12 Scripting

In this chapter:

12 Scripting

Scripting languages allow you to control Windows and Mac OS applications with program-like scripts.You can write scripts to perform tasks as simple as opening and printing a form, or as complex as controlling a sophisticated workflow process.

This chapter provides an overview of Informed's scripting capabilities. You'll learn how to use Informed Designer's Scripts command to attach scripts to templates so that they can be configured to run when the Informed Filler user invokes certain actions.

This chapter includes a thorough reference to Informed's implementation of the JavaScript scripting language. The reference material includes a description of each Informed JavaScript object as well as examples and descriptions of the terminology to use when writing scripts for Informed Filler. For information about Informed's AppleScript implementation, please refer to the "Using AppleScript" topic in Informed Designer's on-line help system or the on-line document "AScript.PDF" on the Informed CD-ROM.

Overview

A single script can automate a task that normally requires several steps. For example, you might write a script that would find and print all invoices that exceed five hundred dollars. A different script could create a new purchase order form and fill it in with information from one or more purchase requisition forms. For the Informed Filler user, performing such tasks becomes as simple as selecting a script

Informed's scripting features rely on Informed scripting plug-ins. By using plug-ins, Shana Corporation can easily support new scripting languages simply by implementing new scripting plug-ins. At the time this documentation was prepared, Informed Designer included plug-ins for the following scripting languages:

- JavaScript
- AppleScript

Informed's JavaScript support gives you the ability to create powerful cross-platform scripts that will operate equally well on both Windows and MacOS computers. Informed's AppleScript features are available only on MacOS compatible computers.

Informed Designer can store scripts in form documents. Whenever you copy a form document to another place, or mail a form to another person, the scripts remain part of the form. Applications that can store scripts, such as Informed Designer and Informed Filler, are often called *attachable* applications. This is because scripts can be *attached* to particular actions in the application. When the user performs an action, the application triggers a script.

You configure forms with Informed Designer so that Informed Filler invokes scripts when the user performs certain actions. You can attach scripts to the following actions:

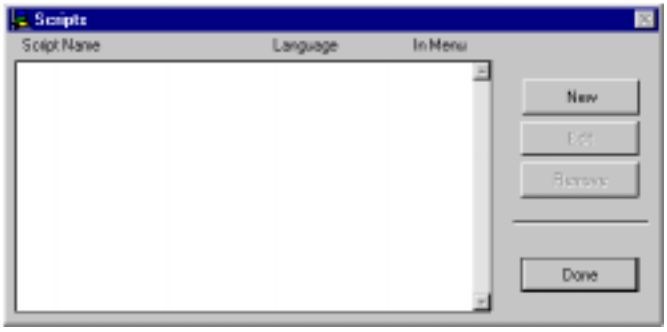
- selecting a menu item
- clicking a button
- typing a value in a lookup cell
- submitting a form

Before configuring an action, however, you must first enter the script using Informed Designer's Scripts command. See the following section for instructions on how to enter and edit scripts.

Entering and Editing Scripts

The procedure for entering a script is the same on both Windows and Mac OS compatible computers. You use Informed Designer's Scripts command to add, remove, and edit scripts.

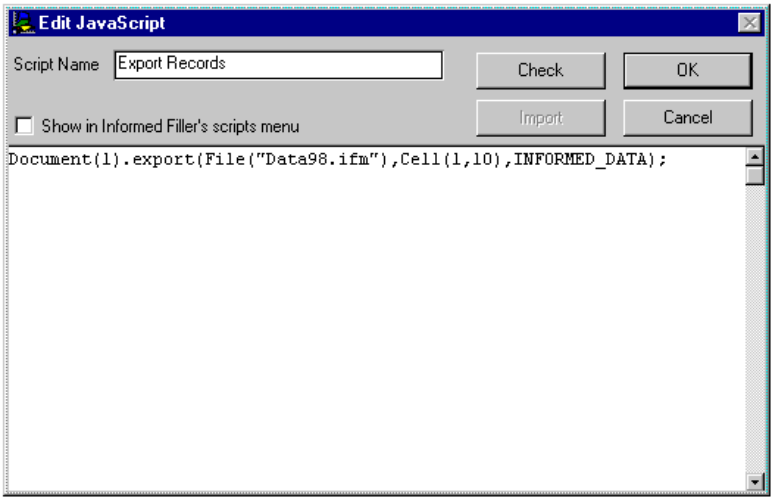
- Choose **Scripts** from the Configure submenu under the Form menu. The Scripts dialog appears.



If any scripts are currently attached to the form, their names will appear in the scrolling list in the order that they were created.

To add a new script to your form:

- Click 'New.' The Edit Script dialog box appears.

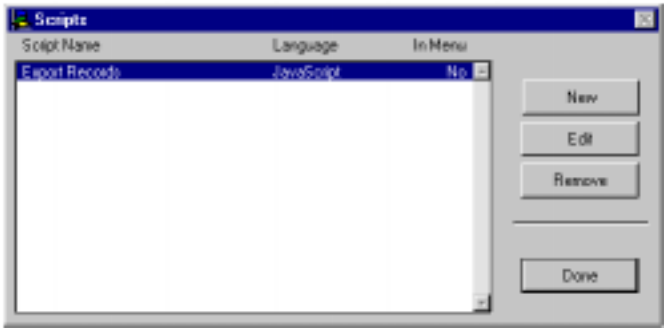


- Type the name of the new script in the ‘Script Name’ text box. This is the name that you’ll see when you configure an action to invoke a script.
- Enter a script by typing in the text box, or click the ‘Import’ button to import a script from another file.

In version 2.5 (or later) of Informed, a script that’s attached to a form does not automatically appear in Informed Filler’s Scripts menu. If you want a script to appear in Informed Filler’s Scripts menu:

- Click the ‘Show in Informed Filler’s Scripts menu’ checkbox.

- Check for errors by clicking the ‘Check’ button. If an error is detected, you’ll see a message describing the error. If there are no errors, the script will display properly formatted.
- Click ‘OK’ on the Edit Script dialog box. Informed Designer will store the script with the form and display it in the scrolling list on the Scripts dialog.



To edit an existing script:

- Select the script name in the Scripts dialog scrolling list, then click ‘Edit.’
- Make the appropriate changes on the Edit Scripts dialog.

To remove a script:

- Select the script name in the Scripts dialog scrolling list, then click ‘Remove.’

Informed JavaScript Implementation

This section provides an overview of Informed’s JavaScript implementation. It is assumed that you already understand the basics of JavaScript and are familiar with the Informed Designer and Informed Filler applications.

Reference Objects

Using JavaScript to communicate with Informed Filler can be thought of as a conversation with various objects in the Informed Filler application. To initiate this ‘conversation’ you need to create reference objects. A reference object is a JavaScript object that refers to one or more elements of the Informed user interface. For example, a Record object refers to one or more records in Informed Filler and a Cell object refers to one or more cells.

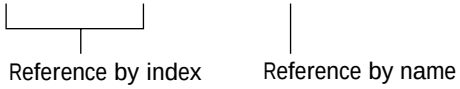
You use JavaScript’s new operator to construct reference objects as shown below:

```
//create a Document object
theDoc = new Document(2);

//create a new Menu object
theMenu = new Menu("File");
```

Reference objects can refer to their corresponding Informed elements in a number of different ways. For example, a Cell object can refer to one or more cells in Informed by name, index, id, absolute position, relative position, range, or test.

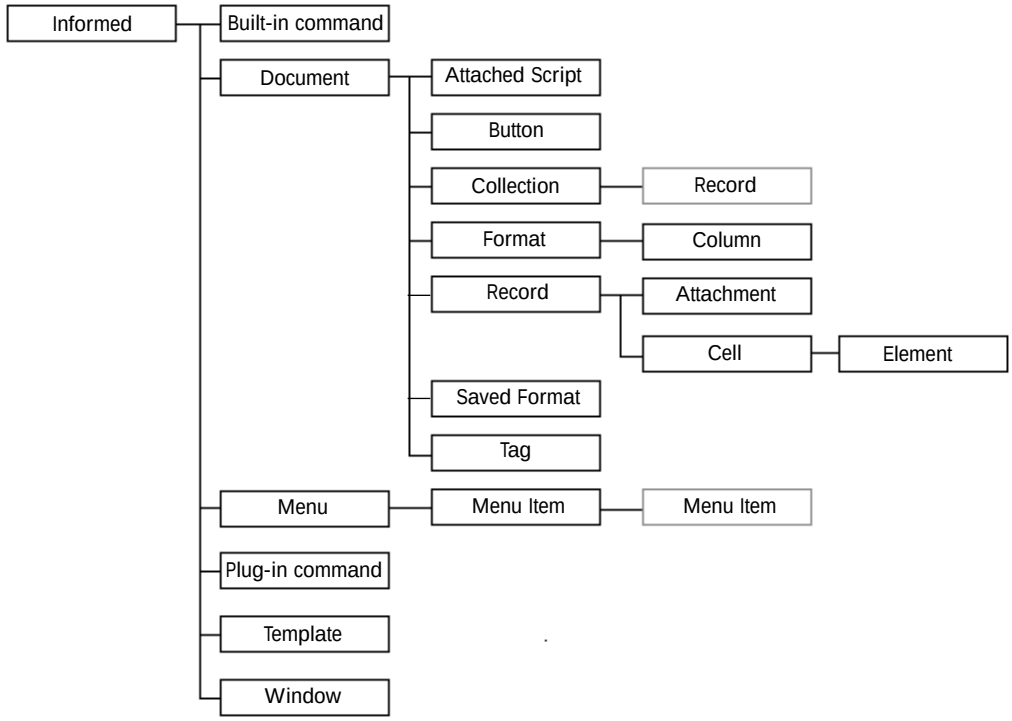
```
//refers to the cell named "Cost" in the third record of the second document
Document(2).Record(3).Cell("Cost")
```



For detailed information about reference objects, see “Reference Object Descriptions” and “Reference Object Types” later in this chapter.

Containment

Reference objects are organized hierarchically so that some objects are elements of others. For example, an Attachment object is an element of a Record object, and a Record object is an element of a Document object. This organization is referred to as *containment*. One object is *contained* by another or one object is the *container* of another. The object containment hierarchy in Informed is shown in the following diagram.



Syntax Shortcuts

To refer to a specific object, you would normally be required to specify its complete containment. The example below specifies the complete containment for a cell called “Age” in the first record of a document called “Contacts”:

```
theDoc = new Document("Contacts", Informed);
theRec = new Record(1, theDoc);
theCell = new Cell("Age", theRec);
```

Complete containment specifications can be quite long, and typing them would quickly become tiresome. Fortunately, Informed provides syntax shortcuts which can simplify your scripts. Using a syntax shortcut, the above containment specification can be condensed to a single line:

```
theCell = Informed.Document("Contacts").Record(1).Cell("Age");
```

Using Variables

Another way to simplify your scripts is to use variables to store frequently used containment. The example below uses the same containment in each line of the script.

```
Informed.Document(2).currentRecord.Cell("Name").value = "Brent Taylor";
Informed.Document(2).currentRecord.Cell("Age").value = 37;
Informed.Document(2).currentRecord.Cell("Sex").value = "Male";
```

By using a variable, the above code can be simplified as follows:

```
var theRecord = Informed.Document(2).currentRecord;
theRecord.Cell("Name").value = "Brent Taylor";
theRecord.Cell("Age").value = 37;
theRecord.Cell("Sex").value = "Male";
```

Implied Containment

Another way to simplify your scripts is to use “implied containment.” This means that certain objects can be left out of the object specification and appropriate defaults will be provided. The following objects are optional in a containment specification:

- Informed
- Document
- Record

The example below shows a complete containment specification:

```
// Refers to the fourth cell of the third record of the second document
Informed.Document(2).Record(3).Cell(4)
```

The above example could be shortened by leaving out the Informed object. Therefore, the Informed object is “implied.”

```
// Refers to the fourth cell of the third record of the second document
Document(2).Record(3).Cell(4)
```

Rules for Implied Containment

If you use implied containment in your scripts, you must be aware of the following rules:

- If no document is specified, then the implied document is the current document:

  ```
  // refers to the fourth cell of the third record of the current document
  Informed.Record(3).Cell(4)
  ```

- If no record is specified, then all records of the specified container are implied:

  ```
  // refers to the fourth cell of every record of the second document
  Document(2).Cell(4)
  ```

- If no container is specified (or if the specified container is Informed) then the current record of the current document is implied:

```
// refers to the fourth cell of the current record of the current document
Informed.Cell(4)
```

Reference Object Types

A reference object can refer to its corresponding Informed element by index, name, id, absolute position, or relative position. A reference object may also reference a group of Informed elements which fall within a particular range or list, or which match a particular test. Each reference object provides constructors for the variety of ways in which the Informed elements can be referenced.

