CodeBase® 6.2 Reference Guide

Java Edition

The Database Engine For Database Management
Clipper Compatible
dBASE Compatible
FoxPro Compatible

Sequiter® Software Inc.

Printed in the United States of America.

Table of Contents

Introduction 5
Code4 Class 6
Data4 Class 21
Exception Classes 47
Field Manipulation Classes 50
Field4 Class 56
Field4boolean Class 58
Field4date Class 60
Field4double Class 64
Field4stringBuffer Class 66
Field4stringBufferUnicode Class 68
Field4deleteFlag Class 70
Field4info Class 72
Status4 Class 77
Tag4info Class 78
Appendix A: dBASE Expressions 82
Appendix B: CodeBase Limits 91
Index 93

Introduction

Congratulations on your purchase of CodeBase, one of the fastest, most flexible database development tools available for C, C++, Visual Basic and Delphi programmers and now available for Java. By using Java and CodeBase you can now build Web pages with full database capabilities. Futhermore, CodeBase's xBase file compatibility gives you the freedom to use CodeBase with many other commercially available tools.

This is the *CodeBase Java API Reference Guide*. Its purpose is to provide a way to quickly lookup information on CodeBase. The *Reference Guide* systematically documents all of the CodeBase classes and their methods in alphabetical order. It comprehensively covers all of the information you need to know about any aspect of the library. The *Reference Guide* also contains two appendices, one of which discusses dBASE expressions and the other CodeBase limits.

First time users of CodeBase should read the *Getting Started with Java* booklet and refer to the *CodeBase 6.0 Getting Started* book. It is also recommended that you read the introductions to each chapter of the *Reference Guide* for important information on how to use the CodeBase classes in Java applications. The *Reference Guide* also provides many example programs to illustrate how to use the CodeBase API in Java applications. The examples that appear in this manual are provided on-line and instructions on how to run the examples are given in depth in the *Getting Started with Java* booklet.

Code4 Class

Code4 Instance Methods

Code4	readLock
accessMode	readOnly
connect	safety
defaultUnique	unlock
lock	unlockAuto
lockClear	

CodeBase uses the **Code4** class to connect a client to the server and to lock data files when desired. Generally, only one **Code4** class object is used in any one application. The **Code4.connect()** method must be executed successfully before a **Data4** object can be constructed and used to manipulate a data file.

Note

It is generally recommended that only one **Code4** object be constructed per application. If more than one server connection is necessary within an application, multiple **Code4** objects may be used.

```
import codebase.*;
import java.applet.Applet;
import java.io.IOException;
import java.net.UnknownHostException;

class Example1
{
   public static void main(String args[])
       throws Error4,UnknownHostException,IOException
  {
        Code4 client = new Code4();
        client.connect(null, 23165, "user1", "");
        client.safety(true);
        client.accessMode(Code4.DENY_WRITE);
        Data4 dataFile = new Data4(client,"NAMES");
        dataFile.top();
        System.out.println("At the top of the data file");
        dataFile.close();
   }
}
```

Public Instance Methods

Code4.Code4()

Usage: public Code4( )

Description: This method creates a **Code4** object.

Code4.accessMode()

Usage: public byte accessMode( )
public byte accessMode(byte newValue)

Description: This method sets and retrieves the setting that is used to determine the access mode for the files. Changing this setting will only affect the files that are subsequently opened or created.

It is recommended that **Code4.accessMode()** be set to **Code4.DENY_RW** whenever creating files. Failure to do so can result in errors being generated by other applications accessing the files while the given process is executing.

Code4.accessMode() is set to **Code4.DENY_NONE** by default.

Parameters:

newValue The possible values are **Code4** class constants that are of the type **byte**:

Code4.DENY_NONE	Open the data files in shared mode. Other users have read and write access.
Code4.DENY_WRITE	Open the data files in shared mode. The application may read and write to the files, but other users may only open the files in read only mode and may not change the files.
Code4.DENY_RW	Open the data files exclusively. Other users may not open the files.

Note

Code4.accessMode() specifies what OTHER users will be able to do with the file. **Code4.readOnly()** specifies what the CURRENT user will be able to do with the file.

When a file is opened (which also occurs upon creation), its read/write permission attributes are obtained from **Code4.accessMode()**. If this setting is altered, the change will only affect files that are subsequently opened. Thus, to modify the read/write permission of an open file, it must first be closed and then reopened after the permission has been changed.

Returns: **byte accessMode(byte newValue)** returns the previous value.

byte accessMode() returns the current value.

Exceptions: **byte accessMode(byte newValue)** throws **IOException** and **Error4usage**.

byte accessMode() throws **Error4usage**.

Client-Server: In the client-server configuration, the **Code4.accessMode()** determines the client access to the file, but **Code4.accessMode()** does not necessarily reflect how the file is physically opened. Refer to the *openMode* field of the server configuration file for more information.

See Also: **Code4.readOnly()**

```
// Imported Classes
import codebase.*;
import java.io.IOException;
import java.net.UnknownHostException;

class Example2
{
    public static void main(String args[])
        throws Error4,IOException,UnknownHostException
    {
            Code4 client = new Code4();
            client.connect(null, -1, "user1", "");

            //open the next data file in shared mode
            client.accessMode(Code4.DENY_NONE);

            Data4 dataFile1 = new Data4(client);
            dataFile1.open("Country");

            //open next data file in read-only mode
            client.accessMode(Code4.DENY_WRITE);
            Data4 dataFile2 = new Data4(client, "employee");

            //open all subsequent data files in exclusive mode
            client.accessMode(Code4.DENY_RW);
            Data4 dataFile3 = new Data4(client, "age");

            dataFile1.close();
            dataFile2.close();
            dataFile3.close();
    }
}
```

Code4.connect

Usage: public void connect(String host, int port, String userName, String password)

Description: This method performs all of the operations necessary to connect a client application to the server.

Note

Code4.connect() must be called before the **Data4** object may constructed.

Parameters:

host This string specifies the name of the machine on which the server executable is running. If this parameter is null, the default "localhost" is used instead.

port This value specifies the port number on the server machine that the server executable is monitoring for connections. If *port* is less than 0 or greater than 32767, then the default number 23165 is used instead.

userName The server uses this string to specify the name of the user who is attempting to access the server. This parameter may be null if the server does not use name authorizations. If public access is desired, the default *userName* "public" is used.

password The server uses this string as a password to validate the name of the user identified by *userName*. This parameter may be null if the server does not use password authorizations.

Exceptions: **UnknownHostException**, **IOException**, **Error4message**, **Error4unexpected**, **Error4usage**

Code4.defaultUnique()

Usage: public byte defaultUnique( )
public byte defaultUnique(byte newValue)

Description: This setting determines how duplicate keys will be handled in unique tags. Only unique tags are affected by this setting. If this setting is altered, the change will only affect index files that are subsequently opened.

Code4.defaultUnique() is set to **Tag4info.UNIQUE_CONTINUE** by default.

This setting has no effect on Visual FoxPro tags created with **Tag4info.CANDIDATE**.

Parameters:

newValue Possible values for this setting are as follows:

Tag4info.UNIQUE This setting means that if the tag is unique and a duplicate key is encountered an **Error4unique** exception is thrown.

Tag4info.UNIQUE_CONTINUE This setting means that any duplicate keys are discarded from the tag. Consequently, there may not be a tag entry for a particular record.

Returns: **byte defaultUnique(byte newValue)** returns the previous value.

byte defaultUnique() returns the current value.

Exceptions: **byte defaultUnique(byte newValue)** throws **IOException** and **Error4usage**.

byte defaultUnique() throws **Error4usage**.

See Also: **Tag4info class**

```
import codebase.*;
import codebase.Data4;
import java.io.IOException;
import java.net.UnknownHostException;

class Example3
{
    public static void main(String args[])
        throws Error4,IOException,UnknownHostException
    {
            Code4 client = new Code4();
            client.connect(null, -1, "user1", "");

            client.safety(true);
            client.defaultUnique(Tag4info.UNIQUE);

            Data4 dataFile = new Data4(client, "country");
            //open() automatically opens a production index
            //which has a unique tag
            Field4stringBuffer country = new Field4stringBuffer(dataFile,"country");
            dataFile.top();
            System.out.println("record num before " + dataFile.recCount());
            try
            {
                //appends a copy of the top record, Error4unique thrown
                dataFile.append();
```

```
            }
            catch(Error4unique e)
            {
                System.out.println("An attempt to add a duplicate tag failed");
            }
            dataFile.close();
    }
}
```

Code4.lock

Usage: public void lock( )

Description: This method locks a group of records or files. A group lock functions as if it were a single lock on a single record, however many interdependent records and files may be locked.

The entire lock group is either in a state of having all locks held, or no locks held. If any of the locks fail, all successful locks are removed and an **Error4locked** is thrown. That is, if all of the locks were successfully performed, but the last lock failed, all of the successful locks would be removed and **Code4.lock()** would throw an **Error4locked** exception.

This high-level approach to locking minimizes the possibility of deadlock, while giving maximum flexibility.

The process involved in performing a group lock is as follows:

- Queue one or more locks with **Data4.lockAdd()**, **Data4.lockAddAll(), Data4.lockAddAppend()** or **Data4.lockAddFile()**
- Clear the queued locks, if desired, with **Code4.lockClear()** and begin the process again, or
- Attempt to place the queued locks with a single call to **Code4.lock()**. **Code4.lock()** throws an exception when it can not place all the locks.

Code4.lock() automatically clears the queue of locks once the locks are successfully performed. If **Code4.lock()** is unsuccessful, the queue of locks is maintained for a future locking attempt by **Code4.lock().**

Exceptions: **IOException**, **Error4locked**, **Error4unexpected**, **Error4usage**

See Also: **Code4.unlock(), Data4.lockAdd(), Data4.lockAddFile(), Data4.lockAddAll(), Data4.lockAddAppend()**

```
import codebase.*;
import java.io.IOException;
import java.net.UnknownHostException;

class Example4
{
    public static void main(String args[])
            throws Error4,IOException,UnknownHostException
    {
            Code4 client = new Code4();
            client.connect(null, -1, "user1", "");

            Data4 dataFile = new Data4(client, "age");
            //add record 1 to list of locks
            dataFile.lockAdd(1);
            //add record 3 to list of locks
            dataFile.lockAdd(3);

            //lock list of locks as a group of locks
            client.lock();
            System.out.println("oK");
            dataFile.close();
    }
}
```

Code4.lockClear

Usage: public void lockClear( )

Description: This method removes any group locks previously placed with **Data4.lockAdd(), Data4.lockAddAll(), Data4.lockAddAppend()** or **Data4.lockAddFile()**. No locking or unlocking is performed by this method, but rather, any queued locks are removed from the queue.

Exceptions: **Error4usage**

See Also: **Code4.lock(), Data4.lockAdd(), Data4.lockAddAll(), Data4.lockAddAppend(), Data4.lockAddFile()**

Code4.readLock()

Usage: public boolean readLock(boolean newValue)
public boolean readLock( )

Description: This method retrieves and changes the true/false flag that specifies whether other **Data4** methods should automatically lock a data record before reading. Specifically, this flag applies to the following methods, which read in a new record: **Data4.top()**, **Data4.bottom()**, **Data4.seek()**, **Data4.seekUnicode(), Data4.skip()**, **Data4.go()** and **Data4.positionSet()**.

This flag is initialized to false.

Setting **Code4.readLock()** to true can reduce performance, since locking a record often takes as long as a write to disk. For the best performance, set **Code4.readLock()** to true only when modifying several records one after another.

Parameters:

newValue When this parameter is set to true, the current record must be locked before it can be read by a method. When it is false, then the current record is not locked before it is read.

Returns: **boolean readLock(boolean newValue)** returns the previous value of the flag.

boolean readLock() returns the current value of the flag.

Exceptions: **boolean readLock(boolean newValue)** throws **IOException** and **Error4usage.**

boolean readLock() throws **Error4usage**.

See Also: **Code4.lock(), Code4.unlock()**

```
//Run this program twice at the same time. The first program will successfully lock the record
and wait for some input. The second program will attempt to lock the same record and will fail
and then it will ask you if you want to retry the lock. Entering some input for the first
program will cause it to finish executing and release the lock. Now the second program can
successfully lock the record when the lock is retried.

import codebase.*;
import java.io.IOException;
import java.net.UnknownHostException;

class Example5
{
    public static final byte SUCCESS = 0;
    public static final byte RETRY = 1;
```

```
    public static final byte QUIT = 2;

    public boolean retryLock()
    {
        char c, garbageInput;
        boolean retry = false;

        System.out.println("Retry the lock, enter Y or N");
        try
        {
           c = (char)System.in.read();
           while ((garbageInput = (char)System.in.read()) != '\n'){}
        }
        catch(IOException e)
        { c = 'N'; }

        if (c == 'Y' || c == 'y')
            retry = true;
        else
            retry = false;

        return retry;
    }
    public byte GoToRecord(Data4 data, int recordNum)
    {
        byte retry = SUCCESS;

        try
        {
            data.go(recordNum);
        }
        catch (Error4locked e)
        {
            boolean retryLock;
            retryLock = retryLock();
            if (retryLock == true)
                retry = RETRY;
            else
                retry = QUIT;
        }
        catch(Error4 e){}
        catch(IOException e){}

        return retry;
    }
    public void init()
    {

        try
        {
            byte succeed;
            int c;

            Code4 client = new Code4();
            client.connect(null, -1, "user1", "");
            client.readLock(true);
            Data4 dataFile = new Data4(client, "age");
            while ((succeed = GoToRecord(dataFile, 2)) == RETRY);
            if (succeed == SUCCESS)
                System.out.println("read lock succeeded");
            else
                System.out.println("read lock failed");
            System.out.println("enter char");
            c = System.in.read();
            dataFile.close();
```

```
        }
        catch(Error4 e)
        {
            System.out.println("An error occurred");
        }
        catch (UnknownHostException e){}
        catch (IOException e){}
    }

    public static void main(String args[])
    {
        Example5 me = new Example5();
        me.init();
    }
}
```

Code4.readOnly

Usage: public boolean readOnly( )
public boolean readOnly(boolean newValue)

Description: This true/false flag specifies whether files are to be opened in read-only mode. If this setting is altered, the change will only affect files that are subsequently opened.

