Ernest Hemway"); // adds element
// some other work
// notice that author name is spelled wrong
book.Set("author", "Ernest Hemingway"); // replaces value
```

The previous example could be written using indexers as:

```
BsonDocument book = new BsonDocument();
book["author"] = "Ernest Hemway"; // adds element
// some other work
// notice that author name is spelled wrong
book["author"] = "Ernest Hemingway"; // replaces value
```

## ToBson method

This method will serialize the BsonDocument to the binary BSON format and return the serialized document as a byte array.

## ToJson method

This method will serialize the BsonDocument as a JSON formatted string. It is possible to have some control over the resulting JSON representation by providing some BsonJsonWriterSettings.

## ToString method

This method is overridden to call ToJson using default settings. It is primarily useful while debugging. If you are intentionally creating JSON call ToJson instead.

## TryGetElement and TryGetValue methods

If you're not sure the document contains an element with a given name you can either test first using the Contains method or use the TryGetElement or TryGetValue methods. For example:

```
BsonValue age;
if (person.TryGetValue("age", out age)) {
    // do something with age
}
```

```

    } else {
        // handle the missing age case
    }

```

## BsonArray class

This class is used to represent BSON arrays. Note that while a BSON array is represented externally as a BSON document (with a special naming convention for the elements), in the C# Driver the `BsonArray` class is NOT related to the `BsonDocument` class. That is because the fact that a `BsonArray` is represented externally as a `BsonDocument` is accidental, and the actual behavior of a `BsonArray` is very different from that of a `BsonDocument`.

## BsonArray constructors

You can use the no-argument constructor to create an empty array, and you can use C#'s collection initializer syntax to add items to the array at the same time:

```

BsonArray a1 = new BsonArray(); // empty
BsonArray a2 = new BsonArray { 1 }; // 1 element
BsonArray a3 = new BsonArray { 1, 2, 3 }; // 3 elements

```

There is also a constructor that takes an `IEnumerable<BsonValue>` parameter (it calls the matching `AddRange` method). For example:

```

IEnumerable<BsonValue> values; // from somewhere
BsonArray a = new BsonArray(values); // adds values to array

```

A number of additional overloads (like `IEnumerable<int>`) are provided to support common cases like:

```

int[] values = new int[] { 1, 2, 3 };
BsonArray a = new BsonArray(values);

```

This example would not compile without the `IEnumerable<int>` overload because `IEnumerable<int>` cannot be automatically converted to `IEnumerable<BsonValue>`. `IEnumerable<T>` overloads are provided for `bool`, `DateTime`, `double`, `int`, `long`, `object` and `string`.

## Indexer property

This class provides an indexer property to get or set elements in the array. The indexer return type is `BsonValue`, so you usually have to do some kind of conversion on it. For example:

```

array[0] = "Tom";
array[1] = 39;
string name = array[0].AsString;
int age = array[1].AsInt32;

```

## Count property

This property returns the number of values stored in the BsonArray

## Add method

There is only one Add method, and it adds a BsonValue to the array (but see also the AddRange methods). For example:

```
BsonValue value; // from somewhere
array.Add(value);
array.Add(1); // implicit conversion from int to BsonInt32
```

## AddRange methods

One overload of the AddRange method takes an IEnumerable<BsonValue> parameter and adds multiple items to the array. For example:

```
IEnumerable<BsonValue> values; // from somewhere
array.AddRange(values); // adds values to array
```

A number of additional overloads (like IEnumerable<int>) are provided to support common cases like:

```
int[] values = new int[] { 1, 2, 3 };
array.AddRange(values);
```

This example would not compile without the IEnumerable<int> overload because IEnumerable<int> cannot be automatically converted to IEnumerable<BsonValue>. IEnumerable<T> overloads are provided for bool, DateTime, double, int, long, object and string.

## Clear method

This method clears all values from the array.

## IndexOf methods

The following IndexOf methods can be used to find the index of a value in the array:

```
int IndexOf(BsonValue value)
int IndexOf(BsonValue value, int index)
int IndexOf(BsonValue value, int index, int count)
```

They all work just like the matching IndexOf methods of List<T>, and they return -1 if there is no matching item in the array.

## Insert method

This method is used to add a value at a particular location in the array. For example:

```
array.Insert(2, value);
```

This example assumes that the array contains at least 2 elements. If it contains more than two elements, the elements starting at index 2 will be moved over one position to make room for the new element.

### **Remove method**

The Remove method removes a particular value from the array. If the value occurs more than once in the array, only the first instance is removed. All remaining elements are shifted to the right so as to leave no gaps in the array and the resulting array will be shorter (unless no matching elements were found in which case this method has no effect).

### **RemoveAt method**

This method removes a value from a particular location in the array. Any elements following it are moved one position to the left and the array is now one element shorter.