クラスの定義と使用

Version 2024.1
2024-06-03

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

クラスの定義と使用
InterSystems IRIS Data Platform　Version 2024.1　2024-06-03

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

図一覧

テーブル一覧

1

クラス・プログラミングの基本的な考え方

このトピックは、クラス・プログラミングの詳しい知識がない人に、この種のプログラミングがどのように機能するのかという観念を紹介することを目的としています。クラス・プログラミングの詳しい知識がある場合は、単にコード・サンプルに目を通すことで InterSystems IRIS® データ・プラットフォームにおけるクラス・プログラミングがどのようなものか把握しておくと役立ちます。

例では ObjectScript を使用していますが、ここで説明する概念は、言語にほとんど依存していません。

1.1 オブジェクトとプロパティ

クラス・プログラミングでは、中心となる概念はオブジェクトです。オブジェクトは、1 つのセットとして共に格納されたり渡される一連の値のコンテナです。オブジェクトは、多くの場合、患者、診断結果、取引などの実社会のエンティティに対応しています。

クラス定義は、多くの場合、指定されたタイプのオブジェクトに対するテンプレートです。クラス定義には、それらのオブジェクトに対する値を格納するためのプロパティがあります。例えば、MyApp.Clinical.PatDiagnosis というクラスがあり、このクラスに Date、EnteredBy、PatientID、DiagnosedBy、Code、およびその他のプロパティがあるとします。

クラスのインスタンスを作成することで、テンプレートを使用します。これらのインスタンスはオブジェクトです。例えば、ユーザが診断結果をユーザ・インタフェースに入力し、そのデータを保存するとします。基礎となるコードは、以下のロジックを持ちます。

1. 診断結果テンプレートから新しい診断結果オブジェクトを作成します。
2. 必要に応じて、そのオブジェクトのプロパティの値を設定します。その値は、必須である場合、既定値がある場合、他の値に基づいて計算される場合、完全にオプションである場合があります。
3. オブジェクトを保存します。

   この操作によって、データが保存されます。

以下は、ObjectScript の使用例です。

ObjectScript

```
//create the object
set diagnosis=##class(MyApp.Clinical.PatDiagnosis).%New()

//set a couple of properties by using special variables
set diagnosis.Date=$SYSTEM.SYS.TimeStamp()
set diagnosis.EnteredBy=$username

//set other properties based on variables set earlier by
//the user interface
set diagnosis.PatientID=patientid
set diagnosis.DiagnosedBy=clinicianid
set diagnosis.Code=diagcode

//save the data
//the next line tries to save the data and returns a status to indicate
//whether the action was successful
set status=diagnosis.%Save()
//always check the returned status
if $$$ISERR(status) {do $System.Status.DisplayError(status) quit status}
```

以下の点に注意してください。

- オブジェクトのプロパティを参照するには、構文 object_variable.property_name を使用します。例えば、`diagnosis.DiagnosedBy` とします。
- %New() および %Save() は、MyApp.Clinical.PatDiagnosis クラスのメソッドです。

  次のセクションでは、メソッドのタイプと、ここに示すようなさまざまな方法で呼び出す理由について説明します。

1.2 メソッド

メソッドは、ほとんどの場合、プロシージャです（"変数の可用性と範囲"も参照してください）。メソッドは互いに呼び出すことができ、プロパティおよびパラメータを参照できます。

クラス言語には、インスタンス・メソッドとクラス・メソッドの 2 種類のメソッドがあります。これらは、異なる目的を持ち、使用法も異なります。

1.2.1 インスタンス・メソッド

インスタンス・メソッドが意味を持つのは、クラスのインスタンスから呼び出された場合のみです。これは、通常、そのインスタンスに、またはそのインスタンスで、何かを実行しているためです。以下はその例です。

ObjectScript

```
set status=diagnosis.%Save()
```

例えば、患者を表すクラスを定義しているとします。このクラスで、以下の処理を実行するインスタンス・メソッドを定義するとします。

- 患者の BMI（ボディマス指数）を計算する
- 患者の情報をまとめたレポートを印刷する
- 患者が特定の処置の対象となるかどうかを判別する

これらの各処理には、患者に対して格納されているデータの知識が必要であり、それが、多くのプログラマがそれらをインスタンス・メソッドとして作成する理由です。インスタンス・メソッドの実装は、通常、内部でそのインスタンスのプロパティを参照します。以下は、2 つのプロパティを参照するインスタンス・メソッドの定義の例を示しています。

Class Member

```
Method GetBMI() as %Numeric
{
 Set bmi=..WeightKg / (..HeightMeter**2)
 Quit bmi
}
```

このメソッドを使用するには、アプリケーション・コードに以下のような行を含めます。

ObjectScript

```
 //open the requested patient given an id selected earlier
 set patient=##class(MyApp.Clinical.PatDiagnosis).%OpenId(id)

 //get value to display in BMI Display field
 set BMIDisplay=patient.GetBMI()
```

1.2.2 クラス・メソッド

メソッドのもう 1 つのタイプがクラス・メソッド (他の言語では静的メソッドと呼ぶ) です。このタイプのメソッドを呼び出すには、インスタンスを参照しない構文を使用します。以下はその例です。

ObjectScript

```
 set patient=##class(MyApp.Clinical.PatDiagnosis).%New()
```

クラス・メソッドを作成する一般的な理由は、以下のとおりです。

- クラスのインスタンスを作成するアクションを実行する必要がある。

  当然ながら、このアクションはインスタンス・メソッドにすることができません。
- 複数のインスタンスに作用するアクションを実行する必要がある。

  例えば、患者のグループを別の一次診療医に再び割り当てることが必要な場合があります。
- どのインスタンスにも作用しないアクションを実行する必要がある。

  例えば、時刻、ランダムな数、または特定の方法でフォーマットされた文字列を返すメソッドを作成できます。

1.2.3 メソッドと変数の範囲

メソッドは、通常、変数の値を設定します。ほとんどすべてのクラスで、これらの変数はこのメソッド内でのみ使用可能です。例えば、以下のクラスを考えてみます。

Class Definition

```
Class GORIENT.VariableScopeDemo
{

ClassMethod Add(arg1 As %Numeric, arg2 As %Numeric) As %Numeric
{
    Set ans=arg1+arg2
    Quit ans
}

ClassMethod Demo1()
{
   set x=..Add(1,2)
   write x
}

ClassMethod Demo2()
{
   set x=..Add(2,4)
   write x
```

```
}
}
```

Add() メソッドは、ans という名前の変数を設定し、その変数に含まれている値を返します。

メソッド Demo1() は、メソッド Add() を引数 1 と 2 で呼び出し、その答えを書き込みます。メソッド Demo2() も類似していますが、別のハードコードされた引数を使用します。

メソッド Demo1() または Demo2() が変数 ans を参照しようとしても、その変数はそのコンテキストでは定義されておらず、InterSystems IRIS によってエラーがスローされます。

同様に、Add() は変数 x を参照できません。また、Demo1() 内の変数 x は、Demo2() 内の変数 x とは異なる変数です。

これらの変数の範囲は制限されています。それが、InterSystems IRIS クラスの既定の動作 (および他のクラス言語における通常の動作) であるためです。

クラス定義内では、ほとんどすべての場合に、値を、メソッドに対する引数として含めることで渡すことができます。これは、クラス・プログラミングにおける慣例です。この慣例によって、変数の範囲を決定する作業が簡素化されます。

対照的に、ルーチンを作成する場合は、範囲設定を制御するルールを理解する必要があります。これらについては、″ObjectScript の使用法″ で説明します。

1.3 クラス定数 (パラメータ)

場合によっては、クラスが定数値に簡単にアクセスできると便利です。InterSystems IRIS クラスでは、そのような値がクラス・パラメータです。他の言語では、代わりにクラス定数という用語を使用します。以下に例を示します。

Class Member

```
Parameter MYPARAMETER = "ABC" ;
```

クラス・パラメータは、コンパイル時に値を取得し、後で変更することはできません。

メソッドはパラメータを参照でき、それがパラメータを定義する理由です。以下はその例です。

ObjectScript

```
 set myval=..#MYPARAMETER * inputvalue
```

1.4 クラス定義とタイプ

以下は、クラス定義の例を示しています。これはクラス定義におけるタイプの説明に使用します。

Class Definition

```
Class MyClass Extends %Library.Persistent
{
Parameter MYPARAMETER = "ABC" ;

Property DateOfBirth As %Library.Date;

Property Home As Sample.Address;

Method CurrentAge() As %Library.Integer
{
 //details
}

ClassMethod Addition(x As %Library.Integer, y As %Library.Integer) As %Library.Integer
{
 //details
}

}
```

このクラス定義は 1 つのパラメータ (MYPARAMETER)、2 つのプロパティ (DateOfBirth と Home)、1 つのインスタンス・メソッド (CurrentAge())、および 1 つのクラス・メソッド (Addition()) を定義します。

クラス・プログラミングでは、タイプは、以下の主要な場所で指定できます。

- クラス自体に対して。Extends の後の要素がタイプです。

  各タイプは、クラスの名前です。
- パラメータに対して。この場合と残りの場合は、As の後の要素がタイプです。
- プロパティに対して。Home プロパティの場合、タイプはそれ自体がプロパティを含むクラスです。

  この場合、タイプはオブジェクト値を持ちます。この例では、これはオブジェクト値プロパティです。

  オブジェクト値プロパティは、他のオブジェクト値プロパティを含むことができます。
- メソッドの返り値に対して。
- メソッドによって使用される引数の値に対して。

1.5 継承

大部分のクラスベースの言語の主な特徴は、継承です。1 つのクラスは、他のクラスを継承でき、したがって、他のクラスのパラメータ、プロパティ、メソッド、および他の要素を取得できます。このパラメータ、プロパティ、メソッド、および他の要素は、総称してクラス・メンバと呼ばれます。

1.5.1 用語と基本事項

クラス A がクラス B から継承する場合、次の用語を使用します。

- クラス A はクラス B のサブクラスです。または、クラス A はクラス B を拡張したものです。

  場合によっては、クラス A はクラス B のサブタイプであるともいいます。
- クラス B はクラス A のスーパークラスです。

  場合によっては、クラス A はクラス B の子クラスであり、クラス B は親クラスであるともいいます。この用語は、一般的ですが誤解を与えることもあります。用語親と子が、SQL テーブルを説明するときにまったく異なる意味で使用されるためです。

クラスが他のクラスから継承する場合、それらの他のクラスのクラス・メンバを、そのスーパークラス自体が継承したメンバも含めて取得します。サブクラスは、継承したクラス・メンバをオーバーライドできます。

1 つのクラスの複数のスーパークラスで、同じ名前のメソッド、同じ名前のプロパティなどが定義されていることがあります。したがって、どのスーパークラスの定義をサブクラスで使用するのかを決定する規則が必要です ("継承" を参照してください)。

InterSystems IRIS クラス・ライブラリでは、スーパークラスは、通常、それぞれ用途が異なり、メンバは異なる名前を持っているため、メンバ名の競合はほとんどありません。

1.5.2 例

以下に例を示します。

```
/// Finds files in a FilePath directory and submits all that match a FileSpec wildcard to
/// an associated BusinessService for processing within InterSystems IRIS
Class EnsLib.File.InboundAdapter Extends (Ens.InboundAdapter, EnsLib.File.Common)
```

この例は、クラスが、どのようにしてさまざまなスーパークラスからのロジックを結合できるのかを例示することのみを目的としています。この EnsLib.File.InboundAdapter クラスは、まったく異なることを実行する以下の 2 つのクラスから継承します。

- Ens.InboundAdapter。InterSystems IRIS の概念の 1 つであるインバウンド・アダプタと呼ばれるものに対するビジネス・ロジックを含んでいます。
- EnsLib.File.Common。指定されたディレクトリの一連のファイルを操作するためのロジックを含んでいます。

EnsLib.File.InboundAdapter では、メソッドは、これらのクラスの両方およびそれらのスーパークラスからのロジックを使用します。

1.5.3 継承されるクラス・メンバの使用

編集ツールでクラスの定義を表示しても、それに含まれる継承されたメンバは表示されませんが、コードはそれらを参照できます。

例えば、クラス A が 2 つのプロパティを持ち、それぞれが以下のように既定値を持つとします。

Class Definition

```
Class Demo.A
{
Property Prop1 as %Library.String [InitialExpression = "ABC"];

Property Prop2 as %Library.String [InitialExpression = "DEF"];
}
```

クラス B は、以下のように表示されます。

Class Definition

```
Class Demo.B Extends Demo.A
{
Method PrintIt()
{
 Write ..Prop1,!
 Write ..Prop2,!
}

}
```

前述したとおり、サブクラスは、継承したクラス・メンバをオーバーライドできます。例えば、クラス C もクラス A から継承できますが、そのプロパティの 1 つの既定値を以下のようにオーバーライドできます。

Class Definition

```
Class Demo.C Extends Demo.A
{
Property Prop2 as %Library.String [InitialExpression = "GHI"];

}
```

1.5.4 サブクラスの使用

クラス B が クラス A から継承している場合、クラス A のインスタンスを使用できる場所であればどこでもクラス B のインスタンスを使用できます。

例えば、以下のようなユーティリティ・メソッドがあるとします。

Class Member

```
ClassMethod PersonReport(person as MyApp.Person) {
 //print a report that uses properties of the instance
}
```

MyApp.Person のインスタンスをこのメソッドへの入力として使用できます。また、MyApp.Person のどのサブクラスのインスタンスも使用できます。以下はその例です。

ObjectScript

```
 //id variable is set earlier in this program
 set employee=##class(MyApp.Employee).%OpenId(id)
 do ##class(Util.Utils).PersonReport(employee)
```

同様に、メソッドの返り値（それが値を返す場合）は、指定されたタイプのサブクラスのインスタンスにすることができます。例えば、MyApp.Employee と MyApp.Patient がどちらも MyApp.Person のサブクラスであるとします。以下のようにメソッドを定義できます。

Class Member

```
ClassMethod ReturnRandomPerson() as MyApp.Person
{
 Set randomnumber = $RANDOM(10)
 If randomnumber > 5 {
     set person=##class(MyApp.Employee).%New()
 }
 else {
     set person=##class(MyApp.Patient).%New()
 }
 quit person
}
```

1.6 メソッドのコンテナとしてのクラス

前述のとおり、クラス定義は、多くの場合、オブジェクトのテンプレートです。また、クラスは、同類の一連のクラス・メソッドのコンテナとなることもあります。この場合、このクラスのインスタンスを作成することはありません。その中のクラス・メソッドを呼び出すのみです。

例については、%SYSTEM パッケージのクラスを参照してください。

1.7 抽象クラス

また、抽象クラスを定義すると役立ちます。抽象クラスは、通常、汎用インタフェースを定義するものであり、インスタンス化することはできません。このクラス内のメソッド定義は、メソッドのシグニチャを宣言しますが、その実装は宣言しません。

開発者は抽象クラスを定義し、インタフェースを記述します。次に、その開発者または他の開発者がサブクラスを作成し、そのサブクラス内でメソッドを実装します。その実装は、抽象クラスで指定されているシグニチャと一致している必要があります。このシステムによって、わずかに用途が異なるが同一のインタフェースを持つ複数の類似したクラスを開発できます。このため、システム・クラスの多くは共通のインタフェースを持っています。

クラスが抽象でない場合でもメソッドを抽象として指定することもできます。

1.8 関連項目

- クラスの定義
- クラス・ドキュメントの作成

2
クラスの定義

ここでは、InterSystems IRIS® データ・プラットフォームにおけるクラス定義の基本について説明します。

2.1 用語の概要

以下に、簡単な InterSystems IRIS クラス定義と、いくつかの代表的な要素を示します。

Class Definition

```
Class Demo.MyClass Extends %RegisteredObject
{

Property Property1 As %String;

Property Property2 As %Numeric;

Method MyMethod() As %String
{
   set returnvalue=..Property1_..Property2
   quit returnvalue
}

}
```

以下の点に注意してください。

- 完全なクラス名は `Demo.MyClass` です。パッケージ名は `Demo`、短いクラス名は `MyClass` です。
- このクラスは、クラス **%RegisteredObject** を拡張しています。同様に、このクラスは、**%RegisteredObject** を継承しています。

  %RegisteredObject は、このクラスのスーパークラスです。つまり、このクラスは、**%RegisteredObject** のサブクラスになります。InterSystems IRIS クラスは、このトピックで後述するように、複数のスーパークラスを持つことができます。

  クラスのスーパークラスにより、そのクラスの用途が決まります。
- このクラスでは、2 つのプロパティ `Property1` と `Property2` を定義しています。プロパティ `Property1` のタイプは **%String** です。また、プロパティ `Property2` のタイプは **%Numeric** です。
- このクラスでは、1 つのメソッド `MyMethod()` を定義しています。このメソッドは、**%String** タイプの値を返します。

このクラスでは、InterSystems IRIS が提供するシステム・クラスを参照しています。これに該当するクラスは、**%RegisteredObject** (完全名は **%Library.RegisteredObject**)、**%String** (**%Library.String**)、および **%Numeric** (**%Library.Numeric**) です。**%RegisteredObject** は、オブジェクト・インタフェースを定義しているため、InterSystems IRIS の重要なクラスの 1 つといえます。これにより、オブジェクト・インスタンスを作成するメソッドとオブジェクト・インスタンスを

操作するメソッドが提供されます。%String と %Numeric はデータ型クラスです。そのため、それに対応するプロパティは、リテラル値を保持することになります (その他の種類の値は保持しません)。

2.2 クラスの種類

InterSystems IRIS には、クラス定義の大規模セットが用意されています。これは、以下のような一般的な方法でクラスで使用できます。

- クラスは、クラスのスーパークラスとして使用できます。
- クラスは、プロパティの値として、メソッドへの引数の値として、メソッドから返される値などとして使用できます。
- 特定の API を提供するだけのクラスもあります。通常、このようなクラスは、上記のいずれの方法でも使用してはいけません。その代わりに、API のメソッドを呼び出すコードを作成してください。

スーパークラスの最も一般的な選択項目は、以下のとおりです。

- %RegisteredObject ― このクラスは、最も汎用的な形式でオブジェクト・インタフェースを表します。
- %Persistent ― このクラスは、永続クラスを表します。オブジェクト・インタフェースを提供することに加え、このクラスは、データベースにオブジェクトを保存するためのメソッドと、データベースからオブジェクトを読み取るためのメソッドも提供します。
- %SerialObject ― このクラスは、別のオブジェクトに埋め込み可能な (別のオブジェクト内でシリアル化できる) オブジェクトを表します。
- 上記のクラスのいずれかのサブクラス。
- なし ― クラスを作成するときに、スーパークラスを指定する必要はありません。

プロパティの値、メソッドへの引数の値、メソッドから返される値などについて、最も一般的な選択項目は以下のとおりです。

- オブジェクト・クラス (前述のリストに含まれるクラス)
- リテラル・プロパティに使用されるデータ型クラス
- コレクション・クラス
- ストリーム・クラス

2.2.1 オブジェクト・クラス

オブジェクト・クラスとは、%RegisteredObject のサブクラスを指す語句です。オブジェクト・クラスでは、クラスのインスタンスの作成、インスタンスのプロパティの指定、およびインスタンスのメソッドの呼び出しができます。

オブジェクトとは、オブジェクト・クラスのインスタンスを指す一般的な用語です。

オブジェクト・クラスには、3 つの一般的なカテゴリがあります。

- 一時的なオブジェクト・クラスまたは登録オブジェクト・クラスは、%RegisteredObject のサブクラスですが、%Persistent や %SerialObject のサブクラスではありません (以下の項目を参照)。

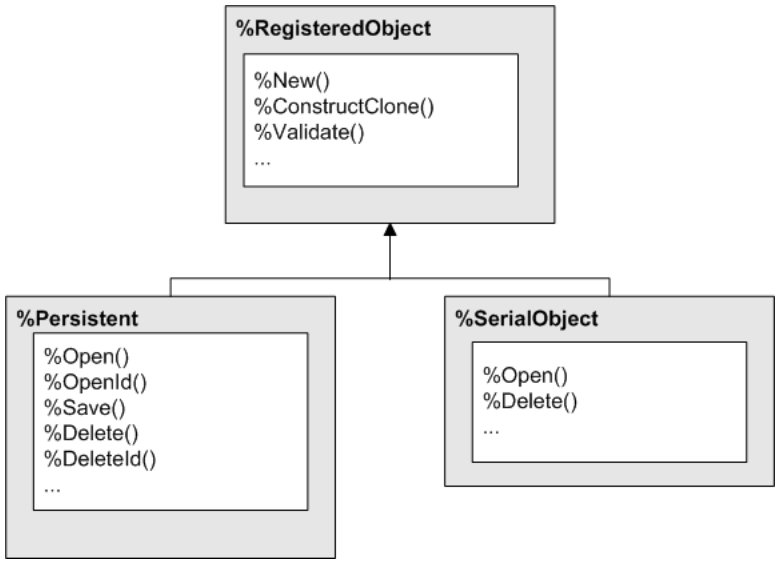
  詳細は、"登録オブジェクトを使用した作業" を参照してください。
- 永続クラスは、%RegisteredObject の直接サブクラスである %Persistent のサブクラスです。%Persistent クラスには、%RegisteredObject の動作が組み込まれています。さらに、オブジェクトをディスクに保存する機能、そのオブジェクトを再度開く機能などが追加されています。

詳細は、"永続オブジェクトの概要" を参照してください。

- シリアル・クラスは、%RegisteredObject の直接サブクラスである %SerialObject のサブクラスです。%SerialObject クラスには、%RegisteredObject の動作が組み込まれています。さらに、オブジェクトの状態を表現する文字列の作成機能が追加されています。この機能は、別のオブジェクト (通常は、一時的なオブジェクトまたは永続オブジェクト) のプロパティとして組み込むことを目的としています。オブジェクトのシリアル化とは、この文字列を作成を指す語句です。

  詳細は、"オブジェクト値プロパティの定義と使用" を参照してください。

以下の図は、これら 3 つのクラス間の継承関係を示しています。ボックスには、クラスで定義されたメソッドのいくつかをリストしています。



コレクション・クラスとストリーム・クラスは、シリアル化の動作を備えたオブジェクト・クラスです。

2.2.2 データ型クラス

データ型クラスとは、ClassType キーワードが `datatype` に設定されているクラス、または、そのようなクラスのサブクラスのことを指す語句です。これに該当するクラスは、オブジェクト・クラスではありません (データ型クラスでは、プロパティを定義できません。また、クラスのインスタンスを作成できません)。データ型クラス (より厳密には、データ型ジェネレータ・クラス) の用途は、オブジェクト・クラスのプロパティのタイプとして使用することです。

