# JACK Intelligent Agents®
# JACOB Manual

JACK® AUTONOMOUS SOFTWARE

A|O|S™

## Publisher Information

Agent Oriented Software Pty. Ltd.
P.O. Box 639,
Carlton South, Victoria, 3053
AUSTRALIA

| | |
|---|---|
| **Phone:** | +61 3 9349 5055 |
| **Fax:** | +61 3 9349 5088 |
| **Web:** | http://www.agent-software.com |

If you find any errors in this document or would like to suggest improvements, please let us know.

## C++ Platforms

The JACOB libraries to interface with C++ applications are not delivered as part of the JACK product and must be purchased separately. The C++ libraries are available precompiled for the following list of platforms. Other platforms can be made available on request.

| Operating System | Compiled With |
|---|---|
| `Linux` | g++ 2.95 |
| `Linux` | g++ 3.0 |
| `Windows 9x` | VC++ 6.0 or greater (static/dynamic/multi-threaded) |
| `Windows NT/2000/XP` | VC++ 6.0 or greater (static/dynamic/multi-threaded) |

JACOB C++ libraries

The JACK™ documentation set includes the following manuals and practicals:

| **Document** | **Description** |
|---|---|
| `Agent Manual` | Describes the JACK programming language and infrastructure. JACK can be used to develop applications involving BDI agents. |
| `Teams Manual` | Describes the JACK Teams programming language extensions. JACK Teams can be used to develop applications that involve coordinated activity among teams of agents. |
| `Development Environment Manual` | Describes how to use the JACK Development Environment (JDE). The JDE is a graphical development environment that can be used to develop JACK agent and team-based applications. |
| `JACOB Manual` | Describes how to use JACOB. JACOB is an object modelling language that can be used for inter-process transport and object initialisation. |
| `WebBot Manual` | Describes how to use the JACK WebBot to develop JACK enabled web applications. |
| `Design Tool Manual` | Describes how to use the Design Tool to design and build an application within the JACK Development Environment. |
| `Graphical Plan Editor Manual` | Describes how to use the Graphical Plan Editor to develop graphical plans within the JACK Development Environment. |
| `JACK Sim Manual` | Describes how to use the JACK Sim framework for building and running repeatable agent simulations. |
| `Tracing and Logging Manual` | Describes the tracing and logging tools available with JACK. |
| `Agent Practicals` | A set of practicals designed to introduce the basic concepts involved in JACK programming. |
| `Teams Practicals` | A set of practicals designed to introduce the basic concepts involved in Teams programming. |

# Table of Contents

# 1 Introduction

The JACOB™ Object Modeller (JACOB) is a system providing machine and language independent object structures that can be stored or transmitted.

JACOB provides distributed objects in a similar way to CORBA, COM or Java object serialisation but the emphasis is on simplicity, portability, efficiency and low overheads.

Language specific code can be generated from JACOB object structures, allowing code in the target language to create, manipulate, send and receive JACOB objects easily.

JACOB consists of four major components:

- Compiler
- Writer
- Reader
- Initialisation

These components are explained in the *Functional Components of JACOB* chapter of this document.

## 1.1 Uses of JACOB

JACOB is useful in situations that require data structures that are machine and language independent. Two important uses of JACOB are:

- as an object initialisation tool
- as an object communication tool.

### 1.1.1 Object Initialisation

JACOB can be used to initialise objects within an application. JACOB-generated code together with user written code allows applications to read and/or write objects to and from a data file or stream.

An example of object initialisation can be found in the *Sample Applications* chapter.

### 1.1.2 Object Communication

JACOB may be used to allow communication between different processes. JACOB-generated code together with user written code allows populated objects to be sent and received between different processes.

For example, JACOB may be used to transport data between a Java class and a C++ class, or a JACK Agent and a C++ class. Populated objects are sent and received between the processes via a socket. JACOB code is inserted into each class to allow objects to be sent and/or received.

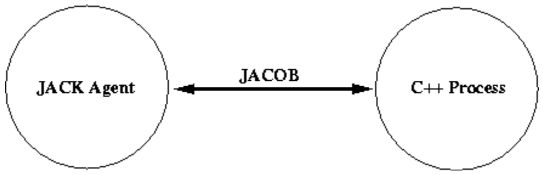


**Figure 1-1:** Communication between a JACK Agent and a C++ process using JACOB

An example of transporting objects can be found in the Sample Applications section below.

## 1.2 Dictionary Files

JACOB object structures that are stored and transported contain objects that are defined in dictionary files, using the JACOB Data Definition Language. A dictionary file defines the structure and legal fields of one or more objects, and usually has an `.api` extension.

**Note:** A dictionary associated with a data file may consist of a one or more dictionary files.

## 1.3 Compiling Dictionaries

Once one or more objects are defined in a dictionary file, the dictionary file is compiled using JACOB Build. JACOB Build can generate either Java or C++ code (Java code is generated by default). Once code is generated, the resulting classes can then be compiled and used for object initialisation and transportation.

## 1.4 Building Object Structures

JACOB object structures may be created using objects defined in a dictionary, and stored in a data file. Object structures can be created and edited by opening a data file in the JACOB Object Browser, which provides a graphical environment, or a text editor.

JACOB object structures may be used to store objects in applications. The types of objects allowed in an application are defined in a dictionary. The objects used in the application may be stored in one or more data files, which can be edited in the JACOB Object Browser.

## 1.5 JACOB File Formats

Data files containing JACOB object structures may currently be in one of four formats:

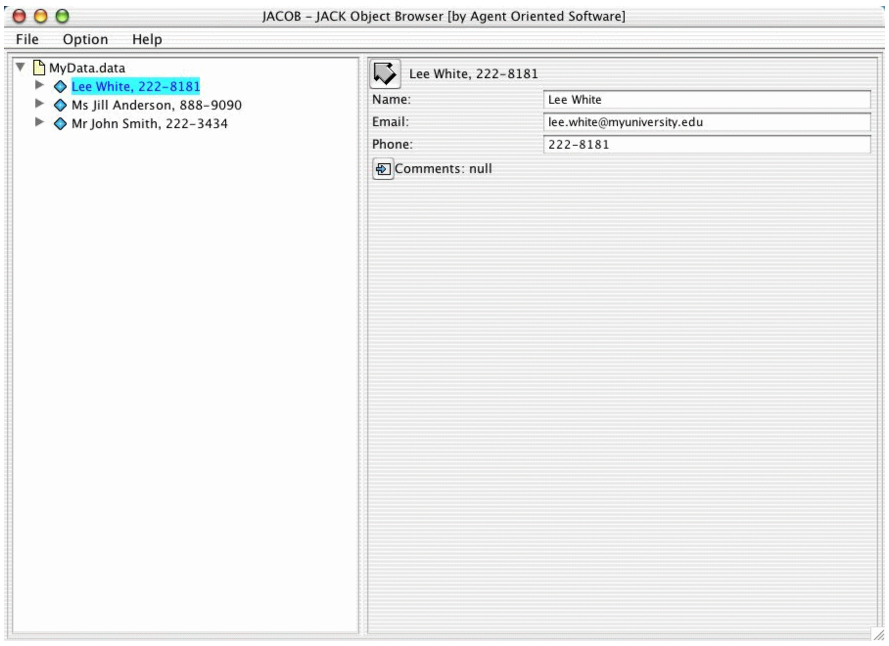
- ASCII
- Binary
- JDBC
- XML

Data files may be converted to another format using the Convert option in JACOB Build. Note that currently only ASCII and Binary formats can be used with C++ code.

JACOB dictionary files can be compiled to generate Java or C++ code. Code in other languages may be generated if a new driver is defined for that language.

## 1.6 The JACOB Object Browser

The JACOB Object Browser provides a graphical environment which allows users to view or edit JACOB object structures contained in data files. Data files that contain objects defined according to a dictionary file may be created, edited and saved. Objects can be edited by opening a data file and loading the associated dictionary file. New objects, object copies and object references may be added to a data file. The JACOB Object Browser may also be used to delete objects from a data file.



MyData.data
- Lee White, 222-8181
- Ms Jill Anderson, 888-9090
- Mr John Smith, 222-3434

Lee White, 222-8181

Name: Lee White
Email: lee.white@myuniversity.edu
Phone: 222-8181
Comments: null

**Figure 1-2:** The JACOB Object Browser Window

## 1.7 Using JACOB

To use JACOB to initialise and/or transport data structures, the user must first define the objects in the structure in one or more dictionary files. Objects and their fields are defined in dictionary files using the JACOB Data Definition Language.

After defining objects in one or more dictionary files, the dictionary files must be compiled with JACOB Build.

### 1.7.1 JACOB Objects

JACOB dictionary files contain definitions of objects and their fields. The contents of a data file or data stream may only include objects of the types that are defined in one or more dictionary files.

An example object `MyData` is defined below.

```
<Class :name "MyData"
    :fields (
        <Field :name "objectID" :type :int>
        <Field :name "objectName" :type :string>
        <Field :name "description" :type :string>
    )
>
```

A corresponding `MyData` object in a data file would be :

```
<MyData
    :objectID 12
    :objectName "ExampleObject"
    :description "An example of a MyData object"
>
```

## 1.7.2 Using JACOB to transport objects

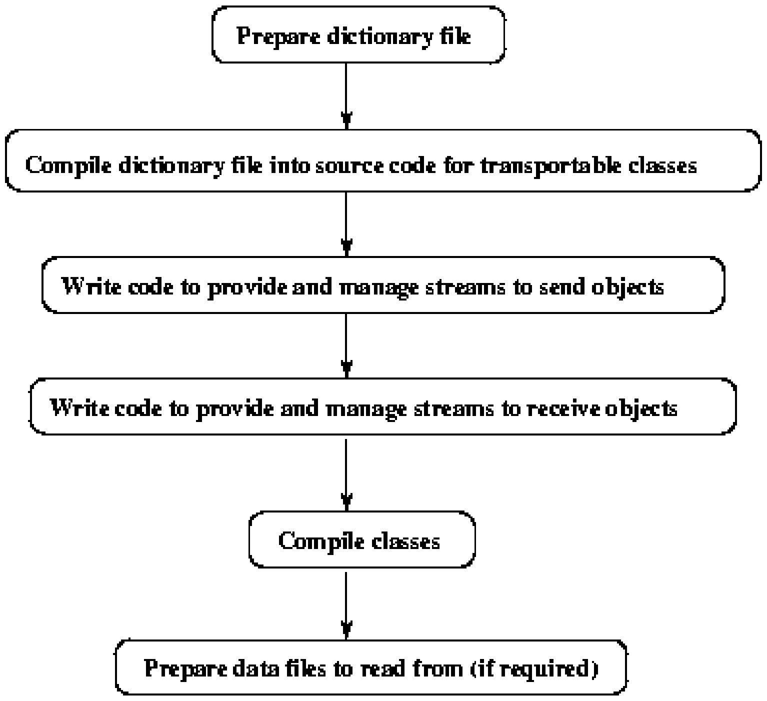

Prepare dictionary file
Compile dictionary file into source code for transportable classes
Write code to provide and manage streams to send objects
Write code to provide and manage streams to receive objects
Compile classes
Prepare data files to read from (if required)

**Figure 1-3:** Development steps for JACOB object transportation

To use JACOB for communication between processes, JACOB-generated code together with user written code must be used to transport JACOB objects. A dictionary written in the JACOB Data Definition Language is compiled into source code for transportable classes using JACOB Build. Either Java or C++ source code may be generated. The source code is then compiled into classes with a standard Java or C++ compiler.

The transportable classes generated are able to read and write object structures to and from streams. Additional code must be written by the user to provide and manage streams for transporting objects. The streams used may be files or sockets, or a JDBC connection (for Java only). The streams may also be any other kind of byte stream interprocess communication infrastructure.

To send objects, code written by the user must be included in either existing or new classes. The code must decide which stream to use, and then use it to send objects.

Code written by the user for receiving objects must also be included in existing or new classes. The code for receiving JACOB objects must decide which dictionary to use (a dictionary may include definitions from multiple dictionary files). The code must also construct and initialise the chosen dictionary. Once a dictionary has been chosen, the code must decide which stream and associate it with the dictionary. The stream may then be used to receive objects.

