JSON の使用

Version 2023.1
2024-01-02

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

JSON の使用
InterSystems IRIS Data Platform　Version 2023.1　2024-01-02
Copyright © 2024 InterSystems Corporation
All rights reserved.

InterSystems®, HealthShare Care Community®, HealthShare Unified Care Record®, IntegratedML®, InterSystems Caché®, InterSystems Ensemble® InterSystems HealthShare®, InterSystems IRIS®, および TrakCare は、InterSystems Corporation の登録商標です。HealthShare® CMS Solution Pack™ HealthShare® Health Connect Cloud™, InterSystems IRIS for Health™, InterSystems Supply Chain Orchestrator™, および InterSystems TotalView™ For Asset Management は、InterSystems Corporation の商標です。TrakCare は、オーストラリアおよび EU における登録商標です。

ここで使われている他の全てのブランドまたは製品名は、各社および各組織の商標または登録商標です。

このドキュメントは、インターシステムズ社(住所：One Memorial Drive, Cambridge, MA 02142)あるいはその子会社が所有する企業秘密および秘密情報を含んでおり、インターシステムズ社の製品を稼動および維持するためにのみ提供される。この発行物のいかなる部分も他の目的のために使用してはならない。また、インターシステムズ社の書面による事前の同意がない限り、本発行物を、いかなる形式、いかなる手段で、その全てまたは一部を、再発行、複製、開示、送付、検索可能なシステムへの保存、あるいは人またはコンピュータ言語への翻訳はしてはならない。

かかるプログラムと関連ドキュメントについて書かれているインターシステムズ社の標準ライセンス契約に記載されている範囲を除き、ここに記載された本ドキュメントとソフトウェアプログラムの複製、使用、廃棄は禁じられている。インターシステムズ社は、ソフトウェアライセンス契約に記載されている事項以外にかかるソフトウェアプログラムに関する説明と保証をするものではない。さらに、かかるソフトウェアに関する、あるいはかかるソフトウェアの使用から起こるいかなる損失、損害に対するインターシステムズ社の責任は、ソフトウェアライセンス契約にある事項に制限される。

前述は、そのコンピュータソフトウェアの使用およびそれによって起こるインターシステムズ社の責任の範囲、制限に関する一般的な概略である。完全な参照情報は、インターシステムズ社の標準ライセンス契約に記され、そのコピーは要望によって入手することができる。

インターシステムズ社は、本ドキュメントにある誤りに対する責任を放棄する。また、インターシステムズ社は、独自の裁量にて事前通知なしに、本ドキュメントに記載された製品および実行に対する代替と変更を行う権利を有する。

インターシステムズ社の製品に関するサポートやご質問は、以下にお問い合わせください：

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

目次

1

ObjectScript での JSON の使用

InterSystems ObjectScript 構文は、JSON (https://json.org/) を統合的にサポートしています。高速でシンプルかつ強力な一連の機能を使用して、オブジェクトやテーブルと同じくらい簡単に JSON のデータ構造を扱うことができます。

- JSON 用の ObjectScript 構文では、メソッド呼び出しの代わりに標準の ObjectScript 代入文を使用して、実行時にダイナミック・エンティティを作成および変更できます。オブジェクト・プロパティと配列要素の値は、JSON の文字列リテラルまたは ObjectScript のダイナミック式として指定できます。
- 2 つのクラスである %Library.DynamicObject と %Library.DynamicArray を使用して、標準の JSON データ構造を簡単かつ効率的にカプセル化して操作できます。これらのクラスのインスタンスは、ダイナミック・エンティティと呼ばれます。
- ダイナミック・エンティティには、JSON のシリアル化 (ダイナミック・エンティティとキャノニック形式 JSON フォーマット間の変換)、反復処理、データ型の指定、作成/読み取り/更新/削除の操作などの有用な機能を実行するためのメソッドが含まれています。

このドキュメントで取り上げる内容の詳細なリストは、"目次" を参照してください。

1.1 JSON の機能の実用例

ここでは、ObjectScript で利用可能な JSON の機能の使用例をいくつか紹介します。

実行時のダイナミック・エンティティの作成と操作

実行時にダイナミック・エンティティを作成して、それらのダイナミック・エンティティの任意のスキーマを定義できます。

```
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
```

リテラル JSON コンストラクタを使用したダイナミック・エンティティの作成

リテラル JSON 文字列を割り当てることで、ダイナミック・エンティティを作成することもできます。リテラル JSON コンストラクタの {} と [] は、%New() コンストラクタの代わりに使用できます。例えば、`set x=##class(%DynamicArray).%New()` の代わりに `set x=[]` を使用して動的配列を作成できます。%New()

とは異なり、リテラル JSON コンストラクタは、プロパティや要素を指定する JSON リテラル文字列を取ることもできます。したがって、以下の単純な代入文を使用して、上記の例の dynObject1 とまったく同じオブジェクトを作成できます。

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

この例では、コンストラクタごとに 1 つの文を使用していますが、配列コンストラクタを同じくらい簡単にオブジェクト・コンストラクタ内で入れ子にすることができます。

dynObject1 と dynObject2 がまったく同じであることを示すために、これらのオブジェクトを、%ToJSON() メソッドによって返されたシリアル化された JSON 文字列として表示できます。

```
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]}
```

ダイナミック式を使用した値の定義

リテラル・コンストラクタの {} と [] で囲まれたテキストでは、有効な JSON 構文を使用する必要がありますが、1 つだけ例外があります。要素またはプロパティの値として、JSON リテラルの代わりに括弧で囲まれた式を使用できます。この ObjectScript のダイナミック式 (set 文の右側と同等) は、実行時に評価されて有効な JSON 値に変換されます。次の例のダイナミック式には、$ZDATE 関数の呼び出しが含まれています。

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

評価されたダイナミック式の値は、dynObject.Date を取得したときに表示されます。

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

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

(これらのトピックの詳細は、“ダイナミック式とドット構文の使用”を参照)。

ダイナミック・エンティティとキャノニック形式の JSON 文字列の間の変換

ダイナミック・エンティティで提供されているシリアル化メソッドを使用すると、それらのダイナミック・エンティティを JSON 文字列に変換したり、その逆方向に変換したりできます。以下の例では、リテラル・コンストラクタを使用してダイナミック・オブジェクトを作成し、オブジェクトの %ToJSON() メソッドを呼び出して、それを myJSONstring にシリアル化します。

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

このシリアル化された JSON オブジェクトは、その他の文字列と同様に格納および取得できます。クラス・メソッド %FromJSON() および %FromJSONFile() は、任意のソースから有効な JSON 文字列を取得して、ダイナミック・オブジェクトに変換できます。以下のコードでは、myJSONstring をダイナミック・オブジェクト myObject に非シリアル化し、%ToJSON() を使用して表示します。

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

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

(シリアル化の詳細は、“ダイナミック・エンティティと JSON の間の変換”を参照してください)。

ダイナミック・エンティティ・メソッドの連鎖

一部のダイナミック・エンティティ・メソッドは連鎖させることができます。この例では、2 つの要素を持つ動的配列を作成してから、%Push() メソッドを連鎖させて、この配列の末尾に 3 つの要素を追加します。最後の連鎖された %ToJSON() の呼び出しによって、シリアル化された文字列が表示されます。

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

(連鎖可能なメソッドの詳細は、“メソッドの連鎖”を参照してください)。

反復処理とデータ型の確認

ダイナミック・エンティティ・メソッドは、反復処理やデータ型の確認などの目的用にも提供されています。この例では 2 つの JSON 文字列を作成し、それらのうち 1 つを dynEntity に非シリアル化して (いずれか一方が機能する)、dynEntity の反復子を取得します。

```
   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()
```

while ループの反復ごとに、%GetNext() はプロパティ名または配列インデックスを key に返して、メンバ値を val に返します。%GetTypeOf() の返り値は、この値のデータ型を示す文字列です。

```
   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
```

(これらおよび関連するメソッドの詳細は、“反復処理とスパース配列”および“データ型を使用した作業”を参照)。

1.2 ダイナミック・エンティティ・メソッドの概要

ダイナミック・エンティティ・メソッドは、次のカテゴリにグループ分けできます。

作成、読み取り、更新、削除

%Set() は、既存のダイナミック・エンティティ・メンバ (プロパティまたは要素) の値を変更したり、新しいメンバを作成してそのメンバに値を割り当てたりできます。%Remove() は、既存のメンバを削除します。%Get() は、メンバの値を取得します。詳細は、“ダイナミック・エンティティの作成と変更”を参照してください。

反復処理とスパース配列

%GetIterator() は、ダイナミック・エンティティの各メンバを指すポインタが含まれた反復子を返します。%GetNext() は、反復子によって指定されたメンバのキーと値を返して、カーソルを次のメンバに進めます。%Size() は、メンバの数を返します (スパース配列内の未割り当て要素を含む)。%IsDefined() は、メンバに値が割り当てられているかどうかを判定します。詳細は、“反復処理とスパース配列”を参照してください。

スタック関数

%Push() は、動的配列の末尾に新しい要素を追加します。%Pop() は、配列の最終要素を削除して、その要素の値を返します。これらのメソッドをダイナミック・オブジェクトに対して使用することはできません。オブジェクトのプロパティは予測可能な順序で格納されていないからです。詳細は、“動的配列での %Push と %Pop の使用”を参照してください。

JSON のシリアル化と非シリアル化

%FromJSON() は任意の JSON 文字列をダイナミック・エンティティに変換し、%FromJSONFile() は、ファイルに格納された JSON 文字列をダイナミック・エンティティに変換します。%ToJSON() は、ダイナミック・エンティティをキャノニック形式の JSON 文字列にシリアル化します。詳細は、“ダイナミック・エンティティと JSON の間の変換”を参照してください。

