Progress®



Use JSON with ABL Applications

OpenEdge®

Copyright

© **2019 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.**

These materials and all Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. The information in these materials is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear therein. The references in these materials to specific platforms supported are subject to change.

Corticon, DataDirect (and design), DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, DataRPM, Deliver More Than Expected, Icenium, Kendo UI, Kinvey, NativeScript, OpenEdge, Powered by Progress, Progress, Progress Software Developers Network, Rollbase, SequeLink, Sitefinity (and Design), Sitefinity, SpeedScript, Stylus Studio, TeamPulse, Telerik, Telerik (and Design), Test Studio, and WebSpeed are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries. Analytics360, AppServer, BusinessEdge, DataDirect Autonomous REST Connector, DataDirect Spy, SupportLink, DevCraft, Fiddler, JustAssembly, JustDecompile, JustMock, Kinvey Chat, NativeChat, NativeScript Sidekick, OpenAccess, ProDataSet, Progress Results, Progress Software, ProVision, PSE Pro, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, and WebClient are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

February 2019

Last updated with new content: Release 12.0

Updated: 2019/03/01

Table of Contents

Preface....7

Purpose....7
Audience....7
Organization....8
Using ABL documentation....8
References to ABL compiler and run-time features....8
References to ABL data types....8
Typographical conventions....9
Examples of syntax descriptions....10
Long syntax descriptions split across lines....11
Complex syntax descriptions with both required and optional elements....12
OpenEdge messages....12
Obtaining more information about OpenEdge messages....13

Chapter 1: Using JSON with OpenEdge....15

About JSON....15
Benefits of JSON....16
Benefits of using JSON in OpenEdge....16
JSON basics....16
JSON data types....16
Simple values....17
Complex values....18
Parsing JSON into and serializing JSON from an ABL data object model....18
JavaScript data model....18
Features....18
Use Cases....19
ABL serialization of ProDataSets, Temp-tables, Temp-Table buffer objects to/from JSON....20
Features....20
Use cases....20
ABL support....21

Chapter 2: Parsing and Serializing JSON Objects and Arrays....23

Introduction to the JSON object model in ABL....23
JsonObject....25
JsonArray....25
Parsing JSON strings into objects and arrays....25
Handling object model errors....26

Chapter 3: Reading and Serializing JSON to/from ProDataSets and Temp-tables....29

ABL support for JSON....29
Methods and attributes....30
Data type mapping....31
Writing JSON from a temp-table, temp-table buffer, or a ProDataSet....33
Writing JSON from a ProDataSet....36
Writing JSON from a temp-table....38
Minimizing the size of JSON data....41
Writing JSON from a temp-table buffer's current row....44
Reading JSON into a temp-table, temp-table buffer, or ProDataSet....47
Reading JSON into a data object with defined schema....49
Inferring ABL schemas from JSON data....50
Using JSON with WebSpeed....53

Index....57

Preface

For details, see the following topics:

- Purpose
- Audience
- Organization
- Using ABL documentation
- Typographical conventions
- Examples of syntax descriptions
- OpenEdge messages

Purpose

This book provides a programming guide for OpenEdge® developers who want to use JavaScript Object Notation (JSON) with their ABL (Advanced Business Language) applications.

Audience

This book is intended for ABL programmers working with JSON data in their ABL applications.

Organization

Using JSON with OpenEdge on page 15

This chapter provides an introduction to JSON and an overview of ABL support for serializing ABL data objects to and from JSON.

Parsing and Serializing JSON Objects and Arrays on page 23

This chapter discusses the ABL features that enable you to serialize ABL data objects to and from JSON. It covers data type mapping between ABL and JSON, writing ABL data objects to JSON data, reading JSON data into ABL data objects, and inferring ABL data object schema from JSON data.

Reading and Serializing JSON to/from ProDataSets and Temp-tables on page 29

This chapter discusses the ABL features that enable you to parse and serialize JSON objects and arrays.

Using ABL documentation

OpenEdge provides a special purpose programming language for building business applications. In the documentation, the formal name for this language is *ABL (Advanced Business Language)*. With few exceptions, all keywords of the language appear in all `UPPERCASE`, using a font that is appropriate to the context. All other alphabetic language content appears in mixed case.

For the latest documentation updates go to https://docs.progress.com.

References to ABL compiler and run-time features

ABL is both a compiled and an interpreted language that executes in a run-time engine. The documentation refers to this run-time engine as the *ABL Virtual Machine (AVM)*. When the documentation refers to ABL source code compilation, it specifies *ABL* or *the compiler* as the actor that manages compile-time features of the language. When the documentation refers to run-time behavior in an executing ABL program, it specifies *the AVM* as the actor that manages the specified run-time behavior in the program.

For example, these sentences refer to the ABL compiler's allowance for parameter passing and the AVM's possible response to that parameter passing at run time: "ABL allows you to pass a dynamic temp-table handle as a static temp-table parameter of a method. However, if at run time the passed dynamic temp-table schema does not match the schema of the static temp-table parameter, the AVM raises an error." The following sentence refers to run-time actions that the AVM can perform using a particular ABL feature: "The ABL socket object handle allows the AVM to connect with other ABL and non-ABL sessions using TCP/IP sockets."

References to ABL data types

ABL provides built-in data types, built-in class data types, and user-defined class data types. References to built-in data types follow these rules:

- Like most other keywords, references to specific built-in data types appear in all `UPPERCASE`, using a font that is appropriate to the context. No uppercase reference ever includes or implies any data type other than itself.
- Wherever *integer* appears, this is a reference to the `INTEGER` or `INT64` data type.

- Wherever *character* appears, this is a reference to the CHARACTER, LONGCHAR, or CLOB data type.
- Wherever *decimal* appears, this is a reference to the DECIMAL data type.
- Wherever *numeric* appears, this is a reference to the INTEGER, INT64, or DECIMAL data type.

References to built-in class data types appear in mixed case with initial caps, for example, Progress.Lang.Object. References to user-defined class data types appear in mixed case, as specified for a given application example.

Typographical conventions

This documentation uses the following typographical and syntax conventions:

Convention	Description
Bold	Bold typeface indicates commands or characters the user types, provides emphasis, or the names of user interface elements.
Italic	Italic typeface indicates the title of a document, or signifies new terms.
SMALL, BOLD CAPITAL LETTERS	Small, bold capital letters indicate OpenEdge key functions and generic keyboard keys; for example, **GET** and **CTRL**.
KEY1+KEY2	A plus sign between key names indicates a **simultaneous** key sequence: you press and hold down the first key while pressing the second key. For example, **CTRL+X**.
KEY1 KEY2	A space between key names indicates a **sequential** key sequence: you press and release the first key, then press another key. For example, **ESCAPE H**.
Syntax:	
Fixed width	A fixed-width font is used in syntax, code examples, system output, and file names.
Fixed-width italics	Fixed-width italics indicate variables in syntax.
Fixed-width bold	Fixed-width bold italic indicates variables in syntax with special emphasis.
UPPERCASE fixed width	ABL keywords in syntax and code examples are almost always shown in upper case. Although shown in uppercase, you can type ABL keywords in either uppercase or lowercase in a procedure or class.
Period (.) or colon (:)	All statements except DO, FOR, FUNCTION, PROCEDURE, and REPEAT end with a period. DO, FOR, FUNCTION, PROCEDURE, and REPEAT statements can end with either a period or a colon.
[ ]	Large brackets indicate the items within them are optional.
[ ]	Small brackets are part of ABL.

Convention	Description
{ }	Large braces indicate the items within them are required. They are used to simplify complex syntax diagrams.
{ }	Small braces are part of ABL. For example, a called external procedure must use braces when referencing arguments passed by a calling procedure.
|	A vertical bar indicates a choice.
. . .	Ellipses indicate repetition: you can choose one or more of the preceding items.

Examples of syntax descriptions

In this example, `ACCUM` is a keyword, and *`aggregate`* and *`expression`* are variables:

Syntax

```
ACCUM aggregate expression
```

`FOR` is one of the statements that can end with either a period or a colon, as in this example:

```
FOR EACH Customer NO-LOCK:
  DISPLAY Customer.Name.
END.
```

In this example, `STREAM` *`stream`*, `UNLESS-HIDDEN`, and `NO-ERROR` are optional:

Syntax

```
DISPLAY [ STREAM stream ] [ UNLESS-HIDDEN ] [ NO-ERROR ]
```

In this example, the outer (small) brackets are part of the language, and the inner (large) brackets denote an optional item:

Syntax

```
INITIAL [ constant [ , constant ] ]
```

A called external procedure must use braces when referencing compile-time arguments passed by a calling procedure, as shown in this example:

Syntax

```
{ &argument-name }
```

In this example, EACH, FIRST, and LAST are optional, but you can choose only one of them:

Syntax

```
PRESELECT [ EACH | FIRST | LAST ] record-phrase
```

In this example, you must include two expressions, and optionally you can include more. Multiple expressions are separated by commas:

Syntax

```
MAXIMUM ( expression , expression [ , expression ] ... )
```

In this example, you must specify MESSAGE and at least one *expression* or SKIP [(*n*)], and any number of additional *expression* or SKIP [(*n*)] is allowed:

Syntax

```
MESSAGE { expression | SKIP [ ( n ) ] } ...
```

In this example, you must specify {*include-file*, then optionally any number of *argument* or &*argument-name* = "*argument-value*", and then terminate with }:

Syntax

```
{ include-file
    [ argument | &argument-name = "argument-value" ] ... }
```

Long syntax descriptions split across lines

Some syntax descriptions are too long to fit on one line. When syntax descriptions are split across multiple lines, groups of optional and groups of required items are kept together in the required order.

In this example, WITH is followed by six optional items:

Syntax

```
WITH [ ACCUM max-length ] [ expression DOWN ]
  [ CENTERED ] [ n COLUMNS ] [ SIDE-LABELS ]
  [ STREAM-IO ]
```

Complex syntax descriptions with both required and optional elements

Some syntax descriptions are too complex to distinguish required and optional elements by bracketing only the optional elements. For such syntax, the descriptions include both braces (for required elements) and brackets (for optional elements).

In this example, `ASSIGN` requires either one or more *`field`* entries or one *`record`*. Options available with *`field`* or *`record`* are grouped with braces and brackets:

Syntax

```
ASSIGN   { [ FRAME frame ] { field [ = expression ] }
           [ WHEN expression ] } . . .
       | { record [ EXCEPT field . . . ] }
```

OpenEdge messages

OpenEdge displays several types of messages to inform you of routine and unusual occurrences:

- **Execution messages** inform you of errors encountered while OpenEdge is running a procedure; for example, if OpenEdge cannot find a record with a specified index field value.
- **Compile messages** inform you of errors found while OpenEdge is reading and analyzing a procedure before running it; for example, if a procedure references a table name that is not defined in the database.
- **Startup messages** inform you of unusual conditions detected while OpenEdge is getting ready to execute; for example, if you entered an invalid startup parameter.