2.3 クラス・メンバの種類

InterSystems IRIS クラスの定義には、以下の項目を含めることができ、これらはすべてクラス・メンバと呼ばれます。

- パラメータ ー パラメータは、このクラスによって使用される定数値を定義します。ほとんどの場合、この値はコンパイル時に設定されます。
- メソッド ー InterSystems IRIS は、インスタンス・メソッドとクラス・メソッドの 2 種類のメソッドをサポートしています。インスタンス・メソッドは、クラスの特定のインスタンスから呼び出され、そのインスタンスに関連する何らかのアクションを実行します。このタイプのメソッドは、オブジェクト・クラス内でのみ役立ちます。クラス・メソッドは、そのクラスのインスタンスがメモリ内にあるかどうかにかかわらず、呼び出すことができます。このタイプのメソッドは、他の言語では静的メソッドと呼ばれています。

- プロパティープロパティには、クラスのインスタンスのデータが格納されます。プロパティは、オブジェクト・クラスでのみ役立ちます。詳細は、以下のセクションで説明します。
- クラス・クエリ ー クラス・クエリは、クラスが使用できる SQL クエリを定義し、クエリのためのコンテナとして使用するクラスを指定します。多くの場合 (必須ではない)、永続クラスには、クラス・クエリを定義します。これは、そのクラスの保存済みデータにクエリを実行するためのものです。ただし、クラス・クエリは、どのようなクラスにでも定義できます。
- その他の種類のクラス・メンバ (永続クラスにのみ関連するメンバ) :
  - ストレージ定義
  - インデックス
  - 外部キー
  - SQL トリガ
- XData ブロック ー XData ブロックは、クラス内に定義されたデータの名前付きユニットです。通常、クラスのメソッドで使用します。これらには、多数の用途があります。
- プロジェクション ー クラス・プロジェクションは、クラス・コンパイラの動作を拡張する方法を提供します。

  プロジェクションのメカニズムは、Java プロジェクションで使用されます。これが、プロジェクションという用語の語源です。

さまざまなクラス・メンバの定義に使用される言語 (ObjectScript、Python、SQL、その他メンバの実装に使用されるコードを除く) は、クラス定義言語 (CDL) と呼ばれることもあります。

2.4 プロパティの種類

形式上、プロパティには、属性およびリレーションシップという 2 種類のプロパティがあります。

属性は値を保持します。属性プロパティは、通常、単にプロパティと呼びます。プロパティの定義に応じて、保持できる値は、以下のいずれかにできます。

- リテラル値 (`"MyString"` や 1 など)。リテラル値を保持するプロパティは、データ型クラスに基づいているため、データ型プロパティと呼ばれることもあります。"リテラル・プロパティの定義と使用" を参照してください。
- ストリーム。ストリームとは、文字列には長すぎる値を格納するオブジェクトです。"ストリームを使用した作業" を参照してください。
- コレクション。InterSystems IRIS には、プロパティをリストまたは配列として定義する機能があります。リストまたは配列の項目は、リテラル値になるか、オブジェクトにできます。"コレクションを使用した作業" を参照してください。
- その他の種類のオブジェクト。"オブジェクト値プロパティの定義と使用" を参照してください。

リレーションシップは、オブジェクト間の関連付けを保持します。リレーションシップ・プロパティはリレーションシップと呼ばれます。リレーションシップは、永続クラスでのみサポートされます。"リレーションシップの定義と使用" を参照してください。

2.5 クラスの定義法: 基本

このセクションでは、基本的なクラス定義について詳細に説明します。以下のトピックについて説明します。

- スーパークラスの選択

- クラス・キーワードの指定
- インクルード・ファイル
- クラス・パラメータの定義法の概要
- プロパティの定義法の概要
- メソッドの定義法の概要

一般に、クラスを定義するには、統合開発環境 (IDE) を使用します。クラス定義クラスまたは XML クラス定義ファイルを使用して、プログラム的にクラスを定義することもできます。SQL DDL 文を使用して SQL テーブルを定義する場合、システムはそれに対応するクラス定義を作成します。

2.5.1 スーパークラスの選択

クラスを定義する場合、設計上の最初の決定事項の 1 つは、独自のクラスのベースにするクラス (複数可) を選択することです。スーパークラスが 1 つのみの場合は、Extends に続けてスーパークラスの名前をクラス定義の冒頭部分に含めます。

Class Definition

```
Class Demo.MyClass Extends Superclass
{
//...
}
```

スーパークラスが複数ある場合は、それらをコンマ区切りのリストとして、括弧で囲って指定します。

Class Definition

```
Class Demo.MyClass Extends (Superclass1, Superclass2, Superclass3)
{
//...
}
```

クラスを作成するときに、スーパークラスを指定する必要はありません。一般に、クラスがどのような種類のオブジェクトも表現しないとしても、共通に使用される多数のマクロにクラスからアクセスできるようにするために、スーパークラスとして %RegisteredObject を使用します。その代わりに、それらが格納されたインクルード・ファイルを直接取り込むこともできます。

2.5.2 インクルード・ファイル

%RegisteredObject またはそのサブクラスを拡張しないクラスを作成するときには、以下のインクルード・ファイルを取り込むことができます。

- %occStatus.inc。%Status の値を操作するためのマクロが定義されています。
- %occMessages.inc。メッセージを操作するためのマクロが定義されています。

  これらのインクルード・ファイルで定義されているマクロの詳細は、"システムにより提供されるマクロの使用" を参照してください。

%RegisteredObject またはそのサブクラスを拡張するクラスの場合は、これらのマクロが自動的に利用できるようになります。

また、独自のインクルード・ファイルを作成して、そのファイルを必要に応じてクラス定義に取り込むことができます。

クラス定義の先頭にインクルード・ファイルを取り込むには、以下の形式の構文を使用します。インクルード・ファイルの拡張子 .inc は省略する必要がある点に注意してください。

```
Include MyMacros
```

例：

```
Include %occInclude

Class Classname
{
}
```

クラス定義の先頭に複数のインクルード・ファイルを取り込むには、以下の形式の構文を使用します。

```
Include (MyMacros, YourMacros)
```

この構文では、先頭にシャープ記号が付かない点に注意してください（習慣的に必要とする構文とは異なります）。また、Include 指示文は、大文字と小文字が区別されません。例えば、INCLUDE を使用することも可能です。インクルード・ファイルの名前は、大文字と小文字が区別されます。

"#include" も参照してください。

2.5.3 クラス・キーワードの指定

場合によっては、クラス・コンパイラが生成するコードの詳細を制御することが必要になります。例えば、永続クラスについて、既定のテーブル名の使用を望まない場合（または使用できない場合）は、SQL テーブルの名前を指定できます。もう 1 つ例を挙げると、クラスに Final のマークを付けて、そのクラスのサブクラスが作成されないようにすることもできます。クラス定義では、そのような目的に応じた特定のキーワードのセットをサポートしています。クラス・キーワードの指定が必要な場合は、以下に示すように、スーパークラスの後の角括弧でクラス・キーワードを囲みます。

```
Class Demo.MyClass Extends Demo.MySuperclass [ Keyword1, Keyword2, ...]
{

//...

}
```

例えば、使用可能なクラス・キーワードには、Abstract と Final があります。概要は、"コンパイラ・キーワードの概要" を参照してください。InterSystems IRIS には、各種のクラス・メンバに固有のキーワードも用意されています。

2.5.4 クラス・パラメータの定義法の概要

クラス・パラメータは、特定のクラスのすべてのオブジェクトに対する定数値を定義します。クラス・パラメータをクラス定義に追加するには、以下のいずれかのような要素をクラスに追加します。

Class Member

```
Parameter PARAMNAME as Type;

Parameter PARAMNAME as Type = value;

Parameter PARAMNAME as Type [ Keywords ] = value;
```

Keywords はパラメータ・キーワードを表します。キーワードの概要は、"コンパイラ・キーワードの概要" を参照してください。パラメータ・キーワードの詳細は、"パラメータの構文とキーワード" を参照してください。これはオプションです。

2.5.5 プロパティの定義法の概要

オブジェクト・クラスには、プロパティを含めることができます。

プロパティをクラス定義に追加するには、以下のいずれかのような要素をクラスに追加します。

Class Member

```
Property PropName as Classname;

Property PropName as Classname [ Keywords ] ;

Property PropName as Classname(PARAM1=value,PARAM2=value) [ Keywords ] ;

Property PropName as Classname(PARAM1=value,PARAM2=value) ;
```

PropName はプロパティの名前です。Classname はオプションのクラス名です (これを省略すると、プロパティのタイプは %String と見なされます)。

Keywords はプロパティ・キーワードを表します。キーワードの概要は、"コンパイラ・キーワードの概要" を参照してください。プロパティ・キーワードの詳細は、"プロパティの構文とキーワード" を参照してください。これはオプションです。

プロパティで使用するクラスによっては、3 番目と 4 番目のバリエーションに示すように、プロパティ・パラメータも指定できるようになります。

プロパティ・パラメータを含める場合は、パラメータを括弧で囲み、プロパティ・キーワードの前に置きます。プロパティ・キーワードを含める場合、キーワードは角括弧で囲む点にも注意してください。

2.5.6 メソッドの定義法の概要

InterSystems IRIS クラスには、クラス・メソッドとインスタンス・メソッドという 2 種類のメソッドを定義できます。

クラス・メソッドをクラス定義に追加するには、以下に示すような要素をクラスに追加します。

```
ClassMethod MethodName(arguments) as Classname [ Keywords]
{
//method implementation
}
```

MethodName はメソッドの名前です。また、arguments は引数のコンマ区切りリストです。Classname は、このメソッドで返される値 (ある場合) のタイプを表す、オプションのクラス名です。メソッドが値を返さない場合は、As Classname 部分を省略します。

Keywords はメソッド・キーワードを表します。キーワードの概要は、"コンパイラ・キーワードの概要" を参照してください。メソッド・キーワードについては、"クラス定義リファレンス" の Method キーワードを参照してください。これはオプションです。

インスタンス・メソッドを追加するには、同じ構文で `ClassMethod` の代わりに `Method` を使用します。

```
Method MethodName(arguments) as Classname [ Keywords]
{
//method implementation
}
```

インスタンス・メソッドは、オブジェクト・クラスでのみ存在価値があります。

2.6 名前付け規約

クラスおよびクラス・メンバは、ここで概要を説明する名前付け規約に従います。

詳細は、"識別子のルールとガイドライン" および "ネームスペースで何にアクセス可能か" を参照してください。

2.6.1 一般的なルール

すべての識別子は、そのコンテキスト内で一意であることが必要です (例えば、指定されたネームスペース内で 2 つのクラスが同時に完全に同じ名前を持つことはできません)。

大文字と小文字は区別されます。つまり、名前の大文字と小文字が正確に一致している必要があります。また、2 つのクラスに、大文字と小文字が異なるだけの同じ名前を付けることはできません。例えば、識別子 `id1` と `ID1` は一意性を保つ目的からは同一と見なされます。

2.6.2 クラス名

完全なクラス名は、パッケージ名とクラス名の 2 つの部分で構成されています。名前の中で最後の . 文字以降がクラス名になります。クラス名はそのパッケージ内で一意でなければならず、パッケージ名も InterSystems IRIS ネームスペース内で一意である必要があります。完全なクラス名 (つまり、パッケージ名で開始される名前) は、文字または % 記号で始まる必要があります。% 記号で始まるパッケージ名を持つクラスはいずれも、すべてのネームスペース内で利用できます。

永続クラスは自動的に SQL テーブルとして投影されるので、クラス定義は SQL 予約語ではないテーブル名を指定する必要があります。永続クラスの名前が SQL 予約語である場合、クラス定義は SQLTableName キーワードに対して予約語でない有効な値を指定する必要もあります。

パッケージの詳細は、"パッケージのオプション" を参照してください。

2.6.3 クラス・メンバ名

すべてのクラス・メンバ (プロパティやメソッドなど) の名前は、そのクラスの範囲で一意である必要があります。異なるタイプのメンバであるとしても、2 つのメンバに同一の名前を付与しないことを強くお勧めします。予期しない結果が生じる可能性があります。

さらに、永続クラスのメンバは、そのメンバの識別子として SQL 予約語を使用できません。ただし、そのメンバの SQLName または SQLFieldName キーワードを使用して、別名を定義することはできます (適切な場合)。

メンバ名の範囲を指定できます。これにより、それ以外では許可されない文字をメンバ名に使用できます。範囲指定したメンバ名を作成するには、名前の最初と最後の文字に二重引用符を使用します。以下はその例です。

Class Member

```
Property "My Property" As %String;
```

2.7 継承

InterSystems IRIS クラスは、既存のクラスを継承できます。あるクラスを別のクラスから継承する場合、継承して発生するクラスをサブクラスと呼び、その派生元のクラスをスーパークラスと呼びます。

2 つのスーパークラスを使用するクラス定義の例を以下に示します。

Class Definition

```
Class User.MySubclass Extends (%Library.Persistent, %Library.Populate)
{
}
```

スーパークラスからメソッドを継承するクラスに加え、プロパティはシステム・プロパティの動作クラスから追加のメソッドを継承します。また、データ型属性の場合は、データ型クラスから継承します。

例えば、以下の Person という定義済みクラスがあるとします。

Class Definition

```
Class MyApp.Person Extends %Library.Persistent
{
Property Name As %String;
Property DOB As %Date;
}
```

このクラスから、新規クラスの Employee を容易に派生できます。

Class Definition

```
Class MyApp.Employee Extends Person
{
Property Salary As %Integer;
Property Department As %String;
}
```

この定義は、Employee クラスを Person クラスのサブクラスとして確立するものです。独自のクラス・パラメータやプロパティ、メソッドに加え、Employee クラスは Person クラスからのすべての要素を含みます。

2.7.1 サブクラスの使用

スーパークラスを使用するすべての場所で、サブクラスを使用できます。例えば、上で定義した Employee クラス、および Person クラスを使用すると、Employee オブジェクトを開いて、Person として参照できます。

ObjectScript

```
 Set x = ##class(MyApp.Person).%OpenId(id)
 Write x.Name
```

以下のように、Employee 固有の属性、またはメソッドにもアクセスできます。

ObjectScript

```
 Write x.Salary // displays the Salary property (only available in Employee instances)
```

2.7.2 プライマリ・スーパークラス

サブクラスの拡張元の中で最も左側にあるスーパークラスを、そのサブクラスのプライマリ・スーパークラスと呼びます。クラスはそのプライマリ・スーパークラスのすべてのメンバを継承します。このメンバには、該当のクラス・キーワード、プロパティ、メソッド、クエリ、インデックス、クラス・パラメータ、継承したプロパティと継承したメソッドのパラメータとキーワードなどがあります。Final としてマークされている項目を除き、継承したメンバの特性をサブクラスでオーバーライドできます (削除はされません)。

重要　　インデックス (永続クラスのオプション) は、プライマリ・スーパークラスからのみ継承されます。

2.7.3 多重継承

多重継承によって、クラスの動作やクラス・タイプを複数のスーパークラスから継承できます。多重継承を構築するには、括弧内に複数のスーパークラスをリストします。一番左側にあるスーパークラスがプライマリ・スーパークラスになります。

例えば、クラス X がクラス A、B、C から継承する場合、その定義は以下のようになります。

Class Definition

```
Class X Extends (A, B, C)
{
}
```

クラス・コンパイラの既定の継承順序は左から右なので、スーパークラスどうしでメンバ定義に相違がある場合は、最も左に記述されたスーパークラスを優先することで解決されます (この例では、A が B と C よりも優先し、B が C よりも優先します)。

具体的には、クラス X の場合、クラス・パラメータ値、プロパティ、およびメソッドの値は、クラス A (リストの先頭にあるスーパークラス) から継承され、次にクラス B、最後にクラス C から継承されます。また、X は、A で定義されていないクラス・メンバを B から継承し、A でも B でも定義されていないクラス・メンバを C から継承します。A から既に継承したクラス・メンバと同じ名前のメンバが B にある場合、X では A から継承した値を使用します。同様に、A または B のいずれかから継承したクラス・メンバと同じ名前のメンバが C にある場合は、A、B、C の優先順位で値を使用します。

既定では左から右への継承になるため、継承順序を指定する必要はありません。そのため、前述のクラス定義の例は、次の場合と等しくなります。

Class Definition

```
Class X Extends (A, B, C) [ Inheritance = left ]
{
}
```

スーパークラス間で右から左への継承を指定するには、以下のように Inheritance キーワードを使用して `right` の値を指定します。

Class Definition

```
Class X Extends (A, B, C) [ Inheritance = right ]
{
}
```

右から左への継承では、複数のスーパークラスが同じ名前のメンバを持つ場合、右側のスーパークラスにあるメンバの値が優先します。

注釈　右から左への継承でも、最も左に記述したスーパークラス (最初のスーパークラスと呼ばれることもあります) がプライマリ・スーパークラスであることは変わりません。つまり、サブクラスは最も左にあるスーパークラスからクラス・キーワード値のみを継承します。この動作はオーバーライドされません。

例えば、クラス A、B、および C から X が右から左への継承順序で継承する場合に、クラス A と B からそれぞれ継承したメンバ間に競合があると、既に継承済みのメンバをクラス B のメンバでオーバーライド (置換) します。クラス A と B のメンバに対するクラス C のメンバの関係も同様です。クラス X のクラス・キーワードは独占的にクラス A から継承されます。左から右への継承順序でクラス A を拡張してから B を拡張した結果と、右から左への継承順序でクラス B を拡張してから A を拡張した結果が異なるのはこのためです。いずれの定義でもキーワードは最も左のスーパークラスから継承するので、この 2 つの場合で相違が発生します。

2.7.4 追加のトピック

"%ClassName() および最も適切なタイプのクラス (MSTC)" も参照してください。

2.8 コンパイラ・キーワードの概要

"クラスの定義法：基本" に示したように、クラス定義またはクラス・メンバの定義にキーワードを含めることができます。クラス属性とも呼ばれる、これらのキーワードは、一般にコンパイラに影響を与えます。このセクションでは、一般的なキーワードと、それらのキーワードを InterSystems IRIS が提供する方法について説明します。

2.8.1 例

以下の例では、クラス定義と一般的に使用されるキーワードを示しています。

```
/// This sample persistent class represents a person.
Class MyApp.Person Extends %Persistent [ SqlTableName = MyAppPerson ]
{

/// Define a unique index for the SSN property.
Index SSNKey On SSN [ Unique ];

/// Name of the person.
Property Name As %String [ Required ];

/// Person's Social Security number.
Property SSN As %String(PATTERN = "3N1""-""2N1""-""4N") [ Required ];

}
```

この例では、以下のキーワードを示しています。

- クラス定義では、Extends キーワードがこのクラスの継承元のスーパークラスを指定します。

  別の方法でクラスを表示すると、Extends キーワードの名前が異なるので注意が必要です。"クラス・ドキュメントの作成" を参照してください。

- クラス定義では、SqlTableName キーワードは、既定の名前が使用されない場合に関連付けられたテーブルの名前を決定します。このキーワードは永続クラスに対してのみ意味があります (このドキュメントで後述します)。

- インデックス定義では、Unique キーワードにより、InterSystems IRIS はインデックスが基にしたプロパティ (この例では SSN) に対して一意性を強制します。

- 2 つのプロパティでは、Required キーワードにより、InterSystems IRIS はプロパティに NULL 以外の値を要求します。

PATTERN はキーワードでなく、プロパティ・パラメータです。PATTERN は角括弧ではなく括弧で囲まれる点に注意してください。

ストレージに関連するキーワードは別として (これは、ほとんどドキュメント化されていません)、キーワードの詳細は "クラス定義リファレンス" で調べられます。通常の編集モードでクラスを表示すると、このリファレンス情報には、適用される構文の用例が示されます。

2.9 関連項目

- クラスのコンパイルと配置
- クラス・ドキュメントの作成
- 最上位レベル・クラスの構文とキーワード
- 登録オブジェクトを使用した作業
- 永続オブジェクトの概要

3

クラスのコンパイルと配置

クラスのコンパイルは、IDE（他の箇所で説明）でも、ここで説明するようにプログラムによっても実行できます。ここでは、コンパイラの動作の詳細も説明します。最後に、コンパイルされたクラスを配置モードに入れる方法についても説明します。

3.1 プログラムによるコンパイル

プログラムによってクラスをコンパイルするには、%SYSTEM.OBJ クラスの Compile() メソッドを使用します。

ObjectScript

```
Do $System.OBJ.Compile("MyApp.MyClass")
```

%SYSTEM.OBJ クラスには、複数のクラスのコンパイルに使用できる追加のメソッドがあります。

3.2 クラス・コンパイラの基本

InterSystems IRIS® データ・プラットフォームのクラス定義は、使用する前にコンパイルする必要があります。

クラス・コンパイラは、Java など他のプログラミング言語で使用できるコンパイラとは、2 つの点で大きく異なります。第 1 に、コンパイルの結果は、ファイル・システムではなく、共有リポジトリ（データベース）に保存されます。第 2 に、コンパイラは永続クラスに対するサポートを提供します。

特に、クラス・コンパイラは以下のことを行います。

1. 依存関係のリストを生成します。最初にコンパイルされる必要があるクラスのリストです。使用されているコンパイル・オプションによっては、最後のコンパイル以降に変更された依存関係もコンパイルされます。
2. 継承を決定します。いずれのメソッド、プロパティ、および他のクラス・メンバが、スーパークラスから継承されるかを決定します。この継承情報を、後で参照するためにクラス・ディクショナリに保存します。
3. 永続クラスとシリアル・クラスでは、データベース内のオブジェクトを保存するのに必要なストレージ構造を決定し、クラスの SQL 表現に必要な実行時情報を作成します。
4. クラスで定義（または継承）されているすべてのメソッド・ジェネレータを実行します。

5. クラスに対する実行時コードを含む、1 つまたは複数のルーチンを生成します。クラス・コンパイラは、言語 (ObjectScript と Basic) に応じてメソッドをグループ化し、別個のルーチンを生成します。各ルーチンには、その言語のメソッドが含まれます。
6. 生成されたすべてのルーチンを実行可能なコードにコンパイルします。
7. クラス記述子を作成します。これは、(ルーチンとして保存された) 特別なデータ構造で、クラスのサポートに必要な実行時のすべてのディスパッチ情報 (プロパティの名前、メソッドの位置など) を含みます。

3.3 クラス・コンパイラに関する留意事項

3.3.1 コンパイルの順序

クラスのコンパイル時、コンパイルしているクラスに依存関係に関する情報が含まれている場合は、システムにより、その他のクラスもリコンパイルされます。例えば、システムは、クラスに含まれるすべてのサブクラスをコンパイルします。場合によっては、クラスのコンパイル順序を制御することが必要になります。そのためには、System、DependsOn、および CompileAfter キーワードを使用します。詳細は、"クラス定義リファレンス" を参照してください。

特定のクラスをコンパイルするときに、コンパイラによってリコンパイルされることになるクラスを調べるには、$SYSTEM.OBJ.GetDependencies() メソッドを使用します。以下に例を示します。

```
TESTNAMESPACE>d $system.OBJ.GetDependencies("Sample.Address",.included)

TESTNAMESPACE>zw included
included("Sample.Address")=""
included("Sample.Customer")=""
included("Sample.Employee")=""
included("Sample.Person")=""
included("Sample.Vendor")=""
```

このメソッドのシグニチャは、以下のとおりです。

```
classmethod GetDependencies(ByRef class As %String,
                            Output included As %String,
                            qspec As %String) as %Status
```

