Using JSON

Version 2023.1
2024-07-11

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Using JSON
InterSystems IRIS Data Platform Version 2023.1 2024-07-11

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

1

Using JSON in ObjectScript

InterSystems ObjectScript syntax includes integrated support for JSON (https://json.org/). A set of fast, simple, powerful features allow you to work with JSON data structures as easily as you do with objects or tables:

- With ObjectScript syntax for JSON, you can use standard ObjectScript assignment statements rather than method calls to create and alter dynamic entities at run time. The values of object properties and array elements can be specified either as JSON string literals or as ObjectScript *dynamic expressions*.
- Two classes, %Library.DynamicObject and %Library.DynamicArray, provide a simple, efficient way to encapsulate and work with standard JSON data structures. Instances of these classes are called *dynamic entities*.
- Dynamic entities contain methods for JSON serialization (conversion between dynamic entities and canonical JSON format), iteration, data typing, create/read/update/delete operations, and other useful functions.

See the Table of Contents for a detailed listing of the subjects covered in this document.

1.1 JSON Features in Action

Here are some examples of the JSON features available in ObjectScript:

Create and manipulate dynamic entities at runtime

You can create dynamic entities and define an arbitrary schema for them at run time:

```
set dynObject1 = ##class(%DynamicObject).%New()
set dynObject1.SomeNumber = 42
set dynObject1.SomeString = "a string"
set dynObject1.SomeArray = ##class(%DynamicArray).%New()
set dynObject1.SomeArray."0" = "an array element"
set dynObject1.SomeArray."1" = 123
```

Create dynamic entities with literal JSON constructors

You can also create a dynamic entity by assigning a literal JSON string. Literal JSON constructors **{}** and **[]** are can be used in place of the **%New()** constructor. For example you can create a dynamic array with `set x=[]` rather than `set x=##class(%DynamicArray).%New()`. Unlike **%New()**, a literal JSON constructor can also take a JSON literal string that specifies properties or elements. This means you can create an object identical to *dynObject1* in the previous example with these simple assignment statements:

```
set dynObject2 = {"SomeNumber":42,"SomeString":"a string"}
set dynObject2.SomeArray = ["an array element",123]
```

This example uses a statement for each constructor, but the array constructor could just as easily be nested inside the object constructor.

To demonstrate that *dynObject1* and *dynObject2* are identical, we can display them as serialized JSON strings returned by the **%ToJSON()** method:

```
   write "object 1: "_dynObject1.%ToJSON(),!,"object 2: "_dynObject2.%ToJSON()

object 1: {"SomeNumber":42,"SomeString":"a string","SomeArray":["an array element",123]}
object 2: {"SomeNumber":42,"SomeString":"a string","SomeArray":["an array element",123]}
```

Define values with dynamic expressions

The text enclosed in literal constructors **{}** and **[]** must use valid JSON syntax, with one exception. For the value of an element or property, you can use an expression enclosed in parentheses rather than a JSON literal. This ObjectScript *dynamic expression* (equivalent to the right side of a `set` statement) will evaluated at runtime and converted to a valid JSON value. The dynamic expression in this example includes a call to the **$ZDATE** function:

```
   set dynObj = { "Date":($ZD($H,3)) }
```

The evaluated dynamic expression value is displayed when we retrieve *dynObject.Date*:

```
   write "Value of dynamic expression is: "_dynObject.Date

Value of dynamic expression is: 2016-07-27
```

(See "Dynamic Expressions and Dot Syntax" for a detailed discussion of these topics).

Convert between dynamic entities and canonical JSON strings

Dynamic entities have serialization methods that allow them to be converted to and from JSON strings. In the following example, a literal constructor is used to create a dynamic object, and the object's **%ToJSON()** method is called to serialize it to *myJSONstring*:

```
   set myJSONstring = {"aNumber":(21*2),"aDate":($ZD($H,3)),"anArray":["string",123]}.%ToJSON()
```

This serialized JSON object can be stored and retrieved like any other string. Class methods **%FromJSON()** and **%FromJSONFile()** can take a valid JSON string from any source and convert it to a dynamic object. The following code deserializes *myJSONstring* to dynamic object *myObject* and uses **%ToJSON()** to display it:

```
   set myObject = ##class(%DynamicAbstractObject).%FromJSON(myJSONstring)
   write myObject.%ToJSON()

{"aNumber":42,"aDate":"2016-08-29","anArray":["string",123]}
```

(See "Converting Dynamic Entities to and from JSON" for more information on serialization).

Chain dynamic entity methods

Some dynamic entity methods can be chained. This example creates a dynamic array with two elements, and then chains the **%Push()** method to add three more elements to the end of the array. A final chained call to **%ToJSON()** displays the serialized string:

```
   set dynArray = ["a","b"]
   write dynArray.%Push(12).%Push({"a":1,"b":2}).%Push("final").%ToJSON()

["a","b",12,{"a":1,"b":2},"final"]
```

(See "Method Chaining" for more information on chainable methods).

Iteration and datatype discovery

Dynamic entity methods are also provided for purposes such as iteration and datatype discovery. This example creates two JSON strings, deserializes one of them to *dynEntity* (either one will work), and then gets an iterator for *dynEntity* :

```
set arrayStr = [12,"some string",[1,2]].%ToJSON()
set objectStr = {"a":12,"b":"some string","c":{"x":1,"y":2}}.%ToJSON()
set dynEntity = {}.%FromJSON(objectStr)
set itr = dynEntity.%GetIterator()
```

For each iteration of the `while` loop, **%GetNext()** will return the property name or array index in *key* and the member value in *val*. The return value of **%GetTypeOf()** is a string indicating the datatype of the value:

```
while itr.%GetNext(.key,.val) {write !,key_": "_"/"_val_"/, type: "_dynEntity.%GetTypeOf(key)}

a: /12/, type: number
b: /some string/, type: string
c: /1@%Library.DynamicObject/, type: object
```

(See "Iteration and Sparse Arrays" and "Working with Datatypes" for more information on these and related methods).

1.2 Overview of Dynamic Entity Methods

Dynamic entity methods can be grouped into the following categories:

Create, read, update, delete

%Set() can either change the value of an existing dynamic entity member (property or element), or create a new member and assign a value to it. **%Remove()** removes an existing member. **%Get()** retrieves the value of a member. See "Creating and Modifying Dynamic Entities" for details.

Iteration and Sparse Arrays

%GetIterator() returns an iterator containing pointers to each member of a dynamic entity. **%GetNext()** returns the key and value of a member identified by the iterator, and advances the cursor to the next member. **%Size()** returns the number of members (including unassigned elements in a sparse array). **%IsDefined()** tests whether a member has an assigned value. See "Iteration and Sparse Arrays" for details.

Stack functions

%Push() adds a new element to the end of a dynamic array. **%Pop()** removes the final element of the array and returns its value. These methods are not available for dynamic objects because object properties are not stored in a predictable sequence. See "Using %Push and %Pop with Dynamic Arrays" for details.

JSON serialization and deserialization

%FromJSON() converts a JSON string, and **%FromJSONFile()** converts a JSON string stored within a file, to a dynamic entity. **%ToJSON()** serializes a dynamic entity to a canonical JSON string. See "Converting Dynamic Entities to and from JSON" for details.

Datatype information

%GetTypeOf() returns a string indicating the datatype of a specified member value. **%Set()** and **%Push()** provide an optional third argument to explicitly specify the datatype of a value. See "Working with Datatypes" for details.

See the "Quick Reference for Dynamic Entity Methods" for a description of each method and links to further information.

2

Creating and Modifying Dynamic Entities

This chapter provides basic information on how dynamic entities work. The following topics are discussed:

2.1 Using JSON Literal Constructors

Dynamic entities are instances of %DynamicObject or %DynamicArray, which are designed to integrate JSON data manipulation seamlessly into ObjectScript applications. Although you can create instances of these classes with the standard **%New()** method, dynamic entities support a much more flexible and intuitive set of constructors. *JSON literal constructors* allow you to create dynamic entities by directly assigning a JSON string to a variable. For example, the following code creates empty instances of %DynamicObject and %DynamicArray:

```
   set dynamicObject = {}
   set dynamicArray = []
   write dynamicObject,!,dynamicArray
```

```
3@%Library.DynamicObject
1@%Library.DynamicArray
```

Unlike the **%New()** constructor, literal constructors **{}** and **[]** can accept a string in JSON format as an argument. For example, the following code creates a dynamic object with a property named *prop1*:

```
   set dynamicObject = {"prop1":"a string value"}
   write dynamicObject.prop1
```

```
a string value
```

In fact, JSON literal constructors **{}** and **[]** can be used to specify any valid JSON array or object structure. In simple terms, any valid JSON literal string is also a valid ObjectScript expression that evaluates to a dynamic entity.

Note: **JSON property names must always be quoted**

The JSON language specification (see https://json.org/) is a subset of JavaScript Object Notation, and enforces stricter rules in some areas. One important difference is that the JSON specification requires all property names to be enclosed in double-quotes. JavaScript syntax, on the other hand, permits unquoted names in many cases.

A dynamic entity stores an exact representation of each object property or array element in the JSON string. Any dynamic entity can use the **%ToJSON()** method to return the stored data as a JSON string. There is no loss or corruption of data when converting to or from a literal string. The following example creates a dynamic array and then calls **%ToJSON()** to construct and return a new JSON string representing the stored data:

```
   set dynamicArray = [[1,2,3],{"A":33,"a":"lower case"},1.23456789012345678901234,true,false,null,0,1,""]
   write dynamicArray.%ToJSON()
[[1,2,3],{"A":33,"a":"lower case"},1.23456789012345678901234,true,false,null,0,1,""]
```

This dynamic array has stored and returned several significant values:

- The first two elements are a nested array and a nested object. In JSON syntax, array and object structures can be nested to any depth.
- Property names are case-sensitive. The nested object has two distinct properties named `"A"` and `"a"`.
- The third value is a very high-precision decimal number. This value would have been rounded down if it were stored as a standard floating point number, but the dynamic array has retained an exact representation of the original value.
- The final six elements contain JSON datatype values `true`, `false`, and `null`, and corresponding ObjectScript values `0`, `1`, and `""`. Once again, dynamic entities preserve an exact representation of each value.