After displaying a message, OpenEdge proceeds in one of several ways:

- Continues execution, subject to the error-processing actions that you specify or that are assumed as part of the procedure. This is the most common action taken after execution messages.
- Returns to the Procedure Editor, so you can correct an error in a procedure. This is the usual action taken after compiler messages.
- Halts processing of a procedure and returns immediately to the Procedure Editor. This does not happen often.
- Terminates the current session.

OpenEdge messages end with a message number in parentheses. In this example, the message number is 200:

```
** Unknown table name table. (200)
```

If you encounter an error that terminates OpenEdge, note the message number before restarting.

Obtaining more information about OpenEdge messages

In Windows platforms, use OpenEdge online help to obtain more information about OpenEdge messages. Many OpenEdge tools include the following Help menu options to provide information about messages:

- Choose **Help** > **Recent Messages** to display detailed descriptions of the most recent OpenEdge message and all other messages returned in the current session.
- Choose **Help** > **Messages** and then type the message number to display a description of a specific OpenEdge message.
- In the Procedure Editor, press the **HELP** key or **F1**.

On UNIX platforms, use the OpenEdge `pro` command to start a single-user mode character OpenEdge client session and view a brief description of a message by providing its number.

To use the pro command to obtain a message description by message number:

1. Start the Procedure Editor:

   ```
   OpenEdge-install-dir/bin/pro
   ```

2. Press **F3** to access the menu bar, then choose **Help** > **Messages**.
3. Type the message number and press **ENTER**. Details about that message number appear.
4. Press **F4** to close the message, press **F3** to access the Procedure Editor menu, and choose **File** > **Exit**.

1

Using JSON with OpenEdge

For details, see the following topics:

- About JSON
- JSON basics
- Parsing JSON into and serializing JSON from an ABL data object model
- ABL serialization of ProDataSets, Temp-tables, Temp-Table buffer objects to/from JSON

About JSON

JavaScript Object Notation (JSON) is a data interchange format created from a subset of JavaScript. The Internet Engineering Task Force's RFC 4627 describes JSON as follows:

"JavaScript Object Notation (JSON) is a lightweight, text-based, language-independent data interchange format. It was derived from the ECMAScript Programming Language Standard. JSON defines a small set of formatting rules for the portable representation of structured data."

JSON developers use JSON as an alternative data interchange format to XML. XML is widely used to exchange data in a heterogeneous environment. However, some developers consider XML as too verbose for exchanges between a web browser and a web server as part of a rich internet application. For more information about XML, see *Use XML with ABL Applications*.

JSON's appeal as an alternative to XML comes from smaller HTTP messages and less complicated syntax. The smaller messages and simpler syntax can provide a significant performance advantage over XML. As a result, JSON is popular for rich internet applications, especially AJAX applications. JSON's popularity in AJAX applications makes it ideal for WebSpeed® developers.

Benefits of JSON

JSON offers the same kind of benefits that XML does for exchanging data in a heterogeneous environment, such as the following:

- JSON is self-describing. The syntax and hierarchical structure of the JSON strings can in some cases be interpreted by applications that do not already know what data to expect.
- JSON is simple text. This fact makes it suitable and safe for transferring across platforms and operating systems that do not readily share more complex document types. As text, JSON can also be readily displayed and edited in simple editors.
- JSON is compact. An average JSON string is about two thirds of the size of the same data in XML.
- JSON is easy to learn, easy to read, and easy to understand.

Benefits of using JSON in OpenEdge

Some of the benefits of using JSON in your OpenEdge development work include:

- OpenEdge includes built-in parsers for reading and writing JSON
- JSON parser functionality is accessed using the same types of ABL objects, methods, and attributes as the built-in XML parsers
- ABL JSON-enablement features are simple enough to quickly implement basic use cases
- ABL JSON-enablement features can provide improved performance with, as well as native communication to, AJAX applications

JSON basics

This section provides a brief overview of JSON with the following topics:

- JSON data types on page 16
- Simple values on page 17
- Complex values on page 18

For more information about JSON, start with the JSON Web site, http://www.progress.com/esd/index.ssp. This site includes basic information about JSON and links to other sites describing support for JSON in various languages and applications.

JSON data types

Because JSON is designed to be lightweight, it supports only four primitive data types, as shown in the following table.

Table 1: JSON primitive data types

Data type	Description	Examples
string	A string of Unicode characters enclosed in double quotes. A backslash (\ ) serves as the escape character.	"jump rope"
number	An unquoted numeric value, which can include an exponent	1754.350.9582e-42
boolean	An unquoted lowercase literal string of true or false	true
null	An unquoted lowercase literal string of null	null

The data type of a value is determined by the format of the value. In addition to these primitive data types, there are some non-standard data types in common usage for certain values. For more information about ABL support of non-standard data types, see Data type mapping on page 31.

JSON also supports two complex data types, as shown in the following table:

Table 2: JSON complex data types

Data type	Description	Examples
Object	A comma-delimited list of named values, either simple or complex, enclosed in braces	{ "myString" : "jump rope", "myNum" : 17, "myBool" : false }
Array	A comma-delimited list of unnamed values, either simple or complex, enclosed in brackets	["jump rope", 17, false]

Simple values

A *simple value* is a name/value pair. The name is always quoted and separated from the value by a colon, as shown:

```
"name" : value
```

The exception to this is the list of values in an array, as shown in the above table. In an array, you access a particular value by a numeric index, rather than by a name.

Complex values

JSON enables you to name objects and arrays and combine them into complex values. A *complex value* combines simple values to represent more complicated data. The following example shows an object made up of other objects and arrays:

```
{
  "salesRep" : { "name" : "Dorothy Gale",
                 "age" : 38,
                 "region" : "Kansas, USA"
               },
  "tractorSales" : { "2009Quarterly" : [ 13, 27, 18, 9 ],
                     "2008Quarterly" : [ 11, 17, 32, 5 ],
                     "2007Quarterly" : [ 9, 25, 16, 10 ]
                   }
}
```

Note: The previous JSON example has been formatted for easier reading.

Parsing JSON into and serializing JSON from an ABL data object model

This section provides a brief overview of JSON and includes the following topics:

- JavaScript data model on page 18
- Features on page 18
- Use Cases on page 19
- ABL Support on page 19

JavaScript data model

JSON objects and arrays in the ABL are based on the objects and arrays of the JavaScript data model.

JavaScript objects are associative arrays. Object property names can be any Unicode string. Properties and their values can be added, changed, or deleted at runtime. In addition to the properties being added and deleted, the data type of a property can also be changed at runtime.

A JavaScript array is also a type of JavaScript object. In these arrays, the properties are accessed by a numeric index. Each property is referred to as an element. Each element can have its data type changed at runtime. Hence, it supports heterogeneous arrays (arrays with elements that do not all have the same data type).

Features

The ABL JSON object model allows you to:

- Parse JSON strings or read ABL data objects (such as temp-tables and ProDataSets) into a hierarchy of JSON object and JSON array classes
- Update the individual elements (properties) of each object or array in the hierarchy
- Serialize the entire JSON object hierarchy to a variety of JSON string types (memory-resident character strings, files, etc.)

Use Cases

The JSON object model provides the flexibility to interact with JSON in a rich internet application (RIA) that you might develop using WebSpeed or AppServer back end. Frameworks such as AJAX contain JavaScript object libraries with full JSON support. A typical use of the API might be as follows:

1. The AppServer application receives a request containing JSON.
2. The JSON is parsed to create a tree of JsonObjects and JsonArrays.
3. Based on the content of the generated objects, the application responds to the request, and gathers results.
4. These results are used to modify the existing tree or create a new tree of JSON objects.
5. The application serializes the resulting tree and sends the serialized JSON back as a response.
6. The application deletes the tree(s) of JSON objects.

ABL Support

ABL provides various built-in classes that support the JavaScript object model:

Progress.Json.ObjectModel.ObjectModelParser class

This class provides public methods that parse a JSON string from a variety of sources and builds a corresponding JSON object hierarchy.

Progress.Json.ObjectModel.JsonObject class

This class encapsulates a single JSON object, which can contain any number of properties(name/value pairs) of various JSON data types, including other JSON objects or arrays. It provides public methods to access and change the object's data in a number of ways.

Progress.Json.ObjectModel.JsonArray class

This class encapsulates a single JSON array, which can contain any number of elements of various JSON data types, including other JSON objects or arrays. It provides public methods to access and change the array's data in a number of ways.

Progress.Json.ObjectModel.JsonDataType

This class provides public properties that allow you to identify the JSON data type of a given property or element in a JSON object or array.

For more information on classes supporting JavaScript object model, see, Parsing and Serializing JSON Objects and Arrays on page 23

ABL serialization of ProDataSets, Temp-tables, Temp-Table buffer objects to/from JSON

ABL temp-tables and ProDataSets have the ability to serialize their data to a JSON string. Similarly, you can read JSON data into a temp-table or ProDataSet. These features parallel the features for XML serialization of temp-tables and ProDataSets.

Features

The JSON features of temp-tables and ProDataSets allow you to take advantage of their rich relational features while providing a standards-based method for sharing data with other application. These JSON features include the following:

- Read JSON data to populate an empty temp-table or ProDataSet
- Read JSON data into temp-tables and ProdataSets that already contain data
- Write JSON data from a temp-table or ProDataSet to a JSON string, a JsonArray object, or a JsonObject object
- Write JSON data from the current row in a temp-table buffer to a JSON string or JsonObject object

Unlike the matching XML features, you cannot perform a perfect round-trip of JSON write/reads. If the ABL virtual machine (AVM) parses an incoming JSON string into a dynamic temp-table or ProDataSet that does not already have a schema, the AVM must infer the schema from the data. This process means that you are unlikely to have data objects with identical schema on both sides of a transaction. Meta data, like key fields, is lost in the serialization to and from JSON.

The JSON features are available as attributes and methods on:

- Temp-table objects
- Temp-table buffer objects (can act on the entire temp-table or the buffers' current contents)
- ProDataSet objects
- JsonObject or JsonArray objects

Use cases

The JSON read and write features are robust and versatile. The following examples demonstrate common use cases that can be solved with the features:

- Provide interoperability between OpenEdge and another JSON-enabled platform or application
- Use JSON data as a persistent storage mechanism between ABL sessions
- Simplify or replace existing ABL code that performs JSON reads and writes on temp-tables, temp-table buffers, and ProDataSets

ABL support

The serialization features are provided by the following methods on buffers, temp-tables, and ProDataSets:

- **`READ-JSON( )` method** — Reads a JSON string into a temp-table or ProDataSet
- **`SERIALIZE-ROW( )` method** — Writes the current contents of a buffer to a JSON string
- **`WRITE-JSON( )` method** — Writes the contents of a temp-table or ProDataSet to a JSON string

ABL also supports the following methods:

- **`JsonObject:Read( )` method** — Reads the data from a ProDataSet, Temp-table or Temp-table buffer object to generate a JSON Object.
- **`JsonArray:Read( )` method** — Reads the data from Temp-Table object to generate a JSON Array.