データ型の情報

%GetTypeOf() は、指定されたメンバ値のデータ型を示す文字列を返します。%Set() と %Push() では、値のデータ型を明示的に指定するためのオプションの 3 つ目の引数が用意されています。詳細は、“データ型を使用した作業”を参照してください。

各メソッドの説明と詳細情報へのリンクは、“ダイナミック・エンティティ・メソッドのクイック・リファレンス”を参照してください。

2

ダイナミック・エンティティの作成と変更

この章では、ダイナミック・エンティティが機能する仕組みについての基本的な情報を提供します。以下の項目について説明します。

- JSON リテラル・コンストラクタの使用
- ダイナミック式とドット構文の使用
- %Set()、%Get()、および %Remove() の使用
- メソッドの連鎖
- エラー処理
- ダイナミック・エンティティと JSON の間の変換
  - 大きいダイナミック・エンティティからストリームへのシリアル化

2.1 JSON リテラル・コンストラクタの使用

ダイナミック・エンティティは、**%DynamicObject** または **%DynamicArray** のインスタンスで、JSON のデータ操作を ObjectScript アプリケーションにシームレスに統合するように設計されています。標準の %New() メソッドを使用してこれらのクラスのインスタンスを作成できますが、ダイナミック・エンティティは、これよりはるかに柔軟で直感的な一連のコンストラクタをサポートしています。JSON リテラル・コンストラクタを使用すると、JSON 文字列を変数に直接代入することでダイナミック・エンティティを作成できます。例えば、以下のコードは **%DynamicObject** と **%DynamicArray** の空のインスタンスを作成します。

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

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

%New() コンストラクタとは異なり、リテラル・コンストラクタの {} と [] は JSON フォーマットの文字列を引数として受け取ることができます。例えば、以下のコードは prop1 という名前のプロパティを持つダイナミック・オブジェクトを作成します。

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

```
a string value
```

実際には、JSON リテラル・コンストラクタの {} と [] を使用して、任意の有効な JSON 配列構造またはオブジェクト構造を指定できます。簡単に言うと、有効な JSON リテラル文字列はすべて、評価結果がダイナミック・エンティティとなる有効な ObjectScript 式でもあります。

注釈 **JSON プロパティ名は常に引用符で囲む必要があります。**

JSON 言語仕様 (https://json.org/ を参照) は Javascript Object Notation のサブセットであり、この言語仕様では一部の領域でより厳格な規則が適用されます。重要な相違点は、JSON 仕様ではすべてのプロパティ名を二重引用符で囲む必要があることです。これに対して、JavaScript 構文では多くのケースで引用符なしの名前が許可されます。

ダイナミック・エンティティには、JSON 文字列内のそれぞれのオブジェクト・プロパティまたは配列要素の正確な表現が格納されます。すべてのダイナミック・エンティティは %ToJSON() メソッドを使用して、格納されたデータを JSON 文字列として返すことができます。リテラル文字列への変換時やリテラル文字列からの変換時に、データが失われたり壊れたりすることはありません。次の例では、動的配列を作成してから %ToJSON() を呼び出して、格納されているデータを表す新しい JSON 文字列を構成して返します。

```
   set dynamicArray = [[1,2,3],{"A":33,"a":"lower case"},1.2345678901234567890123 4,true,false,null,0,1,""]
   write dynamicArray.%ToJSON()
[[1,2,3],{"A":33,"a":"lower case"},1.23456789012345678901234,true,false,null,0,1,""]
```

この動的配列は、いくつかの重要な値を格納して返しました。

- 最初の 2 つの要素は、入れ子になった配列と入れ子になったオブジェクトです。JSON 構文では、配列とオブジェクトの構造を任意の深さまで入れ子にすることができます。
- プロパティ名は、大文字と小文字を区別します。この入れ子になったオブジェクトは、"A" と "a" という名前の 2 つの異なるプロパティを持っています。
- 3 つ目の値は、非常に高精度な小数です。この値が標準の浮動小数点数として格納されていた場合は、この値の端数が切り捨てられていましたが、この動的配列には元の値の正確な表現が保持されています。
- 最後の 6 つの要素には、JSON データ型の値である true、false、および null と、対応する ObjectScript 値である 0、1、および "" が格納されています。この場合でも、ダイナミック・エンティティには各値の正確な表現が保持されています。

2.2 ダイナミック式とドット構文の使用

値が JSON で格納される方法と、これらの値が ObjectScript で表現される方法の間には、大きな違いがあります。ObjectScript の値を使用しようとするたびに、その値を JSON 構文との間で変換する必要がある場合は、JSON のデータ格納はあまり便利ではありません。このため、ダイナミック・エンティティはこの変換プロセスを透過的なものにするように設計されています。JSON 構文における ObjectScript の値の表現について心配することなく、ObjectScript の値をいつでも格納および取得できます。

この規則は、リテラル JSON コンストラクタにも当てはまります。これまでに紹介したすべての例は全面的に JSON 構文に従っていましたが、リテラル・コンストラクタはダイナミック式で定義された値を受け取ることもできます。ダイナミック式は単に、括弧で囲まれた ObjectScript 式です。

例えば、次の動的配列コンストラクタは 2 つの Unicode 文字を格納します。実行時に、このリテラル・コンストラクタは各要素を評価して、評価された値を格納します。1 つ目の要素は JSON 構文で定義されており、2 つ目の要素は ObjectScript 関数呼び出しですが、結果として得られる保存値はまったく同じです。

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

ObjectScript の式は、set 文の右側のコードと見なすことができます。オブジェクト参照ではなく値として評価される ObjectScript 式は、いずれも JSON リテラル文字列にシリアル化できます。以下の例では、$LIST 値 (オブジェクトではなく区切り文字列) をオブジェクト・プロパティ obj.list に格納します。次に array を作成し、obj.list の各リスト項目を別々の要素に抽出します。

```
   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"]
```

ダイナミック式を使用してプロパティ名を定義することはできません (ただし、プロパティ名をプログラムによって定義するための方法が用意されています。詳細は、“%Set()、%Get()、および %Remove() の使用” を参照してください)。

当然ながら、リテラル・コンストラクタはオブジェクト・プロパティや配列要素を操作するための唯一の方法ではありません。例えば、以下のコードは空のダイナミック・オブジェクトを作成して、標準のオブジェクト・ドット構文を使用して内容を定義します。

```
   set dynArray = []
   set dynArray."0" = "02"_"33"
   set dynArray."1" = {}
   set dynArray."1".foo = $CHAR(dynArray."0")
   write dynArray.%ToJSON()

[233,{"foo":"é"}]
```

この例では、リテラル・コンストラクタは空のダイナミック・エンティティを作成するためだけに使用されています。これらの代入文は、以下に示すいくつかの簡単な規則に従っています。

- 割り当てられる値は、標準の ObjectScript 式です(この例では、"02"_"33" は整数 233 と評価される ObjectScript 文字列式です)。
- 配列要素は配列インデックス番号によって指定されます。配列インデックス番号は、二重引用符で囲まれた数値リテラルである必要があります。動的配列は 0 から始まります。
- オブジェクト・プロパティはプロパティ名によって指定されます。プロパティ名は文字列リテラルですが、そのプロパティ名が有効なクラス・メンバ名である場合は、二重引用符を省略可能です。
- 指定されたエンティティ・メンバがまだ存在していない場合は、そのメンバに値を割り当てたときにそのメンバが作成されます。

前述のとおり、値は、JSON 構文でどのように表現されているかにかかわらず、常に ObjectScript フォーマットで格納および取得されます。以下の例では、ドット構文の使用時に留意すべきいくつかの事項を示しています。

ドット構文を使用したダイナミック・オブジェクト・プロパティの作成

この例では、リテラル・コンストラクタとドット構文を使用して、A、a、および C quote という名前のプロパティが含まれたダイナミック・オブジェクト dynObj を作成します。リテラル文字列内では、すべてのプロパティ名を引用符で囲む必要があります。set 文と write 文では、a および A というプロパティ名は引用符で囲む必要はありませんが、C quote は引用符で囲む必要があります。

```
   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\" "}
```

ダイナミック・オブジェクトは箇条書きリストであるため、値は必ずしもそれらが作成された順序で格納されるわけではありません。このことを示す例については、“%GetNext() を使用したダイナミック・エンティティの反復処理” を参照してください。

ドット構文を使用した動的配列要素の作成

動的配列は 0 から始まります。この例では、配列要素 3 に値を代入してから、要素 2 を定義します。要素は順序どおり定義される必要はないため、要素 2 を未定義のままにしておいてもかまいません。詳細は、“スパース配列と未割り当て値の理解”を参照してください。

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

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

最初の 2 つの要素は、JSON ブーリアン値である `true` および `false` として定義されて格納されましたが、これらに対応する ObjectScript ブーリアン値である整数 `1` および `0` として返されます。

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

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

格納されている値は常に ObjectScript フォーマットで返されるため、JSON の `true`、`false`、および `null` という値は、ObjectScript の `0`、`1`、および `""` (空の文字列) という値として返されます。ただし、元の JSON 値はダイナミック・エンティティ内に保持されるため、必要に応じて復元できます。格納されている値の元のデータ型を特定する方法については、“データ型を使用した作業”を参照してください。

注釈 **ドット構文は、非常に長いプロパティ名で使用しないでください。**

ダイナミック・オブジェクトのプロパティには任意の長さの名前を割り当てることができますが、ObjectScript では 181 文字以上のプロパティ名を使用できません。この制限を超えるダイナミック・オブジェクトのプロパティ名をドット構文で使用しようとすると、そのプロパティが存在しており、その名前が有効であるにもかかわらず、`<PROPERTY DOES NOT EXIST>` という誤解を招くエラー・メッセージが発行されます。このエラーを回避するには、任意の長さのプロパティ名を受け付ける %Set() メソッドと %Get() メソッドを使用します。