以下はその説明です。

- class は、1 つのクラス名 (この例の場合)、クラス名のコンマ区切りリスト、またはクラス名の多次元配列になります。(多次元配列の場合、この引数は必ず参照で渡してください。)ワイルドカードを含めることもできます。
- included は、class がコンパイルされるときに、コンパイルされることになるクラスの名前の多次元配列です。
- qspec は、コンパイラのフラグと修飾子の文字列です。これを省略すると、このメソッドは、現在のコンパイラのフラグと修飾子を使用します。

3.3.2 ビットマップ・インデックスを含むクラスのコンパイル

ビットマップ・インデックスを含むクラスをコンパイルする場合、そのクラスに対するビットマップ・エクステント・インデックスが定義されていないと、クラス・コンパイラはビットマップ・エクステント・インデックスを生成します。プロダクション・システムのクラスにビットマップ・インデックスを追加するときには特別な注意が必要です。詳細は、"ビットマップ・エクステント・インデックスの生成" を参照してください。

3.3.3 メモリにクラスの既存のインスタンスが存在する場合のコンパイル

コンパイル対象のクラスのインスタンスが開いているときにコンパイラを呼び出しても、エラーは発行されません。既に開いているインスタンスでは、その既存のコードが引き続き使用されます。コンパイル後に開いた別のインスタンスでは、新しくコンパイルしたコードが使用されます

3.4 クラスを配置モードに入れる方法

いくつかの独自のクラスを顧客に送信する前に配置モードに入れることができます。この処理によって、ソース・コードが表示されなくなります。

クラス定義に、顧客には見せたくないメソッド定義が含まれている場合は、そのクラスをコンパイルしてから、$SYSTEM.OBJ.MakeClassDeployed() を使用します。以下に例を示します。

ObjectScript

```
do $system.OBJ.MakeClassDeployed("MyApp.MyClass")
```

代替手段については、"顧客のデータベースへのコンパイル済みコードの追加" を参照してください。

3.4.1 配置モードについて

クラスが配置モードになると、そのクラスのメソッド定義とトリガ定義は削除されます。(クラスがデータ型クラスである場合、クエリ・キャッシュによって実行時にメソッド定義が必要になる可能性があるために、メソッド定義が保持されることに注意してください。)

配置モードのクラスは、エクスポートやコンパイルができません。ただし、そのクラスのサブクラスのコンパイルは可能です(配置モードになっていない場合)。

クラスの配置を元に戻す手段はありません。ただし、前もって定義をディスクに保存しておくと、ファイルからその定義をインポートし、クラスを置き換えることができます。(これは、独自のクラスのいずれかを誤って不完全に配置モードにしてしまった場合に役立ちます。)

3.5 関連項目

- クラスの定義
- クラス・ドキュメントの作成
- 最上位レベル・クラスの構文とキーワード

4

クラス・ドキュメントの作成

InterSystems IRIS® データ・プラットフォームには、インターシステムズ・クラス・リファレンスという Web ページが用意されています。このページには、インターシステムズが提供しているクラスと、ユーザが作成するクラスについて、自動的に生成されるリファレンス情報が表示されます。クラス・リファレンスは **%CSP.Documatic** クラスによって生成されるため、非公式には Documatic とも呼ばれています。

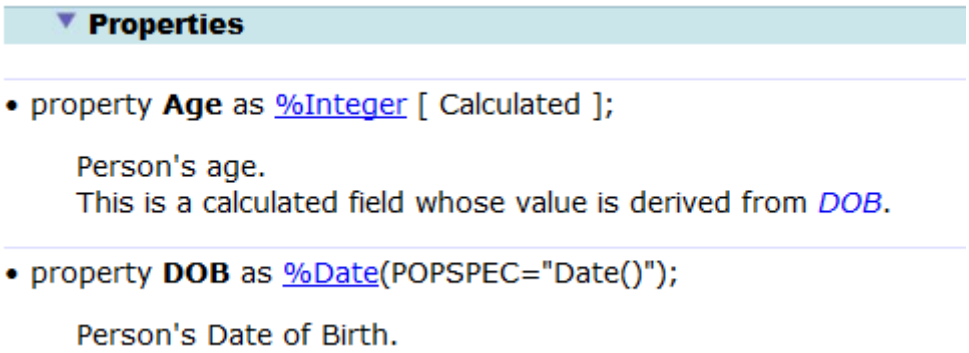
URL の形式は以下のとおりです。〈baseURL〉 はインスタンスのベース URL です。

```
https:<baseURL>/csp/documatic/%25CSP.Documatic.cls
```

この Web ページの外観は、オンラインのクラス・リファレンスとは異なっています (オンラインのクラス・リファレンスは残りのオンライン・ドキュメントとも接続されているため)。

4.1 クラス・リファレンスの概要

クラス・リファレンスの目的は、クラスの使用可能な部分と、その部分の使用方法をプログラマに公表することです。以下に例を示します。



このリファレンス情報には、クラス・メンバの定義が示されていますが、それらの実際の実装ではありません。例えば、メソッド・シグニチャが示されていますが、それらの内部定義ではありません。これには、要素間のリンクが含まれており、コードのロジックを迅速にたどることができます。また、検索オプションも用意されています。

4.2 クラス・リファレンスに含めるドキュメントの作成

クラス・リファレンスに含めるドキュメントを作成するには、クラス定義内にコメントを作成します。コメントは、/// で開始します。このようなコメントの付いたクラス宣言を前に置くと、そのコメントはクラスのページの最上位に示されます。このようなコメントの付いた特定のクラス・メンバを前に置くと、コメントは生成されたそのクラス・メンバの情報の後に示されます。クラスをコンパイルしておくと、次にクラス・リファレンス・ドキュメントを開いたときに、生成されたクラス・ドキュメントを表示できます。クラス・リファレンスのコメントを追加しない場合、クラスやパッケージに追加した項目は、そのクラスやパッケージの内容のリストに正しく表示されますが、それらを説明するテキストは表示されません。

クラス・リファレンスの既存のコメントは、クラス定義を変更することによって拡張できます。クラス・リファレンスのコメントの構文規則は以下のように厳密です。

- クラス・リファレンスの中でクラスまたはクラス・メンバを説明するすべてのコメントは、そのコメントで説明している項目の宣言の直前に、連続するブロックとして記述する必要があります。
- コメントのブロックを構成する各行の先頭には、3 つのスラッシュ (///) を記述します。

  Tipヒント 既定では、この表示は、/// のすべての行のテキストをまとめ、その結果を 1 つの段落として扱うことに注意してください。HTML の改行 (<br>) を挿入できます。または、HTML フォーマット (、 など) を使用できます。サブセクションを参照してください。

- この 3 つのスラッシュは、行の先頭 (左端) に配置する必要があります。
- クラス・リファレンスのコメントに空白行を入れることはできません。
- クラス・リファレンスのコメントの最終行とそのコメントで説明している項目の宣言の間に空白行を入れることはできません。
- クラス・リファレンスのコメントの長さ (すべての行の合計) は、文字列長の制限 (きわめて長い) 未満である必要があります。“文字列長の制限”を参照してください。

クラス・リファレンスのコメントには、プレーン・テキストのほか、あらゆる標準 HTML 要素および少数の専用要素を使用できます。

4.3 クラス・ドキュメントでの HTML マークアップの使用

クラスのコメントでは、HTML タグを使用できます。マークアップでは、可能な限り厳密な HTML 標準 (XHTML など) に従い、どのブラウザでも結果を表示できるようにしてください。%CSP.Documatic クラスは完全な HTML ページを生成し、クラス・ドキュメントはそのページ上の <body> 要素の内部に格納されます。このため、HTML タグ <html>、<body>、<head>、または <meta> をマークアップに含めないでください。これらのタグはすべて無視されます。また、クラス名は <h1> の見出しとして表示されるため、見出しを使用する場合は <h2> 以下の見出しを使用してください。検索エンジンは <h1> が 1 つだけ含まれる HTML ページを優先するため、マークアップ内に <h1> を含めた場合、その見出しは <h2> に変換されます。

標準的な HTML のほかに、<CLASS>、<METHOD>、<PROPERTY>、<PARAMETER>、<QUERY>、および <EXAMPLE> の各タグも使用できます。(標準的な HTML タグと同様に、これらのタグの名前は大文字と小文字を区別しません。)最も一般的に使用するタグについては、ここで説明されています。その他の詳細は、%CSP.Documatic のドキュメントを参照してください。

<CLASS>

クラス名をタグ付けします。クラスが存在する場合、コンテンツはクラスのドキュメントに対するリンクとして表示されます。以下に例を示します。

```
/// This uses the <CLASS>MyApp.MyClass</CLASS> class.
```

<EXAMPLE>

プログラム例をタグ付けします。このタグは、テキストの外観に影響します。それぞれの /// 行は、例の中で独立した行になります (行が単一の段落に結合される通常の場合とは対照的です)。以下に例を示します。

```
/// <EXAMPLE>
/// set o=..%New()
/// set o.MyProperty=42
/// set o.OtherProp="abc"
/// do o.WriteSummary()
/// </EXAMPLE>
```

<METHOD>

メソッド名をタグ付けします。メソッドが存在する場合、コンテンツはメソッドのドキュメントに対するリンクとして表示されます。以下に例を示します。

```
/// This is identical to the <METHOD>Unique</METHOD> method.
```

<PROPERTY>

プロパティ名をタグ付けします。プロパティが存在する場合、コンテンツはプロパティのドキュメントに対するリンクとして表示されます。以下に例を示します。

```
/// This uses the value of the <PROPERTY>State</PROPERTY> property.
```

これは、HTML マークアップを使用した複数行にわたる説明です。

```
/// The <METHOD>Factorial</METHOD> method returns the factorial
/// of the value specified by <VAR>x</VAR>.
```

4.4 関連項目

- クラスの定義

5
パッケージのオプション

ここでは、パッケージについて詳しく説明します。

永続クラスの場合は、パッケージが SQL スキーマとして SQL で表現されます。詳細は、"パッケージからスキーマへのプロジェクション" を参照してください。

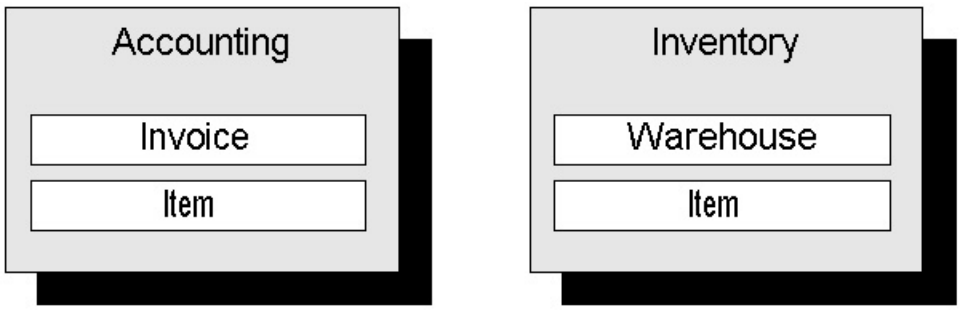
重要　パッケージ名が含まれていないクラスへの参照を InterSystems IRIS® データ・プラットフォームが検出したときに、クラス名の先頭が "%" の場合、InterSystems IRIS は、そのクラスが %Library パッケージに含まれているものと見なします。

5.1 パッケージの概要

InterSystems IRIS は、特定のデータベース内の関連するクラスをグループ化するパッケージをサポートしています。パッケージには、以下の利点があります。

- 大規模なアプリケーションを構築し、他の開発者とコードを共有する簡単な方法を開発者に提供します。
- クラス間で名前が衝突する問題を容易に回避できます。
- オブジェクト・ディクショナリ内の SQL スキーマを、単純明快に表現する論理的な方法を提供します。つまり、1 つのパッケージが 1 つのスキーマに対応します。

パッケージは、関連するクラスを共通の名前の下に 1 つのグループとしてまとめる簡単な方法です。例えば、アプリケーションに Accounting システムと Inventory システムがあるとします。これらのアプリケーションを構成するクラスを、Accounting パッケージと Inventory パッケージに編成できます。



上記のクラスは、いずれも (パッケージ + クラス名から成る) フル・ネームを使用して参照されます。

ObjectScript

```
Do ##class(Accounting.Invoice).Method()
Do ##class(Inventory.Item).Method()
```

パッケージ名がコンテキストから決定される場合は（下記参照）、パッケージ名を省略できます。

ObjectScript

```
Do ##class(Invoice).Method()
```

クラスと同様に、パッケージ定義は、InterSystems IRIS データベース内に存在します。パッケージのデータベースからネームスペースへのマッピングの詳細は、"パッケージ・マッピング" を参照してください。

5.2 パッケージ名

パッケージ名は、文字列です。. (ピリオド) は使用できますが、その他の句読点は使用できません。パッケージ名のピリオドで区切られた各部は、1 つのサブパッケージであり、複数のサブパッケージが存在することもあります。あるクラスに Test.Subtest.TestClass と名前を付けた場合、パッケージ名は Test、サブパッケージ名は Subtest、クラス名は TestClass になります。

パッケージ名の長さと使用法には、以下のような制限があります。

- パッケージ名は長さ制限に従います。"識別子のルールとガイドライン" を参照してください。
- 1 つのネームスペース内では、各パッケージ名が一意になる必要があります（大文字と小文字は区別されません）。つまり、1 つのネームスペース内に “ABC” と “abc” というパッケージが同時に存在することはなく、“abc.def” というパッケージとサブパッケージは、“ABC” パッケージの一部として扱われるということです。

詳細は、"識別子のルールとガイドライン" および "ネームスペースで何にアクセス可能か" を参照してください。

5.3 パッケージの定義

パッケージは、暗にクラス名の意味を含みます。クラスを作成すると、パッケージは自動的に定義されます。同様に、パッケージ内のクラスをすべて削除すると、そのパッケージも自動的に削除されます。

以下に、パッケージ名が `Accounting` でクラス名が `Invoice`、完全修飾クラス名が `Accounting.Invoice` の例を示します。

Class Definition

```
Class Accounting.Invoice
{
}
```

5.4 パッケージ・マッピング

定義では、各パッケージは、特定のデータベースに含まれます。しばしば、各データベースは、ネームスペースに関連付けられ、データベースとネームスペースは、共通の名前を共有します。データベース内のパッケージ定義をそのデー

タベースと関連付けられていないネームスペースで使用できるようにするには、パッケージ・マッピングを使用します。この手順は、"ネームスペースへのマッピングの追加" で説明されています。

パッケージをネームスペースを超えてマップすると、パッケージの定義はマップされますが、そのデータはマップされません。

5.5 クラス参照時のパッケージの使用

クラスの参照には、2 つの方法があります。

- 完全修飾された名前 (すなわち、Package.Class) を使用します。例えば、以下のようになります。

  ObjectScript

  ```
  // create an instance of Lab.Patient
  Set patient = ##class(Lab.Patient).%New()
  ```

- 短いクラス名を使用し、クラスがいずれのパッケージに属するのかをクラス・コンパイラに解決させます。

  既定では、短いクラス名を使用すると、InterSystems IRIS はそのクラスを、使用しているコードを含むクラスのパッケージ (存在する場合)、**%Library** パッケージ、または **User** パッケージのいずれかに含まれるクラスであると見なします。

  コンパイラで別のパッケージのクラスを検索する場合は、それらのパッケージをインポートします。

  注釈　不明確な短いクラス名を使用するとエラーになります。つまり、2 つ以上のパッケージに同じ短いクラス名が付いていて、そのパッケージすべてをインポートする場合、コンパイラがパッケージ名を解決する時点でエラーになります。このようなエラーを避けるためにフル・ネームを使用してください。

5.6 パッケージのインポート

パッケージをインポートするときに、InterSystems IRIS はインポートするパッケージ内で短いクラス名を検索します。クラス定義では、クラスの Import 指示文または ObjectScript の #IMPORT 指示文を使用してパッケージをインポートできます。このセクションでは、これらの指示文について説明し、User パッケージに対する影響と、サブクラスに対する影響について述べ、いくつかのヒントも紹介します。

5.6.1 クラスの Import 指示文

クラス定義の最上位 (`Class` 行の前) に、クラスの Import 指示文を含めることができます。この指示文の構文は、以下のようになります。

```
Import packages
Class name {}
```

packages は、単一のパッケージ、または括弧で囲んだ複数のパッケージのコンマ区切りリストのいずれかにします。`Import` は大文字と小文字を区別しませんが、ここに示すように、通常は最初の文字を大文字にします。

クラス・コンテキストでは、現在のパッケージが常に暗黙的にインポートされます。

5.6.2 ObjectScript の #IMPORT 指示文

ObjectScript メソッドでは、#IMPORT 指示文でパッケージをインポートし、短いクラス名を使用して、パッケージ内のクラスを参照できます。この指示文の構文は、以下のようになります。

```
#import packagename
```

packagename はパッケージの名前です。`#import` は大文字と小文字が区別されません。例えば、以下のようになります。

ObjectScript

```
#import Lab
 // Next line will use %New method of Lab.Patient, if that exists
 Set patient = ##class(Patient).%New()
```

#IMPORT 指示文は複数使用できます。

ObjectScript

```
#import Lab
#import Accounting

 // Look for "Patient" within Lab & Accounting packages.
 Set pat = ##class(Patient).%New()

 // Look for "Invoice" within Lab & Accounting packages.
 Set inv = ##class(Invoice).%New()
```

#IMPORT 指示文の順序は、コンパイラが短いクラス名を解決する方法に影響しません。

5.6.3 明示的なパッケージのインポートによる User パッケージへのアクセスに対する影響

コードによって明示的にパッケージをインポートすると、User パッケージは自動的にインポートされません。そのパッケージが必要な場合は、パッケージも明示的にインポートする必要があります。例えば、以下のようになります。

ObjectScript

```
#import MyPackage
#import User
```

このような論理である理由は、User パッケージをインポートしたくない場合があるからです。

5.6.4 パッケージのインポートおよび継承

クラスはスーパークラスから明示的にインポートされたパッケージを継承します。

クラスの名前は、現在のクラス名ではなく、その名前が最初に使用されたコンテキストで解決されます。例えば、User.MyClass で MyMethod() メソッドを定義してから、User.MyClass を継承する MyPackage.MyClass クラスを作成し、これをコンパイルするとします。この場合、InterSystems IRIS は継承した MyMethod() メソッドを MyPackage.MyClass でコンパイルしますが、このメソッドのクラス名は User.MyClass のコンテキストで解決されます (このメソッドをこのコンテキストで定義しているからです)。

5.6.5 パッケージのインポートのヒント

パッケージをインポートすることで、さらに柔軟性のあるコードを作成できます。例えば、以下のようなコードを作成できます。

ObjectScript

```
#import Customer1
 Do ##class(Application).Run()
```

App.MAC を以下のように変更します。

ObjectScript

```
#import Customer2
 Do ##class(Application).Run()
```

App.MAC をリコンパイルするときには、Customer2.Application クラスを使用します。このようなコードを作成するには計画が必要です。つまり、コードの互換性だけでなく、ストレージ構造への影響も考慮する必要があります。

5.7 関連項目

- パッケージからスキーマへのプロジェクション
- ネームスペースへのマッピングの追加
- クラスの定義

6

クラス・パラメータの定義と参照

ここでは、クラス・パラメータを定義する方法と、そのパラメータをプログラムで参照する方法について説明します。"一般的なプロパティ・パラメータ" も参照してください。

6.1 クラス・パラメータの概要

クラス・パラメータでは、特定のクラスのすべてのオブジェクトで使用できる特別な定数値を定義します。このクラス・パラメータの値は、クラス定義を作成するときに (またはコンパイル前の任意の時点で) 設定できます。既定では、各パラメータの値は NULL 文字列になりますが、パラメータ定義の一環として NULL 以外の値を指定することもできます。コンパイル時に、パラメータの値はクラスのすべてのインスタンスに対して構築されます。わずかな例外はありますが、この値を実行時に変更することはできません。

クラス・パラメータは一般的に、以下の目的で使用されます。

- さまざまなデータ型クラスの振る舞いをカスタマイズするため (検証情報の提供など)。
- クラス定義に関するユーザ指定の情報を定義するため。クラス・パラメータは、単純な任意の名前と値の組み合わせです。この組み合わせを使用して、クラスに関する必要な情報を保存します。
- クラス固有の定数値を定義するため
- 使用するメソッド・ジェネレータ・メソッドに対して、パラメータ化された値を提供するため。メソッド・ジェネレータは、特別なタイプのメソッドで、この実装は実際に、コンパイル時に実行される短いプログラムです。メソッドに対する実際の実行時コードを生成します。メソッド・ジェネレータの多くは、クラス・パラメータを使用します。

6.2 クラス・パラメータの定義

クラス・パラメータをクラス定義に追加するには、以下のいずれかのような要素をクラスに追加します。

```
Parameter PARAMNAME;
Parameter PARAMNAME as Type;
Parameter PARAMNAME as Type = value;
Parameter PARAMNAME as Type [ Keywords ] = value;
```

以下はその説明です。

- PARAMNAME は、パラメータの名前です。慣習により、InterSystems IRIS® データ・プラットフォームのシステム・クラスのパラメータは、ほとんどすべてが大文字になる点に注意してください。この慣習により、その他のクラス・メンバのパラメータを名前だけで簡単に区別できるようになります。同じようにする必要はありません。
- Type はパラメータのタイプです。次のセクションを参照してください。
- value は、パラメータの値です。ほとんどの場合、これは、100 や "MyValue" などのリテラル値になります。いくつかのタイプでは、ObjectScript 式にすることもできます。次のセクションを参照してください。
- Keywords はパラメータ・キーワードを表します。これらはオプションです。キーワードの概要は、"コンパイラ・キーワードの概要" を参照してください。パラメータ・キーワードの詳細は、"パラメータの構文とキーワード" を参照してください。

6.3 パラメータのタイプと値

パラメータのタイプは必ずしも指定する必要はありません。指定する場合、この情報は主に開発環境での使用が意図されることに注意してください。

パラメータのタイプには、BOOLEAN、STRING、INTEGER などがあります。これらは、InterSystems IRIS のクラス名ではありません。"一般的なプロパティ・パラメータ" を参照してください。

コンパイラはパラメータのタイプを無視します。ただし、COSEXPRESSION タイプと CONFIGVALUE タイプを除きます。

6.3.1 コンパイル時に評価されるクラス・パラメータ (中括弧)

コンパイル時に評価される ObjectScript 式として、パラメータを定義できます。そのためには、以下のようにタイプを指定せずに、値を中括弧で囲んで指定します。

Class Member

```
Parameter PARAMNAME = {ObjectScriptExpression};
```

PARAMNAME は定義するパラメータです。ObjectScriptExpression はコンパイル時に評価される ObjectScript 式です。この式では %classname 変数を使用できますが、使用しない場合は、現在のクラスまたはそのメンバを参照することはできません。この式は、このクラスよりも前にコンパイルされている場合に限り、他のクラスのコードを呼び出すことができます。

例 :