2.2 Using Dynamic Expressions and Dot Syntax

There are significant differences between the way values are stored in JSON and the way they are expressed in ObjectScript. JSON data storage would not be very useful if you had to convert an ObjectScript value to or from JSON syntax every time you wanted to use it, so dynamic entities are designed to make this conversion process transparent. You can always store and retrieve an ObjectScript value without worrying about its representation in JSON syntax.

Literal JSON constructors are no exception to this rule. So far, all of our examples have been entirely in JSON syntax, but literal constructors can also accept values defined in *dynamic expressions*, which are simply ObjectScript expressions enclosed in parentheses.

For example, the following dynamic array constructor stores two Unicode characters. At runtime, the literal constructor evaluates each element and stores the evaluated value. The first element is defined in JSON syntax and the second element is an ObjectScript function call, but the resulting stored values are identical:

```
   write ["\u00E9",($CHAR(233))].%ToJSON()
["é","é"]
```

You can think of an ObjectScript expression as the code on the right side of a `set` statement. Any ObjectScript expression that evaluates to a value rather than an object reference can be serialized to a JSON literal string. The following example

stores a $LIST value (which is a delimited string, not an object) in object property *obj.list*. It then creates *array* and extracts each list item in *obj.list* to a separate element:

```
   set obj = {"list":($LISTFROMSTRING("Deborah Noah Martha Bowie"," "))}
   set array = [($LIST(obj.list,1)),($LIST(obj.list,2)),($LIST(obj.list,3)),($LIST(obj.list,4))]
   write obj.%ToJSON(),!,array.%ToJSON()

{"list":"\t\u0001Deborah\u0006\u0001Noah\b\u0001Martha\u0007\u0001Bowie"}
["Deborah","Noah","Martha","Bowie"]
```

You cannot use a dynamic expression to define a property name (although there are ways to define property names programmatically. See "Using %Set(), %Get(), and %Remove()" for details).

Of course, literal constructors are not the only way to manipulate object properties and array elements. For example, the following code creates an empty dynamic object and uses standard object dot syntax to define the contents:

```
   set dynArray = []
   set dynArray."0" = "200" + "33"
   set dynArray."1" = {}
   set dynArray."1".foo = $CHAR(dynArray."0")
   write dynArray.%ToJSON()

[233,{"foo":"é"}]
```

In this example, literal constructors are used only to create empty dynamic entities. The assignment statements obey a few simple rules:

- The assigned values are standard ObjectScript expressions. The value for *dynArray."0"* is evaluated as a numeric expression and the sum is returned as canonical form integer `233`. The $CHAR function later uses that value to return ASCII character 233, which is `"é"`.
- Array elements are addressed by array index numbers, which must be numeric literals enclosed in double quotes. Dynamic arrays are zero-based.
- Object properties are addressed by property names. Although property names are string literals, double quotes are optional if the property name is a valid class member name.
- If the specified entity member does not yet exist, it will be created when you assign a value to it.

As previously mentioned, values are always stored and retrieved in ObjectScript format regardless of how they are represented in JSON syntax. The following examples demonstrate a few more facts that you should be aware of when using dot syntax.

Creating dynamic object properties with dot syntax

This example uses a literal constructor and dot syntax to create dynamic object *dynObj*, containing properties named *A*, *a*, and *C quote*. In the literal string, all property names must be quoted. In the `set` statements and the `write` statement, quotes are not required for property names *a* or *A*, but must be used for *C quote*:

```
   set dynObj = {"a":"stuff"}
   set dynObj."C quote" = " ""C quote"" contains a space "
   set dynObj.a = " lower case ""a"" "
   set dynObj.A = " upper case ""A"" "
   write !,dynObj.%ToJSON()

{"a":" lower case \"a\" ","C quote":" \"C quote\" contains a space ","A":" upper case \"A\" "}
```

Dynamic objects are unordered lists, so values will not necessarily be stored in the order they were created. See "Iterating over a Dynamic Entity with %GetNext()" for examples that demonstrate this.

Creating dynamic array elements with dot syntax

Dynamic arrays are zero-based. This example assigns a value to array element 3 before defining element 2. Elements do not have to be defined in order, and element 2 could have been left undefined. See "Understanding Sparse Arrays and Unassigned Values" for detailed information.

```
set dynArray = [true,false]
set dynArray."3" = "three"
set dynArray."2" = 0
write dynArray.%ToJSON()
```

```
[true,false,0,"three"]
```

Although the first two elements were defined and stored as JSON boolean values `true` and `false`, they are returned as integers `1` and `0`, which are the equivalent ObjectScript boolean values:

```
write "0=/"_dynArray."0"_"/, 1=/"_dynArray."1"_"/, 2=/"_dynArray."2"_"/, 3=/"_dynArray."3"_"/"
```

```
0=/1/, 1=/0/, 2=/0/, 3=/three/
```

Since stored values are always returned in ObjectScript format, JSON `true`, `false`, and `null` are returned as ObjectScript `0`, `1`, and `""` (empty string). However, the original JSON values are preserved in the dynamic entity and can be recovered if necessary. See "Working with Datatypes" for information on identifying the original datatype of a stored value.

Note: **Dot syntax should not be used with very long property names**

Although dynamic object properties can have names of any length, ObjectScript cannot use property names longer than 180 characters. If a dynamic object property name exceeds this limit, an attempt to use the name in dot syntax will result in a misleading `<PROPERTY DOES NOT EXIST>` error, even though the property exists and the name is valid. You can avoid this error by using the **%Set()** and **%Get()** methods, which accept property names of any length.

2.3 Using %Set(), %Get(), and %Remove()

Although literal constructors and dot syntax can be used to create dynamic entity members and manipulate values, they are not adequate for all purposes. Dynamic entities provide **%Set()**, **%Get()**, and **%Remove()** methods for full programmatic control over create, read, update, and delete operations.

One of the most important advantages to these methods is that member identifiers (property names and array index numbers) do not have to be literals. You can use ObjectScript variables and expressions to specify both values and identifiers.

Specifing values and identifiers programmatically with %Set(), %Get(), and %Remove()

The following example creates an object using literal constructor **{}**, and calls the **%Set()** method of the new object to add a series of properties named `prop`*n* with a value of `100+`*n*. Both names and values are defined by ObjectScript expressions:

```
set dynObj = {}
for i=1:1:5 { do dynObj.%Set("prop"_i,100+i) }
write dynObj.%ToJSON()
```

```
{"prop1":101,"prop2":102,"prop3":103,"prop4":104,"prop5":105}
```

The same variables can be used with **%Get()** to retrieve the property values:

```
for i=1:1:5 { write dynObj.%Get("prop"_i)_" " }
```

```
101 102 103 104 105
```

The **%Remove()** method deletes the specified member from the dynamic entity and returns the value. This example removes three of the five properties and concatenates the return values to string *removedValues*. The `write` statement displays the string of removed values and the current contents of *dynObj*:

```
   set removedValues = ""
   for i=2:1:4 { set removedValues = removedValues_dynObj.%Remove("prop"_i)_" " }
   write "Removed values: "_removedValues,!,"Remaining properties: "_dynObj.%ToJSON()

Removed values: 102 103 104
Remaining properties: {"prop1":101,"prop5":105}
```

Note: Although a `for` loop is used in these simple examples, the normal iteration method would be **%GetNext()** (described later in "Iterating over a Dynamic Entity with %GetNext()").

Both **%Get()** and **%Remove()** return an ObjectScript value for the specified member, but there is an important difference in how embedded dynamic entities are returned:

- **%Get()** returns the value by reference. The return value is an OREF (object reference) to the property or element, which in turn contains a reference to the embedded entity.
- **%Remove()** destroys the specified property or element (making the member OREF invalid), but returns a valid OREF that points directly to the formerly embedded entity.

Retrieving a nested dynamic entity with %Get() and %Remove()

In the following example, the value of property *dynObj.address* is a dynamic object. The **%Get()** statement stores a reference to the property (not the property value) in variable *addrPointer*. At this point, *addrPointer* can be used to access the *road* property of embedded entity *address*:

```
   set dynObj = {"name":"greg", "address":{"road":"Old Road"}}
   set addrPointer = dynObj.%Get("address")
   set dynObj.address.road = "New Road"
   write "Value of "_addrPointer_" is "_addrPointer.road

Value of 2@%Library.DynamicObject is New Road
```

The **%Remove()** statement destroys the property and returns a new OREF to the property value.

```
   set addrRemoved =  dynObj.%Remove("address")
   write "OREF of removed property: "_addrPointer,!,"OREF returned by %Remove(): "_addrRemoved

OREF of removed property: 2@%Library.DynamicObject
OREF returned by %Remove(): 3@%Library.DynamicObject
```

After the call to **%Remove()**, *addrRemoved* contains a valid OREF to the formerly embedded dynamic object.

```
   write addrRemoved.%ToJSON()

{"road":"New Road"}
```

You can use the **%Remove()** method to remove members in any order. This has different implications for objects and arrays, as demonstrated in the following examples.