Once the classes containing the user-written sender and receiver code have been compiled, they can be used to send and receive objects.

# 2 Packages

Each JACOB component is installed automatically during the installation of JACK. The JACOB components can be found in the following locations:

| Component | Location |
|---|---|
| Runtime | `aos.apib.*` |
| Compiler | `aos.main.JacobBuild` |
| Browser | `aos.main.Jacob` |

**Table 2-1:** JACOB Packages

# 3 JACOB Object Browser

## 3.1 Introduction

The general form of the command to start the browser is

```
 java aos.main.Jacob [[<data-file>]* [-t <dict-file>]*]*
```

where `<data-file>` is a data file or a collection of data files, and `<dict-file>` is a dictionary or multiple dictionaries associated with the preceding data file(s). If the command ends with one or more data files, the last collection of data files will be associated with the preceding dictionary set (if any). The JACOB Object Browser may also be started with multiple groups of data file collections with associated dictionaries.

Note that both `<data-file>` and `<dict-file>` are optional. The browser can also be started as

```
  java aos.main.Jacob
```

## 3.2 The Browser Environment

Objects defined in the JACOB Object Modelling language can be viewed and modified using the JACOB Object Browser. Each object structure consists of two parts:

- a *dictionary* which defines the structure and legal fields in an object hierarchy; and
- a *data file* which contains objects defined according to this dictionary.

### 3.2.1 Menu Bar

The JACOB editor has a menu bar at the top of the work area. The following menus are available from the menu bar.

#### 3.2.1.1 File Menu

Use this menu to perform operations on the current project.

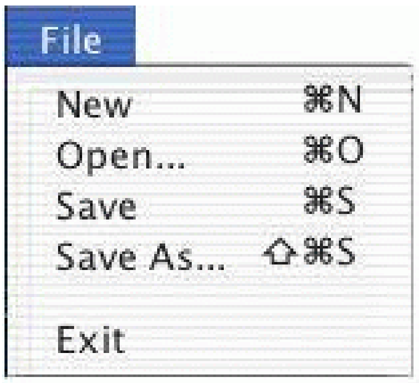


**Figure 3-1:** The JACOB Object Browser File Menu

- **New**: Creates a new data file in the editor. Initially this data file is empty with no structure. This file can be populated by first adding a dictionary (`.api`) to it and adding entities defined in the dictionary as required.
- **Open**: Loads an existing data file from disk. Until a dictionary (`.api`) file is added to this data file, it is unstructured and cannot be viewed. Adding a dictionary file is described in a later section.
- **Save**: Saves the current data file to disk under the current name.
- **Save As**: Saves the current data file to disk, allowing a new name to be specified.
- **Exit**: Quits the JACOB Editor. If the project currently open in the editor is unsaved, the option to save is given before quitting.

### 3.2.1.2 Option Menu

Use this menu to change JACOB Editor options.

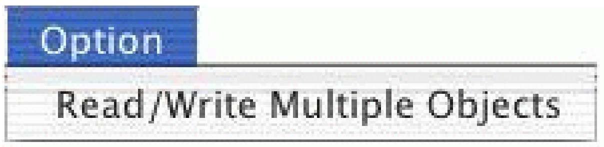


**Figure 3-2:** The JACOB Object Browser Option Menu

- **Read/Write Multiple Objects**: Turns the Read/Write Multiple Objects option on or off. This option is off by default.

When this options is on, a list of multiple objects with unique identity numbers is written. A list of multiple objects that is read or written may have references between the objects. Note that this option is not supported for binary files.

**Note:** The best way to write multiple objects is to wrap them in a parent container.

#### 3.2.1.3 Help Menu

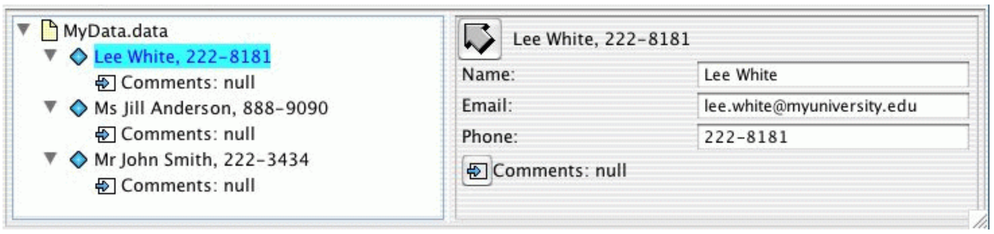
Use this menu to access the About JACOB screen.

**Figure 3-3:** The JACOB Object Browser Help Menu

- **About**: Displays information about the JACOB Object Browser in the work area of the browser.

### 3.2.2 Work Area

Objects and their contents are manipulated in the *work area* of the JACOB browser. The work area is depicted below.



Name: Lee White
Email: lee.white@myuniversity.edu
Phone: 222-8181

**Figure 3-4:** The JACOB Object Browser Work Area

The work area consists of two *panes*: the Object Tree pane and the Object Field pane. These panes are explained below.

**Note:** The divider between each of the panes can be moved back and forth as required to resize the panes.

### 3.2.2.1 Object Tree Pane

The left pane in the main work area is the Object Tree pane. This pane is depicted below.

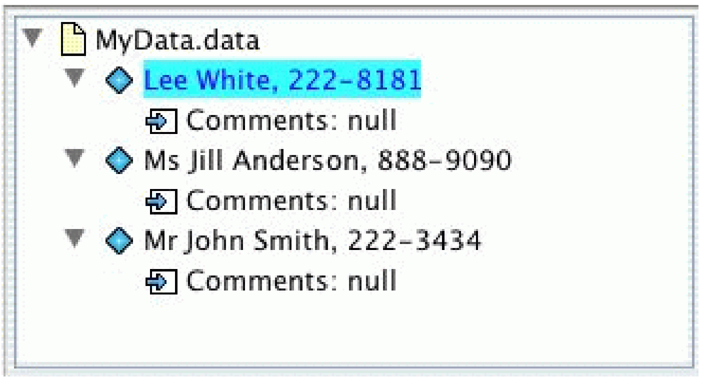

MyData.data
Lee White, 222-8181
Comments: null
Ms Jill Anderson, 888-9090
Comments: null
Mr John Smith, 222-3434
Comments: null

**Figure 3-5:** The Object Tree Pane

This is where new object files appear when loaded or created. Once a dictionary is added to a data file, the browser in the Object Tree pane can be used to navigate the object structure.

### 3.2.2.2 Object Field Pane

The right pane in the main work area is the Object Field pane. This pane is depicted below.

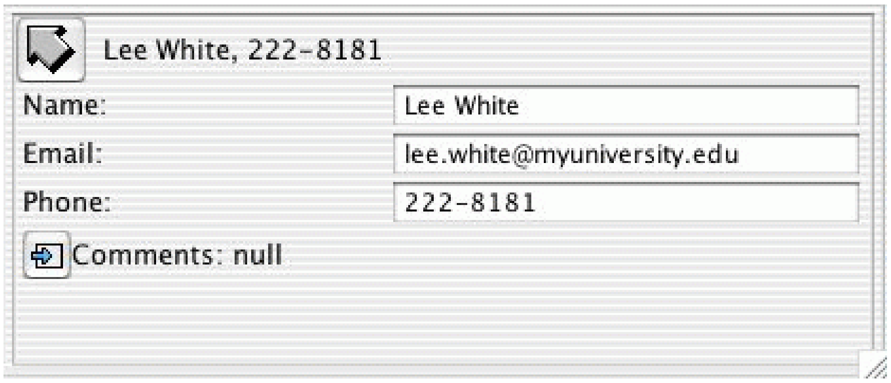

Lee White, 222-8181
Name: Lee White
Email: lee.white@myuniversity.edu
Phone: 222-8181
Comments: null

**Figure 3-6:** The Object Field Pane

This is where the fields that an object contains are displayed and manipulated. These can include text and numeric fields as well as pointers to other objects.

The arrow in the top left-hand corner is used to go up a level in an object structure.

### 3.2.3 Context-sensitive Menus

Context-sensitive menus are available extensively throughout the editor environment. These menus are invoked by moving the pointer over an item and clicking the right mouse button. This presents options that are relevant to the current object or work space in the current context.

**Note:** On some platforms there is no right button on the mouse. On these platforms, context-sensitive menus are invoked by pressing a key and the mouse button in combination.

The context-sensitive menus introduce some new functionality and duplicate some options available in the File menu.

## 3.3 Using the Browser

### 3.3.1 Creating Data Files

Selecting New from the file menu creates a new Data File which appears in the Object Tree pane. Initially this file is unstructured and empty.

Adding a dictionary to an empty file such as this will allow new objects to be declared.

### 3.3.2 Loading Data Files

Data files can be loaded by selecting Open from the file menu. These files appear in the Object Tree pane, but their structure is not known so they cannot be browsed or edited until a dictionary is added to them.

When loading data files, make sure that the file name filter in the dialog is set to All Files so that data files are visible.

### 3.3.3 Adding a Dictionary to a Data File

Invoking the context-sensitive menu for a Data File presents the menu below.

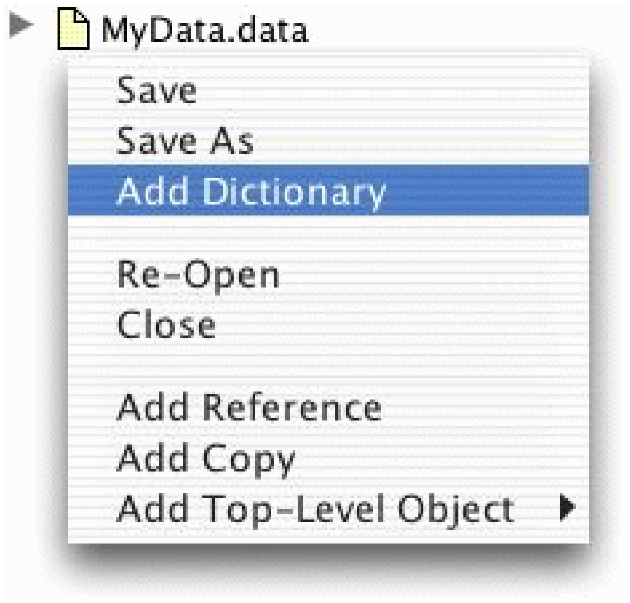


**Figure 3-7:** Add Dictionary in a data file context-sensitive menu

In order to populate a Data File one or more dictionary (`.api`) files must be attached to it. This is achieved by selecting Add Dictionary from the above menu and loading the appropriate file from disk.

This dictionary defines what objects and fields exist in each data file, and describes how they are structured. Once an appropriate dictionary has been added to a data file, its structure and contents can be viewed in the browser.

Once a set of dictionary files has been attached to a Data File, the set cannot be edited. The only way to change a set of associated dictionary files is to close the Data File, open it again and then add dictionary files.

**Note:** The dictionary is applied only for the current session and must be added to the data file each time it is loaded.

## 3.3.4 Adding Objects to Data Files

A sub-menu named Add Top-Level Object is available in the context-sensitive menu for a Data File. An example of this menu is depicted below.