Class Member

```
Parameter COMPILETIME = {$zdatetime($h)};
```

6.3.2 実行時に評価されるクラス・パラメータ (COSEXPRESSION)

実行時に評価される ObjectScript 式として、パラメータを定義できます。そのためには、以下のように COSEXPRESSION としてタイプを指定し、値として ObjectScript 式を指定します。

Class Member

```
Parameter PARAMNAME As COSEXPRESSION = "ObjectScriptExpression";
```

PARAMNAME は定義するパラメータです。ObjectScriptExpression は実行時に評価される ObjectScript コンテンツです。

以下にクラス・パラメータ定義の例を示します。

Class Member

```
Parameter DateParam As COSEXPRESSION = "$H";
```

6.3.3 実行時に更新されるクラス・パラメータ (CONFIGVALUE)

クラス定義の外側で変更できるように、パラメータを定義できます。そのためには、CONFIGVALUE としてタイプを指定します。この場合、$SYSTEM.OBJ.UpdateConfigParam() メソッドによってパラメータを変更できます。例えば、以下のようにして、MyApp.MyClass クラスのパラメータ MYPARM の値を新しい値の 42 に変更します。

```
set sc=$system.OBJ.UpdateConfigParam("MyApp.MyClass","MYPARM",42)
```

$SYSTEM.OBJ.UpdateConfigParam() は、新しいプロセスで使用される生成済みのクラス記述子に影響しますが、クラス定義には影響しません。クラスをリコンパイルすると、InterSystems IRIS により、クラス記述子が再生成され、クラス定義に記載されたようにこのパラメータの値を使用します。そのため、$SYSTEM.OBJ.UpdateConfigParam() によって行った変更は上書きされます。

6.4 クラスのパラメータの参照

クラスのパラメータを参照するには、以下のいずれかを実行できます。

- 関連するクラスのメソッド内で、以下の式を使用します。

  ```
  ..#PARMNAME
  ```

  この式は、DO コマンドおよび SET コマンドと共に使用することも、別の式の一部として使用することもできます。以下に一例を示します。

  ```
  set object.PropName=..#PARMNAME
  ```

  その次のバリエーションでは、クラス内のメソッドでパラメータの値を確認し、その値を後続の処理の制御に使用します。

  ```
  if ..#PARMNAME=1 {
    //do something
  } else {
    //do something else
  }
  ```

- クラス内のパラメータにアクセスするには、以下の式を使用します。

  ```
  ##class(Package.Class).#PARMNAME
  ```

  Package.Class はクラスの名前、PARMNAME はパラメータの名前です。

  この構文では、指定のクラス・パラメータにアクセスして、その値を返します。この式は、DO コマンドや SET コマンドなどと共に使用することも、別の式の一部として使用することもできます。以下に例を示します。

  ObjectScript

  ```
  w ##class(%XML.Adaptor).#XMLENABLED
  ```

  この例では、XML アダプタによって生成されたメソッドが XML 対応かどうかが表示されます。既定の設定は 1 です。

- 実行時までパラメータ名が決定されないパラメータにアクセスするには、以下のように $PARAMETER 関数を使用します。

  ```
  $PARAMETER(classnameOrOref,parameter)
  ```

  classnameOrOref は、クラスの完全修飾名か、クラスのインスタンスの OREF です。また、parameter は関連するクラスのパラメータ名に評価されます。

  OREF の詳細は、"登録オブジェクトを使用した作業" を参照してください。

6.5 関連項目

- パラメータの構文とキーワード
- 一般的なプロパティ・パラメータ
- クラスの定義

7

メソッドの定義と呼び出し

ここでは、InterSystems IRIS® データ・プラットフォームのクラスのメソッドを作成する際の規則とオプションについて説明します。また、そのようなメソッドを呼び出す際の規則とオプションについても説明します。

インスタンス・メソッドはオブジェクト・クラスにのみ適用され、ここでは説明していません。

7.1 メソッドの概要

メソッドとは、クラスによって定義される実行可能な要素です。InterSystems IRIS は、インスタンス・メソッドとクラス・メソッドの 2 種類のメソッドをサポートしています。インスタンス・メソッドは、クラスの特定のインスタンスから呼び出され、通常はそのインスタンスに関連するアクションを実行します。クラス・メソッドとは、オブジェクト・インスタンスへの参照なしに呼び出せるメソッドのことです。その他の言語では、静的メソッドと呼ばれています。

通常、メソッドは、インスタンス・メソッドを指します。それよりも具体的な表現のクラス・メソッドは、クラス・メソッドを表すために使用します。

インスタンス・メソッドは、オブジェクトのインスタンスが存在しないと実行できません。そのため、インスタンス・メソッドは、オブジェクト・クラスで定義されている場合にのみ役立ちます。これに対して、クラス・メソッドは、どの種類のクラスで定義してもかまいません。

7.2 メソッドの定義

クラスにクラス・メソッドを追加するには、クラス定義に以下に示すような要素を追加します。

```
ClassMethod MethodName(Arguments) as Classname [ Keywords]
{
//method implementation
}
```

以下はその説明です。

- MethodName は、メソッドの名前です。ルールについては、"名前付け規約" を参照してください。
- Arguments は、引数のコンマ区切りリストです。詳細は、"メソッドの引数の指定: 基本" を参照してください。
- Classname は、このメソッドで返される値 (ある場合) のタイプを表す、オプションのクラス名です。メソッドが値を返さない場合は、As Classname 部分を省略します。

クラスは、データ型クラス、オブジェクト・クラス、またはタイプのないクラス（あまり一般的ではありません）になります。クラス名は完全なクラス名か、短いクラス名になります。詳細は、"クラス参照時のパッケージの使用"を参照してください。

- Keywords はメソッド・キーワードを表します。これはオプションです。"コンパイラ・キーワードの概要"を参照してください。
- メソッドの実装は、メソッドの実装言語とタイプによって異なります。"実装言語の指定" および "メソッドのタイプ" を参照してください。既定では、メソッドの実装はゼロ行以上の ObjectScript で構成されます。

クラスにインスタンス・メソッドを追加するには、同じ構文で `ClassMethod` の代わりに `Method` を使用します。

```
Method MethodName(arguments) as Classname [ Keywords]
{
 //method implementation
}
```

インスタンス・メソッドは、オブジェクト・クラスでのみ存在価値があります。

7.3 メソッドの引数の指定: 基本

メソッドは、任意の数の引数を取ることができます。メソッドの定義では、メソッドが取る引数を指定する必要があります。また、各引数のタイプと既定値も指定できます。(この場合のタイプは、あらゆる種類のクラスを指します。特にデータ型クラスを指すわけではありません。)

以下の汎用クラスのメソッド定義について考えてみます。

```
ClassMethod MethodName(Arguments) as Classname [ Keywords]
{
 //method implementation
}
```

Arguments は、以下に示す一般的な形式になります。

```
argname1 as type1 = value1, argname2 as type2 = value2, argname3 as type3 = value3, [and so on]
```

以下はその説明です。

- argname1、argname2、argname3 などは、引数の名前です。これらの名前は、変数名の規則に従う必要があります。
- type1、type2、type3 などは、クラス名です。メソッド定義のこの部分は、このメソッドを使用することになるプログラマに向けて、どのタイプの値を対応する引数に渡すかについて説明することを目的としています。通常は、各メソッド引数のタイプを明示的に指定することをお勧めします。

  一般に、このタイプはデータ型クラスかオブジェクト・クラスになります。

  クラス名は完全なクラス名か、短いクラス名になります。詳細は、"クラス参照時のパッケージの使用"を参照してください。

  構文のこの部分は省略できます。その場合は、`as` 部分も省略します。
- value1、value2、value3 などは、引数の既定値です。引数に値が指定されていない状態でメソッドが呼び出されると、引数はメソッドでこの値に自動的に設定されます。

  各値は、リテラル値（`"abc"` や 42）、または中括弧で囲まれた ObjectScript 式のどちらかにできます。例えば、以下のようになります。

Class Member

```
ClassMethod Test(flag As %Integer = 0)
{
 //method implementation
}
```

別の例を示します。

Class Member

```
ClassMethod Test(time As %Integer = {$horolog} )
{
 //method implementation
}
```

構文のこの部分は省略できます。その場合は、等号 (=) も省略します。

例えば、3 つの引数を取る Calculate() メソッドを考えてみます。

Class Member

```
ClassMethod Calculate(count As %Integer, name, state As %String = "CA")
{
 // ...
}
```

ここで、count と state はそれぞれ %Integer、%String として宣言されます。引数のデータ型の既定は %String なので、タイプが指定されていない場合は %String になります。上記では、name がその例です。

7.4 引数の渡し方の指示

メソッド定義では、このメソッドを使用することになるプログラマに向けて、各引数の予期された渡し方についても指示します。引数は、値 (既定の方法) または参照によって渡すことができます。"メソッドに引数を渡す方法" を参照してください。

特定の変数を参照で渡すことが適切な場合と、適切でない場合があります。詳細はメソッドの実装によって異なります。そのため、メソッドを定義するときには、メソッド・シグニチャを使用して、他のプログラマに各パラメータの用途について指示する必要があります。

引数を参照で渡す必要があることを指示するには、引数名の前方のメソッド・シグニチャに ByRef 修飾子を含めます。両方の引数に ByRef を使用する例を以下に示します。

Class Member

```
/// Swap value of two integers
Method Swap(ByRef x As %Integer, ByRef y As %Integer)
{
    Set temp = x
    Set x = y
    Set y = temp
}
```

同様に、引数を参照で渡す必要があることに加えて、値を受け取らないことを指示するには、引数名の前方のメソッド・シグニチャに Output 修飾子を含めます。以下に例を示します。

Class Member

```
Method CreateObject(Output newobj As MyApp.MyClass) As %Status
{
    Set newobj = ##class(MyApp.MyClass).%New()
    Quit $$$OK
}
```

7.5 可変個数の引数の指定

可変個数の引数を受け入れるメソッドを定義できます。これを行うには、以下の例に示すように、最後の引数の名前の後ろに ... を含めます。この例では、この機能がどのように使用されるかも示しています。

Class Member

```
ClassMethod MultiArg(Arg1... As %String)
{
    Set args = $GET(Arg1, 0)
    Write "Invocation has ",
        args,
        " element",
        $SELECT((args=1):"", 1:"s"), !
    For i = 1 : 1 : args {
        Write "Argument[", i , "]:", ?15, $GET(Arg1(i), "<NULL>"), !
    }
}
```

以下のターミナル・セッションで、このメソッドの動作を示します。

```
MYNAMESPACE>do ##class(VarNumArg.Demo).MultiArg("scooby","shaggy","velma","daphne","fred")
Invocation has 5 elements
Argument[1]:   scooby
Argument[2]:   shaggy
Argument[3]:   velma
Argument[4]:   daphne
Argument[5]:   fred
```

この ObjectScript 機能の詳細は、"可変の引数" を参照してください。

7.6 値を返す方法

値を返すようにメソッドを定義するには、そのメソッド内で以下のいずれかを使用します (ObjectScript でメソッドを実装する場合)。

```
Return returnvalue
```

または

```
Quit returnvalue
```

returnvalue は、このメソッドが返す適切な値です。これは、宣言したメソッド返りタイプと一致する必要があります。返りタイプがデータ型クラスの場合、メソッドはリテラル値を返す必要があります。返りタイプがオブジェクト・クラスの場合、メソッドは、そのクラスのインスタンス (具体的には、OREF) を返す必要があります。詳細は、"登録オブジェクトを使用した作業" を参照してください。

例 :

Class Member

```
ClassMethod Square(input As %Numeric) As %Numeric
{
    Set returnvalue = input * input
    Return returnvalue
}
```

もう 1 つ例を挙げると、このメソッドはオブジェクト・インスタンスを返します。

Class Member

```
ClassMethod FindPerson(id As %String) As Person
{
    Set person = ##class(Person).%OpenId(id)
    Return person
}
```

値を返す構文は、メソッドの実装言語によって異なります。

7.7 特権チェックを使用したアクセスの制限

Requires キーワードを使用して、メソッドへのアクセスを制限できます。指定された特権を持っているユーザまたはプロセスのみがメソッドを呼び出すことができます。

特権を指定するには、リソースの名前と適切なレベルの許可 (Use、Read、または Write) をコロンで区切って指定します。複数の特権を指定するには、コンマ区切りリストを使用します。

以下の例では、MyAction メソッドは Service_FileSystem リソースに対する Use 許可を必要とします。

Class Member

```
ClassMethod MyAction() [ Requires = Service_FileSystem: Use ]
{
  write "You have access to this method."
}
```

必要な特権がない場合にこのメソッドを呼び出すと、<PROTECT> エラーが発生します。

```
<PROTECT> *Method MyAction' requires resource 'Service_FileSystem: Use'
```

メソッドがスーパークラスから Requires キーワードを継承する場合、キーワードに新しい値を設定して、必要な特権のリストに追加できます。この方法で必要な特権を削除することはできません。

詳細は、"特権および許可" を参照してください。また、"Requires" キーワードのリファレンス情報も参照してください。

7.8 実装言語の指定

メソッドを作成するときには、実装言語を選択できます。実際には、1 つのクラス内で、複数のメソッドを異なる言語で実装することもできます。すべてのメソッドは、実装言語とは関係なく相互運用します。

既定では、メソッドは、そのメソッドが属するクラスの Language キーワードで指定された言語を使用します。このキーワードについては、既定が `objectscript` (ObjectScript) になります。もう 1 つのオプションは `tsql` (TSQL) です。

これは、そのメソッドの Language キーワードを以下のように設定することで、特定のメソッドごとにオーバーライドできます。

Class Definition

```
Class MyApp.Test {

Method TestC() As %Integer [ Language = objectscript ]
{
    // This is ObjectScript
    Write "This is a test"
    Quit 1
}
}
```

7.9 メソッドのタイプ (CodeMode オプション)

InterSystems IRIS では、クラス・コンパイラによる処理が異なる 4 つのタイプのメソッドをサポートしています。

- コード・メソッド (既定であり最も一般的)
- 式メソッド
- 呼び出しメソッド
- メソッド・ジェネレータ

7.9.1 コード・メソッド

コード・メソッドは、その実装が単純なコード行であるメソッドです。これは最も一般的なメソッドのタイプで、既定です。

メソッド実装には、実装言語に応じた有効なコードを含めることができます。

注釈　InterSystems IRIS には、多用する簡単なタスクを実行する一連のシステム定義メソッドが用意されています。ユーザ定義メソッドでこれらのタスクのいずれかを実行するようにした場合、コンパイラではそのユーザ定義メソッドの実行可能コードが生成されません。その代わり、このようなユーザ定義メソッドは該当のシステム定義メソッドに関連付けられます。このユーザ定義メソッドを呼び出すと、関連付けられたシステム定義メソッドが呼び出されるので、パフォーマンスの向上が望めます。また、デバッガはそのようなシステム定義メソッドのステップ実行を行いません。

7.9.2 式メソッド

式メソッドは、ある特定の状況で、クラス・コンパイラによって、指定された式の直接インライン代替メソッドに置換される場合があるメソッドです。式メソッドは一般的に、高速な実行を必要とする (データ型クラス内の) 単純なメソッドに使用されます。

例えば、前の例の Dog クラスの Speak() メソッドを、式メソッドに変換することが可能です。

Class Member

```
Method Speak() As %String [CodeMode = expression]
{
    "Woof, Woof"
}
```

dog が Dog オブジェクトを参照しているとすると、以下のようにこのメソッドを使用できます。

ObjectScript

```
   Write dog.Speak()
```

これは、以下のコードを生成する結果となります。

ObjectScript

```
Write "Woof, Woof"
```

式メソッドのすべての仮変数に、既定値を与えることをお勧めします。これは、実行時に実際の変数が見つからないことで発生する、潜在的なインライン置換問題を防ぎます。

注釈　InterSystems IRIS は、式メソッド内のマクロや参照渡し引数を許可していません。

7.9.3 呼び出しメソッド

呼び出しメソッドは、既存の InterSystems IRIS ルーチンの周囲のメソッド・ラッパを作成する、特別なメカニズムです。これは一般的に、従来のコードをオブジェクト・ベースのアプリケーションに移行するときに便利です。

メソッドを呼び出しメソッドとして定義すると、メソッドが呼び出されるときは常に、指定されたルーチンが呼び出されます。呼び出しメソッドの構文は以下のとおりです。

Class Member

```
Method Call() [ CodeMode = call ]
{
    Tag^Routine
}
```

Tag^Routine は、ルーチン内のタグ名を指定します。

7.9.4 メソッド・ジェネレータ

メソッド・ジェネレータは実際には、クラスのコンパイル中にクラス・コンパイラによって呼び出される小さなプログラムです。この出力は、メソッドの実際の実行時実装です。メソッド・ジェネレータは強力なクラス継承の方法を提供し、クラスやプロパティ継承の必要性に応じてカスタマイズされた、高性能で特別なコードを生成します。InterSystems IRIS ライブラリ内で、メソッド・ジェネレータは、データ型やストレージ・クラスによって広範囲に使用されます。

詳細は、"メソッド・ジェネレータとトリガ・ジェネレータの定義" を参照してください。

7.10 メソッドを SQL ストアド・プロシージャとして投影する方法

クラス・メソッド (インスタンス・メソッドを除く) は、SQL ストアド・プロシージャとしても使用できるように定義できます。そのためには、メソッド定義に SqlProc キーワードを含めます。

プロシージャの既定の名前は CLASSNAME_METHODNAME になります。別の名前を指定するには、SqlName キーワードを指定します。

詳細は、"ストアド・プロシージャの定義" を参照してください。

7.11 クラス・メソッドの呼び出し

このセクションでは、ObjectScript のクラス・メソッドを呼び出す方法について説明します。このセクションは、あらゆる種類のクラスに適用されます。インスタンス・メソッドはオブジェクト・クラスにのみ適用され、ここでは説明していません。

- 任意のクラスのクラス・メソッドを呼び出すには (そのメソッドがプライベートでない場合)、以下の式を使用します。

  ```
  ##class(Package.Class).Method(Args)
  ```

  Package.Class はクラスの名前、Method はメソッドの名前、Args はメソッドの引数です。

  ##class は、大文字と小文字を区別しません。

  この式は指定されたクラス・メソッドを呼び出し、その返り値を取得します (ある場合)。この式は、DO コマンドや SET コマンドなどと共に使用することも、別の式の一部として使用することもできます。以下に、いくつかのバリエーションを示します。

  ObjectScript

  ```
  do ##class(Package.Class).Method(Args)
  set myval= ##class(Package.Class).Method(Args)
  write ##class(Package.Class).Method(Args)
  set newval=##class(Package.Class).Method(Args)_##class(Package2.Class2).Method2(Args)
  ```

  パッケージは省略できます。その場合は、使用に適したパッケージ名がクラス・コンパイラによって判断されます (この名前解決は、"パッケージ" で説明されています)。

- (クラス・メソッド内で) 以下の式を使用して、そのクラスの別のクラス・メソッド (継承されたメソッドでもかまいません) を呼び出します。

  ```
  ..MethodName(args)
  ```

  この式は DO コマンドと共に使用できます。メソッドが値を返す場合は、SET を使用することも、別の式の一部として使用することもできます。以下に、いくつかのバリエーションを示します。

  ```
  do ..MethodName()
  set value=..MethodName(args)
  ```

  注釈　この構文をクラス・メソッドで使用してプロパティやインスタンス・メソッドを参照することはできません。これらを参照するにはインスタンス・コンテキストが必要になります。

- 実行時までメソッド名が決定されないクラス・メソッドを実行するには、以下のように $CLASSMETHOD 関数を使用します。

  ```
  $CLASSMETHOD(classname, methodname, Arg1, Arg2, Arg3, ... )
  ```

  classname は、クラスの完全修飾名に評価されます。methodname は、そのクラス内のクラス・メソッドの名前に評価されます。また、Arg1、Arg2、Arg3 などは、そのメソッドへの引数になります。以下に例を示します。

  ObjectScript

  ```
  set cls="Sample.Person"
  set clsmeth="PrintPersons"
  do $CLASSMETHOD(cls,clsmeth)
  ```

  この例では、Sample.Person クラスから PrintPersons メソッドが実行されます。

  詳細は、"$CLASSMETHOD" を参照してください。

指定されたメソッドが存在しない場合や、そのメソッドがインスタンス・メソッドの場合は、システムによって <METHOD DOES NOT EXIST> エラーが生成されます。指定されたメソッドがプライベートの場合は、システムによって <PRIVATE METHOD> エラーが生成されます。

7.11.1 メソッドに引数を渡す方法

メソッドに引数を渡す既定の方法は、値渡しです。この方法では、前述の例に示したように、変数、リテラル値、またはその他の式として、メソッド呼び出しに単に引数を含めます。

また、引数を参照で渡すことも可能です。

その動作を説明します。システムには、各ローカル変数の値を格納するためのメモリ位置があります。変数の名前は、メモリ位置のアドレスとして機能します。メソッドにローカル変数を渡すときには、値で変数を渡します。つまり、システムによってその値のコピーが作成され、元の値が変更されることはありません。その代わりに、メモリ・アドレスを渡すことができます。この方法は、参照渡しと呼ばれています。これは、引数として多次元配列を渡す唯一の方法でもあります。

ObjectScript の場合、引数を参照で渡すには、その引数の前にピリオドを置きます。以下に例を示します。

ObjectScript

```
 set MyArg(1)="value A"
 set MyArg(2)="value B"
 set status=##class(MyPackage.MyClass).MyMethod(.MyArg)
```

この例では、値（多次元配列）を参照で渡すことで、その値をメソッドが受け取れるようにしています。その他の場合でも、引数を参照で渡すことが役立ちます。これは、メソッドの実行後に、その値を使用できるようになるためです。以下に例を示します。

```
 set status=##class(MyPackage.MyClass).GetList(.list)
 //use the list variable in subsequent logic
```

その他の場合、変数に値を指定し、その値を変更する（その値を参照で返す）メソッドを呼び出して、変更された値を使用することもできます。

7.12 メソッドのキャスト

あるクラスのメソッドを、別のクラスのメソッドとしてキャストするには、以下のどちらかの構文を使用します（ObjectScript の場合）。

```
Do ##class(Package.Class1)Class2Instance.Method(Args)
Set localname = ##class(Package.Class1)Class2Instance.Method(Args)
```

クラス・メソッドとインスタンス・メソッドの両方ともキャストできます。

例えば、2 つのクラス MyClass.Up と MyClass.Down が、両方とも Go() メソッドを持つとします。MyClass.Up の場合は、このメソッドが以下のようになります。

Class Member

```
Method Go()
{
    Write "Go up.",!
}
```

MyClass.Down の場合は、Go() メソッドが以下のようになります。

Class Member

```
Method Go()
{
    Write "Go down.",!
}
```

ユーザは、MyClass.Up のインスタンスを生成することができ、これを使用して MyClass.Down.Go メソッドを呼び出します。

```
>Set LocalInstance = ##class(MyClass.Up).%New()

>Do ##class(MyClass.Down)LocalInstance.Go()
Go down.
```