Removing an object property

Object properties have no fixed order. This means that properties can be destroyed in any order, but removing a property and adding another may also change the order in which properties are serialized and returned. The following example creates a dynamic object, and defines three properties with three consecutive calls to **%Set()**:

```
   set dynObject={}.%Set("propA","abc").%Set("PropB","byebye").%Set("propC",999)
   write dynObject.%ToJSON()

{"propA":"abc","PropB":"byebye","propC":999}
```

Now **%Remove()** is called to destroy property *PropB*, after which new property *PropD* is added. The resulting dynamic object does not serialize its properties in the order they were created:

```
do dynObject.%Remove("PropB")
set dynObject.propD = "added last"
write dynObject.%ToJSON()
```

```
{"propA":"abc","propD":"added last","propC":999}
```

This also affects the order in which iterator method **%GetNext()** returns properties. See Iterating over a Dynamic Entity with %GetNext() for a similar example that uses **%GetNext()**.

Removing an array element

An array is a zero-based ordered list. When you call **%Remove()** on an element, all elements after that one will have their array index number decremented by 1. The following example makes three consecutive calls to `%Remove(1)`, removing a different element each time:

```
set dynArray = ["a","b","c","d","e"]
set removedValues = ""
for i=1:1:3 { set removedValues = removedValues_dynArray.%Remove(1)_" " }
write "Removed values: "_removedValues,!,"Array size="_dynArray.%Size()_": "_dynArray.%ToJSON()
```

```
Removed values: b c d
Array size=2: ["a","e"]
```

A stack operation is usually implemented with **%Push()** and **%Pop()** rather than **%Set()** and **%Remove()**, but you can implement a queue by replacing **%Pop()** with `%Remove(0)` (see "Using %Push and %Pop with Dynamic Arrays").

%Remove() works the same way with all arrays, including those that contain elements with undefined values. See "Understanding Sparse Arrays and Unassigned Values" for an example demonstrating how **%Remove()** works with sparse arrays.

2.3.1 Assigning Dynamic Entities as Property Values

You can use **%Set()** or **%Push()** to nest a dynamic entity within another dynamic entity. For example, you can assign a dynamic object as a property value or an array element. An earlier example in this chapter showed how to retrieve a nested object (see "Retrieving a nested dynamic entity with %Get() and %Remove()"). The following example demonstrates one way to create a nested object.

Assigning a dynamic entity as a property value

This example creates a dynamic object with a property named *myData*, which has another dynamic object as its value:

```
{"myData":{"myChild":"Value of myChild"}}
```

The following code creates this object. It is not necessary to specify **%Set()** arguments as variables, but doing so will allow you to assign any valid name or value at runtime:

```
set mainObj = {}
set mainPropName="myData"

set nestedObj = {}
set nestedPropName="myChild"
set nestedPropValue="Value of myChild"

do nestedObj.%Set(nestedPropName, nestedPropValue)
do mainObj.%Set(mainPropName,nestedObj)
write mainObj.%ToJSON()
```

This code produces the following output:

```
USER>write mainObj.%ToJSON()
{"myData":{"myChild":"Value of myChild"}}
```

Note: **Do not use the *type* parameter with object values**

The **%Set()** method has an optional *type* parameter that allows you to specify the datatype of the *value* argument in some limited cases (see "Overriding a Default Datatype with %Set() or %Push()"). The *type* parameter cannot be used when the *value* argument is a dynamic entity. An error will be thrown if you attempt to do so.

2.4 Method Chaining

The **%Set()**, and **%Push()** methods return a reference to the entity that they have modified. The returned reference can immediately be used to call another method on the same entity, within the same expression.

The dynamic entity that begins a chain can be either a constructor (**{}**, or **[]**) or an existing entity. Methods **%Set()** and **%Push()** return chainable references and can be called from anywhere in the chain. The last item in a chain can be any method available to the entity.

In the following example, a single `write` statement uses chained calls to **%FromJSON()**, **%Set()**, **%Push()**, and **%ToJSON()** to create, modify, and display a dynamic array:

```
   set jstring = "[123]"
  write [].%FromJSON(jstring).%Set(1,"one").%Push("two").%Push("three").%Set(1,"final value").%ToJSON()

[123,"final value","two","three"]
```

%FromJSON() is only useful as the first method call in a chain, since it does not return a modified version of the calling entity. Instead, it simply ignores the calling entity and returns an entirely new instance deserialized from a JSON string. For more information, see "Converting Dynamic Entities to and from JSON".

You could also start a chain by retrieving a nested entity with **%Get()**, **%Pop()**, **%GetNext()**, or **%Remove()**.

2.5 Error Handling

Dynamic entities throw exceptions in the case of an error, rather than returning a %Status value. In the following example, the thrown exception includes enough information to conclude that the second character in the method argument is invalid:

```
   set invalidObject = {}.%FromJSON("{:}")

<THROW>%FromJSON+37^%Library.DynamicAbstractObject.1 *%Exception.General Parsing error 3 Line 1 Offset
 2
```

When dealing with dynamic data, it is always wise to assume that some data will not fit your expectations. Any code that makes use of dynamic objects should be surrounded with a `TRY-CATCH` block at some level (see "The TRY-CATCH Mechanism"). For example:

```
   TRY {
      set invalidObject = {}.%FromJSON("{:}")
   }
   CATCH errobj {
      write errobj.Name_", "_errobj.Location_", error code "_errobj.Code,!
      RETURN
   }

Parsing error, Line 1 Offset 2, error code 3
```

2.6 Converting Dynamic Entities to and from JSON

You can use the **%ToJSON()** method to serialize a dynamic entity (convert it to a JSON string) and the **%FromJSON()** and **%FromJSONFile()** methods to deserialize (convert JSON to a dynamic entity).

Serializing a dynamic entity to JSON

The following example creates and modifies a dynamic object, and then uses **%ToJSON()** to serialize it and display the resulting string:

```
set dynObject={"prop1":true}.%Set("prop2",123).%Set("prop3","foo")
set objString = dynObject.%ToJSON()
write objString

{"prop1":true,"prop2":123,"prop3":"foo"}
```

A dynamic array is serialized in the same way:

```
set dynArray=[].%Push("1st value").%Push("2nd value").%Push("3rd value")
set arrayString = dynArray.%ToJSON()
write arrayString

["1st value","2nd value","3rd value"]
```

Both of these examples use method chaining (see "Method Chaining" earlier in this chapter).

Deserializing from JSON to a dynamic object

The **%FromJSON()** method converts a JSON string to a dynamic entity. The following example constructs a dynamic array and serializes it to string *jstring*. A call to **%FromJSON()** deserializes *jstring* to a new dynamic entity named *newArray*, which is then modified and displayed:

```
set jstring=["1st value","2nd value","3rd value"].%ToJSON()
set newArray={}.%FromJSON(jstring)
do newArray.%Push("new value")
write "New entity:"_newArray.%ToJSON()

New entity:["1st value","2nd value","3rd value","new value"]
```

Notice this example calls **%FromJSON()** from a dynamic object constructor (**{}**) even though the returned value is a dynamic array. **%FromJSON()** is a class method of %DynamicAbstractObject, and can therefore be called from any dynamic entity or constructor.

If you have JSON data stored in a `.json` file, you can deserialize the data by using the **%FromJSONFile()** method instead of **%FromJSON()**.

Cloning with %ToJSON() and %FromJSON()

Since each call to **%FromJSON()** creates a new dynamic entity, it can be used to duplicate an existing entity or initialize a set of identical entities.

In the following example, the value of property *dynObj.address* is a dynamic object. The property is referenced by variable *addrPointer*, and the property value is cloned by calling **%FromJSON()** to create new dynamic object *addrClone*:

```
set dynObj = {}.%FromJSON({"name":"greg", "address":{"road":"Dexter Ave."}}.%ToJSON())
set addrPointer = dynObj.address
set addrClone = {}.%FromJSON(dynObj.address.%ToJSON())
```

Variable *addrPointer* is just a reference to property *dynObj.address*, but *addrClone* is an independent instance of %DynamicObject that can be modified without affecting the original value:

```
   set addrPointer.road = "Wright Ave."
   set addrClone.road = "Sinister Ave."
   write !,"Property = "_dynObj.address.%ToJSON(),!,"Clone = "_addrClone.%ToJSON()

Property = {"road":"Wright Ave."}
Clone = {"road":"Sinister Ave."}
```

If you have JSON data stored in a `.json` file, you can clone the data by using the **%FromJSONFile()** method instead of **%FromJSON()**.

2.6.1 Serializing Large Dynamic Entities to Streams

If a dynamic entity is large enough, the output of **%ToJSON()** may exceed the maximum possible length for a string (see "String Length Limit"). The examples in this section use a maximum length string named *longStr*. The following code fragment demonstrates how *longStr* is generated:

```
   set longStr=""
   for i=1:1:$SYSTEM.SYS.MaxLocalLength() { set longStr = longStr_"x" }
   write "Maximum string length = "_$LENGTH(longStr)

Maximum string length = 3641144
```

Whenever an expression uses the return value of **%ToJSON()**, the string is built on the program stack, which is subject to the string length limit. For example, a read/write statement such as `write dyn.%ToJSON()` or an assignment statement such as `set x=dyn.%ToJSON()` will attempt to put the string on the stack. The following example adds two copies of *longStr* to a dynamic array and attempts to assign the serialized string to a variable, causing ObjectScript to return a `<MAXSTRING>` error:

```
   set longArray = [(longStr),(longStr)]
   set tooBig = longArray.%ToJSON()

SET tooBig = longArray.%ToJSON()
^
<MAXSTRING>
```

The general solution to this problem is to pass the **%ToJSON()** output by reference in a `DO` command, without actually examining the return value. Output is written directly to the current device, and there is no limit on the length of the output. In the following examples, the device is a stream.