There are two reasons why this switch is used. First, the application may only have read-only permission on the file. Secondly, if the application is designed to only read files and not to modify them, then opening in read-only mode would protect against application bugs that may modify the file accidentally.

This flag has no effect on how files are created.

The default setting is false.

Parameters:

newValue If *newValue* is set to true, then any subsequently opened data files are opened in read-only mode. Writing to data files is legal when *newValue* is set to false.

Returns: **byte readOnly(byte newValue)** returns the previous value.

byte readOnly() returns the current value.

Exceptions: **byte readOnly(byte newValue)** throws **IOException** and **Error4usage**.

byte readOnly() throws **Error4usage**.

```
import codebase.*;
import java.io.IOException;
import java.net.UnknownHostException;

class Example6
{
    public static void main(String args[])
            throws Error4,IOException,UnknownHostException
    {
      Code4 client = new Code4();
      client.connect(null, -1, "user1", "");
      client.readOnly(true);
      //open a file in read-only mode and then try to append a record
      Data4 dataFile1 = new Data4(client, "readonly.dbf");
      dataFile1.bottom();
      try
      {
        dataFile1.append();
      }
      catch(Error4 e)
      {
        if (e instanceof Error4unexpected)
            System.out.println("Tried to append a record to a file in read-only mode");
      }
      dataFile1.close();

      client.readOnly(false);
      //open a read-only data file (but not in read-only mode)
      //and then try to append a record
      Data4 dataFile = new Data4(client, "readonly.dbf");
      try
      {
        dataFile.append();
      }
      catch(Error4 e)
      {
         if(e instanceof Error4unexpected)
            System.out.println("Tried to append a record to a read-only file");
      }
      dataFile.close();
    }
}
```

Code4.safety()

Usage: public boolean safety( )
public boolean safety(boolean newValue)

Description: This true/false flag determines if files are protected from being automatically over-written when an attempt is made to re-create them.

This flag is initialized to true.

Parameters:

newValue Possible settings for **Code4.safety** are:

true	Files are protected from erasure by methods such as **Data4.create()** and **Data4.createIndex().** These methods will throw an **Error4file** exception, when an attempt is made to re-create a file with the safety flag set to true.
false	Files are automatically erased when an attempt is made to re-create them.

Returns: **boolean safety(boolean newValue)** returns the previous value.

boolean safety() returns the current value.

Exceptions: **boolean safety(boolean newValue)** throws **IOException** and **Error4usage**.

boolean safety() throws **Error4usage**.

```
import codebase.*;
import java.io.IOException;
import java.net.UnknownHostException;

class Example7
{
    public static void main(String args[])
            throws Error4,IOException,UnknownHostException
    {
        Code4 client = new Code4();
        client.connect(null, -1, "user1", "");
        client.safety(true);
        //try to re-create a data file
        Field4info fieldInfo = new Field4info();
        fieldInfo.add("Country", Field4info.CHARACTER, 20, 0);
        fieldInfo.add("Capital", Field4info.CHARACTER, 20, 0);
        Data4 dataFile = new Data4(client);
        try
        {
            dataFile.create("Country", fieldInfo);
        }
        catch(Error4file e)
        {
            System.out.println("Could not re-create the file");
        }
    }
}
```

Code4.unlock

Usage: public void unlock( )

Description: **Code4.unlock()** removes all locks on all open data, index, and memo files.

Exceptions: **IOException**, **Error4unexpected**, **Error4usage**

Locking: All data, index and memo files are unlocked once **Code4.unlock()** completes.

```
import codebase.*;
import java.io.IOException;
import java.net.UnknownHostException;

class Example8
{
    public static void main(String args[])
            throws Error4,IOException,UnknownHostException
    {
            Code4 client = new Code4();
            client.connect(null, -1, "user1", "");
            //lock some records
            Data4 dataFile = new Data4(client,"Country" );
            //add some records to be locked to the lock queue
            dataFile.lockAdd(1);
            dataFile.lockAdd(5);
            //lock the records
            client.lock();

            //some other code...

            //unlock the records
            client.unlock();
            dataFile.close();
    }
}
```

Code4.unlockAuto

Usage: public byte unlockAuto( )
public byte unlockAuto(byte newValue)

Description: If no parameter is passed to this method, it returns the setting of the automatic unlocking capability of CodeBase. Otherwise the automatic unlocking capability can be set according to the parameter *newValue*.

By default, CodeBase performs automatic unlocking as defined under **Code4.AUTO_ALL**, below.

Parameters: *newValue* is used to set the type of automatic unlocking used within the application. See the "Returns" section of this method for the possible settings for *newValue*.

Returns: **byte Code4.unlockAuto()** returns the current setting of the automatic unlocking. **byte Code4.unlockAuto(byte newValue)** returns the previous value.

One of the following **Code4** class constants is returned:

Code4.AUTO_OFF CodeBase performs no automatic unlocking. It is up to the application developer to unlock a record, file, or append bytes after it is no longer necessary. This setting gives maximum flexibility to the developer, but should be used with care, since it can possibly result in deadlock.

Code4.AUTO_ALL CodeBase performs automatic unlocking on the entire set of open data files. Any new lock placed using any method forces an automatic removal of every lock placed on every open data file associated with the **Code4** object. This setting is the default action taken by CodeBase.

The following scenarios illustrate how a lock is placed in CodeBase. Consider the case when a data file method such as **Data4.update()** updates a record. A record must be locked before it can be updated. At this point the record may already be locked or may need to be locked.

If a new lock must be placed, all the previous locks are unlocked according to **Code4.unlockAuto()** before the new lock is be placed. In this case, two possible results can occur depending on the setting of **Code4.unlockAuto()**. If **Code4.unlockAuto()** is set to **AUTO_OFF,** no automatic unlocking is done and new lock is placed. Thus after the update takes place all the previous locks remain in place as does the new lock. If the **Code4.unlockAuto()** is set to **AUTO_ALL,** all the locks on all open data files are removed before the new lock is placed and when the update is completed, only the new locks remain in place.

If the record is locked already, then no new locking is needed and the update can proceed. Since there are no new locks to be placed, no unlocking is required, so the setting of **Code4.unlockAuto()** is irrelevant. The locks that were in place before the update are still in place after the update is completed.

The above discussion of locking procedures not only applies to updating but to any case where locking is performed.

Note

The purpose of automatic unlocking is to make application code simpler and shorter by making it unnecessary to program unlocking protocols. In addition, automatic unlocking can prevent deadlock from occurring.

Exceptions: **byte Code4.unlockAuto()** throws **IOException** and **Error4usage**.

byte Code4.unlockAuto(byte newValue) throws **Error4usage**.

See Also: **Code4.lock(), Code4.unlock(), Data4.update()**

Data4 Class

Data4 Instance Methods

Data4	openIndex
append	pack
blank	position
bottom	positionSet
close	recCount
create	recNo
createIndex	reindex
go	seek
lockAdd	seekUnicode
lockAddAll	select
lockAddAppend	skip
lockAddFile	top
open	update

The **Data4** class contains instance methods that are used to store and retrieve information from data files. Each data file has a current record and a selected tag. The **Data4** also keeps track of the end of file (eof) and the beginning of file (bof). When the program skips past the last record, the contents of the field objects become blank. When the program attempts to skip before the first record, the contents of the field objects are unchanged.

The **Data4** constructor can not be executed successfully until a **Code4** object is created and **Code4.connect()** has been successful. A data file may be opened automatically by calling the **Data4** constructor with the name of the data file to be opened. A data file may also be explicitly opened by calling **Data4.open()** only after the **Data4.Data4()** constructor has been called. A new data file may be created by calling **Data4.create()** after the constructor has been called. A data file must be opened or created in the above manner before the other **Data4** instance methods can be called. A particular data file may be opened more than once in an application and thus there may be multiple instances of a **Data4** class for a particular data file. Any desired field objects may be created only after a data file has been opened or created. For more information concerning field objects refer to the "Field Manipulation" chapter.

Note

There are **Data4** methods which change the current record position. They are **Data4.bottom(), Data4.go(), Data4.positionSet(), Data4.seek(), Data4.seekUnicode(), Data4.skip** and **Data4.top().** These methods load the field values of the new record into the appropriate field objects and any changes to the previous record are lost. Therefore a call to **Data4.update()** must be made before the above repositioning methods are called, to save any changes to the previous record.

Once the storage and retrieval of information is completed, use **Data4.close()** to close the data file.

```
import codebase.*;
import java.io.IOException;
import java.net.UnknownHostException;

class Example9
{
    public static void main(String args[])
            throws Error4,IOException,UnknownHostException
    {
         int recCount;

         Code4 client = new Code4();
         client.connect(null, -1, "user1", "");
         Data4 dataFile = new Data4(client,"Country" );
         Field4stringBuffer country = new Field4stringBuffer(dataFile, "Country");
         dataFile.botto m();
         //now change the last record
         country.contents = new StringBuffer("Italy");
         dataFile.update();
         System.out.println(country.contents);
         dataFile.close();
    }
}
```

Public Instance Methods

Data4.Data4()

Usage: public Data4(Code4 code)
public Data4(Code4 code, String dataFileName)

Description: Both of these constructors create a **Data4** object, but the second constructor also opens a data file and its associated memo files and automatically opens a "production index file". This aspect of the second constructor is exactly the same as **Data4.open()**.

Parameters:

code — This is a an instance of the class **Code4**.

dataFileName — This is the name of the data file. If no file extension is present, extension ".DBF" is used. The name is also used to open a "production index file". See **Data4.open()** for a list of default index and memo file extensions.

If a path is provided in the *dataFileName*, it is used. Otherwise, the file is assumed to be in the current directory. In the client-server configuration the current directory is defined by the server.

Exceptions: **Data4(Code4 code)** throws **Error4usage.**

Data4(Code4 code, String dataFileName) throws **IOException**, **Error4unexpected**, **Error4file**, **Error4usage**

See Also: **Data4.open()**

```
import codebase.*;
import java.io.IOException;
import java.net.UnknownHostException;

class Example10
{
    public static void main(String args[])
            throws Error4,IOException,UnknownHostException
    {
       Code4 client = new Code4();
       client.connect(null, -1, "user1", "");
       //use Data4 constructor to automatically open a data file
       //and production index
       Data4 countryData = new Data4(client,"Country" );

       //or just construct the Data4 object and open or create a
       //data file explicitly
       Data4 ageData = new Data4(client);
```

```
      ageData.open("age");

      Field4info fieldInfo = new Field4info();
      fieldInfo.add("Name", Field4info.CHARACTER, 10, 0);
      Data4 newData = new Data4(client);
      client.safety(false);
      newData.create("NewDataFile",fieldInfo);
      countryData.close();
      ageData.close();
      newData.close();
   }
}
```

Data4.append()

Usage: public void append( )

Description: **Data4.append()** adds a new record to the end of a data file.

Data4.append() does not change the contents of the field objects. Any field that is not referenced by a field object is filled with spaces. In order to append a blank record to the data file, call **Data4.blank()** before calling **Data4.append()**. **Data4.blank()** blanks out the contents of the field objects and sets the deletion flag to false.

WARNING

The **Field4deleteFlag** class keeps track of the 'record deletion' flag associated with each record in the data file. This flag does not change when **Data4.append()** is called. To ensure that the 'record deletion' flag is false when appending a potentially deleted copy of another record, explicitly set **Field4deleteFlag.deleted** to false, or call **Data4.blank()** to reinitialize the field objects and the deletion flag.

Data4.append() maintains all open index files by adding keys to respective tags. The current record is set to the newly appended record.

Note

When many records are being appended at once, the most efficient method for locking the files is to use **Data4.lockAddAll()** followed by **Code4.lock()**.

Exceptions: **IOException**, **Error4locked**, **Error4unexpected**, **Error4unique**, **Error4usage**

Locking: The append bytes and the new record to be appended, must be locked before the append can proceed.

In the case where the append bytes required locking, the append bytes are unlocked according to **Code4.unlockAuto()** after the append is completed. On the other hand, if the append bytes were previously locked, they remain locked after the append.

The new record remains locked after the append.

See **Code4.unlockAuto()** for details on how any necessary locking and unlocking is accomplished.

See Also: **Data4.blank(), Field4deleteFlag.deleted**

```
import codebase.*;
import java.io.IOException;
import java.net.UnknownHostException;

class Example11
{
    public static void main(String args[])
           throws Error4,IOException,UnknownHostException
    {
       Code4 client = new Code4();
       client.connect(null, -1, "user1", "");
       Data4 countryData = new Data4(client,"Country" );
       Field4stringBuffer country =
                                    new Field4stringBuffer(countryData, "Country");
       Field4stringBuffer taxRate =
                                    new Field4stringBuffer(countryData, "Tax_rate");
       //append a copy of the last record
       countryData.bottom();
       countryData.append();
       System.out.println("Country field = "+country.contents);
       System.out.println("Tax Rate field = "+taxRate.contents);

       //append a blank record
       countryData.blank();
       countryData.append();
       System.out.println("Country field = "+country.contents);
       System.out.println("Tax Rate field = "+taxRate.contents);
       countryData.close();
    }
}
```

Data4.blank()

Usage: public void blank( )

Description: When **Data4.blank()** is called, the field objects associated with the data file are filled with blank values. Refer to the "Field Manipulation" chapter for the details on what constitutes a blank value.