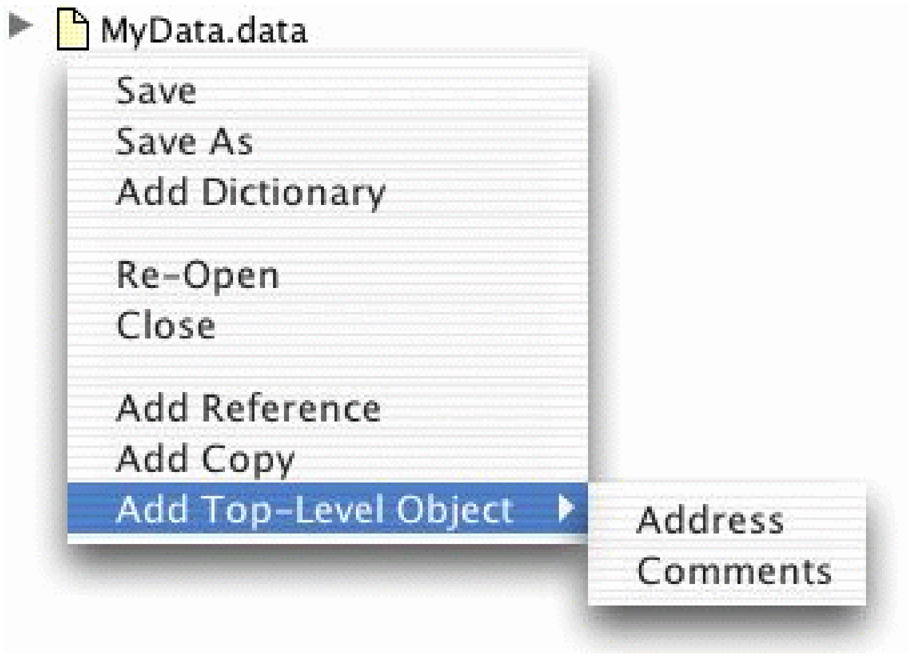


**Figure 3-8:** Add Top-Level Object in a data file context-sensitive menu

**Note:** This option is also available in the context-sensitive menu for top-level objects.

This menu displays all of the top-level objects that are declared in the current dictionary file in alphabetical order. Selecting any of these objects adds a new object of that type to the current data file.

Additionally, either a reference to or a copy of the currently selected object can be added to the data file by selecting Add Reference or Add Copy (respectively).

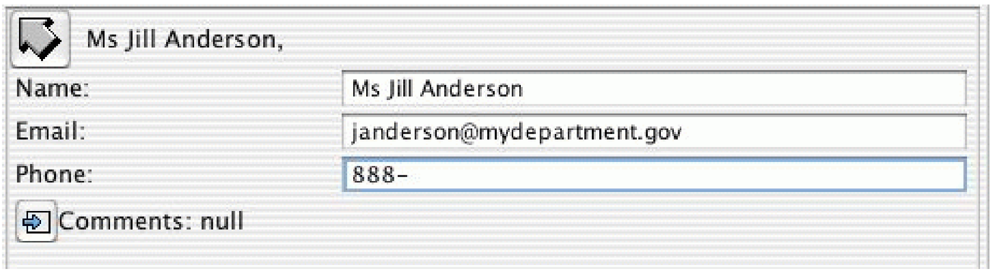
Objects can contain fields and object pointers, which are references to other entities. Object pointers are initially null, but can be made to point to newly created objects.

In the example above, if `Address` is selected, an object of that type would be added to the current data file. The `Address` object has three fields; `Name`, `Email` and `Phone`. It also has a pointer to a `Comments` object which can be added optionally to include extra information about a person.

See the section entitled *Replacing Object Pointers* for details on how to make object pointers non-null.

## 3.3.5 Adding Data to Objects

Once an object has been added to a data file, its fields can be modified. In the example above, an `Address` object is added to the data file, which would result in the following appearing in the Object Field pane:



Ms Jill Anderson,

| | |
|---|---|
| Name: | Ms Jill Anderson |
| Email: | janderson@mydepartment.gov |
| Phone: | 888- |

Comments: null

**Figure 3-9:** An `Address` object in the Object Field pane

Any changes made to the above fields are mirrored in the Object Tree pane if the dictionary file declares these fields as visible there.

Clicking on the `Comments` object pointer will display any comments pertaining to this address book entry. Clicking the arrow in the top left-hand corner of the Object Field pane will return to the previous fields view.

## 3.3.6 Replacing Object Pointers

If an object pointer is null, it must to be replaced with a reference to an object before data can be added to it. This can be achieved by invoking the context-sensitive menu on the object pointer and selecting Refer to New. This menu is depicted below.

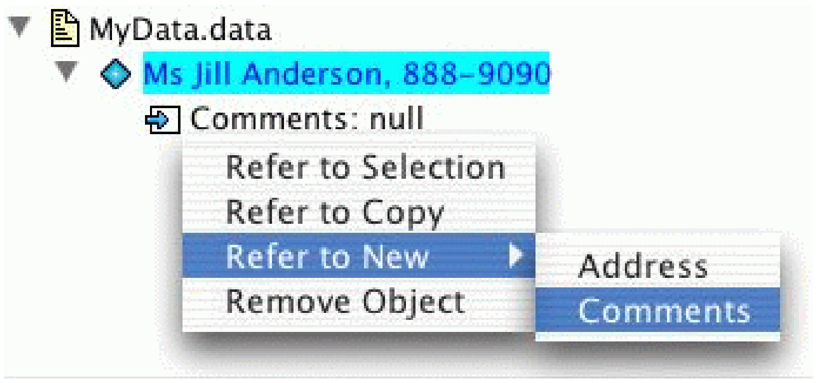


**Figure 3-10:** Refer to New in an object pointer context-sensitive menu

This menu displays objects of the correct type that can be added in place of the current value.

In the example in Figure 2.10, the `Comments` object pointer can be set to point to a newly created `Comments` object. Once created, the contents of this object can be changed and saved to disk like any other field.

In addition to this, an object pointer can be replaced with a pointer to the object currently selected in the Object Tree pane by selecting Refer to Selection. This would allow, for example, the object pointer for two different `Address` entries to point to the same `Comments` object. Further, an object pointer can be replaced with a pointer to a *copy* of the currently selected object by selecting Refer to Copy.

### 3.3.7 Removing Objects

To remove an object pointer, invoke the context-sensitive menu on an object pointer and select Remove Object. This menu is depicted below.

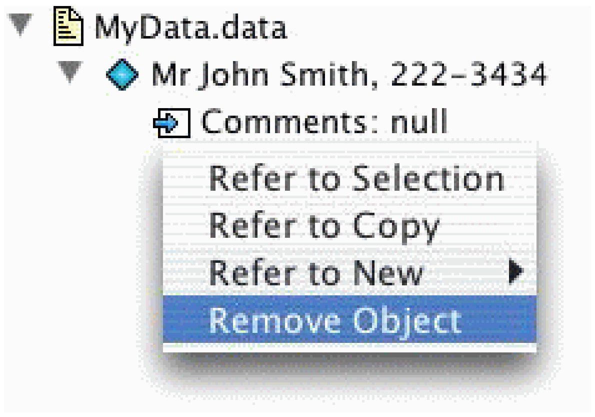


**Figure 3-11:** Remove Object in an object pointer context-sensitive menu

This deletes the reference to that object and makes the object pointer null.

### 3.3.8 Aggregate Objects

An aggregate object is a 'container' into which other objects can be placed. Aggregate objects can be empty, or they can contain one or more objects.

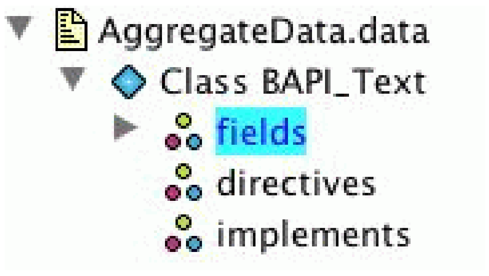


**Figure 3-12:** An unexpanded aggregate object

An 'empty' aggregate object does not have an expansion control next to it; in the image above, 'directives' and 'implements' are empty aggregate objects.

To put something into an aggregate object, open the contextual menu for the aggregate object.

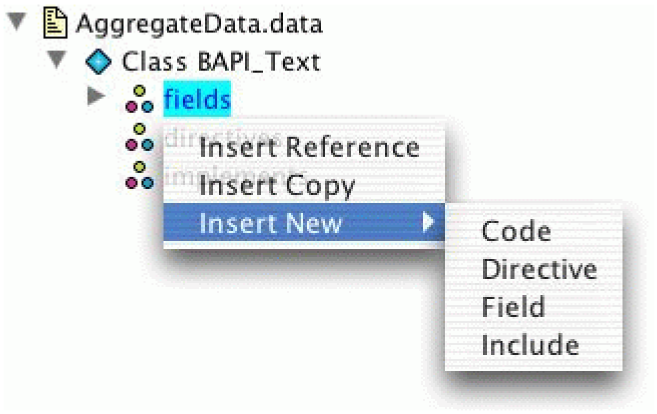


**Figure 3-13:** Insert New in an aggregate context-sensitive menu

With the Insert New submenu, the user can put in a new object of any type the aggregate can hold. Alternatively, you can choose Insert Reference or Insert Copy to put either a reference to or a copy of the currently selected object into the aggregate object.

The user can expand an aggregate object to see its contents. Each object contained in an aggregate object behaves just as if it were a top-level object. The user can also use the contextual menu for an object contained in an aggregate object to add further objects to the aggregate.

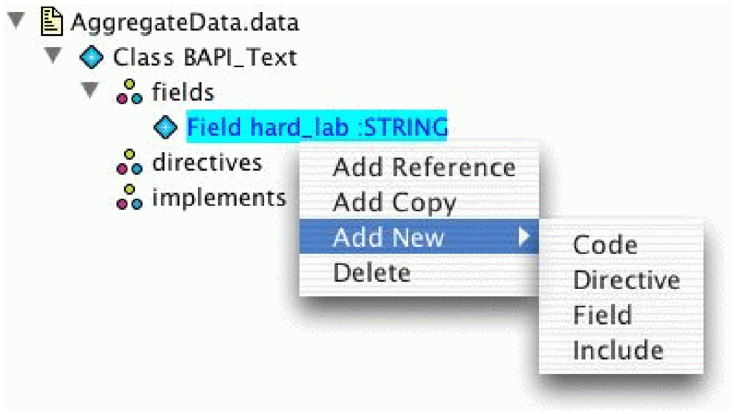


**Figure 3-14:** Add New in an object context-sensitive menu

The Add New, Add Reference and Add Copy menu items behave much the same as the Insert New, Insert Reference and Insert Copy menu items in the contextual menu for the aggregate object itself. However, whereas the Insert items add an object at the top of the aggregate's list, the Add items place the newly added object immediately after the object from whose contextual menu the item was chosen. For example, in the image above, if you choose Add New and Field, the new field will be placed after the `hard_lab` field already in the aggregate.

Items can be deleted from aggregate objects by choosing Delete from the item's contextual menu. You cannot, however, delete the aggregate objects themselves (except by deleting the top-level object of which they are a part).

## 3.3.9 Deleting Top-Level Objects

To delete a top-level object, invoke the context-sensitive menu on an object and select Delete. This menu is depicted below.

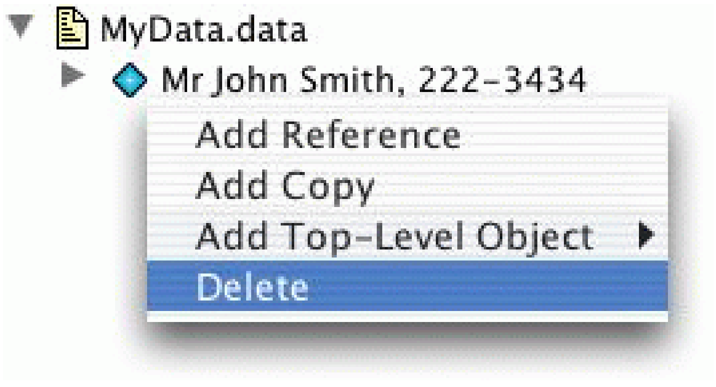


**Figure 3-15:** Delete in an object context-sensitive menu

This causes the selected object to be deleted from the current Data File.

## 3.3.10 Reverting to Saved Data Files

If all changes from an editing session need to be discarded, then invoke the context-sensitive menu on a Data File in the Object Tree pane and select Re-Open. This menu is depicted below.