Writing to a file stream

This example writes dynamic object *longObject* to a file and then retrieves it. Variable *longStr* is the value defined at the beginning of this section:

```
   set longObject = {"a":(longStr),"b":(longStr)}
   set file=##class(%File).%New("c:\temp\longObjectFile.txt")
   do file.Open("WSN")
   do longObject.%ToJSON(file)
   do file.Close()

   do file.Open("RS")
   set newObject = {}.%FromJSONFile(file)
   write !,"Property newObject.a is "_$LENGTH(newObject.a)_" characters long."

Property newObject.a is 3641144 characters long.
```

This solution can also be used to read input from other streams.

Reading and writing global character streams

In this example, we serialize two large dynamic entities (using temporary streams because **%ToJSON()** can only serialize one entity per stream). Standard stream handling methods are used to store each temporary stream as a separate line in stream *bigLines*:

```
set tmpArray = ##class(%Stream.GlobalCharacter).%New()
set dyn = [(longStr),(longStr)]
do dyn.%ToJSON(tmpArray)

set tmpObject = ##class(%Stream.GlobalCharacter).%New()
set dyn = {"a":(longStr),"b":(longStr),"c":(longStr)}
do dyn.%ToJSON(tmpObject)

set bigLines = ##class(%Stream.GlobalCharacter).%New()
do bigLines.CopyFrom(tmpArray)
do bigLines.WriteLine()
do bigLines.CopyFrom(tmpObject)
```

Later, we can deserialize each dynamic entity from *bigLines*:

```
do bigLines.Rewind()
while ('bigLines.AtEnd) {
   write !,{}.%FromJSON(bigLines.ReadLineIntoStream())
}
```

```
7@%Library.DynamicArray
7@%Library.DynamicObject
```

3

Iteration and Sparse Arrays

Dynamic entities use a standard iteration method, **%GetNext()**, that works with both objects and arrays. You can also iterate over an array by addressing each element in sequence (with a `for` loop or similar structure), but this may require some knowledge of sparse arrays, which have elements that do not contain values. Since **%GetNext()** avoids problems by skipping those elements, it should be the preferred iteration method whenever possible.

This chapter discusses when and how to use each iteration method. The following topics are covered:

3.1 Iterating over a Dynamic Entity with %GetNext()

All dynamic entities provide the **%GetIterator()** method, which returns an instance of %Iterator (either %Iterator.Object or %Iterator.Array) containing pointers to members of the dynamic object or array. The %Iterator object provides a **%GetNext()** method to get the key and value of each member.

Each call to the **%GetNext()** method advances the iterator cursor and returns `1` (true) if it is positioned on a valid member or `0` (false) if it is beyond the last member. The name or index number of the member is returned in the first output argument and the value in the second. For example:

```
   set test = ["a","b","c"]  // dynamic arrays are zero-based
   set iter = test.%GetIterator()
   while iter.%GetNext(.key, .value) { write "element:"_key_"=/"_value_"/  "}

element:0=/a/  element:1=/b/  element:2=/c/
```

The iterator cursor only moves in one direction; it cannot go back to a previous member or iterate arrays in reverse order.

When iterating over a sparse array, the iterator skips elements with no assigned value. When iterating over an object, properties are not necessarily returned in a predictable order. The following examples demonstrate these differences between array iteration and object iteration.

Iterating over an array

This example creates a sparse array. The array is zero-based and has six elements, but only elements 0, 1, and 5 have an assigned value. The null elements displayed in the JSON string are just placeholders for the unassigned values:

```
   set dynArray=["abc",999]
   set dynArray."5" = "final"
   write dynArray.%Size()_" elements: "_dynArray.%ToJSON()

6 elements: ["abc",999,null,null,null,"final"]
```

%GetNext() will return only the three elements with values, skipping all unassigned elements:

```
   set iterator=dynArray.%GetIterator()
   while iterator.%GetNext(.key,.val) { write !, "Element index: "_key_", value: "_val }

Element index: 0, value: abc
Element index: 1, value: 999
Element index: 5, value: final
```

See the next section (“Understanding Sparse Arrays and Unassigned Values”) for more on sparse arrays.

Iterating over an object

Object properties have no fixed order, which means that properties can be created and destroyed in any order without creating unassigned values, but changing the object may also change the order in which properties are returned by **%GetNext()**. The following example creates an object with three properties, calls **%Remove()** to destroy one property, and then adds another property:

```
   set dynObject={"propA":"abc","PropB":"byebye","propC":999}
   do dynObject.%Remove("PropB")
   set dynObject.propD = "final"
   write dynObject.%Size()_" properties: "_dynObject.%ToJSON()

3 properties: {"propA":"abc","propD":"final","propC":999}
```

When we iterate over the object, **%GetNext()** does not return items in the order they were created:

```
   set iterator=dynObject.%GetIterator()
   while iterator.%GetNext(.key,.val) { write !, "Property name: """_key_""", value: "_val }

Property name: "propA", value: abc
Property name: "propD", value: final
Property name: "propC", value: 999
```

3.2 Understanding Sparse Arrays and Unassigned Values

Dynamic arrays can be *sparse arrays*, meaning that not all elements of the array contain values. You can, for example, assign a value to element 100 of a dynamic array even if the array does not already contain elements 0 to 99. Space in memory is allocated only for the value at element 100. Elements 0 to 99 are *unassigned*, meaning that 0 to 99 are valid element identifiers but do not point to any values in memory. The **%Size()** method would return an array size of 101, but the **%GetNext()** method would skip over the unassigned elements and return only the value in element 100.

The following example creates a sparse array by assigning new values to elements 8 and 11:

```
   set array = ["val_0",true,1,"",null,"val_5"] // values 0 through 5
   do array.%Set(8,"val_8")                     // undefined values 6 and 7 will be null
   set array."11" = "val_11"                    // undefined values 9 and 10 will be null
   write array.%ToJSON()

["val_0",true,1,"",null,"val_5",null,null,"val_8",null,null,"val_11"]
```

No values have been assigned to elements 6, 7, 9, and 10, and they take no space in memory, but they are represented in the JSON string by null values because JSON does not support undefined values.

Using %Remove in sparse arrays

The **%Remove()** method treats an unassigned element just like any other element. It is possible to have an array that consists of nothing but unassigned values. The following example creates a sparse array and then removes unassigned element 0. It then removes element 7, which is now the only element containing a value:

```
   set array = []
   do array.%Set(8,"val_8")
   do array.%Remove(0)
   do array.%Remove(7)
   write "Array size = "_array.%Size()_":",!,array.%ToJSON()

Array size = 7:
[null,null,null,null,null,null,null]
```

See "Using %Set(), %Get(), and %Remove()" for more examples demonstrating **%Remove()**.

Note: **JSON cannot preserve the distinction between null and unassigned values**

Dynamic entities contain metadata that allows them to distinguish between null and unassigned values. JSON does not specify a separate undefined datatype, so there is no canonical way to preserve this distinction when a dynamic entity is serialized to a JSON string. If you do not want the extra null values in your serialized data, you must either remove the unassigned elements before serializing (see "Using %IsDefined() to Test for Valid Values"), or use some application-dependent means to record the distinction as metadata.

3.2.1 Sparse Array Iteration with %Size()

The **%Size()** method returns the number of properties or elements in a dynamic entity. For example:

```
   set dynObject={"prop1":123,"prop2":[7,8,9],"prop3":{"a":1,"b":2}}
   write "Number of properties: "_dynObject.%Size()

Number of properties: 3
```

In sparse arrays, this number includes elements with unassigned values, as demonstrated in the following example. The array created in this example has six elements, but only elements 0, 1, and 5 have an assigned value. The null elements displayed in the JSON string are just placeholders for the unassigned values:

```
   set test=["abc",999]
   set test."5" = "final"
   write test.%Size()_" elements: "_test.%ToJSON()

6 elements: ["abc",999,null,null,null,"final"]
```

Elements 2, 3, and 4 do not have assigned values, but are still treated as valid array elements. Dynamic arrays are zero-based, so the index number of the final element will always be %Size()-1. The following example iterates through all six elements of array *test* in reverse order and uses **%Get()** to return their values:

```
   for i=(test.%Size()-1):-1:0 {write "element "_i_" = /"_test.%Get(i)_"/",!}

element 5 = /final/
element 4 = //
element 3 = //
element 2 = //
element 1 = /999/
element 0 = /abc/
```

The **%Get()** method will return "" (empty string) for numbers greater than %Size()-1, and will throw an exception for negative numbers. See "Working with Datatypes" for information on how to distinguish between unassigned values, empty strings, and null values.

Note: The iteration technique shown here is useful only for specialized purposes (such detecting unassigned values in an array or iterating over an array in reverse order). In most cases you should use **%GetNext()**, which skips over unassigned elements and can be used for dynamic objects as well as dynamic arrays. See the previous section ("Iterating over a Dynamic Entity with %GetNext()") for details.

3.2.2 Using %IsDefined() to Test for Valid Values

The **%IsDefined()** method tests for the existence of a value at a specified property name or array index number. The method returns `1` (true) if the specified member has a value, and `0` (false) if the member does not exist. It will also return false for elements in a sparse array that do not have assigned values.

Unassigned values will be encountered if you use a `for` loop to iterate through a sparse array. The following example creates an array where the first three elements are a JSON `null`, an empty string, and an unassigned value. The `for` loop is deliberately set to go past the end of the array and test for an element with array index `4`:

```
   set dynarray = [null,""]
   set dynarray."3" = "final"
   write dynarray.%ToJSON()
[null,"",null,"final"]

   for index = 0:1:4 {write !,"Element "_index_": "_(dynarray.%IsDefined(index))}

Element 0: 1
Element 1: 1
Element 2: 0
Element 3: 1
Element 4: 0
```

%IsDefined() returns `0` in two cases: element `2` does not have an assigned value, and element `4` does not exist.