2.3 %Set()、%Get()、および %Remove() の使用

リテラル・コンストラクタとドット構文を使用すると、ダイナミック・エンティティ・メンバを作成して値を操作できますが、これらはすべての目的に十分に対応できるものではありません。ダイナミック・エンティティで提供されている %Set()、%Get()、および %Remove() メソッドを使用すると、作成、読み取り、更新、および削除の操作を完全にプログラム制御できます。

これらのメソッドの最も重要な利点の 1 つは、メンバ識別子 (プロパティ名や配列インデックス番号) がリテラルである必要がないことです。ObjectScript の変数と式を使用して、値と識別子の両方を指定できます。

%Set()、%Get()、および %Remove() を使用したプログラムによる値と識別子の指定

次の例では、リテラル・コンストラクタ {} を使用してオブジェクトを作成し、この新しいオブジェクトの %Set() メソッドを呼び出して、`100+n` という値を持つ `prop`n という名前の一連のプロパティを追加します。名前と値の両方が ObjectScript の式によって定義されます。

```
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}
```

同じ変数を %Get() と共に使用して、プロパティ値を取得できます。

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

101 102 103 104 105
```

%Remove() メソッドは、指定されたメンバをダイナミック・エンティティから削除し、その値を返します。この例では、5 つのプロパティのうち 3 つを削除し、戻り値を文字列 removedValues に結合します。`write` 文は、削除された値の文字列と dynObj の現在の内容を表示します。

```
   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}
```

注釈　これらの簡単な例では `for` ループが使用されていますが、通常の反復メソッドは %GetNext() です (このメソッドについては “%GetNext() を使用したダイナミック・エンティティの反復処理” で後述します)。

%Get() と %Remove() はいずれも指定されたメンバの ObjectScript 値を返しますが、埋め込みダイナミック・エンティティが返される方法には重要な違いがあります。

- %Get() は参照によって値を返します。戻り値は、プロパティまたは要素への OREF (オブジェクト参照) です。これには埋め込みエンティティへの参照が含まれています。
- %Remove() は、指定されたプロパティまたは要素を破棄 (メンバ OREF を無効化) しますが、以前に埋め込まれたエンティティを直接指す、有効な OREF を返します。

%Get() と %Remove() を使用した入れ子になったダイナミック・エンティティの取得

次の例では、dynObj.address プロパティの値はダイナミック・オブジェクトです。%Get() 文は、変数 addrPointer にプロパティへの参照 (プロパティ値ではなく) を格納します。この時点で、addrPointer を使用して埋め込みエンティティ address の road プロパティにアクセスできます。

```
   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
```

%Remove() 文はプロパティを破棄して、新しい OREF へのプロパティ値に返します。

```
   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
```

%Remove() の呼び出し後、以前に埋め込まれたダイナミック・オブジェクトへの有効な OREF が addrRemoved に記述されます。

```
   write addrRemoved.%ToJSON()

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

%Remove() メソッドを使用して、メンバを任意の順序で削除できます。これは、以下の例で示すように、オブジェクトの場合と配列の場合とで異なる意味を持ちます。

オブジェクト・プロパティの削除

オブジェクト・プロパティには固定された順序はありません。つまり、プロパティは任意の順序で破棄できますが、プロパティを削除して別のプロパティを追加すると、プロパティがシリアル化され返される順序も変更される可能性があります。以下の例では、ダイナミック・オブジェクトを作成し、%Set() の呼び出しを 3 回連続して行い、3 つのプロパティを定義します。

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

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

ここでは、%Remove() を呼び出してプロパティ PropB を破棄し、その後で新しいプロパティ PropD を追加します。結果のダイナミック・オブジェクトは、そのプロパティを作成順序でシリアル化しません。

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

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

これは、反復子のメソッド %GetNext() がプロパティを返す順序にも影響します。%GetNext() を使用する同様の例については、"%GetNext() を使用したダイナミック・エンティティの反復処理" を参照してください。

配列要素の削除

配列は 0 から始まる順序付きリストです。要素に対して %Remove() を呼び出すと、その要素のすべての後続要素の配列インデックス番号は 1 だけ小さくなります。以下の例では、**%Remove(1)** の呼び出しを 3 回連続して行い、毎回異なる要素を削除します。

```
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"]
```

通常、スタック操作は %Set() と %Remove() ではなく %Push() と %Pop() で実装されますが、%Pop() を **%Remove(0)** に置き換えるとキューを実装できます ("動的配列での %Push と %Pop の使用" を参照)。

%Remove() は、すべての配列と同じ方法で動作します。これには、未定義の値を持つ要素を格納する配列も含まれます。スパース配列で %Remove() がどのように機能するかを示す例は、"スパース配列と未割り当て値の理解" を参照してください。

2.3.1 プロパティ値としてのダイナミック・エンティティの割り当て

%Set() または %Push() を使用して、ダイナミック・エンティティを別のダイナミック・エンティティ内に入れ子にできます。例えば、ダイナミック・オブジェクトをプロパティ値または配列要素として割り当てることができます。この章の前述の例では、入れ子になったオブジェクトを取得する方法を示しました ("%Get() と %Remove() を使用した入れ子になったダイナミック・エンティティの取得" を参照)。以下の例では、入れ子になったオブジェクトを作成する 1 つの方法を示しています。

プロパティ値としてのダイナミック・エンティティの割り当て

この例では、myData という名前のプロパティが指定されたダイナミック・オブジェクトが作成されます。このプロパティには、値として別のダイナミック・オブジェクトが指定されています。

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

以下のコードでこのオブジェクトが作成されます。変数として %Set() 引数を指定する必要はありませんが、指定すると、実行時に任意の有効な名前または値を割り当てることができるようになります。

```
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()
```

このコードによって、以下のような出力が生成されます。

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

注釈 **オブジェクト値と一緒に type パラメータを使用しないでください**

%Set() メソッドには、いくつかの限定された状況で value 引数のデータ型を指定できるようにするオプションの type パラメータがあります（“%Set() または %Push() を使用したデフォルトデータ型のオーバーライド” を参照）。type パラメータは、value 引数がダイナミック・エンティティの場合には使用することができません。使用しようとすると、エラーがスローされます。

2.4 メソッドの連鎖

%Set() メソッドと %Push() メソッドは、これらのメソッドによって変更されたエンティティの参照を返します。返された参照を直ちに使用して、同じ式内で同じエンティティに対して別のメソッドを呼び出すことができます。

連鎖の開始元となるダイナミック・エンティティは、コンストラクタ ({} または []) であっても既存のエンティティであってもかまいません。%Set() メソッドと %Push() メソッドは連鎖可能な参照を返し、連鎖内のどこからでも呼び出し可能です。連鎖の最終要素は、そのエンティティで使用できるどのメソッドでもかまいません。

次の例では、単一の `write` 文で、%FromJSON()、%Set()、%Push()、および %ToJSON() の連鎖呼び出しを使用して、動的配列を作成、変更、および表示します。

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

%FromJSON() は、呼び出し元エンティティの変更版を返さないため、連鎖の最初のメソッド呼び出しとしてのみ有用です。代わりに、このメソッドは呼び出し元エンティティを単に無視して、JSON 文字列から非シリアル化されたまったく新しいインスタンスを返します。詳細は、“ダイナミック・エンティティと JSON の間の変換” を参照してください。

%Get()、%Pop()、%GetNext()、または %Remove() を使用して入れ子になったエンティティを取得することによって、連鎖を開始することもできます。

2.5 エラー処理

ダイナミック・エンティティは、エラーの場合は %Status 値を返す代わりに例外をスローします。次の例では、スローされた例外に含まれている情報から、メソッド引数の 2 つ目の文字が無効であると判断できます。

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

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

動的データを扱う際は常に、一部のデータは期待に反すると想定することが推奨されます。ダイナミック・オブジェクトを利用するコードはすべて、いずれかのレベルで TRY-CATCH ブロックで囲む必要があります（"ObjectScript の使用法" の "TRY-CATCH メカニズム" を参照してください）。以下に例を示します。

```
   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 ダイナミック・エンティティと JSON の間の変換

%ToJSON() メソッドを使用してダイナミック・エンティティをシリアル化（JSON 文字列に変換）でき、%FromJSON() メソッドと %FromJSONFile() メソッドを使用して非シリアル化（JSON をダイナミック・エンティティに変換）できます。

ダイナミック・エンティティから JSON へのシリアル化

次の例では、ダイナミック・オブジェクトを作成して変更してから、%ToJSON() を使用してこのダイナミック・オブジェクトをシリアル化して、結果として得られる文字列を表示します。

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

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

動的配列は同じ方法でシリアル化されます。

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

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

これらの両方の例ではメソッドの連鎖を使用しています（本章で前出の "メソッドの連鎖" を参照してください）。

JSON からダイナミック・オブジェクトへの非シリアル化

%FromJSON() メソッドは、JSON 文字列をダイナミック・エンティティに変換します。次の例では、動的配列を作成して、この動的配列を jstring という文字列にシリアル化します。%FromJSON() を呼び出すことによって、jstring が newArray という名前の新しいダイナミック・エンティティに非シリアル化された後、このダイナミック・エンティティが変更されて表示されます。

```
   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"]
```

この例では、返される値は動的配列ですが、%FromJSON() がダイナミック・オブジェクト・コンストラクタ ({}) から呼び出されることに注目してください。%FromJSON() は **%DynamicAbstractObject** のクラス・メソッドであるため、任意のダイナミック・エンティティまたはコンストラクタから呼び出すことができます。

`.json` ファイルに格納した JSON データがある場合は、%FromJSON() メソッドではなく、%FromJSONFile() メソッドを使用してそのデータを非シリアル化できます。

%ToJSON() と %FromJSON() を使用した複製

%FromJSON() を呼び出すたびに新しいダイナミック・エンティティが作成されるため、このメソッドを使用して、既存のエンティティを複製したり一連の同一エンティティを初期化したりできます。

次の例では、dynObj.address プロパティの値はダイナミック・オブジェクトです。このプロパティは変数 addrPointer によって参照され、このプロパティの値は、%FromJSON() を呼び出して新しいダイナミック・オブジェクト addrClone を作成することによって複製されます。

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

変数 addrPointer はプロパティ dynObj.address の単なる参照ですが、addrClone は、元の値に影響を与えることなく変更可能な **%DynamicObject** の独立したインスタンスです。