If the current record has one or more memo entries, calling **Data4.blank()** will remove the record's reference to the entries.

Exceptions: **IOException**, **Error4usage**

See Also: **Field4stringBuffer, Field4stringBufferUnicode, Field4date, Field4double, Field4boolean, Field4deleteFlag**

Example: **See Data4.append()**

Data4.bottom()

Usage: public int bottom( )

Description: The appropriate field values are loaded from the bottom record of the data file into the corresponding field objects. The selected tag is used to determine which record is at the bottom. If no tag is selected, the last physical record of the data file is loaded.

Returns: **Status4.SUCCESS**, **Status4.EOF**

Exceptions: **IOException**, **Error4locked**, **Error4message**, **Error4unexpected**, **Error4usage**

Locking: If **Code4.readLock()** is set to true, the new record is locked before it is read.

Example: See **Data4.append()**

Data4.close()

Usage: public void close( )

Description: **Data4.close()** closes the data file and its corresponding index and memo files. **Data4close()** does not do any automatic updating. If the data file has been modified, call **Data4.update()** before **Data4.close()**.

Exceptions: **IOException**, **Error4unexpected**, **Error4usage**

Data4.create()

Usage: public void create(String dataFileName, Field4info dataFields)
public void create(String dataFileName, Field4info dataFields,
Tag4info indexTags
)

Description: **Data4.create()** creates a data file, possibly a "production" index file, and possibly a memo file. A "production" index file is created if the *indexTags* parameter is not null. A memo file is created if *dataFields* contains a memo field. See the **Field4info** and **Tag4info** classes for more information on fields and tags, respectively.

Parameters:

dataFileName The name for the data file, index file, and memo file. The default data file name extension is .DBF. See **Data4.open()** for a list of default index and memo file extensions.

If a path is provided in the string *dataFileName*, it is used. Otherwise, the file is assumed to be in the current directory. In the client-server configuration the current directory is defined by the server.

dataFields *dataFields* is a **Field4info** object. This parameter is used to specify the field definitions for the new data file. Each element in the **Field4info** object specifies a field to be created in the new data file.

indexTags This parameter is used to specify the tag definitions for the production index file. Each element in the **Tag4info** object specifies a tag to be created in the new index file. If *indexTags* is null or missing, no index file is created.

Exceptions: **IOException**, **Error4file**, **Error4unexpected**, **Error4usage**

See Also: **Field4info**, **Tag4info**

```
import codebase.*;
import java.io.IOException;
import java.net.UnknownHostException;

class Example12
{
    public static void main(String args[])
            throws Error4,IOException,UnknownHostException
    {
       Code4 client = new Code4();
       client.connect(null, -1, "user1", "");
       client.safety(false);
```

```
      Data4 newDataFile = new Data4(client);
      Field4info fieldInfo = new Field4info();
      fieldInfo.add("LastName", Field4info.CHARACTER, 10, 0);
      fieldInfo.add("Age", Field4info.NUMERIC, 3, 0);

      //create the new data file;
      newDataFile.create("NEWDATAFILE", fieldInfo);
      System.out.println("successfully created NEWDATAFILE.DBF");
      newDataFile.close();

      //try to re-create the data file with a production index
      Tag4info tags = new Tag4info();
      tags.add("Name", "LastName", null, Tag4info.DUPLICATE, false);
      newDataFile.create("NEWDATAFILE", fieldInfo, tags);
      System.out.println("successfully re-created NEWDATAFILE.DBF and
                                                    NEWDATAFILE production index");
        newDataFile.close();
    }
}
```

Data4.createIndex()

Usage: public void createIndex(String indexFileName, Tag4info indexTags)

Description: This method creates a new index file to be associated with the current data file.

Parameters:

indexFileName This string contains the name of the new index file. If no extension is provided, the default extension of .MDX is used for dBASE IV index files and .CDX is used for FoxPro index files.

When Clipper files are being used, the *indexFileName* parameter specifies the name of the index group file (.CGP). This index group file is filled with a list of the created tag files.

If a path is provided in the string *indexFileName*, it is used. Otherwise, the file is assumed to be in the current directory. In the client-server configuration the current directory is defined by the server.

If a production index file does not already exist, one can be created. The first step is to open the data file exclusively, which can be done by setting **Code4.accessMode()** with **Code4.DENY_RW** and then opening the data file. The next step is to call **Data4.createIndex()** with *indexFileName* set to null. The production index file that is created has the same name as the data file.

indexTags This parameter is used to specify the tag definitions for the index file. Each element in the **Tag4info** object specifies a tag to be created in the new index file.

Exceptions: **IOException**, **Error4file**, **Error4unexpected**, **Error4unique**, **Error4usage**

Data4.go()

Usage: public void go(int recordNumber)

Description: The field values from the specified record are loaded into the field objects.

Parameters:

recordNumber This **integer** value specifies the physical record number. To succeed, *recordNumber* must be greater than 0 and less than or equal to the value returned by **Data4.recCount()**.

Exceptions: **IOException**, **Error4entry**, **Error4locked**, **Error4message**, **Error4unexpected**, **Error4usage**

Locking: If **Code4.readLock()** returns true, then the new record is locked**.**

See Also: **Code4.readLock(), Data4.recCount()**

```
import codebase.*;
// Imported Exceptions
import java.io.IOException;
import java.net.UnknownHostException;

class example14
{
   static public void main(String args[])
   {
        try
        {
            Code4 client = new Code4();
            client.connect(null, -1, "user1", "");
            Data4 DataFile = new Data4(client);
             //open the data file;
            DataFile.open("NAMES");
            System.out.println("successfully opened NAMES.DBF");
            Field4stringBuffer name=new Field4stringBuffer(DataFile,"name");
            Field4stringBuffer f_name=new Field4stringBuffer(DataFile,"f_name");
            //Go to the 17th record
            DataFile.go(17);
            //print the current record
            System.out.println(DataFile.recNo()+" "
                +name.contents.toString()+" "
                +f_name.contents.toString());
            DataFile.close();
        }
        catch(Error4 e)
        {
```

```
            System.out.println("An error occurred");
         }
         catch (UnknownHostException e){}
         catch (IOException e){}
   }
}
```

Data4.lockAdd()

Usage: public void lockAdd(int recordNumber)

Description: This method is used to add the specified record to the list of locks placed with the next call to **Code4.lock()**.

Parameters:

recordNumber This parameter contains the physical record number of the record to be placed in the queue to be locked with **Code4.lock()**.

Note

This method performs no locking. It merely places the specified record on the list of locks to be locked by **Code4.lock()**.

Exceptions: **Error4usage**

See Also: **Code4.lock(), Data4.lockAddFile(), Data4.lockAddAll(), Data4.lockAddAppend()**

Data4.lockAddAll()

Usage: public void lockAddAll( )

Description: The data file, along with corresponding index and memo files, are added to the list of locks placed with the next call to **Code4.lock()**.

Note

This method performs no locking. It merely places the specified files on the list of locks to be locked by **Code4.lock()**.

Exceptions: **Error4usage**

See Also: **Code4.lock(), Data4.lockAdd(), Data4.lockAddFile(), Data4.lockAddAppend()**

Data4.lockAddAppend()

Usage: public void lockAddAppend( )

Description: This method is used to add the append bytes, to the list of locks placed with the next call to **Code4.lock()**.


Note

This method performs no locking. It merely places the specified file on the list of locks to be locked by **Code4.lock()**.


WARNING

If multiple updates are being made, use **Data4.lockAddAll()** instead of **Data4.lockAddAppend().** Using **Data4.lockAddAll()** will significantly improve performance.

Exceptions: **Error4usage**

See Also: **Code4.lock(), Data4.lockAdd(), Data4.lockAddAll(), Data4.lockAddFile()**

Data4.lockAddFile()

Usage: public void lockAddFile( )

Description: This method is used to add the entire data file, including the append bytes, to the list of locks placed with the next call to **Code4.lock()**.


Note

This method performs no locking. It merely places the specified file on the list of locks to be locked by **Code4.lock()**.


WARNING

If multiple updates are being made, use **Data4.lockAddAll()** instead of **Data4.lockAddFile().** Using **Data4.lockAddAll()** will significantly improve performance.

Exceptions: **Error4usage**

See Also: **Code4.lock(), Data4.lockAdd(), Data4.lockAddAll(), Data4.lockAddAppend()**

Data4.open()

Usage: public void open(String dataFileName)

Description: Method **Data4.open()** opens a data file and its corresponding memo file (if applicable). **Data4.open()** automatically tries to open a "production index file" corresponding to the data file. A data file may be opened multiple times by a single application. Thus an application may have multiple instances of a **Data4** class for a particular data file.

Under FoxPro and dBASE IV, a production index file is an index file created at the same time as the data file, using **Data4.create()**. It can also be created by using **Data4.createIndex()** with a null *indexFileName* parameter. When a production index file is automatically opened, no tag is initially selected.

When Clipper index files are being used, **Data4.open()** assumes that there is a corresponding group file (.CGP) and attempts to open it. If there is no group file, an **Error4file** exception is thrown.

Listed below are the default file extensions for the different file formats that are compatible with CodeBase. If an extension is not provided to **Data4.open()**, the default extension is used.

	Data	Index	Memo
dBASE IV	".DBF"	".MDX"	".DBT"
Clipper	".DBF"	".CGP"	".DBT"
FoxPro	".DBF"	".CDX"	".FPT"

WARNING

Data4.open() does not leave the data file at a valid record position. Call a positioning method such as **Data4.top()** to move to a valid record. It is inappropriate to call **Data4.skip()** until a valid record has been loaded into the record buffer.

Parameters:

dataFileName — This is the name of the data file to open. If a path is provided in the string, it is used. Otherwise, the file is assumed to be in the current directory. In the client-server configuration the current directory is defined by the server.

Note

The backslash '\' is an escape character (eg. '\n' is the code for newline). Therefore, use a double backslash '\\' to represent a backslash in a static string specifying a path under Windows 95 and Windows NT otherwise use a forward slash.

Exceptions: **IOException, Error4unexpected, Error4file, Error4usage**

See Also: **Code4.connect()**, **Data4.close()**

```
import codebase.*;
//imported exceptions
import java.io.IOException;
import java.net.UnknownHostException;

class example15
{
   static public void main(String args[])
   {
        try
        {
            Code4 client = new Code4();
            client.connect(null, -1, "user1", "");
            Data4 DataFile = new Data4(client);
            //open the data file;
            DataFile.open("NAMES");
            System.out.println("successfully opened NAMES.DBF");
            System.out.println("Record Count: "+DataFile.recCount());
            DataFile.close();
        }
        catch(Error4 e)
        {
            System.out.println("An error occurred");
        }
        catch (UnknownHostException e){}
        catch (IOException e){}
   }
}
```

Data4.openIndex()

Usage: public void openIndex(String fileName)

Description: This method opens a index file specified by *fileName* for the current data file.

Parameters:

fileName This parameter is the name of the index file to be opened. If a path is provided in the string, it is used. Otherwise, the file is assumed to be in the current directory. In the client-server configuration the current directory is defined by the server.

Exceptions: **IOException**, **Error4unexpected**, **Error4file**, **Error4usage**

See Also: **Data4.open()**

Data4.pack()

Usage: public void pack( )

Description: **Data4.pack()** physically removes all records marked for deletion from the data file. **Data4.pack()** automatically reindexes all open index files attached to the data file.

After **Data4.pack()** completes, the contents of the field objects is undefined. Explicitly call a positioning function such as **Data4.top()** to position to a desired record.

Appropriate backup measures should be taken before calling this function.

Data4.pack() does not alter the memo file. Consequently, memo entries referenced by removed records will be wasted disk space.

Consider opening the data file exclusively (with **Code4.accessMode()** set with **Code4.DENY_RW**) before packing, since **Data4.pack()** can seriously interfere with the data retrieved by other users.

Exceptions: **IOException, Error4locked, Error4unexpected, Error4unique, Error4usage**

Locking: The data file and its index files are locked. See **Code4.unlockAuto()** for details on how any necessary locking and unlocking is accomplished.

See Also: **Code4.unlockAuto()**

```
import codebase.*;
import java.util.Date;
import java.io.IOException;
import java.net.UnknownHostException;

class Example26
{
    public static void main(String args[])
            throws Error4,IOException,UnknownHostException
    {
            Code4 client = new Code4();
            client.connect(null, -1, "user1", "");
            client.safety(false);
            client.accessMode(Code4.DENY_RW);
            Data4 db=new Data4(client);
            db.open("NAMES");
            Field4deleteFlag df=new Field4deleteFlag(db);
            System.out.println(db.recCount());
            db.top();
            df.deleted=true;
            db.update();
            System.out.println(db.recCount());
            db.lockAddAll();
            client.lock();
            db.pack();
            client.unlock();
            System.out.println(db.recCount());
            db.close();
    }
}
```

Data4.position()

Usage: public double position( )

Description: This method evaluates the relative position of the current record in the data file and returns the value as a percentage. For example, if the current position is half way down the data file, **Data4.position()** returns 0.5.

If there is a currently selected tag, **Data4.position()** calculates the position of the current record according to its relative position in the tag. If no tag is selected, record number ordering is used.

The main use of **Data4.position()** is to facilitate the use of scroll bars when developing edit and browse methods.

Returns:

>= 0 The current position in the data file represented as a percentage. This function returns **(double)** 1.1 if the end of file condition is true, and **(double)** 0.0 if the beginning of file condition is true or if the data file is empty.