ObjectScript returns `""` (empty string) for JSON `null` values such as element `0` in this example. If you need to test for `""` and `null` as well as unassigned values, use **%GetTypeOf()** rather than **%IsDefined()** (see "Resolving Null, Empty String, and Unassigned Values").

Note: As mentioned in the previous section, you should not use a `for` loop for iteration except in a few unusual situations. In most cases you should use the **%GetNext()** method, which skips unassigned values (see "Iterating over a Dynamic Entity with %GetNext()").

The **%IsDefined()** method can also be used to test for the existence of an object property. The following code creates dynamic array *names* with three string values, and then uses the first two strings to create object *dynobj* with properties *prop1* and *prop2*.

```
   set names = ["prop1","prop2","noprop"]
   set dynobj={}.%Set(names."0",123).%Set(names."1",456)
   write dynobj.%ToJSON()

{"prop1":123,"prop2":456}
```

The following code uses **%IsDefined()** to determine which strings have been used as property names in *dynobj*:

```
   for name = 0:1:2 {write !,"Property "_names.%Get(name)_": "_(dynobj.%IsDefined(names.%Get(name)))}

Property prop1: 1
Property prop2: 1
Property noprop: 0
```

3.3 Using %Push and %Pop with Dynamic Arrays

The **%Push()** and **%Pop()** methods are only available for dynamic arrays. They are work exactly like **%Set()** and **%Remove()** except that they always add or remove the last element of the array. For example, the following code produces the same results with either set of methods (see "Method Chaining" for details about calling **%Set()** or **%Push()** several times in the same statement):

```
   set array = []
   do array.%Set(array.%Size(), 123).%Set(array.%Size(), 456)
   write "removed "_array.%Remove(array.%Size()-1)_", leaving "_array.%ToJSON()

removed 456, leaving [123]

   set array = []
   do array.%Push(123).%Push(456)
   write "removed "_array.%Pop()_", leaving "_array.%ToJSON()

removed 456, leaving [123]
```

Although **%Push()** and **%Pop()** are intended for stack operations, you could implement a queue by substituting `%Remove(0)` for `%Pop()`.

The following example builds an array with **%Push()**, and then removes each element in reverse order with **%Pop()**.

Using %Push() and %Pop() to build an array and tear it down

Build an array containing a nested array. The final call to **%Push()** specifies the optional *type* argument to store a boolean value as JSON `false` rather than ObjectScript `0` (see "Overriding a Default Datatype with %Set() or %Push()"):

```
   set array=[]
   do array.%Push(42).%Push("abc").%Push([])
   do array."2".%Push("X").%Push(0,"boolean")
   write array.%ToJSON()

[42,"abc",["X",false]]
```

Remove all elements of the nested array. Like all dynamic entity methods, **%Pop()** will return ObjectScript `0` rather than JSON `false`:

```
   for i=0:1:1 {write "/"_array."2".%Pop()_"/ "}
/0/ /X/

   write array.%ToJSON()
[42,"abc",[]]
```

Now remove all elements of the main array, including the empty nested array:

```
   for i=0:1:2 {write "/"_array.%Pop()_"/ "}
/2@%Library.DynamicArray/ /abc/ /42/

   write array.%ToJSON()
[]
```

These examples use hard coded `for` loops for simplicity. See "Sparse Array Iteration with %Size()" for more realistic examples of array iteration.

4

Working with Datatypes

ObjectScript has no distinct constants equivalent to JSON `true`, `false`, and `null`, and JSON has no concept of array elements with undefined values. This chapter discusses these mismatches and describes the tools provided to deal with them.

4.1 Discovering the Datatype of a Value with %GetTypeOf()

You can use the **%GetTypeOf()** method to get the datatype of a dynamic entity member. A dynamic object property or array element can have any one of following datatypes:

- An object datatype:
  - `array` — a dynamic array reference
  - `object` — a dynamic object reference
  - `oref` — a reference to an object that is not a dynamic entity
- A literal value:
  - `number` — a canonical numeric value
  - `string` — a string literal or an expression that evaluates to a string literal
- A JSON literal:
  - `boolean` — a JSON literal `true` or `false`
  - `null` — a JSON literal `null`
- No datatype:
  - `unassigned` — the property or element exists, but has no assigned value.

Using %GetTypeOf with objects

When you use this method with an object, the argument is the name of the property. For example:

```
   set dynobj={"prop1":123,"prop2":[7,8,9],"prop3":{"a":1,"b":2}}
   set iter = dynobj.%GetIterator()
   while iter.%GetNext(.name) {write !,"Datatype of "_name_" is "_(dynobj.%GetTypeOf(name))}

Datatype of prop1 is number
Datatype of prop2 is array
Datatype of prop3 is object
```

Using %GetTypeOf with arrays

When you use this method with an array, the argument is the index of the element. The following example examines a sparse array, where element 2 does not have an assigned value. The example uses a `for` loop because **%GetNext()** would skip the unassigned element:

```
   set dynarray = [12,34]
   set dynarray."3" = "final"
   write dynarray.%ToJSON()
[12,34,null,"final"]

   for index = 0:1:3 {write !,"Datatype of "_index_" is "_(dynarray.%GetTypeOf(index))}
Datatype of 0 is number
Datatype of 1 is number
Datatype of 2 is unassigned
Datatype of 3 is string
```

Distinguishing between `array` or `object` and `oref`

The datatype of a dynamic entity will be either `array` or `object`. An InterSystems IRIS object that is not a dynamic entity will be datatype `oref`. In the following example, each property of object *dyn* is one of these three datatypes. Property `dynobject` is class %DynamicObject, property `dynarray` is %DynamicArray, and property `streamobj` is %Stream.GlobalCharacter:

```
   set
dyn={"dynobject":{"a":1,"b":2},"dynarray":[3,4],"streamobj":(##class(%Stream.GlobalCharacter).%New())}

   set iterator=dyn.%GetIterator()
  while iterator.%GetNext(.key,.val) { write !, "Datatype of "_key_" is: "_dyn.%GetTypeOf(key)
 }

Datatype of dynobject is: object
Datatype of dynarray is: array
Datatype of streamobj is: oref
```

4.2 Overriding a Default Datatype with %Set() or %Push()

By default, the system automatically interprets a **%Set()** or **%Push()** value argument as an object datatype (`object`, `array`, or `oref`) or an ObjectScript literal datatype (`string` or `number`). You can not directly pass JSON literals `null`, `true` or `false` as values because the argument is interpreted as an ObjectScript literal or expression. For example, the following code throws an error because the value `true` is interpreted as a variable name:

```
   do o.%Set("prop3",true)

DO o.%Set("prop3",true)
^
<UNDEFINED> *true
```

ObjectScript uses "" (an empty string) for null, 0 for boolean false, and a non-zero number for boolean true. To deal with this problem, **%Set()** and **%Push()** take an optional third argument to specify the datatype of the value. The third argument can be JSON `boolean` or `null`. For example:

```
   write {}.%Set("a",(2-4)).%Set("b",0).%Set("c","").%ToJSON()
{"a":-2,"b":0,"c":""}

   write {}.%Set("a",(2-4),"boolean").%Set("b",0,"boolean").%Set("c","","null").%ToJSON()
{"a":true,"b":false,"c":null}
```

The third argument can also be `string` or `number` if the value could be interpreted as a number:

```
   write [].%Push("023"_"04").%Push(5*5).%ToJSON()
["02304",25]

   write [].%Push(("023"_"04"),"number").%Push((5*5),"string").%ToJSON()
[2304,"25"]
```

4.3 Resolving JSON Null and Boolean Values

In JSON syntax, the values `true`, `false`, and `null` are distinct from values `1`, `0`, and "" (empty string), but ObjectScript does not make this distinction. When JSON values are retrieved from an element or property, they are always cast to ObjectScript-compatible values. This means that JSON `true` is always returned as `1`, `false` as `0`, and `null` as "". In most cases this will be the desired result, since the return value can be used in an ObjectScript expression without first converting it from JSON format. The dynamic entity retains the original JSON or ObjectScript value internally, so you can use **%GetTypeOf()** to identify the actual datatype if necessary.

In the following example, the dynamic array constructor specifies JSON `true`, `false`, and `null` values, numeric and string literal values, and ObjectScript dynamic expressions (which evaluate to ObjectScript boolean values `1` and `0`):

```
   set test = [true,1,(1=1),false,0,(1=2),"",null]
   write test.%ToJSON()

[true,1,1,false,0,0,"",null]
```

As you can see above, the values assigned in the constructor have been preserved in the resulting dynamic array, and are displayed properly when serialized as a JSON string.

The following example retrieves and displays the array values. As expected, JSON values `true`, `false`, and `null` are cast to ObjectScript-compatible values `1`, `0`, and "":

```
   set iter = test.%GetIterator()
   while iter.%GetNext(.key,.val){write "/"_val_"/ "}

/1/ /1/ /1/ /0/ /0/ /0/ // //
```

This example uses **%GetNext()**, but you would get the same results if you retrieved values with **%Get()**, **%Pop()**, or dot syntax.

When necessary, you can use the **%GetTypeOf()** method to discover the original datatype of the value. For example:

```
   set iter = test.%GetIterator()
   while iter.%GetNext(.key,.val) {write !,key_": /"_test.%Get(key)_"/ = "_test.%GetTypeOf(key)}

0: /1/ = boolean
1: /1/ = number
2: /1/ = number
3: /0/ = boolean
4: /0/ = number
5: /0/ = number
6: // = string
7: // = null
```

Note: **Datatypes in Dynamic Objects**

Although this chapter concentrates on dynamic arrays, the same datatype conversions apply to dynamic object values. The examples in this section will work exactly the same if dynamic array *test* is replaced with the following dynamic object:

```
set test = {"0":true,"1":1,"2":(1=1),"3":false,"4":0,"5":(1=2),"6":"","7":null}
```

Except for this line, none of the example code has to be changed. The property names in this object are numeric strings corresponding to the index numbers of the original array, so even the output will be identical.