You do not need to be familiar with JSON to use these methods and the associated attributes.

2

Parsing and Serializing JSON Objects and Arrays

For details, see the following topics:

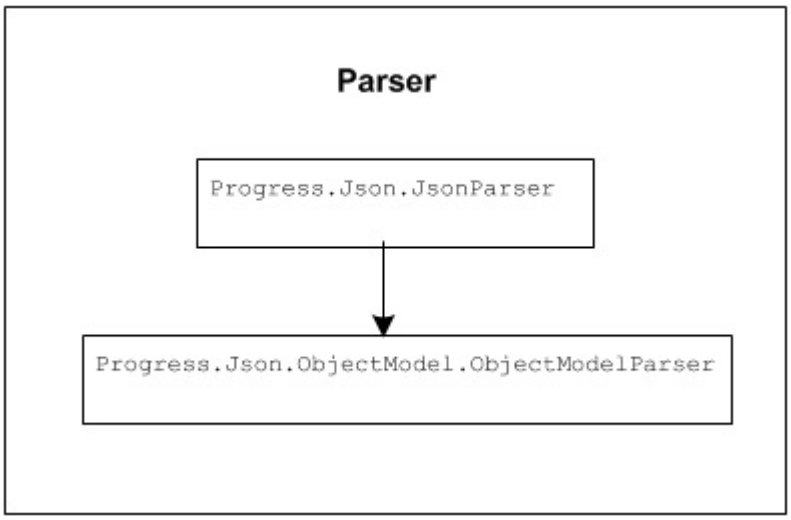
- Introduction to the JSON object model in ABL
- Parsing JSON strings into objects and arrays
- Handling object model errors

Introduction to the JSON object model in ABL

The object model represented by the `Progress.Json.ObjectModel` package is based on the JavaScript concepts of objects and arrays. The programming model for the ABL JSON parser is to complete the parsing all at once to generate a tree of constructs. These constructs represent JSON objects and arrays.

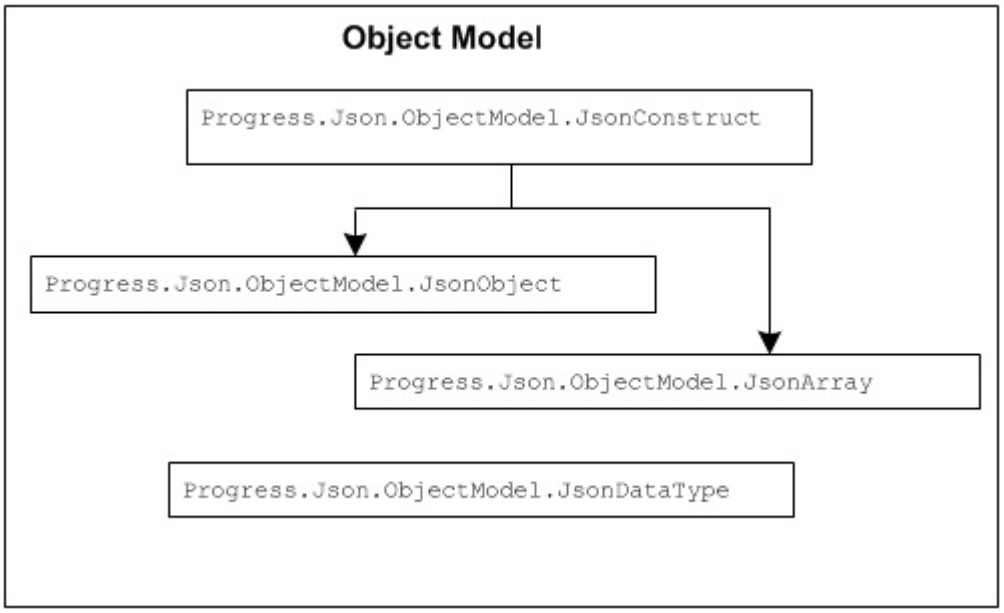
The following figure shows the inheritance relationship between the parser classes:

Figure 1: Parser class hierarchy



The following figure shows the inheritance relationship between the classes in the Progress.Json.ObjectModel package:

Figure 2: JSON Object Model Hierarchy



Note: Objects of type `JsonObject`, `JsonArray`, and `ObjectModelParser` from the `Progress.Json.ObjectModel` package can be passed between an AppServer and an ABL client.

JsonObject

The ABL JsonObject is a map collection. It maps string keys to values. Each key/value pair is called a property. The object's properties are not listed in any specific order. The only way to reference a property is by name.

Because property names are Unicode strings, there is no limitation on the nomenclature of the string. For instance, whitespace characters may also be included in a property name.

JsonArray

The ABL JsonArray is an array collection. It maps integer keys to values. Each key/value pair is called an element, and the only way to access an array element's value is by index in the array. There is a strict order to an array's elements. A JsonArray can grow and shrink in size. Elements can be inserted into and removed from the middle of a JavaScript array.

JavaScript values are loosely typed. That means, the data type of a value within a JavaScript object or array can be changed by setting a new value.

When a property or element is set to null, it loses its data type. In a sense null is a data type (or lack of data type) as well as a value (or lack of value).

Unlike ABL Arrays, JsonArrays can be heterogeneous, meaning that the elements within the array can be of different data types.

The following is an example of building a heterogeneous JsonArray:

```
myArray = NEW JsonArray().
myArray:Add(1).
myArray:Add(2).
myArray:Add(3).
myArray:Add(FALSE).
myArray.Add("jump rope").
myArray:Write(myLongchar, TRUE).
```

The first three elements are numbers, the fourth element is a boolean, and the fifth element is a string.

The following is the result of myLongchar:

```
[1, 2, 3, false, "jump rope"].
```

Parsing JSON strings into objects and arrays

The ABL JSON object model provides various classes to complete the parsing and generating of a tree of constructs that represent JSON objects and arrays. The two parser classes which help parsing of JSON strings are `Progress.Json.JsonParser` and `Progress.Json.ObjectModel.JsonModelParser`.

JsonParser is an abstract class that contains properties common to JSON parsers supported by ABL.

The `Parse( )` method from the ObjectModelParser class is used to identify the source of the JSON string and parses the string. The value the method returns is either a `Progress.Json.ObjectModel.JsonObject` or a `Progress.Json.ObjectModel.JsonArray`.

The `ParseFile( )` method from the ObjectModelParser class parses the JSON from the specified file. The value the method returns is either a `Progress.Json.ObjectModel.JsonObject` or a `Progress.Json.ObjectModel.JsonArray`.

The following example shows the reuse of a parser:

```
DEFINE VARIABLE myLongchar AS LONGCHAR NO-UNDO.
DEFINE VARIABLE myParser   AS ObjectModelParser NO-UNDO.
DEFINE VARIABLE Request    AS JsonConstruct NO-UNDO.
DEFINE VARIABLE settings   AS JsonObject NO-UNDO.
. . .
myParser = NEW ObjectModelParser().
settings = CAST(myParser:Parse(myLongchar), JsonObject).
Request = myParser:Parse(&WEBSTREAM).
```

In the example above, the AVM parses a LONGCHAR to initialize some settings for the procedure, and then reads and parses from the WebSpeed input stream.

Handling object model errors

ABL provides classes to raise errors when a failure specific to the `Progress.Json` family of packages occurs. For example, a JsonError is raised when a parameter's value is out of the permissible range for a JsonArray or a reference is made to a non-existent property in a JsonObject. The following table describes the available classes for handling object model errors.

The following figure shows the inheritance relationship between the various error classes:

Figure 3: Error class hierarchy

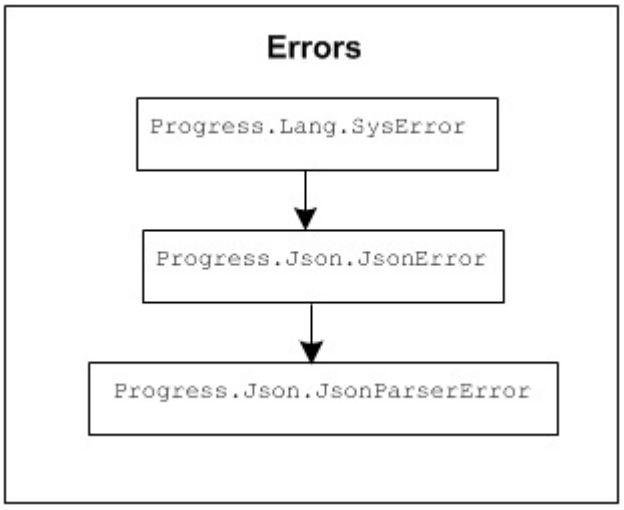


Table 3: JSON classes for object model errors

Class	Description
`Progress.Json.JsonError`	A JsonError is raised when a failure specific to the `Progress.Json` family of packages occurs.
`Progress.Json.JsonParserError`	A JsonParserError is raised when the syntax of the JSON being parsed is incorrect.

For more information, see the descriptions of these classes in *ABL Reference*.

3

Reading and Serializing JSON to/from ProDataSets and Temp-tables

The AVM can serialize the data from ABL temp-tables and ProDataSets to JSON data. Similarly, you can read or read and load JSON data into a temp-table or ProDataSet.

For details, see the following topics:

- ABL support for JSON
- Data type mapping
- Writing JSON from a temp-table, temp-table buffer, or a ProDataSet
- Writing JSON from a temp-table buffer's current row
- Reading JSON into a temp-table, temp-table buffer, or ProDataSet
- Using JSON with WebSpeed

ABL support for JSON

The ABL support for serializing data objects to and from JSON parallels the ABL support for serializing those data objects into XML. The ABL support for JSON includes the following:

- Serializing to and from JSON with methods on ProDataSet, temp-table, and temp-table buffer object handles
- Serializing before-image data for ProDataSets

- Serializing fields with data types based on ABL class definitions (see the `WRITE-JSON( )` entry in *ABL Reference* for more details and restrictions)
- Affecting JSON serialization through attributes on the data object handles
- Nesting temp-tables
- Hiding foreign fields
- Omitting initial values
- Identifying JSON strings through an attribute on the `WEB-CONTEXT` system handle
- Built in ABL objects to represent JSON

Note: ABL **does not** support serializing schema to and from JSON

Methods and attributes

To use the JSON read and write features described in this chapter, use the handle to a temp-table, temp-table buffer, or ProDataSet to access methods and attributes of the object. The following table describes the available methods.