Index Reference Objects

A reference object that refers to its corresponding Informed element by index is called an index reference object. Any Informed element which has an index property can be referenced by an index reference object. The constructor for an index reference object requires the object's index as the first argument and the object's container as the second argument. If no container is specified, the implied container will be used.

```
// references the fourth cell of the record referenced by theRec
theCell = new Cell(4, theRec);

// references the fourth cell of the current record
theRec.Cell(4)
```

The way an object's index is determined depends on the type of object, as shown in the table below:

Determining Index

Object	How Index is Determined
Document	A Document's index represents its front to back position on the screen. The frontmost Document would be Document (1), the Document behind it would be Document (2).

Record	A Record's index refers to its position in its container. A Record's container can be a document or a collection:
	`//references the first record of the second document`
	`Doc(2).Record(1)`
	`//references the first record of the current collection of the`
	`//second document`
	`Doc(2).currentCollection.Record(1)`

Cell	A Cell's index is taken from its Tab position on the form. For example, the cell with Tab position 4 would be referred to as Cell (4).

Name Reference Objects

Any Informed element which has a name property can be referenced by a name reference object. The constructor for a name reference object requires the object's name as the first argument and the object's container for the second argument. If no container is specified, the implied container will be used.

```
// references the attachment named "Addendum" which belongs
// to the record referenced by theRec
theAttachment = new Attachment("Addendum", theRec);
```

ID Reference Objects

Any object which has an id property can be referenced by an id reference object. The constructor for an ID reference object requires the object's ID reference as the first argument and the object's container as the second argument. If no container is specified, the implied container will be used.

Depending on the class, the id property of an object should be either an integer or a string. A reference id is obtained by passing the appropriate integer or string to the built-in id() function.

```
// references the record whose id is 1620 of the
// document referenced by theDoc
theRec = new Record (id(1620), theDoc);
```

Absolute Position Reference Objects

Any object which has an index property can be referenced by an absolute position reference object. The constructor for an absolute position reference object requires a constant which represents the object's absolute position as the first argument and the object's container as the second argument. If no container is specified, the implied container will be used.

The Informed object's absolute position can be referenced by any of the following constants: FIRST, LAST, MIDDLE, ANY, or ALL. The ALL constant references all Informed objects of the specified class contained within the specified container.

```
//references the first cell of the record referenced by theRec
theCell = new Cell(FIRST, theRec);

// references every record in the current collection
// of the document referenced by theDoc
theRecList = theDoc.currentCollection.Record(ALL);
```

Relative Position Reference Objects

Any object which has an index property can be referenced by a relative position reference object. The constructor for a relative position reference object requires a constant which represents the object's relative position as the first argument and the object's container as the second argument. If no container is specified, the implied container will be used. Both the object and its container must have the same class.

The object's absolute position can be referenced by any of the following constants: NEXT, or PREVIOUS.

```
// references the next cell after the cell referenced by theRefCell
theCell = new Cell(NEXT, theRefCell);
```

Range Reference Objects

A range reference object is used to reference any range of indexed objects within the same container. A range consists of all objects with indices between two boundary objects, inclusive. The constructor for a range reference object requires the lower boundary reference object as the first argument, the upper boundary reference object as the second argument, and the objects' container as the third argument. Each boundary object must be an index, name, id, or absolute position.

```
// references cells 5 through the last of the document referenced by theDoc
theRecList = new Record(5, LAST, theDoc);
```

Note When using a range reference, the cells between the lower boundary and the upper boundary are determined by Tab position.

List Reference Objects

A list reference object refers to an arbitrary list of objects within the same container. The constructor for a list reference object requires an Array of object references as the first argument and the objects' container as the second argument. Each object reference must be an index or a name.

```
//refers to a list consisting of the cells "Qty" and "Cost"
theRefList = new Array("Qty", "Cost");
theCellList = theRec.Cell(theRefList);
```

Test Reference Objects

A test reference object is used to reference any group of objects within the same container which pass a particular test. The constructor for a test reference object requires a test descriptor as the first argument and the objects' container as the second object.

A test descriptor describes a comparison test or a logical test to be performed on each object within the container. If an object passes the test, then it will be included among the objects referenced by the test reference object.

Comparison Test Descriptors

Comparison test descriptors are obtained by calling one of the built-in functions testEQ(), testNE(), testGT(), testGE(), testLT(), testLE(), testBEG(), testEND(), or testCON(). Each of these functions requires two test arguments on which it performs the appropriate comparison. Each argument can be a reference object contained by the object being tested, or a property of the object being tested. If so, use `self` as the container of the argument. Each argument can also be any other valid JavaScript value.

```
//tests each record of the second document to see if the value of the
//"Name" cell is equal to "Bob"
Doc(2).Record(testEQ(self.Cell("Name"), "Bob"));

//references every record in the document referenced by
// theDoc whose index > 10
theRecList = new Record(testGT(self.Index, 10), theDoc);
```

Logical Tests Descriptors

Logical test descriptors are obtained by calling one of the built-in functions testAND(), testOR(), or testNOT(). The functions testAND() and testOR() require two or more test descriptor arguments and testNOT() requires one test descriptor argument. Each argument may be either a comparison test descriptor object or a logical test descriptor object.

```
// references every record in the current document whose
// cell "Name" contains "Brent" and whose cell "Date" equals today's date
test1 = testCON (self.Cell("Name").value, "Brent");
test2 = testEQ (self.Cell("Date".value), new LongDate);
theTest = testAND (test1, test2);
theRecList = Informed.currentDocument.Record(theTest);
```

Index Test Reference Objects

An index test reference object references a specific object from those which pass a particular test. The constructor for an index test reference object requires an index value as the first argument, a test descriptor as the second argument, and the object's container as the third argument. The index argument can be specified by either an integer value or one of the following absolute position constants: FIRST, LAST, MIDDLE, ANY, or ALL.

```
// references the second record of the document referenced
// by theDoc whose cell "Qty" <= 100
theTest = testLE (self.Cell("Qty").value, 100));
theRecList = theDoc.Record(2, theTest);

// references the last record of the document referenced
// by theDoc whose cell "Price" > 1000
theTest = testGT (self.Cell("Price").value, 1000));
theRecList = theDoc.Record(2, theTest);
```

Range Test Reference Objects

A range test reference object references a specific range of objects from those which pass a particular test. The constructor for a range test reference object requires index values as the first and second arguments, a test descriptor as the third argument, and the objects' container as the fourth argument. Each of the index arguments can be specified by either an integer value or one of the following absolute position constants: FIRST, LAST, MIDDLE, ANY, or ALL.

```
// references the first five records of the document referenced
// by theDoc whose cell "Cost" > 10000
theTest = testGT (self.Cell("Cost").value, 10000);
theRecList = theDoc.Record (1, 5, theTest);
```

Reference Object Descriptions

The following sections describe each of the reference objects supported by Informed. Each object description lists the reference methods, properties, and methods supported by the object.

AttachedScript

A document can contain a number of attached scripts. An attached script can be linked to a button or a menu item and can be executed by selecting the associated menu item or button. It can also be executed directly by another script. An **`AttachedScript`** object represents one or more scripts that are attached to a document.

Reference

An AttachedScript object can reference attached scripts by:

- name
- index
- id
- absolute position
- relative position
- range
- test

Properties

The following table lists the properties of an AttachedScript object.

AttachedScript Properties

Property	Writeable?	Description
container	no	The container for the attached script. An attached script is always contained by a Document object.
id	no	The unique id of the attached script.
index	no	The index of the attached script.
name	no	The name of the attached script.
objectClass	no	The AttachedScript class.

Methods

The following methods are defined for an AttachedScript object:

execute ()

The **execute** method executes an attached script and returns its result.

```
// Execute the script named "Post Submit" of the current document and store its
// result in the cell named "Submit Results" of the current record of the current
// cell.
// This example uses implied containment.See "Containment" earlier in this chapter
// for more information.
Cell("Submit Results").value = AttachedScript("Post Submit").execute();
```

exists ()

The **exists** method verifies the existence of an attached script.

```
// If it exists, execute the script named "Process Orders" of the document
// named "PO Batch" ignoring its result.
theScript = Document("PO Batch").AttachedScript("Process Orders");
if (theScript.exists())
    theScript.execute();
```

Attachment

With paper forms, associated documents such as receipts or drawings are often attached to a form with a paper clip. Informed Filler provides this same capability by allowing you to attach electronic documents to electronic forms. An **Attachment** object represents one or more files that are attached to a record.

Reference

An Attachment object can reference attachments by:

- name
- index
- id
- absolute position
- relative position
- range
- test

Properties

The following table lists the properties of an Attachment object.

Attachment Properties

Property	Writeable?	Description
container	no	The container for the attachment. An attachment is always contained by a record.
id	no	The unique id of the attachment.
index	no	The index of the attachment.
name	no	The name of the attachment.
objectClass	no	The Attachment class.

Methods

The following methods are defined for an Attachment object:

exists ()

The **exists** method verifies the existence of an attachment.

```
// Set the variable "ok" to true if an attachment named "Sample.txt" exists
// in the current record of the current document.
ok = Informed.currentDocument.currentRecord.Attachment("Sample.txt").exists();
```

remove ()

The **remove** method deletes an attachment from a record.

```
// Remove every attachment from every record of the current collection of the
// document referenced by the variable theDoc.
// This example uses implied containment.See "Containment" earlier in this chapter
// for more information.
theDoc.currentCollection.Attachment(ALL).remove();
```

save (inFile)

The **save** method saves an attachment.

Arguments for the save method

Argument	Description
inFile	This argument must be a File object which specifies the file into which the attachment will be saved.

```
// Save every attachment of the current record of the document named "Submissions"
// with its own name into the directory "c:\submit\".
theRec = Document("Submissions").currentRecord;
for (i = 1; i <= theRec.count(Attachment); i++) {
    theAttachment = theRec.Attachment(i);
    theAttachment.save(File("c:\submit\\" + theAttachment.name));
}
```

BuiltinCommand

Built-in commands correspond to the commands that are built into Informed Filler. The “Send Mail” command is an example of a built-in command. A **BuiltinCommand** object represents one or more built-in commands in Informed Filler. For a list of Informed’s built-in commands, please see “Appendix B” in this manual.

Reference

A BuiltinCommand object can reference built-in commands by:

- name
- index
- id
- absolute position
- relative position
- range
- test

Properties

The following table lists the properties of a BuiltinCommand object.

BuiltinCommand Properties

Property	Writeable?	Description
enabled	yes	Is the built-in command enabled?

id	no	The unique id of the built-in command.
index	no	The index of the built-in command.
name	no	The name of the built-in command.
objectClass	no	The BuiltinCommand class.

Methods

The following methods are defined for the BuiltinCommand object:

execute ()

The **execute** method executes a built-in command.

```
// Execute the built-in command "Log Off Service" to log off the signing service.
BuiltinCommand("Log Off Service").execute();
```

exists ()

The **exists** method verifies the existence of a built-in command.

```
// Set the variable ok to true if the built-in command "Send Mail" exists.
ok = BuiltinCommand("Send Mail").exists();
```

Button

A button on a form can be configured to invoke commands that are built into Informed Filler, commands that are available through Informed plug-ins, or scripts that are attached to the form. A **Button** object represents one or more buttons on a form.

Reference

A Button object can reference buttons by:

- name
- index
- absolute position
- relative position
- range
- test

Properties

The following table lists the properties of a Button object.

Button Properties

Property	Writeable?	Description
container	no	The container for the button. A button is always contained by a document.
enabled	yes	Is the button enabled?
index	no	The index of the button.
name	no	The name of the button.
objectClass	no	The Button class.
title	no	The title of the button.

Methods

The following methods are defined for a Button object:

execute ()

The **execute** method executes a button's configured action.

```
// Execute the button named "Approve" of the current document if it is enabled.
theButton = Button("Approve");
if (theButton.enabled)
    theButton.execute();
```

exists ()

The **exists** method verifies the existence of a button.

```
// Check for the existence of the fifth button of the second document.
buttonExists = Document(2).Button(5).exists();
```

Cell

A **Cell** object represents one or more cells in a record. Cells include single value fields (drawn with the Field tool in Informed Designer) and multiple value fields, (drawn with the Table tool in Informed Designer).

Reference

A Cell object can reference cells by:

- name
- index
- id
- absolute position
- relative position
- range
- test

Properties

The following table lists the properties of a Cell object.

Cell Properties

Property	Writeable?	Description
container	no	The container for the cell. A cell is always contained by a record.
currentElement	yes	The current element.
displayOnly	yes	Is the cell display only?
extraChoices	yes	The list of extra choices for the cell.
id	no	The unique id of the cell.
index	no	The index of the cell, relative to other cells.
mainChoices	no	The list of main choices for the cell.
name	no	The name of the cell.
objectClass	no	The Cell class.
signed	no	Is the cell signed?
tableID	no	The table id for a column cell.
title	no	The title of the cell.
value	yes	The value of the cell.