Exceptions: **IOException, Error4locked, Error4unexpected, Error4usage**

See Also: **Data4.positionSet()**

```
import codebase.*;
//imported exceptions
import java.io.IOException;
import java.net.UnknownHostException;

class example16
{
   static public void main(String args[])
    throws Error4,IOException,UnknownHostException
   {
        try
        {
            Code4 client = new Code4();
            client.connect(null, -1, "user1", "");
            Data4 DataFile = new Data4(client);
             //open the data file;
            DataFile.open("INFO");
            DataFile.select("AGE_TAG");
            DataFile.positionSet(.25);
            System.out.println("Record Number: "+DataFile.recNo());
            System.out.println("Position: "+DataFile.position());
            DataFile.close();
        }
        catch(Error4 e)
        {
            System.out.println("An error occurred");
        }
        catch (UnknownHostException e){}
        catch (IOException e){}
   }
}
```

Data4.positionSet()

Usage: public int positionSet(double newPosition)

Description: When **Data4.positionSet()** is called, *newPosition* is taken as a percentage, and the record closest to that percentage becomes the current record. (i.e. the field values from the record are loaded into the field objects).

This method uses the currently selected tag to specify the ordering. If no tag is selected, record number ordering is used.

The main use of **Data4.positionSet()** is to facilitate the use of scroll bars when developing edit and browse methods.

If **Data4.positionSet()** cannot find a record at the precise location specified by *newPosition*, the next record in sequence is used. For example, if a data file has three records and **Data4.positionSet**(0.25) is called, the second record (which is actually at 0.5) is used instead. In this case, **Data4.position()** does not return the same value as passed to **Data4.positionSet()**.

When **Data4.positionSet()** is used with a selected tag, it may not position to the exact position within the tag. This is due to the nature of the index file structure. The inaccuracy becomes less apparent in larger data files.

Parameters:

newPosition This is a percentage that indicates where in the data file the current record will come from.

Returns: **Status4.SUCCESS**, **Status4.EOF**

Exceptions: **IOException**, **Error4entry**, **Error4locked**, **Error4message**, **Error4unexpected**, **Error4usage**

Locking: If **Code4.readLock()** returns true, the new record is locked before it is read.

See Also: **Data4.update()**

Example: **See Data4.position()**

Data4.recCount()

Usage: public int recCount( )

Description: The number of records in the data file is returned.

If the data file is shared, the record count might not reflect the most recent additions made by other users.

Returns:

>= 0 This is the number of records in the data file.

Exceptions: **IOException**, **Error4unexpected**, **Error4usage**

See Also: **Data4.recNo()**

```
import codebase.*;
import java.io.IOException;
import java.net.UnknownHostException;

class Example17
{
    public static void main(String args[])
            throws Error4,IOException,UnknownHostException
    {
            Code4 client = new Code4();
            client.connect(null, -1, "user1", "");
            Data4 db=new Data4(client);
            db.open("NAMES");
            int count=db.recCount();
            System.out.println("Record Count: "+count);
            db.close();
    }
}
```

Data4.recNo()

Usage: public int recNo( )

Description: The current record number of the data file is returned. If the record number returned is greater than the number of records in the data file, this indicates an end of file condition.

Returns:

`>= 1` The current record number.

`0` There is no current record number. **Data4.append()** has just been called to append a record.

`-1` There is no current record number. The file has just been opened or created, so there is no current record.

Exceptions: **IOException, Error4unexpected, Error4usage**

See Also: **Data4.recCount()**

```
import codebase.*;
import java.io.IOException;
import java.net.UnknownHostException;

class Example18
{
    public static void main(String args[])
            throws Error4,IOException,UnknownHostException
    {
            Code4 client = new Code4();
            client.connect(null, -1, "user1", "");
            Data4 db=new Data4(client);
            db.open("INFO");
            db.select("NAME_TAG");
            int count=0;
            for (int rc=db.top();rc==Status4.SUCCESS;rc=db.skip(1))
```

```
            {
                System.out.println("Position in tag: "+ (++count));
                System.out.println("RecordNumber: "+ db.recNo());
            }
            db.close();
    }
}
```

Data4.reindex()

Usage: public void reindex( )

Description: All of the index files corresponding to the data file are recreated. It is generally a good idea to open the files exclusively (set **Code4.accessMode()** to **Code4.DENY_RW** before opening the data file) before reindexing.

After **Data4.reindex()** completes, the contents of the field objects undefined. Explicitly call a positioning function such as **Data4.top()** to position to a desired record.

Exceptions: **IOException, Error4locked, Error4unexpected, Error4unique, Error4usage**

Locking: The data file and corresponding index files are locked. It is recommended that the index files be opened exclusively if a reindex is to occur. This will ensure that other users cannot access the file while it is being reindexed. Applications that access index files that have not been fully reindexed may generate errors or obtain incorrect database information.

```
import codebase.*;
import java.util.Date;
import java.io.IOException;
import java.net.UnknownHostException;

class Example27
{
    public static void main(String args[])
            throws Error4,IOException,UnknownHostException
    {
            Code4 client = new Code4();
            client.connect(null, -1, "user1", "");
            client.safety(false);
            client.accessMode(Code4.DENY_RW);
            Data4 db=new Data4(client);
            db.open("INFO");
            db.reindex();

            db.close();
    }
}
```

Data4.seek()

Usage: public int seek(String seekValue)
public int seek(double seekDouble)
public int seek(Date seekDate)

Description: Method **Data4.seek()** searches for a record using the currently selected tag if one is selected (See **Data4.select()**). If a tag has not been selected, then the first tag of the first opened index is used. Once the search value is located in the tag, the corresponding data file record is read into the field objects associated with the fields. Searching always begins from the first logical record in the database as determined by the selected tag.

Use **Data4.seek(String seekValue)** to search in data files that store their character field data in ASCII format. **Field4stringBuffer** objects are used to write data to disk in ASCII format. **Data4.seek(String seekValue)** converts the Unicode characters in *seekValue* into ASCII characters and then searches the data file.

In order to use **Data4.seek(double seekDouble),** the tag key must be of type Numeric or Floating Point. **Data4.seek(Date seekDate)** is used for seeking in Date tags. **Data4.seek**(**String seekValue)** may be used with any tag type, as long as it is formatted correctly. If a character field is composed of binary data, **Data4.seek(String seekValue)** may be used to seek without regard for nulls. Seeking on memo fields is not allowed.

Parameters:

seekValue *seekValue* is a **String** object containing the value for which the search is conducted.

If the tag is of type Date, the date search value should be formatted "CCYYMMDD" (Century, Year, Month, Day).

If the tag is of type Character, the *seekValue* may have fewer characters than the tag's key length. In this case, a search is done for the first key which matches the supplied characters. If *seekValue* is longer than the tag key length, the extra characters are ignored.

If the tag is of type Numeric, the character value should represent a valid number. E.g.: data.seek("33.7"). This number is internally converted to a **double** value before the seek is performed.

Data4.seek(double seekDouble) does not need to make this conversion, and is generally preferable when used on numeric keys.

seekDouble *seekDouble* is a **double** value used to seek in Numeric or Floating Point keys.

seekDate *seekDate* is a Java Date object, which is set to a value that is used to seek in Date keys.

Returns: **Status4.SUCCESS**, **Status4.AFTER**, **Status4.EOF**

Exceptions: **IOException**, **Error4entry**, **Error4locked**, **Error4message**, **Error4unexpected**, **Error4usage**

Locking: If **Code4.readLock()** returns true, the new record is locked before it is read.

See Also: **Data4.select()**, **Code4.readLock()**, **Data4.seekUnicode()**

```
import codebase.*;
import java.io.IOException;
import java.net.UnknownHostException;

class Example19
{
    public static void main(String args[])
            throws Error4,IOException,UnknownHostException
    {
            Code4 client = new Code4();
            client.connect(null, -1, "user1", "");
            Data4 db=new Data4(client);
            db.open("INFO");
            Field4double age=new Field4double(db,"age");
            db.select("NAME_TAG");
            int rc;
            rc=db.seek("Ginger");
            if (rc==Status4.SUCCESS)
            {
                System.out.println("Ginger is record: "+ db.recNo());
            }
            db.select("AGE_TAG");
            rc=db.seek(0.0);
            if ((rc==Status4.SUCCESS)||(rc==Status4.AFTER))
            {
                System.out.println("The youngest age is: "+ age.contents);
            }

            db.close();
    }
}
```

Data4.seekUnicode()

Usage: public int seekUnicode(String seekValue)

Description: Method **Data4.seekUnicode()** searches for a record using the currently selected tag if one is selected (See **Data4.select()**). If a tag has not been selected, then the first tag of the first opened index is used. Once the search value is located in the tag, the corresponding data file record is read into the field objects associated with the fields. Searching always begins from the first logical record in the database as determined by the selected tag.

Use **Data4.seekUnicode(String seekValue)** to search in data files that store their character field data in Unicode format. **Field4stringBufferUnicode** objects are used to write data to disk in Unicode format. **Data4.seekUnicode(String seekValue)** uses the Unicode characters in *seekValue* and then searches the data file directly.

Data4.seekUnicode(**String seekValue)** may be used with any tag type, as long as it is formatted correctly. **Data4.seekUnicode(String seekValue)** may be used to seek without regard for nulls. Seeking on memo fields is not allowed.

Parameters:

seekValue *seekValue* is a **String** object containing the value for which the search is conducted.

If the tag is of type Date, the date search value should be formatted "CCYYMMDD" (Century, Year, Month, Day).

If the tag is of type Character, the *seekValue* may have fewer characters than the tag's key length. In this case, a search is done for the first key which matches the supplied characters. If *seekValue* is longer than the tag key length, the extra characters are ignored.

If the tag is of type Numeric, the character value should represent a valid number. E.g.: data.seekUnicode("33.7"). This number is internally converted to a **double** value before the seek is performed.

Returns: **Status4.SUCCESS**, **Status4.AFTER**, **Status4.EOF**

Exceptions: **IOException, Error4entry, Error4locked, Error4message**, **Error4unexpected, Error4usage**

Locking: If **Code4.readLock()** returns true, the new record is locked before it is read.

See Also: **Data4.select(), Code4.readLock(), Data4.seek()**

Data4.select()

Usage: public void select(String tagName)

Description: **Data4.select()** selects a tag to be used for the next data file positioning statements. The selected tag is used by positioning calls to **Data4.skip()**, **Data4.seek(), Data4.seekUnicode(), Data4.positionSet()**, **Data4.top()**, and **Data4.bottom()**.

Parameters:

tagName This is a string containing the name of the tag for which a search is conducted. If *tagName* cannot be found, **Data4.select()** throws an exception and the previously selected tag remains selected. To select record number ordering, call **Data4.select()** with a null string.

See Also: **Data4.skip(), Data4.seek(), Data4.positionSet(), Data4.top(), Data4.bottom()**

Example: **See Data4.seek()**

Data4.skip()

Usage: public int skip(int numRecords)

Description: This method skips *numRecords* from the current record number using the selected tag. If no tag is selected, record number ordering is used. Make sure that the data file is positioned to a valid record before attempting to skip.

The field values from the new record are loaded into the field objects and the new record becomes the current record.

It is possible to skip one record past the last record in the data file and create an end of file condition. If an attempt is made to skip above the first record in the data file, a beginning of file condition is created and data file remains positioned at the first record..

Parameters:

numRecords The number of logical records to skip forward. If *numRecords* is negative, the skip is made backwards.

Returns: **Status4.SUCCESS**, **Status4.BOF**, **Status4.EOF**

Exceptions: **IOException, Error4locked, Error4message**, **Error4unexpected, Error4usage**

Locking: If **Code4.readLock()** returns true, the new record is locked before it is read.

See Also: **Code4.readLock()**

```
import codebase.*;
import java.io.IOException;
import java.net.UnknownHostException;

class Example20
{
    static void printFile(Data4 db)
     throws Error4, IOException
    {
      int rc;
      Field4stringBuffer name=new Field4stringBuffer(db,"name");
      Field4stringBuffer f_name=new Field4stringBuffer(db,"f_name");
      rc=db.top();
      while (rc==Status4.SUCCESS)
      {
         System.out.println(db.recNo()
                                +" "+name.contents.toString()
                                +" "+f_name.contents.toString());
         rc=db.skip(1);
      }
    }

    public static void main(String args[])
            throws Error4,IOException,UnknownHostException
    {
            Code4 client = new Code4();
            client.connect(null, -1, "user1", "");
            Data4 db=new Data4(client);
            db.open("NAMES");
            //print the entire file using skip
            printFile(db);
            db.close();
    }
}
```

Data4.top()

Usage: public int top( )

Description: The field values from the top record of the data file are read into the field objects and the current record number is updated. The selected tag is used to determine logically which is the top record. If no tag is selected, the first physical record of the data file is used.

Returns: **Status4.SUCCESS, Status4.EOF**

Exceptions: **IOException, Error4locked, Error4message**, **Error4unexpected, Error4usage**

Locking: If **Code4.readLock()** returns true, the top record is locked before it is read.

See Also: **Data4.bottom()**, **Code4.readLock()**

```
import codebase.*;
import java.io.IOException;
import java.net.UnknownHostException;

class Example21
{
    public static void main(String args[])
            throws Error4,IOException,UnknownHostException
    {
            Code4 client = new Code4();
            client.connect(null, -1, "user1", "");
            Data4 db=new Data4(client);
            db.open("INFO");
            db.select("NAME_TAG");
            int count=0;
            for (int rc=db.top();rc==Status4.SUCCESS;rc=db.skip(1))
                count++;
            System.out.println(count+" records in tag");
            db.close();
    }
}
```

Data4.update()

Usage: public void update( )

Description: The current record is updated. Any changes to the field objects associated with the data file are written to disk.