4.4 Resolving Null, Empty String, and Unassigned Values

Although you can assign a JSON `null` value to an element or property, the value will always be returned as `""` (ObjectScript empty string). An empty string will also be returned if you attempt to get the value of an unassigned element. You can use **%GetTypeOf()** to identify the actual datatype in each case.

This example will test a sparse array containing a JSON `null` value and an empty string. Although array element `2` has no assigned value, it will be represented in the JSON string by a `null`:

```
set array = [null,""]
do array.%Set(3,"last")
write array.%ToJSON()

[null,"",null,"last"]
```

In most cases you would use **%GetNext()** to retrieve array values, but this example uses a `for` loop to return unassigned values that **%GetNext()** would skip. The index number of the last element is `array.%Size()-1`, but the loop counter is deliberately set to go past the end of the array:

```
for i=0:1:(array.%Size()) {write !,i_". value="""_array.%Get(i)_""" type="_array.%GetTypeOf(i)}

0. value="" type=null
1. value="" type=string
2. value="" type=unassigned
3. value="last" type=string
4. value="" type=unassigned
```

In this example, **%Get()** returns an empty string in four different cases:

1. element `0` is a JSON `null` value, which **%GetTypeOf()** identifies as datatype `null`.
2. element `1` is an empty string, which is identified as datatype `string`.
3. Element `2` has no value, and is identified as datatype `unassigned`.
4. Although element `3` is the last one in the array, the example attempts to get a datatype for non-existent element `4`, which is also identified as datatype `unassigned`. Valid array index numbers will always be less than `array.%Size()`.

Note: The distinction between `null` and `unassigned` is ObjectScript metadata that will not be preserved when a dynamic entity is serialized to a JSON string. All `unassigned` elements will be serialized as `null` values. See "Understanding Sparse Arrays and Unassigned Values" for details.

5

Using the JSON Adaptor

The JSON Adapter is a means for mapping ObjectScript objects (registered, serial or persistent) to JSON text or dynamic entities. This chapter covers the following topics:

- Exporting and Importing — introduces JSON enabled objects and demonstrates %JSON.Adaptor import and export methods.
- Mapping with Parameters — describes property parameters that control how object properties are converted to JSON fields.
- Using XData Mapping Blocks — describes a way to apply multiple parameter mappings to a single class.
- Formatting JSON — demonstrates how to format JSON strings with %JSON.Formatter.
- %JSON Quick Reference — provides a brief description of each %JSON class member discussed in this chapter.

5.1 Exporting and Importing

Any class you would like to serialize from and to JSON needs to subclass %JSON.Adaptor, which includes the following methods:

- %JSONExport() serializes a JSON enabled class as a JSON document and writes it to the current device.
- %JSONExportToStream() serializes a JSON enabled class as a JSON document and writes it to a stream.
- %JSONExportToString() serializes a JSON enabled class as a JSON document and returns it as a string.
- %JSONImport() imports either JSON as a string or stream, or a subclass of %DynamicAbstractObject, and returns an instance of the JSON enabled class.

To demonstrate these methods, the examples in this section will use these two classes:

JSON enabled classes Model.Event and Model.Location

```
Class Model.Event Extends (%Persistent, %JSON.Adaptor)
{
  Property Name As %String;
  Property Location As Model.Location;
}
```

and

```
Class Model.Location Extends (%Persistent, %JSON.Adaptor)
{
  Property City As %String;
  Property Country As %String;
}
```

As you can see, we have a persistent event class, which links to a location. Both classes inherit from %JSON.Adaptor. This enables us to populate an object graph and directly export it as a JSON string.

Exporting an object to a JSON string

```
set event = ##class(Model.Event).%New()
set event.Name = "Global Summit"
set location = ##class(Model.Location).%New()
set location.Country = "United States of America"
set event.Location = location
do event.%JSONExport()
```

This code displays the following JSON string:

```
{"Name":"Global Summit","Location":{"City":"Boston","Country":"United States of America"}}
```

You can assign the JSON string to a variable by using **%JSONExportToString()** instead of **%JSONExport()**:

```
do event.%JSONExportToString(.jsonEvent)
```

Finally, you can reverse the process and convert the JSON string back to an object with **%JSONImport()**. This example takes string variable *jsonEvent* from the previous example and converts it back to a Model.Event object:

Importing a JSON string into an object

```
set eventTwo = ##class(Model.Event).%New()
do eventTwo.%JSONImport(jsonEvent)
write eventTwo.Name,!,eventTwo.Location.City
```

writes:

```
Global Summit
Boston
```

Both import and export work for arbitrarily nested structures.

5.2 Mapping with Parameters

You can specify the mapping logic for each individual property by setting corresponding parameters. (If you are familiar with the %XML.Adaptor, this is a similar process).

We can change the mapping for the Model.Event class (defined in the previous section) by specifying property parameters:

```
Class Model.Event Extends (%Persistent, %JSON.Adaptor)
{
  Property Name As %String(%JSONFIELDNAME = "eventName");
  Property Location As Model.Location(%JSONINCLUDE = "INPUTONLY");
}
```

This mapping introduces two changes:

- The property *Name* will be mapped to a JSON field named *eventName*.

- The *Location* property will still be used as input by **%JSONImport()**, but will be ignored by **%JSONExport()** and other export methods.

Previously, **%JSONExport()** was called on an instance of the unmodified Model.Event class, and the following JSON string was returned:

```
{"Name":"Global Summit","Location":{"City":"Boston","Country":"United States of America"}}
```

If we call **%JSONExport()** on an instance of the remapped Model.Event (using the same property values), the following string will be returned:

```
{"eventName":"Global Summit"}
```

There are various parameters available to allow you to tweak the mapping:

- %JSONFIELDNAME (properties only) sets the string to be used as the field name in JSON content (value is the property name by default).
- %JSONIGNOREINVALIDFIELD controls handling of unexpected fields in the JSON input.
- %JSONIGNORENULL allows the developer to override the default handling of empty strings for string properties.
- %JSONINCLUDE (properties only) specifies whether this property will be included in the JSON output or input (valid values are `"inout"` (the default), `"outputonly"`, `"inputonly"`, or `"none"`).
- %JSONNULL specifies how to store empty strings for string properties.
- %JSONREFERENCE specifies how to project object references to JSON fields. Options are `"OBJECT"` (the default), `"ID"`, `"OID"` and `"GUID"`.

See the reference section "%JSON.Adaptor Class and Property Parameters" later in this chapter for more information.

5.3 Using XData Mapping Blocks

Instead of setting the mapping parameters on the property level, you can specify a mapping in a special XData Mapping block and apply the mapping when you call an import or export method.

The following code defines another version of the Model.Event class used in the previous two sections. In this version, no property parameters are specified, but we define an XData Mapping block named *OnlyLowercaseTopLevel* that specifies the same parameter settings as the properties in the previous version:

```
Class Model.Event Extends (%Persistent, %JSON.Adaptor)
{
  Property Name As %String;
  Property Location As Model.Location;

  XData OnlyLowercaseTopLevel
  {
    <Mapping xmlns="http://www.intersystems.com/jsonmapping">
      <Property Name="Name" FieldName="eventName"/>
      <Property Name="Location" Include="INPUTONLY"/>
    </Mapping>
  }
}
```

There is one important difference: JSON mappings in XData blocks do not change the default behavior, but you can apply them by specifying the block name in the optional *%mappingName* parameter of the import and export methods. For example:

```
do event.%JSONExport("OnlyLowercaseTopLevel")
```

displays:

```
{"eventName":"Global Summit"}
```

just as if the parameters had been specified in the property definitions.

If there is no XData block with the provided name, the default mapping will be used. With this approach, you can configure multiple mappings and reference the mapping you need for each call individually, granting you even more control while making your mappings more flexible and reusable.

5.3.1 Defining an XData Mapping Block

A JSON enabled class can define an arbitrary number of additional mappings. Each mapping is defined in a separate XData block of the following form:

```
XData {MappingName}
{
  <Mapping  {ClassAttribute}="value" [...] xmlns="http://www.intersystems.com/jsonmapping".>
    <{Property Name}="PropertyName" {PropertyAttribute}="value" [...] />
    [... more Property elements]
  </Mapping>
}
```

where *{MappingName}*, *{ClassAttribute}*, *{Property Name}*, and *{PropertyAttribute}* are defined as follows:

- *MappingName*

  The name of the mapping for use by the %JSONREFERENCE parameter or Reference attribute.

- *ClassAttribute*

  Specifies a class parameter for the mapping. The following class attributes can be defined:

  - `Mapping` — name of an XData mapping block to apply.
  - `IgnoreInvalidField` — specifies class parameter %JSONIGNOREINVALIDFIELD.
  - `Null` — specifies class parameter %JSONNULL.
  - `IgnoreNull` — specifies class parameter %JSONIGNORENULL.
  - `Reference` — specifies class parameter %JSONREFERENCE.

- *PropertyName*

  The name of the property which is being mapped.

- *PropertyAttribute*

  Specifies a property parameter for the mapping. The following property attributes can be defined:

  - `FieldName` — specifies property parameter %JSONFIELDNAME (same as property name by default).
  - `Include` — specifies property parameter %JSONINCLUDE (valid values are `"inout"` (the default), `"outputonly"`, `"inputOnly"`, or `"none"`).
  - `Mapping` — name of a mapping definition to apply to an object property.
  - `Null` — overrides class parameter %JSONNULL.
  - `IgnoreNull` — overrides class parameter %JSONIGNORENULL.
  - `Reference` — overrides class parameter %JSONREFERENCE.