Methods

The following methods are defined for a Cell object:

clear ()

The **clear** method clears the cell of any data.

```
// Clear the cell named "Signature" of the current record of the document named
// "Authorization".
Document("Authorization").currentRecord.Cell("Signature").clear();

// Clear the cell named "Signature" of every record of the current collection of
// the document named "Authorization".
// This example uses implied containment.See "Containment" earlier in this chapter
// for more information.
Document("Authorization").currentCollection.Cell("Signature").clear();

// Clear the cell named "Signature" of every record of the document named
// "Authorization".
// This example uses implied containment.See "Containment" earlier in this chapter
// for more information.
Document("Authorization").Cell("Signature").clear();

// Clear every row of the table column cell named "Description" of the current
// record of the current document.
// This example uses implied containment.See "Containment" earlier in this chapter
// for more information.
Cell("Description").clear();
```

commit ([withLookup])

The **commit** method commits the cell data, triggering any formatting, check calculations, or lookups configured for the cell.

Arguments for the commit method

Argument	Description
withLookup	Specifies whether or not a lookup is performed. If `true`, the lookup is performed unconditionally. If `false`, the lookup is ignored. If `null`, a lookup is performed only if the cell's value has changed. The default value for the withLookup argument is `null`. This argument is ignored if the specified cell is not a lookup cell.

```
// Perform a deferred lookup by committing the cell data at a later time.
theDoc = Document("Purchase Order");
lookupCell = theDoc.currentRecord.Cell("Part Number");
lookupCell.set("PA123", true, false); // set lookup cell data but suppress lookup
DoSomeStuff();                        // do some other stuff before doing lookup
theDoc.currentCell = lookupCell;      // make lookup cell the current cell
lookupCell.commit(true);              // force lookup now
```

count (elementClass)

The **count** method returns the number of elements in a cell.

Arguments for the count method

Argument	Description
elementClass	This argument specifies the class of the elements to be counted. Its value must be Element.

```
// Count the number of elements in the table cell "Result" of the current record of
// the second document.
rowCount = Document(2).currentRecord.Cell("Result").count(Element);
```

dataSize ()

The **dataSize** method returns the size of the cell data in bytes.

```
// Get the data size of the current cell of the current document.
theSize = Informed.currentDocument.currentCell.dataSize();
```

exists ()

The **exists** method verifies the existence of a cell.

```
// Check for the existence of Cell with an index of 999 in the third record
// of the frontmost document.
exists = Document(1).Record(3).Cell(999).exists();
```

get ()

The **get** method gets the value of a cell.

```
// Get the value of the cell named "Signed Date" of the current record of the
// frontmost document.
theValue = Document(FIRST).currentRecord.Cell("Signed Date").get();

// Get the value of the first cell in the current document.
// This example uses implied containment.See "Containment" earlier in this chapter
// for more information.
theValue = Cell(FIRST).get();

// Get the value of the table cell named "Description" of the current record of the
// current document. The result is an array.
theValueList = Informed.currentDocument.currentRecord.Cell("Description").get();

// Get the value of the cell named "Last Name" of every record of the current
// collection of the second document.
// This example uses implied containment.See "Containment" earlier in this chapter
// for more information.
theValueList = Document(2).currentCollection.Cell("Last Name").get();
```

```
// Get the value of the cell named "Birthdate" of every record of the document
// named "Student Info".
// This example uses implied containment.See "Containment" earlier in this chapter
// for more information.
theValueList = Document("Student Info").Cell("Birthdate").get();
```

set (value,[withCommit],[withLookup])

The **set** method sets the value of a cell.

Arguments for the set method

Argument	Description
value	The value to which the cell will be set.
withCommit	If true, the data is committed to the cell immediately and any check calculations or formatting options are triggered. If false, the data is not committed immediately. The default is true.
withLookup	Specifies whether or not a lookup is performed. If true, the lookup is performed unconditionally. If false, the lookup is ignored. If null, a lookup is performed only if the cell's value has changed. The default value for the withLookup argument is null. This argument is ignored if the specified cell is not a lookup cell. **Note**: If withLookup is specified (true or false), then withCommit must be true or null.

```
// Clear the current cell of the current document.
Informed.currentDocument.currentCell.set("");

// Set a list of cells to a list of values.
cellNames = new Array ("Company Name", "Phone Number");
theCells = Document(2).currentRecord.Cell(cellNames);
theData = new Array ("Shana Corporation", "(403) 433-3690");
theCells.set(theData);

// Set a table column cell named "Values" from an Array of values.
theValues = new Array ("A", "B", "C");
theDoc.currentRecord.Cell("Values").set(theValues);

// Set each row of a table cell named "Items" of the current record of the document
// named "Summary" to the value of the cell named "Item" of each record of the
// current collection of the document named "Detail".
// This example uses implied containment.See "Containment" earlier in this chapter
// for more information.
theValueList = Document("Detail").currentCollection.Cell("Item").get();
Document("Summary").currentRecord.Cell("Items").set(theValueList);
```

```
// Set the cell named "Phone Number" of the current record of the document named
// "Address Info" to "(403)" but donít commit it.
theCell = Document("Address Info").currentRecord.Cell("Phone Number");
theCell.set("(403)", false);

// Copy cell "Part Number" from the current record of the document named "Purchase
// Requisition" to the current record of "Purchase Order" without performing the
// resulting lookup.
theVal = Document("Purchase Requisition").currentRecord.Cell("Part Number").get();
orderCell = Document("Purchase Order").currentRecord.Cell("Part Number");
orderCell.set(theVal, true, false);
```

sign ([signingSystem])

The **sign** method signs the indicated signature cells.

Arguments for the sign method

Argument	Description
signingSystem	Specifies which signing service to use. This argument can be a constant such as ENTRUST, ISIGNIMAP, or ISIGNPOP, a string that specifies the name of a signing plug-in, or null. The default value for this argument is null, which uses the signing service selected on the user's Preferences dialog, or displays a dialog asking the user to select a signing service.

```
// Sign the signature cell named "Signature" of the current record of the
// document named "Authorization" using the Entrust signing system.
Document("Authorization").currentRecord.Cell("Signature").sign(ENTRUST);

// Sign the cell named "Signature" of every record of the current collection of the
// document named Authorization" using the signing system configured for the cell.
// This example uses implied containment.See "Containment" earlier in this chapter
// for more information.
Document("Authorization").currentCollection.Cell("Signature").sign();
```

verify ()

The **verify** method verifies the indicated signature cells and returns true if the cell contains a valid signature, and false if the signature is not valid (or if the cell is not signed).

```
// Verify the signature in the cell named "Signature" of the current record of the
// document named "Authorization".
isValid = Document("Authorization").currentRecord.Cell("Signature").verify();

// Verify the cell named "Signature" of every record of the document named
// "Authorization". The result is an Array of boolean values.
// This example uses implied containment.See "Containment" earlier in this chapter
// for more information.
isValidList = Document("Authorization").Cell("Signature").verify();
```

Collection

A **Collection** object represents the current collection of records in a document.

Reference

A Collection object can reference collections by:

- index
- id
- absolute position
- test

Properties

The following table lists the properties of a Collection object.

Collection Properties

Property	Writeable?	Description
container	no	The container for the collection. A collection is always contained by a document.
id	no	The unique id of the collection
index	no	The index of the collection.
objectClass	no	The Collection class.

Methods

The following methods are defined for a Collection object:

count (elementClass)

The **count** method returns the number of records in a collection.

Arguments for the count method

Argument	Description
elementClass	This argument specifies the class of the elements to be counted. Its value must be Record.

```
// Count the number of records in the current collection of the current document.
theCount = Informed.currentDocument.currentCollection.count(Record);
```

exists ()

The **exists** method verifies the existence of a collection.

```
// Check if the current collection of the second document exists.
exists = Document(2).currentCollection.exists();
```

export (toFile,[whichCells],[format],[doAppend],[rowwise],[useQuotes] [doMerge],[includeNotes])

The **export** method exports every record in a collection to a file.

Arguments for the export method

Argument	Description
toFile	This argument must be a File object which specifies the file into which the data will be exported.
whichCells	Specifies the cells which will be exported. This argument can be a single Cell object, an array of Cell objects, a single column, or an array of columns. The default value for whichCells is `null`, which specifies all cells. No container is allowed for this argument.
format	Specifies the file format of the export file. This argument can be one of the following constants: `INTERCHANGE`, `TAB_DELIMITED`, or `COMMA_DELIMITED`, or a string that specifies a format name. The default value for the format argument is `INTERCHANGE`.
doAppend	If `true`, the exported records are appended to the end of the export file. If `false`, the export file is replaced with the exported records. The default value is `false`.
rowwise	If `true`, then tables will be exported in row order. If `false`, tables will be exported in column order. The default value is `false`.
useQuotes	If `true`, then all exported values—except numbers—are surrounded with quotes. If `false`, then only those values which contain delimiter characters are surrounded with quotes. This argument is ignored if the format argument is not `TAB_DELIMITED` or `COMMA_DELIMITED`.
doMerge	If `true`, Informed will list each cell name on the first line of a new export file. The default value is `true`. This argument is ignored if the format argument is not `TAB_DELIMITED` or `COMMA_DELIMITED`.
includeNotes	If `true`, then any notes attached to the form will be exported. The default value is `true`. This argument is ignored if the format argument is not `INTERCHANGE`.

```
// Export all cells of every record of the current collection of the current
// document to the Informed Interchanged file named "info.iif".
theFile = File("HD:info.iif");
Informed.currentDocument.currentCollection.export(theFile);
```

```
// Append the cells named "LastName", "FirstName", "Address", and "Phone" of
// every record in the current collection of the document named "Customers" to the
// tab delimited file named "names.txt" using quotes for all non-numeric data.
theCollection = Document("Customers").currentCollection;
theFile = File("c:\\names.txt");
theCells = Cell(new Array ("LastName", "FirstName", "Address", "Phone"));
theCollection.export(theFile, theCells, TAB_DELIMITED, true, null, true);
```

make (elementClass)

The **make** method creates a new record in a collection.

Arguments for the make method

Argument	Description
elementClass	This argument specifies the class of the new element. Its value must be Record.

```
// Create a new record in the current collection of the document named
// "Receivables".
theRec = Document("Receivables").currentCollection.make(Record);
```

print ([as],[copies],[fromPage],[toPage],[fromPart],[toPart],[printTemplate], [printData],[collate])

The **print** method prints every record in a collection.

Arguments for the print method

Argument	Description
as	Specifies whether to print as forms or as a list. This argument can be either FORMS or RECORD_LIST. The default value is FORMS.
copies	Specifies the number of copies to print. The default value is 1.
fromPage	Specifies the page to start printing from. The default value is the first page.
toPage	Specifies the page to stop printing at. The default value is the last page.
fromPart	Specifies which part of a multipart form to start printing from. The default value is the first part.
toPart	Specifies which part of a multipart form to stop printing at. The default value is the last part.
printTemplate	If false, then don't print the template. The default value is true.
printData	If false, then don't print the data. The default value is true.
collate	Specifies whether or not to collate pages. The default value is true.

```
// Print every record in the collection of the current document as a single
// form.
Informed.currentDocument.currentCollection.print();

// Print two copies of the first page of every record in the second documentís
// current collection as a record list.
Document(2).currentCollection.print(RECORD_LIST, 2, 1, 1);
```

send ([recipients],[subject],[body],[format],[encloseAs],
[messageAttachments])

The **send** method sends every record in a collection using an electronic mail service.

Arguments for the send method

Argument	Description
recipients	Specifies the recipients. This argument can be a string or an array of strings.
subject	Specifies the subject of the message. The default is the name of the document being sent.
body	The body of the mail message. The default value is `null` (no body).
format	The format to send the form in. This argument can be one of the following constants: `DATA`, `PACKAGE`, `INTERCHANGE`, `COMMA_DELIMITED`, or `TAB_DELIMITED`, or a string that specifies the name of a format. The default value is `DATA`.
encloseAs	Specifies the name of the form attachment. The default value is the name of the data document.
messageAttachments	Specifies any additional attachments. This argument can be a File object or an array of File objects.

```
// Send every record of the current collection of the current document to the
// recipient "someone@someplace.com".
Informed.currentDocument.currentCollection.send("someone@someplace.com");

// Send every record of the current collection of the document named "Purchase
// Request" using the provided addressing information.
theCollection = Document("Purchase Request").currentCollection;
theRecipients = new Array ("someone@someplace.com", "someoneelse@someplace.com");
theSubject = "Purchase Request Form";
theBody = "Here's the purchase request form you requested.";
theAttach = new File ("c:\\request.xls");
theCollection.send(theRecipients, theSubject, theBody, PACKAGE, null, theAttach);
```

sendExt ([usingStep],[recipients],[ccRecipients],[bccRecipients],
[appendRecipients],[subject],[body],[format],[encloseAs],
[messageAttachments],[appendAttachments],[mailSystem])

The **sendExt** method sends every record in a collection.