```
   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."}
```

`.json` ファイルに格納した JSON データがある場合は、%FromJSON() メソッドではなく、%FromJSONFile() メソッドを使用してそのデータをクローン化できます。

2.6.1 大きいダイナミック・エンティティからストリームへのシリアル化

ダイナミック・エンティティが非常に大きい場合、%ToJSON() の出力が文字列の最大許容長を超える可能性があります (″サーバ側プログラミングの入門ガイド″ の “文字列長の制限” を参照)。このセクションの例では、longStr という名前の最大長の文字列を使用しています。以下のコード例は、longStr の生成方法を示しています。

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

Maximum string length = 3641144
```

式で %ToJSON() の返り値が使用されるたびに、プログラム・スタック上にその文字列が構築されます (この文字列には文字列長の制限が適用されます)。例えば、`write dyn.%ToJSON()` などの読み取り/書き込み文や、`set x=dyn.%ToJSON()` などの代入文は、その文字列をスタック上に配置しようとします。次の例では、longStr の 2 つのコ

ピーを動的配列に追加して、このシリアル化された文字列を変数に代入しようとしており、その結果として ObjectScript は <MAXSTRING> エラーを返しています。

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

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

この問題の一般的な解決策は、返り値を実際に調べることなく、DO コマンド内の参照によって %ToJSON() の出力を渡すことです。出力は現在のデバイスに直接書き込まれるため、出力の長さに制限はありません。次の例では、デバイスはストリームです。

ファイル・ストリームへの書き込み

この例では、ダイナミック・オブジェクト longObject をファイルに書き込んでから、このダイナミック・オブジェクトを取得します。変数 longStr は、このセクションの冒頭で定義した値です。

```
   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.
```

この解決策を使用して、他のストリームから入力を読み取ることもできます。

グローバル文字ストリームの読み取りと書き込み

この例では、2 つの大きいダイナミック・エンティティをシリアル化します (%ToJSON() はストリームごとに 1 つのエンティティしかシリアル化できないため、一時ストリームを使用しています)。標準のストリーム処理メソッドを使用して、各一時ストリームを別々の行としてストリーム bigLines に格納します。

```
   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)
```

後で、bigLines から各ダイナミック・エンティティを非シリアル化できます。

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

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

3

反復処理とスパース配列

ダイナミック・エンティティは、標準の反復メソッド %GetNext() を使用します。これは、オブジェクトと配列の両方で機能します。各要素を順番に指定して配列を反復処理することもできます (`for` ループまたは同様の構造体を使用) が、これには、値を含まない要素を持つスパース配列の知識が必要になる場合があります。%GetNext() は、これらの要素をスキップすることにより問題を回避するため、可能な限り、優先される反復メソッドになります。

この章では、各反復メソッドを使用するタイミングと方法ついて説明します。以下の項目について説明します。

- %GetNext() を使用したダイナミック・エンティティの反復処理
- スパース配列と未割り当て値の理解
  - %Size() を使用したスパース配列の反復処理
  - %IsDefined() を使用した有効値の有無の確認
- 動的配列での %Push と %Pop の使用

3.1 %GetNext() を使用したダイナミック・エンティティの反復処理

すべてのダイナミック・エンティティで提供されている %GetIterator() メソッドは、そのダイナミック・オブジェクトまたは動的配列のメンバを指すポインタが含まれた **%Iterator** (**%Iterator.Object** または **%Iterator.Array**) のインスタンスを返します。**%Iterator** オブジェクトは、各メンバのキーと値を取得するための %GetNext() メソッドを提供します。

%GetNext() メソッドを呼び出すたびに、反復子カーソルが進められて、カーソルの位置が有効なメンバである場合は `1` (true) が返され、カーソルが最後のメンバを越えている場合は `0` (false) が返されます。メンバの名前またはインデックス番号が 1 つ目の出力引数内に返されて、値が 2 つ目の出力引数内に返されます。以下に例を示します。

```
   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/
```

反復子カーソルは一方向のみに移動するため、前のメンバに戻ったり、配列を逆順に反復処理したりすることはできません。

スパース配列を反復処理する際は、反復子は値が割り当てられていない要素をスキップします。オブジェクトを反復処理する際は、プロパティは必ずしも予測可能な順序で返されません。以下の例では、配列の反復処理とオブジェクトの反復処理の間に存在するこれらの相違点を示しています。

配列の反復処理

この例では、スパース配列を作成します。配列は 0 から始まり、6 つの要素が指定されていますが、要素 0、1、および 5 のみに値が割り当てられています。JSON 文字列に表示されている null 要素は、未割り当て値の単なるプレースホルダです。

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

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

%GetNext() は、値が割り当てられている 3 つの要素のみを返して、すべての未割り当て要素をスキップします。

```
   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
```

スパース配列の詳細は、次のセクション (“スパース配列と未割り当て値の理解”) を参照してください。

オブジェクトの反復処理

オブジェクト・プロパティには固定された順序はありません。つまり、未割り当て値を作成しなくても、プロパティを任意の順序で作成したり破棄したりすることはできますが、オブジェクトを変更すると、%GetNext() によってプロパティが返される順序も変更される可能性があります。以下の例では、3 つのプロパティを持つオブジェクトを作成し、%Remove() を呼び出して 1 つのプロパティを破棄し、別のプロパティを追加します。

```
   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}
```

オブジェクトを反復処理する際は、%GetNext() はアイテムをそれらが作成された順序で返しません。

```
   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 スパース配列と未割り当て値の理解

動的配列は、スパース配列であってもかまいません (すなわち、すべての配列要素に値が保持されていなくてもかまいません)。例えば、動的配列に 0 ～ 99 の要素がまだ含まれていない場合でも、この配列の要素 100 に値を割り当てることができます。メモリ領域は、要素 100 の値のみに割り当てられます。要素 0 ～ 99 は未割り当てです。つまり、0 ～ 99 は有効な要素識別子ですが、メモリ内のどの値も指していません。%Size() メソッドは 101 という配列サイズを返しますが、%GetNext() メソッドは未割り当て要素をスキップして、要素 100 の値のみを返します。

次の例では、要素 8 と 11 に新しい値を割り当てることで、スパース配列を作成します。

```
   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"]
```

要素 6、7、9、および 10 には値が割り当てられていません。また、これらの要素はメモリ領域を使用しませんが、JSON 文字列には null 値で表されます。これは JSON が未定義の値をサポートしていないためです。

スパース配列での %Remove の使用

%Remove() メソッドは、未割り当ての要素を他の要素と同様に扱います。未割り当ての値のみで構成される配列を持つことは可能です。次の例では、スパース配列を作成して未割り当ての要素 0 を削除します。次に、要素 7 を削除します。これは現在、値を含む唯一の要素です。

```
   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]
```

%Remove() の使用例については、“%Set()、%Get()、および %Remove() の使用”を参照してください。

注釈 **JSON は null 値と未割り当て値の区別を保持できません**

ダイナミック・エンティティには、null 値と unassigned 値を区別することを可能にするメタデータが含まれています。JSON では、独立した undefined データ型が指定されていないため、ダイナミック・エンティティが JSON 文字列にシリアル化される際に、この区別を保持するためのキャノニック形式の方法はありません。シリアル化されたデータに追加の null 値が不要な場合は、シリアル化する前に未割り当ての要素を削除するか(“%IsDefined() を使用した有効値の有無の確認”を参照)、アプリケーションに依存する何らかの手段を使用して、この区別をメタデータとして記録する必要があります。

3.2.1 %Size() を使用したスパース配列の反復処理

%Size() メソッドは、ダイナミック・エンティティ内のプロパティまたは要素の数を返します。以下に例を示します。

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

```
Number of properties: 3
```

スパース配列では、この数には、値が割り当てられていない要素が含まれます (次の例を参照)。この例で作成される配列には 6 つの要素がありますが、要素 0、1、および 5 のみに値が割り当てられています。JSON 文字列に表示されている null 要素は、未割り当て値の単なるプレースホルダです。

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

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

要素 2、3、および 4 には値が割り当てられていませんが、有効な配列要素として扱われます。動的配列は 0 から始まるため、最終要素のインデックス番号は常に %Size()-1 となります。次の例では、配列 test の 6 つの要素すべてを逆順に反復処理して、%Get() を使用してこれらの要素の値を返します。

```
   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/
```

%Get() メソッドは、%Size()-1 より大きい値の場合は "" (空の文字列) を返し、負の数の場合は例外をスローします。未割り当て値、空の文字列、および null 値を区別する方法については、“データ型を使用した作業”を参照してください。

注釈　ここで紹介した反復手法は、特別な目的に対してのみ有用です (配列内の未割り当て値の検出や、配列の逆順の反復処理など)。ほとんどの場合は、未割り当て要素をスキップする %GetNext() メソッドを使用してください。このメソッドは、ダイナミック・オブジェクトや動的配列に対して使用できます。詳細は、前のセクション (“%GetNext() を使用したダイナミック・エンティティの反復処理”) を参照してください。

3.2.2 %IsDefined() を使用した有効値の有無の確認

%IsDefined() メソッドは、指定されたプロパティ名や配列インデックス番号に値が存在するかどうかを判定します。このメソッドは、指定されたメンバが値を持つ場合は 1 (true) を返して、そのメンバが存在しない場合は 0 (false) を返します。このメソッドは、スパース配列内の値が割り当てられていない要素についても false を返します。

for ループを使用してスパース配列を反復処理する場合は、未割り当て値が検出されます。次の例では、最初の 3 つの要素が JSON の null、空の文字列、および未割り当て値である配列を作成します。for ループは、配列の末尾を越えて、配列インデックス 4 の要素をテストするように、意図的に設定されています。