Table 4: JSON methods for temp-tables and ProDataSets

Method	Description
`READ-JSON( )`	Reads a specified JSON string, JsonArray object, or JsonObject object into a corresponding ProDataSet, a temp-table, or a temp-table buffer object.
`WRITE-JSON( )`	Writes a specified JSON string, JsonArray object, or JsonObject object from a corresponding ProDataSet, a temp-table, or a temp-table buffer object. In the case of the temp-table buffer object, all the records of the temp-table associated with the buffer object are written to the JSON string, JsonArray object, or JsonObject object—not just the one record in the buffer.
`SERIALIZE-ROW ( )`	Serializes a temp-table buffer's current row either to a JSON or XML string or to a JsonObject object.

The following table describes the JSON related attributes.

Table 5: JSON related attributes for temp-tables and ProDataSets

Attribute	Data type	Applies to	Description
FOREIGN-KEY-HIDDEN[1] (Readable and writeable)	LOGICAL	Data-relation	Specifies whether the WRITE-JSON( ) method should hide foreign key fields in the child records of a nested data-relation in a ProDataSet.
NESTED (Readable and writeable)	LOGICAL	Data-relation	Specifies whether the AVM embeds child rows within a parent row in the JSON. This affects both the data and schema.
SERIALIZE-HIDDEN (Readable and writeable)	LOGICAL	Buffer-field	Indicates whether this field is written when the temp-table is serialized, for example into JSON or XML. This attribute als interacts with the XML-NODE-TYPE attribute.
SERIALIZE-NAME (Readable and writeable)	CHARACTER	ProDataSet temp-table temp-table buffer temp-table buffer field	Optionally specifies the name of a ProDataSet, a temp-table, a temp-table buffer, or a temp-table buffer-field object as it should appear when serialized, for example into JSON or XML. This attribute also interacts with the XML-NODE-NAME attribute.

Note: The SERIALIZE-HIDDEN and SERIALIZE-NAME attributes provide generalized support for serializing data objects to either JSON or XML. These attributes interact with the XML-specific attributes in ABL. For more information, see the descriptions of these attributes in *ABL Reference*.

Data type mapping

In JSON basics on page 16, you saw that JSON uses only a few primitive data types. In addition, the JSON community has created a few non-standard data types to cover a few other common data types.

The following table shows the non-standard JSON data types that ABL supports.

Table 6: Supported non-standard JSON data types

Data type	JSON representation
Date-time	A JSON string in the ISO 8601 format, "yyyy-mm-ddThh:mm:ss.sss+hh:mm"
Binary data	A JSON string consisting of the Base64 encoded equivalent of the binary data

[1] See Minimizing the size of JSON data on page 41 for more information.

For serializing ABL data objects to JSON strings, only the ABL data types that you can assign to a temp-table field are mapped to JSON data types. The exception to this is the ABL `CLASS` data type. The ABL serialization features do no support serializing fields with data types based on ABL class definitions.

The following table shows how the AVM maps ABL data types to JSON data types. These mappings apply when the AVM writes ABL data to JSON and when the AVM reads JSON data into an ABL data object with a defined schema.

Table 7: ABL to JSON data type mapping

ABL data type	JSON data type
`BLOB`	`string` (Base64 encoded)
`CHARACTER`	`string`
Class-based object	`Object`
`CLOB`	`string`
`COM-HANDLE`	`number`
`DATE`	`string` (ISO 8601 formatted string of the form `"yyyy-mm-dd"`)
`DATETIME`	`string` (ISO 8601 formatted string of the form `"yyyy-mm-ddThh:mm:ss.sss"`)
`DATETIME-TZ`	`string` (ISO 8601 formatted string of the form `"yyyy-mm-ddThh:mm:ss.sss+hh:mm"`)
`DECIMAL`	`number`
`HANDLE`	`number`
`INT64`	`number`
`INTEGER`	`number`
`LOGICAL`	`boolean` (`true` or `false`)
`RAW`	`string` (Base64 encoded)
`RECID`	`number`
`ROWID`	`string` (Base64 encoded)

Note:

When parsing values to or from JSON, the AVM equates the ABL Unknown value (`?`) to a JSON null value.

If a JSON string's value is too long for the ABL data type, the AVM generates an error message and the `READ-JSON( )` method returns `FALSE`.

If a JSON number's value is out of the ABL data type's range, the AVM generates an error message and the `READ-JSON( )` method returns `FALSE`.

If you read JSON data into an ABL data object that has a schema, the AVM assumes that the JSON values are formatted appropriately for the data object's schema. For example, if the AVM reads JSON data into an ABL temp-table field of the `DATETIME` data type, the AVM assumes that the value is a string in the ISO 8601 format. If the JSON value cannot be converted to the expected ABL data type, `READ-JSON( )` generates an error message and returns `FALSE`.

If you read a JSON string into a dynamic ABL data object that does not have a schema, the AVM infers the schema from the JSON data. When you read JSON into an ABL data object with an inferred schema, the AVM uses different rules to assign ABL data types to JSON values. With the inferred data types, the AVM makes no attempt to determine if the JSON values represent one of the non-standard data types. For example, all quoted values are mapped to the `CHARACTER` data type. For more information about inferring schema, see Inferring ABL schemas from JSON data on page 50.

Writing JSON from a temp-table, temp-table buffer, or a ProDataSet

The `WRITE-JSON( )` method writes a JSON string, `Progress.Json.ObjectModel.JsonArray` object, or `Progress.Json.ObjectModel.JsonObject` object containing the data from a corresponding temp-table, temp-table buffer, or ProDataSet object handle on which you call the method. If you call the method on a temp-table buffer handle, it writes the entire contents of the associated temp-table, not just the contents of the buffer. When calling the method on a ProDataSet object handle, `WRITE-JSON( )` writes the current version of data in each row of each table in the ProDataSet object. The AVM ignores any before-image data for a ProDataSet unless you specify an option to include it.

Note that you cannot call `WRITE-JSON( )` on a database buffer handle.

Note also that you cannot write a JsonArray from a ProDataSet object. If you call the method on a ProDataSet object handle to write a JsonArray, `WRITE-JSON( )` generates an error message and returns a value of `FALSE`.

For JSON string output only, `WRITE-JSON( )` provides the option to format the JSON in a hierarchical manner using extra white space and line breaks.

`WRITE-JSON( )` does not write the data object's schema to JSON, because JSON has no standardized support for schema. The lack of schema information means that the JSON string does not explicitly include indexes and data-relations.

The syntax for `WRITE-JSON( )` is shown below. The method returns `TRUE` or `FALSE` to indicate whether the operation was successful.

Syntax

```
WRITE-JSON ( target-type ,
  { file | stream | stream-handle | memptr | longchar
    | JsonArray | JsonObject }
  [ , formatted [ , encoding [ , omit-initial-values [ , omit-outer-object
  [ , write-before-image ] ] ] ] ] ] )
```

target-type

A CHARACTER expression that specifies the target type for the JSON output. Valid values are "FILE", "STREAM", "STREAM-HANDLE", "MEMPTR", "LONGCHAR", "JsonArray", and "JsonObject".

All values except "JsonArray" and "JsonObject" specify a target for a JSON string.

file

A CHARACTER expression that specifies the name of a file to which the AVM writes the JSON string. You can specify an absolute pathname or a pathname relative to the current working directory. If a file with the specified name already exists, the AVM verifies that the file is writeable and overwrites the file.

stream

A CHARACTER expression that specifies the name of a stream. If you specify the empty string (""), the AVM writes the JSON string to the default unnamed output stream. For WebSpeed, write the JSON string to the WebSpeed-defined output stream (WEBSTREAM).

stream-handle

A HANDLE variable that specifies a stream object handle.

memptr

A MEMPTR variable to contain the JSON string in memory. If you do not specify the encoding parameter, the AVM encodes the text written to the MEMPTR as "UTF-8". This method allocates the required amount of memory for the JSON string and sets the size of the variable. When you are finished using the MEMPTR, you must free the associated memory, by setting the MEMTER to zero bytes with the SET-SIZE statement.

longchar

A LONGCHAR variable to contain the JSON string in memory.

The AVM saves the JSON string to the LONGCHAR variable in the code page that corresponds to the character encoding you specify in the *encoding* option. If you do not specify a character encoding for the JSON string, the AVM saves the LONGCHAR variable in "UTF-8".

If the LONGCHAR variable's code page is fixed (that is, set using the FIX-CODEPAGE statement) and the fixed code page is not equivalent to the character encoding you specify in the *encoding* option, the WRITE-JSON( ) method generates an error and returns FALSE. The JSON string is not saved to the LONGCHAR.

JsonArray

A reference to a Progress.Json.ObjectModel.JsonArray object to which the AVM writes the contents of the associated temp-table. Any prior contents of the specified *JsonArray* are removed.

The *JsonArray* must be a valid instance of type Progress.Json.ObjectModel.JsonArray. If not, the WRITE-JSON( ) method generates an error message and returns FALSE.

JsonObject

A reference to a Progress.Json.ObjectModel.JsonObject object to which the AVM writes the contents of the associated ProDataSet or temp-table. Any prior contents of the specified *JsonObject* are removed.

The *JsonObject* must be a valid instance of type Progress.Json.ObjectModel.JsonObject. If not, the WRITE-JSON( ) method generates an error message and returns FALSE.

formatted

An optional LOGICAL expression where TRUE directs the AVM to format the JSON string in a hierarchical manner using extra white space, carriage returns, and line feeds. The default value is FALSE. If you specify the Unknown value (?), the method uses the default value of FALSE. The *formatted* option is ignored for *JsonArray* and *JsonObject* targets.

encoding

An optional CHARACTER expression that specifies the name of the character encoding the AVM uses to write the JSON output. The default encoding is "UTF-8".

The encoding name must specify a Unicode transformation format. Valid values are "UTF-8", "UTF-16", "UTF-16BE", "UTF-16LE", "UTF-32", "UTF-32BE", and "UTF-32LE".

Note: If you specify the empty string ("") or the Unknown value (?), the AVM uses the default encoding of "UTF-8".

---

omit-initial-values

An optional LOGICAL expression where TRUE directs the AVM to exclude temp-table fields containing their initial values from the JSON output, and FALSE directs the AVM to include all temp-table field data in the output. The default value is FALSE. If you specify the Unknown value (?), the method uses the default value of FALSE.

For more information about this option, see Minimizing the size of JSON data on page 41 .

omit-outer-object

An optional LOGICAL expression that indicates whether the outer-most object is included in the JSON output. TRUE directs the AVM to remove the outer-most object on output. FALSE directs the AVM to include the outer-most object in the output.

For more information about this option, see Omitting the outer object in JSON data on page 41.

write-before-image

An optional LOGICAL expression where TRUE directs the AVM to include ProDataSet before-image data and error information in the JSON output. The default value is FALSE.

This element can only be set to TRUE for a ProDataSet. If its value is set to TRUE for a temp-table or buffer handle, the WRITE-JSON( ) method returns an error and a value of FALSE.