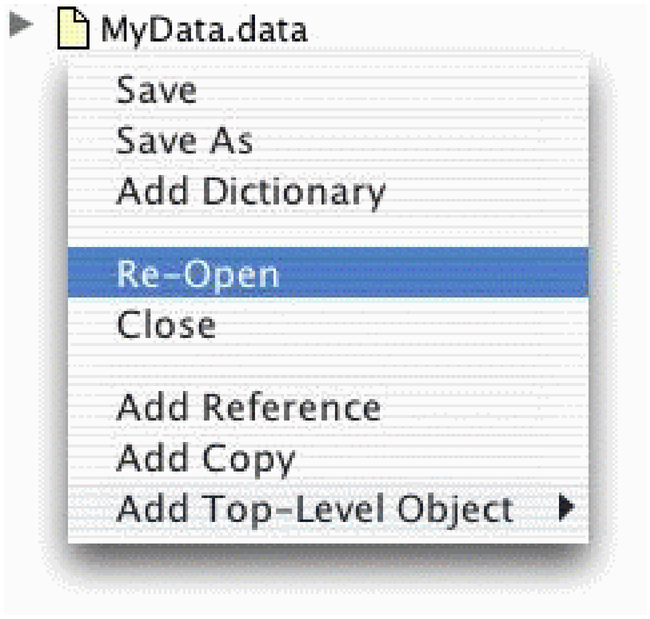


**Figure 3-16:** Re-Open in a data file context-sensitive menu

This option allows the version of the data file most recently saved to disk to be loaded into the editor. A dialog appears to confirm that this action is what was intended.

**Note:** If OK is clicked, all changes are discarded and the original file is reloaded.

## 3.3.11 Saving Data Files

Selecting Save from the file menu or the context-sensitive menu on a Data File, causes the current data file to be saved to disk. If the data file was newly created, the user is given the opportunity to change the default name before saving the file to disk.

The editor automatically creates backup files by copying the original file to a file with a `.bak` extension. The current file is then written to disk.

## 3.3.12 Closing Data Files

Selecting Close from the context-sensitive menu on a Data File closes the current data file. The user is given the option to save any changes before closing the file.

### 3.3.13 Exiting the Editor

Selecting Exit from the File menu causes the JACOB editor to quit. The user is given the option to save any changes before exiting the editor.

# 4 JACOB Object Modelling

## 4.1 JACOB Data Definition Language

This section describes the JACOB Data Definition Language in Extended Backus-Naur Form (BNF) and describes the purpose of each of the fields and keywords.

### 4.1.1 Syntax Definition

In this syntax definition, non-terminal tokens are written as words within angle brackets, like `<Object>`, and square and curly brackets mark repeated (zero or more) and optional (zero or one) parts respectively. Terminal tokens are enclosed by single-quotes, and a vertical bar separates alternatives.

The JACOB Data Definition Language has a syntax as defined below.

```
<Stream> ::= [ <Object> ] <EOF>

<Object> ::= '<' <Ident> { '=' <Number> } { <Object> } [ <Field> ] '>' |
'<' '&' <Number> '>'

<Field> ::= <Fieldname> <Value>

<Value> ::= <Number> | <String> | <Object> | <Aggregation>

<Aggregation> ::= '(' [ <Object> ] ')'
```

The lexical level defines `<Ident>`, `<Fieldname>`, `<Number>`, `<String>` and `<EOF>` as follows:

- `<Ident>` is a sequence of letters, digits and underscores, beginning with a letter or an underscore.
- `<Fieldname>` is an `<Ident>` immediately preceded by a colon character. A `<Fieldname>` is not case-sensitive.
- `<Number>` is an optional minus, followed by either one or more digits and an optional period or a period and a digit, followed by zero or more digits, then optionally an upper or lower case `e`, an optional minus and one or more digits (i.e. standard scientific notation).
- `<String>` is a sequence of characters enclosed by double-quotes, where back-slash ("\") is a meta character, also recognising unicode character encoding.
- `<EOF>` denotes 'end-of-file', which means to say that the input file must be a (possibly empty) sequence of complete `<Object>`, otherwise the compiler will complain.

There may also be Java-style comments interspersed with definitions in the usual way, i.e. each comment is treated as a token separator.

All string values to be used as identifiers in generated code must adhere to the appropriate identifier syntax of the target language, and must not be any of its reserved words.

A JACOB Data Definition Language file is itself a sequence of objects in the language described above. There are six different types of objects to use:

- `Class` for defining a new class.
- `Field` for defining fields within classes.
- `Enum` for defining an enumeration of symbolic values.
- `Member` for defining identifiers of enumerations.
- `Code` for defining in-line code to be transferred to the output source being generated.
- `Include` for declaring code from a file to be transferred to the output source being generated.

#### 4.1.1.1 Class

Objects of type `Class` are definitions of new classes. The fields of such objects are:

- `:name` – a `:string` identifying the transport name of the defined class.
- `:comment` – a `:string` that is treated as a user level (short) description of the defined class. The `:comment` is in particular available when objects of the defined class are written, and may, for example, be written to the output to increase its readability.
- `:classname` – a `:string` identifying the name of the class to define. If omitted, `:name` will be used.
- `:target` – a `:string` identifying the name of the class to construct. If this field is not defined, the constructed class will be named by the `:classname` field.
- `:extends` – a `:string` that identifies the base class of the defined class.
- `:fields` – an `:aggregation` of `Field` objects defining the fields of the defined class.
- `:directives` – an `:aggregation` of `Directive` objects defining compile time directives for JACOB.
- `:tostring` – a format string to implement the Java `toString` method. The string must be in the form of `"%(field-name)"` where `field-name` is the name of any of the fields of a JACOB object.
- `:usedeepequals` – a boolean field used to generate code to do deep equality comparison (Java only). It is `"true"` by default.
- `:javaconstrcode` – Java code to be inserted into the Java constructor.
- `:icon` – the icon to display when showing this class.
- `:implements` – a collection specifying the interface it implements (Java only).
- `:api_extends` – a collection of extra JACOB classes it extends. (C++ only).
- `:other_extends` – a collection of extra non-JACOB classes it extends. (C++ only).

### 4.1.1.2 Field

Objects of type `Field` are used for defining the fields of a class. The fields of a `Field` object are:

- `:name` – a `:string` identifying the transport name of the field being defined.
- `:comment` – a `:string` that is treated as a user level (short) description of the field. The `:comment` is in particular available when objects of the defined class are written and may, for example, be written to the output to increase readability.
- `:fieldname` – a `:string` identifying the actual name of the field defined. If omitted, `:name` will be used.
- `:type` – a token that marks the type of the field being defined. The token must be one of `:char`, `:short`, `:int`, `:long`, `:bool`, `:enum`, `:float`, `:double`, `:string`, `:class` or `:aggregation`. The token determines the data representation associated with the field according to the list below. Certain data representations are only available on certain platforms, and will cause an error on the attempt to transfer to a platform where the data representation is not supported.
  - `:char` field holds a character value; the `char` type in Java is 16-bit unicode, and the `char` value in C++ is 8-bit.
  - `:short` field holds a 16-bit `integer` value in standard 2-complement representation; the `short` type in Java.
  - `:int` field holds a 32-bit `integer` value in standard 2-complement representation; the `int` type in Java.
  - `:long` field holds a 64-bit `integer` value in standard 2-complement representation; the `long` type in Java.
  - `:bool` field holds a 1-bit logical truth value, marked by one of the tokens `:true` or `:false`; the `boolean` type in Java.
  - `:enum` field holds an enumeration member as value.
  - `:float` field holds a 32-bit single precision real number; the `float` type in Java.
  - `:double` field holds a 64-bit single precision real number; the `double` type in Java.
  - `:string` field holds an arbitrary length string of 8-bit characters in the ISO-8859-1 representation.
  - `:class` field holds a reference to another object.
  - `:aggregation` field holds an aggregation of references to other objects. There is a default implementation, but an application program may register its own aggregation.
- `:subtype` – a `string` identifying the enumeration type for an `:enum`, the (base) class of the referred object for a `:class`, and the element class for an `:aggregation`.

- `:value` – a `:string` that is written to the generated output as the means for obtaining the default value for the field.
- `:defaultFlag` – a `:string` that defines a `boolean` member in the generated class to be associated with this field to mark whether the field value was read from the input stream or not. The flag is true by default, and is reset by JACOB upon reading and recognising the field from the input stream.
- `:directives` – a list of `Directive` objects to be processed in conjunction with generating the output.
- `:implemented_by`, which is an attribute for an `:aggregation` field. Its value is a string identifying the actual class by which to implement the aggregation. The named class must then implement the Aggregate interface, and be available as part of the Reader functionality in the application. For C++ code, use `:implemented_in_cxx_by` instead.
- `:implemented_in_java_by`, is the same as `:implemented_by`.
- `:allowed`, which is an attribute for an `:aggregation` field. Its value is a list of `APIString` objects identifying a specific classes (derivation of `:subtype`) allowed as element type. If this attribute is used, then the aggregation is restricted to only derivations of the allowed classes, otherwise any derivation of `:subtype` is allowed.
- `:label` – a `String` which stores the label associated with the field.
- `:isPublic` – a boolean field which denotes whether the field is `public` or not. The default value is `"true"`.
- `:isHidden` – a boolean field which denotes whether the field should be hidden when editing in the JACOB editor. The default value is `"false"`.
- `:isStatic` – a boolean field which denotes whether the field is `static` or not. The default value is `"false"`.
- `:isTransient` – a boolean field which denotes whether the field is `transient` or not. The default value is `"false"`.
- `:javaInit` – a `Text` object that contains the Java initialiser expression for the field.
- `:cxxInit` – a `Text` object that contains the C++ initialiser expression for the field.
- `:inherited` – a boolean field which denotes whether this field is inherited from a superclass or not. The default value is `"false"`.
- `:genReader` – a boolean field which denotes whether the field has a `get` accessor method or not. The default value is `"false"`.
- `:genWriter` – a boolean field which denotes whether the field has a `set` accessor method or not. The default value is `"false"`.
- `:javaReader` – a `String` identifying the `get` accessor method name for the field.
- `:javaWriter` – a `String` identifying the `set` accessor method name for the field.

#### 4.1.1.3 Enum

Objects of type `Enum` are definitions of enumeration types. The fields of such objects are:

- `:name` – a `:string` identifying the name of the enum type as a whole.
- `:comment` – a `:string` that is treated as a user level (short) description of the defined enumerated type. The `:comment` is in particular available when objects of the defined class are written and may, for example, be written to the output to increase its readability.
- `:members` – an `:aggregation` of Member objects defining the enumeration members.

#### 4.1.1.4 Member

Objects of type `Member` are used for defining the members of an enumeration. The fields of a Member object are:

- `:name` – a `:string` stating the member identifier.
- `:comment` – a `:string` that is treated as a user level (short) description of the field. The `:comment` is in particular available when objects of the defined class are written and may, for example, be written to the output to increase readability.
- `:value` – an `:int` designating a value for the identifier. If the value is omitted, the default value is computed as `0` for the first `Member` of an enumeration type, otherwise as `X+1`, where `X` is the value of the preceding `Member` of the enumeration type.
- `:label` – a `String` which stores the label associated with the member.
- `:icon` – the icon to display when showing this member.

#### 4.1.1.5 Code

Objects of type `Code` are portions of target language source code to be copied to the output 'in place'. The fields of a `Code` object are:

- `:lang` – "java" or "cxx".
- `:code` – the code that should appear in the output.

A `Code` object is a `Directive` object.

#### 4.1.1.6 Include

Objects of type `Include` declare files containing portions of target language source code to be copied to the output (usually) 'in place'. The fields of an `Include` object are:

- `:lang` – "java" or "cxx".
- `:filename` – the file that should be copied into the output.

- `:inDeclaration` – a boolean field which denotes whether the file should be included in the C++ declaration section or not. This is only used for C++ code. The default value is `"true"`.

A `Include` object is a `Directive` object.

# 5 Running JACOB

JACOB is used to compile `*.api` files. The compilation procedure results in the generation of C++ or JAVA files which are then compiled using a standard C++ or JAVA compiler (as appropriate). The resulting object code defines the JACOB objects.

## 5.1 JACOB Build

The general form of the command to compile `*.api` files is

```
        java aos.main.JacobBuild <dict-file>
```

where `<dict-file>` is a dictionary file. JACOB Build may also be run with the command line options below.