Arguments for the sendExt method

Argument	Description
usingStep	Specifies the name or the index of the routing step to use. The default value is `null` (no routing step). If a routing step is supplied, all other optional arguments override their corresponding settings in the routing step configuration.
recipients	Specifies the recipients. This argument can be a string or an array of strings. The default is `null` (no recipients).
ccRecipients	Specifies a list of recipients to cc. This argument can be a string or an array of strings. The default is `null` (no cc recipients).
bccRecipients	Specifies a list of recipients to bcc. This argument can be a string or an array of strings. The default is `null` (no bcc recipients).
appendRecipients	Specifies whether or not to append other recipients to those already specified in a routing step. The default value is `false`.
subject	Specifies the subject of the message. The default is the name of the document being sent.
body	The body of the mail message. The default value is `null` (no body).
format	The format to send the form in. This argument can be one of the following constants: `DATA`, `PACKAGE`, `INTERCHANGE`, `COMMA_DELIMITED`, or `TAB_DELIMITED`, or a string that specifies the name of a format. The default value is `DATA`.
encloseAs	Specifies the name of the form attachment. The default value is the name of the data document.
messageAttachments	Specifies any additional attachments. This argument can be a File object or an array of File objects. The default value is null (no additional attachments).
appendAttachments	Specifies whether or not to append other attachments to those already specified in a routing step. The default value is `false`.

mailSystem	This argument is illegal if a routing step is provided. If no routing step is provided, this argument can be a constant, a string that specifies the name of a mail plug-in, or null which calls the default mail system on the user's machine. Informed provides the following constants for mail systems that have the same name on both Windows and MacOS:
	`SMTP`, `EUDORA`, `MSMAIL`, `CCMAIL`, and `GROUPWISE`
	The following constants are for Windows only:
	`EXCHANGE, MAPI, VIM, MHS, MHSLOCAL`
	The following constants are for MacOS only:
	`QUARTERDECK, QUICKMAIL`

```
// Send every record of the current collection of the current document as specified
// by the 2nd routing step, except override the message body and CC to the
// recipient "somebody@someplace.com".
theCollection = Informed.currentDocument.currentCollection;
theRecipient = "someone@someplace.com";
theBody = "This is the new message body.";
theCollection.sendExt(2, theRecipient, null, null, true, null, theBody);
```

sendStep ([usingStep])

The **sendStep** method sends every record in a collection using a preconfigured routing step.

Arguments for the sendStep method

Argument	Description
usingStep	Specifies the name or the index of the routing step to use. The default value is `null` (no routing step). If a routing step is supplied, all other optional arguments override their corresponding settings in the routing step configuration.

```
// Send every record of the current collection of the current document using the
// routing step named "Route To Manager".
Informed.currentDocument.currentCollection.sendStep("Route To Manager");
```

sort (sortCell,[descending])

The **sort** method sorts the records in the current collection.

Arguments for the sort method

Argument	Description
sortCell	Specifies the cells or columns on which to sort. Its value must be a Cell object, Column object, or Array of Cell or Column objects. If an Array object is specified, the current collection is sorted on each cell in the Array, beginning with the first cell in the Array and ending with the last. No container is allowed for this argument.
descending	Specifies whether or not to sort in descending order. The default is `false`.

```
// Sort the current collection of the document named "Employees" according to the
// "Employee No." cell.
Document("Employees").currentCollection.sort(Cell("Employee No."));

// Sort the current collection of the document "Employees" by the column named
// "Age" in descending order.
Document("Employees").currentCollection.sort(Column("Age"), true);

// Sort the current collection of the current document first by the cell "First
// Name" and then by the cell "Last Name". The result will be that the cells will
// be sorted by first name within last name as would be found in a phone book.
theSortCells = Cell(new Array("First Name", "Last Name"));
Informed.currentDocument.currentCollection.sort(theSortCells);
```

Column

Informed Filler's Record List is a standard window that displays records in a list format. Information on the Record List is divided into rows and columns. Each row in the list represents one record. Each column corresponds to one cell on the form. A **Column** object represents one or more columns on the Record List.

Reference

A Column object can reference columns by:

- name
- index
- id
- absolute position
- relative position
- range
- test

Properties

The following table lists the properties of a Column object.

Column Properties

Property	Writeable?	Description
alignment	no	The alignment of the data in the column. This property can be one of the following constants: `LEFT`, `CENTER`, or `RIGHT`.
average	no	The average of the data in the column. This property only works with averaged columns.
cellID	no	The id of the cell associated with the column.
container	no	The container for the column. A column is always contained by a format.
id	no	The unique id of the column.
index	no	The index of the column.
name	no	The name of the column.
objectClass	no	The Column class.
selected	no	Is the column selected?
sorted	no	Is the column sorted?
total	no	The total of the data in the column. This property only works with totalled columns.
totalsType	no	The totals type of the column. This property can be one of the following constants: `NONE`, `TOTALLED`, or `AVERAGED`.
width	no	The width of the column in pixels.

Methods

The following methods are defined for a Column object:

clear ()

The **clear** method clears the column of any data.

```
// Clears the column named "Approval" of the current format of the current
// document.
Informed.currentDocument.currentFormat.Column("Approval").clear();
```

exists ()

The **exists** method verifies the existence a column.

```
// Get the total of the third column of the current format of the document
// referenced by theDoc.
theColumn = theDoc.currentFormat.Column(3);
if (theColumn.exists())
    if (theColumn.totalsType == TOTALLED)
       theTotal = theColumn.total;
```

Document

A **Document** object represents an opened data document in Informed Filler.

Reference

A Document object can reference documents by:

- name
- index
- id
- absolute position
- relative position
- range
- test

Properties

The following table lists the properties of a Document object.

Document Properties

Property	Writeable?	Description
attachmentsWindow	no	The attachments window
authorName	no	The template author's name as entered on the Template Information dialog.
authorOrganization	no	The template author's organization as entered on the Template Information dialog.
checkPeriod	yes	The revision check period. This property can be one of the following constants: EVERY_TIME, DAILY, WEEKLY, MONTHLY, or NEVER.
currentCell	yes	The current cell.
currentCollection	yes	The current collection of records.

currentFormat	yes	The current Record List format.
currentPage	yes	The current page of the form.
currentRecord	yes	The current record in the document.
description	no	The description of the template as entered on the Template Information dialog.
diskFile	no	The disk file that contains the data document.
distributed	no	Is the template distributed?
formWindow	no	The form window.
id	no	The unique id of the data document.
index	yes	The index of the data document.
lastChecked	no	When the last revision check occurred.
modified	no	Has the data document been modified?
name	no	The name of the data document.
objectClass	no	The Document class.
pageCount	no	The number of pages in the document.
recordListWindow	no	The Record List window.
revision	no	The revision number of the template as entered on the Template Information dialog.
status	no	The status of the template. This property will be one of the following constants: `CURRENT`, `NONCURRENT`, or `DISCONTINUED`.
statusMessage	no	The status message for the template as entered on the Template Information dialog.
templateID	no	The template document's template ID as entered on the Template Information dialog.
templateName	no	The template document's template name as entered on the Template Information dialog.

Methods

The following methods are defined for a Document object:

close ([saving],[savingIn])

The **close** method closes a document.

Arguments for the close method

Argument	Description
saving	Specifies whether changes should be saved before closing. If `true`, changes will be saved before closing. If `false`, changes will not be saved. If `null`, Informed will display a dialog asking if the user wants to save the changes if necessary. The default value is `null`.
savingIn	Specifies the file in which to save the document. The default value is the file into which the document was previously saved. If the document was not previously saved, the standard Save dialog is displayed.

```
// Close the second document. Prompt the user to save if necessary.
Document(2).close();

// Close the document whose id is 104 without saving.
Document(id(104)).close(false);

// Close and save the document named "Inventory".
Document("Inventory").close(true);

// Close and save the current document as "Invoice".
thePlatform = Informed.platform;
if (thePlatform = WIN32 || thePlatform = WIN16)
   thePath = "c:\\informed\\data\\invoice.ifm";
else
   thePath = "HD:InformedÆ:Data:Invoice";
Informed.currentDocument.close(true, thePath);
```

collect (fileList,[append])

The **collect** method imports one or more files into a document.

Arguments for the collect method

Argument	Description
fileList	Specifies the files to import into the document. Its value must be a File object or an Array of File objects.
append	If `true`, the imported records are appended to the end of the current collection. If `false`, the imported records become the current collection. The default value is `false`.

```
// Import the tab-delimited text file called "transact.txt" into the current
// document without appending.
Informed.currentDocument.collect(File("c:\\transact.txt"));

// Import the Informed Interchange file called "Temp Records" into the second
// document and append the records to the current collection.
Document(2).collect(File("HD:SomeDirectory:Temp Records"), true);

// Import each of the files "Temp1", "Temp2", and "Temp3" into the document named
// "Archive" without appending to the current collection.
file1 = File("Temp1");
file2 = File("Temp2");
file3 = File("Temp3");
theFileList = new Array(file1, file2, file3);
Document("Archive").collect(theFileList);
```

count (elementClass)

The **count** method returns the number of attached scripts, buttons, collections, formats, records, saved formats, or tags within a document.

Arguments for the count method

Argument	Description
elementClass	This argument specifies the class of the elements to be counted. Its value must be AttachedScript, Button, Collection, Format, Record, SavedFormat, or Tag.

```
// Count the number of records in the current document.
theCount = Informed.currentDocument.count(Record);

// Count the number of attachments in each record of the first document.
// Note: The result will be an array of integers.
theCountList = Document(1).count(Attachment);
```

exists ()

The **exists** method verifies the existence a document.

```
// Close the document named "Attendance" if it exists.
if (Document("Attendance").exists())
   Document("Attendance").close(true);
```

export (toFile,[whichCells],[format],[doAppend],[rowwise],[useQuotes] [doMerge],[includeNotes])

The **export** method exports every record in a document to a file.

Arguments for the export method

Argument	Description
toFile	This argument must be a File object which specifies the file into which the data will be exported.
whichCells	Specifies the cells which will be exported. This argument can be a single Cell object, an array of Cell objects, a single column, or an array of columns. The default value for whichCells is `null`, which specifies all cells. No container is allowed for this argument.
format	Specifies the file format of the export file. This argument can be one of the following constants: `INTERCHANGE`, `TAB_DELIMITED`, or `COMMA_DELIMITED`, or a string that specifies a format name. The default value for the format argument is `INTERCHANGE`.
doAppend	If `true`, the exported records are appended to the end of the export file. If `false`, the export file is replaced with the exported records. The default value is `false`.
rowwise	If `true`, then tables will be exported in row order. If `false`, tables will be exported in column order. The default value is `false`.
useQuotes	If `true`, then all exported values—except numbers—are surrounded with quotes. If `false`, then only those values which contain delimiter characters are surrounded with quotes. This argument is ignored if the format argument is not `TAB_DELIMITED` or `COMMA_DELIMITED`.
doMerge	If `true`, Informed will list each cell name on the first line of a new export file. The default value is `true`. This argument is ignored if the format argument is not `TAB_DELIMITED` or `COMMA_DELIMITED`.
includeNotes	If `true`, then any notes attached to the form will be exported. The default value is `true`. This argument is ignored if the format argument is not `INTERCHANGE`.

```
// Export all cells of every record in the current document to the Informed
// Interchange file named "details.iif".
Informed.currentDocument.export(File("c:\\details.iif"));

// Append the "LastName", "FirstName", "Address", and "Phone" cells of every record
// in the document named "Customers" to the tab-delimited text file "Address Info".
theFile = File("HD:Address Info");
theCellList = Cell(new Array ("LastName", "FirstName", "Address", "Phone"));
Document("Customers").export(theFile, theCellList, TAB_DELIMITED, true);
```

make (elementClass, withData,[withProperties])

The **make** method creates a new record, saved format, or tag in a document.

Arguments for the make method

Argument	Description
elementClass	Specifies the class of the new element. Its value must be Record, SavedFormat, or Tag
withData	This argument is not used. Its value must be `null`.
withProperties	Specifies the initial values for the properties of the new element. This argument is not used when creating a new record. This argument must be an object with a name property that specifies the name of the new SavedFormat or Tag.

```
// Create a new record in the current document.
Informed.currentDocument.make(Record);

// Create a new saved format named "Special Report" in the document named "Roster".
theFormatProperties = new Object;
theFormatProperties.name = "Special Report";
Document("Roster").make(SavedFormat, null, theFormatProperties);

// Create a new tag named "Deadbeats" in the document named "Receivable" for every
// record whose cell "Balance Owing" is greater than 0.
theDoc = Document("Receivable");
theTest = testGT(self.Cell("Balance Owing").value, 0);
theDoc.currentCollection = Record(theTest);
theTagProperties = new Object;
theTagProperties.name = "Deadbeats";
theDoc.make(Tag, null, theTagProperties);
```

print ([as],[copies],[fromPage],[toPage],[fromPart],[toPart],[printTemplate], [printData],[collate])

The **print** method prints every record in a document.