Writing JSON from a ProDataSet

The following code example defines a static ProDataSet object, attaches its data sources, fills the ProDataSet object, and writes the ProDataSet object to a JSON string:

```
/* write-json-pds1.p */
{pi-json-parameterVarDefs.i} /* parameter variable definitions */
{pi-write-json-pds1.i}       /* dsOrderLog definition - no nesting */

DEFINE VARIABLE hdsOrderLog AS HANDLE  NO-UNDO.
DEFINE VARIABLE lRetOK      AS LOGICAL NO-UNDO.

hdsOrderLog = DATASET dsOrderLog:HANDLE.

DATA-SOURCE dsCustomer:FILL-WHERE-STRING = "WHERE Customer.CustNum = 2 ".
DATASET dsOrderLog:FILL().

ASSIGN
  cTargetType = "file"
  cFile       = "dsOrderLog.json"
  lFormatted  = TRUE
  cEncoding   = ?.

lRetOK = hdsOrderLog:WRITE-JSON(cTargetType, cFile, lFormatted).
```

The following is an excerpt of the JSON produced by this procedure:

```
{"dsOrderLog": {
  "ttCustomer": [
    {
      "CustNum": 2,
       • • •
    }
  ],
  "ttOrder": [
    {
      "Ordernum": 94,
       • • •
    },
    • • •
    {
      "Ordernum": 6070,
       • • •
    }
  ],
  "ttInvoice": [
    {
      "Invoicenum": 94,
       • • •
    },
    {
      "Invoicenum": 124,
       • • •
    }
  ]
}}
```

Because the ProDataSet definition did not include the `NESTED` option for the data-relations, the records from each temp-table are presented after each other. If you do not nest child tables, the JSON does not contain the data relation information. The serialization process also loses any information about key columns.

If you run `write-json-pds2.p` which uses a ProDataSet with nested child tables, the resulting JSON looks like this:

```
{"dsOrderLog": {
  "ttCustomer": [
    {
      "CustNum": 2,
       • • •
      "EmailAddress": "",
      "ttOrder": [
        {
          "Ordernum": 94,
           • • •
          "Carrier": "Standard Mail",
          "ttInvoice": [
            {
              "Invoicenum": 94,
               • • •
            }
          ]
        },
        {
```

```
          "Ordernum": 125,
          • • •
          "Carrier": "FlyByNight Courier",
          "ttInvoice": [
            {
              "Invoicenum": 124,
              • • •
            }
          ]
        },
        • • •       ]
    }
  ]
}}
```

Writing JSON from a temp-table

The following code example defines a static temp-table, fills the temp-table object, and writes the temp-table object to a JSON string:

```
/* write-json-tt.p */
{pi-json-parameterVarDefs.i} /* parameter variable definitions */
{pi-write-json-tt.i}         /* ttCust definition */

DEFINE VARIABLE httCust      AS HANDLE  NO-UNDO.
DEFINE VARIABLE lReturnValue AS LOGICAL NO-UNDO.

httCust = TEMP-TABLE ttCust:HANDLE.

FOR EACH Customer WHERE CustNum < 4:
  CREATE ttCust.
  BUFFER-COPY Customer TO ttCust.
END.

ASSIGN
  cTargetType = "FILE"
  cFile       = "ttCust.json"
  lFormatted  = TRUE
  cEncoding   = ?.

lReturnValue = httCust:WRITE-JSON(cTargetType, cFile, lFormatted, cEncoding).
```

The following is an excerpt of the JSON produced by this procedure:

```
{"ttCust": [
  {
    "CustNum": 1,
    "Country": "USA",
    "Name": "Lift Tours",
    "Address": "276 North Drive",
    "Address2": "",
    "City": "Burlington",
    "State": "MA",
    "PostalCode": "01730",
    "Contact": "Gloria Shepley",
    "Phone": "(617) 450-0086",
    "SalesRep": "HXM",
```

```
    "CreditLimit": 66700.00,
    "Balance": 903.64,
    "Terms": "Net30",
    "Discount": 35,
    "Comments": "This customer is on credit hold.",
    "Fax": "",
    "EmailAddress": ""
  },
  {
    "CustNum": 2,
    • • •  },
  {
    "CustNum": 3,
    • • •  }
]}
```

Writing JSON from a ProDataSet with before-image data

The following code example defines and fills a static ProDataSet object, turns on tracking, and makes a change to the ProDataSet. The code then writes the ProDataSet object to a JSON string and includes the before-image data:

```
/* write-json-pds3.p */
{pi-json-parameterVarDefs.i} /* parameter variable definitions */
{pi-write-json-pds1.i} /* dsOrderLog definition - no nesting */

DEFINE VARIABLE hdsOrderLog AS HANDLE NO-UNDO.
DEFINE VARIABLE lRetOK AS LOGICAL NO-UNDO.

hdsOrderLog = DATASET dsOrderLog:HANDLE.
DATA-SOURCE dsCustomer:FILL-WHERE-STRING = "WHERE Customer.CustNum = 2 ".
DATASET dsOrderLog:FILL().

/* set tracking-changes so that before-table records are created */
TEMP-TABLE ttCustomer:TRACKING-CHANGES = TRUE.
TEMP-TABLE ttCustomer:TRACKING-CHANGES = TRUE.

/* modify ttCustomer 2 */
FIND ttCustomer WHERE ttCustomer.CustNum = 2.
ASSIGN ttCustomer.EmailAddress = "www.progress.com".

/* set error string on ttCustomer 2 */
BUFFER ttCustomer:ERROR = TRUE.
BUFFER ttCustomer:ERROR-STRING = "ttCustomer 2 error".

/* delete an order */
FIND ttOrder WHERE ttOrder.OrderNum = 94.
DELETE ttOrder.

TEMP-TABLE ttCustomer:TRACKING-CHANGES = FALSE.
TEMP-TABLE ttCustomer:TRACKING-CHANGES = FALSE.

/* write-json with before-image data to file */
ASSIGN
cTargetType = "file"
cFile = "dsOrderLogWithBefore.json"
lFormatted = TRUE
cEncoding = ?
lOmitInitialValues = FALSE
lOmitOuterObject = FALSE
lWriteBeforeImage = TRUE.
```

```
lRetOK = hdsOrderLog:WRITE-JSON(cTargetType, cFile, lFormatted,
                                cEncoding, lOmitInitialValues,
                                lOmitOuterObject, lWriteBeforeImage).
```

The following is an excerpt of the JSON produced by this procedure:

```
{"dsOrderLog": {
  "prods:hasChanges": true,
  "ttCustomer": [
   {
    "prods:id": "ttCustomer14592",
      "prods:rowState": "modified",
      "prods:hasErrors": true,
      "CustNum": 2,
      ...
      "EmailAddress": "www.progress.com"
    }
  ],
  "ttOrder": [
    {
      "Ordernum": 125,
      ...
    },
    ...
    {
      "Ordernum": 6070,
      ...
  ],
  "ttInvoice": [
    {
      "Invoicenum": 94,
      ...
    },
    {
      "Invoicenum": 124,
      ...
    }
  ],
  "prods:before": {
    "ttCustomer": [
      {
        "prods:id": "ttCustomer14592",
        "prods:rowState": "modified",
        "CustNum": 2,
        ...
        "EmailAddress": ""
      }
    ],
    "ttOrder": [
      {
        "prods:id": "ttOrder12545",
        "prods:rowState": "deleted",
        "CustNum": 2,
        "OrderNum": 94,
        ...
      }
    ]
  },
  "prods:errors": {
    "ttCustomer": [
      {
        "prods:id": "ttCustomer14592",
        "prods:error": "ttCustomer 2 error"
      }
    ]
```

```
  }
}}
```

The `prods:before` object contains the before-image data for the ProDataSet.

Minimizing the size of JSON data

When using JSON to transfer large amounts of data, it is always worthwhile to design your schema to minimize the size of the JSON data. Smaller JSON data reduces memory, network bandwidth, and disk usage. The resource savings can be significant for large JSON objects.

There are two features of ABL JSON support that can reduce the size of JSON data:

- For ProDataSets with nested data relations, you can choose to omit the duplicate entries of foreign key fields that appear in each child record nested within a parent record.
- For any ProDataSet or temp-table, you can choose to omit fields from the JSON whose current value is the same as the initial value of that field.

Omitting the outer object in JSON data

When writing JSON data using the `WRITE-JSON( )` method, an optional `LOGICAL` expression, *`omit-outer-object`*, indicates whether the outer-most object is included in the JSON output. `TRUE` directs the AVM to remove the outer-most object on output, and `FALSE` directs the AVM to include the outer-most object in the output.

When writing a JSON string or *`JsonObject`* target, the default value is `FALSE`. If you specify the Unknown value (`?`), `WRITE-JSON( )` also uses the default value of `FALSE`. When writing a temp-table to a *`JsonObject`* target, *`omit-outer-object`* **must** be `FALSE`. If you specify `TRUE`, `WRITE-JSON( )` generates an error message and returns `FALSE`.

When writing a *`JsonArray`* target, the default value is `TRUE`. If you specify the Unknown value (`?`), `WRITE-JSON( )` also uses the default value of `TRUE`. When writing a temp-table or temp-table buffer object to a *`JsonArray`* target, *`omit-outer-object`* **must** be `TRUE`. If you specify `FALSE`, `WRITE-JSON( )` generates an error message and returns `FALSE`.

The following examples demonstrate the difference between the value of `TRUE` and `FALSE` for *`omit-outer-object`* when writing a JSON string on a temp-table or temp-table buffer object handle.

```
{"tt": [
  {"f1": 11, "f2": 12, "f3": 13},
  {"f1": 21, "f2": 22, "f3": 23},
  {"f1": 31, "f2": 32, "f3": 33}
]}
```

The above values are generated when the *`omit-outer-object`* is `FALSE`.

```
[
  {"f1": 11, "f2": 12, "f3": 13},
  {"f1": 21, "f2": 22, "f3": 23},
  {"f1": 31, "f2": 32, "f3": 23}
]
```

The above values are generated when the *omit-outer-object* is TRUE.

The following examples demonstrates the difference between the value of TRUE and FALSE for *omit-outer-object* when writing a JSON string on a ProDataSet object handle.

```
{"pds" : {
  "tt1" : [
    {"f1": 11, "f2": 12, "f3": 13},
    {"f1": 21, "f2": 22, "f3": 23},
    {"f1": 31, "f2": 32, "f3": 33}
  ],
  "tt2": [
    {"f1": 11, "f2": 12, "f3": 13},
    {"f1": 21, "f2": 22, "f3": 23},
    {"f1": 31, "f2": 32, "f3": 33}
  ],
  "tt3": [
    {"f1": 11, "f2": 12, "f3": 13},
    {"f1": 21, "f2": 22, "f3": 23},
    {"f1": 31, "f2": 32, "f3": 33}
  ]
}}
```

The above values are generated when the *omit-outer-object* is FALSE.

```
{
  "tt1": [
    {"f1": 11, "f2": 12, "f3": 13},
    {"f1": 21, "f2": 22, "f3": 23},
    {"f1": 31, "f2": 32, "f3": 33}
  ],
  "tt2": [
    {"f1": 11, "f2": 12, "f3": 13},
    {"f1": 21, "f2": 22, "f3": 23},
    {"f1": 31, "f2": 32, "f3": 33}
  ],
  "tt3": [
    {"f1": 11, "f2": 12, "f3": 13},
    {"f1": 21, "f2": 22, "f3": 23},
    {"f1": 31, "f2": 32, "f3": 33}
  ]
}
```

For the same ProDataSet, the above values are generated when the *omit-outer-object* is TRUE.