以下のように、式の一部に ##class を使用することもできます。

ObjectScript

```
 Write ##class(Class).Method(args)*2
```

変数を、返り値と同じ値に設定する必要はありません。

より一般的な方法は、インスタンスには $METHOD 関数を使用し、クラス・メソッドには $CLASSMETHOD 関数を使用することです。

7.13 継承されたメソッドの上書き

クラスは、その 1 つまたは複数のスーパークラスからメソッド (クラスおよびインスタンス・メソッドの両方) を継承します。Final の指定があるメソッドを除くと、それらの定義は、このクラス内に定義を用意することでオーバーライドできます。その場合は、以下の規則に注意してください。

- メソッドがスーパークラスのクラス・メソッドの場合、サブクラスでインスタンス・メソッドとしてオーバーライドできません (逆の場合も同じです)。
- サブクラス・メソッドの返り値の型は、元の返り値の型または元の返り値の型のサブクラスのいずれかと同じでなければなりません。
- サブクラスのメソッドは、スーパークラスのメソッドより多くの引数を持つことができます。(さらに、"引数の数" も参照してください。)
- サブクラス内のメソッドは、引数のための異なる既定値を指定できます。
- サブクラス・メソッド内の引数の型は、元のメソッドの引数の型と一致していなければなりません。特に、指定するすべての引数は、元の型または元の型のサブクラスのいずれかと同じでなければなりません。

  引数に指定された型がない場合には、コンパイラは引数を %String として扱うことに注意してください。したがって、スーパークラスのメソッドの引数に型がない場合、サブクラスのメソッドの対応する引数は、%String になるか、%String のサブクラスになるか、または型なしになる可能性があります。
- サブクラスのメソッドは、スーパークラスのメソッドと同じ方法で引数値を受け取る必要があります。例えば、指定した引数がスーパークラス内で参照によって渡される場合、同じ引数をサブクラス内でも参照によって渡す必要があります。

  この点でメソッド・シグニチャに整合性がないと、他の開発者がメソッドの適切な使用方法を知ることは難しくなります。ただし、コンパイラはエラーを発行しないことに注意してください。

メソッドの実装が、スーパークラスで定義されているメソッドと同じ名前のメソッドを呼び出す必要がある場合、##super() 構文を使用できます。これについては、サブセクションで説明します。この説明は、ObjectScript で記述されたコードに適用されます。

7.13.1 ##super()

メソッド内で以下の式を使用して、最も近接しているスーパークラスで定義されたメソッドと同じ名前のメソッドを呼び出します。

```
##super()
```

この式は DO コマンドと共に使用できます。メソッドが値を返す場合は、SET を使用することも、別の式の一部として使用することもできます。以下に、いくつかのバリエーションを示します。

```
 do ##super()
 set returnvalue=##super()_"additional string"
```

注釈　　##super は、大文字と小文字を区別しません。

これは、スーパークラスの既存のメソッドを呼び出して、いくつかの追加手順を実行 (そのメソッドから返された値の変更など) する必要のあるメソッドを定義する場合に役立ちます。

7.13.2 ##super と メソッドの引数

##super は、引数を許可するメソッドでも機能します。サブクラス・メソッドで引数の既定値を指定していない場合、スーパークラスへの参照で引数をメソッドが渡していることを確認してください。

例えば、スーパークラス (MyClass.Up.SelfAdd()) のメソッドのコードが以下のとおりだとします。

Class Member

```
ClassMethod SelfAdd(Arg As %Integer)
{
    Write Arg,!
    Write Arg + Arg
}
```

出力は、以下のようになります。

```
>Do ##Class(MyClass.Up).SelfAdd(2)
2
4
>
```

サブクラス (MyClass.Down.SelfAdd()) のメソッドでは、##super を使用し、引数を参照で渡します。

Class Member

```
ClassMethod SelfAdd(Arg1 As %Integer)
{
    Do ##super(.Arg1)
    Write !
    Write Arg1 + Arg1 + Arg1
}
```

出力は、以下のようになります。

```
>Do ##Class(MyClass.Down).SelfAdd(2)
2
4
6
>
```

MyClass.Down.SelfAdd() では、引数名の前にあるピリオドに注目してください。これを省略して、引数を指定せずにメソッドを呼び出すと、<UNDEFINED> エラーを受け取ることになります。

7.13.3 ##super が作用する呼び出し

##super は、現在のメソッド・コールにのみ作用します。そのメソッドで他の呼び出しを実行する場合、それらの呼び出しは、スーパークラスに対してではなく、現在のオブジェクトまたはクラスに対して実行されます。例えば、以下のように、MyClass.Up に MyName() メソッドと CallMyName() メソッドがあるとします。

```
Class MyClass.Up Extends %Persistent
{

ClassMethod CallMyName()
{
    Do ..MyName()
}

ClassMethod MyName()
{
    Write "Called from MyClass.Up",!
}

}
```

また、以下のように、MyClass.Down で上記のメソッドをオーバーライドするとします。

```
Class MyClass.Down Extends MyClass.Up
{

ClassMethod CallMyName()
{
    Do ##super()
}

ClassMethod MyName()
{
    Write "Called from MyClass.Down",!
}

}
```

ここで、CallMyName() メソッドを呼び出すと、以下のような結果が返されます。

```
USER>d ##class(MyClass.Up).CallMyName()
Called from MyClass.Up

USER>d ##class(MyClass.Down).CallMyName()
Called from MyClass.Down
```

MyClass.Down.CallMyName() の出力は、MyClass.Up.CallMyName() とは異なっています。これは、MyClass.Down.CallMyName() の CallMyName() メソッドには ##super が含まれており、MyClass.Up.CallMyName() メソッドを呼び出した後、キャストされない MyClass.Down.MyName() メソッドを呼び出しているからです。

7.13.4 引数の数

場合によっては、新しい引数をスーパークラス内のメソッドに追加して、サブクラス内のメソッドで定義されているよりも引数の数を多くすることが必要な場合もあります。コンパイラが (便宜上の理由で) サブクラス内のメソッドに引数を追加するために、サブクラスは引き続きコンパイルします。たいていの場合、メソッドを拡張するすべてのサブクラスを調べ、追加の引数を構成するシグニチャを編集し、コードを編集するかどうかを決定することも引き続き必要です。シグニチャまたはコードを編集しない場合でも、次の 2 つの点を考慮する必要があります。

- 追加の引数名が、サブクラス内のメソッドで使用されているどの変数の名前とも同じでないことを確認します。コンパイラは追加された引数をサブクラス内のメソッドに追加します。これらの引数が、サブクラスのメソッドで使用されている変数と同じ名前を持っていると、予期しない結果が生じることがあります。
- サブクラス内のメソッドが (そのメソッドが ##super を使用しているために) 追加の引数を使用する場合は、スーパークラス内のメソッドが追加の引数に対して既定値を指定していることを確認します。

7.14 関連項目

- クラスの定義
- メソッドの構文とキーワード

8

登録オブジェクトを使用した作業

%RegisteredObject クラスは、InterSystems IRIS® データ・プラットフォームの基本オブジェクト API です。ここでは、この API の使用方法について説明します。このトピックの情報は、%RegisteredObject のすべてのサブクラスに適用されます。

8.1 オブジェクト・クラスの概要

オブジェクト・クラス とは、%RegisteredObject を継承するあらゆるクラスのことです。オブジェクト・クラスでは、以下のことが実行できます。

- クラスのインスタンスの作成。これに該当するインスタンスをオブジェクトと呼びます。
- これに該当するオブジェクトのプロパティの設定。
- これに該当するオブジェクトのメソッド (インスタンス・メソッド) の呼び出し。

これらのタスクはオブジェクト・クラスでのみ可能です。

%Persistent クラスおよび %SerialObject クラスは、%RegisteredObject のサブクラスです。概要は、"オブジェクト・クラス" を参照してください。

8.2 OREF の基本

オブジェクトを作成すると、そのオブジェクトについての情報を保持するメモリ内構造がシステムによって作成されます。また、その構造へのポインタである OREF (オブジェクト参照) も作成されます。

オブジェクト・クラスには、OREF を作成する複数のメソッドがあります。どのようなオブジェクト・クラスを使用するときにも、広範囲にわたって OREF を使用することになります。これを使用してオブジェクトのプロパティ値を指定し、オブジェクトのプロパティ値にアクセスし、オブジェクトのインスタンス・メソッドを呼び出します。以下の例を考えてみます。

```
MYNAMESPACE>set person=##class(Sample.Person).%New()
MYNAMESPACE>set person.Name="Carter,Jacob N."
MYNAMESPACE>do person.PrintPerson()
Name: Carter,Jacob N.
```

最初の手順では、Sample.Person という名前のクラスの %New() メソッドを呼び出しています。このメソッドは、オブジェクトを作成して、そのオブジェクトをポイントする OREF を返します。この OREF と等しくなるように、変数 person を設定します。その次の手順では、オブジェクトの Name プロパティを設定しています。3 番目の手順では、オブジェクトの PrintPerson() インスタンス・メソッドを呼び出しています。(Name プロパティと PrintPerson() メソッドは、どちらも単なる例です。これらは Sample.Person クラスで定義されていますが、汎用オブジェクト・インタフェースの一部ではありません。)

OREF は一時的なものです。その値はオブジェクトがメモリ内にあるときにだけ存在し、呼び出しのたびに値が同じになることは保証されていません。

注意　OREF はそれが作成されたネームスペース内でのみ有効です。したがって、既存の OREF が存在しているときに現在のネームスペースを変更すると、前のネームスペースからの OREF は無効になります。別のネームスペースからの OREF を使用してみたときに、即時にエラーが発生しないことがありますが、その結果から OREF が有効または使用可能であると見なすことはできず、現行のネームスペースでは深刻な状況が発生する可能性があります。

8.2.1 INVALID OREF エラー

単純な式で、変数のプロパティを設定する場合や変数のプロパティにアクセスする場合、または変数のインスタンス・メソッドを呼び出す場合に、その変数が OREF でないと、<INVALID OREF> エラーを受け取ることになります。以下に例を示します。

```
MYNAMESPACE>write p2.PrintPerson()

WRITE p2.PrintPerson()
^
<INVALID OREF>
MYNAMESPACE>set p2.Name="Dixby,Jase"

SET p2.Name="Dixby,Jase"
^
<INVALID OREF>
```

注釈　式に OREF のチェーンが含まれているときは、詳細が異なります。"ドット構文の概要" を参照してください。

8.2.2 OREF のテスト

InterSystems IRIS には、$ISOBJECT という関数があります。この関数を使用すると、特定の変数が OREF を保持しているかどうかをテストできます。この関数は、変数に OREF が格納されている場合に 1 を返します。それ以外の場合は 0 を返します。特定の変数が OREF を格納していない可能性のある場面では、変数のプロパティを設定する前や変数のプロパティにアクセスする前、または変数のインスタンス・メソッドを呼び出す前に、この関数を使用するようお勧めします。

8.2.3 OREF、スコープおよびメモリ

あらゆる OREF はメモリ内のオブジェクトへのポインタであり、そのオブジェクトは別の OREF がポイントしている可能性があります。つまり、OREF (変数) は、メモリ内のオブジェクトとは別のものであるということです (ただし、実際には OREF という用語とオブジェクトという用語は、ほとんどの場合に同じ意味で使用されます)。

InterSystems IRIS では、メモリ内の構造が以下のように自動的に管理されます。メモリ内のオブジェクトごとに、InterSystems IRIS は参照カウント (そのオブジェクトへの参照数) を維持します。変数やオブジェクトのプロパティにオブジェクトへの参照を設定すると、常に、そのオブジェクトの参照カウントが自動的にインクリメントされるようになります。変数がオブジェクトへの参照を停止すると (範囲外になった場合、削除された場合、または新しい値が設定された場合)、そのオブジェクトの参照カウントはデクリメントされます。このカウントが 0 に達すると、オブジェクトは自動的に消滅し (メモリから削除され)、%OnClose() メソッド (存在する場合) が呼び出されます。

例えば、以下のメソッドを考えてみます。

Class Member

```
Method Test()
{
    Set person = ##class(Sample.Person).%OpenId(1)
    Set person = ##class(Sample.Person).%OpenId(2)
}
```

このメソッドは Sample.Person のインスタンスを作成し、そのインスタンスに対する参照を変数 person に配置します。そして、Sample.Person の別のインスタンスを作成し、person の値を新しいインスタンスに対する参照に変更します。ここで、最初のオブジェクトは参照しているものがなくなり、消滅します。メソッドの最後で、person は範囲外になり、2 番目のオブジェクトは消滅します。

8.2.4 OREF の削除

必要に応じて、KILL コマンドを以下のように使用することで、OREF を削除できます。

```
kill OREF
```

この OREF は、OREF を格納している変数です。このコマンドにより、変数が削除されます。オブジェクトへの参照がなくなった場合は、前述したように、このコマンドによりオブジェクトもメモリから削除されます。

8.2.5 OREF、SET コマンド、およびシステム関数

一部のシステム関数 ($Piece、$Extract、$List など) について、InterSystems IRIS では既存の値を変更するために使用できる代替構文をサポートしています。この構文では、以下のように、SET コマンドと関数を組み合わせます。

```
SET function_expression = value
```

function_expression はシステム関数の呼び出し (引数付き) であり、value は値です。例えば、以下の文では、"Magenta" と等しくなるように colorlist 文字列の最初の部分を設定しています。

```
SET $PIECE(colorlist,",",1)="Magenta"
```

この方法で OREF やそのプロパティを変更することはサポートされていません。

$DATA()、$GET()、および $INCREMENT() は、プロパティが多次元である場合にのみ、プロパティを引数として取ることができます。これらの処理は、インスタンス変数構文 i%PropertyName を使用してオブジェクト・メソッド内で実行できます。プロパティを MERGE コマンドで使用することはできません。

8.3 新しいオブジェクトの作成

特定のオブジェクト・クラスの新しいインスタンスを作成するには、そのクラスのクラス・メソッド %New() を使用します。このメソッドにより、オブジェクトが作成され、OREF が返されます。以下に例を示します。

ObjectScript

```
Set person = ##class(MyApp.Person).%New()
```

%New() メソッドは 1 つの引数を受け入れます。この引数は、既定では無視されます。存在する場合、この引数はクラスの %OnNew() コールバック・メソッドに渡されます (定義されている場合)。%OnNew() が定義されている場合は、新しく作成したオブジェクトを初期化するために引数を使用できます。詳細は、"コールバック・メソッドの実装" を参照してください。

特定のクラスの新規オブジェクトを作成する方法に影響を与える複雑な要件がある場合は、そのクラスのインスタンスを作成するために使用する代替のメソッドを用意できます。そのようなメソッドでは、%New() を呼び出してから、必要に応じてオブジェクトのプロパティを初期化することになります。このようなメソッドは、ファクトリ・メソッドと呼ばれることがあります。

8.4 オブジェクトのコンテンツの表示

WRITE コマンドは、OREF について、以下の形式の出力を書き込みます。

```
n@Classname
```

Classname は、クラスの名前です。また、n は、このクラスのメモリ内の特定のインスタンスを示す整数です。以下に例を示します。

```
MYNAMESPACE>write p
8@Sample.Person
```

ZWRITE コマンドを OREF に使用すると、InterSystems IRIS は関連するオブジェクトについての詳細な情報を表示します。

```
MYNAMESPACE>zwrite p
p=<OBJECT REFERENCE>[8@Sample.Person]
+----------------- general information ---------------
|      oref value: 1
|      class name: Sample.Person
|           %%OID: $lb("3","Sample.Person")
| reference count: 2
+----------------- attribute values ------------------
|       %Concurrency = 1  <Set>
|                DOB = 33589
|               Name = "Clay,George O."
|                SSN = "480-57-8360"
+----------------- swizzled references ---------------
|   i%FavoriteColors = ""  <Set>
|   r%FavoriteColors = ""  <Set>
|             i%Home = $lb("5845 Washington Blvd","St Louis","NM",55683)  <Set>
|             r%Home = ""  <Set>
|           i%Office = $lb("3413 Elm Place","Pueblo","WI",98532)  <Set>
|           r%Office = ""  <Set>
|           i%Spouse = ""
|           r%Spouse = ""
+-----------------------------------------------------
```

この情報には、オブジェクトに関して、クラス名、OID、参照カウント、およびプロパティの現在の値（メモリ内）が表示される点に注目してください。`swizzled references` のセクションでは、`i%` で始まる名前の項目がインスタンス変数です。（`r%` で始まる名前の項目は、内部使用専用です。）

8.5 ドット構文の概要

OREF では、ドット構文を使用して、関連するオブジェクトのプロパティとメソッドを参照できます。このセクションでは、ドット構文の概要について説明します。ドット構文については、オブジェクトのメソッドとプロパティを参照する代替手段と共に後続のセクションでも説明します。

一般的なドット構文は、以下のようになります。

```
oref.membername
```

例えば、オブジェクトのプロパティの値を指定する場合は、以下のような文を使用できます。

ObjectScript

```
Set oref.PropertyName = value
```

oref は特定のオブジェクトの OREF で、PropertyName は設定するプロパティの名前です。value は、目的の値に評価される ObjectScript 式です。これは、定数でも複雑な式でもかまいません。

同じ構文をオブジェクトのメソッド (インスタンス・メソッド) の呼び出しに使用できます。インスタンス・メソッドは、クラスの特定のインスタンスから呼び出され、通常はそのインスタンスに関連するアクションを実行します。以下の例では、Name プロパティを設定したばかりのオブジェクトの PrintPerson() メソッドを呼び出しています。

ObjectScript

```
set person=##class(Sample.Person).%New()
set person.Name="Carter,Jacob N."
do person.PrintPerson()
```

メソッドが値を返す場合は、以下のように SET コマンドを使用することで、返り値を変数に代入できます。

```
SET myvar=oref.MethodName()
```

メソッドが値を返さない場合 (または、返り値を無視する場合) は、以下のように DO を使用するか JOB を使用します。

```
Do oref.MethodName()
```

メソッドが引数を受け入れる場合は、括弧で囲んで引数を指定します。

ObjectScript

```
Set value = oref.methodName(arglist)
```

8.5.1 カスケード・ドット構文

クラス定義によっては、プロパティがオブジェクト値になることがあります。これは、そのプロパティのタイプがオブジェクト・クラスであることを意味します。このような場合は、OREF のチェーンを使用することで、複数のプロパティのうちの 1 つのプロパティ (または、プロパティのメソッド) を参照できます。これを、カスケード・ドット構文といいます。例えば、以下の構文では、`Person` オブジェクトの `HomeAddress` プロパティのうちの `Street` プロパティを参照しています。

```
set person.HomeAddress.Street="15 Mulberry Street"
```

この例では、`person` 変数は OREF であり、式 `person.HomeAddress` も OREF です。

注釈　一般に、クラス・メンバを参照するときには、非公式の参照である PackageName.ClassName.Member (例えば、Accounting.Invoice.LineItem プロパティ) を使用することがあります。この形式がコード内に現れることはありません。

8.5.2 NULL OREF が含まれたカスケード・ドット構文

プロパティを参照するために OREF のチェーンを使用しているときに、中間オブジェクトが設定されていない場合、一般に、そのオブジェクトに関する `<INVALID OREF>` エラーを受け取るよりも、式に NULL 文字列を返したほうが好都合になります。そのため、中間オブジェクトが設定されていない (NULL 文字列と等しくなる) 場合、チェーンによるプロパティの参照には NULL 文字列の値が返されます。

例えば、pers が Sample.Person の有効なインスタンスであり、pers.Spouse が `""` と等しい場合、以下の文では name 変数に `""` が設定されます。

```
set name=pers.Spouse.Name
```

この動作が適していないコンテキストでは、中間オブジェクトの参照をコードで明示的にチェックする必要があります。

8.6 オブジェクトの検証

%RegisteredObject クラスには、インスタンスのプロパティを検証する %ValidateObject, メソッドが用意されています。このメソッドは、以下の条件に当てはまる場合、1 を返します。

- すべての必須プロパティに値がある。プロパティを必須にするには、Required キーワードを使用します。プロパティのタイプが %Stream の場合、そのストリームを null ストリームにできません。つまり、そのプロパティに、%IsNull() メソッドが 1 を返す値を指定することはできません。
- 各プロパティの値が (NULL でない場合)、それに関連付けられたプロパティ定義に照らして有効である。

  例えば、プロパティのタイプが %Boolean の場合、値 "abc" は有効ではありませんが、値 0 と値 1 は有効です。
- 各リテラル・プロパティの値が (NULL でない場合)、プロパティ定義で定義されたすべての制約に従っている。制約という用語は、プロパティ値に制限を適用するプロパティ・キーワードを指します。例えば、MAXLEN、MAXVAL、MINVAL、VALUELIST、PATTERN などです。完全なリストは、"リテラル・プロパティの定義と使用" を参照してください。

  例えば、あるプロパティのタイプが %String で MAXLEN の既定値が 50 の場合、そのプロパティは 50 文字以下の長さに制限されます。
- オブジェクト値プロパティの場合、参照されるオブジェクトのすべてのプロパティが前述のルールに従っている。

%ValidateObject() は、各プロパティの検証ロジックを順番に呼び出します。詳細は、"プロパティ・メソッドの使用とオーバーライド" を参照してください。

いずれかのプロパティが検証チェックに失敗した場合、このメソッドはエラー・ステータスを返します。

Sample.Person クラスを見てみましょう。このクラスのオブジェクトには、必須の社会保障番号プロパティ SSN が含まれます。このプロパティは、NNN-NN-NNNN の形式で指定されている必要があります。

Class Member

```
Property SSN As %String(PATTERN = "3N1""-""2N1""-""4N") [ Required ];
```

Sample.Person オブジェクトを新規作成し、このプロパティを含めていない場合、%ValidateObject() は、このプロパティが欠落していてパターンに一致しないことを示すエラー・ステータスを返します。

ObjectScript

```
set person = ##class(Sample.Person).%New()
set status = person.%ValidateObject()
do $system.OBJ.DisplayError(status)
```

```
ERROR #5659: Property 'Sample.Person::SSN(9@Demo.Person,ID=)' required
ERROR #7209: Datatype value '' does not match PATTERN '3N1"-"2N1"-"4N'
  > ERROR #5802: Datatype validation failed on property 'Sample.Person:SSN', with value equal to ""
```

このプロパティを設定しても指定パターンに一致しない場合、%ValidateObject() は、パターン・マッチングのエラー・ステータスのみを返します。

ObjectScript

```
    set person.SSN = "0000000000"
    set status = person.%ValidateObject()
    do $system.OBJ.DisplayError(status)

ERROR #7209: Datatype value '' does not match PATTERN '3N1"-"2N1"-"4N'
  > ERROR #5802: Datatype validation failed on property 'Sample.Person:SSN', with value equal to ""
```

このプロパティを適切に設定すると、このオブジェクトは %ValidateObject() チェックに合格し、ステータス 1 を返します。

永続オブジェクトが有効と見なされるためには、そのプロパティが、定義済みのどの一意制約にも違反していない必要もあります。%ValidateObject() は、一意性をチェックしません。このチェックの対象はオブジェクト内のプロパティに限られ、データベースに保存される他のオブジェクトのプロパティはチェックの対象とならないためです。一意性のチェックは、%Save() を呼び出してオブジェクトを保存するときに行われます。