Arguments for the print method

Argument	Description
as	Specifies whether to print as forms or as a list. This argument can be either FORMS or RECORD_LIST. The default value is FORMS.
copies	Specifies the number of copies to print. The default value is 1.
fromPage	Specifies the page to start printing from. The default value is the first page.
toPage	Specifies the page to stop printing at. The default value is the last page.
fromPart	Specifies which part of a multipart form to start printing from. The default value is the first part.
toPart	Specifies which part of a multipart form to stop printing at. The default value is the last part.
printTemplate	If false, then don't print the template. The default value is true.
printData	If false, then don't print the data. The default value is true.
collate	Specifies whether or not to collate pages. The default value is true.

```
// Print every record in the current document as a single form.
Informed.currentDocument.print();

// Print two copies of every record in the first document as a record list.
Document(FIRST).print(RECORD_LIST, 2);
```

save ([inFile],[format])

The **save** method saves a document.

Arguments for the save method

Argument	Description
inFile	This argument must be a File object which specifies the file into which the document will be saved. The default value is the file into which the document was previously saved. If the document was not previously saved, the standard Save dialog is displayed.
format	This argument specifies the file format of the saved file. This argument will be one of the following constants: DATA or PACKAGE. The default value is DATA.

```
// Save the document named "Samples".
Document("Samples").save();

// Save the current document as "c:\\summary.ifm".
Informed.currentDocument(File("c:\\summary.ifm"));
```

```
// Save the current document as a package in "c:\\summary.ipk".
Informed.currentDocument(File("c:\\summary.ipk"), PACKAGE);
```

search (matchCell, matchValue,[matchOption],[findOption])

The **search** method searches for specific records in a document.

Arguments for the search method

Argument	Description
matchCell	Specifies the cell to search in. This argument cannot have a container and must be either a single cell or column.
matchValue	The value to search for. This argument must be an Array of two values if the match option is RNG.
matchOption	The match option. This argument will be one of the following constants: EQ (is equal to) NE (is not equal to) LT (less than) LE (less than or equal to) GT (greater than) GE (greater than or equal to) BEG (begins with) END (ends with) CON (contains) RNG (range) The default value is RNG if the matchValue argument is an Array; CON if the matchValue argument is a text, character or name cell; or EQ other wise.
findOption	Specifies which records to look through and what to do with the records found. This argument will be one of the following constants: ALL_RECORDS (look through all records) COLLECTED_RECORDS (look through collected records) ADD_TO_COLLECTION (add to collection) OMIT_FROM_COLLECTION (omit from collection) FIRST_MATCH (go to first match in collection) SELECT_MATCHES (select matches in Record List). The default value is ALL_RECORDS.

```
// Set the current collection to every record whose cell "Name" contains
// "Smith".
Informed.currentDocument.search(Cell("Name"), "Smith");
```

```
// Add every record whose cell "Date" contains a date between January 1,
// 1998 and January 31, 1998 to the current collection of the third document.
rangeStart = new LongDate(0, 1998, 0, 1);
rangeEnd = new LongDate(0, 1998, 0, 31);
dateRange = new Array(rangeStart, rangEnd);
Document(3).search(Cell("Date"), dateRange, RNG, ADD_TO_COLLECTION);
```

send ([recipients],[subject],[body],[format],[encloseAs],
[messageAttachments]

The **send** method sends every record in a document using an electronic mail service.

Arguments for the send method

Argument	Description
recipients	Specifies the recipients. This argument can be a string or an array of strings.
subject	Specifies the subject of the message. The default is the name of the document being sent.
body	The body of the mail message. The default value is `null` (no body).
format	The format to send the form in. This argument can be one of the following constants: `DATA`, `PACKAGE`, `INTERCHANGE`, `COMMA_DELIMITED`, or `TAB_DELIMITED`, or a string that specifies the name of a format. The default value is `DATA`.
encloseAs	Specifies the name of the form attachment. The default value is the name of the data document.
messageAttachments	Specifies any additional attachments. This argument can be a File object or an array of File objects.

```
// Send every record of the current record to the recipient "jshmoe@worldcorp.com"
Informed.currentDocument.send("jshmoe@worldcorp.com");

// Send every record in the third document using the provided mail addressing
// information.
theRecipients = new Array("someone@worldcorp.com", "someoneelse@worldcorp.com");
theSubject = "Requested Form";
theBody = "Hereís the form you requested";
Document(3).send(theRecients, theSubject, theBody, INTERCHANGE);
```

sendExt ([usingStep],[recipients],[ccRecipients],[bccRecipients],
[appendRecipients],[subject],[body],[format],[encloseAs],
[messageAttachments],[appendAttachments],[mailSystem])

The **sendExt** method sends every record in a document. This method provides extended options over the **send** method.

Arguments for the sendExt method

Argument	Description
usingStep	Specifies the name or the index of the routing step to use. The default value is `null` (no routing step). If a routing step is supplied, all other optional arguments override their corresponding settings in the routing step configuration.
recipients	Specifies the recipients. This argument can be a string or an array of strings. The default is `null` (no recipients).
ccRecipients	Specifies a list of recipients to cc. This argument can be a string or an array of strings. The default is `null` (no cc recipients).
bccRecipients	Specifies a list of recipients to bcc. This argument can be a string or an array of strings. The default is `null` (no bcc recipients).
appendRecipients	Specifies whether or not to append other recipients to those already specified in a routing step. The default value is `false`.
subject	Specifies the subject of the message. The default is the name of the document being sent.
body	The body of the mail message. The default value is `null` (no body).
format	The format to send the form in. This argument can be one of the following constants: `DATA`, `PACKAGE`, `INTERCHANGE`, `COMMA_DELIMITED`, or `TAB_DELIMITED`, or a string that specifies the name of a format. The default value is `DATA`.
encloseAs	Specifies the name of the form attachment. The default value is the name of the data document.
messageAttachments	Specifies any additional attachments. This argument can be a File object or an array of File objects. The default value is null (no additional attachments).
appendAttachments	Specifies whether or not to append other attachments to those already specified in a routing step. The default value is `false`.

mailSystem	This argument is illegal if a routing step is provided. If no routing step is provided, this argument can be a constant, a string that specifies the name of a mail plug-in, or null which calls the default mail system on the user's machine. Informed provides the following constants for mail systems that have the same name on both Windows and MacOS:
	`SMTP`, `EUDORA`, `MSMAIL`, `CCMAIL`, and `GROUPWISE`
	The following constants are for Windows only:
	`EXCHANGE, MAPI, VIM, MHS, MHSLOCAL`
	The following constants are for MacOS only:
	`QUARTERDECK, QUICKMAIL`

```
// Send every record of the document referenced by the variable theDoc using the
// fifth routing step with the additional attachments specified.
theAttachments = new Array(File("c:\\report.wpf"), File("c:\\summary.xls"));
theDoc.sendExt(5, null, null, null, null, null, null, null, null, theAttachments);
```

sendStep ([usingStep])

The **sendStep** method sends every record in a document using a preconfigured routing step.

Arguments for the sendStep method

Argument	Description
usingStep	Specifies the name or the index of the routing step to use. The default value is `null` (no routing step). If a routing step is supplied, all other optional arguments override their corresponding settings in the routing step configuration.

```
// Send every record in the current document using the routing step called "Send to
// Manager".
Informed.currentDocument.sendStep("Send to Manager");
```

Element

An **Element** object refers to an element of a cell. A field cell contains only one element, whereas a table cell contains one element for each row of that table cell.

Reference

An Element object can reference elements by:

- index
- absolute position
- relative position
- range
- test

Properties

The following table lists the properties of an Element object.

Element Properties

Property	Writeable?	Description
container	no	The container for the element. An element is always contained by a cell.
index	no	The index of the row element.
objectClass	no	The Element class.
value	yes	The value of the row element.

Methods

The following methods are defined for an Element object:

clear ()

The **clear** method clears the element of any data.

```
// Clear the third element of the cell named "Person" of the current record of the
// current document.
Informed.currentDocument.currentRecord.Cell("Person").Element(3).clear();

// Clear the last element of the cell named "Price" of the current record of the
// second document.
// This example uses implied containment.See "Containment" earlier in this chapter
// for more information.
Document(2).currentRecord.Cell("Price").Element(LAST).clear();
```

commit ([withLookup])

The **commit** method commits the element data, triggering any formatting, check calculations or lookups configured for the element.

Arguments for the commit method

Argument	Description
withLookup	Specifies whether or not a lookup will be performed. If `true`, the lookup is performed unconditionally. If `false`, the lookup is ignored. If `null`, a lookup is performed only if the cell's value has changed. The default value for the withLookup argument is `null`. This argument is ignored if the specified cell is not a lookup cell.

```
// Perform a deferred lookup by committing the element data at a later time.
theDoc = Document("Purchase Order");
lookupElem = theDoc.currentRecord.Cell("Part Number").currentElement;
lookupElem.set("PA1234", true, false);// set element data but suppress lookup
DoSomeStuff();// do some other stuff before doing lookup
theDoc.currentCell = lookupElem;// make lookup element the current cell
lookupElem.commit(true);// force lookup now
```

dataSize ()

The **dataSize** method returns the size of the element data in bytes.

```
// Return the size of the data in the second element of the first cell of the last
// record of the current collection of the third document.
Document(3).currentCollection.Record(LAST).Cell(FIRST).Element(2).dataSize();

// Get the size of each element of the cell named "Names" of the current record of
the document "Player Roster".
theElements = Document("Player Roster").currentRecord.Cell("Names").Element(ALL);
theSizes = theElements.dataSize();
```

exists ()

The **exists** method verifies the existence an element.

```
// Check to see if the 99th element of the first cell exists in the current record
// of the current Document.
// This example uses implied containment.See "Containment" earlier in this chapter
// for more information.
exists = Cell(FIRST).Element(99).exists();
```

get ()

The **get** method gets the value of an element in the specified format.

```
// Get the value of the last element of the table cell named "Qty" of the current
// record of the document named "Purchase Detail".
theElement = Document("Purchase Detail").currentRecord.Cell("Qty").Element(LAST);
theValue = theElement.get();
```

set (value,[withCommit],[withLookup])

The **set** method sets the value of a cell.

Arguments for the set method

Argument	Description
value	The value to which the cell will be set.
withCommit	If `true`, the data is committed to the cell immediately and any check calculations or formatting options are triggered. If `false`, the data is not committed immediately. The default is `true`.
withLookup	Specifies whether or not a lookup is performed. If `true`, the lookup is performed unconditionally. If `false`, the lookup is ignored. If `null`, a lookup is performed only if the cell's value has changed. The default value for the withLookup argument is `null`. This argument is ignored if the specified cell is not a lookup cell. **Note**: If withLookup is specified (`true` or `false`), then withCommit must be `true` or `null`.

```
// Set the next available element of the cell named "Item Number" of the
// current record of the current document to its row number.
theCell = Informed.currentDocument.currentRecord.Cell("Item Number");
nextIndex = theCell.count(Element) + 1;
theCell.Element(nextIndex).set(nextIndex);

// Set the current element of the cell named "Phone" of the last record of the
// document named "Personnel" to "403" but don't commit it.
theElement = Document("Personnel").Record(LAST).Cell("Phone").currentElement;
theElement.set("(403)", false);

// Set the last element of the cell named "Employee Number" of the current record
of the document reference by theDoc to 350213 and force a lookup.
theElement = theDoc.currentRecord.Cell("Employee Number").Element(LAST);
theElement.set(350213, true, true);
```

File

A **File** object represents a file. For example, if you write a script to export records from a data document, you use the File object to specify the file that you're exporting the records to.

Reference

A File object can reference files by:

- name

Properties

No properties.

Methods

The following methods are defined for a File object:

open ([withRevisionCheck])

The **open** method opens a file and returns the resulting Document object.

Arguments for the open method

Argument	Description
withRevisionCheck	Specifies whether or not to do a revision check when opening the file. If `true`, this argument forces a revision check. If `false`, no check is performed. If `null`, the settings on the Revision Preferences panel take effect. The default value is `null`.

```
// Open the Macintosh file named "Temporary" without a revision check.
theTempDoc = File("HD:Informed:Data:Temporary").open(false);

// Open the Windows file named "payroll.ifm" and force a revision check.
thePayrollDoc = File("c:\\informed\\data\\payroll.ifm").open(true);

// Open the file named "Register" on either platform and perform a revision check
// only if it is required as per the Revision Preferences settings.
if (Informed.platform == WIN16 || Informed.platform == WIN32)
    registerFile = "c:\\informed\\data\\register.ifm";
else
    registerFile = "HD:Informed:Data:Register";
registerDoc = registerFile.open();
```

print ()

The **print** method prints the specified file. Informed always displays the standard Print dialog when this method is used on a File object.

```
// Print the file named "Schedule" on either platform.
if (Informed.platform == MACOS68K || Informed.platform == MACOSPPC)
    File("HD:Informed:Data:Schedule").print();
else
    File("c:\\informed\\data\\schedule.ifm").print();
```

Format

A **Format** object represents the current Record List format.