## 5.2 Command Line Options

### 5.2.1 `-Wlang`

Displays language warnings.

usage : `-Wlang`

arguments : none

default value (option not used) : no language warning errors are reported

### 5.2.2 `-lang`

Sets the default output language of JACOB. The language may either be Java or C++. Another language may be used if a driver for that language is created.

usage : `-lang` *`string`*

default value : java

### 5.2.3 `-pkg`

When Java code is generated, this identifies the location of JACOB classes for the generated code.

usage : `-pkg` *`string`*

default value : `aos.apib`

### 5.2.4 `-syntax`

This defines the file defining the location of the class that initialises JACOB. This effectively sets the syntax of the JACOB language by allowing an alternative definition of the default language file.

Another language, such as XML or SQL may be used if a class that initialises the language exists.

usage : `-syntax` *`string`*

default value : `aos.apib.apic.boot.DefReader`

### 5.2.5 `-dj`

This option identifies the directory where generated files are placed.

usage : `-dj` *`string`*

### 5.2.6 `-dos`

Files generated have the extension `.cpp` instead of `.C`. Note that this option must be used with the `-lang cxx` option.

usage: `-dos`

This option is not used by default.

### 5.2.7 `-v`

Displays the version of the JACOB Compiler.

usage: `-v` or `-version`

### 5.2.8 `-h`

Displays the command line options that may be used with the JACOB Compiler.

usage: `-h` or `-help`

### 5.2.9 `-convert`

Converts data from one format to another format.

usage: `-convert -v` *`mode infile outfile typedict`*

`-v` (optional): verbose mode prints messages about opening files, and reading and writing objects.

*`mode`* is an inmode letter immediately followed by an outmode letter. e.g. `AX` converts ASCII to XML.

`a`,`A` = ASCII; `x`,`X` = XML; `b`,`B` = Binary; `u`,`U` = unspecified (input only)

`baxu` uses the `read/writeMultipleObjects()` interface. `AXBU` uses the `read/writeObject()` interface.

Note that the `readMultipleObjects()` interface is not supported for Binary.

*`typedict`* (optional): a compiled `Init__XXX.class` file or any JACOB definition file. Note that the `.class` extension should not be included.

# 6 Functional Components of JACOB

## 6.1 Compiler

The JACOB Compiler processes one or more definition files, and generates code modules to link in with an application that intends to use, read and write the classes defined through the definition files. The compiler is a Java application that takes one or more definition files as input, and generates one or more Java or C++ files as output.

### 6.1.1 JACOB Data Definition Language

The definition files input to the compiler are defined using the JACOB Data Definition Language. The syntax is defined in Extended Baccus-Naur Form (EBNF) in the section entitled *JACOB Object Modelling*.

### 6.1.2 Mappings in JAVA

For generating Java code, the field type mappings are as described in the section entitled *JACOB Object Modelling*.

All `Enum` definitions result in separate classes; one for each `Enum` class. Within that class, each enumeration member is layed out as a `public static final` data member. Fields of type `:enum` are translated into `int` values.

Each `Class` definition results in its own Java file containing the corresponding Java class definition. In addition, each class definition is associated with a `StreamerSupport` extension class that contains the code for reading and writing objects of the type defined. These `StreamerSupport` extension classes are non-public classes appended to the Java files defining the data classes.

One additional class is also generated for the definition source file. The class is named by adding the prefix `Init__` to the input file name. This class includes all the initialisation needed for using the definition set in an application. Further, when compiling this class with `javac`, all classes concerned are compiled automatically.

Inline source code fragments from Code directives for Java are copied to the output file as they occur among the fields.

The default aggregation is an array. The `Aggregate` interface includes methods for adding elements and constructing an `Enumeration` for accessing elements in order.

### 6.1.3 Mappings in C++

For generating C++ code, the field type mappings are as described in the section entitled *JACOB Object Modelling*, with the exception of the `:bool` type, which is mapped into `unsigned char` such that `0` represents `:false` and `1` represents `:true`.

Each `Enum` definition is mapped into a class definition as in Java, but the member set is also reconstructed as an anonymous `public enum` declaration within the class.

All classes of a definition source file are layed out together in a single C++ source file pair (separating headers from implementation).

The generated code further includes an additional class that includes all the initialisation needed for using the definition set in an application.

Inline source code fragments from `Code` directives for C++ are copied to the output header file essentially as they occur among the fields; in particular, within a `public:` segment.

The default aggregation is a class extending `JACOB_Aggregate`. This includes methods for adding elements and constructing a `JACOB_Enumeration` for accessing the elements in order.

## 6.2 Writer

The Writer is the combined functionality of code libraries and generated code and enables the mapping of an object structure onto a stream representation so that the same object structure is reconstructed when the reader is applied to the stream (see below).

The Writer functionality involves the processing of an in-core data structure for generating an output stream in one of several formats. The algorithm for this is a plain depth-first tree traversal that is implemented partly by pre-compiled library code and partly by the code generated for the class definitions. The latter is contained within the `StreamerSupport` extension classes, and each such class is therefore (made) capable of accessing the transportable fields. (In C++ code the class is called `JACOB_StreamerSupport`.)

### 6.2.1 Java Writer Classes

There is a generic class for output streaming, named `OutStream`, and there are four separate extensions of this:

- `AsciiOutStream` for ASCII file representation.
- `BinaryOutStream` for binary file representation.
- `JDBCOutStream` for JDBC file representation.
- `XMLOutStream` for XML file representation.

### 6.2.2 C++ Writer Classes

The generic C++ class for output streaming is `JACOB_OutStream`. This class currently has the separate extensions `JACOB_AsciiOutStream` for ASCII file representation and `JACOB_BinaryOutStream` for binary file representation.

### 6.2.3 Writing Objects

The writing of objects is generated as part of the `StreamerSupport` or `JACOB_StreamerSupport` derivations as straightforward depth-first tree traversal. The `AsciiOutStream` or `JACOB_AsciiOutStream` identifies and keeps track of objects written out (including the one in progress); upon the second and subsequent occurrence of an object its identification number is written out, preceded by an `&` character instead of traversing the object. The identification number is unique within the object structure.

## 6.3 Reader

The Reader is the combined functionality of code libraries and generated code, and enables the processing of an object stream and the reconstruction of the corresponding runtime object structures it represents. The Reader functionality involves the processing of an input stream in one of several formats into a corresponding in-core data structure, by using a `TypeDict` of definitions regarding which objects the input stream may contain.

### 6.3.1 Java Reader Classes

There is a generic class for input streaming, named `InStream`, and there are separate extensions of this, `AsciiInStream` for the ASCII file representation described earlier, `JDBCInStream` for JDBC file representation, `XMLInStream` for XML file representation and `BinaryInStream` for a compact binary file representation. The `InStream` constructor uses reflection to find the class of an object if it is not known.

To open a JACOB stream regardless of the format, use the static method

```
        InStream.open(InputStream stream, TypeDict dict)
```

which will inspect the start of the stream, and return the correct derivation of InStream after resetting the input stream.

### 6.3.2 C++ Reader Classes

The generic C++ class for input streaming is named `JACOB_InStream`. This class currently has the separate extensions `JACOB_AsciiInStream` for ASCII file representation and `JACOB_BinaryInStream` for binary file representation.

To open a JACOB stream, the format must first be determined. The correct derivation of `JACOB_InStream` can then be opened (e.g. a new `JACOB_AsciiInStream` for ASCII data).

**Note:** When reading from or writing to data streams in Windows, a corresponding JACOB function must be registered. These functions are: `fileRead`, `fileWrite`, `socketRead` and `socketWrite`. An example of registering a read or write function can be seen in the C++ examples in the Sample Applications section.

### 6.3.3 Reading Objects

Reading is invoked by a stream method `readObject`. This reader recognises the `<Object>` introduction phrase (the angle bracket and the class name of the object to come in the input stream). This is sufficient to construct a blank object of the indicated type, and invoke the read method of the associated `StreamerSupport` (or `JACOB_StreamerSupport` for C++).

## 6.4 Initialisation

The Initialisation is the combined functionality of code libraries, generated code, and user-written code by which the reading/writing capability is enabled in an application program.

The JACOB functionality is enabled in an application, apart from any linking of object files, by constructing a `TypeDict` object. The `TypeDict` object defines the mapping between class names occurring in transport streams and actual class definitions generated by the JACOB utility.

When JACOB processes a definition file, it also constructs a definition group class with a static method `addGroup(TypeDict d)`, which at runtime adds all the definition classes of the definition file into the given dictionary (`TypeDict`). The generated definition group class is named after the definition file.

The application program thus constructs and builds the `TypeDict` object or objects as part of its initialisation. The `TypeDict` object is then passed to the reader method of the input stream for reading the input file.

### 6.4.1 Dictionary file

A dictionary file defines the object(s) to be manipulated. When JACOB is run on an `.api` file, say `filename.api`, two files are generated. If Java code is generated, one of the files is called `Init__filename.java`. If C++ code is generated, one of the files is a header file, containing an `Init__filename` class definition.

The `Init_filename` code implements the dictionary specification for the object. Objects are read using the dictionary specified during the input of the stream. The Sample Application section contains examples demonstrating the implementation of dictionaries.

## 6.5 JACOB File Formats

Apart from the simple ASCII language described earlier, JACOB can read and write files in several other formats. These formats encode exactly the same information but may be preferred for other reasons.

There are currently four supported JACOB file formats:

- ASCII
- Binary
- JDBC
- XML.

**Note:** Only the ASCII and Binary formats are currently supported for C++.

### 6.5.1 ASCII

ASCII is a concise and readable format which can be edited by hand quite easily if necessary. The input streamer is called `AsciiInStream` and the output streamer is called `AsciiOutStream`.

### 6.5.2 Binary

The binary format is an extremely compact object representation. It can be used when the speed of object transfer is paramount or if it is desirable to make the stream not human-readable. The input streamer is called `BinaryInStream` and the output streamer is called `BinaryOutStream`.

### 6.5.3 JDBC

The JDBC format is a representation of data in tabular form such as a relational database or spreadsheet. JACOB objects in JDBC format are in the form of multiple tables. These tables are in generic form. If multiple JACOB object structures are stored in a database with other data, they are identified as distinct JACOB object structures by 'stream identifiers'. Stream identifiers are used to uniquely identify JACOB objects transported in a JDBC output streamer.

The input streamer for JACOB objects in JDBC format is called `JDBCInStream` and the output streamer is called `JDBCOutStream`. Both of these classes use methods from the `JDBCSupport` class.

See the JDBC API documentation (`http://java.sun.com/products/jdbc`) for further information.

## 6.5.4 XML

The XML format is very similar to the default ASCII representation.

Where the ASCII representation would write:

```
<Derived
    <InBetween
        <BaseClass :name "ash">
        :ibw 999>
    :x 2
    :base "foobar"
    :y 100
    :doc  <Other :foo 10>
>
```

The XML representation would be:

```
<?xml version="1.0"?>
<jacob version="3.0">

<object type="Derived" id="0">
    <base type="InBetween">
        <base type="BaseClass">
            <field name="doc">
                <object type="Other" id="1">
                    <field name="foo">10</field>
                </object>
            </field>
            <field name="name">ash</field>
            <field name="base">foobar</field>
        </base>

        <field name="ibw">999</field>
    </base>

    <field name="x">2</field>
</object>
```

The XML format is useful if you want to use XML tools to manipulate the JACOB objects or object definitions. The input streamer is called `XMLInStream` and the output streamer is called `XMLOutStream`.

# 7 Sample Applications

JACOB was designed to be used for two primary application types:

1. An object initialisation tool, allowing flexible and consistent initialisation of data objects.
2. An object communication device, allowing populated objects to be sent between different processes.

The first example provides code used to initialise an array of objects using a data file and JACOB.

The second example describes code contained in the JACK distribution defining a client/server system that sends and receives objects in either binary or ASCII. There is code for the client and the server in both C++ and Java.

## 7.1 Object Initialisation example

One of the most common uses of JACOB is as a utility to input data directly into an application. This section lists and describes some simple code that uses JACOB generated code to read data into an application. Java code for this example is described in the Java development steps section. C++ code is included with the JACOB C++ distribution, and is described in the C++ development steps section.