5.4 Formatting JSON

%JSON.Formatter is a class with a very simple interface that allows you to format your dynamic objects and arrays and JSON strings into a more human-readable representation. All methods are instance methods, so you always start by retrieving an instance:

```
set formatter = ##class(%JSON.Formatter).%New()
```

The reason behind this choice is that you can configure your formatter to use certain characters for line terminators and indentation once (for example, whitespaces vs. tabs; see the property list at the end of this section), and then use it wherever you need it.

The **Format()** method takes either a dynamic entity or a JSON string. Here is a simple example using a dynamic object:

```
dynObj = {"type":"string"}
do formatter.Format(dynObj)
```

The resulting formatted string is displayed on the current device:

```
{
  "type":"string"
}
```

Format methods can direct the output to the current device, a string, or a stream:

- Format() formats a JSON document using the specified indentation and writes it to the current device.
- FormatToStream() formats a JSON document using the specified indentation and writes it to a stream.
- FormatToString() formats a JSON document using the specified indentation and writes it to a string, or serializes a JSON enabled class as a JSON document and returns it as a string.

In addition, the following properties can be used to control indentation and line breaks:

- Indent specifies whether the JSON output should be indented
- IndentChars specifies the sequence of characters to be used for each indent level (defaults to one space per level).
- LineTerminator specifies the character sequence to terminate each line when indenting.

5.5 %JSON Quick Reference

This section provides a quick reference for the %JSON methods, properties, and parameters discussed in this chapter. For the most complete and up to date information, see %JSON in the class reference.

5.5.1 %JSON.Adaptor Methods

These methods provide the ability to serialize from and to JSON. See “Exporting and Importing” for more information and examples.

%JSONExport()

%JSON.Adaptor.**%JSONExport()** serializes a JSON enabled class as a JSON document and writes it to the current device.

```
method %JSONExport(%mappingName As %String = "") as %Status
```

parameters:

- *%mappingName* (optional) — the name of the mapping to use for the export. The base mapping is represented by "" and is the default.

%JSONExportToStream()

%JSON.Adaptor.**%JSONExportToStream()** serializes a JSON enabled class as a JSON document and writes it to a stream.

```
method %JSONExportToStream(ByRef export As %Stream.Object,
   %mappingName As %String = "") as %Status
```

parameters:

- *export* — stream containing the serialized JSON document.
- *%mappingName* (optional) — the name of the mapping to use for the export. The base mapping is represented by "" and is the default.

%JSONExportToString()

%JSON.Adaptor.**%JSONExportToString()** serializes a JSON enabled class as a JSON document and returns it as a string.

```
method %JSONExportToString(ByRef %export As %String,
   %mappingName As %String = "") as %Status
```

parameters:

- *export* — string containing the serialized JSON document.
- *%mappingName* (optional) — the name of the mapping to use for the export. The base mapping is represented by "" and is the default.

%JSONImport()

%JSON.Adaptor.**%JSONImport()** imports JSON or dynamic entity input into this object.

```
method %JSONImport(input, %mappingName As %String = "") as %Status
```

parameters:

- *input* — either JSON as a string or stream, or a subclass of %DynamicAbstractObject.
- *%mappingName* (optional) — the name of the mapping to use for the import. The base mapping is represented by "" and is the default.

%JSONNew()

%JSON.Adaptor.**%JSONNew()** gets an instance of an JSON enabled class. You may override this method to do custom processing (such as initializing the object instance) before returning an instance of this class. However, this method should not be called directly from user code.

```
classmethod %JSONNew(dynamicObject As %DynamicObject,
   containerOref As %RegisteredObject = "") as %RegisteredObject
```

parameters:

- *dynamicObject* — the dynamic entity with the values to be assigned to the new object.
- *containerOref* (optional) — the containing object instance when called from **%JSONImport()**.

5.5.2 %JSON.Adaptor Class and Property Parameters

Unless noted otherwise, a parameter can be specified either for a class or for an individual property. As a class parameter, it specifies the default value of the corresponding property parameter. As a property parameter, it specifies a value that overrides the default. See "Mapping with Parameters" for more information and examples.

%JSONENABLED

Enables generation of property conversion methods.

```
parameter %JSONENABLED = 1;
```

Valid values are:

- `1` — (the default) JSON enabling methods will be generated.
- `0` — method generators will not produce a runnable method.

%JSONFIELDNAME (properties only)

Sets the string to be used as the field name in JSON content.

```
parameter %JSONFIELDNAME
```

By default, the property name is used.

%JSONIGNOREINVALIDFIELD

Controls handling of unexpected fields in the JSON input.

```
parameter %JSONIGNOREINVALIDFIELD = 0;
```

Valid values are:

- `0` — (the default) treat an unexpected field as an error.
- `1` — unexpected fields will be ignored.

%JSONIGNORENULL

Specifies how to store empty strings for string properties. This parameter applies to only true strings (determined by `XSDTYPE = "string"` and `JSONTYPE="string"`).

```
parameter %JSONIGNORENULL = 0;
```

Valid values are:

- `0` — (the default) empty strings in the JSON input are stored as `$char(0)` and `$char(0)` is written to JSON as the string `""`. A missing field in the JSON input is always stored as `""` and `""` is always output to JSON according to the %JSONNULL parameter.
- `1` — both empty strings and missing JSON fields are input as `""`, and both `""` and `$char(0)` are output as field value `""`.

%JSONINCLUDE (properties only)

Specifies whether this property will be included in JSON output or input.

```
parameter %JSONINCLUDE = "inout"
```

Valid values are:

- `"inout"` (the default) — include in both input and output.
- `"outputonly"` — ignore the property as input.
- `"inputOnly"` — ignore the property as output.
- `"none"` — never include the property.

%JSONNULL

Controls handling of unspecified properties.

```
parameter %JSONNULL = 0;
```

Valid values are:

- `0` — (the default) the field corresponding to an unspecified property is skipped during export.
- `1` — unspecified properties are exported as the null value.

%JSONREFERENCE

Specifies how to project object references to JSON fields.

```
parameter %JSONREFERENCE = "OBJECT";
```

Valid values are:

- `"OBJECT"` — (the default) the properties of the referenced class are used to represent the referenced object.
- `"ID"` — the id of a persistent or serial class is used to represent the reference.
- `"OID"`— the oid of a persistent or serial class is used to represent the reference. The oid is projected to JSON in the form *classname,id*.
- `"GUID"` — the GUID of a persistent class is used to represent the reference.

5.5.3 %JSON.Formatter Methods and Properties

The %JSON.Formatter class can be used to format JSON strings, streams, or objects that subclass %DynamicAbstractObject. See the section on "Formatting JSON" for more information and examples.

Format()

%JSON.Formatter.**Format()** formats a JSON document using the specified indentation and writes it to the current device.

```
method Format(input) as %Status
```

parameters:

- *input* — either JSON as a string or stream, or a subclass of %DynamicAbstractObject.

FormatToStream()

%JSON.Formatter.**FormatToStream()** formats a JSON document using the specified indentation and writes it to a stream.

```
method FormatToStream(input, ByRef export As %Stream.Object) as %Status
```

parameters:

- *input* — either JSON as a string or stream, or a subclass of %DynamicAbstractObject.
- *export* — the formatted JSON stream.

FormatToString()

%JSON.Formatter.**FormatToString()** formats a JSON document using the specified indentation and writes it to a string, or serializes a JSON enabled class as a JSON document and returns it as a string.

```
method FormatToString(input, ByRef export As %String = "") as %Status
```

parameters:

- *input* — either JSON as a string or stream, or a subclass of %DynamicAbstractObject.
- *export* (optional) — the formatted JSON stream.

Indent

The %JSON.Formatter.Indent property specifies whether the JSON output should be indented. Defaults to true.

```
property Indent as %Boolean [ InitialExpression = 1 ];
```

IndentChars

The %JSON.Formatter.IndentChars property specifies the character sequence to be used for each indent level if indenting is on. Defaults to one space.

```
property IndentChars as %String [ InitialExpression = " " ];
```

LineTerminator

The %JSON.Formatter.LineTerminator property specifies the character sequence to terminate each line when indenting. Defaults to `$char(13,10)`.

```
property LineTerminator as %String [ InitialExpression = $char(13,10) ];
```

6

Quick Reference for Dynamic Entity Methods

This section provides an overview and references for each of the available dynamic entity methods. Dynamic entities are instances of %Library.DynamicObject or %Library.DynamicArray, both of which extend %Library.DynamicAbstractObject. Each listing in this chapter includes a link to the appropriate online *class reference* documentation.

6.1 Method Details

This section lists all available dynamic entity methods, briefly describing each one and providing links to further information. All methods are available for both objects and arrays, with the exception of **%Push()** and **%Pop()**, which apply only to arrays.