Reference

A Format object can reference formats by:

- index
- id
- absolute position
- test

Properties

The following table lists the properties of a Format object.

Format Properties

Property	Writeable?	Description
container	no	The container for the format. A format is always contained by a document.
id	no	The unique id for the format.
index	no	The index of the format.
objectClass	no	The Format class.
totalsVisible	yes	Are the totals for the columns visible?

Methods

The following methods are defined for a Format object:

count (elementClass)

The **count** method returns the number of columns in a format.

Arguments for the count method

Argument	Description
elementClass	This argument specifies the class of the elements to be counted. Its value must be Column.

```
// Count the number of columns in the current format of the second document.
theCount = Document(2).currentFormat.count(Column);
```

exists ()

The **exists** method verifies the existence of a format.

```
// Check for the existence of the current format in the Document named "Payroll".
ok = Document("Payroll").currentFormat.exists();
```

Informed

The **Informed** object represents the Informed application.

Reference

No reference required.

Properties

The following table lists the properties of the Informed object.

Informed Properties

Property	Writeable?	Description
currentDocument	yes	The current document displayed by the application.
frontmost	yes	Is this the frontmost application?
name	no	The name of the application.
objectClass	no	The Informed class.
platform	no	The operating system the application is running on. This property can be MacOS68K, MacOSPPC, Win16, or Win32.
registeredCompany	no	The name of the company the application is registered to.
registeredName	no	The name of the user the application is registered to.
serialNumber	no	The serial number of the application.
suppressUI	yes	Suppress the display of dialogs and error messages?
version	no	The version number of the application.

Methods

The following methods are defined for the Informed object:

count ([elementClass])

The **count** method returns the number of built-in commands, documents, menus, plug-in commands, templates, or windows within the Informed application.

Arguments for the count method

Argument	Description
elementClass	This argument specifies the class of the elements to be counted. Its value must be BuiltinCommand, Document, Menu, PluginCommand, Template, or Window

```
// Count the number of open documents.
documentCount = Informed.count(Document);
```

exists ()

The **exists** method verifies the existence of the Informed object.

```
// Check for the existence of the Informed object.
exists = Informed.exists();
```

make (elementClass, withData])

The **make** method creates a new document in Informed.

Arguments for the make method

Argument	Description
elementClass	This argument specifies the class of the new element. Its value must be Document.
withData	This argument specifies the data from which the new element will be created. Its value must be a Template object.

```
// Create a new document from the template with template id "Transact B308".
theTemplate = Template(id("Transact B308"));
theDoc = Informed.make(Document, theTemplate);
```

quit ([saving])

The **quit** method quits Informed.

Arguments for the quit method

Argument	Description
saving	Specifies whether to save changes before quitting. If `true`, changes will be saved. If `false`, changes will not be saved. If `null`, Informed will display a dialog asking if the user wants to save the changes. The default value is `null`.

```
// Prompt to save changes in each modified document and then quit Informed.
Informed.quit();

// Quit Informed without saving.
Informed.quit(false);

// Save changes in each modified document without asking and then quit Informed.
Informed.quit(true);
```

Menu

A **Menu** object represents one or more menus in Informed Filler's menu bar.

Reference

A Menu object can reference menus by:

- name
- index
- absolute position
- relative position
- range
- test

Properties

The following table lists the properties of a Menu object.

Menu Properties

Property	Writeable?	Description
enabled	yes	Is the menu enabled?
index	no	The index of the menu.
name	no	The name of the menu.
objectClass	no	The Menu class.

Methods

The following methods are defined for a Menu object:

count (elementClass)

The **count** method returns the number of menu items within a menu.

Arguments for the count method

Argument	Description
elementClass	This argument specifies the class of the elements to be counted. Its value must be MenuItem.

```
// Count the number of menu items in the "File" menu.
menuItemCount = Menu("File").count(MenuItem);
```

exists ()

The **exists** method verifies the existence of a menu.

```
// Get the name of the sixth menu if it exists.
theMenu = Menu(6);
if (theMenu.exists())
    theName = theMenu.name;
```

MenuItem

A **MenuItem** object represents one or more menu items in a particular menu.

Reference

A MenuItem object can reference menu items by:

- name
- index
- absolute position
- relative position
- range
- test

Properties

The following table lists the properties of a MenuItem object.

Menu Item Properties

Property	Writeable?	Description
container	no	The container for the menu item. A menu item is always contained by a menu.
enabled	yes	Is the menu item enabled?
index	no	The index of the menu item.
name	no	The name of the menu item.
objectClass	no	The MenuItem class.

Methods

The following methods are defined for a MenuItem object:

count (elementClass)

The **count** method returns the number of menu items within a menu item.

Arguments for the count method

Argument	Description
elementClass	This argument specifies the class of the elements to be counted. Its value must be MenuItem.

```
// Count the number of menu items in the menu item "Tags" of the menu "Database".
menuItemCount = Menu("Database").MenuItem("Tags").count(MenuItem);
```

execute ()

The **execute** method executes a menu item's configured action.

```
// Execute the "Submit" command of the "File" menu if it is enabled.
theMenuItem = Menu("File").MenuItem("Submit");
if (theMenuItem.enabled)
    theMenuItem.execute();
```

exists ()

The **exists** method verifies the existence of a menu item.

```
// Execute the menu item "Assign Next Value" of the "Cell" menu if it exists.
theMenuItem = Menu("Cell").MenuItem("AssignNextValue");
if (theMenuItem.exists())
    theMenuItem.execute();
```

PluginCommand

Some of Informed Filler's features are made available by installing Informed plug-ins. Certain plug-ins have commands associated with them. A **PluginCommand** object represents one or more plug-in commands in Informed Filler.

Reference

A PluginCommand object can reference plug-in commands by:

- name
- index
- id
- absolute position
- relative position
- range
- test

Properties

The following table lists the properties of a PluginCommand object.

PluginCommand Properties

Property	Writeable?	Description
enable	yes	Is the plug-in command enabled?
id	no	The unique id of the plug-in command.
index	no	The index of the plug-in command.
name	no	The name of the plug-in command.
objectClass	no	The PluginCommand class.

Methods

The following methods are defined for a PluginCommand object:

execute ([withData])

The **execute** method executes a plug-in command.

Arguments for the execute method

Argument	Description
withData	Specifies the data string required by the plug-in.

```
// If it exists, execute the plug-in command referenced by the variable
// thePluginCmd with the data referenced by the string variable theData.
if (thePluginCmd.exists())
    thePluginCmd.execute(theData);
```

exists ()

The **exists** method verifies the existence of a plug-in command.

```
// Set the variable ok to true if the plug-in command reference by the variable
// thePluginCmd exists.
ok = thePluginCmd.exists();
```

Record

In Informed Filler, completed forms are stored as records in a data document. A **Record** object represents one or more records in a data document.

Reference

A Record object can referenced records by:

- index
- id
- absolute position
- relative position
- range
- test

Properties

The following table lists the properties of a Record object.

Record Properties

Property	Writeable?	Description
attachmentCount	no	The number of attachments enclosed in the record.
container	no	The container for the record. A record is always contained by a document.
created	no	When the record was created.
id	no	The unique id of the record.
index	no	The index of the record.
lastMailed	no	When the record was last mailed.
lastModified	no	When the record was last modified.
lastPrinted	no	When the record was last printed.
modified	no	Has the record been modified?
objectClass	no	The Record class.
selected	yes	Is the record selected in the Record List?

Methods

The following methods are defined for a Record object:

clear ()

The **clear** method clears the record of any data.

```
// Clear the current record of the current document.
Informed.currentDocument.currentRecord.clear();
```

commit ()

The **commit** method commits the record data to the document.

```
// Commit the data of the current record of the document named "Transactions".
Document("Transactions").currentRecord.commit();
```

count ([elementClass])

The **count** method returns the number of attachments or cells within a record.

Arguments for the count method

Argument	Description
elementClass	This argument specifies the class of the elements to be counted. Its value must be Attachment or Cell.

```
// Count the number of attachments in the current record of the current document.
theCount = Informed.currentDocument.currentRecord.count(Attachment);
```

duplicate ()

The **duplicate** method duplicates a record.

```
// Duplicate the current record of the current collection of the first document.
theNewRecord = Document(FIRST).currentRecord.duplicate();
```

exists ()

The exists method verifies the existence of a record.

```
// Check if the third record of the first document exists.
exists = Document(FIRST).Record(3).exists();
```

export (toFile,[whichCells],[format],[doAppend],[rowwise],[useQuotes] [doMerge],[includeNotes])

The **export** method exports a record to a file.

Arguments for the export method

Argument	Description
toFile	This argument must be a File object which specifies the file into which the data will be exported.
whichCells	Specifies the cells which will be exported. This argument can be a single Cell object, an array of Cell objects, a single column, or an array of columns. The default value for whichCells is `null`, which specifies all cells. No container is allowed for this argument.
format	Specifies the file format of the export file. This argument can be one of the following constants: `INTERCHANGE`, `TAB_DELIMITED`, or `COMMA_DELIMITED`, or a string that specifies a format name. The default value for the format argument is `INTERCHANGE`.
doAppend	If `true`, the exported records are appended to the end of the export file. If `false`, the export file is replaced with the exported records. The default value is `false`.
rowwise	If `true`, then tables will be exported in row order. If `false`, tables will be exported in column order. The default value is `false`.

useQuotes	If `true`, then all exported values—except numbers—are surrounded with quotes. If `false`, then only those values which contain delimiter characters are surrounded with quotes. This argument is ignored if the format argument is not `TAB_DELIMITED` or `COMMA_DELIMITED`.
doMerge	If `true`, Informed will list each cell name on the first line of a new export file. The default value is `true`. This argument is ignored if the format argument is not `TAB_DELIMITED` or `COMMA_DELIMITED`.
includeNotes	If `true`, then any notes attached to the form will be exported. The default value is `true`. This argument is ignored if the format argument is not `INTERCHANGE`.

```
// Export all cells of the current record of the current document to the comma
// delimited file named "details.txt".
theRecord = Informed.currentDocument.currentRecord;
theRecord.export(File("c:\\details.txt"), null, COMMA_DELIMITED);

// Export the cells named "Name", "Address", and "Balance Owing" of every record
// of the document named "Accounts" whose cell "Balance Owing" is greater than 0 to
// the Informed Interchange file named "Receivables".
theTest = testGT(self.Cell("Balance Owing").value, 0);
theRecordList = Document("Accounts").Record(theTest);
theFile = File("HD:Receivables");
theCells = Cell(new Array ("Name", "Address", "Balance Owing"));
theRecordList.export(theFile, theCells, INTERCHANGE);
```

make (elementClass, withData)

The **make** method creates a new attachment in a record.

Arguments for the make method

Argument	Description
elementClass	The class of the new element. Its value must be Attachment.
withData	Specifies the data from which the new element will be created. Its value must be a File object.

```
// Create a new attachment for the third record of the current record of the
// current document from the file "Photo 1".
// This example uses implied containment.See "Containment" earlier in this chapter
// for more information.
Record(3).make(Attachment, File("HD:Photo 1"));
```

omit ()

The **omit** method omits a record from the current collection.

```
// Omit the first record from the current collection of the document referenced by
// the variable theDoc.
theDoc.currentCollection.Record(FIRST).omit();

// Omit every record of the current collection of the document "Receivables" whose
// cell "Last Payment" contains a value on or after January 1, 1998.
theTest = testGE(self.Cell("LastPayment").value, new LongDate(0, 1998, 0, 1));
Document("Receivables").currentCollection.Record(theTest).omit();
```

print ([as],[copies],[fromPage],[toPage],[fromPart],[toPart],[printTemplate], [printData],[collate])

The **print** method prints a specified record.