### 7.1.1 Development steps

The procedure used consists of the following steps:

1. The object structure must be defined in an `.api` file using the JACOB Data Definition Language.
2. The `.api` file must be compiled using JACOB.
3. The `.java` or `.pp` files generated by the compilation of each `.api` file must be compiled using a standard Java or C++ compiler.
4. The application input code must be written using the JACOB components (described below).
5. The data file(s) to be read by the application must be written in the appropriate format.
6. The application can be used.

### 7.1.2 Waypoint Initialiser code and description

The example in this section reads a list of waypoints from a data file and stores them in the form of an array of objects of type `Waypoint`. The data is then written to the screen, written to an ASCII file, and written to a binary file. The ASCII and binary files could be read in using the JACOB code by another application, or at a different time by the current application. This code could of course be combined with components of the client/server code and the object read in could be sent to other processes.

### 7.1.2.1 Java development steps

**Step One:** The waypoint object is defined in the file named `waypoint.api` below.

```
<Code :lang "java">

<Class :name "Waypoint"
   :fields (
      <Field :name "x" :type :int>
      <Field :name "y" :type :int>
      <Field :name "altitude" :type :int>
      <Field :name "speed" :type :int>
      <Field :name "id" :type :bool :value "true">
   )
>
```

Note that the embedded 'package' statement should be changed to the actual package being built. Alternatively, this line can be removed entirely if an empty (unnamed) package is being built.

**Step Two:** This must be compiled using the command

```
java aos.main.JacobBuild waypoint.api
```

This will generate two files, `Waypoint.java` and `Init__waypoint.java`.

**Step Three:** These two files must be compiled using the command

```
javac Waypoint.java Init__waypoint.java
```

**Step Four:** The following is a printout of a class called `WaypointInitialiser` that reads a list of waypoints in the form of the `Waypoint` class. The waypoints are stored in an array, printed to the screen, and written to an ASCII file and a binary file.

Lines 1 to 5 import the required classes.

Lines 7 to 10 identify the class.

Line 11 declares a `Base` class type.

Line 13 declares an object of type `TypeDict`. This is the JACOB dictionary declaration.

Lines 16 to 17 declare the variable waypoints as type `Vector`.

Lines 20 to 21 initialise the JACOB dictionary using the JACOB generated class `Init__waypoint`. This defines the waypoint class.

Lines 24 to 70 are the main body of the code. The entire code is contained in the `try` beginning from line 24. While this is not textbook Java, the method was chosen as it simplifies the main body of the code.

Lines 27 to 30 define a JACOB ASCII input stream as the data file `fighter_1.dat`. Note that the stream is associated with the dictionary.

Lines 39 to 42 define the ASCII and binary output streams used for output to the ASCII and binary files.

Lines 46 to 56 read the data file.

Line 46 controls the loop. Objects are read from the data file until there are no more objects to read.

Lines 49 to 52 store each waypoint read in in the `Vector` waypoints.

Lines 54 and 55 write each waypoint object to the binary and ASCII stream as they are read from the input file.

Lines 58 to 62 store the `Vector` waypoints in an array of type `Waypoint`, `w`.

Lines 63 to 70 extract the data from the array `w` and writes it to the screen

Source code for the application Waypoint Initialiser is listed below.

```
 1  import aos.apib.apic.boot.*;
 2  import aos.apib.*;
 3  import java.io.*;
 4  import java.util.Vector;
 5  import java.util.Enumeration;
 6
 7  public class WaypointInitialiser {
 8    public static void
 9    main(String[] args)
10    {
11      Base input_base;
12
13      TypeDict dict;
14
15      // declare the vector waypoints
16      Vector waypoints;
17      waypoints = new Vector();
18
19      //initialise dictionary
20      dict = new TypeDict();
21      dict.initialize("Init__waypoint");
22
23      // Read in waypoint objects from data file
24      try {
25        // Set up input and output streams
26        // Data file
27        InputStream data_file =
          new FileInputStream("fighter_1.dat");
      InStream in = new AsciiInStream(new BufferedReader(
          new InputStreamReader(data_file)),dict);

      // Binary output
      OutputStream binary_output_file = new BufferedOutputStream(
          new FileOutputStream("waypoint_out_file_Binary.txt"));
      BinaryOutStream binary_out =
         new BinaryOutStream(binary_output_file);

      // ASCII output
       OutputStream ASCII_output_file = new BufferedOutputStream(
           new FileOutputStream("waypoint_out_file_Ascii.txt"));
       AsciiOutStream ASCII_out =
           new AsciiOutStream(ASCII_output_file);

       // read objects from data file and store them, and write
       // them to an ASCII and a binary file
       while((input_base  = in.readObject()) != null)
       {
         // store waypoints in the vector waypoints
         if (input_base instanceof Waypoint)
         {
           waypoints.addElement(input_base);
         }
         // write objects to ASCII and binary file
         binary_out.writeObject(input_base);
         ASCII_out.writeObject(input_base);
       }
       // write data stored in waypoints vector to the screen
       Waypoint [] w = new Waypoint[waypoints.size()];
       int i = 0;
       for(Enumeration e = waypoints.elements();
           e.hasMoreElements();)
         w[i++]= (Waypoint)e.nextElement();
       for (int ii=0; ii<w.length; ii++) {
         int waypoint_number = ii+1;
         System.out.println("Waypoint " + waypoint_number + ":"
             + "   x = " + w[ii].x
             + "   y = " + w[ii].y
             + "   altitude = " + w[ii].altitude
             + "   speed = " + w[ii].speed);
       }
     }
     catch (Throwable e) {
       System.err.println("Caught "+e);
       e.printStackTrace();
       // close binary_out and ASCII_out if you want to see the
       // partially generated output files for debugging purposes
       System.exit(1);
     }
   }
}
```

**Step Five:** A sample data file containing a list of six waypoints, `fighter_1.dat`, is listed below.

```
<Waypoint :x 10 :y 20 :altitude 1000 :speed 100 :id :false>
<Waypoint :x 40 :y 20 :altitude 1000 :speed 100 :id :false>
<Waypoint :x 50 :y 60 :altitude 1000 :speed 100 :id :false>
<Waypoint :x 100 :y 200 :altitude 10000 :speed 100 :id :true>
<Waypoint :x 150 :y 300 :altitude 10000 :speed 100 :id :true>
<Waypoint :x 200 :y 200 :altitude 10000 :speed 100 :id :true>
```

When the above code is executed:

- The data file `fighter_1.dat` is read.
- The waypoint data is written to the screen.
- A file, `waypoint_out_file_Binary.txt`, is created containing the waypoint data as generated by JACOB in binary form.
- A file, `waypoint_out_file_Ascii.txt`, is created containing the waypoint data as generated by JACOB in ASCII form.

The code has demonstrated the reading of an ASCII file, and the writing of ASCII and binary files. The reading of binary files is also possible using a similar method to the ASCII read.

### 7.1.2.2 C++ development steps

The code in this example has been tested under Linux and VC++ but it may still require some changes, depending on the C++ environment and where the JACOB libraries are installed.

The C++ object initialisation example may be compiled with the provided Makefile by typing 'make', or by following the steps below.

**Step One:** The waypoint object is defined in the file named `waypoint.api` below.

```
<Class :name "Waypoint"
    :fields (
        <Field :name "x" :type :int>
        <Field :name "y" :type :int>
        <Field :name "altitude" :type :int>
        <Field :name "speed" :type :int>
        <Field :name "id" :type :bool :value "true">
    )
>
```

Note that this file is the same as the `waypoint.api` file in Step One of the Java development steps section apart from the `Code` object. JacobBuild will ignore a `Code` object if the `:lang` field does not match the language of the source code being generated.

**Step Two:** The `waypoint.api` file must be compiled using the command

```
java aos.main.JacobBuild -dos -lang cxx waypoint.api
```

which generates two files, `waypoint.h` and `waypoint.cpp`.

Note that using the `-dos` command line option creates `.cpp` files instead of `.C` files.

**Step Three:** The generated C++ file must be compiled, either with the provided Makefile, using an IDE such as Visual Studio or using the command

```
g++ -c -Wall -I../.. waypoint.cpp
```

**Step Four:** The following is a printout of a file called `WaypointInitialiser.cpp` that reads a list of waypoints in the form of the `waypoint` class. The waypoints are stored in an array that is printed to the screen and written to an ASCII file and a binary file.

Lines 1 to 15 include the required header files.

Lines 17 to 19 are the application ID.

Lines 20 to 24 declare the `Streams` and other parameters used in the application.

Lines 25 to 27 declare the files used in the application.

Line 23 declares a `JACOB_Base` class type.

Line 24 declares a an object of type `JACOB_TypeDict`. This is the JACOB dictionary declaration.

Lines 39 to 40 construct and initialise the JACOB dictionary using the JACOB generated class `Init__waypoint`. This defines the waypoint class.

Line 62 defines a JACOB ASCII input stream as the data file `fighter_1.dat`. Note that the stream is associated with the dictionary.

Lines 108 to 111 define the ASCII and binary output streams used for output to the ASCII and binary files.

Lines 115 to 129 read the data file.

Line 124 stores each waypoint read in the `vector` waypoints.

Lines 127 to 128 write each waypoint object to the binary and ASCII stream as they are read from the input file.

Lines 140 to 152 extract the data from vector and write it to the screen.

Source code for the application WaypointInitialiser is provided below.

```
#include <jacob/AsciiOutStream.h>
#include <jacob/AsciiInStream.h>
#include <jacob/BinaryOutStream.h>
#include <jacob/BinaryInStream.h>
#include "jacob/JACOB.h"

#ifndef _WIN32
#include <fcntl.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <vector.h>

#include "waypoint.h"

int
main(int ac)
{
    JACOB_BinaryOutStream    *binary_out = 0;
    JACOB_AsciiOutStream     *ascii_out = 0;
    JACOB_InStream           *data_file = 0;
    JACOB_Base               *input_base = 0;
    JACOB_TypeDict           *dict = 0;
    char   *filename = "fighter_1.dat";
    char   *binaryfilename = "waypoint_out_file_Binary.txt";
    char   *asciifilename = "waypoint_out_file_Ascii.txt";

#ifdef _WIN32
        WSADATA foo;
        WSAStartup(1, &foo);
#endif

    int infd = 0;

    vector<Waypoint*> waypoints;

    //initialise dictionary
    dict = new JACOB_TypeDict();
    Init__waypoint::init(dict);

#ifdef _WIN32
    infd = (int)CreateFile(filename, GENERIC_READ,
                FILE_SHARE_READ, NULL, OPEN_ALWAYS,
                FILE_ATTRIBUTE_NORMAL, NULL);
    if ((RW_HANDLE)infd == INVALID_HANDLE_VALUE)
    {
        fprintf(stderr,
                "OpenFile: GetLastError() => %d\n",
                GetLastError());
        exit(1);
    }
#else
    infd = open(filename, O_RDONLY);
    if(infd < 0)
    {
        fprintf(stderr, "failed to open %s\n", filename);
        exit(1);
    }
```

```
#endif

    data_file = new JACOB_AsciiInStream(infd, dict);

    data_file->registerReadFunction(JACOB_InStream::fileRead);

#ifdef _WIN32
    int binary_output_file =  (int)CreateFile(binaryfilename,
        GENERIC_WRITE, FILE_SHARE_READ, NULL, CREATE_ALWAYS,
        FILE_ATTRIBUTE_NORMAL, NULL);
    if ((RW_HANDLE)infd == INVALID_HANDLE_VALUE)
    {
        fprintf(stderr,
                "OpenFile: GetLastError() => %d\n", i
                GetLastError());
        exit(1);
    }

    int ascii_output_file =  (int)CreateFile(asciifilename,
        GENERIC_WRITE, FILE_SHARE_READ, NULL, CREATE_ALWAYS,
        FILE_ATTRIBUTE_NORMAL, NULL);
    if ((RW_HANDLE)infd == INVALID_HANDLE_VALUE)
    {
        fprintf(stderr,
                "OpenFile: GetLastError() => %d\n",
                GetLastError());
        exit(1);
    }
#else
    // open file for Binary output, create file if not present
    int binary_output_file = open(binaryfilename,
            O_WRONLY | O_CREAT, 0644);

    if(binary_output_file < 0)
    {
        fprintf(stderr, "failed to open %s\n", binaryfilename);
        exit(1);
    }

    // open file for ASCII output, create file if not present
    int ascii_output_file = open(asciifilename,
            O_WRONLY | O_CREAT, 0644);

    if(ascii_output_file < 0)
    {
        fprintf(stderr, "failed to open %s\n", asciifilename);
    }
#endif
    binary_out =
        new JACOB_BinaryOutStream(binary_output_file);
    ascii_out =
        new JACOB_AsciiOutStream(ascii_output_file);

    // read objects from data file, store them, and write
    // them to an ASCII and a binary file
    for(;;)
    {
        input_base = data_file->readObject();
        if(input_base == 0) break;

        Waypoint *waypt =
            reinterpret_cast<Waypoint*>(input_base);

        // store waypoints in the vector waypoints
        waypoints.push_back(waypt);

        // write objects to ASCII and binary file
        binary_out->writeObject(input_base);
        ascii_out->writeObject(input_base);
    }
    close(infd);
    close(binary_output_file);
    close(ascii_output_file);

    delete binary_out;
    delete ascii_out;

    delete input_base;

    // write data stored in waypoints vector to the screen
    for(int i=0; i < (int)waypoints.size(); i++)
    {
        int waypoint_number = i+1;
        fprintf(stderr,
                "Waypoint %i:  x= %d  y = %d
                 altitude = %d  speed = %d \n",
                waypoint_number, waypoints[i]->x,
                waypoints[i]->y,
                waypoints[i]->altitude,
                waypoints[i]->speed);

        delete waypoints[i];
    }

    return 0;
}
```