前述の例を基に、Sample.Person クラスで SSN プロパティを一意とするよう制約するとします。

```
    Index SSNIndex On SSN [ Unique ]
```

新しい person を作成して、前の person と同じ社会保障番号を設定すると、%ValidateObject() はステータス 1 を返し、$system.OBJ.DisplayError はこのステータスに対してエラーを表示しません。

ObjectScript

```
    do person.%Save() // Save first object to database

    set person2 = ##class(Sample.Person).%New()
    set person2.SSN = "000-00-0000"
    set status = person2.%ValidateObject()
    do $system.OBJ.DisplayError(status)
```

ただし、このオブジェクトをデータベースに保存しようとすると失敗します。これは、SSN プロパティに一意制約が適用されているためです。

ObjectScript

```
    set status = person2.%Save()
    do $system.OBJ.DisplayError(status)

ERROR #5808: Key not unique: Sample.Person:SSNIndex:^Sample.PersonI("SSNIndex"," 000-00-0000")
```

永続オブジェクトを保存する場合は、自動的に %ValidateObject() メソッドが最初に呼び出されます。オブジェクトがこのチェックに失敗し、さらに一意性チェックにも失敗した場合、このオブジェクトは保存されません。詳細は、"永続オブジェクトの概要" を参照してください。

8.7 オブジェクト・タイプの判別

オブジェクトが存在すれば、%RegisteredObject クラスに、そのオブジェクトの継承を判断するためのメソッドがあります。このセクションでは、これについて説明します。

8.7.1 %Extends()

オブジェクトが特定のスーパークラスを継承しているかどうかを確認するには、そのオブジェクトの %Extends() メソッドを呼び出して、スーパークラスの名前を引数として渡します。このメソッドから 1 が返される場合、インスタンスは、そのクラスを継承しています。0 が返される場合、インスタンスは、そのクラスを継承していません。以下に例を示します。

```
MYNAMESPACE>set person=##class(Sample.Person).%New()

MYNAMESPACE>w person.%Extends("%RegisteredObject")
1
MYNAMESPACE>w person.%Extends("Sample.Person")
1
MYNAMESPACE>w person.%Extends("Sample.Employee")
0
```

8.7.2 %IsA()

オブジェクトにプライマリ・スーパークラスとして特定のクラスがあるかどうかを確認するには、そのオブジェクトの %IsA() メソッドを呼び出して、スーパークラスの名前を引数として渡します。このメソッドから 1 が返される場合、オブジェクトには特定クラスがそのプライマリ・スーパークラスとして存在します。

8.7.3 %ClassName() および最も適切なタイプのクラス (MSTC)

オブジェクトは複数のクラスのうちの 1 つのインスタンスである可能性がありますが、オブジェクトには必ず 1 つの 最も適切なタイプのクラス (MSTC) があります。オブジェクトがクラスのインスタンスであり、さらにそのクラスのどのサブクラスのインスタンスにもなっていない場合、そのクラスはそのオブジェクトの最も適切なタイプのクラスとなります。

例えば、GradStudent クラスが Student クラスを継承し、Student クラスが Person クラスを継承する場合に、以下のコマンドで作成するインスタンスについて考えてみます。

ObjectScript

```
 set MyInstance1 = ##class(MyPackage.Student).%New()
 set MyInstance2 = ##class(MyPackage.GradStudent).%New()
```

MyInstance1 に対しては Student が MSTC になります。それは、MyInstance1 は Person と Student のインスタンスですが、GradStudent のインスタンスではないからです。MyInstance2 に対しては GradStudent が MSTC になります。それは、MyInstance2 が GradStudent、Student、および Person のインスタンスであるからです。

オブジェクトの MSTC に関しては、以下のルールも適用されます。

- オブジェクトの MSTC はプライマリ継承にのみ基づきます。
- インスタンス化できないクラスは、オブジェクトの MSTC になることはできません。オブジェクト・クラスは、インスタンス化できません (そのクラスが抽象の場合)。

オブジェクトの MSTC を確認するには、%RegisteredObject から継承された %ClassName() メソッドを以下のように使用します。

```
classmethod %ClassName(fullname As %Boolean) as %String
```

fullname はブーリアン引数です。1 はこのメソッドでパッケージ名とクラス名を返すことを指定し、0 (既定) はこのメソッドでクラス名のみを返すことを指定します。

例 :

```
 write myinstance.%ClassName(1)
```

(同様に、%PackageName() を使用すると、パッケージの名前だけを取得できます。)

8.8 オブジェクトのクローン化

オブジェクトをクローン化するには、そのオブジェクトの %ConstructClone() メソッドを呼び出します。このメソッドが新規の OREF を作成します。

下記のターミナル・セッションでこれを示します。

```
MYNAMESPACE>set person=##class(Sample.Person).%OpenId(1)

MYNAMESPACE>set NewPerson=person.%ConstructClone()

MYNAMESPACE>w

NewPerson=<OBJECT REFERENCE>[2@Sample.Person]
person=<OBJECT REFERENCE>[1@Sample.Person]
MYNAMESPACE>
```

ここで、NewPerson 変数が元の person オブジェクトとは異なる OREF を使用していることがわかります。NewPerson は person のクローンです (より正確には、それらの変数は別の同一オブジェクトのポインタです)。

一方、以下のターミナル・セッションを考えてみます。

```
MYNAMESPACE>set person=##class(Sample.Person).%OpenId(1)

MYNAMESPACE>set NotNew=person

MYNAMESPACE>w

NotNew=<OBJECT REFERENCE>[1@Sample.Person]
person=<OBJECT REFERENCE>[1@Sample.Person]
```

ここで、両変数が同じ OREF を参照していることに注意してください。つまり、NotNew は person のクローンではありません。

このメソッドの引数の詳細は、"インターシステムズ・クラス・リファレンス" で %Library.RegisteredObject を参照してください。

8.9 インスタンスのプロパティの参照

インスタンスのプロパティを参照するには、以下のいずれかを実行します。

- 関連するクラスのインスタンスを作成し、前述したようにドット構文を使用して、そのインスタンスのプロパティを参照します。
- 関連するクラスのインスタンス・メソッド内で、以下に示すように、相対ドット構文を使用します。

  ```
  ..PropName
  ```

  この式は、SET コマンドと共に使用することも、別の式の一部として使用することもできます。以下に、いくつかのバリエーションを示します。

  ```
  set value=..PropName
  write ..PropName
  ```

- 実行時までプロパティ名が決定されないプロパティにアクセスするには、$PROPERTY 関数を使用します。プロパティが多次元の場合、プロパティの値にアクセスするときのインデックスとして、プロパティ名に続く引数を使用します。このシグニチャは以下のようになります。

  ```
  $PROPERTY (oref, propertyName, subscript1, subscript2, subscript3... )
  ```

oref は OREF です。propertyName は関連するクラスのプロパティ・メソッドの名前に評価されます。また、subscript1、subscript2、subscript3 は、プロパティの添え字に対応する値です (多次元プロパティの場合にのみ指定します)。

詳細は、"$PROPERTY" を参照してください。

- (インスタンス・メソッド内で) 変数 $this を使用します。
- (インスタンス・メソッド内で) インスタンス変数を使用します。
- 適切なプロパティ・アクセサ (ゲッターおよびセッター) メソッドを使用します。"プロパティ・メソッドの使用とオーバーライド" を参照してください。

8.10 インスタンスのメソッドの呼び出し

インスタンスのメソッドを呼び出すには、以下のいずれかを実行できます。

- 関連するクラスのインスタンスを作成し、前述したようにドット構文を使用して、そのインスタンスのメソッドを呼び出します。
- (インスタンス・メソッド内で) 以下の式を使用して、そのクラスの別のインスタンス・メソッド (継承されたメソッドでもかまいません) を呼び出します。

```
..MethodName(args)
```

この式は DO コマンドと共に使用できます。メソッドが値を返す場合は、SET を使用することも、別の式の一部として使用することもできます。以下に、いくつかのバリエーションを示します。

```
do ..MethodName()
set value=..MethodName(args)
```

- 実行時までメソッド名が決定されないインスタンス・メソッドを実行するには、以下のように $METHOD 関数を使用します。

```
$METHOD(oref, methodname, Arg1, Arg2, Arg3, ... )
```

oref は OREF です。methodname は関連するクラスのインスタンス・メソッドの名前に評価されます。また、Arg1、Arg2、Arg3 などは、そのメソッドの引数です。

詳細は、"$METHOD" を参照してください。

- (インスタンス・メソッド内で) 変数 $this を使用します。

8.11 インスタンスからのクラス名の取得

クラスの名前を取得するには、以下のように $CLASSNAME 関数を使用します。

```
$CLASSNAME(oref)
```

oref は OREF です。

詳細は、"$CLASSNAME" を参照してください。

8.12 $this 変数 (現在のインスタンス)

$this 構文は、現在のインスタンスの OREF へのハンドルを提供します。例えば、現在のインスタンスを別のクラスに渡したり、別のクラスが現在のインスタンスのメソッドのプロパティを参照するために使用します。インスタンスが、そのインスタンスのプロパティやメソッドを参照する場合は、相対ドット構文のほうが高速であるために推奨されています。

注釈　$this は大文字と小文字を区別しません。このため、$this、$This、$THIS、およびその他の変異形は同じ値を持ちます。

例えば、Accounting.Order クラスと Accounting.Utils クラスを持つアプリケーションがあるとします。Accounting.Order.CalcTax() メソッドは、Accounting.Utils.GetTaxRate() メソッドおよび Accounting.Utils.GetTaxableSubtotal() メソッドを呼び出し、現在のインスタンスの "city" と "state" の値を GetTaxRate() メソッドに渡し、注文された品目と関連する税金関係の情報のリストを GetTaxableSubtotal() に渡します。CalcTax() は、返された値を使用して、注文に対する消費税を算出します。従って、コードは以下のようになります。

Class Member

```
Method CalcTax() As %Numeric
{
    Set TaxRate = ##Class(Accounting.Utils).GetTaxRate($this)
    Write "The tax rate for ",..City,", ",..State," is ",TaxRate*100,"%",!
    Set TaxableSubtotal = ##class(Accounting.Utils).GetTaxableSubTotal($this)
    Write "The taxable subtotal for this order is $",TaxableSubtotal,!
    Set Tax = TaxableSubtotal * TaxRate
    Write "The tax for this order is $",Tax,!
}
```

メソッドの最初の行では、##Class 構文 (前述) を使用して別のメソッドを呼び出しています。ここでは $this 構文を使用して、そのメソッドに現在のオブジェクトを渡します。メソッドの 2 行目では、.. 構文 (前述) を使用して、City プロパティおよび State プロパティの値を取得しています。3 行目の動作は、1 行目と同様です。

Accounting.Utils の GetTaxRate() メソッドは、渡されたインスタンスのハンドルを使用して、さまざまなプロパティへのハンドルを (値を取得したり設定したりするために) 取得します。

Class Member

```
ClassMethod GetTaxRate(OrderBeingProcessed As Accounting.Order) As %Numeric
{
    Set LocalCity = OrderBeingProcessed.City
    Set LocalState = OrderBeingProcessed.State
    // code to determine tax rate based on location and set
    // the value of OrderBeingProcessed.TaxRate accordingly
    Quit OrderBeingProcessed.TaxRate
}
```

GetTaxableSubtotal() もインスタンスのハンドルを使用し、そのプロパティを確認して TaxableSubtotal プロパティの値を設定します。

そのため、Accounting.Order クラスの MyOrder インスタンスにある CalcTax() メソッドを呼び出すと、以下のような内容が表示されます。

```
>Do MyOrder.CalcTax()
The tax rate for Cambridge, MA is 5%
The taxable subtotal for this order is $79.82
The tax for this order is $3.99
```

8.13 i%PropertyName (インスタンス変数)

このセクションでは、インスタンス変数の概要について説明します。このような変数は、プロパティのアクセサ・メソッドをオーバーライドしていない場合は参照する必要がありません。これについては、"プロパティ・メソッドの使用とオーバーライド" を参照してください。

どのようなクラスのインスタンスを作成するときでも、システムは、そのクラスの非計算プロパティごとに 1 つのインスタンス変数を作成します。インスタンス変数は、プロパティの値を保持します。プロパティ PropName の場合、インスタンス変数には i%PropName という名前が付けられます。この変数名は、大文字と小文字が区別されます。このような変数は、クラスのインスタンス・メソッド内で使用できます。

例えば、あるクラスにプロパティ Name と DOB がある場合は、そのクラスのあらゆるインスタンス・メソッド内で、インスタンス変数 i%Name と i%DOB を使用できます。

InterSystems IRIS では、r%PropName や m%PropName という名前の付いた追加のインスタンス変数も内部的に使用しますが、このような変数の直接使用はサポートされていません。

インスタンス変数は、その変数に割り当てられたプロセス・プライベートのメモリ内領域を保持します。このような変数は、ローカル変数シンボル・テーブルに保持されないため、Kill コマンドの影響は受けません。

8.14 関連項目

- クラスの定義
- 永続オブジェクトの概要
- リテラル・プロパティの定義と使用

9

永続オブジェクトの概要

ここでは、永続クラスを使用した作業時の理解に役立つ概念について説明します。

ここで示されているサンプルの一部は、Samples-Data サンプル (https://github.com/intersystems/Samples-Data) からのものです。(例えば) SAMPLES という名前の専用ネームスペースを作成し、そのネームスペースにサンプルをロードすることをお勧めします。 一般的な手順は、"サンプルのダウンロード" を参照してください。

9.1 永続クラス

永続クラスとは、%Persistent を継承するあらゆるクラスのことです。永続オブジェクトとは、そのようなクラスのインスタンスのことです。

%Persistent クラスは、%RegisteredObject のサブクラスであるため、オブジェクト・クラスになります。オブジェクト・インスタンス・メソッドを提供することに加え、%Persistent クラスでは、永続インタフェースも定義します。これは、メソッドのセットです。これらのメソッドにより、データベースにオブジェクトを保存すること、データベースからオブジェクトをロードすること、オブジェクトを削除すること、および存在をテストすることができるようになります。

9.2 既定の SQL プロジェクションの概要

どのような永続クラスについても、コンパイラが SQL テーブル定義を生成します。この定義により、オブジェクト・インタフェースに加えて、保存されたデータに SQL でアクセスできます。

このテーブルには、保存されたオブジェクトごとに 1 つのレコードが格納されています。また、このテーブルは InterSystems SQL でクエリできます。以下に、Sample.Person テーブルのクエリ結果を示します。

ID	Age	DOB	FavoriteColors	Name	SSN	Spouse	Home_City	Home_State	Home_Street	Home_Zip	Office_City
1	1	09/19/2013	Green Green	Humby,Michael G.	732-63-4007		Boston	MD	4033 Second Avenue	87184	Vail
2	17	08/26/1997		Jenkins,Usha V.	921-79-1526		Jackson	KS	1156 Washington Place	73402	Tampa
3	16	06/02/1998	White Black	Diavolo,Emma T.	842-30-8977		Larchmont	AL	5794 Elm Street	13434	Miami
4	26	07/09/1988	Yellow Purple	Anderson,Alvin T.	235-56-7318		Albany	ND	4145 Ash Place	57771	Vail
5	27	07/23/1987	Blue	Zemaitis,Christen D.	958-46-8312		Oak Creek	VT	6581 Ash Drive	89034	Gansevoort
6	89	08/09/1925	Red	Adams,Frances Q.	297-20-9962		Chicago	MT	6141 Washington Blvd	42799	Vail
7	78	04/18/1936		Press,Laura F.	586-94-4188		Larchmont	OH	5772 Washington Drive	38850	Islip

以下のテーブルに、既定のプロジェクションをまとめます。

テーブル 9-1: オブジェクト SQL プロジェクション

投影元 (オブジェクト・コンセプト)	投影先 (リレーショナル・コンセプト)
パッケージ	スキーマ
クラス	テーブル
OID	ID フィールド
データ型プロパティ	フィールド
参照プロパティ	参照フィールド
埋め込みオブジェクト	一連のフィールド
リスト・プロパティ	リスト・フィールド
配列プロパティ	子テーブル
ストリーム・プロパティ	BLOB
インデックス	インデックス
クラス・メソッド	ストアド・プロシージャ

他のトピックでは、詳細情報を示し、変更可能な内容について説明します。

- テーブル名と、そのテーブルが属するスキーマの名前の詳細は、"永続クラスの定義" を参照してください。

  同じトピックでは、サブクラスのプロジェクションを制御する方法についても説明しています。
- リテラル・プロパティのプロジェクションについての詳細は、"リテラル・プロパティの定義と使用" を参照してください。
- コレクション・プロパティのプロジェクションについての詳細は、"コレクション・プロパティのストレージと SQL プロジェクション" を参照してください。
- ストリーム・プロパティのプロジェクションについての詳細は、"ストリームを使用した作業" を参照してください。
- オブジェクト値プロパティのプロジェクションについての詳細は、"オブジェクト値プロパティの定義と使用" を参照してください。
- リレーションシップのプロジェクションについての詳細は、"リレーションシップの定義と使用" を参照してください。

9.3 保存したオブジェクトの識別子 : ID および OID

初めてオブジェクトを保存するとき、システムは、2 つの永続識別子を作成します。どちらの識別子も保存したオブジェクトに後でアクセスする場合や、保存したオブジェクトを削除する場合に使用できます。最も広範に使用される識別子は、オブジェクト ID です。ID は、テーブル内で一意である単純なリテラル値です。既定では、システムは ID として使用する整数を生成します。

OID は、より一般的です。OID にはクラス名も含まれていて、データベース内で一意です。実際には、アプリケーションは OID 値を使用する必要はありません。ID 値で十分に機能します。

%Persistent クラスには、ID または OID を使用するメソッドがあります。ID は、%OpenId()、%ExistsId()、%DeleteId() などのメソッドを使用するときに指定します。OID は、%Open()、%Exists()、%Delete() などのメソッドに引数として指定します。つまり、引数として ID を使用するメソッドは、そのメソッドの名前に `Id` が含まれているということです。引数として OID を使用するメソッドは、そのメソッドの名前に `Id` が含まれていません。このようなメソッドは、ほとんど使用されません。

永続オブジェクトがそのデータベースに保存されると、その参照属性 (つまり、他の永続オブジェクトへの参照) の値はすべて OID 値として保存されます。OID を持たないオブジェクト属性については、オブジェクトのリテラル値が、その他のオブジェクト状態と共に保存されます。

9.3.1 オブジェクト ID から SQL へのプロジェクション

オブジェクトの ID は、対応する SQL テーブルで使用できます。可能な場合、InterSystems IRIS は `ID` というフィールド名を使用します。また、InterSystems IRIS には、どのフィールド名を使用しているかわからない場合でも、ID にアクセスする方法があります。このシステムは、以下のとおりです。

- オブジェクト ID は、オブジェクトのプロパティではないため、プロパティとは異なる扱いを受けます。
- `ID` という名前のプロパティ (どのような大文字小文字の組合せであっても) がクラスに含まれていない場合、テーブルにはフィールド `ID` も含まれるようになり、そのフィールドにオブジェクト ID が格納されます。この例については、前のセクションを参照してください。
- `ID` という名前 (どのような大文字小文字の組合せであっても) を SQL に投影したプロパティがクラスに含まれている場合、テーブルにはフィールド `ID1` も含まれるようになり、このフィールドにオブジェクト ID の値が保持されます。

  同様に、`ID` および `ID1` として投影されたプロパティがクラスに含まれている場合、テーブルにはフィールド `ID2` も含まれるようになり、このフィールドにオブジェクト ID の値が保持されます。
- すべての場合で、テーブルに疑似フィールド `%ID` も用意され、オブジェクト ID の値が保持されます。

OID は SQL テーブルでは使用できません。

9.3.2 SQL でのオブジェクト ID

InterSystems IRIS は、ID フィールドに一意性を強制します (実際の名前は、どのようなものであってもかまいません)。また、InterSystems IRIS は、このフィールドの変更を禁止します。つまり、このフィールドに対する SQL UPDATE または INSERT 操作は実行できないということです。例として、新しいレコードをテーブルに追加するために必要な SQL を以下に示します。

SQL

```
INSERT INTO PERSON (FNAME, LNAME)VALUES (:fname, :lname)
```

この SQL は ID フィールドを参照しません。InterSystems IRIS は、ID フィールドの値を生成し、要求されたレコードの作成時にその値を挿入します。

9.4 永続クラスに固有のクラス・メンバ

InterSystems IRIS クラスには、永続クラスでのみ意味のある数種のクラス・メンバを含めることができます。これに該当するものは、ストレージ定義、インデックス、外部キー、およびトリガです。

9.4.1 ストレージ定義

ほとんどの場合 (後述するように)、永続クラスごとにストレージ定義が存在します。ストレージ定義の用途は、グローバル構造を記述することです。InterSystems IRIS では、クラスのデータを保存するときや、クラスの保存データを読み込むときに、この構造を使用します。統合開発環境 (IDE) では、クラス定義の最後の部分にストレージ定義が表示されます。以下は、部分的な例です。

```
<Storage name="Default">
<Data name="PersonDefaultData">
<Value name="1">
<Value>%%CLASSNAME</Value>
</Value>
<Value name="2">
<Value>Name</Value>
</Value>
<Value name="3">
<Value>SSN</Value>
</Value>
<Value name="4">
<Value>DOB</Value>
</Value>
<Value name="5">
<Value>Home</Value>
</Value>
<Value name="6">
<Value>Office</Value>
</Value>
<Value name="7">
<Value>Spouse</Value>
</Value>
<Value name="8">
<Value>FavoriteColors</Value>
</Value>
</Data>
<DataLocation>^Sample.PersonD</DataLocation>
<DefaultData>PersonDefaultData</DefaultData>
<ExtentSize>200</ExtentSize>
<IdLocation>^Sample.PersonD</IdLocation>
<IndexLocation>^Sample.PersonI</IndexLocation>
<Property name="%%CLASSNAME">
<Selectivity>50.0000%</Selectivity>
</Property>
...
```

また、ほとんどの場合、コンパイラでもストレージ定義が生成および更新されます。永続クラスに使用するグローバルの詳細は、"永続オブジェクトとグローバル" を参照してください。

9.4.2 インデックス

その他の SQL テーブルのように、InterSystems SQL テーブルはインデックスを保持できます。インデックスを定義するには、対応するクラス定義にインデックス定義を追加します。

インデックスにより、特定のフィールドまたはフィールドの組合せの一意性を確保する制約を追加できます。このようなインデックスの詳細は、"永続クラスの定義" を参照してください。

インデックスのもう 1 つの用途は、クエリの実行速度を速くするために、クラスに関連付けられ頻繁に要求されるデータのソート済み特定サブセットを定義することです。例えば、一般に、特定のフィールドを使用する WHERE 節がクエリに含まれている場合は、そのフィールドにインデックスが作成されていると、クエリの実行速度が速くなります。一方、そのフィールドにインデックスがない場合、エンジンは、すべての行に対して指定の条件と一致しているかどうかを確認する