```
   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() によって 0 が返される 2 つのケースは、要素 2 に値が割り当てられていない場合と、要素 4 が存在しない場合です。

ObjectScript は、JSON の null 値 (この例の要素 0 など) については "" (空の文字列) を返します。""、null、および未割り当て値の有無を判定する必要がある場合は、%IsDefined() ではなく %GetTypeOf() を使用してください (“NULL、空の文字列、および未割り当て値の解決”を参照してください)。

注釈　前のセクションで述べたとおり、いくつかの特殊な場合を除いて、反復処理のために for ループを使用しないでください。ほとんどの場合は、未割り当て値をスキップする %GetNext() メソッドを使用してください (“%GetNext() を使用したダイナミック・エンティティの反復処理”を参照してください)。

%IsDefined() メソッドは、オブジェクト・プロパティが存在するかどうかを判定するためにも使用できます。以下のコードは、3 つの文字列値を持つ動的配列 names を作成してから、最初の 2 つの文字列を使用して、prop1 および prop2 というプロパティを持つオブジェクト dynobj を作成します。

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

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

以下のコードは、%IsDefined() を使用して dynobj 内でプロパティ名として使用された文字列を特定します。

```
   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 動的配列での %Push と %Pop の使用

%Push() メソッドと %Pop() メソッドは、動的配列に対してのみ使用できます。これらは %Set() および %Remove() とまったく同じように機能しますが、配列の最後の要素を常に追加または削除する点が異なります。例えば、以下のコードは、どちらのメソッドのセットでも同じ結果が得られます (同じ文で %Set() または %Push() を複数回呼び出すことについて、詳細は、“メソッドの連鎖”を参照してください)。

```
   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]
```

%Push() と %Pop() はスタック操作用ですが、`%Pop()` の代わりに `%Remove(0)` を使用すると、キューを実装できます。

以下の例では、%Push() を使用して配列を作成し、%Pop() を使用して各要素を逆の順序で削除します。

%Push() および %Pop() を使用した配列の構築と分解

入れ子になった配列を含む配列を構築します。%Push() の最後の呼び出しは、オプションの type 引数を指定して、ブーリアン値を ObjectScript の `0` ではなく JSON の `false` として格納します (“%Set() または %Push() を使用したデフォルトデータ型のオーバーライド”を参照)。

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

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

入れ子になった配列のすべての要素を削除します。すべてのダイナミック・エンティティ・メソッドと同様に、%Pop() は JSON の `false` ではなく ObjectScript の `0` を返します。

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

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

空の入れ子になった配列を含む、メイン配列のすべての要素を削除します。

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

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

簡単にするために、これらの例ではハード・コード化された `for` ループを使用しています。配列の反復処理の実例は、“%Size() を使用したスパース配列の反復処理”を参照してください。

4

データ型を使用した作業

ObjectScript には、JSON の `true`、`false`、および `null` に相当する個別の固定値は用意されておらず、JSON には値が定義されていない配列要素という概念はありません。この章では、これらの不一致点について説明して、これらの不一致点に対処するために用意されているツールを紹介します。

- %GetTypeOf() を使用した値のデータ型の検出
- %Set() または %Push() を使用したデフォルトデータ型のオーバーライド
- JSON の NULL 値とブーリアン値の解決
- NULL、空の文字列、および未割り当て値の解決

4.1 %GetTypeOf() を使用した値のデータ型の検出

%GetTypeOf() メソッドを使用して、ダイナミック・エンティティ・メンバのデータ型を取得できます。ダイナミック・オブジェクトのプロパティや配列要素のデータ型は、以下のいずれでもかまいません。

- オブジェクトのデータ型 :
  - `array` ― 動的配列の参照
  - `object` ― ダイナミック・オブジェクトの参照
  - `oref` ― ダイナミック・エンティティではないオブジェクトの参照
- リテラル値 :
  - `number` ― キャノニック形式の数値
  - `string` ― 文字列リテラル、または評価結果が文字列リテラルになる式
- JSON リテラル :
  - `boolean` ― JSON リテラルの `true` または `false`
  - `null` ― JSON リテラルの `null`
- データ型なし :
  - `unassigned` ― そのプロパティまたは要素は存在しているが、値が割り当てられていない

オブジェクトでの %GetTypeOf の使用

このメソッドをオブジェクトに使用する場合、引数は、プロパティの名前になります。例えば、以下のようになります。

```
   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
```

配列での %GetTypeOf の使用

このメソッドを配列に使用する場合、引数は、要素のインデックスになります。次の例では、要素 2 に値が割り当てられていないスパース配列を調べます。この例では `for` ループを使用しています。%GetNext() は未割り当て要素をスキップするからです。

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

`array` または `object` および `oref` の区別

ダイナミック・エンティティのデータ型は `array` または `object` になります。ダイナミック・エンティティではない InterSystems IRIS オブジェクトのデータ型は `oref` になります。次の例では、オブジェクト dyn の各プロパティはこれら 3 つのデータ型のいずれかです。プロパティ `dynobject` はクラス %DynamicObject であり、プロパティ `dynarray` は %DynamicArray であり、プロパティ `streamobj` は %Stream.GlobalCharacter です。

```
   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 %Set() または %Push() を使用したデフォルトデータ型のオーバーライド

既定では、システムは自動的に、%Set() または %Push() の値引数をオブジェクトのデータ型 (`object`、`array`、または `oref`) または ObjectScript リテラルのデータ型 (`string` または `number`) として解釈します。JSON リテラルの `null`、`true`、または `false` を値として直接渡すことはできません。この引数は ObjectScript のリテラルまたは式として解釈されるからです。例えば、次のコードでは、値 `true` が変数名として解釈されるため、エラーがスローされます。

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

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

ObjectScript は、null の代わりに "" (空の文字列) を使用し、ブーリアン値 false の代わりに 0 を使用し、ブーリアン値 true の代わりにゼロ以外の数字を使用します。この問題に対処するために、%Set() と %Push() は、対象の値のデータ型を指定するためのオプションの 3 つ目の引数を取ります。この 3 つ目の引数には、JSON の boolean または null を使用できます。以下に例を示します。

```
   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}
```

対象の値を数値として解釈可能な場合は、3 つ目の引数には string または number を使用することもできます。

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

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

4.3 JSON の NULL 値とブーリアン値の解決

JSON 構文では、true、false、および null という値は、1、0、および "" (空の文字列) という値とは異なりますが、ObjectScript ではこのような区別はされません。JSON の値が要素またはプロパティから取得されると、それらの値は常に ObjectScript に対応した値にキャストされます。すなわち、JSON の true は常に 1 として返され、false は 0 として返され、null は "" として返します。ほとんどの場合は、これは望ましい結果となります。その返り値は、最初に JSON フォーマットから変換することなく、ObjectScript の式で使用できるからです。ダイナミック・エンティティの内部には JSON や ObjectScript の元の値が保持されるため、%GetTypeOf() を使用して必要に応じて実際のデータ型を特定できます。

次の例では、動的配列コンストラクタによって、JSON の true、false、および null の各値、数値と文字列のリテラル値、および ObjectScript のダイナミック式 (評価結果が ObjectScript のブーリアン値 1 および 0 になるもの) が指定されます。

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

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

上記からわかるように、コンストラクタで割り当てられた値は、結果として得られる動的配列内に保持されていたため、JSON 文字列としてシリアル化されたときに適切に表示されます。

次の例では、配列の値を取得して表示します。予想どおり、JSON の値 true、false、および null は ObjectScript に対応した値 1、0、および "" にキャストされます。

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

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

この例では %GetNext() を使用していますが、%Get()、%Pop()、またはドット構文を使用して値を取得した場合でも同じ結果が得られます。

必要に応じて、%GetTypeOf() メソッドを使用して値の元のデータ型を確認できます。以下に例を示します。

```
   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
```

注釈 **ダイナミック・オブジェクト内のデータ型**

この章では動的配列を主に扱っていますが、同じデータ型変換がダイナミック・オブジェクトの値にも適用されます。このセクションの例は、動的配列 test がダイナミック・オブジェクトに置き換わる場合でもまったく同じように動作します。

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

この行を除いて、サンプル・コードのどの部分も変更する必要はありません。このオブジェクト内のプロパティ名は、元の配列のインデックス番号に対応する数値文字列であるため、出力もまったく同じになります。

4.4 NULL、空の文字列、および未割り当て値の解決

JSON の `null` 値を要素やプロパティに割り当てることができますが、その値は常に `""` (ObjectScript の空の文字列) として返されます。未割り当て要素の値を取得しようとした場合にも、空の文字列が返されます。%GetTypeOf() を使用して、各ケースの実際のデータ型を特定できます。

この例では、JSON の `null` 値と空の文字列が含まれたスパース配列を調べます。配列要素 `2` には値が割り当てられていませんが、この要素は JSON 文字列内では `null` で表現されます。

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

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

ほとんどの場合は、%GetNext() を使用して配列値を取得しますが、この例では、`for` ループを使用して、%GetNext() を使用した場合はスキップされる未割り当て値を返します。最終要素のインデックス番号は `array.%Size()-1` ですが、ループ・カウンタは配列の末尾を越えて処理を続行するように意図的に設定されています。

```
   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
```

この例では、%Get() は以下の 4 つのケースで空の文字列を返します。

1. 要素 `0` は JSON の `null` 値であり、%GetTypeOf() によってデータ型 `null` として識別されます。
2. 要素 `1` は空の文字列であり、データ型 `string` として識別されます。
3. 要素 `2` は値を持っていないため、データ型 `unassigned` として識別されます。