**Step Five:** The application can be compiled using

```
g++ -c -Wall -I../.. WaypointInitialiser.cpp
g++ -o WaypointInitialiser WaypointInitialiser.o -L../../lib/linux/g++-2.95
-ljacob
```

A sample data file containing a list of six waypoints, `fighter_1.dat`, is listed in Step Five of the Java development steps section.

When the above code is executed:

- The data file `fighter_1.dat` is read.
- The waypoint data is written to the screen.
- A file, `waypoint_out_file_Binary.txt`, is created containing the waypoint data as generated by JACOB in binary form.
- A file, `waypoint_out_file_Ascii.txt`, is created containing the waypoint data as generated by JACOB in ASCII form.

The code has demonstrated the reading of an ASCII file, and the writing of ASCII and binary files. The reading and writing of binary files is also possible using similar methods to the ASCII reading and writing methods.

## 7.2 Object Communication example

The object communication capability of JACOB allows the sending and receiving of data objects between JAVA and C++ classes. One of the major potential uses of this code is to implement rapid data transport between C++ and JAVA code. This section lists and describes some simple code that demonstrates this capability.

Both Java and C++ code for a simple client and server system is included with the JACOB C++ distribution. Java code for this example is described in the Java development steps section. C++ code is described in the C++ development steps section.

### 7.2.1 Development steps

1. The data object structure(s) must be defined in an `.api` file (or multiple `.api` files if desired) using the JACOB Data Definition Language.
2. The `.api` file(s) must be compiled using JACOB.
3. The `.java` or `.cpp` files generated by the compilation of each `.api` file must be compiled using a standard Java or C++ compiler.
4. The communication code can be written to use the compiled dictionaries to send and receive the defined objects.
5. The application can be used.

### 7.2.2 Client/Server code and description

The example described in this section consists of two applications, a server and a client. The server receives data objects from a client and sends the data back to the client. The client sends a specified data object to the server (and receives the object back again) a specified number of times. This code can be used as test code to obtain timing information for data object sending. The data object can be sent or received in either ASCII or binary form, or in XML form (Java only).

#### 7.2.2.1 Java development steps

**Step One:** Some test data objects are defined in the file `msg.api`. This file is partially reproduced below.

**Note:** The excluded code (not printed below) contains more data objects exhibiting some of the more complex features of the JACOB Data Definition Language.

## msg.api

```
<Class
    :name "RealSimple"
    :tostring "%T@%I %(val)"
    :fields     (
        <Field
            :name "val"
            :type :int
        >
    )
>

<Class
    :name       "Test"
    :fields     (
        <Field
            :name "val"
            :type :double
        >
    )
>

<Class
    :name       "Test10"
    :fields     (
        <Field
            :name "val0"
            :type :int
        >
        <Field
            :name "val1"
            :type :int
        >
        <Field
            :name "val2"
            :type :int
        >
        <Field
            :name "val3"
            :type :int
        >
        <Field
            :name "val4"
            :type :int
        >
        <Field
            :name "val5"
            :type :int
        >
        <Field
            :name "val6"
            :type :int
        >
        <Field
            :name "val7"
            :type :int
        >
        <Field
            :name "val8"
            :type :int
        >
        <Field
            :name "val9"
            :type :int
        >
    )
>
    :
    :
   (The remainder of the file has not been printed)
    :
```

The three data objects defined above are very simple but can be used to illustrate this application, as will be shown later in this section in Step Five.

**Step Two:** This must be compiled using the command

```
java aos.main.JacobBuild msg.api
```

This will generate several files. For each of the objects (*ObjectName*) defined in the `msg.api` file it will generate an `ObjectName.java`. It will also generate an `Init__msg.java` file.

**Step Three:** These generated java files must be compiled, for example using the command

```
javac *.java
```

**Step Four**: The server and client code are printed and explained below.

**Server**

The server receives data objects sent to it by a client, and sends the objects back to the client.

Lines 1 to 3 consist of the required imports.

Lines 5 to 9 are the class and application ID.

Line 10 declares an object of type `TypeDict` (the dictionary).

Lines 11 to 14 declare the `Streams` and other parameters used in the class.

Lines 16 to 22 check the number of command line arguments.

Line 24 constructs the new dictionary (`TypeDict`).

Line 25 initialises the dictionary.

Lines 27 to 34 initialise the socket with the user specified port number.

Lines 35 to 73 contain the code that reads the objects and writes them in the appropriate type (binary, ASCII or XML). This code loops forever, accepting a connection and reading all objects present on the stream.

Line 67 declares an object of type `Base`.

Lines 68 and 69 read the data object and then write the data object back to the client.

### Server code

```
  1  import aos.apib.*;
  2  import java.io.*;
  3  import java.net.*;
  4
  5  public class Server
  6  {
  7    public static void
  8    main(String[] args)
  9    {
 10      TypeDict          dict;
 11      Socket            sock;
 12      OutStream         out = null;
 13      InStream          in = null;
 14      ServerSocket      serv = null;
 15
 16      if ( args.length != 4 ) {
 17        System.err.println( "usage:  java Server "+
 18            "<Init__*> <port> <send-format> <recv-format>");
 19        System.err.println("       <send-format> = A, B or X");
 20        System.err.println("       <recv-format> = A, B or X");
 21        System.exit(1);
 22      }
 23
 24      dict = new TypeDict();
 25      dict.initialize(args[0]);
 26
 27      try {
 28        serv = new ServerSocket(Integer.parseInt(args[1]));
 29      }
 30      catch (Exception e) {
 31       System.err.println("Can't accept the connection");
 32        e.printStackTrace();
 33        System.exit(1);
 34      }
 35      for (;;) {
 36        // Loop forever, accepting a connection and reading
 37        // all objects present on the stream.
 38        try {
 39          System.err.println("waiting for a connection...");
 40          sock = serv.accept();
 41          if ( args[2].equalsIgnoreCase("A") )
 42            out = new AsciiOutStream(sock.getOutputStream());
 43          else if ( args[2].equalsIgnoreCase("B") )
 44            out = new BinaryOutStream(sock.getOutputStream());
 45          else if ( args[2].equalsIgnoreCase("X") )
 46            out = new XMLOutStream(sock.getOutputStream());
 47          else {
 48            System.err.println("Bad send format");
 49            System.exit(10);
 50          }
 51          if ( args[3].equalsIgnoreCase("A") )
 52            in = new AsciiInStream(sock.getInputStream(),dict);
 53          else if ( args[3].equalsIgnoreCase("B") )
 54            in = new BinaryInStream(sock.getInputStream(),dict);
 56          else if ( args[3].equalsIgnoreCase("X") )
 57            in = new XMLInStream(sock.getInputStream(),dict);
 58          else {
 59            System.err.println("Bad recv format");
 60            System.exit(10);
 61          }
 62        }
 63        catch (Exception e) {
 64          e.printStackTrace();
 65          System.exit(1);
 66        }
 67        Base   obj;
 68        while ( (obj = in.readObject()) != null ) {
 69          out.writeObject(obj);
 70        }
 71        System.err.println("no more objects to read");
 72      }
 73    }
 74  }
```

### Client

The client sends a specified data object to the server (and receives the object back again) a specified number of times.

Lines 1 to 3 consist of the required imports.

Lines 5 to 8 are the class and application ID.

Line 10 declares an object of type `TypeDict` (the dictionary).

Lines 11 to 14 declare the `Streams` and other parameters used in the class.

Lines 16 to 29 check the number of command line arguments.

Line 31 constructs the dictionary.

Line 32 initialises the dictionary.

Lines 34 to 46 use the command line arguments to set the host, input file, port number and the number of times a data object is to be sent by the client.

Lines 50 to 61 set up the input stream to read in the data object.

Line 62 reads the data object into `obj`.

Lines 67 to 93 set up the socket (using the user-specified port number and host), and the input and output streams (ASCII, binary or XML streams as defined by the command line arguments).

Line 95 records the time at the start.

Lines 97 to 106 write the data object to the output stream and read the data object when it is returned. This is repeated the number of times specified by the command line argument.

Lines 107 to 110 print the time taken to send and receive the data object the specified number of times.

## Client code

```
 1  import aos.apib.*;
 2  import java.io.*;
 3  import java.net.*;
 4
 5  public class Client
 6  {
 7    public static void
 8    main(String[] args)
 9    {
10      TypeDict        dict;
11      Socket          sock;
12      OutStream       out = null;
13      InStream        in = null;
14      ServerSocket    serv = null;
15
16      if ( args.length != 7 ) {
17        System.err.println(
18            "usage: java Client <Init__*> "+
19            "<host> <port> <file> <count> <send> <recv>");
20        System.err.println(
21            "   <send> = A, B or X (Ascii, Binary, XML)");
22        System.err.println(
23            "   <recv> = A, B or X");
24        System.err.println(
25            "   <file> contains a JACOB object");
26        System.err.println(
27            "      if <file> = \"-\", reads from stdin");
28        System.exit(1);
29      }
30
31      dict = new TypeDict();
32      dict.initialize(args[0]);
33
34      int     count = 0;
35      int     port = 0;
36      String  host = args[1];
37      String  file = args[3];
38
39      try {
40        port  = Integer.parseInt(args[2]);
41        count = Integer.parseInt(args[4]);
42      }
43      catch (Exception e) {
44        System.err.println("Bad port or count given");
45        System.exit(10);
46      }
47
48      int tot = count;
49
50      try {
51        if ( file.equals("-") )
```

```
    in = new AsciiInStream(new BufferedReader(
        new InputStreamReader(System.in)), dict);
   else
    in = new AsciiInStream(
        new BufferedReader(new FileReader(file)),dict);
  }
  catch (Exception e) {
   e.printStackTrace();
   System.exit(1);
  }
  Base obj = in.readObject();
  if (obj == null) {
   System.err.println("Failed to read an object to send");
   System.exit(1);
  }
  try {
   sock = new Socket(host, port);
   if ( args[5].equalsIgnoreCase("A") )
    out = new AsciiOutStream(sock.getOutputStream());
   else if ( args[5].equalsIgnoreCase("B") )
    out = new BinaryOutStream(sock.getOutputStream());
   else if ( args[5].equalsIgnoreCase("X") )
    out = new XMLOutStream(sock.getOutputStream());
   else {
    System.err.println("Bad send format");
    System.exit(10);
   }
   if ( args[6].equalsIgnoreCase("A") )
    in = new AsciiInStream(sock.getInputStream(),dict);
   else if ( args[6].equalsIgnoreCase("B") )
    in = new BinaryInStream(sock.getInputStream(),dict);
   else if ( args[6].equalsIgnoreCase("X") )
    in = new XMLInStream(sock.getInputStream(),dict);
   else {
    System.err.println("Bad recv format");
    System.exit(10);
   }
  }
  catch (Exception e) {
   e.printStackTrace();
   System.exit(1);
  }

  long start_t = System.currentTimeMillis();

  while (count > 0) {
   count--;
   out.writeObject(obj);
   obj = in.readObject();
   if (obj == null) {
    System.err.println(
        "Error with "+count+" messages to be read");
     break;
    }
  }
  System.err.println(
    (System.currentTimeMillis()-start_t)+
    " ms to send and recv "+(tot-count)+" messages");
 }

}
```

**Step Five**: The application can be compiled using

```
 javac *.java
```

If the data is being sent to the server in ASCII form and being sent back to the client in ASCII form then the server may be started using the following command

```
 java Server Init__msg 8888 A A
```

The last two command line arguments specify the form in which the data object is sent and received. This can be either A (ASCII), B (binary) or X (XML). Note that the server can be started in its own window or it can be executed in the background.