Omitting foreign key fields in nested child records

When working with a nested data relation in a ProDataSet, the fields that define the relationship appear in both the parent record and the child records. Because the child records are contained within the parent record, the foreign key fields in the child records are redundant. The WRITE-JSON( ) method omits these foreign key fields if you:

- Specify the FOREIGN-KEY-HIDDEN option on a DEFINE DATA-RELATION statement
- Specify the foreign-key-hidden argument on the ADD-RELATION( ) method of a Data-relation object
- Set the FOREIGN-KEY-HIDDEN attribute of a Data-relation object to TRUE

In all three cases, you must also specify the NESTED option for the data-relation.

Before using this option, consider carefully whether the consumers of the JSON data will handle the missing key fields appropriately. In ABL, the `READ-JSON( )` method automatically populates foreign keys in nested child records with the value in the outer parent record when the foreign key is omitted from the JSON data. Unless you are sure that a non-ABL consumer of the JSON data will do the same, do not use this option in your nested data-relations.

For example, while a Web browser can read the JSON data and populate a JavaScript object, it will create rows in the nested table without the foreign key field.

Omitting fields with initial values

When a field is defined, it gets an initial value either by using the default initial value for the field's data type or from the `INITIAL` option in the definition statement or `ADD-NEW-FIELD( )` method. The following table shows the default initial values for ABL data types.

Table 8: Default initial values for ABL data types

Data type	Default initial value
BLOB	Unknown value (?)
CHARACTER	Empty string ("")
CLOB	Unknown value (?)
COM-HANDLE	Unknown value (?)
DATE	Unknown value (?)
DATETIME	Unknown value (?)
DATETIME-TZ	Unknown value (?)
DECIMAL	0
HANDLE	Unknown value (?)
INT64	0
INTEGER	0
LOGICAL	No
RAW	Zero-length sequence of bytes
RECID	Unknown value (?)
ROWID	Unknown value (?)

Omitting initial value fields from your JSON can be useful if:

- Fields with initial values are not important to the business or application logic of your JSON consumer
- The JSON consumer knows how to recreate missing fields and populate them with initial values

To omit these fields, specify TRUE for the optional *omit-initial-values* argument of the WRITE-JSON( ) method.

When working with large ProDataSets, omitting fields containing their initial values can yield smaller JSON values, more efficient network transfers, and performance gains with the READ-JSON( ) and WRITE-JSON( ) methods.

Although using the *omit-initial-values* option can give your application performance and resource use improvements, you must be sure that the consumers of the generated JSON output can correctly handle it. The ABL READ-JSON( ) method always populates created records with initial values from the temp-table or ProDataSet definition. Other applications might not do this.

Writing JSON from a temp-table buffer's current row

The SERIALIZE-ROW( ) method serializes the current row of the temp-table buffer handle on which you call it either as a JSON string or as a Progress.Json.ObjectModel.JsonObject object.

Note that you cannot call SERIALIZE-ROW( ) on a database buffer handle.

For string outputs only, SERIALIZE-ROW( ) provides the option to format the output in a hierarchical manner using extra white space and line breaks.

The syntax for SERIALIZE-ROW( ) is shown below. The syntax is similar to that for the WRITE-JSON( ) method, with the addition of the *target-format* parameter. The method returns TRUE or FALSE to indicate whether the operation was successful or not.

Syntax

```
SERIALIZE-ROW ( target-format , target-type ,
    { file | stream | stream-handle | memptr | longchar | JsonObject }
    [ , formatted [ , encoding [ , omit-initial-values
    [ , omit-outer-object ] ] ] ] )
```

target-format

A CHARACTER expression that specifies the target format used to serialize the current row. Valid values are "JSON" and "XML". However, this method description focuses on serializing to JSON.

For information on using this method to serialize the current row of a temp-table buffer to XML, see *Use XML with ABL Applications*.

target-type

A CHARACTER expression that specifies the target type for the JSON output. Valid values are "FILE", "STREAM", "STREAM-HANDLE", "MEMPTR", "LONGCHAR", and "JsonObject".

All values except "JsonObject" specify a target for a JSON string.

file

A CHARACTER expression that specifies the name of a file to which the AVM writes the JSON string. You can specify an absolute path or a path relative to the current working directory. If a file with the specified name already exists, the AVM verifies if the file is writeable and overwrites the file.

stream

A CHARACTER expression that specifies the name of a stream. If you specify the empty string (""), the AVM writes the JSON string to the default unnamed output stream. For WebSpeed, write the JSON string to the output stream defined by WebSpeed (WEBSTREAM).

stream-handle

A HANDLE variable that specifies a stream object handle.

memptr

A MEMPTR variable to contain the JSON string in memory. If you do not specify the encoding parameter, the AVM encodes the text written to the MEMPTR as "UTF-8". This method allocates the required amount of memory for the JSON string and sets the size of the variable. When you finish using the MEMPTR, you must free the associated memory by setting the MEMTER to 0 bytes with the SET-SIZE statement.

longchar

A LONGCHAR variable which contains the JSON string in memory.

The AVM saves the JSON string to the LONGCHAR variable in the code page which corresponds to the character encoding you specify in the *encoding* option. If you do not specify a character encoding for the JSON string, the AVM saves the LONGCHAR variable in "UTF-8".

If the LONGCHAR variable's code page is fixed (that is, set using the FIX-CODEPAGE statement) and the fixed code page is not equivalent to the character encoding you specify in the *encoding* option, SERIALIZE-ROW( ) generates an error and returns FALSE. The JSON string is not saved to the LONGCHAR.

JsonObject

A reference to a Progress.Json.ObjectModel.JsonObject object to which the AVM writes the buffer's current row. Any prior contents of the specified *JsonObject* are removed.

The *JsonObject* must be a valid instance of type Progress.Json.ObjectModel.JsonObject. If not, SERIALIZE-ROW( ) generates an error message and returns FALSE.

formatted

An optional LOGICAL expression where TRUE directs the AVM to format a JSON string in a hierarchical manner using white space, carriage returns, and line feeds. The default value is FALSE. If you specify the Unknown value (?), the method uses the default value FALSE. The *formatted* option is ignored for *JsonObject* targets.

encoding

An optional CHARACTER expression that specifies the name of the character encoding the AVM uses to write the JSON output. The default encoding is "UTF-8".

The encoding name must specify a Unicode transformation format. Valid values are "UTF-8", "UTF-16", "UTF-16BE", "UTF-16LE", "UTF-32", "UTF-32BE", and "UTF-32LE".

Note: If you specify the empty string ("") or the Unknown value (?), the AVM uses the default encoding "UTF-8".

omit-initial-values

An optional LOGICAL expression where TRUE directs the AVM to exclude temp-table fields containing their initial values from the JSON output, and FALSE directs the AVM to include all temp-table field data in the output. The default value is FALSE. If you specify the Unknown value (?), the method uses the default value of FALSE.

For more information about this option, see the Minimizing the size of JSON data on page 41.

omit-outer-object

An optional LOGICAL expression indicating whether the outer-most object in JSON output is included. TRUE directs the AVM to remove the outer-most object on output. FALSE directs the AVM to include the outer-most object in the output.

For more information about this option, see the Omitting the outer object in JSON data on page 41.

The following example demonstrates the use of SERIALIZE-ROW( ):

```
DEFINE TEMP-TABLE ttCust NO-UNDO
  FIELD custNum AS INTEGER
  FIELD Name AS CHARACTER
  FIELD Address AS CHARACTER.

DEFINE VARIABLE bcust AS HANDLE.

FIND FIRST sports2000.Customer.
CREATE ttCust.
BUFFER-COPY sports2000.Customer TO ttCust.
bcust = TEMP-TABLE ttCust:DEFAULT-BUFFER-HANDLE.
bcust:SERIALIZE-ROW("JSON", "file", "bcust.json", TRUE).
bcust:SERIALIZE-ROW("JSON", "file", "bcust2.json", TRUE, ?, ?, TRUE).
```

The SERIALIZE-ROW( ) method produces the following JSON string outputs. The JSON output below is bcust.json with *omit-outer-object* set to FALSE.

```
{"ttCust":
  {
    "custNum": 1,
    "Name": "Lift Tours",
    "Address": "276 North Drive"
  }
}
```

The JSON output below is bcust2.json with *omit-outer-object* set to TRUE.

```
{
  "custNum": 1,
  "Name": "Lift Tours",
  "Address": "276 North Drive"
}
```

Reading JSON into a temp-table, temp-table buffer, or ProDataSet

The READ-JSON( ) method loads data from a JSON string, Progress.Json.ObjectModel.JsonArray object, or Progress.Json.ObjectModel.JsonObject object into a corresponding static or dynamic temp-table, temp-table buffer, or ProDataSet object handle on which you call the method. If the ABL data object has a defined schema, the AVM assumes that the JSON values match up with the ABL fields' data types as shown in Table 4. READ-JSON( ) generates an error message and returns FALSE if the JSON value cannot be converted to the expected ABL data type.

Note that you cannot call READ-JSON( ) on a database buffer handle.

For a dynamic ProDataSet or temp-table that is in the CLEAR state, the AVM infers the object's schema from the data in the JSON value. If a dynamic temp-table is not in the PREPARED or CLEAR state, the method generates an error and returns FALSE. For more information about inferring schema from a JSON string, see the Inferring ABL schemas from JSON data on page 50.

READ-JSON( ) accepts a JsonArray or JsonObject as a source as long as the source object conforms to an accepted pattern for the temp-table or ProDataSet into which you are loading the JSON data. Note, however, that you cannot read a JsonArray into a ProDataSet object. If you attempt to load data from a JsonArray into a ProDataSet object, READ-JSON( ) generates an error message and returns FALSE.

If a JSON string contains ProDataSet before-image data, the READ-JSON( ) method populates the after-table and before-table data for the ProDataSet.

Here is the syntax for READ-JSON( ). The method returns TRUE or FALSE to indicate if the operation was successful.

Syntax

```
READ-JSON ( source-type ,
  { file | memptr | handle | longchar | JsonArray | JsonObject }
  [ , read-mode ] )
```

source-type

A CHARACTER expression that specifies the source JSON string type. Valid values are "FILE", "MEMPTR", "HANDLE", "LONGCHAR", "JsonArray", and "JsonObject".