%FromJSON()

Given a valid JSON string, parse it and return an object of datatype %DynamicAbstractObject containing the parsed JSON. If an error occurs during parsing, an exception will be thrown. See "Converting Dynamic Entities to and from JSON" for details and examples.

```
classmethod %FromJSON(str) as %DynamicAbstractObject
```

parameters:

- *str* — The input can be from any one of the following sources:
  - string value containing the source. The value can be an empty string (""), but an error will be thrown if the string contains only white space characters.
  - stream object to read the source from. An error will be thrown if the stream does not contain any characters.

see also: **%FromJSONFile()**, **%ToJSON()**, Serializing Large Dynamic Entities to Streams

class reference: %DynamicAbstractObject.**%FromJSON()**

%FromJSONFile()

Given a valid JSON source, parse the source and return an object of datatype %DynamicAbstractObject containing the parsed JSON. See "Converting Dynamic Entities to and from JSON" for details and examples.

```
classmethod %FromJSONFile(filename) as %DynamicAbstractObject
```

parameters:

- *filename* — name of file URI where the source can be read. The file must be encoded as UTF-8. An exception will be thrown if the file does not contain any characters.

see also: **%FromJSON()**, **%ToJSON()**, Serializing Large Dynamic Entities to Streams

class reference: %DynamicAbstractObject.**%FromJSON()**

%Get()

Given a valid object key or array index, returns the value. If the value does not exist, a null string `""` is returned. See "Using %Set(), %Get(), and %Remove()" for details and examples.

```
method %Get(key, default, type) as %RawString
```

parameters:

- *key* — the object key or array index of the value to be retrieved. An array index must be passed as a canonical integer value. Array indexes begin at position 0.
- *default* — optional value to be returned when the selected array element is undefined. Defaults to empty string if not specified.
- *type* — if defined, indicates that the value of *key* should be returned as the specified type (see %DynamicObject.**%Get()** and %DynamicArray.**%Get()** for details on how values are converted). If this argument is specified, the value must be one of the following strings:
  - `""` (empty string) — same as calling %Get(key) without conversions
  - `"string"` — convert to text string
  - `"string>base64"` — convert to text string then encode into base64
  - `"string<base64"` — convert to text string then decode from base64
  - `"stream"` — place string conversion into %Stream
  - `"stream>base64"` — string encoded into base64 into %Stream
  - `"stream<base64"` — string decoded from base64 into %Stream
  - `"json"` — convert to JSON representation

see also: **%Set()**, **%Remove()**, **%Pop()**

class reference: %DynamicObject.**%Get()** and %DynamicArray.**%Get()**

%GetIterator()

Returns a %Iterator object allowing iteration over all members of a dynamic entity. See "Iterating over a Dynamic Entity with %GetNext()" for details and examples.

```
method %GetIterator() as %Iterator.AbstractIterator
```

see also: **%GetNext()**

class reference: %DynamicObject.**%GetIterator()**, %DynamicArray.**%GetIterator()**, %Iterator.Object, %Iterator.Array

%GetNext()

This is a method of the %Iterator object returned by **%GetIterator()**. It advances the iterator and returns `true` if the iterator is positioned on a valid element, `false` if it is beyond the last element. The *key* and *value* arguments return values for a valid element at the current iterator position. The optional *type* argument returns the original type of *value*. See "Iterating over a Dynamic Entity with %GetNext()" for details and examples.

```
method getNext(Output key, Output value, Output type...) as %Integer
```

parameters:

- *key* — returns the object key or array index of the element at the current position
- *value* — returns the value of the element at the current position.
- *type* — (optional) returns a string containing a **%GetTypeOf()** return value representing the original type of *value*. When this third argument variable is present it changes some of the conversion rules for converting the element into an ObjectScript value (see %Iterator.Object.**%GetNext()** and %Iterator.Array.**%GetNext()** for detailed conversion rules).

see also: **%GetIterator()**

class reference: %Iterator.Object.**%GetNext()** and %Iterator.Array.**%GetNext()**

%GetTypeOf()

Given a valid object key or array index, returns a string indicating the datatype of the value. See "Working with Datatypes" for details and examples.

```
method %GetTypeOf(key) as %String
```

parameters:

- *key* — the object key or array index of the value to be tested.

return values:

One of the following strings will be returned:

- `"null"` — a JSON null
- `"boolean"` — a zero ("false") or non-zero ("true") numeric value
- `"number"` — any canonical numeric value
- `"oref"` — a reference to another object
- `"object"` — a nested object
- `"array"` — a nested array
- `"string"` — a standard text string
- `"unassigned"` — the property or element exists, but does not have an assigned value

see also: **%IsDefined()**

class reference: %DynamicAbstractObject.**%GetTypeOf()**

%IsDefined()

Tests if the item specified by *key* is defined within an object. Returns false if the item is unassigned or does not exist. See "Using %IsDefined() to Test for Valid Values" for details and examples.

```
method %IsDefined(key) as %Boolean
```

parameters:

- *key* — the object key or array index of the item to be tested. An array index must be passed as a canonical integer value. Array indexes begin at position 0.

see also: Resolving Null, Empty String, and Unassigned Values

class reference: %DynamicObject.**%IsDefined()** and %DynamicArray.**%IsDefined()**

%Pop()

Returns the value of the last member of the array. The value is then removed from the array. If the array is already empty, the method returns the empty string, `""`. See "Using %Push and %Pop with Dynamic Arrays" for details and examples.

```
method %Pop() as %RawString
```

see also: **%Push()**, **%Get()**, **%Remove()**, Resolving Null, Empty String, and Unassigned Values

class reference: %DynamicArray.**%Pop()**

%Push()

Append a new value to the end of the current array, increasing the length of the array. Returns an oref pointing to the current modified array so that calls to **%Push()** can be chained. See "Using %Push and %Pop with Dynamic Arrays" for details and examples.

```
method %Push(value, type) as %DynamicAbstractObject
```

parameters:

- *value* — value to be assigned to the new array element.
- *type* — (Optional) string indicating the datatype of *value* (see %DynamicArray.**%Push()** for details on how values are converted). The following strings may be used:
  - `"null"` — a JSON `null`. The *value* argument must be `""` (empty string).
  - `"boolean"` — zero/nonzero becomes JSON *false*/*true*
  - `"number"` — convert *value* to a canonical numeric value
  - `"string"` — convert *value* to a text string
  - `"string>base64"` — convert to text string then encode into base64
  - `"string<base64"` — convert to text string then decode from base64
  - `"stream"` — %Stream contents converted to text string
  - `"stream>base64"` — %Stream contents are encoded into base64 string
  - `"stream<base64"` — %Stream is decoded from base64 into byte string

NOTE: The optional *type* parameter cannot be used if the specified *value* is an object or an oref. For example, if the specified *value* is a dynamic entity, an error will be thrown no matter what value you specify for *type*. See "Overriding a Default Datatype with %Set() or %Push()" for more information.

see also: **%Pop()**, **%Set()**, Method Chaining

class reference: %DynamicArray.**%Push()**

%Remove()

Removes the specified element from a dynamic object or array, and returns the value of the removed element. If the value of the element is an embedded dynamic object or array, all subordinate nodes are removed as well. In a dynamic array, all elements following the removed element will have their subscript position decremented by 1. See "Using %Set(), %Get(), and %Remove()" for details and examples.

```
method %Remove(key) as %DynamicAbstractObject
```

parameters:

- *key* — the object key or array index of the element you wish to remove. An array index must be passed as a canonical integer value. Array indexes begin at position 0.

see also: **%Set()**, **%Get()**, **%Pop()**

class reference: %DynamicObject.**%Remove()** and %DynamicArray.**%Remove()**

%Set()

Create a new value or update an existing value. Returns a reference to the modified array, allowing calls to **%Set()** to be nested. See "Using %Set(), %Get(), and %Remove()" for details and examples.

```
method %Set(key, value, type) as %DynamicAbstractObject
```

parameters:

- *key* — object key or array index of the value you wish to create or update. An array index must be passed as a canonical integer value. Array indexes begin at position 0.
- *value* — new value with which to update the previous value or create a new value.
- *type* — (Optional) string indicating the datatype of *value* (see %DynamicObject.**%Set()** and %DynamicArray.**%Set()** for details on how values are converted). The following strings may be used:
  - `"null"` — a JSON `null`. The *value* argument must be `""` (empty string).
  - `"boolean"` — a JSON `false` (*value* argument must be `0`) or `true` (*value* argument must be `1`).
  - `"number"` — convert *value* to a canonical numeric value
  - `"string"` — convert *value* to a text string
  - `"string>base64"` — convert to text string then encode into base64
  - `"string<base64"` — convert to text string then decode from base64
  - `"stream"` — %Stream contents converted to text string
  - `"stream>base64"` — %Stream contents are encoded into base64 string
  - `"stream<base64"` — %Stream is decoded from base64 into byte string

NOTE: The optional *type* parameter cannot be used if the specified *value* is an object or an oref. For example, if the specified *value* is a dynamic entity, an error will be thrown no matter what value you specify for *type*. See "Overriding a Default Datatype with %Set() or %Push()" for more information.

see also: **%Get()**, **%Remove()**, **%Push()**, Method Chaining

class reference: %DynamicObject.**%Set()** and %DynamicArray.**%Set()**

%Size()

Returns an integer showing the size of a dynamic object or array. In the case of an array, the size includes unassigned entries within the array. In the case of an object, the size only includes elements that have assigned values. See "Sparse Array Iteration with %Size()" for details and examples.

```
method %Size() as %Integer
```

see also: **%GetNext()**

class reference: %DynamicAbstractObject.**%Size()**

%ToJSON()

Converts an instance of %DynamicAbstractObject. into a JSON string. If **%ToJSON()** doesn't return an error, it will always return a string containing at least two characters ([] for an array or {} for an object). See "Converting Dynamic Entities to and from JSON" for details and examples.

```
method %ToJSON(outstrm As %Stream.Object) as %String
```

parameters:

- *outstrm* — optional. There are a number of possibilities:
  - If *outstrm* is not specified and the method is called via DO, the JSON string is written to the current output device.
  - If *outstrm* is not specified and the method is called as an expression, the JSON string becomes the value of the expression.
  - If *outstrm* is specified as an instance of %Stream.Object, the JSON string will be written to the stream (see "Serializing Large Dynamic Entities to Streams" for details and examples).
  - If *outstrm* is an object but is not an instance of %Stream.Object then an exception will be thrown.
  - If *outstrm* is not an object and is not null then it is presumed to be a fully qualified file specification (the full path to the file must be defined). The file is linked to a newly created %Stream.FileCharacter stream, the JSON string is written to the stream, and the stream is saved to the file on completion.

see also: **%FromJSON()**, **%FromJSONFile()**

class reference: %DynamicAbstractObject.**%ToJSON()**