Exceptions: **IOException**, **Error4locked**, **Error4unexpected**, **Error4unique**, **Error4usage**

Locking: If changes have been made to any field, the record must be locked before it can be updated. If a new lock is required, everything is unlocked according to the **Code4.unlockAuto()** setting and then the required lock is placed. If the required lock is already in place, **Data4.update()** does not unlock or lock anything and after the update is completed, the previous locks remain in place.

The index files and append bytes may need to be locked during updates. If index files require locking, they are locked prior to the update. After the update is complete, the index files are unlocked. If the index files are already locked, no new locking is required and after the update is finished, the index files remain locked.

```
import codebase.*;
import java.io.IOException;
import java.net.UnknownHostException;

class Example22
{
    public static void main(String args[])
            throws Error4,IOException,UnknownHostException
    {
            Code4 client = new Code4();
            client.connect(null, -1, "user1", "");
            Data4 db=new Data4(client);
            db.open("INFO");
            Field4double age=new Field4double(db,"AGE");
            db.go(2);
            age.contents=new Double(49);
            db.update();
            db.close();
    }
}
```

Exception Classes

The **Error4** class is an abstract class that is a subclass of the Java Exception class. **Error4** is the base class for the other CodeBase exception classes.

The CodeBase exception classes that are subclassed from the **Error4** are as follows: **Error4entry**, **Error4field**, **Error4file**, **Error4locked**, **Error4message**, **Error4unexpected**, , **Error4tagName**, **Error4unique**, **Error4usage**. These subclassed classes may be captured specifically or **Error4** can be used to capture all the exceptions. If **Error4** is captured then the exact exception that was thrown can be determined by an **if** statement using the Java operator **instanceof .** The **if** statement will return true if the exception is an **instanceof** the specified exception. See below for an example.

```
try
{
    //some code ...
}
catch ( Error4locked e )
{
    //an exception captured specifically
    System.out.println( "An Error4locked exception" )
}
catch ( Error4 e )
{
    //the general Error4 exception captured and then the specific exception determined
    if ( e instanceof Error4usage )
        System.out.println( "An Error4usage Exception" )
}
```

The **Error4unexpected** exception is thrown when the server returns an error. **Error4unexpected** has a public instance method called **getCode()** that will return the value of the server error. This exception covers a number of different errors, whereas the other exception classes are only thrown under specific circumstances. Descriptions of exception classes are given in the table below.

Exception Classes

Class	Meaning of Exception
Error4	This exception class is the base class for the other exception classes listed below. This exception may be used to capture any of the subclassed exceptions.
Error4entry	This exception is thrown when a method tries to reposition the data file to a record that doesn't exist.
Error4field	This exception is thrown under three different circumstances. **Error4field** is thrown when the specified field name is not valid. This exception can also mean that the specified field can not be associated with the chosen field object because the field types do not correspond. For example, if you try to create an instance of a **Field4boolean** object and associate it with a Date field, this exception will be thrown. This exception may also mean that an attempt was made to associate a field object with a field that already has a field object associated with it. A particular instance of a **Data4** class for a data file may only have one field object associated with a particular field.
Error4file	This exception is thrown when data file or an index file can not be created or opened.
Error4locked	This exception is thrown when a method attempts to lock an item but another user has already locked the item, thus resulting in a failed lock attempt.
Error4message	This exception means that there is a problem with the messaging protocol used to connect to the server.
Error4tagName	This exception means that the specified tag name was not valid.
Error4unexpected	This exception means that an error was returned from the server.
Error4unique	This exception means that an attempt was made to add a duplicate key to a unique tag when **Code4.defaultUnique()** was set to **Tag4info.UNIQUE** or the tag was created with the **Tag4info.CANDIDATE** setting.
Error4usage	This exception means that method was used in an incorrect manner or the parameters were invalid.

Public Instance Method

Error4unexpected.getCode()

Usage: public short getCode( )

Description: This method returns the server error that produced the **Error4unexpected** exception. Refer the *CodeBase C Reference Guide*, "Appendix A: Error Codes" for more details on the exact meaning of the server error.

```
try
{
    Code4 client = new Code4();
    client.connect(null, -1, "user1", "");
    Data4 dataFile = new Data4(client,"Country" );
}
catch (Error4unexpected e)
{
    short error = e.getCode()  //the server error is returned
}
```

Field Manipulation Classes

The **Field4stringBuffer, Field4stringBufferUnicode, Field4double, Field4date** and **Field4boolean** classes are used to access and store field values from the current record. Each of these classes can be mapped to several different field types. The **Field4stringBuffer** and **Field4stringBufferUnicode** classes can be mapped to any field type, which includes Character, Date, Double, Logical, Numeric, Floating Point, Memo, General and Binary. The **Field4double** class can be mapped to Numeric, Floating Point and Character fields. The **Field4date** class can be mapped to a Date or Character field. The **Field4boolean** class can be mapped to Logical and Character fields.

Note

Only one field object may be associated with a particular field while a data file is open. If you want to change the field objects associated with a data file, you must first close the data file, reopen it, and then associate new field objects. Alternatively, you can open a second instance of a data file and associate different field objects with it.

We recommend that you create a new class that extends the **Data4** class and create all the field objects that will be needed in this class. The following code illustrates how to create the subclass and how to use it in an example.

```
import codebase.*;
import java.io.IOException;
import java.net.UnknownHostException;


public class customer extends Data4
{
    Field4stringBuffer lastName, firstName;

    public customer(Code4 code)
        throws Error4,IOException,UnknownHostException
    {
      super(code);
      open("names.dbf");
      lastName = new Field4stringBuffer(this, "NAME");
      firstName = new Field4stringBuffer(this, "F_NAME");
    }

    public void printRecords()
        throws Error4,IOException,UnknownHostException
    {
        top();
        for (int i=1; i<=recCount(); i++)
        {
```

```
            System.out.println(firstName.contents+" "+lastName.contents);
            skip(1);
         }
    }
}
```

```
import codebase.*;
import customer;
import java.io.IOException;
import java.net.UnknownHostException;

class fieldEg
{
   public static void main(String args[])
       throws Error4,UnknownHostException,IOException
  {

    Code4 client = new Code4();
    client.connect(null, -1, "user1", "");
    customer newCust = new customer(client);
    newCust.printRecords();
    newCust.close();
  }
}
```

For every field in the data file that needs to be accessed, an appropriate object must be created and associated with a field. Each of these objects has an instance variable called **contents** that contains the value of that field from the current record. It is through the variable **contents** that the field can be retrieved and modified directly. When the data file is positioned to a new record, the field values of the new record are copied into the instance variables of the field objects associated with the data file. Any changes made to the previous record will be lost unless **Data4.update()** is called before a reposition method. If the data file is positioned to a valid record and a new field object is created, the **contents** will automatically be filled with the corresponding field value of the current record.

CodeBase supports the Visual FoxPro null fields. If a field supports nulls and the **contents** variable of the field object is null then **Data4.update()** will save the null field value to disk. If the field does not support nulls, a null value in the **contents** variable of the field object is saved to disk as spaces. In CodeBase null fields can be created for a new data file by using **Field4info.add()** and setting the *allowNull* parameter to be true. The *allowNull* parameter is ignored if Visual FoxPro is not supported and the field will not support nulls.

Note

There is a difference between a field supporting nulls and storing nulls in the field object associated with a field. Null can be assigned to the **contents** variable of any kind of field object. It is when **Data4.update()** is called to write the changes to disk that the difference arises. Nulls are written to disk only when using Visual FoxPro and the field supports nulls. In any other case, nulls in the field objects are not written to disk; instead the field is filled with spaces.

The **Field4stringBuffer, Field4stringBufferUnicode, Field4double, Field4date** and **Field4boolean** classes are based on the **Field4** class. The **Field4** class has three public instance methods that are inherited by the above subclasses. These methods are **decimals()**, **type()** and **length()**, which return the number of decimals, the field type and the length of the field respectively.

Access to the contents of a database field depend upon the database being positioned to a valid record. That is, no assignments or retrieval of information may be done if the database is just opened or created and has not been explicitly positioned (e.g. calling **Data4.top()**, **Data4.go()**, etc.).

The **Field4deleteFlag** class keeps track of whether the current record is marked as deleted or not.

The **Field4info** class is used to define fields for a new data file and is used by **Data4.create().**

The following discussion of field types refers to how the field is stored on disk. The field values can only be accessed through the field objects and the field values are stored in the objects in a different manner than on disk. Refer to individual field classes for information on how the field values are stored.

Field Types

dBASE data files, including those created and used by CodeBase, have several possible field types:

Character Fields

Character fields usually store character information. The maximum width, for dBASE/FoxPro file compatibility, is 254 characters. However, you can increase the width to the CodeBase maximum, 32K, and still maintain Clipper data file compatibility.

CodeBase lets you store any binary data, including normal alphanumeric characters, in a Character field.

Date Fields

Date fields, which have a width 8, contain information in the following character format: CCYYMMDD (Century, Year, Month, Day).

E.g. "19900430" is April 30th, 1990

Floating Point Fields

dBASE IV introduced this field type. This field type is identical to a Numeric field with regard to how data is stored in the data file. CodeBase treats this field type as a Numeric field.

Logical Fields

This field type, of width 1, stores Boolean data as one of the following characters: Y, y, N, n, T, t, F or f.

Memo Fields

This is a memo field. It is more complicated than other field types because the variable length memo field data is stored in a separate memo file. The data field contains a reference into the memo file.

By using the **Field4stringBuffer** or **Field4stringBufferUnicode** class this extra complexity is hidden. From a user perspective, the memo fields are similar to Character fields.

There can be lots of data for a single memo field entry, up to 1 billion bytes per memo entry. A memo entry may store binary as well as character data.

Numeric Fields

This field type is used to store numbers in character format. The maximum length of the field depends on the format of the file.

File Format	Field Length	Maximum Number of Decimals
Clipper	1 to 19	minimum of (length - 2) and 15
FoxPro	1 to 20	(length - 1)
dBASE IV	1 to 20	(length - 2)

	In the data file, the numbers are represented by using the following characters: '+', '-', '.', and '0' through '9'.
Binary Fields	CodeBase treats this field type as though it was a memo field, except that the associated memo file contains binary information. The **Field4stringBuffer** or **Field4stringBufferUnicode** class should be used to manipulate the binary entry. This field type provides compatibility with other products that can manipulate binary fields. This field type is only supported by the dBASE IV file format.
General Fields	CodeBase treats this field type as though it was a memo field, except that the associated memo file contains OLEs. This field type is not directly supported by CodeBase, but it provides compatibility with other products, such as FoxPro, which can manipulate OLEs. This field type is only supported by the FoxPro file format.

Note

Since the dBASE data file standard (used by Clipper and FoxPro) stores all information in the data file as characters, the **Field4stringBuffer** and **Field4stringBufferUnicode** classes may be used no matter the defined type of the field.

```
import codebase.*;
import java.util.Date;
import java.io.IOException;
import java.net.UnknownHostException;

class Example24
{
    public static void main(String args[])
            throws Error4,IOException,UnknownHostException
    {
            Code4 client = new Code4();
            client.connect(null, -1, "user1", "");
            client.safety(false);
            Data4 db=new Data4(client);

            Field4info fi=new Field4info();
            fi.add("boolean",'L',1,0);
            fi.add("string",'C',10,0);
            fi.add("date",'D',8,0);
            fi.add("double",'N',8,0);
            fi.add("unicode",'C',10,0);

            db.create("NEWDATA",fi);
            Field4boolean mboolean=new Field4boolean(db,"boolean");
            Field4stringBuffer mstring=new Field4stringBuffer(db,"string");
            Field4date mdate=new Field4date(db,"date");
            Field4double mdouble=new Field4double(db,"double");
            Field4stringBufferUnicode municode=
                                    new Field4stringBufferUnicode(db,"unicode");
```

```
            db.top();
            mboolean.contents=new Boolean(true);
            mstring.contents=new StringBuffer("Stuff");
            Date now=new Date();
            mdate.contents=now;
            mdouble.contents=new Double(123);
            municode.contents=new StringBuffer("Stuff");
            db.append();

            db.top();
            System.out.println(mboolean.contents + " " +
                mstring.contents + " " + mdate.contents + " " +
                mdouble.contents + " " + municode.contents);

            db.close();
    }
}
```

Field4 Class

The **Field4stringBuffer, Field4stringBufferUnicode, Field4double, Field4date** and **Field4boolean** classes are based on the **Field4** class. Therefore they can inherit the **decimals(), length()** and **type()** instance methods from the **Field4** class.

Public Instance Methods

Field4.decimals()

Usage: public int decimals( )

Description: This method returns a the number of decimal places in the field associated with the field object.

Exceptions: **IOException**, **Error4unexpected**, **Error4usage**

Field4.length()

Usage: public int length( )

Description: This method returns the length of the field associated with the field object.

Note

When using **Field4stringBufferUnicode**, the characters are stored to disk using the Unicode format. **Field4stringBufferUnicode.length()** returns the dBASE length of the field and not the length relative to the number of Unicode characters that fit in the field. Unicode characters are represented by two bytes, whereas in dBASE, a character is represented by one byte. Therefore if the field has a dBASE length of 10, only 5 Unicode characters can fit in the field.

Exceptions: **IOException**, **Error4unexpected**, **Error4usage**

Field4.type()

Usage: public char type( )

Description: This method returns a character representing the field type of the associated field.

Returns: This method returns one of the following **Field4info** class constants representing the field type.

BINARY	Binary field.
CHARACTER	Character field.
DATE	Date field.
FLOAT	Floating Point field.
GENERAL	General field.
LOGICAL	Logical field.
MEMO	Memo field.
NUMERIC	Numeric field.