All values except "JsonArray" and "JsonObject" specify a source for a JSON string.

file

A CHARACTER expression that specifies the name of a file. You can specify an absolute pathname or one relative to the current working directory. The AVM verifies that the file exists and is accessible.

memptr

A MEMPTR variable that contains the JSON string in memory. The size of the MEMPTR variable must match the size of the JSON string.

handle

A HANDLE variable that specifies the WEB-CONTEXT system handle.

This method reads a JSON string from the WebSpeed Transaction Server. The method verifies that the JSON string was posted to the WebSpeed Transaction Server by checking that the handle's IS-JSON attribute is YES. The method also verifies that ABL is running in a WebSpeed environment.

longchar

A LONGCHAR variable that contains the JSON string in memory.

JsonArray

A reference to a Progress.Json.ObjectModel.JsonArray object, which can contain additional JsonArrays and JsonObjects. This *JsonArray* must fit one of the valid patterns for a temp-table object. If any part of the *JsonArray* does not fit one of the accepted patterns, READ-JSON( ) generates an error message and returns FALSE. If the *JsonArray* fits an accepted pattern, but contains fields whose names do not match the existing temp-table schema, the mismatched fields are ignored.

JsonObject

A reference to a Progress.Json.ObjectModel.JsonObject object, which can contain additional JsonArrays and JsonObjects. This *JsonObject* must fit one of the valid patterns for a ProDataSet or temp-table object. If any part of the *JsonObject* does not fit one of the accepted patterns, READ-JSON( ) generates an error message and returns FALSE. If the *JsonObject* fits an accepted pattern, but contains tables whose names do not match the existing ProDataSet schema, or contains fields whose names do not match the existing temp-table schema, the mismatched tables or fields are ignored.

read-mode

An optional CHARACTER expression that specifies the mode in which this method reads data from the JSON source into a temp-table or a ProDataSet member buffer. The expression must evaluate to "APPEND", "EMPTY", "MERGE", or "REPLACE". The default value is "MERGE".

The following table lists the READ-JSON( ) method modes for reading data.

Table 9: READ-JSON( ) method read modes

When the mode is . . .	The READ-JSON( ) method . . .
"APPEND"	Reads data from the JSON source into the ProDataSet or temp-table object by adding new records to the existing records, without performing any record comparisons. If a record from the JSON source exists in the object (that is, it results in a duplicate unique key conflict), the method generates an error message and returns FALSE.
"EMPTY"	Empties the contents of the ProDataSet or temp-table object before reading in data from the JSON source.

When the mode is . . .	The READ-JSON( ) method . . .
"MERGE"	Reads data from the JSON source into the ProDataSet or temp-table object by merging new records with existing records in the table. If a record from the JSON source exists in the object (that is, it results in a duplicate unique key conflict), the method does not replace the existing record. If the record from the JSON source does not exist in the object, the method creates a new record.
"REPLACE"	Reads data from the JSON source into the ProDataSet or temp-table object by merging new records with existing records in the table. If the record from the JSON source exists in the object (that is, it results in a duplicate unique key conflict), the method replaces the existing record with the new record. If the record from the JSON source does not exist in the object, the method creates a new record.

For a static ProDataSet or temp-table, the serialize name or object name must match the corresponding name found in the JSON source. If the names do not match, the AVM generates an error message and the method returns FALSE. The AVM ignores any columns in the JSON source that do not map to temp-table columns. If you use the SERIALIZE-NAME option in the DEFINE DATASET or DEFINE TEMP-TABLE statement, the AVM uses that name for matching, rather than the ABL object name.

Note: While reading JSON data into an ABL data object, the AVM does not respond to ProDataSet events. The AVM also does not track changes to the data in the ProDataSet or temp-table object, meaning that before-image tables are not updated.

Reading JSON into a data object with defined schema

The READ-JSON( ) method functions in one of two ways, depending on whether or not its target ABL data object has a defined schema. The most predictable results come when you read JSON data into a data object with a defined schema. In this case, the AVM uses the names to match the ABL ProDataSet, temp-tables, and fields to the JSON objects and name/value pairs. It ignores JSON data for any unmatched data object or field. The AVM raises an error if a matching JSON value does not match the corresponding ABL field's data type.

The following procedure reads the JSON data that was output by write-json-pds2.p into a ProDataSet that uses a subset of the fields in the original temp-table. It then writes the new ProDataSet out as JSON to another file:

```
/* read-json-pds2.p */
{pi-json-parameterVarDefs.i} /* parameter variable definitions */

DEFINE TEMP-TABLE ttCustomer NO-UNDO
  FIELD CustNum     LIKE Customer.CustNum
  FIELD Name        LIKE Customer.Name
  FIELD State       LIKE Customer.State.

DEFINE TEMP-TABLE ttOrder    NO-UNDO
  FIELD OrderNum    LIKE Order.Ordernum
  FIELD CustNum     LIKE Order.CustNum
  FIELD ShipDate    LIKE Order.ShipDate.

DEFINE TEMP-TABLE ttInvoice  NO-UNDO
  FIELD Invoicenum  LIKE Invoice.Invoicenum
  FIELD OrderNum    LIKE Invoice.OrderNum
  FIELD InvoiceDate LIKE Invoice.InvoiceDate.
```

```
DEFINE DATASET dsOrderLog FOR ttCustomer, ttOrder, ttInvoice
  DATA-RELATION CustOrd FOR ttCustomer,
    ttOrder   RELATION-FIELDS(CustNum,CustNum)   NESTED
  DATA-RELATION OrdInv  FOR ttOrder,
    ttInvoice RELATION-FIELDS(OrderNum,OrderNum) NESTED.

DEFINE VARIABLE hdsOrderLog AS HANDLE  NO-UNDO.
DEFINE VARIABLE lRetOK      AS LOGICAL NO-UNDO.

hdsOrderLog = DATASET dsOrderLog:HANDLE.

ASSIGN
  cSourceType = "file"
  cFile       = "dsOrderLog2.json"
  cReadMode   = "EMPTY".

lRetOK = hdsOrderLog:READ-JSON(cSourceType, cFile, cReadMode).

ASSIGN
  cTargetType = "file"
  cFile       = "dsOrderLog3.json"
  lFormatted  = TRUE
  cEncoding   = ?.

lRetOK = hdsOrderLog:WRITE-JSON(cTargetType, cFile, lFormatted).
```

When you compare the JSON files, you see that only the data that fit into the new ProDataSet made it from `dsOrderLog2.json` to `dsOrderLog3.json`.

Inferring ABL schemas from JSON data

The `READ-JSON( )` method has to infer a schema when its target ABL data object does not have a schema. Unlike XML, JSON does not have a standard schema language. Because the format of each JSON value indicates its data type, the AVM can infer a schema from the source, whether it is a JSON string, a `Progress.Json.ObjectModel.JsonArray`, or a `Progress.Json.ObjectModel.JsonObject`.

Caution: In general, reading JSON into an ABL data object with an inferred schema is less predictable than the alternative. You should consider carefully how this technique might affect the data, especially if you plan to return the data after processing it.

When the AVM has to infer schema for the data object, the AVM makes two passes through the JSON data: one to build the schema and one to fill in the data. On the first pass, the AVM reads all the records before finalizing the schema, which has the following effects:

- When the AVM parses a JSON `null`, it provisionally assigns a `CHARACTER` data type to the column. If a subsequent record includes a non-null value for that column, the AVM assigns that data type to the column. In either case, the AVM equates the JSON `null` value to the Unknown value (`?`).
- If different rows contain different fields, the final schema includes all the fields.

The AVM infers ABL schema from JSON data using the following guidelines:

- Any JSON object containing an array of objects is a temp-table. The temp-table's name is the array's name.
- The entries in an array of objects are the rows of a single temp-table.
- Each name/value pair in a row's object is a column in the temp-table. The column's name is the JSON value's name.

- If a value in a row object is a JSON array, it is an array column. The AVM infers the data type of the array column from the first value in the inner array.
- Any JSON object that is not an array of objects, but that contains at least one object from which the AVM infers a temp-table, is a ProDataSet. The ProDataSet's name is the JSON object's name.
- If the AVM encounters an array of objects within another array of objects, the AVM infers it to be a nested temp-table inside the ProDataSet.
- If the AVM infers a temp-table nested within another inferred temp-table, the AVM attempts to create a relationship between the two tables. If there is only one pair of fields with matching names in the parent and child tables, the AVM creates a data-relation between the parent table and nested child table using the matching fields for the pairs-list. If there are no matching fields, the AVM creates a parent-id-relation between the parent and nested child and adds a `RECID` field to the child table to maintain the relationship. If there is more than one pair of matching fields between the tables, the AVM generates an error message and the `READ-JSON( )` method returns `FALSE`.

Note: If you call `READ-JSON( )` on a temp-table object and the AVM infers a nested temp-table, the method generates an error message and returns `FALSE`. If you call `READ-JSON( )` on a ProDataSet object and the JSON data contains only a temp-table, the method generates an error message and returns `FALSE`.

The following table shows how the AVM maps JSON data types to ABL data types when inferring the schema of a temp-table. By comparing the following table with Table 7, you can see that the differences in the data type mapping make it unlikely that an inferred temp-table matches the original object from which the data was read.

Table 10: Inferring ABL data types from JSON values

JSON value	ABL data type
`string`[2]	`CHARACTER`
`number`[3]	`DECIMAL`
`boolean` (`true` or `false`)	`LOGICAL`
`null`	`CHARACTER`

Note: When inferring ABL data types, the AVM does not try to determine if a JSON value represents one of the non-standard data types listed in Table 6. The AVM infers a JSON value representing a non-standard data type as a JSON string and assigns it a `CHARACTER` data type in ABL. For example, a JSON string value in ISO 8601 format is interpreted as a `CHARACTER` field, not a `DATETIME` field.

[2] If a JSON string's value is too long for the ABL data type, the AVM generates an error message and the READ-JSON( ) method returns FALSE.

[3] If a JSON number's value is out of the ABL data type's range, the AVM generates an error message and the READ-JSON( ) method returns FALSE.

As an example for the inferring process, take the following JSON object:

```
{"ttCust":
  [
    {"Name": ["L", "Frank", "Baum"], "CustNum": 1, "GoldStatus": null},
    {"Name": ["Alfred", "E", "Newman"], "CustNum": 2, "GoldStatus": false},
    {"Name": ["Bullwinkle", "J", "Moose"], "CustNum": 3, "GoldStatus": true}
  ]
}
```

The JSON object contains an array name, `"ttCust"`. The AVM sees that it is an array of objects and creates a temp-table named `ttCust` to hold the data. The first name/value pair in the row's object is an array of JSON `string` values named `"Name"`. The AVM creates a `CHARACTER` field, `Name`, of `EXTENT 3` as the temp-table's first column. The next name/value pair is a JSON number named `"CustNum"`. The AVM creates a `DECIMAL` field, `CustNum`, as the second column. The final name/value pair contains a JSON `null` named `"GoldStatus"`. Because the value is a `null`, the AVM temporarily chooses `CHARACTER` as the final column's data type. The AVM then reads the next record and determines that it contains a JSON `boolean` for the final pair and creates a `LOGICAL` field, `GoldStatus`, as the final column.