The client may be started using the following command

```
 java Client Init__msg localhost 8888 data.RealSimple 10 A A
```

This also specifies that the data is being sent and received to/from the server in ASCII form. `data.RealSimple` is the file that contains the data. For example it could contain

```
<RealSimple :val 10>
```

### 7.2.2.2 C++ development steps

The code in this example has been tested under Linux and VC++ but it may still require some changes depending on the C++ environment and where the JACOB libraries are installed.

The C++ object communication example may be compiled with the provided Makefile by typing 'make', or by following the steps below. Further information about running this example is provided in the `readme` file in the example directory.

**Step One:** Some test data objects are defined in the file `msg.api`. Part of this file is reproduced in Step One of the Java development steps section.

**Step Two:** The `msg.api` file must be compiled using the command

```
   java aos.main.JacobBuild -dos -lang cxx msg.api
```

This will generate two files, `msg.h` and `msg.cpp`.

**Step Three:** The generated C++ file must be compiled, either with the provided Makefile or using the command

```
   g++ -c -Wall -I.. msg.cpp
```

**Step Four**: The server and client code are printed and explained below.

**Server**

The server receives data objects sent to it by a client, and sends the objects back to the client.

Lines 1 to 9 include the required header files.

Lines 14 to 20 declare the `Streams` and other parameters used in the application.

Line 16 declares an object of type `JACOB_TypeDict` (the dictionary).

Line 17 declares an object of type `JACOB_Base`.

Lines 27 to 30 check the number of command line arguments.

Lines 31 constructs the new dictionary (`JACOB_TypeDict`).

Line 32 initialises the dictionary.

Line 34 initialises the socket with the user specified port number.

Lines 35 to 82 contain the code that reads the objects and writes them in the appropriate type (binary or ASCII). This code loops while a connection is accepted.

Lines 71 and 77 read the data object and write the data object back to the client. This code loops forever, reading all objects present on the stream.

**Server code**

```
  1  #include <jacob/AsciiOutStream.h>
  2  #include <jacob/AsciiInStream.h>
  3  #include <jacob/BinaryOutStream.h>
  4  #include <jacob/BinaryInStream.h>
  5  #include <jacob/socket.h>
  6
  7  #include <stdlib.h>
  8
  9  #include "msg.h"
 10
 11  int
 12  main(int ac, char **av)
 13  {
 14      JACOB_OutStream     *x;
 15      JACOB_InStream      *y;
 16      JACOB_TypeDict      *d;
 17      JACOB_Base          *b;
 18      int                 fd;
 19      JACOB_ServerSocket  *c;
 20      int                 port;
 21
 22  #ifdef _WIN32
 23      WSADATA foo;
 24      WSAStartup(1, &foo);
 25  #endif
 26
 27      if (ac != 3) {
 28          fprintf(stderr, "usage: server <port> [AB][AB]\n");
 29          exit(1);
 30      }
```

```
    d = new JACOB_TypeDict();
    Init__msg::init(d);
    sscanf(av[1], "%d", &port);
    c = new JACOB_ServerSocket( port);
    while ((fd = c->Accept()) >= 0) {
        if (av[2][0] == 'A') {
            y = new JACOB_AsciiInStream(fd, d);
            fprintf(stderr,
                "server: opened input stream in ascii mode\n");
        } else if (av[2][0] == 'B') {
            y = new JACOB_BinaryInStream(fd, d);
            fprintf(stderr,
                "server: opened input stream in binary mode\n");
        } else {
            fprintf(stderr,
                "%c not a valid input stream type\n", av[2][0]);
            exit(1);
        }

        if (av[2][1] == 'A') {
            x = new JACOB_AsciiOutStream(fd);
            fprintf(stderr,
                "server: opened output stream in ascii mode\n");
        } else if (av[2][1] == 'B') {
            x = new JACOB_BinaryOutStream(fd);
            fprintf(stderr,
               "server: opened output stream in binary mode\n");
        } else {
            fprintf(stderr,
               "%c not a valid output stream type\n", av[2][1]);
            exit(1);
        }

        // Needed under Windows
        // (doesn't hurt under UNIX but not necessary).
        y->registerReadFunction(JACOB_InStream::socketRead);
        x->registerWriteFunction(JACOB_OutStream::socketWrite);

        int msgs = 0;

        for (;;) {
            b = y->readObject();
            if (b == 0) break;
            msgs++;
            x->writeObject(b);
            delete b;
        }
        close(fd);
        fprintf(stderr, "received %d messages\n", msgs);
        delete x;
        delete y;
    }
    return 0;
}
```

## Client

The client sends a specified data object to the server (and receives the object back again) a specified number of times.

Lines 1 to 15 include the required header files.

Lines 20 to 32 declare the `Streams` and other parameters used in the application.

Line 22 declares an object of type `JACOB_TypeDict` (the dictionary).

Lines 41 to 65 check the number of command line arguments.

Lines 42 to 58 use the sixth command line argument to set the input file.

Lines 66 to 68 use the command line arguments to set the port number and the number of times a data object is to be sent by the client.

Line 69 initialises the socket with the user specified port number and host.

Line 72 constructs the dictionary (`JACOB_TypeDict`).

Line 73 initialises the dictionary.

Lines 75 to 80 set up the input stream to read in the data object.

Line 82 reads the data object into `b`.

Lines 91 to 116 set up the input and output streams (ASCII or Binary streams as defined by the command line arguments).

Line 122 records the time at the start.

Lines 125 to 131 write the data object to the output stream and read the data object when it is returned.

Line 140 prints the time taken to send and receive the data object the specified number of times.

### Client code

```
  1  #include <jacob/AsciiOutStream.h>
  2  #include <jacob/AsciiInStream.h>
  3  #include <jacob/BinaryOutStream.h>
  4  #include <jacob/BinaryInStream.h>
  5  #include "jacob/JACOB.h"
  6  #include "jacob/socket.h"
  7
  8  #ifndef _WIN32
  9  #include <sys/time.h>
 10  #include <fcntl.h>
 11  #endif
 12
 13  #include <stdlib.h>
 14
```

```
#include "msg.h"

int
main(int ac, char **av)
{
    JACOB_OutStream     *x = 0;
    JACOB_InStream      *y = 0;
    JACOB_TypeDict      *d = 0;
    JACOB_Base          *b = 0;
    int                 i;
    int                 fd;
    JACOB_ClientSocket  *c = 0;
    int                 port;
    int                 n;

#ifndef _WIN32
    struct timeval      t1, t2;
#endif

#ifdef _WIN32
    WSADATA foo;
    WSAStartup(1, &foo);
#endif

    int infd = 0;

    if (ac == 6) {
#ifdef _WIN32
        infd = (int)CreateFile(av[5], GENERIC_READ,
            FILE_SHARE_READ, NULL, OPEN_ALWAYS,
            FILE_ATTRIBUTE_NORMAL, NULL);
        if ((RW_HANDLE)infd == INVALID_HANDLE_VALUE) {
            fprintf(stderr, "OpenFile: GetLastError() => %d\n",
                GetLastError());
            exit(1);
        }
#else
        infd = open(av[5], 0);
        if ( infd < 0 ) {
            fprintf(stderr, "failed to open %s\n", av[5]);
            perror(av[5]);
            exit(1);
        }
#endif
        ac = 5;
    }
    if (ac != 5) {
        fprintf(stderr,
            "usage: client host port count [AB][AB] [file]\n");
        exit(1);
    }
    sscanf(av[2], "%d", &port);
    sscanf(av[3], "%d", &i);
    n = i;
    c = new JACOB_ClientSocket(av[1], port);
    fd = c->Connect();
    if (fd < 0) exit(1);
    d = new JACOB_TypeDict();
    Init__msg::init(d);

    y = new JACOB_AsciiInStream(infd, d);

    fprintf(stderr, "infd=%d\n", infd);
    // probably passed a real file on the cmd line
    if ( infd > 0 )
        y->registerReadFunction(JACOB_InStream::fileRead);

    b = y->readObject();
    delete y;
    if ( b == 0 ) {
        fprintf(stderr,
            "failed to read a valid object - try again\n");
        exit(1);
    }
    fprintf(stderr, "object was read ok\n");

    if (av[4][0] == 'A') {
        y = new JACOB_AsciiInStream(fd, d);
        fprintf(stderr,
            "client: opened input stream in ascii mode\n");
    } else if (av[4][0] == 'B') {
        y = new JACOB_BinaryInStream(fd, d);
        fprintf(stderr,
            "client: opened input stream in binary mode\n");
    } else {
        fprintf(stderr,
            "%c not a valid input stream type\n", av[4][0]);
        exit(1);
    }
    if (av[4][1] == 'A') {
        x = new JACOB_AsciiOutStream(fd);
        fprintf(stderr,
            "client: opened output stream in ascii mode\n");
    } else if (av[4][1] == 'B') {
        x = new JACOB_BinaryOutStream(fd);
        fprintf(stderr,
       "client: opened output stream in binary mode\n");
    } else {
        fprintf(stderr,
            "%c not a valid output stream type\n", av[4][1]);
        exit(1);
    }

    y->registerReadFunction(JACOB_InStream::socketRead);
    x->registerWriteFunction(JACOB_OutStream::socketWrite);

#ifndef _WIN32
    gettimeofday(&t1, 0);
#endif
    int count = 0;
    for (;i > 0; i--) {
        x->writeObject(b);
        delete b;
        b = y->readObject();
        if (b == 0) break;
        count++;
    }
    fprintf(stderr, "sent %d messages\n", count);
#ifndef _WIN32
    gettimeofday(&t2, 0);
    long j;
    j = t2.tv_sec - t1.tv_sec;
    j *= 1000;
    j += (t2.tv_usec - t1.tv_usec)/1000;
    fprintf(stderr,
   "Took %ld ms to send/recv %d msgs\n", j, (n - i)*2);
#endif
    return 0;
}
```

**Step Five**: The application can be compiled using

```
g++ -c -Wall -I../.. client.cpp
g++ -o client client.o msg.o -L../../lib/linux/g++-2.95 -ljacob
g++ -c -Wall -I../.. server.cpp
g++ -o server server.o msg.o -L../../lib/linux/g++-2.95 -ljacob
```

If the data is being sent to the server in ASCII form and being sent back to the client in ASCII form, the server may be started using the following command

```
./server 5555 AA
```

The last command line argument specifies the form in which the data object is sent and received. This can be either A (ASCII) or B (binary).

Note that the server can be started in its own window or it can be executed in the background.

The client may be started using the following command

```
./client localhost 5555 200 AA xx.dat
```

This also specifies that the data is being sent and received to/from the server in ASCII form. `xx.dat` is the file that contains the data. For example, it could contain

```
<RealSimple :val 8>
```

The client may also be started without a data file. Objects may be entered manually, by typing them in to the same window the client is running in. e.g. type `<RealSimple :val 8>` and press Enter.

# Index