Arguments for the print method

Argument	Description
as	Specifies whether to print as forms or as a list. This argument can be either FORMS or RECORD_LIST. The default value is FORMS.
copies	Specifies the number of copies to print. The default value is 1.
fromPage	Specifies the page to start printing from. The default value is the first page.
toPage	Specifies the page to stop printing at. The default value is the last page.
fromPart	Specifies which part of a multipart form to start printing from. The default value is the first part.
toPart	Specifies which part of a multipart form to stop printing at. The default value is the last part.
printTemplate	If false, then don't print the template. The default value is true.
printData	If false, then don't print the data. The default value is true.
collate	Specifies whether or not to collate pages. The default value is true.

```
// Print the last record in the current collection of the current document as a
// single form.
Informed.currentDocument.currentCollection.Record(LAST).print();

// Print two copies of the third part of the current record in the first document.
Document(FIRST).currentRecord.print(null, 2, null, null, 3, 3);

// Print every record of the current collection of the current document whose cell
// "Company" contains the value "Shana" as a record list.
theTest = testCON(self.Cell("Company").value, "Shana");
theRecordList = Informed.currentDocument.currentCollection.Record(theTest);
theRecordList.print(RECORD_LIST);
```

remove ()

The **remove** method deletes a record from a document.

```
// Delete the current record of the current document.
Informed.currentDocument.currentRecord.remove();

// Delete every record of the current collection of the document referenced by
// theDoc whose cell "LastSubscribed" contains a date before December 31, 1995.
lastValidDate = new LongDate (0, 1995, 11, 31);
theTest = testLT(self.Cell("LastSubscribed").value, lastValidDate);
theDoc.currentCollection.Record(theTest).remove();
```

revert ()

The **revert** method restores a record to its last saved state.

```
// This command reverts the current record of the current document.
Informed.currentDocument.currentRecord.revert();
```

send ([recipients],[subject],[body],[format],[encloseAs], [messageAttachments])

The **send** method sends a record using an electronic mail service.

Arguments for the send method

Argument	Description
recipients	Specifies the recipients. This argument can be a string or an array of strings.
subject	Specifies the subject of the message. The default is the name of the document being sent.
body	The body of the mail message. The default value is `null` (no body).
format	The format to send the form in. This argument can be one of the following constants: `DATA`, `PACKAGE`, `INTERCHANGE`, `COMMA_DELIMITED`, or `TAB_DELIMITED`, or a string that specifies the name of a format. The default value is `DATA`.
encloseAs	Specifies the name of the form attachment. The default value is the name of the data document.
messageAttachments	Specifies any additional attachments. This argument can be a File object or an array of File objects.

```
// Send the current record of the document named "Requests" to the recipient
// "someone@someplace.com".
Document("Requests").currentRecord.send("someone@someplace.com");
```

```
// Send every record of the current document whose cell "FirstName" equals "Brent"
// and cell "LastName" equals "Taylor" with the provided addressing information.
test1 = testEQ(self.Cell("FirstName").value, "Brent");
test2 = testEQ(self.Cell("LastName").value, "Taylor");
theRecordList = Informed.currentDocument.Record(testAND(test1, test2));
theRecipients = "btaylor@worldcorp.com";
theSubject = "Hi there";
theBody = "Just called to say hi!";
theRecordList.send(theRecipients, theSubject, theBody);
```

sendExt ([usingStep],[recipients],[ccRecipients],[bccRecipients],
[appendRecipients],[subject],[body],[format],[encloseAs],
[messageAttachments],[appendAttachments],[mailSystem])

The **sendExt** method sends a record using an electronic mail service. This method provides extended options over the **send** method.

Arguments for the sendExt method

Argument	Description
usingStep	Specifies the name or the index of the routing step to use. The default value is `null` (no routing step). If a routing step is supplied, all other optional arguments override their corresponding settings in the routing step configuration.
recipients	Specifies the recipients. This argument can be a string or an array of strings. The default is `null` (no recipients).
ccRecipients	Specifies a list of recipients to cc. This argument can be a string or an array of strings. The default is `null` (no cc recipients).
bccRecipients	Specifies a list of recipients to bcc. This argument can be a string or an array of strings. The default is `null` (no bcc recipients).
appendRecipients	Specifies whether or not to append other recipients to those already specified in a routing step. The default value is `false`.
subject	Specifies the subject of the message. The default is the name of the document being sent.
body	The body of the mail message. The default value is `null` (no body).
format	The format to send the form in. This argument can be one of the following constants: `DATA`, `PACKAGE`, `INTERCHANGE`, `COMMA_DELIMITED`, or `TAB_DELIMITED`, or a string that specifies the name of a format. The default value is `DATA`.
encloseAs	Specifies the name of the form attachment. The default value is the name of the data document.

messageAttachments	Specifies any additional attachments. This argument can be a File object or an array of File objects. The default value is null (no additional attachments).
appendAttachments	Specifies whether or not to append other attachments to those already specified in a routing step. The default value is `false`.

mailSystem	This argument is illegal if a routing step is provided. If no routing step is provided, this argument can be a constant, a string that specifies the name of a mail plug-in, or null which calls the default mail system on the user's machine. Informed provides the following constants for mail systems that have the same name on both Windows and MacOS:
	`SMTP`, `EUDORA`, `MSMAIL`, `CCMAIL`, and `GROUPWISE`
	The following constants are for Windows only:
	`EXCHANGE, MAPI, VIM, MHS, MHSLOCAL`
	The following constants are for MacOS only:
	`QUARTERDECK, QUICKMAIL`

```
// Send the current record of the document named "Summary" as specified by the
// third routing step, except send it to "someone@worldcorp.com" instead of to the
// specified recipient.
theRecord = Document("Summary").currentRecord;
theRecord.send (3, "someone@worldcorp.com", null, null, false);
```

sendStep ([usingStep])

The **`sendStep`** method sends a record using a preconfigured routing step.

Arguments for the sendStep method

Argument	Description
usingStep	Specifies the name or the index of the routing step to use. The default value is `null` (no routing step). If a routing step is supplied, all other optional arguments override their corresponding settings in the routing step configuration.

```
// Send the last record of the current collection of the document named "Invoices"
// using the routing step named "Submit Invoice".
Document("Invoices").currentCollection.Record(LAST).sendStep("Submit Invoices");
```

SavedFormat

Informed Filler's Record List window can be customized to display data in a variety of different formats. A **SavedFormat** object represents one or more saved Record List formats.

Reference

A SavedFormat object can reference saved formats by:

- name
- index
- id
- absolute position
- relative position
- range
- test

Properties

The following table lists the properties of a SavedFormat object.

SavedFormat Properties

Property	Writeable?	Description
container	no	The container for the saved format. A saved format is always contained by a document.
id	no	The unique id for the saved format.
index	no	The index of the saved format.
name	no	The name of the saved format.
objectClass	no	The SavedFormat class.

Methods

The following methods are defined for a SavedFormat object:

exists ( )

The **exists** method verifies the existence of a saved format.

```
// Set the current format of the first document to the saved format
// "QuickCheck Format" if it exists.
theDocument = Document(1);
theSavedFormat = theDocument.SavedFormat("QuickCheck Format");
if (theSavedFormat.exists())
    theDocument.currentFormat = theSavedFormat;
```

remove ()

The **remove** method deletes a saved format from a document.

```
// Delete all saved formats from the document named "Tax Form".
Document("Tax Form").SavedFormat(ALL).remove();
```

Tag

Informed Filler's Tag feature provides an easy way to identify unique collections of records so that they can be quickly recalled and viewed. A **Tag** object represents one or more tags in a document.

Reference

A Tag object can reference tags by:

- name
- index
- id
- absolute position
- relative position
- range
- test

Properties

The following table lists the properties of a Tag object.

Tag Properties

Property	Writeable?	Description
container	no	The container for the tag. A tag is always contained by a document.
id	no	The unique id for the tag.
index	no	The index of the tag.
name	no	The name of the tag.
objectClass	no	The Tag class.

Methods

The following methods are defined for a Tag object:

exists ()

The exists method verifies the existence of a tag.

```
// Set the current collection of the current document to the tag named "California
// Invoices" if it exists.
theDoc = Informed.currentDocument;
theTag = theDoc.Tag("California Invoices");
if (theTag.exists())
    theDoc.currentCollection = theTag;
```

remove ()

The **remove** method deletes a tag from a document.

```
// Delete the tag named "Invoices" from the first document.
Document(1).Tag("Invoices").remove();

// Delete every tag in the current document.
// This example uses implied containment.See "Containment" earlier in this chapter
// for more information.
Tag(ALL).remove();
```

Template

A **Template** object represents the form template used by a data document.

Reference

A Template object can reference templates by:

- name
- index
- id
- absolute position
- relative position
- range
- test

Properties

The following table lists the properties of a Template object.

Template Properties

Property	Writeable?	Description
diskFile	no	The disk file that contains the template
id	no	The template id for the template as entered on the Template Information dialog.
index	no	The index of the template.
name	no	The template name of the template as entered on the Template Information dialog.
objectClass	no	The Template class.
revision	no	The revision number of the template as entered on the Template Information dialog.

Methods

The following methods are defined for a Template object:

exists ()

The **exists** method verifies the existence of a template.

```
// Create a new document from the template with template id "Timesheet" if it
// exists.
theTemplate = Template(id("TimeSheet"));
if (theTemplate.exists())
    theDocument = Informed.make(Document, theTemplate);
```

Window

A **Window** object represents a window in Informed Filler.

Reference

A Window object can reference windows by:

- name
- index
- id
- absolute position
- relative position

- range
- test

Properties

The following table lists the properties of a Window object.

Window Properties

Property	Writeable?	Description
bounds	yes	The boundary rectangle for the window.
closeable	no	Does the window have a close box?
floating	no	Is the Window a floating window?
id	no	The unique ID of the window.
index	yes	The number of the window.
kind	no	The window kind. This property can be one the following constants: `ATTACHMENTS_KIND`, `CLIPBOARD_KIND`, `FORM_KIND`, or `RECORD_LIST_KIND`.
modal	no	Is the window modal?
name	no	The title of the window.
objectClass	no	The class of object.
parentDocument	no	The document to which the window belongs.
resizable	no	Is the window resizable?
titled	no	Does the window have a title bar?
visible	yes	Is the window visible?
zoomable	no	Is the window zoomable?
zoomed	yes	Is the window zoomed?

Methods

The following methods are defined for a Window object:

close `([saving],[savingIn])`

The **close** method closes a window.

Arguments for the close method

Argument	Description
saving	Specifies whether changes should be saved before closing. If `true`, changes will be saved. If `false`, changes will not be saved. If `null`, Informed will display a dialog asking if the user wants to save the changes. The default value is `null`. This argument is ignored if the window is not a form window.
savingIn	Specifies the file in which to save the window. The default value is the file into which the window's parent document was previously saved. It the document was not previously saved, the standard Save dialog is displayed. This argument is ignored if the window is not a form window.

```
// Close the frontmost window and prompt to save (if it is a form window).
Window(FIRST).close();

// Close the record list window of the current document.
Informed.currentDocument.recordListWindow.close();

// Close the form window referenced by the variable theFormWind without saving.
theFormWind.close(false);

// Close the form window referenced by the variable theFormWind and save to the
// file "NewFile".
thePlatform = Informed.platform;
if (thePlatform == MACOS68K || thePlatform == MACOSPPC)
    thePath = "HD:Informed:Data:NewFile";
else
    thePath = "c:\\informed\\data\\newfile.ifm";
theFormWind.close(true, File(thePath));
```

exists ()

The **exists** method verifies the existence of a window.

```
// Check for the existence of a window named "Invoice".
invoiceWindowExists = Window("Invoice").exists();
```

open ()

The **open** method opens a window and returns the resulting Window object.

```
// Open the attachments window of the second document.
theWindow = Document(2).attachmentsWindow.open();
```

Additional Built-in Objects

Informed's JavaScript implementation provides several additional built-in objects to specify data required by Informed.

LongDate Object

The Date object is a built-in object which provides system-independent dates and times. Unfortunately, the Date object only supports date and time values since January 1, 1970. Therefore, JavaScript for Informed provides a LongDate object which supports the entire range of data and time values required by Informed.

The LongDate object supports virtually the same set of constructors as the Date object except for an additional argument which specifies the era (0 = AD, -1 = BC). The LongDate object also provides methods for getting and setting the era.

The value of an Informed cell can be set to either a Date object or a LongDate object. The value obtained from an Informed cell is always a LongDate object.

For example, the following script sets the value of a cell to the date "November 6, 1960 AD".

```
theDate = new LongDate (0, 1960, 11, 6);
theCell . value = theDate;
```

Communicating with Other Applications

An important feature of Informed's scripting functionality is the ability to integrate with other applications. By controlling different applications, a single script can effectively combine different features from different products to provide more powerful solutions. For example, you can write a script which instructs Informed Filler to collect data from a collection of records, chart the data using a spreadsheet application, then insert the results into a letter using a word processor.

On Mac OS, this type of functionality is available through AppleScript. On Windows, Informed provides two Windows-only built-in objects—Application and DDE—which allow a script to communicate with another Windows application using DDE (Dynamic Data Exchange).

Application Object

The **Application** object is a Windows-only built-in object which can be used to launch and terminate another application.

The constructor for the Application object requires a single string argument which contains the command line (filename plus optional parameters) for the application to be launched. If the argument does not contain the full path for the application, Windows searches the directories in the following order.