The following procedure reads the output from `write-json-pds2.p` into a dynamic ProDataSet, inferring the ProDataSet's schema from the JSON data. It then outputs the schema and data to another file, so you can examine the results:

```
/*  read-json-infer-pds2.p */
DEFINE VARIABLE hDataset AS HANDLE  NO-UNDO.
DEFINE VARIABLE lRetOK   AS LOGICAL NO-UNDO.
DEFINE VARIABLE hQuery   AS HANDLE  NO-UNDO.
DEFINE VARIABLE hBuffer  AS HANDLE  NO-UNDO.
DEFINE VARIABLE hField   AS HANDLE  NO-UNDO.
DEFINE VARIABLE idx1     AS INTEGER NO-UNDO.
DEFINE VARIABLE idx2     AS INTEGER NO-UNDO.
DEFINE VARIABLE idx3     AS INTEGER NO-UNDO.

CREATE DATASET hDataset.

OUTPUT TO InferPDS2.out APPEND.
lRetOK = hDataset:READ-JSON("file", "dsOrderLog2.json", "empty").
RUN displayResults.
DELETE OBJECT hDataset NO-ERROR.
OUTPUT CLOSE.

PROCEDURE displayResults:
  MESSAGE "READ-JSON return value: " lRetOK SKIP.
  MESSAGE SKIP "** hDataset schema info **" SKIP.
  MESSAGE "ProDataSet name: " hDataset:NAME
          "Num-buffers " hDataset:NUM-BUFFERS.
    DO idx1 = 1 TO hDataset:NUM-BUFFERS:
      hBuffer = hDataset:GET-BUFFER-HANDLE(idx1).
      MESSAGE SKIP "Buffer " idx1 "Buffer name: " hBuffer:NAME.
      MESSAGE "Buffer Field info".
      DO idx2 = 1 TO hBuffer:NUM-FIELDS:
        hField = hBuffer:BUFFER-FIELD(idx2).
        MESSAGE "Field name: " hField:NAME "Data type: " hField:DATA-TYPE
                " Extent: " hField:EXTENT.
      END. /* idx2 loop */
    END. /* idx1 loop */
  MESSAGE SKIP "** hDataset data **".
    DO idx1 = 1 TO hDataset:NUM-BUFFERS:
      hBuffer = hDataset:GET-BUFFER-HANDLE(idx1).
      MESSAGE "*** Buffer " hBuffer:NAME " Data: ***".
      CREATE QUERY hQuery.
      hQuery:SET-BUFFERS(hBuffer).
```

```
      hQuery:QUERY-PREPARE("for each " + hBuffer:NAME).
      hQuery:QUERY-OPEN.
      hQuery:GET-NEXT() NO-ERROR.
      DO WHILE NOT hQuery:QUERY-OFF-END:
        MESSAGE SKIP.
          DO idx2 = 1 TO hBuffer:NUM-FIELDS:
            hField = hBuffer:BUFFER-FIELD(idx2).
            IF hField:EXTENT = 0 THEN
              MESSAGE hField:NAME ": " hField:BUFFER-VALUE.
            ELSE
              MESSAGE hField:NAME.
              DO idx3 = 1 TO hField:EXTENT:
                MESSAGE hField:NAME ": " hField:BUFFER-VALUE(idx3).
              END. /* idx3 loop */
            END. /* idx2 loop */
            hQuery:GET-NEXT() NO-ERROR.
      END. /* hQuery loop */
        MESSAGE SKIP.
        DELETE OBJECT hQuery NO-ERROR.
    END. /* idx1 loop */
END PROCEDURE.
```

The output from `write-json-pds2.p` is designed with the `READ-JSON( )` method's inferring feature in mind. The ProDataSet's tables have only one possible foreign key between each pair of nested tables. If you ran `write-json-pds2.p` substituting the following ProDataSet definition for the include file and then ran `read-json-infer-pds2.p` on the output, the procedure would generate several errors because there are several fields in each table that match fields in the outer tables.

```
DEFINE TEMP-TABLE ttCustomer NO-UNDO LIKE Customer.
DEFINE TEMP-TABLE ttOrder    NO-UNDO LIKE Order.
DEFINE TEMP-TABLE ttInvoice  NO-UNDO LIKE Invoice.

DEFINE DATASET dsOrderLog FOR ttCustomer, ttOrder, ttInvoice
  DATA-RELATION CustOrd FOR ttCustomer,
    ttOrder   RELATION-FIELDS(CustNum,CustNum) NESTED
  DATA-RELATION OrdInv  FOR ttOrder,
    ttInvoice RELATION-FIELDS(OrderNum,OrderNum) NESTED.

DEFINE DATA-SOURCE dsCustomer FOR Customer.
DEFINE DATA-SOURCE dsOrder    FOR Order.
DEFINE DATA-SOURCE dsInvoice  FOR Invoice.
BUFFER ttCustomer:HANDLE:ATTACH-DATA-SOURCE(DATA-SOURCE dsCustomer:HANDLE).
BUFFER ttOrder:HANDLE:ATTACH-DATA-SOURCE(DATA-SOURCE dsOrder:HANDLE).
BUFFER ttInvoice:HANDLE:ATTACH-DATA-SOURCE(DATA-SOURCE dsInvoice:HANDLE).
```

Using JSON with WebSpeed

Because of JSON's popularity in AJAX applications, WebSpeed applications are an obvious place to use these features. To support using JSON with WebSpeed, the `WEB-CONTEXT` system handle has the `IS-JSON` attribute. The `IS-JSON` attribute indicates whether or not a JSON string was posted to the WebSpeed Transaction Server. The AVM determines this by looking for the following content-type HTTP headers:

- "`application/json`"
- "`text/json`"

The exact procedure for communicating with a rich Internet application vary depending on the application. As a general example, you might have something like the following procedure fragment to handle JSON requests in a WebSpeed application:

```
{src/web2/wrap-cgi.i}        /* standard WebSpeed include */
{pi-json-parameterVarDefs.i} /* parameter variable definitions */
{pi-write-json-tt.i}         /* ttCust definition */

PROCEDURE process-web-request:
  DEFINE VARIABLE httCust AS HANDLE   NO-UNDO.
  DEFINE VARIABLE lRetOK  AS LOGICAL  NO-UNDO.
  DEFINE VARIABLE lcData  AS LONGCHAR NO-UNDO.

  httCust = TEMP-TABLE ttCust:HANDLE.

  /* 2 */ IF NOT WEB-CONTEXT:IS-JSON THEN
  DO ON ERROR UNDO, LEAVE:
    /* 3 */ RUN populateTT.
    httCust:WRITE-JSON("LONGCHAR", lcData).

/* Build and return your initial HTML response with the JSON of the initial temp-table
 values */
    /* 4 */ output-content-type ("text/html":U).
    RUN writeHTML(lcData).
    . . .
  /* 6 */ ELSE DO:
    lRetOK = httCust:READ-JSON("HANDLE", WEB-CONTEXT, "MERGE").
    /* 7 */ IF NOT lRetOK THEN
    DO:
      output-content-type ("text/html":U).
      {&OUT}
        "<html>":U SKIP
        "<head>":U SKIP
        "<title> {&FILE-NAME} </title>":U SKIP
        "</head>":U SKIP
        "<body>":U SKIP
        "<p>Failed to load JSON into data object:</p>":U SKIP
        "<p>" + ERROR-STATUS:GET-MESSAGE(1) + "</p>":U SKIP
        "</BODY>":U SKIP
        "</HTML>":U SKIP.
    END.
    /* 8 */ ELSE DO:
      output-content-type ("application/json":U).
      httCust:WRITE-JSON("STREAM", "WebStream").
    END.
  END.
END PROCEDURE.
```

Using the `process-web-request` procedure, a complete Web request might run as follows:

1. You log in, creating a request for the initial page that starts the `process-web-request` procedure on the WebSpeed Transaction Server.
2. The procedure first checks whether or not the request is a JSON string.
3. The initial request is not a JSON string, so the procedure populates the temp-table with the initial records and serializes the temp-table using `WRITE-JSON( )`.
4. The procedure then builds a HTML page around the JSON and sends it back to the Web browser.
5. You update some of the records and posts the changes back to the server.
6. Since the new Web request is a JSON string, the procedure tries to merge the updates into the temp-table using `READ-JSON( )`.

7. If merging the updates into the temp-table fails, the procedure builds and returns a HTML page reporting the error.
8. If merging the updates into the temp-table succeeds, the procedure serializes the updated temp-table using `WRITE-JSON( )` and posts the updates back to the Web browser.

Index

A

ABL support
    data type mapping *31*
    non-standard JSON data types *31*
AJAX
    JSON popularity *15*
Array
    JSON data type *16*

B

Boolean
    JSON data type *16*

C

Codepages
    encoding for serialization *33*, *44*
Complex data types
    JSON *16*
Complex values
    JSON structure *18*

D

Data types
    array *16*
    Boolean *16*
    complex *16*
    inferred *50*
    mapping *31*
    non-standard JSON *31*
    null *16*
    number *16*
    object *16*
    primitive *16*
    string *16*

I

IS-JSON attribute *53*

J

JavaScript Object Notation (JSON)
    JSON *15*, *23*, *26*
JSON
    ABL support for *29*
    as XML alternative *15*
JSON *(continued)*
    complex data types *16*
    complex values *18*
    data type mapping *31*
    definition *15*, *23*, *26*
    inferring schema *50*
    minimizing output *42*
    non-standard data types *31*
    primitive data types *16*
    serialization related attributes *30*
    simple values *17*
    unsupported ABL features *29*
    use in OpenEdge *16*
    WebSpeed *53*

N

Null
    JSON data type *16*
Number
    JSON data type *16*

O

Object
    JSON data type *16*

P

Parsers
    included in OpenEdge *16*
Primitive data types
    JSON *16*
ProDataSets
    before-image data *33*, *39*
    INITIAL option *43*
    NESTED option *36*
    omitting foreign keys *42*
    reading JSON into *47*, *49*
    serialization example *36*
    serialization related attributes *30*
    serializing to JSON *33*

R

READ-JSON( ) method
    data objects with schema *49*
    inferring schema *50*
    syntax *47*

S

Schema
    inferred data types *50*
    inferring *50*
    READ-JSON( ) method *49*
Serialization
    attribute interaction *30*
    codepages *33*, *44*
    minimizing output *42*
    related attributes *30*
Simple values
    JSON structure *17*
String
    JSON data type *16*

T

Temp-tables
    INITIAL option *43*
Temp-tables *(continued)*
    reading JSON into *47*
    serialization example *38*
    serialization related attributes *30*
    serializing to JSON *33*

W

WebSpeed
    using JSON *53*
WRITE-JSON( ) method
    syntax *33*, *44*

X

XML
    serialization attribute interaction *30*