フル・テーブル・スキャンを実行する必要があります。これは、テーブルが巨大になると、負荷の高い操作になります。"永続クラスのその他のオプション" を参照してください。

9.4.3 外部キー

InterSystems SQL テーブルは、外部キーも保持できます。これを定義するには、対応するクラス定義に外部キー定義を追加します。

外部キーは、新しいデータが追加された場合や、データが変更された場合に InterSystems IRIS が使用するテーブル間に、参照整合性制約を確立します。リレーションシップを使用すると、システムはリレーションシップを自動的に外部キーとして扱います。リレーションシップを使用しない場合や、追加する理由が別にある場合は、外部キーを追加することもできます。

外部キーの詳細は、"永続クラスのその他のオプション" を参照してください。

9.4.4 トリガ

InterSystems SQL テーブルは、トリガも保持できます。これを定義するには、対応するクラス定義にトリガ定義を追加します。

トリガでは、特定のイベントの発生時 (具体的には、レコードの挿入時、変更時または削除時) に自動的に実行されるコードを定義します。

トリガの詳細は、"永続クラスのその他のオプション" を参照してください。

9.5 その他のクラス・メンバ

クラス・メソッドやクラス・クエリは、ストアド・プロシージャとして呼び出せるように定義できます。このストアド・プロシージャは SQL から呼び出せます。

ここで説明していないクラス・メンバの場合は、SQL への投影がありません。つまり、InterSystems IRIS には、それらのクラス・メンバを SQL から直接使用する方法や、SQL から簡単に使用できるようにする方法がないということです。

9.6 関連項目

- 永続オブジェクトとエクステント
- 永続オブジェクトとグローバル
- 永続オブジェクトを使用した作業
- 永続クラスの定義
- 永続クラスのその他のオプション

10

永続オブジェクトとエクステント

エクステントという用語は、特定の永続クラスのすべてのレコード (ディスク上のレコード) を表します。%Persistent クラスには、クラスのエクステントを操作する複数のメソッドが用意されています。

10.1 エクステントの概要

InterSystems IRIS® データ・プラットフォームは、従来にない強力な解釈によるオブジェクトテーブル・マッピングを使用します。既定では、特定の永続クラスのエクステントには、すべてのサブクラスのエクステントが含まれます。したがって、以下のようになります。

- 永続クラス Person にサブクラス Employee がある場合、Person エクステントには、Person のすべてのインスタンスと Employee のすべてのインスタンスが含まれます。
- クラス Employee のどのインスタンスの場合も、そのインスタンスは Person エクステントおよび Employee エクステントに含まれます。

インデックスは、インデックスが定義されるクラスの全範囲を自動的にカバーします。Person に定義されているインデックスには、Person インスタンスと Employee インスタンスの両方が含まれます。Employee エクステントに定義されているインデックスには、Employee インスタンスだけが含まれます。

サブクラスは、そのスーパークラスで定義されていない追加のプロパティも定義できます。これらは、サブクラスのエクステントでは使用できますが、スーパークラスのエクステントでは使用できません。例えば、Employee エクステントには Department フィールドが含まれますが、これは Person エクステントには含まれません。

前述の点は、InterSystems IRIS では、同じタイプのレコードをすべて取得するクエリを比較的簡単に作成できることを意味します。例えば、すべてのタイプの人々をカウントする場合、Person テーブルに対してクエリを実行できます。従業員のみをカウントする場合、同じクエリを Employee テーブルに対して実行します。

同様に、ID を使用するメソッドは、すべて多様な形態で動作します。つまり、渡された ID 値に応じて、さまざまなタイプのオブジェクトを操作できるということです。

例えば、Sample.Person オブジェクトのエクステントには、Sample.Person のインスタンスと、Sample.Employee のインスタンスが含まれます。Sample.Person クラスの %OpenId() を呼び出すと、結果の OREF は、データベースに保存されている内容に応じて Sample.Person か Sample.Employee のどちらかのインスタンスになります。

ObjectScript

```
// Open person "10"
Set obj = ##class(Sample.Person).%OpenId(10)

Write $ClassName(obj),!    // Sample.Person


// Open person "110"
Set obj = ##class(Sample.Person).%OpenId(110)

Write $ClassName(obj),!    // Sample.Employee
```

Sample.Employee クラスの %OpenId() メソッドでは、ID 10 を開こうとしても、オブジェクトが返されません。これは、ID 10 は Sample.Employee のエクステントではないためです。

ObjectScript

```
// Open employee "10"
Set obj = ##class(Sample.Employee).%OpenId(10)

Write $IsObject(obj),!  // 0

// Open employee "110"
Set obj = ##class(Sample.Employee).%OpenId(110)

Write $IsObject(obj),!  // 1
```

10.2 エクステント定義

既定のストレージ・クラス (%Storage.Persistent) を使用するクラスの場合、InterSystems IRIS は、エクステント・マネージャで使用するために登録されているエクステントのエクステント定義とグローバルを管理します。エクステント・マネージャに対するインタフェースには、%ExtentMgr.Util クラスを使用します。この登録プロセスはクラスのコンパイル時に実行されます。

何らかのエラーまたは名前の競合が発生すると、コンパイルに失敗します。以下に例を示します。

```
ERROR #5564: Storage reference: '^This.App.Global used in 'User.ClassA.cls'
is already registered for use by 'User.ClassB.cls'
```

コンパイルを成功させるには競合を解決します。このユーティリティでは、インデックスの名前を変更するか、データのストレージの場所を明確に追加する必要があります。

注釈　グローバル参照を意図的に共有しているクラスがアプリケーションに複数存在する場合は、関連するすべてのクラスの MANAGEDEXTENT に 0 を指定します (関連するクラスが既定のストレージを使用する場合)。

MANAGEDEXTENT クラス・パラメータの既定値は 1 です。この値では、グローバル名の登録および競合使用のチェックが実行されます。値が 0 の場合には、登録も競合のチェックも実行されません。

別の可能性として、InterSystems IRIS に古いエクステント定義が含まれていることがあります。例えば、クラスを削除しても、既定ではエクステント定義は削除されません。

前述の例を引き続き使用すると、User.ClassA.cls は削除されましたが、そのエクステント定義が残っている可能性があります。この場合の解決策は、以下のコマンドを使用して、古いエクステント定義を削除することです。

```
set st = ##class(%ExtentMgr.Util).DeleteExtentDefinition("User.ClassA", "cls")
```

これで、競合することなく、User.ClassB.cls をコンパイルできます。

クラスとそのエクステント定義を削除するには、以下のいずれかの呼び出しを使用します。

- `$SYSTEM.OBJ.Delete(classname,qspec)`。classname は、削除するクラスです。qspec では、フラグ `e` または修飾子 `/deleteextent` を指定します。
- `$SYSTEM.OBJ.DeletePackage(packagename,qspec)`。packagename は、削除するパッケージです。qspec では、フラグ `e` または修飾子 `/deleteextent` を指定します。
- `$SYSTEM.OBJ.DeleteAll(qspec)`。qspec では、フラグ `e` または修飾子 `/deleteextent` を指定します。これにより、ネームスペース内のすべてのクラスが削除されます。

これらの呼び出しは、%SYSTEM.OBJ クラスのメソッドです。

10.3 エクステント・インデックス

エクステント・インデックスは、現在のネームスペースで定義されているすべてのエクステントおよびサブエクステントのインデックスです。クラス・コンパイラは、既定のストレージ・クラス (%Storage.Persistent) を使用する、ローカルでコンパイルされたクラスについてはこのインデックスを保持します。これらのクラスでは、そのネームスペースについて、パッケージ・マッピング、ルーチン・マッピング、またはグローバル・マッピングが変更された場合、InterSystems IRIS は自動的にこのインデックスを再構築します。

ただし、他のネームスペースからマップされたクラスは、クラスのランタイムが元のネームスペースで変更された場合に自動的にインデックスに追加されたり、インデックスから削除されたりすることはありません。代わりに、このような変更は、以下のいずれかの呼び出しを使用して、ローカル・ネームスペースのエクステント・インデックスで更新する必要があります。

- `$SYSTEM.OBJ.RebuildExtentIndex(updatemode,lockmode)` は、指定されたネームスペースのエクステント・インデックス全体を再構築し、成功または失敗を示すステータス値を返します。updatemode が true の場合、インデックスは削除されず、検出された差異のみがインデックスで更新されます。updatemode が false の場合は、インデックスが削除され、全体が再構築されます。lockmode が `1` の場合、インデックス構造全体に対して排他ロックが取得されます。値 `2` の場合は、そのクラスのインデックス作成中に限り、クラス・ノードに対して個々のロックが取得されます。値 `3` の場合は、インデックス構造全体に対して共有ロックが取得されます。要求されたロックを取得できない場合は、呼び出し元にエラーが報告されます。lockmode が `0` の場合、ロックなしを示します。
- `$SYSTEM.OBJ.RebuildExtentIndexOne(classname,lockmode)` は、1 つのクラスのエクステント・インデックスを再構築し、成功または失敗を示すステータス値を返します。このメソッドは、classname によって指定されたクラスのエクステント・インデックスでインデックス・ノードを更新します。lockmode の値は RebuildExtentIndex() の場合と同じですが、値 `1`、`2`、および `3` は、個々のクラス・ノードに対するロックとして解釈されます。

これらの呼び出しは、%SYSTEM.OBJ クラスのメソッドです。

10.4 Extent クエリ

すべての永続クラスには、`"Extent"` というクラス・クエリが自動的に含まれます。このクラス・クエリは、エクステントに含まれるすべての ID を提示します。

クラス・クエリの使用に関する一般的な情報は、"クラス・クエリの定義と使用" を参照してください。以下の例では、クラス・クエリを使用して、Sample.Person クラスのすべての ID を表示します。

ObjectScript

```
  set query = ##class(%SQL.Statement).%New()
  set qStatus = query.%PrepareClassQuery("Sample.Person","Extent")
  if $$$ISERR(qStatus) {write "%Prepare failed:" do $SYSTEM.Status.DisplayError(qStatus) quit}

  set rset=query.%Execute()
  if (rset.%SQLCODE '= 0) {write "%Execute failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message
 quit}

  while rset.%Next()
  {
    write rset.%Get("ID"),!
  }
  if (rset.%SQLCODE < 0) {write "%Next failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message
quit}
```

Sample.Person エクステントは、Sample.Person のすべてのインスタンスとそのサブクラスを含みます。これについての説明は、"永続クラスの定義" を参照してください。

"Extent" クエリは、以下の SQL クエリと同等です。

SQL

```
SELECT %ID FROM Sample.Person
```

これらのいずれの方法でも、返される ID 値の順序には依存できないことに注意してください。この要求を満たすために、他のプロパティ値を使用して順序付けられたインデックスを使用する方が効率的であると InterSystems IRIS が判断する場合があるからです。必要に応じて、ORDER BY %ID 節を SQL クエリに追加できます。

10.5 関連項目

- クラスの定義
- 永続オブジェクトの概要
- 永続オブジェクトを使用した作業

11

永続オブジェクトとグローバル

永続クラスを使用することで、オブジェクトをデータベースに保存した後、オブジェクトとして取得することも、SQL を使用して取得することも可能になります。どのようにアクセスするかにかかわらず、これらのオブジェクトはグローバルに格納されます。グローバルは永続多次元配列と考えることができます。

既定のストレージ・クラス (%Storage.Persistent) を使用するクラスを定義した場合、そのクラスをコンパイルすると、クラスのグローバル名が生成されます。これらの名前は、IDE にあるストレージ定義で確認できます。

以降の各セクションでは、グローバルの既定の名前付け方式、パフォーマンスの向上を目的として短いグローバル名を生成する方法、グローバル名を直接制御する方法、および列指向のストレージを使用した場合にグローバルに発生する相違点について説明します。

11.1 標準のグローバル名

IDE でクラスを定義すると、そのクラスの名前に基づいてクラスのグローバル名が生成されます。

例えば、以下のように GlobalsTest.President クラスを定義するとします。

```
Class GlobalsTest.President Extends %Persistent
{

/// President's name (last,first)
Property Name As %String(PATTERN="1U.L1"","1U.L");

/// Year of birth
Property BirthYear As %Integer;

/// Short biography
Property Bio As %Stream.GlobalCharacter;

/// Index for Name
Index NameIndex On Name;

/// Index for BirthYear
Index DOBIndex On BirthYear;

}
```

クラスのコンパイル後に、クラスの下部に以下のようなストレージ定義が生成されることを確認できます。

```
Storage Default
{
<Data name="PresidentDefaultData">
<Value name="1">
<Value>%%CLASSNAME</Value>
</Value>
<Value name="2">
<Value>Name</Value>
</Value>
```

```
<Value name="3">
<Value>BirthYear</Value>
</Value>
<Value name="4">
<Value>Bio</Value>
</Value>
</Data>
<DataLocation>^GlobalsTest.PresidentD</DataLocation>
<DefaultData>PresidentDefaultData</DefaultData>
<IdLocation>^GlobalsTest.PresidentD</IdLocation>
<IndexLocation>^GlobalsTest.PresidentI</IndexLocation>
<StreamLocation>^GlobalsTest.PresidentS</StreamLocation>
<Type>%Storage.Persistent</Type>
}
```

特に、次のストレージ・キーワードに着目してください。

- `DataLocation` は、クラス・データが格納されるグローバルです。このグローバルの名前は、"D" が追加された完全なクラス名 (パッケージ名を含む) で、ここでは、`^GlobalsTest.PresidentD` となっています。
- `IdLocation` (`DataLocation` と同じである場合が多い) は、ID カウンタが格納されるグローバルで、ルートにあります。
- `IndexLocation` は、クラスのインデックスが格納されるグローバルです。このグローバルの名前は、"I" が追加された完全なクラス名で、`^GlobalsTest.PresidentI` となっています。
- `StreamLocation` は、あらゆるストリーム・プロパティが格納されるグローバルです。このグローバルの名前は、"S" が追加された完全なクラス名で、`^GlobalsTest.PresidentS` となっています。

クラスのオブジェクトをいくつか作成して保存した後は、ターミナルでこれらのグローバルのコンテンツを確認できます。

```
USER>zwrite ^GlobalsTest.PresidentD
^GlobalsTest.PresidentD=3
^GlobalsTest.PresidentD(1)=$lb("",1732,"1","Washington,George")
^GlobalsTest.PresidentD(2)=$lb("",1735,"2","Adams,John")
^GlobalsTest.PresidentD(3)=$lb("",1743,"3","Jefferson,Thomas")

USER>zwrite ^GlobalsTest.PresidentI
^GlobalsTest.PresidentI("DOBIndex",1732,1)=""
^GlobalsTest.PresidentI("DOBIndex",1735,2)=""
^GlobalsTest.PresidentI("DOBIndex",1743,3)=""
^GlobalsTest.PresidentI("NameIndex"," ADAMS,JOHN",2)=""
^GlobalsTest.PresidentI("NameIndex"," JEFFERSON,THOMAS",3)=""
^GlobalsTest.PresidentI("NameIndex"," WASHINGTON,GEORGE",1)=""

USER>zwrite ^GlobalsTest.PresidentS
^GlobalsTest.PresidentS=3
^GlobalsTest.PresidentS(1)="1,239"
^GlobalsTest.PresidentS(1,1)="George Washington was born to a moderately prosperous family of planters
 in colonial ..."
^GlobalsTest.PresidentS(2)="1,195"
^GlobalsTest.PresidentS(2,1)="John Adams was born in Braintree, Massachusetts, and entered Harvard
College at age 1..."
^GlobalsTest.PresidentS(3)="1,202"
^GlobalsTest.PresidentS(3,1)="Thomas Jefferson was born in the colony of Virginia and attended the
College of Willi..."
```

`^GlobalsTest.PresidentD` の添え字は IDKey です。いずれのインデックスも IDKey としては定義していないため、ID が IDKey として使用されます。ID の詳細は、"ID の生成を制御する方法" を参照してください。

`^GlobalsTest.PresidentI` の最初の添え字はインデックスの名前です。

`^GlobalsTest.PresidentS` の最初の添え字は、大統領の ID ではなく、略歴エントリの ID です。

これらのグローバルは、管理ポータルでも確認できます ([**システムエクスプローラ**]→[**グローバル**])。

重要　グローバル名では最初の 31 文字だけが重要であるため、完全なクラス名がきわめて長い場合は、^package1.pC347.VeryLongCla4F4AD のようなグローバル名が生成される場合があります。クラスのすべてのグローバル名が必ず一意になるように、このような名前が生成されます。クラスのグローバルで直接作業する場合、ストレージ定義を調べて、グローバルの実際の名前を確認してください。または、クラス定義内で DEFAULTGLOBAL パラメータを使用して、グローバル名を制御できます。"ユーザ定義のグローバル名" を参照してください。

11.2 ハッシュ化したグローバル名

USEEXTENTSET パラメータを値 1 に設定している場合、短いグローバル名が生成されます(このパラメータの既定値は 0、つまり標準のグローバル名の使用です)。こういった短いグローバル名は、パッケージ名のハッシュとクラス名のハッシュから作成され、後ろに接尾辞が付きます。標準の名前の方が読みやすいですが、短い名前の方がパフォーマンスの向上につながる可能性があります。

USEEXTENTSET を 1 に設定した場合、各インデックスも個別のグローバルに割り当てられます。最初の添え字が異なる単一のインデックス・グローバルは使用されません。繰り返しますが、これはパフォーマンス向上のために行われます。

前述の箇所で定義した GlobalsTest.President クラスにハッシュ化したグローバル名を使用するには、クラス定義に以下を追加します。

```
/// Use hashed global names
Parameter USEEXTENTSET = 1;
```

ストレージ定義を削除し、クラスをリコンパイルした後に、ハッシュ化されたグローバル名が含まれる以下のような新しいストレージ定義を確認できます。

```
Storage Default
{
...
<DataLocation>^Ebnm.EKUy.1</DataLocation>
<DefaultData>PresidentDefaultData</DefaultData>
<ExtentLocation>^Ebnm.EKUy</ExtentLocation>
<IdLocation>^Ebnm.EKUy.1</IdLocation>
<Index name="DOBIndex">
<Location>^Ebnm.EKUy.2</Location>
</Index>
<Index name="IDKEY">
<Location>^Ebnm.EKUy.1</Location>
</Index>
<Index name="NameIndex">
<Location>^Ebnm.EKUy.3</Location>
</Index>
<IndexLocation>^Ebnm.EKUy.I</IndexLocation>
<StreamLocation>^Ebnm.EKUy.S</StreamLocation>
<Type>%Storage.Persistent</Type>
}
```

特に、次のストレージ・キーワードに着目してください。

- ExtentLocation は、このクラスのグローバル名の計算に使用されるハッシュ値です (ここでは ^Ebnm.EKUy)。
- クラス・データが格納される DataLocation (IDKEY インデックスと等しい) は、名前に ".1" が付いたハッシュ値になっています (ここでは ^Ebnm.EKUy.1)。
- 各インデックスに固有の Location が指定されるため、各インデックスのグローバルも固有になります。IdKey インデックス・グローバルの名前は、名前に ".1" が付いたハッシュ値と等しいです。この例では、^Ebnm.EKUy.1 です。これ以外のインデックスのグローバルでは、名前に ".2" から ".N" が付けられます。ここでは、DOBIndex はグローバル ^Ebnm.EKUy.2 に格納され、NameIndex は ^Ebnm.EKUy.3 に格納されます。

- `IndexLocation` は、名前に ".I" が付いたハッシュ値、つまり `^Ebnm.EKUy.I` ですが、このグローバルは多くの場合使用されません。
- `StreamLocation` は、名前に ".S" が付いたハッシュ値、つまり `^Ebnm.EKUy.S` です。

オブジェクトをいくつか作成して保存した後、これらのグローバルのコンテンツは以下のようになります (ここでもターミナルを使用して確認します)。

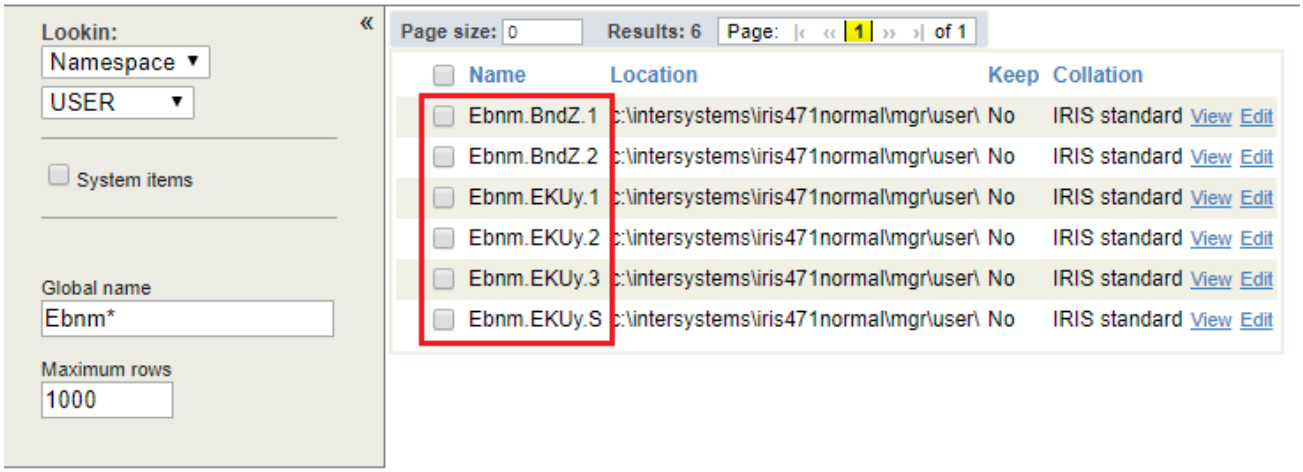
```
USER>zwrite ^Ebnm.EKUy.1
^Ebnm.EKUy.1=3
^Ebnm.EKUy.1(1)=$lb("","Washington,George",1732,"1")
^Ebnm.EKUy.1(2)=$lb("","Adams,John",1735,"2")
^Ebnm.EKUy.1(3)=$lb("","Jefferson,Thomas",1743,"3")

USER>zwrite ^Ebnm.EKUy.2
^Ebnm.EKUy.2(1732,1)=""
^Ebnm.EKUy.2(1735,2)=""
^Ebnm.EKUy.2(1743,3)=""

USER>zwrite ^Ebnm.EKUy.3
^Ebnm.EKUy.3(" ADAMS,JOHN",2)=""
^Ebnm.EKUy.3(" JEFFERSON,THOMAS",3)=""
^Ebnm.EKUy.3(" WASHINGTON,GEORGE",1)=""

USER>zwrite ^Ebnm.EKUy.S
^Ebnm.EKUy.S=3
^Ebnm.EKUy.S(1)="1,239"
^Ebnm.EKUy.S(1,1)="George Washington was born to a moderately prosperous family of planters in colonial
 ..."
^Ebnm.EKUy.S(2)="1,195"
^Ebnm.EKUy.S(2,1)="John Adams was born in Braintree, Massachusetts, and entered Harvard College at age
 1..."
^Ebnm.EKUy.S(3)="1,202"
^Ebnm.EKUy.S(3,1)="Thomas Jefferson was born in the colony of Virginia and attended the College of
Willi..."
```