Exceptions: **IOException**, **Error4unexpected**, **Error4usage**

Field4boolean Class

Use the **Field4boolean** class when trying to access a logical field or a character field. Call the constructor **Field4boolean.Field4boolean()** to create the object and associate it to the particular field. When the data file is positioned to a valid record, the value of field corresponding to that record is copied into the variable **Field4boolean.contents** where it can be retrieved or modified.

When **Data4.blank()** is called, the **Field4boolean.contents** is set to false, which represents the blank value for the **Field4boolean** class. If the **Field4boolean.contents** is set to the blank value and the record is updated by **Data4.update()**, 'F' is saved to the disk. On the other hand, if **Field4boolean.contents** is set to true and the record is updated, 'T' is save to the disk.

WARNING

Field4boolean.contents is set to true if the field value on the disk is equal 'T', 't', 'Y' or 'y', otherwise it is set to false. If a record is updated and **contents** is set to true then 'T' is saved to disk. If **contents** is set to false, 'F' is saved to disk. Therefore, the value on the disk may be implicitly changed when the record is updated even if **contents** was not modified. For example, say the field value in a logical field is set to 'f'. In this case, the **Field4boolean.contents** associated with the field is set to false. If the record is updated, the value saved to disk will be 'F', thus changing the field value from 'f' to 'F' even though **contents** was not modified.

Public Instance Variable

Field4boolean.contents

Usage: public Boolean contents

Description: This variable contains the contents of a field from the current record. To change the field value, modify **contents** directly. Any changes to this variable are not permanent. When the data file is positioned to a different record, the new field value is copied into this variable, so any changes to the previous record field value will be lost. Call **Data4.update()** to write any changes to the disk before the data file is repositioned, thus making the changes permanent.

See Also: **Data4.update()**

Public Instance Methods

Field4boolean.Field4boolean()

Usage: public Field4boolean(Data4 data, String fieldName)

Description: This method is a constructor for the object and it also associates the object with a particular field specified by *fieldName*.

Parameters:

data This is the reference to the data file with which the field is associated.

fieldName This is the name of the field to be associated with the **Field4boolean** object.

Exceptions: **IOException**, **Error4field**, **Error4message**, **Error4unexpected**, **Error4usage**

Field4date Class

Use the **Field4date** class when trying to access a date field or a character field. Call the constructor **Field4date.Field4date()** to create the object and associate it to the particular field. When the data file is positioned to a valid record, the value of that field corresponding to the record is copied into the variable **Field4date.contents** where it can be retrieved or modified. The **Field4date.contents** is a Java Date object, which can store a specific date and time. Even though the time element of the **Field4date.contents** can be altered, it is not save to disk when an update is performed.

Note

The earliest date that can be stored in a **Field4date** object is January 3, 1970. January 1, 1970, midnight is used to represent an "old" date. If the date on the disk is older than January 3, 1970, then the "old" date is loaded into the **Field4date.contents** variable. January 2, 1970, midnight is used as the blank value. If **contents** is set to the blank value and the record is updated, spaces will be stored on the disk. **Data4.blank()** or **Field4date.setBlank()** can be used to set **Field4date** objects to the blank value.

Use a **Field4stringBuffer** object to store dates earlier than January 3, 1970.

Public Instance Variable

Field4date.contents

Usage: public Date contents

Description: This variable contains the contents of a field from the current record. To change the field value, modify **contents** directly. Any changes to this variable are not permanent. When the data file is positioned to a different record, the new field value is copied into this variable, so any changes to the previous record field value will be lost. Call **Data4.update()** to write any changes to the disk before the data file is repositioned, thus making the changes permanent.

See Also: **Data4.update()**

Public Instance Methods

Field4date.Field4date()

Usage: public Field4date(Data4 data, String fieldName)

Description: This method is a constructor for the object and it also associates the object with a particular field specified by *fieldName*.

Parameters:

data This is the reference to the data file with which the field is associated.

fieldName This is the name of the field to be associated with the **Field4date** object.

Exceptions: **IOException**, **Error4field**, **Error4message**, **Error4unexpected**, **Error4usage**

```
import codebase.*;
import java.util.Date;
import java.io.IOException;
import java.net.UnknownHostException;

class Example23
{
    public static void main(String args[])
            throws Error4,IOException,UnknownHostException
    {
```

```
            Code4 client = new Code4();
            client.connect("", -1, "user1", "");
            Data4 db=new Data4(client);
            db.open("INFO");
            Field4double age=new Field4double(db,"AGE");
            Field4date bdate=new Field4date(db,"BIRTH_DATE");
            db.top();
            Date now=new Date();
            long milliAge;
            int dayAge;
            milliAge=now.getTime()-bdate.contents.getTime();
            dayAge=(int)(((milliAge/1000)/3600)/24);
            System.out.println(bdate.contents+" "+now);
            System.out.println("Age in days based on birth date: "+dayAge);
            System.out.println("Age based on age field "+age.contents);
            db.close();
    }
}
```

Field4date.isBlank()

Usage: public boolean isBlank( )

Description: This method checks if the **Field4date.contents** is equal to the blank date.

Returns: This method returns true if the **Field4date.contents** is set with a blank date and false if the contents are not blank.

Field4date.isOld()

Usage: public boolean isOld( )

Description: This method checks if the **Field4date.contents** is equal to the "old" date.

Returns: This method returns true if the **Field4date.contents** is set with the "old" date and false if the contents are not equal to the "old" date.

Field4date.isValid()

Usage: public boolean isValid( )

Description: This method checks whether the **Field4date.contents** value is valid. If the contents are equal to the blank date or the "old" date then the date is invalid.

Returns: This method returns false if the **Field4date.contents** is set with the "old" date or the blank date. Otherwise this method returns true.

Field4date.setBlank()

Usage: public void setBlank( )

Description: This method sets **Field4date.contents** value to the blank date.

Field4double Class

Use the **Field4double** class when trying to access a numeric, floating point or character field. Call the constructor **Field4double.Field4double()** to create the object and associate it to the particular field. When the data file is positioned to a valid record, the value of that field corresponding to the record is copied into the variable **Field4double.contents** where it can be retrieved or modified. **Data4.blank()** sets the **Field4double.contents** to zero and this value is saved to disk when the record is updated by **Data4.update()**.

Public Instance Variable

Field4double.contents

Usage: public Double contents

Description: This variable contains the contents of a field from the current record. To change the field value, modify **contents** directly. Any changes to this variable are not permanent. When the data file is positioned to a different record, the new field value is copied into this variable, so any changes to the previous record field value will be lost. Call **Data4.update()** to write any changes to the disk before the data file is repositioned, thus making the changes permanent.

See Also: **Data4.update()**

Public Instance Methods

Field4double.Field4double()

Usage: public Field4double(Data4 data, String fieldName)

Description: This method is a constructor for the object and it also associates the object with a particular field specified by *fieldName*.

Parameters:

data This is the reference to the data file with which the field is associated.

fieldName This is the name of the field to be associated with the **Field4double** object.

Exceptions: **IOException**, **Error4field**, **Error4message**, **Error4unexpected**, **Error4usage**

```
import codebase.*;
import java.util.Date;
import java.io.IOException;
import java.net.UnknownHostException;

class Example23
{
    public static void main(String args[])
            throws Error4,IOException,UnknownHostException
    {
            Code4 client = new Code4();
            client.connect("", -1, "user1", "");
            Data4 db=new Data4(client);
            db.open("INFO");
            Field4double age=new Field4double(db,"AGE");
            Field4date bdate=new Field4date(db,"BIRTH_DATE");
            db.top();
            Date now=new Date();
            long milliAge;
            int dayAge;
            milliAge=now.getTime()-bdate.contents.getTime();
            dayAge=(int)(((milliAge/1000)/3600)/24);
            System.out.println(bdate.contents+" "+now);
            System.out.println("Age in days based on birth date: "+dayAge);
            System.out.println("Age based on age field "+age.contents);
            db.close();
    }
}
```

Field4stringBuffer Class

Use the **Field4stringBuffer** class when trying to access a character, date, numeric, floating point, logical, memo, binary or general field. Call the constructor **Field4stringBuffer.Field4stringBuffer()** to create the object and associate it to the particular field. When the data file is positioned to a valid record, the value of that character field corresponding to that record is copied into the variable **Field4stringBuffer.contents** where it can be retrieved or modified. **Data4.update()** saves the field value to disk using the ASCII standard. **Data4.blank()** sets the **Field4stringBuffer.contents** to an empty StringBuffer (i.e. ""), which represents a blank value. When the blank value is saved, the field on the disk is filled with spaces.

Public Instance Variable

Field4stringBuffer.contents

Usage: public StringBuffer contents

Description: This variable contains the contents of a field from the current record. To change the field value, modify **contents** directly. Any changes to this variable are not permanent. When the data file is positioned to a different record, the new field value is copied into this variable, so any changes to the previous record field value will be lost. Call **Data4.update()** to write any changes to the disk before the data file is repositioned, thus making the changes permanent.

See Also: **Data4.update()**

Public Instance Methods

Field4stringBuffer.Field4stringBuffer()

Usage: public Field4stringBuffer(Data4 data, String fieldName)

Description: This method is a constructor for the object and it also associates the object with a particular field specified by *fieldName*.

Parameters:

data This is the reference to the data file with which the field is associated.

fieldName This is the name of the field to be associated with the **Field4stringBuffer** object.

Exceptions: **IOException**, **Error4field**, **Error4message**, **Error4unexpected**, **Error4usage**

Field4stringBufferUnicode Class

Use the **Field4stringBufferUnicode** class when trying to access a character, date, numeric, floating point, logical, memo, binary or general field. Call the constructor **Field4stringBufferUnicode.Field4stringBufferUnicode()** to create the object and associate it to the particular character field. When the data file is positioned to a valid record, the value of that character field corresponding to that record is copied into the variable **Field4stringBufferUnicode.contents** where it can be retrieved or modified. **Data4.update()** saves the field value to disk using the Unicode standard. **Data4.blank()** sets the **Field4stringBufferUnicode.contents** to an empty StringBuffer (i.e. ""), which represents a blank value. When the blank value is saved, the field on the disk is filled with Unicode spaces.

Public Instance Variable

Field4stringBufferUnicode.contents

Usage: public StringBuffer contents

Description: This variable contains the contents of a field from the current record. To change the field value, modify **contents** directly. Any changes to this variable are not permanent. When the data file is positioned to a different record, the new field value is copied into this variable, so any changes to the previous record field value will be lost. Call **Data4.update()** to write any changes to the disk before the data file is repositioned, thus making the changes permanent.

See Also: **Data4.update()**

Public Instance Methods

Field4stringBufferUnicode. Field4stringBufferUnicode()

Usage: public Field4stringBufferUnicode(Data4 data, String fieldName)

Description: This method is a constructor for the object and it also associates the object with a particular field specified by *fieldName*.

Parameters:

data This is the reference to the data file with which the field is associated.

fieldName This is the name of the field to be associated with the **Field4stringBufferUnicode** object.

Exceptions: **IOException**, **Error4field**, **Error4message**, **Error4unexpected**, **Error4usage**

Field4deleteFlag Class

The **Field4deleteFlag** stores the value of the deleted flag for the current record. The record is marked as deleted when the deleted flag is set to true. When a record is marked as deleted, it can still be accessed and the field values can be loaded into the field objects. The records are physically removed only when the data file is packed. The value of the deleted flag may be retrieved or modified directly through the **Field4deleteFlag.deleted** variable.

When **Data4.blank()** is called, **Field4deleteFlag.deleted** is set to false. When **Field4deleteFlag.deleted** is set to false and the record is updated, the deleted flag on the disk is set to a space. On the other hand, if **Field4deleteFlag.deleted** is true, then the deleted flag on the disk is set to '*'.

Public Instance Variable

Field4deleteFlag.deleted

Usage: public boolean deleted

Description: This variable contains the value of the deleted flag for the current record. Modify this variable directly to change the value of the flag. Any changes to this variable are not permanent. When the data file is positioned to a different record, the new flag value is copied into this variable, so any changes to the previous deleted flag will be lost. Call **Data4.update()** to write any changes to the disk before the data file is repositioned, thus making the changes permanent.

See Also: **Data4.update()**

Public Instance Methods

Field4deleteFlag.Field4deleteFlag()

Usage: public Field4deleteFlag(Data4 data)

Description: This method is a constructor for the **Field4deleteFlag** object.

Parameters:

data This is the **Data4** object associated with the current data file.

Exceptions: **IOException, Error4message**, **Error4unexpected, Error4usage**

Field4info Class

The **Field4info** class provides programmers with the ability to dynamically create a list of fields and their attributes used by **Data4.create()** to create data files.

Public Class Constants

All of the following constants are **public static final char** constants.

Class Constant	Value	Meaning
BINARY	'B'	This constant is used to specify a Binary field type.
CHARACTER	'C'	This constant is used to specify a Character field type.
DATE	'D'	This constant is used to specify a Date field type.
FLOAT	'F'	This constant is used to specify a Floating Point field type.
GENERAL	'G'	This constant is used to specify a General field type.
LOGICAL	'L'	This constant is used to specify a Logical field type.
NUMERIC	'N'	This constant is used to specify a Numeric field type.
MEMO	'M'	This constant is used to specify a Memo field type.

Public Instance Methods

Field4info.Field4info()

Usage: public Field4info( )

Description: This method is a constructor for the **Field4info** object.

Field4info.add()

Usage: public void add(String fieldName, char type, int length, int decimals)
public void add (String fieldName, char type, int length, int decimals, boolean allowNull)

Description: This method adds a field and its attributes to a list, which can then be used by **Data4.create()** to create a new data file with these specified fields.