4. 要素 3 が最後の配列要素ですが、この例では、存在しない要素 4 のデータ型を取得しようとします。この要素もデータ型 unassigned として識別されます。有効な配列インデックス番号は常に array.%Size() より小さい値になります。

注釈 null と unassigned の区別は、ダイナミック・エンティティが JSON 文字列にシリアル化されたときに保持されない ObjectScript メタデータです。すべての unassigned 要素は null 値としてシリアル化されます。詳細は、“スパース配列と未割り当て値の理解”を参照してください。

5
JSON アダプタの使用

JSON アダプタは、ObjectScript オブジェクト（登録、シリアル、または永続）を JSON テキストまたはダイナミック・エンティティにマップするための手段です。この章では、以下の項目について説明します。

- エクスポートとインポート — JSON 対応のオブジェクトを紹介し、**%JSON.Adaptor** のインポート・メソッドとエクスポート・メソッドを示します。
- パラメータを使用したマッピング — オブジェクト・プロパティを JSON フィールドに変換する方法を制御するプロパティ・パラメータについて説明します。
- XData マッピング・ブロックの使用 — 複数のパラメータ・マッピングを 1 つのクラスに適用する方法について説明します。
- JSON のフォーマット — **%JSON.Formatter** を使用して JSON 文字列をフォーマットする方法を示します。
- %JSON のクイック・リファレンス — この章で解説したそれぞれの %JSON クラス・メンバについて簡単に説明します。

5.1 エクスポートとインポート

JSON との間でシリアル化するクラスは、**%JSON.Adaptor** のサブクラスを作成する必要があります。これには以下のクラスが含まれます。

- %JSONExport() は、JSON 対応のクラスを JSON ドキュメントとしてシリアル化し、それを現在のデバイスに書き込みます。
- %JSONExportToStream() は、JSON 対応のクラスを JSON ドキュメントとしてシリアル化し、それをストリームに書き込みます。
- %JSONExportToString() は、JSON 対応のクラスを JSON ドキュメントとしてシリアル化し、それを文字列として返します。
- %JSONImport() は、文字列またはストリームとしての JSON か、**%DynamicAbstractObject** のサブクラスをインポートし、JSON 対応のクラスのインスタンスを返します。

これらのメソッドを示すために、このセクションの例では以下の 2 つのクラスを使用します。

JSON 対応のクラス Model.Event と Model.Location

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

および

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

ご覧のように、場所にリンクする永続イベント・クラスがあります。どちらのクラスも %JSON.Adaptor を継承します。これにより、オブジェクト・グラフを生成し、それを JSON 文字列として直接エクスポートすることができます。

JSON 文字列へのオブジェクトのエクスポート

```
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()
```

このコードは、以下の JSON 文字列を表示します。

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

%JSONExport() の代わりに %JSONExportToString() を使用することで、この JSON 文字列を変数に割り当てることができます。

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

最後に、%JSONImport() を使用して、このプロセスを逆にたどり、JSON 文字列を再びオブジェクトに変換することができます。この例は、前の例の文字列変数 jsonEvent を取り、それを再び Model.Event オブジェクトに変換します。

オブジェクトへの JSON 文字列のインポート

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

これにより、以下のような出力が表示されます。

```
Global Summit
Boston
```

インポートとエクスポートはどちらも、どのような入れ子構造でも機能します。

5.2 パラメータを使用したマッピング

対応するパラメータを設定することにより、個々のプロパティのマッピング・ロジックを指定できます (%XML.Adaptor を使い慣れている場合、これは同様のプロセスです)。

プロパティ・パラメータを指定することにより、Model.Event クラス (前のセクションで定義したもの) のマッピングを変更できます。

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

このマッピングによって、以下の 2 つの点が変更されます。

- プロパティ Name は、eventName という名前の JSON フィールドにマップされます。
- Location プロパティは、%JSONImport() では引き続き入力として使用されますが、%JSONExport() や他のエクスポート・メソッドでは無視されます。

前述の例では、変更されていない Model.Event クラスのインスタンスで %JSONExport() が呼び出され、以下の JSON 文字列が返されていました。

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

再マップされた Model.Event のインスタンスで (同じプロパティ値を使用して) %JSONExport() を呼び出すと、以下の文字列が返されます。

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

以下のようなさまざまなパラメータを使用して、マッピングを調整できます。

- %JSONFIELDNAME (プロパティのみ) では、JSON コンテンツでフィールド名として使用する文字列を設定します (既定では、値はプロパティ名です)。
- %JSONIGNOREINVALIDFIELD では、JSON 入力に含まれている予期しないフィールドの処理を制御します。
- %JSONIGNORENULL を使用すると、開発者は、文字列プロパティの空文字列の既定の処理をオーバーライドすることができます。
- %JSONINCLUDE (プロパティのみ) では、そのプロパティを JSON の出力または入力に含めるかどうかを指定します (有効な値は "inout" (既定値)、"outputonly"、"inputOnly"、または "none" です)。
- %JSONNULL では、文字列プロパティについて空の文字列を格納する方法を指定します。
- %JSONREFERENCE では、オブジェクト参照を JSON フィールドに投影する方法を指定します。オプションは "OBJECT" (既定値)、"ID"、"OID"、および "GUID" です。

詳細は、この章の後述のリファレンス・セクション“%JSON.Adaptor のクラス・パラメータとプロパティ・パラメータ”を参照してください。

5.3 XData マッピング・ブロックの使用

プロパティ・レベルでマッピング・パラメータを設定する代わりに、特殊な XData マッピング・ブロック内でマッピングを指定し、インポート・メソッドまたはエクスポート・メソッドを呼び出すときにそのマッピングを適用することができます。

以下のコードでは、前の 2 つのセクションで使用した Model.Event クラスの別のバージョンを定義します。このバージョンでは、プロパティ・パラメータを指定するのではなく、前のバージョンのプロパティと同じパラメータ設定を指定する OnlyLowercaseTopLevel という名前の XData マッピング・ブロックを定義します。

```
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>
  }
}
```

重要な違いとして、XData ブロックの JSON マッピングによって既定の動作が変更されるわけではなく、インポート・メソッドとエクスポート・メソッドのオプションの %mappingName パラメータでブロック名を指定することによって、マッピングを適用できるという点があります。以下に例を示します。

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

これにより、以下のような出力が表示されます。

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

この出力は、プロパティ定義でパラメータを指定した場合と同じです。

指定した名前の XData ブロックがない場合は、既定のマッピングが使用されます。この手法では、複数のマッピングを構成し、それぞれの呼び出しに必要なマッピングを個々に参照することができるため、さらに詳細な制御が可能になると同時に、マッピングの柔軟性と再利用性が向上します。

5.3.1 XData マッピング・ブロックの定義

JSON 対応のクラスでは、任意の数の追加マッピングを定義できます。それぞれのマッピングは、以下の形式の別個の XData ブロックで定義します。

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

{MappingName}、{ClassAttribute}、{Property Name}、および {PropertyAttribute} の定義は以下のとおりです。

- MappingName

  %JSONREFERENCE パラメータまたは Reference 属性で使用されるマッピングの名前。

- ClassAttribute

  マッピングのクラス・パラメータを指定します。以下のクラス属性を定義できます。

  - `Mapping` ― 適用する XData マッピング・ブロックの名前。
  - `IgnoreInvalidField` ― クラス・パラメータ %JSONIGNOREINVALIDFIELD を指定します。
  - `Null` ― クラス・パラメータ %JSONNULL を指定します。
  - `IgnoreNull` ― クラス・パラメータ %JSONIGNORENULL を指定します。
  - `Reference` ― クラス・パラメータ %JSONREFERENCE を指定します。

- PropertyName

  マッピングされるプロパティの名前。

- PropertyAttribute

  マッピングのプロパティ・パラメータを指定します。以下のプロパティ属性を定義できます。

  - `FieldName` ― プロパティ・パラメータ %JSONFIELDNAME を指定します (既定では、プロパティ名と同じ)。
  - `Include` ― プロパティ・パラメータ %JSONINCLUDE を指定します (有効な値は `"inout"` (既定値)、`"outputonly"`、`"inputOnly"`、または `"none"` です)。
  - `Mapping` ― オブジェクト・プロパティに適用するマッピング定義の名前。

- `Null` — クラス・パラメータ %JSONNULL をオーバーライドします。
- `IgnoreNull` — クラス・パラメータ %JSONIGNORENULL をオーバーライドします。
- `Reference` — クラス・パラメータ %JSONREFERENCE をオーバーライドします。

5.4 JSON のフォーマット

%JSON.Formatter は、非常に単純なインタフェースを備えたクラスです。このクラスを使用して、ダイナミック・オブジェクト、動的配列、および JSON 文字列をフォーマットして、人が読みやすい形式で表すことができます。すべてのメソッドがインスタンス・メソッドなので、常に、最初にインスタンスを取得します。

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

これを選択する理由は、行ターミネータとインデントに特定の文字 (例えば、空白とタブ。このセクションの末尾にあるプロパティ・リストを参照) を使用するようにフォーマッタを一度構成すれば、後で必要になったときにいつでも使用できるからです。

Format() メソッドは、ダイナミック・エンティティまたは JSON 文字列を取ります。以下は、ダイナミック・オブジェクトを使用した簡単な例です。

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

フォーマットされた結果の文字列が現在のデバイスに表示されます。

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

フォーマット・メソッドでは、現在のデバイス、文字列、またはストリームに出力を送信できます。

- Format() は、指定されたインデントを使用して JSON ドキュメントをフォーマットし、それを現在のデバイスに書き込みます。
- FormatToStream() は、指定されたインデントを使用して JSON ドキュメントをフォーマットし、それをストリームに書き込みます。
- FormatToString() は、指定されたインデントを使用して JSON ドキュメントをフォーマットし、それを文字列に書き込むか、JSON 対応のクラスを JSON ドキュメントとしてシリアル化し、それを文字列として返します。