- the current directory
- the Windows directory
- the directory from which Informed Filler was launched
- the directories listed in the PATH environment variable
- the directories mapped in a network

Methods

The following methods are defined for the Application object.

launch ()

The **launch** method launches the application specified by the constructor argument and internally stores a reference to the application.

```
// launch Excel
theApp = new Application("Excel");
theApp.launch();
```

terminate ()

The **terminate** method uses the internal reference to the application to terminate it. Therefore, an Application object cannot be used to terminate an application that it did not launch.

```
// terminate Excel
theApp.terminate();
```

DDE Object

The **DDE** object is a Windows-only built-in object which can be used to open a DDE conversation with a DDE server application. The conversation protocol is application specific, so you must provide those values required by the server application.

The constructor requires two arguments which represent the application and the topic for the desired DDE conversation.

Methods

The following methods are defined for the DDE object.

connect ( )

The **connect** method is used to open a DDE conversation. The DDE server must be running in order to open the conversation. Use the Application object to launch the DDE server if necessary.

request ( )

The **request** method requests a value from a DDE server. The first argument is the item for the DDE transaction. The second argument is the timeout value measured in milliseconds. The timeout argument is optional. All other arguments are required. If successful, the method returns the result of the request.

poke ( )

The **poke** method passes a value to a DDE server. The first argument is the item for the DDE transaction. The second argument is the value for the DDE transaction. The third argument is the timeout value measured in milliseconds. The timeout argument is optional. All other arguments are required.

execute ( )

The **execute** method passes a command to a DDE server. The first argument is the command for the DDE transaction. The second argument is the timeout value measured in milliseconds. The timeout argument is optional. All other arguments are required.

disconnect ( )

The **disconnect** method is used to terminate the DDE conversation.

The following script uses the Application and DDE objects to exchange data between Informed Filler and Microsoft Excel.

```
// launch Excel
theApp = new Application("Excel");
theApp.launch();

// open the DDE conversation
theDDE = new DDE("Excel", "Sheet1");
theDDE.connect();

// exchange the data
Cell("Cost").value = theDDE.request("R1C1");
theDDE.poke("R1C2", Cell("Price").value);
```

```
// close the DDE conversation
theDDE.disconnect();

// terminate Excel
theApp.terminate();
```

Error Handling

The JavaScript language does not currently provide any form of exception handling. Since most Informed reference object methods can potentially return an error, Informed provides built-in functions which allow these errors to be handled intelligently.

Standard Behaviour

Normally, if an error occurs while executing a script, a dialog with the appropriate error message is displayed and the script terminates at the line on which the error occurred.

For example, if the file “Invoice.ifm” is missing, the following script will display an error dialog and the script will terminate on the second line. The third line of the script will not be executed.

```
theFile = new File("c:\\informed\\data\\Invoice.ifm");
theCoc.currentCollection = theDoc.Record(ALL);
theFile.open;
```

Suspending Errors

The **suspendErrors** built-in function is used to turn error suspension on or off. If an error occurs while error suspension is on, the error dialog is not displayed and execution of the script continues. The suspendErrors function requires a boolean argument which indicates whether error suspension is to be turned on or off. The suspendErrors function also returns a boolean result which indicates the previous error suspension state. The error suspension state for any script is false by default.

The **getLastError** built-in function is used to obtain error information about the most recent operation which may have produced an error. If no error occurred, getLastError returns null. If an error did occur, getLastError returns a built-in Error object whose code and message properties may be inspected to obtain the error code and error message generated by the error, respectively.

For example, the following script attempts to open a file. If an error occurs, the error message is stored into the cell “Error”.

```
// turn error suspension on and remember the old state
oldState = suspendErrors(true);
```

```
// open the file
theFile = new File("Invoice.ifm");
theFile.open;

// handle the error
theError = getLastError ();
if (theError) {
    Cell(”Error code”).value = theError.code;
    Cell (”Error message”).value = theError.message;
return;
}

// restore the error suspension state
suspendErrors (oldState);

// proceed with the script
theDoc.currentCollection = the Doc.Record(ALL);
```

Sample Scripts

The following sample scripts are provided to show you how JavaScript can be used to automate tasks that are performed frequently by Informed Filler users.

Working with Documents

Opening a Document

This script opens a specific document:

```
var theFile, theDocument;

theFile = new File("HD:Test Form");
theDocument = theFile.open();
```

Closing a Document

This script closes the first document:

```
// Close the first document
Document(1).close();
```

This script closes all open documents:

```
// Close every document
Document(ALL).close();
```

This script closes and saves a specific document:

```
// Close and save the document named "Inventory".
Document("Inventory").close(true);
```

Saving a Document

This script saves a document:

```
// Save the document named "Samples".
Document("Samples").save();
```

This script saves a document into a particular file using the package data format:

```
// Save the current document as a package in "c:\\summary.ipk".
Informed.currentDocument(File("c:\\summary.ipk"), PACKAGE);
```

Working with Records

Making a New Record

This script makes a new record in the current document:

```
// Make a new record in the current document
Document(1).make(Record);
```

This script uses the "make" method to return a reference to the new record object, and then uses that reference to manipulate the new record:

```
var theRec;

theRec = Document(1).make(Record);
theRec.Cell("Name").value = "Fred";
theRec.Cell("Era").value = "Jurassic";
```

Setting a Collection of Records

This script sets the current collection to all records in the document:

```
// Set the collection to all records
Document(1).currentCollection = Document(1).Record(ALL);
```

This script sets the current collection to a group of records that match a test. In this sample, the script tests for all records where the "Salary" cell is greater than 35000:

```
// Set the collection to some records that match a test
theTest = testGT(self.Cell("Salary").value, 35000);
Document(1).currentCollection = Document(1).Record(theTest)
```

Counting Records

This script counts the number of records in the current collection:

```
// Count the number of records in the current collection

var count

count = Document(1).currentCollection.count(Record);
```

This script counts the number of records in a document, including records that are not part of the current collection:

```
count = Document(1).count(Record);
```

This script sets the current collection to all records in the document, then counts the total number of records:

```
// Set the collection to "every record", then count.

Document(1).currentCollection = Record(ALL);
count = Document(1).count(Record);
```

Deleting Records

This script deletes the current record:

```
// Delete the current record
Document(1).currentRecord.remove();
```

This script deletes a specific range of records:

```
// Delete the first three records
Document(1).currentCollection.Record(1,3).remove();
```

This script deletes all records that match a specific test. In this sample, the test is all records where the "Name" cell is equal to "Fred".

```
// Delete records that match a test
theTest = testEQ(self.Cell("Name").value, "Fred");
Document(1).currentCollection.Record(theTest).remove();
```

Duplicating Records

This script duplicates the current record:

```
// Duplicate the current record
Document(1).currentRecord.duplicate();
```

Omitting Records

This script omits a specific record from the current collection:

```
// Omit a specific record
Document(1).currentCollection.Record(1).omit();
```

This script omits all records that match a test. In this case, the script tests for all records where the "City" cell equals "New York."

```
// Omit records that match a test
theTest = testEQ(self.Cell("City").value, "New York");
Document(1).currentCollection.Record(theTest).omit();
```

Committing a Record

This script commits any changes to the current record. This is equivalent to the user pressing the “Enter” key:

```
// Commit any changes to the current record.
Document(1).currentRecord.commit();
```

Reverting a Record

This script reverts a record to its last saved state:

```
// Revert any changes to the current record
Document(1).currentRecord.revert();
```

Looping Through Records

This script counts the number of records in a collection, then loops through each record and finds the grand total of all the "Total" cells:

```
// Process each record
var i, n, theTotal;

n = Document(1).currentCollection.count (Record);
theTotal = 0.0;
for (i = 1; i <= n; i++) {
   theTotal += Document(1).currentCollection.Record(i).Cell("Total").value;
}
```

Exporting Records

This script exports the current document using the tab delimited data format:

```
// Export document 1 as tab delimited text

var theFile;

theFile = new File("HD:Data File");
Document(1).export(theFile, null, TAB_DELIMITED);
```

This script exports a single record into a file:

```
// Export the current record to a file

Document(1).currentRecord.export(File("HD:Test.iif"));
```

This script searches for any records where the amount in the "Overdue Amt" cell is greater than zero, and exports those records into a file named "Deadbeats.iif".

```
// Export some specific records
theTest = testGT(self.Cell("Overdue Amt").value, 0);
Document(1).currentCollection.Record(theTest).export(File ("HD:Deadbeats.iif");
```

Importing Records

This script imports a specific file into a document:

```
// Import some data

Document(1).collect(File("HD:Data File"));
```

This script moves data from one document to another by using the export and collect methods:

```
// set the collection of document 1 to every record
// with an overdue amount:
//
var doc1 = Document(1);
var deadbeats = doc1.Record(testGT(self.Cell("Overdue Amt").value, 0));

if (deadbeats.exists ())
{
   theFile = File ("HD:Deadbeats.iif");
   doc1.currentCollection = deadbeats;
   doc1.currentCollection.export(theFile);
   Document("Deadbeats").collect(theFile);
}
```

Printing

This script prints a document:

```
// Print everything
Document(1).print();
```

This script prints the current collection of records as a list rather than forms:

```
// Print the current collection as a list
Document(1).currentCollection.print(RECORD_LIST);
```

This script prints only the data (no template) from the current collection of records:

```
// Print the current collection onto pre-printed forms.
// (print the data only)
//
Document(1).currentCollection.print(FORMS, null, null, null, null, null, false,
true);
```

Working with Cells

Setting a Cell's Value

This script sets the current cell's value to a blank value:

```
// Clear the current cell of the current document.
Informed.currentDocument.currentCell.set("");
```

This script sets the values for multiple cells:

```
// Set a list of cells to a list of values.
cellNames = new Array ("Company Name", "Phone Number");
theCells = Document(2).currentRecord.Cell(cellNames);
theData = new Array ("Shana Corporation", "(403) 433-3690");
theCells.set(theData);
```

Clearing a Cell

This script clears the value of one cell in a single record:

```
// Clear the cell named "Signature" of the current record of the document named
// "Authorization".
Document("Authorization").currentRecord.Cell("Signature").clear();
```

This script clears the value of the same cell of every record in a collection:

```
// Clear the cell named "Signature" of every record of the current collection of
// the document named "Authorization".
Document("Authorization").currentCollection.Cell("Signature").clear();
```

Getting a Cell's Value

This script gets the value of a specific cell in the current record:

```
// Get the value of the cell named "Signed Date" of the current record of the
// frontmost document.
theValue = Document(FIRST).currentRecord.Cell("Signed Date").get();
```

This script gets the value of a table cell:

```
// Get the value of the table cell named "Description" of the current record of the
// current document. The result is an array.
theValueList = Informed.currentDocument.currentRecord.Cell("Description").get();
```

Copying Cell Values Between Documents

This script copies a group of cell values from one document to another. It reads the values into local variables, then writes them to the other document:

```
function Copy ()
{
   var theName, theAddress, theCity, theZip;
   var rec;

   rec = Document("Employees").currentRecord;

   theName = rec.Cell("Name").value;
   theAddress = rec.Cell("Address").value;
   theCity = rec.Cell("City").value;
   theZip = rec.Cell("Zip").value;

   rec = Document("Holidays").currentRecord;

   rec.Cell("Name").value = theName;
   rec.Cell("Address").value = theAddress;
   rec.Cell("City").value = theCity;
   rec.Cell("Zip").value = theZip;
}
```

This script makes an array of cell names, then copies all the values at once into the other document:

```
function Copy ()
{
   var cellNames, cellValues;
   var rec;

   cellNames = new Array("Name", "Address", "City", "Zip");

   rec = Document("Employees").currentRecord;
   cellValues = rec.Cell(cellNames).value;
   rec = Document("Holidays").currentRecord;
   rec.Cell(cellNames).value = cellValues;
}
```

Signing Cells

This script signs a signature cell using the Entrust signing system:

```
// Sign the signature cell named "Signature" of the current record of the
// document named "Authorization" using the Entrust signing system.
Document("Authorization").currentRecord.Cell("Signature").sign(ENTRUST);
```

Working with Attachments

Making a New Attachment

This script makes a new attachment for a specific record:

```
// Create a new attachment for the third record of the current record of the
// current document from the file "Photo 1".
Record(3).make(Attachment, File("HD:Photo 1"));
```

Removing Attachments

This script removes every attachment from every record in the collection:

```
// Remove every attachment from every record of the current collection of the
// document referenced by the variable theDoc.
theDoc.currentCollection.Attachment(ALL).remove();
```

Saving Attachments

This script saves all files attached to the current record:

```
// Save every attachment of the current record of the document named "Submissions"
// with its own name into the directory "c:\submit\".
theRec = Document("Submissions").currentRecord;
for (i = 1; i <= theRec.count(Attachment); i++) {
    theAttachment = theRec.Attachment(i);
    theAttachment.save(File("c:\submit\\" + theAttachment.name));
}
```

Quitting Informed

This script quits the Informed application:

```
Informed.quit();
```