Parameters:

fieldName This is the name of the field. Each field name should be unique to the data file. A field name is up to ten alphanumeric or underscore characters - except for the first character which must be a letter. Any characters in the field name over ten are ignored.

type This is the type of the field. Fields must be one of the following types: Character, Date, Numeric, Floating Point, Logical, Memo, Binary or General. Use the **Field4info** class constants to specify the field type.

length This is the length of the field. Date, Memo and Logical fields, have pre-determined lengths. These pre-determined lengths are used regardless of the lengths specified by this parameter.

decimals This is the number of decimals in Numeric and Floating Point fields.

allowNull This parameter is used only under Visual FoxPro to support null fields. When this parameter is set to true, the field can store nulls. If Visual FoxPro is not being used, this parameter is ignored. See the introduction to the "Field Manipulation" chapter for more information..

Specifics on the types of fields and their limitations are listed in the following table:

Type	Field4info Class Constant	Length	Decimals	Information
Binary	BINARY	Set to 10. Actual data is in a separate file.	0	Binary fields are handled in the same way as memo fields. It stores binary information. The amount of information is dependent upon the size of an **(unsigned int)**. This field type is only supported by the dBASE IV file format.
Character	CHARACTER	1 to 65533 1 to 254 to keep dBASE and FoxPro file compatibility.	0	Character fields can store any type of information including binary.
Date	DATE	8	0	Date Fields store date information only. It is stored in CCYYMMDD format.
Floating Point	FLOAT	The length depends on the format **CLIPPER** 1 to 19 **FOXPRO** 1 to 20 **dBASE IV** 1 to 20	The number of decimals depends on the format **CLIPPER** is the minimum of (len - 2) and 15 **FOXPRO**(len - 1) **dBASE IV**(len - 2)	CodeBase treats this field like it was a Numeric field. This field is useful for compatibility with dBASE and FoxPro, which treat Floating point and Numeric fields differently. Use this field to store values that will be used in floating point calculations.

General	GENERAL	Set to 10. Actual data is in a separate file.	0	General fields are handled in the same way as memo fields. It stores OLEs. The amount of information is dependent upon the size of an **(unsigned int)**. This field type is only supported by the FoxPro file format.
Logical	LOGICAL	1	0	Logical fields store either true or false. The values that represent true are 'T', 't',' Y', or 'y'. The values that represent false are the characters 'F', 'f', N', or 'n'.
Memo	MEMO	Set to 10. Actual data is in a separate file.	0	Memo fields store the same type of information as the Character type. The amount of information is dependent upon the size of an **(unsigned int)**.
Numeric	NUMERIC	The length depends on the format **CLIPPER** 1 to 19 **FOXPRO** 1 to 20 **dBASE IV** 1 to 20	The number of decimals depends on the format **CLIPPER** is the minimum of (len - 2) and 15 **FOXPRO**(len - 1) **dBASE IV**(len - 2)	Numeric fields store numerical information. It is stored internally as a string of digits. This field is useful for compatibility with dBASE and FoxPro, which treat Floating point and Numeric fields differently. Use this field to store values that will NOT be used in floating point calculations.

See Also: **Data4.create()**

WARNING

The Binary and General field types are NOT compatible with some versions of dBASE, FoxPro, Clipper and other dBASE compatible products. Only use Binary and General fields with products that support these field types.

```
import codebase.*;
import java.util.Date;
import java.io.IOException;
import java.net.UnknownHostException;
class Example25
{
    public static void main(String args[])
            throws Error4,IOException,UnknownHostException
    {
            Code4 client = new Code4();
            client.connect(null, -1, "user1", "");
            client.safety(false);
            Data4 db=new Data4(client);

            Field4info fi=new Field4info();
            fi.add("boolean",'L',1,0);
            fi.add("string",'C',10,0);
            fi.add("date",'D',8,0);
            fi.add("double",'N',8,0);
            fi.add("unicode",'C',10,0);

            db.create("NEWDATA2",fi);
            db.close();
    }
}
```

Status4 Class

The **Status4** class contains public class constants that are used as return values for the **Data4** reposition methods, **Data4.bottom(), Data4.go(), Data4.positionSet, Data4.seek(), Data4.seekUnicode(), Data4.skip()** and **Data4.top().**

Status4 Class Constants

All of the **STATUS4** constants are **public static final short** constants.

Class Constant	Meaning
AFTER	This value is returned when **Data4.seek()** or **Data4.seekUnicode()** could not find the requested search key and the data file is positioned to the location after the search key, had it been found.
BOF	This value is returned when there is a beginning of file condition for the data file.
EOF	This value is returned when there is an end of file condition for the data file.
SUCCESS	This value is returned when the method has successfully completed.

Tag4info Class

The **Tag4info** class gives programmers the ability to dynamically create a list of tags, which are used by **Data4.create()** and **Data4.createIndex()** to create index files.

Class Constants

All of the constants are **public static final byte** constants.

Class Constant	Meaning
DUPLICATE	This constant is used by **Tag4info.add()** and it means that the tag is not unique and duplicate keys are allowed in the tag.
UNIQUE_CONTINUE	This constant can be used by **Tag4info.add()** and **Code4.defaultUnique()**. This constant means that duplicate keys are discarded for unique tags.
UNIQUE	This constant can be used by **Tag4info.add()** and **Code4.defaultUnique()**. This constant means that when a duplicate key is encountered for a unique tag, an **Error4unique** exception is thrown.
CANDIDATE	This constant is used by **Tag4info.add()** and it means that the tag is unique. This constant only has an effect when creating Visual FoxPro files. **Code4.defaultUnique()** has no effect when index files are opened and if a duplicate key is encountered an **Error4unique** exception is thrown.

Public Instance Methods

Tag4info.Tag4info ()

Usage: public Tag4info( )

Description: **Tag4info.Tag4info()** constructs a **Tag4info** object and initializes the tag list.

See Also: **Tag4info.add()**

```
import codebase.*;
import java.io.IOException;
import java.net.UnknownHostException;

class Example13
{

    public static void main(String args[])
            throws Error4,IOException,UnknownHostException
    {
        Code4 client = new Code4();
        client.connect(null, -1, "user1", "");
        client.safety(false);
        client.accessMode(Code4.DENY_RW);
        Data4 newDataFile = new Data4(client);

        Field4info fieldInfo = new Field4info();
        fieldInfo.add("LastName", Field4info.CHARACTER, 10, 0);
        fieldInfo.add("Age", Field4info.NUMERIC, 3, 0);

        //create the new data file;
        newDataFile.create("NEWDATAFILE", fieldInfo);
        System.out.println("successfully created NEWDATAFILE.DBF");

        //try to create a production index
        Tag4info tags = new Tag4info();
        tags.add("NameTag", "LastName", null, Tag4info.DUPLICATE, false);
        newDataFile.createIndex( null, tags);
        System.out.println("successfully created NEWDATAFILE production index");

        //try to create index file
        Tag4info newTags = new Tag4info();
        newTags.add("AgeTag", "Age", null, Tag4info.DUPLICATE, false);
        newDataFile.createIndex("AgeIndexFile", newTags);
        System.out.println("successfully created AgeIndexFile index file");
        newDataFile.close();
    }
}
```

Tag4info.add()

Usage: public void add(String name,
String expr,
String filter,
byte unique,
boolean descending)

Description: **Tag4info.add()** adds a tag to the list of tags. All parameters must be defined for every tag, except *filter*, which may set to null.

Parameters:

name This is a string containing the name of the tag. The name may be composed of letters, numbers and underscores. Only letters and underscores are permitted as the first character of the name. This parameter cannot be null.

When using FoxPro or **.MDX** formats, the tag name must be unique to the data file and have a length of ten characters or less.

If you are using the **.NTX** index formats, then this name includes the index file name with a path. In this case, the index file name within the path is limited to eight characters or less, excluding the extension.

expr This string is the tag's index expression. This expression determines the sorting order of the tag. Refer to "Appendix A: dBASE Expressions" for more information on possible key expressions. This is often just a field name. This member cannot be null.

filter This string is the logical dBASE expression used to determine which records are added to the tag. If this filter expression evaluates to true for any given data file record, a key for the record is included in the tag. If a null string is specified, keys for all data file records are included in the tag.

unique This value is used to determine how duplicate keys are handled. *unique* may have one of the following values**: Tag4info.DUPLICATE, Tag4info.UNIQUE, Tag4info.UNIQUE_CONTINUE** or **Tag4info.CANDIDATE**.

- **Tag4info.DUPLICATE** Duplicate keys are allowed.
- **Tag4info.UNIQUE_CONTINUE** When a duplicate key is encountered, it is not added to the tag. Consequently, there may not be a tag entry for a particular record.
- **Tag4info.UNIQUE** Generate an exception **Error4unique** if a duplicate key is encountered.
- **Tag4info.CANDIDATE** This constant only has an effect on Visual FoxPro tags. An **Error4unique** exception is thrown if a duplicate key is encountered.

CodeBase uses the dBASE file format, which only saves to disk a true/false unique flag that indicates whether a tag is unique or non-unique. No information on how to respond to a duplicate key is saved for unique tags.

When an index file containing unique tags is opened, the duplicate keys are treated according to the **Code4.defaultUnique()** setting, which can be set with **Tag4info.UNIQUE** or **Tag4info.UNIQUE_CONTINUE**.

Code4.Code4() initializes **Code4.defaultUnique()** to **Tag4info.UNIQUE_CONTINUE** by default. This setting mimics the dBASE response to duplicate keys. If a duplicate key is encountered for a unique key tag, dBASE responds by ensuring there is no corresponding key for the record. Any duplicate keys are ignored and are not saved in the tag. Consequently, there may be records in the data file that do not have a corresponding tag entry.

An exception will be thrown when a duplicate key is encountered by setting **Code4.defaultUnique()** with **Tag4info.UNIQUE** before the index file is opened.

Note that **Code4.defaultUnique()** only applies to unique key tags. For non-unique key tags, the internal unique flag is initialized to false, meaning duplicate keys can exist and **Code4.defaultUnique()** has no effect.

Visual FoxPro files have an additional setting for unique tags called **Tag4info.CANDIDATE**. Tags created using this setting throw an exception when a duplicate key is encountered. **Code4.defaultUnique** has no effect on unique tags created with the **Tag4info.CANADIDATE** setting.

If **Tag4info.CANDIDATE** is used to create unique tags for non-Visual FoxPro index files, the unique setting is treated as though it was **Tag4info.UNIQUE**. In this case, the setting of **Code4.defaultUnique()** does have an effect when opening index files.

descending This logical flag determines whether the index is stored in ascending or descending order. If this value is true, the index keys are stored in ascending order. Descending order is specified when this parameter is set to false.

See Also: **Data4.create(), Data4.createIndex(), Code4.defaultUnique()**

Appendix A: dBASE Expressions

In CodeBase, a dBASE expression is represented as a character string and is evaluated using the expression evaluation functions. dBASE expressions are used to define the keys and filters of an index file. They can be useful for other purposes such as interactive queries.

WARNING

The dBASE functions listed in this appendix are not Java methods to be called directly from Java programs. They are dBASE functions that are recognized by the CodeBase expression evaluation functions. In the same manner, Java methods and variables cannot appear in a dBASE expression.

General dBASE Expression Information

All dBASE expressions return a value of a specific type. This type can be Numeric, Character, Date or Logical. A common form of a dBASE expression is the name of a field. In this case, the type of the dBASE expression is the type of the field. Field names, constants, and functions may all be used as parts of a dBASE expression. These parts can be combined with other functions or with operators. Example dBASE Expression: "FIELD_NAME"

Note

In this manual all dBASE expressions are contained in double quotes (" "). The quotes are not considered part of the dBASE expression. Any double quotes that are contained within the dBASE expression will be denoted as ' \" '.

Field Name Qualifier

It is possible to qualify a field name in a dBASE expression by specifying the data file.

Example dBASE Expression: "DBALIAS->FLD_NAME"

Observe that the first part, the qualifier, specifies a data file name. Then there is the "->" followed by the field name.

dBASE Expression Constants

dBASE Expressions can consist of a Numeric, Character or Logical constant. However, dBASE expressions that are constants are usually not very useful. Constants are usually used within a more complicated dBASE expression.

A Numeric constant is a number. For example, "5", "7.3", and "18" are all dBASE expressions containing Numeric constants.

Character constants are letters with quote marks around them. " 'This is data' ", " 'John Smith' ", and " \"John Smith\" " are all examples of dBASE expressions containing Character constants. If you wish to specify a character constant with a single quote or a double quote contained inside it, use the other type of quote to mark the Character constant. For example," \"Man's\" " and " ' \"Ok\" ' " are both legitimate Character constants. Unless otherwise specified, all dBASE Character constants in this manual are denoted by single quote characters.

Constants .TRUE. and .FALSE. are the only legitimate Logical constants. Constants .T. and .F. are legitimate abbreviations.

dBASE Expression Operators

Operators like '+' , ' * ', or '<' are used to manipulate constants and fields. For example, "3+8" is an example of a dBASE expression in which the Add operator acts on two numeric constants to return the numeric value "11". The values an operator acts on must have a type appropriate for the operator. For example, the divide '/' operator acts on two numeric values.

Precedence

Operators have a precedence that specifies operator evaluation order. The precedence of each operator is specified in the following tables that describe the various operators. The higher the precedence, the earlier the operation will be performed. For example, 'divide' has a precedence of 6 and 'plus' has a precedence of 5 which means 'divide' is evaluated before 'plus'. Consequently, "1+4/2" is "3". Evaluation order can be made explicit by using brackets. For example, "1+2 * 3" returns "7" and "(1+2) * 3" returns "9".

Numeric Operators

The numeric operators all operate on Numeric values.