さらに、以下のプロパティを使用して、インデントと改行を制御できます。

- Indent では、JSON 出力をインデントするかどうかを指定します。
- IndentChars では、それぞれのインデント・レベルに使用する文字シーケンスを指定します (既定では、レベルごとに 1 つのスペースが使用されます)。
- LineTerminator では、インデントするときにそれぞれの行を終了する文字シーケンスを指定します。

5.5 %JSON のクイック・リファレンス

このセクションでは、この章で解説した %JSON のメソッド、プロパティ、およびパラメータのクイック・リファレンスを提供します。最も包括的な最新情報については、クラス・リファレンスの "%JSON" を参照してください。

5.5.1 %JSON.Adaptor のメソッド

これらのメソッドを使用すると、JSON との間でシリアル化を行うことができます。詳細と例は、"エクスポートとインポート" を参照してください。

%JSONExport()

%JSON.Adaptor.%JSONExport() は、JSON 対応のクラスを JSON ドキュメントとしてシリアル化し、それを現在のデバイスに書き込みます。

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

パラメータ：

- %mappingName (オプション) ー Exportポートに使用するマッピングの名前。基本マッピングは "" で表され、これが既定値です。

%JSONExportToStream()

%JSON.Adaptor.%JSONExportToStream() は、JSON 対応のクラスを JSON ドキュメントとしてシリアル化し、それをストリームに書き込みます。

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

パラメータ：

- export ー シリアル化された JSON ドキュメントを格納するストリーム。
- %mappingName (オプション) ー エクスポートに使用するマッピングの名前。基本マッピングは "" で表され、これが既定値です。

%JSONExportToString()

%JSON.Adaptor.%JSONExportToString() は、JSON 対応のクラスを JSON ドキュメントとしてシリアル化し、それを文字列として返します。

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

パラメータ：

- export ー シリアル化された JSON ドキュメントを格納する文字列。
- %mappingName (オプション) ー エクスポートに使用するマッピングの名前。基本マッピングは "" で表され、これが既定値です。

%JSONImport()

%JSON.Adaptor.%JSONImport() は、JSON またはダイナミック・エンティティの入力をこのオブジェクトにインポートします。

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

パラメータ：

- input ー 文字列またはストリームとしての JSON か、**%DynamicAbstractObject** のサブクラス。
- %mappingName (オプション) ー インポートに使用するマッピングの名前。基本マッピングは "" で表され、これが既定値です。

%JSONNew()

%JSON.Adaptor.%JSONNew() は、JSON 対応のクラスのインスタンスを取得します。このクラスのインスタンスを返す前に、カスタム処理 (オブジェクト・インスタンスの初期化など) を実行するようにこのメソッドをオーバーライドできます。ただし、このメソッドをユーザ・コードから直接呼び出さないでください。

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

パラメータ：

- dynamicObject ー 新しいオブジェクトに割り当てられる値を持つダイナミック・エンティティ。
- containerOref (オプション) ー %JSONImport() から呼び出されたときの格納オブジェクト・インスタンス。

5.5.2 %JSON.Adaptor のクラス・パラメータとプロパティ・パラメータ

特に明記していない限り、パラメータは、クラスに対して指定することも、個々のプロパティに対して指定することもできます。クラス・パラメータとして使用する場合は、対応するプロパティ・パラメータの既定値を指定します。プロパティ・パラメータとして使用する場合は、既定値をオーバーライドする値を指定します。詳細と例は、“パラメータを使用したマッピング” を参照してください。

%JSONENABLED

プロパティ変換メソッドの生成を有効にします。

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

有効な値は以下のとおりです。

- `1` ー (既定値) JSON 対応メソッドが生成されます。
- `0` ー メソッド・ジェネレータは、実行可能なメソッドを生成しません。

%JSONFIELDNAME (プロパティのみ)

JSON コンテンツでフィールド名として使用する文字列を設定します。

```
parameter %JSONFIELDNAME
```

既定では、プロパティ名が使用されます。

%JSONIGNOREINVALIDFIELD

JSON 入力に含まれている予期しないフィールドの処理を制御します。

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

有効な値は以下のとおりです。

- `0` ー (既定値) 予期しないフィールドをエラーとして処理します。
- `1` ー 予期しないフィールドは無視されます。

%JSONIGNORENULL

文字列プロパティについて空の文字列を格納する方法を指定します。このパラメータは、真の文字列 (`XSDTYPE = "string"` および `JSONTYPE="string"` によって指定される) のみに適用されます。

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

有効な値は以下のとおりです。

- `0` ー (既定値) JSON 入力に含まれる空の文字列は `$char(0)` として格納され、`$char(0)` が文字列 `""` として JSON に書き込まれます。JSON 入力で指定されていないフィールドは常に `""` として格納され、`""` は常に、%JSONNULL パラメータに従って JSON に出力されます。
- `1` ー 空の文字列と、指定されていない JSON フィールドの両方が `""` として入力され、`""` と `$char(0)` の両方がフィールド値 `""` として出力されます。

%JSONINCLUDE (プロパティのみ)

そのプロパティを JSON の出力または入力に含めるかどうかを指定します。

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

有効な値は以下のとおりです。

- `"inout"` (既定値) ー 入力と出力の両方に含めます。
- `"outputonly"` ー プロパティを入力としては無視します。
- `"inputOnly"` ー プロパティを出力としては無視します。
- `"none"` ー プロパティを一切含めません。

%JSONNULL

指定されていないプロパティの処理を制御します。

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

有効な値は以下のとおりです。

- `0` ー (既定値) 指定されていないプロパティに対応するフィールドは、エクスポート時に無視されます。
- `1` ー 指定されていないプロパティは、NULL 値としてエクスポートされます。

%JSONREFERENCE

オブジェクト参照を JSON フィールドに投影する方法を指定します。

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

有効な値は以下のとおりです。

- "OBJECT" ー (既定値) 参照されるクラスのプロパティを使用して、参照オブジェクトを表します。
- "ID" ー 永続クラスまたはシリアル・クラスの ID を使用して、参照を表します。
- "OID" ー 永続クラスまたはシリアル・クラスの OID を使用して、参照を表します。OID は、classname,id という形式で JSON に投影されます。
- "GUID" ー 永続クラスの GUID を使用して、参照を表します。

5.5.3 %JSON.Formatter のメソッドとプロパティ

%JSON.Formatter クラスを使用して、JSON 文字列、JSON ストリーム、または %DynamicAbstractObject のサブクラスであるオブジェクトをフォーマットすることができます。詳細と例は、“JSON のフォーマット” のセクションを参照してください。

Format()

%JSON.Formatter.Format() は、指定されたインデントを使用して JSON ドキュメントをフォーマットし、それを現在のデバイスに書き込みます。

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

パラメータ：

- inputー 文字列またはストリームとしての JSON か、%DynamicAbstractObject のサブクラス。

FormatToStream()

%JSON.Formatter.FormatToStream() は、指定されたインデントを使用して JSON ドキュメントをフォーマットし、それをストリームに書き込みます。

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

パラメータ：

- input ー 文字列またはストリームとしての JSON か、%DynamicAbstractObject のサブクラス。
- export ー フォーマットされた JSON ストリーム。

FormatToString()

%JSON.Formatter.FormatToString() は、指定されたインデントを使用して JSON ドキュメントをフォーマットし、それを文字列に書き込むか、JSON 対応のクラスを JSON ドキュメントとしてシリアル化し、それを文字列として返します。

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

パラメータ：

- input ー 文字列またはストリームとしての JSON か、%DynamicAbstractObject のサブクラス。

- export (オプション) ー フォーマットされた JSON ストリーム。

Indent

%JSON.Formatter.Indent プロパティでは、JSON 出力をインデントするかどうかを指定します。既定値は true です。

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

IndentChars

%JSON.Formatter.IndentChars プロパティでは、インデントが有効になっている場合にそれぞれのインデント・レベルに使用する文字シーケンスを指定します。既定では、1 つのスペースが使用されます。

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

LineTerminator

%JSON.Formatter.LineTerminator プロパティでは、インデントするときにそれぞれの行を終了する文字シーケンスを指定します。既定値は `$char(13,10)` です。

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

6

ダイナミック・エンティティ・メソッドのクイック・リファレンス

このセクションでは、使用可能な各ダイナミック・エンティティ・メソッドの概要とリファレンス情報を記載しています。ダイナミック・エンティティは、%Library.DynamicObject または %Library.DynamicArray のインスタンスです。これらのクラスは両方とも %Library.DynamicAbstractObject を継承します。この章の各リストには、適切なオンラインのクラス・リファレンス・ドキュメントへのリンクが含まれています。

6.1 メソッドの詳細

このセクションでは、すべての使用可能なダイナミック・エンティティ・メソッドを列挙して、各メソッドについて簡単に説明して、詳細情報へのリンクを提供します。すべてのメソッドはオブジェクトと配列の両方に使用できますが、例外として、%Push() と %Pop() は配列専用です。

%FromJSON()

指定された JSON ソースを解析して、解析された JSON が格納されたデータ型 %DynamicAbstractObject のオブジェクトを返します。解析時にエラーが発生した場合は、例外がスローされます。詳細と例は、“ダイナミック・エンティティと JSON の間の変換” を参照してください。

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

パラメータ：

- str ― 入力は、以下に示すソースのいずれか 1 つにできます。
  - ソースが含まれた文字列値。
  - ソースの読み取り元となるストリーム・オブジェクト。

関連項目：″%FromJSONFile()″、″%ToJSON()″、″大きいダイナミック・エンティティからストリームへのシリアル化″

クラス・リファレンス：%DynamicAbstractObject.%FromJSON()

%FromJSONFile()