Operator Name	Symbol	Precedence
Add	+	5
Subtract	-	5
Multiply	*	6
Divide	/	6
Exponent	** or ^	7

Character Operators

There are two character operators, named "Concatenate I" and "Concatenate II", which combine two character values into one. They are distinguished from the Add and Subtract operators by the types of the values they operate on.

Operator Name	Symbol	Precedence
Concatenate I	+	5
Concatenate II	-	5

Examples:

" 'John ' + 'Smith' " becomes " 'John Smith' "

" 'ABC' + 'DEF' " becomes " 'ABCDEF' "

Concatenate II is slightly different in that any spaces at the end of the first Character value are moved to the end of the result.

" 'John'-'Smith ' " becomes " 'JohnSmith ' "

" 'ABC' - 'DEF' " becomes " 'ABCDEF' "

" 'A ' - 'D ' " becomes " 'AD ' "

Relational Operators

Relational Operators are operators that return a Logical result (which is either true or false). All operators, except Contain, operate on Numeric, Character or Date values. Contain operates on two character values and returns true if the first is contained in the second.

Operator Name	Symbol	Precedence
Equal To	=	4
Not Equal To	<> or #	4
Less Than	<	4
Greater Than	>	4
Less Than or Equal To	< =	4
Greater Than or Equal To	> =	4
Contain	$	4

Examples: " 'CD' $ 'ABCD' " returns ".T."

" 8<7 " returns ".F."

Logical Operators

Logical Operators return a Logical Result and operate on two Logical values.

Operator Name	Symbol	Precedence
Not	.NOT.	3
And	.AND.	2
Or	.OR.	1

Examples " .NOT. .T. " returns ".F."

" .T. .AND. .F." returns ".F."

dBASE Expression Functions

A function can be used as a dBASE expression or as part of an dBASE expression. Functions return a value like operators, constants and fields. Functions always have a function name and are followed by a left and right bracket. Values (parameters) may be inside the brackets.

Function List

ALLTRIM(CHAR_VALUE)

This function trims all of the blanks from both the beginning and the end of the expression.

ASCEND(VALUE)

This function is not supported by dBASE, FoxPro or Clipper.

ASCEND() accepts all types of parameters, except complex numeric expressions. ASCEND() converts all types into a Character type in ascending order. In the case of numeric types, the conversion is done so that the sorting will work correctly even if negative values are present.

CHR(INTEGER_VALUE)

This function returns the character whose numeric ASCII code is identical to the given integer. The integer must be between 0 and 255.

Example: CHR(65) returns “A”.

CTOD(CHAR_VALUE)

The character to date function converts a character value into a date value:

eg. " CTOD("11/30/88") "

The character representation is always in the format specified by "MM/DD/YY".

DATE()

The system date is returned.

DAY(DATE_VALUE)

Returns the day of the date parameter as a numeric value from "1" to "31".

eg. "DAY(DATE())"

Returns "30" if it is the thirtieth of the month.

DESCEND(VALUE)

This function is not supported by dBASE or FoxPro. DESCEND() is compatible with Clipper, only if the parameter is a Character type.

DESCEND() accepts any type of parameter, except complex numeric expressions. DESCEND() converts all types into a character type in descending order.

For example, the following expression would produce a reverse order sort on the field ORD_DATE followed by normal sub-sort on COMPANY.

eg. DESCEND(ORD_DATE) + COMPANY

See also ASCEND().

DELETED()

Returns .TRUE. if the current record is marked for deletion.

DTOC(DATE_VALUE)

DTOC(DATE_VALUE, 1)

The date to character function converts a date value into a character value. The format of the resulting character value is specified by "MM/DD/YY".

eg. " DTOC(DATE()) "

Returns the character value "05/30/87" if the date is May 30, 1987.

If the optional second argument is used, the result will be identical to the dBASE expression function *DTOS*.

For example, DTOC(DATE(), 1) will return "19940731" if the date is July 31, 1994.

DTOS(DATE_VALUE)

The date to string function converts a date value into a character value. The format of the resulting character value is "CCYYMMDD".

e.g. ." DTOS(DATE()) "

Returns the character value "19870530" if the date is May 30, 1987.

IIF(LOG_VALUE, TRUE_RESULT, FALSE_RESULT)

If 'Log_Value' is .TRUE. then IIF returns the 'True_Result' value. Otherwise, IIF returns the 'False_Result' value. Both True_Result and False_Result must be the same length and type. Otherwise, an error results.

eg. "IIF(VALUE << 0, "Less than zero ", "Greater than zero")"

e.g. ."IIF(NAME = "John", "The name is John", "Not John ")"

LEFT(CHAR_VALUE, NUM_CHARS)

This function returns a specified number of characters from a character expression, beginning at the first character on the left.

eg. "LEFT('SEQUITER', 3)" returns "SEQ".

The same result could be achieved with "SUBSTR('SEQUITER', 1, 3)".

LTRIM(CHAR_VALUE)

This function trims any blanks from the beginning of the expression.

MONTH(DATE_VALUE)

Returns the month of the date parameter as a Numeric.

eg. " MONTH(DT_FIELD) "

Returns 12 if the date field's month is December.

PAGENO()

When using a report module, this function returns the current report page number.

RECCOUNT()

The record count function returns the total number of records in the database:

eg. " RECCOUNT() "

Returns 10 if there are ten records in the database.

RECNO()

The record number function returns the record number of the current record.

STOD(CHAR_VALUE)

The string to date function converts a character value into a date value:

eg. " STOD("19881130") "

The character representation is in the format "CCYYMMDD".

STR(NUMBER, LENGTH, DECIMALS)

The string function converts a numeric value into a character value. "Length" is the number of characters in the new string, including the decimal point. "Decimals" is the number of decimal places desired. If the number is too big for the allotted space, *'s will be returned.

eg. " STR(5.7, 4, 2) " returns " '5.70' "

The number 5.7 is converted to a string of length 4. In addition, there will be 2 decimal places.

eg. " STR(5.7, 3, 2) " returns " '***' "

The number 5.7 cannot fit into a string of length 3 if it is to have 2 decimal places. Consequently, *'s are filled in.

SUBSTR(CHAR_VALUE, START_POSITION, NUM_CHARS)

A substring of the Character value is returned. The substring will be 'Num_Chars' long, and will start at the 'Start_Position' character of 'Char_Value'.

eg. " SUBSTR("ABCDE", 2, 3)" returns " 'BCD' "

eg. "SUBSTR("Mr. Smith", 5, 1)" returns " 'S' "

TIME()

The time function returns the system time as a character representation. It uses the following format: HH:MM:SS.

eg. " TIME() " returns " 12:00:00 " if it is noon.

eg. " TIME() " returns " 13:30:00 " if it is one thirty PM.

TRIM(CHAR_VALUE) This function trims any blanks off the end of the expression.
UPPER(CHAR_VALUE) A character string is converted to uppercase and the result is returned.
VAL(CHAR_VALUE) The value function converts a character value to a numeric value. eg. “VAL('10')” returns "10". eg. “VAL('-8.7')” returns "-8.7".
YEAR(DATE_VALUE) Returns the year of the date parameter as a numeric: eg. "YEAR(STOD('19920830')) " returns " 1992 "

Appendix B: CodeBase Limits

Following are the maximums of CodeBase:

Description	Limit
Block Size	32768 (32K)
Data File Size	1,073,741,824 Bytes
Field Width	254 for dBASE compatibility, 32767 otherwise.
Floating Point Field Width	19
Memo Entry Size	(maximum value of an unsigned integer) minus (overhead) The range of the integer depends on how many bytes are used to store an unsigned integer, which in turn depends on the system. The overhead may vary from o/s to o/s, but the overhead should never exceed 100 bytes. Therefore, if the system uses a 2 byte unsigned integer, then the memo entry size is: 65,536 - 100 = 65, 436 bytes (approx. 64K) If the system uses a 4 byte unsigned integer, then the memo entry size is: 4,294,967,296 - 100 = 4,294,967,196 bytes (approx. 4G)
Number of Fields	128 for dBASE compatibility. 1022 for Clipper compatibility.
Number of Open Files	The number of open files is constrained only by the compiler and the operating environment.
Number of Tags per Index	Unlimited FoxPro 47 dBASE IV 1 Clipper
Numeric Field Width	19
Record Width	65500 (64K)

Index

.

.CDX, 32
.CGP, 32
.DBF, 23, 27, 32
.FPT, 32
.MDX, 32, 80
.NTX, 80

A

ALLTRIM(), 86
ASCEND(), 86, 87

B

Binary Fields, 54

C

Character Fields, 53
CHR(), 86
Clipper, 32, 53, 54, 74, 75, 80, 86, 87, 91
Closing
  current data file, 26
Code4 Class
  constants
    AUTO_ALL, 19
    AUTO_OFF, 19
    DENY_NONE, 7
    DENY_RW, 7
    DENY_WRITE, 7
  methods
    accessMode, 7
    Code4, 7
    connect, 9
    defaultUnique, 9
    lock, 11
    lockClear, 12
    readLock, 13
    readOnly, 15
    safety, 16
    unlock, 18
    unlockAuto, 18
CTOD(), 86

D

Data Files
  auto opening of index files, 23, 32
  beginning of file (bof) condition, 21
  bottom, moving to, 26
  closing, 26
  creating, 27
  end of file (eof) condition, 21
  moving between records, 26, 29, 36, 40, 42, 43, 45
  opening, 23, 32
  overwriting, 16
  packing, 34
  position by percentage, 36
  record count, 37
  record number, 38
  reindexing, 39
  top, moving to, 45
  updating, 26, 45
Data4 Class, 21
  methods
    append, 24
    blank, 25
    bottom, 26
    close, 26
    create, 27
    createIndex, 28
    Data4, 23
    go, 29
    lockAdd, 30
    lockAddAll, 30
    lockAddAppend, 31
    lockAddFile, 31
    open, 32
    openIndex, 34
    pack, 34
    position, 35
    positionSet, 36
    recCount, 37
    recNo, 38

reindex, 39
seek, 40
seekUnicode, 42
select, 43
skip, 43
top, 45
update, 45
Date Fields, 53
DATE(), 86, 87
DAY(), 86
dBASE Expressions
constants, 83
dBASE IV, 32, 53, 74, 80, 91
dBASE Operators, 83
character operators, 84
logical operators, 85
numeric operators, 84
relational operators, 84
DELETED(), 87
DESCEND(), 87
DTOC(), 87
DTOS(), 87

E

Error4 Class, 48
Error4entry Class, 48
Error4field Class, 48
Error4file Class, 48
Error4locked Class, 48
Error4message Class, 48
Error4tagName Class, 48
Error4unexpected Class, 48
method
getCode, 49
Error4unique Class, 48
Error4usage Class, 48
Exception Classes, 47
Exclusive Access
default setting, 7

F

Field Manipulation Classes, 50
Field4 Class, 56
methods
decimals, 56
length, 56
type, 57
Field4boolean Class, 58
method
Field4boolean, 59
variable
contents, 59
Field4date Class, 60
blank date, 60
invalid date, 60
methods
Field4date, 61
isBlank, 62
isOld, 62
isValid, 62
setBlank, 63
variable
contents, 61
Field4deleteFlag Class, 70
method
Field4deleteFlag, 71
variable
deleted, 70
Field4double Class, 64
method
Field4double, 65
variable
contents, 64
Field4info Class, 72
constants
BINARY, 72
CHARACTER, 72
DATE, 72
FLOAT, 72
GENERAL, 72
LOGICAL, 72
MEMO, 72
NUMERIC, 72
methods
add, 73
Field4info, 72
Field4stringBuffer Class, 66
method
Field4stringBuffer, 67
variable
contents, 66
Field4stringBufferUnicode Class, 68
method
Field4stringBufferUnicode, 69

variable
contents, 68
Fields
expressions, using within, 82
supporting nulls, 51
type, 53
type, specifying, 73
Floating Point Fields, 53
FoxPro, 32, 53, 54, 74, 75, 80, 86, 87, 91

G

General Fields, 54
Group Files
opening, 32

I

IIF(), 88
Index Files
automatically opening, 23, 32
closing, 26
creating production indexes, 27
opening, 32, 34
overwriting, 16
production indexes, 32
reindexing, 39
updating, 45

L

LEFT(), 88
Locking
append bytes, 31
automatic record locking, 13
automatic unlocking, 18
data files, 30, 31
index files, 30
memo files, 30
multiple records, 30
queues, adding locks to, 30, 31
queues, clearing the, 12
queues, locking the, 11
queues, unlocking, 18
records, 30
unlocking all, 18
Logical Fields, 53
LTRIM(), 88

M

Memo Fields, 53
Memo Files
closing, 26
updating, 45
MONTH(), 88

N

Numeric Fields, 53

P

PAGENO(), 88

R

Read Only Mode
opening files in, 15
RECCOUNT(), 88
RECNO(), 89
Record Number, 38
Records
adding, 24
blanking, 25
deletion flag, 34
number of, 37
physically removing, 34
position in file, 35
skipping, 43
updating, 45

S

Scroll Bars
determining position, 35, 36
Seeking
binary information, 40, 42
character data, 40, 42
dates, 40
numbers, 40
partial matches, 40, 42
performing seeks, 40, 42
Unicode character data, 42
Status4 Class, 77
constants
AFTER, 77

BOF, 77
EOF, 77
SUCCESS, 77
STOD(), 89, 90
STR(), 89
SUBSTR(), 88, 89

T

Tag4info Class, 78
constants
CANDIDATE, 78
DUPLICATE, 78
UNIQUE, 78
UNIQUE_CONTINUE, 78
methods
add, 79
Tag4info, 78
Tags
default unique action, 9
selecting, 43
unique tag violations, 9, 78, 80
unique tags, 78, 80
TIME(), 89
TRIM(), 90

U

UPPER(), 90

V

VAL(), 90

Y

YEAR(), 90