指定された JSON ソースを解析して、解析された JSON が格納されたデータ型 %DynamicAbstractObject のオブジェクトを返します。解析時にエラーが発生した場合は、例外がスローされます。詳細と例は、“ダイナミック・エンティティと JSON の間の変換”を参照してください。

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

パラメータ：

- filename ー ソースを読み取ることができるファイル URI の名前。このファイルは UTF-8 形式でエンコードされている必要があります。

関連項目：″%FromJSON()″、″%ToJSON()″、″大きいダイナミック・エンティティからストリームへのシリアル化″

クラス・リファレンス：%DynamicAbstractObject.%FromJSON()

%Get()

指定された有効なオブジェクト・キーまたは配列インデックスを受け取って、対応する値を返します。その値が存在しない場合は、NULL 文字列の "" が返されます。詳細と例は、“%Set()、%Get()、および %Remove() の使用”を参照してください。

```
method %Get(key) as %RawString
```

パラメータ：

- key ー 取得する値のオブジェクト・キーまたは配列インデックス。配列インデックスはキャノニック形式の整数値として渡される必要があります。配列インデックスの先頭位置は 0 です。

関連項目：%Set()、%Remove()、%Pop()

クラス・リファレンス：%DynamicObject.%Get()、%DynamicArray.%Get()

%GetIterator()

%Iterator オブジェクトを返して、ダイナミック・エンティティのすべてのメンバの反復処理を可能にします。詳細と例は、“%GetNext() を使用したダイナミック・エンティティの反復処理”を参照してください。

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

関連項目：%GetNext()

クラス・リファレンス：
%DynamicObject.%GetIterator()、%DynamicArray.%GetIterator()、%Iterator.Object、%Iterator.Array

%GetNext()

これは、%GetIterator() によって返される %Iterator オブジェクトのメソッドです。反復子を進めて、反復子の位置が有効な要素の場合は true を返し、反復子が最終要素を越えている場合は false を返します。key 引数と value 引数は、現在の反復子位置にある有効な要素の値を返します。詳細と例は、“%GetNext() を使用したダイナミック・エンティティの反復処理”を参照してください。

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

パラメータ：

- key ー 現在の位置にある要素のオブジェクト・キーまたは配列インデックスを返します。
- value ー 現在の位置にある要素の値を返します。

関連項目：%GetIterator()

クラス・リファレンス：**%Iterator.Object**.%GetNext()、**%Iterator.Array**.%GetNext()

%GetTypeOf()

有効なオブジェクト・キーまたは配列インデックスが指定されると、値のデータ型を示す文字列を返します。詳細と例は、“データ型を使用した作業”を参照してください。

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

パラメータ：

- key ー テストする値のオブジェクト・キーまたは配列インデックス。

返り値：

以下のいずれかの文字列が返されます。

- `"null"` ー JSON の null
- `"boolean"` ー ゼロ（“false”）またはゼロ以外（“true”）の数値
- `"number"` ー 任意のキャノニック形式の数値
- `"oref"` ー 別のオブジェクトの参照
- `"object"` ー 入れ子になったオブジェクト
- `"array"` ー 入れ子になった配列
- `"string"` ー 標準のテキスト文字列
- `"unassigned"` ー そのプロパティまたは要素は存在しているが、値が割り当てられていない

関連項目：%IsDefined()

クラス・リファレンス：**%DynamicAbstractObject**.%GetTypeOf()

%IsDefined()

key で指定された項目がオブジェクト内で定義されているかどうかをテストします。項目が割り当てられていない場合や存在しない場合は、False を返します。詳細と例は、“%IsDefined() を使用した有効値の有無の確認”を参照してください。

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

パラメータ：

- key ー テストする項目のオブジェクト・キーまたは配列インデックス。配列インデックスはキャノニック形式の整数値として渡される必要があります。配列インデックスの先頭位置は 0 です。

関連項目：”NULL、空の文字列、および未割り当て値の解決”

クラス・リファレンス：**%DynamicObject**.%IsDefined()、**%DynamicArray**.%IsDefined()

%Pop()

配列の最終メンバの値を返します。その後、この値は配列から削除されます。配列が既に空である場合は、このメソッドは空の文字列（`""`）を返します。詳細と例は、“動的配列での %Push と %Pop の使用”を参照してください。

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

関連項目：″%Push()″、″%Get()″、″%Remove()″、″NULL、空の文字列、および未割り当て値の解決″

クラス・リファレンス：**%DynamicArray**.%Pop()

%Push()

指定された新しい値を現在の配列の末尾に追加して、配列の長さを拡大します。現在の変更後の配列を指すoref を返して、%Push() の呼び出しを連鎖可能にします。詳細と例は、“動的配列での %Push と %Pop の使用”を参照してください。

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

パラメータ：

- value ー 新しい配列要素に割り当てる値。
- type ー (オプション) value のデータ型を示す文字列。以下の文字列を使用できます。
  - `"null"` ー JSON の `null`。value 引数は必ず `""` (空の文字列)。
  - `"boolean"` ー JSON の `false` (value 引数は必ず `0`) または `true` (value 引数は必ず `1`)。
  - `"false"` ー JSON の `false` (value 引数は必ず `0`)。
  - `"true"` ー JSON の `true` (value 引数は必ず `1`)。
  - `"number"` ー value をキャノニック形式の数値に変換します
  - `"string"` ー value をテキスト文字列に変換します

  注：指定した value がオブジェクトまたは oref の場合、オプションの type パラメータを使用することはできません。例えば、指定した value がダイナミック・エンティティの場合、type に指定した値にかかわらず、エラーがスローされます。詳細は、“%Set() または %Push() を使用したデフォルトデータ型のオーバーライド”を参照してください。

関連項目：″%Pop()″、″%Set()″、″メソッドの連鎖″

クラス・リファレンス：**%DynamicArray**.%Push()

%Remove()

指定された要素をダイナミック・オブジェクトまたは動的配列から削除して、その削除された要素の値を返します。その要素の値が埋め込みダイナミック・オブジェクトまたは埋め込み動的配列である場合は、すべての従属ノードも削除されます。動的配列の場合は、削除された要素の後続要素すべてについて、添え字位置の値が 1 だけ小さくなります。詳細と例は、“%Set()、%Get()、および %Remove() の使用”を参照してください。

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

パラメータ：

- key ー 削除する要素のオブジェクト・キーまたは配列インデックス。配列インデックスはキャノニック形式の整数値として渡される必要があります。配列インデックスの先頭位置は 0 です。

関連項目：%Set()、%Get()、%Pop()

クラス・リファレンス：**%DynamicObject**.%Remove()、**%DynamicArray**.%Remove()

%Set()

新しい値を作成するか、既存の値を更新します。変更された配列の参照を返して、%Set() の呼び出しを入れ子にすることを可能にします。詳細と例は、“%Set()、%Get()、および %Remove() の使用”を参照してください。

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

パラメータ：

- key ー 作成または更新する値のオブジェクト・キーまたは配列インデックス。配列インデックスはキャノニック形式の整数値として渡される必要があります。配列インデックスの先頭位置は 0 です。
- value ー 既存の値を更新するための新しい値、または新たに作成する値。
- type ー (オプション) value のデータ型を示す文字列。以下の文字列を使用できます。
  - `"null"` ー JSON の `null`。value 引数は必ず `""` (空の文字列)。
  - `"boolean"` ー JSON の `false` (value 引数は必ず `0`) または `true` (value 引数は必ず `1`)。
  - `"false"` ー JSON の `false` (value 引数は必ず `0`)。
  - `"true"` ー JSON の `true` (value 引数は必ず `1`)。
  - `"number"` ー value をキャノニック形式の数値に変換します
  - `"string"` ー value をテキスト文字列に変換します

  注：指定した value がオブジェクトまたは oref の場合、オプションの type パラメータを使用することはできません。例えば、指定した value がダイナミック・エンティティの場合、type に指定した値にかかわらず、エラーがスローされます。詳細は、“%Set() または %Push() を使用したデフォルトデータ型のオーバーライド”を参照してください。

関連項目：″%Get()″、″%Remove()″、″%Push()″、″メソッドの連鎖″

クラス・リファレンス：%DynamicObject.%Set()、%DynamicArray.%Set()

%Size()

ダイナミック・オブジェクトまたは動的配列のサイズを示す整数を返します。配列の場合は、このサイズには配列内の未割り当て要素も含まれます。オブジェクトの場合は、このサイズには値が割り当てられている要素のみが含まれます。詳細と例は、“%Size() を使用したスパース配列の反復処理”を参照してください。

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

関連項目：%GetNext()

クラス・リファレンス：%DynamicAbstractObject.%Size()

%ToJSON()

%DynamicAbstractObject. のインスタンスを JSON 文字列に変換します。詳細と例は、“ダイナミック・エンティティと JSON の間の変換”を参照してください。

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

パラメータ：

- outstrm ー オプション。以下のようないくつかのケースが考えられます。
  - outstrm が指定されておらず、このメソッドが DO を介して呼び出された場合は、JSON 文字列は現在の出力デバイスに書き込まれます。

- outstrm が指定されておらず、このメソッドが式として呼び出された場合は、JSON 文字列はその式の値になります。
- outstrm が %Stream.Object のインスタンスとして指定されている場合は、JSON 文字列がストリームに書き込まれます (詳細と例は、“大きいダイナミック・エンティティからストリームへのシリアル化” を参照)。
- outstrm がオブジェクトであるが、%Stream.Object のインスタンスではない場合は、例外がスローされます。
- outstrm がオブジェクトではなく、NULL でもない場合は、完全修飾ファイル指定であると見なされます (ファイルのフル・パスを指定する必要があります)。このファイルは新規作成された %Stream.FileCharacter ストリームにリンクされ、JSON 文字列はストリームに書き込まれ、完了時にストリームがこのファイルに保存されます。

関連項目 : ″%FromJSON()″ %FromJSONFile()

クラス・リファレンス : %DynamicAbstractObject.%ToJSON()