SQL 文 CREATE TABLE を使用して定義したクラスの場合、USEEXTENTSET パラメータの既定値は 1 です。テーブルの作成の詳細は、"テーブルの定義" を参照してください。

例えば、管理ポータルを使用してテーブルを作成してみます ([**システムエクスプローラ**]→[SQL]→[**クエリ実行**])。

```
CREATE TABLE GlobalsTest.State (NAME CHAR (30) NOT NULL, ADMITYEAR INT)
```

テーブルにいくつかのデータを入力した後、管理ポータル ([**システムエクスプローラ**]→[**グローバル**]) でグローバル `^Ebnm.BndZ.1` および `^Ebnm.BndZ.2` を確認します。パッケージ名は GlobalsTest のままなので、GlobalsTest.State のグローバル名の最初のセグメントは、GlobalsTest.President の場合と同じです。



ターミナルで、グローバルのコンテンツは以下のように表示されます。

```
USER>zwrite ^Ebnm.BndZ.1
^Ebnm.BndZ.1=3
^Ebnm.BndZ.1(1)=$lb("Delaware",1787)
^Ebnm.BndZ.1(2)=$lb("Pennsylvania",1787)
^Ebnm.BndZ.1(3)=$lb("New Jersey",1787)

USER>zwrite ^Ebnm.BndZ.2
^Ebnm.BndZ.2(1)=$zwc(412,1,0)/*$bit(2..4)*/
```

グローバル ^Ebnm.BndZ.1 には州のデータ、^Ebnm.BndZ.2 にはビットマップ・エクステント・インデックスが格納されています。"ビットマップ・エクステント・インデックス" を参照してください。

SQL を使用して作成したクラスで標準のグローバル名を使用する場合、以下のように USEEXTENTSET パラメータを 0 に設定できます。

```
CREATE TABLE GlobalsTest.State (%CLASSPARAMETER USEEXTENTSET 0, NAME CHAR (30) NOT NULL, ADMITYEAR INT)
```

これによって、標準のグローバル名 ^GlobalsTest.StateD および ^GlobalsTest.StateI が生成されます。

注釈　コマンド do $SYSTEM.SQL.SetDDLUseExtentSet(0, .oldval) を実行することによって、CREATE TABLE 文によって作成されたクラスについて USEEXTENTSET パラメータに使用される既定値を 0 に変更することができます。以前の既定値は oldval で返されます。

XEP を使用して作成したクラスの場合、USEEXTENTSET パラメータの既定値は 1 で、これを変更することはできません。XEP の詳細は、"InterSystems XEP による Java オブジェクトの永続化" を参照してください。

11.3 ユーザ定義のグローバル名

クラスのグローバル名をより詳細に制御するには、DEFAULTGLOBAL パラメータを使用します。このパラメータは、USEEXTENTSET パラメータと共に機能して、グローバル名前付け方式を決定します。

例えば以下のように、DEFAULTGLOBAL パラメータを追加して、GlobalsTest.President クラスのグローバル名のルートを ^GT.Pres に設定してみます。

```
/// Use hashed global names
Parameter USEEXTENTSET = 1;

/// Set the root of the global names
Parameter DEFAULTGLOBAL = "^GT.Pres";
```

ストレージ定義を削除し、クラスをリコンパイルした後に、以下のようなグローバル名を確認できます。

```
Storage Default
{
...
<DataLocation>^GT.Pres.1</DataLocation>
<DefaultData>PresidentDefaultData</DefaultData>
<ExtentLocation>^GT.Pres</ExtentLocation>
<IdLocation>^GT.Pres.1</IdLocation>
<Index name="DOBIndex">
<Location>^GT.Pres.2</Location>
</Index>
<Index name="IDKEY">
<Location>^GT.Pres.1</Location>
</Index>
<Index name="NameIndex">
<Location>^GT.Pres.3</Location>
</Index>
<IndexLocation>^GT.Pres.I</IndexLocation>
<StreamLocation>^GT.Pres.S</StreamLocation>
<Type>%Storage.Persistent</Type>
}
```

同様に、DEFAULTGLOBAL パラメータは、SQL を使用したクラスの定義にも使用できます。

```
CREATE TABLE GlobalsTest.State (%CLASSPARAMETER USEEXTENTSET 0, %CLASSPARAMETER DEFAULTGLOBAL =
'^GT.State',
NAME CHAR (30) NOT NULL, ADMITYEAR INT)
```

これによって、グローバル名 ^GT.StateD および ^GT.StateI が生成されます。

11.4 列指向ストレージのグローバル

以下のクラス定義は、基本的に ″標準のグローバル名″ で取り上げている定義と同じですが、クラス・パラメータ STORAGEDEFAULT = "columnar" がある点が異なります。このパラメータは、そのクラスの既定のストレージ・レイアウトとして列指向ストレージを使用することを指定します。つまり、データを行単位で格納 (President の各プロパティを 1 行に格納) するのではなく、クラスの各プロパティのデータを 1 つの列に格納します。

```
Class ColumnarTest.President Extends %Persistent [ DdlAllowed, Final ]
{

/// Specify columnar storage at the class level
Parameter STORAGEDEFAULT = "columnar"

/// President's name (last,first)
Property Name As %String(PATTERN="1U.L1"","1U.L");

/// Year of birth
Property BirthYear As %Integer;

/// Short biography
Property Bio As %Stream.GlobalCharacter;

/// Index for Name
Index NameIndex On Name;

/// Index for BirthYear
Index DOBIndex On BirthYear;

}
```

このクラスをコンパイルすると、以下のようなストレージ定義が得られます。

```
Storage Default
{
<Data name="_CDM_Bio">
<Attribute>Bio</Attribute>
<Structure>vector</Structure>
</Data>
<Data name="_CDM_BirthYear">
<Attribute>BirthYear</Attribute>
<Structure>vector</Structure>
</Data>
<Data name="_CDM_Name">
<Attribute>Name</Attribute>
<Structure>vector</Structure>
</Data>
<DataLocation>^ColumnarTest.PresidentD</DataLocation>
<IdLocation>^ColumnarTest.PresidentD</IdLocation>
<IndexLocation>^ColumnarTest.PresidentI</IndexLocation>
<StreamLocation>^ColumnarTest.PresidentS</StreamLocation>
<Type>%Storage.Persistent</Type>
}
```

このストレージ定義は、従来の行ベースのストレージ・レイアウトを使用する標準のクラス定義に対して作成されたストレージ定義とは若干異なります。グローバルの名前は同じですが、列指向ストレージを使用したプロパティごとに <Data> 要素があります(″CDM″ は列指向のデータ・マップを表します)。<Data> タグの name 属性は ^ColumnarTest.PresidentI グローバルの添え字として使用されています。<Attribute> 要素にはプロパティ名が記述され、<Structure> 要素の vector の値は、プロパティの値を列指向形式で表すためにベクトルが使用されることを示します。

新しいクラスのオブジェクトをいくつか追加すると、このグローバルは以下のようになります。

```
USER>zwrite ^ColumnarTest.PresidentD
^ColumnarTest.PresidentD=3
^ColumnarTest.PresidentD(1)=""
^ColumnarTest.PresidentD(2)=""
^ColumnarTest.PresidentD(3)=""

USER>zwrite ^ColumnarTest.PresidentI
^ColumnarTest.PresidentI("$President",1)=$zwc(412,1,0)/*$bit(2..4)*/
^ColumnarTest.PresidentI("DOBIndex",1732,1)=""
^ColumnarTest.PresidentI("DOBIndex",1735,2)=""
^ColumnarTest.PresidentI("DOBIndex",1743,3)=""
^ColumnarTest.PresidentI("NameIndex"," ADAMS,JOHN",2)=""
^ColumnarTest.PresidentI("NameIndex"," JEFFERSON,THOMAS",3)=""
^ColumnarTest.PresidentI("NameIndex"," WASHINGTON,GEORGE",1)=""
^ColumnarTest.PresidentI("_CDM_Bio",1)={"type":"integer", "count":3, "length":4, "vector":[,1,2,3]}
; <VECTOR>
^ColumnarTest.PresidentI("_CDM_BirthYear",1)={"type":"integer", "count":3, "length":4,
"vector":[,1732,1735,1743]}  ; <VECTOR>
^ColumnarTest.PresidentI("_CDM_Name",1)={"type":"string", "count":3, "length":4,
"vector":[,"Washington,George","Adams,John","Jefferson,Thomas"]}  ; <VECTOR>

USER>zwrite ^GlobalsTest.PresidentS
^ColumnarTest.PresidentS=3
^ColumnarTest.PresidentS(1)="1,239"
^ColumnarTest.PresidentS(1,1)="George Washington was born to a moderately prosperous family of planters
 in colonial ..."
^ColumnarTest.PresidentS(2)="1,195"
^ColumnarTest.PresidentS(2,1)="John Adams was born in Braintree, Massachusetts, and entered Harvard
College at age 1..."
^ColumnarTest.PresidentS(3)="1,202"
^ColumnarTest.PresidentS(3,1)="Thomas Jefferson was born in the colony of Virginia and attended the
College of Willi..."
```

ここで、`^ColumnarTest.PresidentI("$President")` はビットマップ・エクステント・インデックスです。

列指向のストレージを使用したプロパティと行ベースのプロパティが混在する "混合ストレージ" を使用するクラスの場合、行ベースのストレージを使用するプロパティは、標準のクラス定義の場合と同様にデータ・グローバル (`^ColumnarTest.PresidentD` など) に格納されます。上記の例では、すべてのプロパティで列指向ストレージが使用されています。`STORAGEDEFAULT = "columnar"` が、プロパティ・レベルではなく、クラス・レベルで指定されているからです。列指向のストレージをプロパティ・レベルで指定したクラス定義の例は、"列指向のストレージの使用法" を参照してください。

ColumnarTest.President クラスにハッシュ化したグローバル名を使用するには、以下のクラス・パラメータを使用します。

```
/// Use hashed global names
Parameter USEEXTENTSET = 1;
```

このクラスをコンパイルすると、以下のようなストレージ定義が得られます。

```
Storage Default
{
<Data name="_CDM_Bio">
<Attribute>Bio</Attribute>
<ColumnarGlobal>^IEA0.EKUy.1.V1</ColumnarGlobal>
<Structure>vector</Structure>
</Data>
<Data name="_CDM_BirthYear">
<Attribute>BirthYear</Attribute>
<ColumnarGlobal>^IEA0.EKUy.1.V2</ColumnarGlobal>
<Structure>vector</Structure>
</Data>
<Data name="_CDM_Name">
<Attribute>Name</Attribute>
<ColumnarGlobal>^IEA0.EKUy.1.V3</ColumnarGlobal>
<Structure>vector</Structure>
</Data>
<DataLocation>^IEA0.EKUy.1</DataLocation>
<ExtentLocation>^IEA0.EKUy</ExtentLocation>
<IdLocation>^IEA0.EKUy.1</IdLocation>
<Index name="$President">
<Location>^IEA0.EKUy.2</Location>
</Index>
<Index name="DOBIndex">
<Location>^IEA0.EKUy.3</Location>
```

```
</Index>
<Index name="IDKEY">
<Location>^IEA0.EKUy.1</Location>
</Index>
<Index name="NameIndex">
<Location>^IEA0.EKUy.4</Location>
</Index>
<IndexLocation>^IEA0.EKUy.I</IndexLocation>
<StreamLocation>^IEA0.EKUy.S</StreamLocation>
<Type>%Storage.Persistent</Type>
}
```

このストレージ定義は、上記の列指向ストレージの例と、ハッシュ化したグローバル名の例を組み合わせたような定義です。ここでは、このクラス定義の各プロパティの <Data> 要素に <ColumnarGlobal> 属性があります。この属性を使用して、列指向形式でそのプロパティの値を格納するために使用するグローバルの名前を指定します。$President インデックスには、ビットマップ・エクステント・インデックスが格納されます。

新しいクラスのオブジェクトをいくつか追加すると、このグローバルは以下のようになります。

```
USER>zwrite ^IEA0.EKUy.1.V1
^IEA0.EKUy.1.V1(1)={"type":"integer", "count":3, "length":4, "vector":[,1,2,3]}  ; <VECTOR>

USER>zwrite ^IEA0.EKUy.1.V2
^IEA0.EKUy.1.V2(1)={"type":"integer", "count":3, "length":4, "vector":[,1732,1735,1743]}  ; <VECTOR>

USER>zwrite ^IEA0.EKUy.1.V3
^IEA0.EKUy.1.V3(1)={"type":"string", "count":3, "length":4,
"vector":[,"Washington,George","Adams,John","Jefferson,Thomas"]}  ; <VECTOR>

USER>zwrite ^IEA0.EKUy.1
^IEA0.EKUy.1=3
^IEA0.EKUy.1(1)=""
^IEA0.EKUy.1(2)=""
^IEA0.EKUy.1(3)=""

USER>zwrite ^IEA0.EKUy.2
^IEA0.EKUy.2(1)=$zwc(412,1,0)/*$bit(2..4)*/

USER>zwrite ^IEA0.EKUy.3
^IEA0.EKUy.3(1732,1)=""
^IEA0.EKUy.3(1735,2)=""
^IEA0.EKUy.3(1743,3)=""

USER>zwrite ^IEA0.EKUy.4
^IEA0.EKUy.4(" ADAMS,JOHN",2)=""
^IEA0.EKUy.4(" JEFFERSON,THOMAS",3)=""
^IEA0.EKUy.4(" WASHINGTON,GEORGE",1)=""

USER>zwrite ^IEA0.EKUy.S
^IEA0.EKUy.S=3
^IEA0.EKUy.S(1)="1,239"
^IEA0.EKUy.S(1,1)="George Washington was born to a moderately prosperous family of planters in colonial
 ..."
^IEA0.EKUy.S(2)="1,195"
^IEA0.EKUy.S(2,1)="John Adams was born in Braintree, Massachusetts, and entered Harvard College at age
 1..."
^IEA0.EKUy.S(3)="1,202"
^IEA0.EKUy.S(3,1)="Thomas Jefferson was born in the colony of Virginia and attended the College of
Willi..."
```

11.5 グローバル名の再定義

既存のグローバル名を再定義する方法（例えば、USEEXTENTSET パラメータまたは DEFAULTGLOBAL パラメータの値の変更）でクラス定義を編集する場合、既存のストレージ定義を削除して、コンパイラで新しいストレージ定義を生成できるようにする必要があります。既存のグローバル内のデータはすべて保持されます。保持されるデータをすべて新しいグローバル構造に移行する必要があります。

詳細は、"データを格納した永続クラスの再定義" を参照してください。

11.6 関連項目

- 永続オブジェクトの概要
- グローバルの使用法

12

永続オブジェクトを使用した作業

%Persistent クラスは、保存可能な (ディスクに書き込める) オブジェクトに対応した API です。ここでは、この API の使用方法について説明します。このトピックの情報は、%Persistent のすべてのサブクラスに適用されます。

このトピックで示されているサンプルのほとんどが、Samples-Data サンプル (https://github.com/intersystems/Samples-Data) からのものです。(例えば) SAMPLES という名前の専用ネームスペースを作成し、そのネームスペースにサンプルをロードすることをお勧めします。一般的な手順は、"サンプルのダウンロード" を参照してください。

12.1 オブジェクトの保存

オブジェクトをデータベースに保存するには、そのオブジェクトの %Save() メソッドを呼び出します。以下に例を示します。

ObjectScript

```
Set obj = ##class(MyApp.MyClass).%New()
Set obj.MyValue = 10

Set sc = obj.%Save()
```

%Save() メソッドは、保存操作が成功したか失敗したかを示す %Status 値を返します。例えば、オブジェクトが無効なプロパティ値を持つ場合や一意性に違反した場合、エラーが発生することがあります。"オブジェクトの検証" を参照してください。

オブジェクトで %Save() を呼び出すと、保存されているオブジェクトからアクセスできる、すべての変更済みオブジェクトが自動的に保存されます。つまり、すべての埋め込みオブジェクト、コレクション、ストリーム、参照オブジェクト、およびオブジェクトに関与するリレーションシップが必要に応じて自動的に保存されます。保存操作全体が 1 つのトランザクションとして実行されます。保存できないオブジェクトがある場合、トランザクション全体が失敗し、ロール・バックされます (ディスクには変更がなく、すべてのメモリ内オブジェクト値が、%Save() メソッドが呼び出される前の値に戻ります)。

初めてオブジェクトが保存されるとき、%Save() メソッドの既定の動作は、自動的にそのオブジェクトにオブジェクト ID 値を割り当てます。この ID 値は後に、データベース内でオブジェクトの検索時に使用されます。既定の場合、ID は、$Increment 関数を使用して生成されます。または、クラスは、idkey index を持つプロパティ値に基づくユーザ提供オブジェクト ID を使用できます (この場合、プロパティ値は、|| 文字列を含むことはできません)。一度割り当てられると、(これがユーザ提供の ID でも) 特定のオブジェクト・インスタンスに対するオブジェクト ID 値を変更できません。

%Id() メソッドを使用して、保存されたオブジェクトに対するオブジェクト ID を検索できます。

ObjectScript

```
// Open person "22"
Set person = ##class(Sample.Person).%OpenId(22)
Write "Object ID: ",person.%Id(),!
```

%Save() メソッドが行う詳細は、以下のとおりです。

1. 最初に、SaveSet として知られる一時的な構造を構築します。SaveSet は単純なグラフで、保存されたオブジェクトから到達可能なすべてのオブジェクトに対する参照を含みます (一般に、オブジェクト・クラス A に、別のオブジェクト・クラス B の値を保持しているプロパティがあれば、A のインスタンスは B のインスタンスに到達できます)。SaveSet の目的は、関連するオブジェクトの複雑なセットに関連する保存操作が、可能な限り効果的に実行できるようにすることです。SaveSet は、オブジェクト間の保存順序の依存関係も解決します。

   各オブジェクトが SaveSet に追加されると、その %OnAddToSaveSet() コールバック・メソッドが存在する場合は呼び出されます。

2. SaveSet 内の各オブジェクトを順番に検索し、それらが変更されていないかをチェックします (つまり、オブジェクトを開いた後で、または最後に保存されて以降、そのプロパティ値が変更されていないかをチェックします)。変更されているオブジェクトがある場合、そのオブジェクトは保存されます。

3. 保存される前に、変更された各オブジェクトは検証されます (そのプロパティ値がテストされます。その %OnValidateObject() メソッドが存在する場合は呼び出され、一意制約がテストされます)。オブジェクトが有効な場合、保存操作が継続します。無効なオブジェクトがある場合は、%Save() の呼び出しが失敗し、現在のトランザクションがロール・バックされます。

4. 各オブジェクトの保存前、または保存後、%OnBeforeSave() メソッドと %OnAfterSave() コールバック・メソッドが存在する場合は実行されます。同様に、定義されている BEFORE INSERT、BEFORE UPDATE、AFTER INSERT、および AFTER UPDATE のトリガがある場合は、それらも呼び出されます。

   これらのコールバックおよびトリガに Insert 引数が渡されます。この引数は、オブジェクトが挿入されているか (初めての保存の場合)、または更新されているかを示すものです。

   これらのコールバック・メソッドまたはトリガのいずれかが失敗した場合 (エラー・コードを返した場合)、%Save() への呼び出しは失敗し、現在のトランザクションがロール・バックされます。

5. 最後に、%OnSaveFinally() コールバック・メソッドが存在する場合は呼び出されます。

   このコールバック・メソッドは、トランザクションが完了し、保存のステータスが確定した後に呼び出されます。

現在のオブジェクトが変更されていない場合、ディスクには %Save() メソッドによって書き込みは行われません。オブジェクトの保存は必要がなく、そのため保存は失敗しないため、成功が返されます。 実際のところ、%Save() の返り値では、保存操作で要求がすべて実行されたか、それとも要求どおりには実行されなかったか (および明確にではなく何かがディスクに書き込まれたかどうか) が示されます。

重要　　マルチプロセス環境では、適切な並行処理の制御を使用していることを確認してください。"オブジェクト同時処理のオプション" を参照してください。

12.1.1 Rollback

%Save() メソッドは、SaveSet 内のすべてのオブジェクトを単独のトランザクションとして自動的に保存します。これらのオブジェクトのいずれかの保存が失敗した場合、トランザクション全体がロール・バックされます。このロールバックの場合、InterSystems IRIS は以下を実行します。

1. 割り当てられた ID を元に戻します。
2. 削除された ID を回復します。
3. 変更されたビットを回復します。
4. 正常にシリアル化されたオブジェクトに対して、%OnRollBack() コールバック・メソッドを呼び出します (実装されている場合)。

   InterSystems IRIS は、正常にシリアル化されていないオブジェクト (つまり、無効なオブジェクト) に対しては、このメソッドを呼び出しません。

12.1.2 オブジェクトとトランザクションの保存

前述したように、%Save() メソッドは、SaveSet 内のすべてのオブジェクトを単独のトランザクションとして自動的に保存します。これらのオブジェクトのいずれかの保存が失敗した場合、トランザクション全体がロール・バックされます。

複数の関連しないオブジェクトを 1 つのトランザクションとして保存する場合は、%Save() の呼び出しを 1 つの明示的なトランザクション内にまとめる必要があります。つまり、TSTART コマンドを使用してトランザクションを開始し、TCOMMIT コマンドを使用してトランザクションを終了する必要があります。

例：

ObjectScript

```
// start a transaction
TSTART

// save first object
Set sc = obj1.%Save()

// save second object (if first was save)
If ($$$ISOK(sc)) {
    Set sc = obj2.%Save()
}

// if both saves are ok, commit the transaction
If ($$$ISOK(sc)) {
    TCOMMIT
}
```

上記の例では、注意すべきことが 2 つあります。

1. %Save() メソッドは、トランザクション内で呼び出されたかどうかを知っています (システム変数 $TLEVEL が 0 より大きいため)。
2. トランザクション内の %Save() メソッドが失敗した場合、トランザクション全体がロール・バックされます (TROLLBACK コマンドが実行されます)。これは、アプリケーションは明示的なトランザクション内のすべての %Save() をテストし、いずれか 1 つが失敗したら他のオブジェクトでの %Save() の呼び出しをスキップし、最終的な TCOMMIT コマンドの呼び出しもスキップする必要があることを意味しています。

12.2 保存したオブジェクトの存在のテスト

特定のオブジェクト・インスタンスがデータベース内に保存されているかをテストするには、2 つの基本的な方法があります。

- ObjectScript を使用する
- SQL を使用する

これらの例では、ID は整数です (InterSystems IRIS が既定で生成する ID)。ID がオブジェクトの固有プロパティに基づくようにクラスを定義することもできます。

12.2.1 ObjectScript によるオブジェクトの存在のテスト

%ExistsId() クラス・メソッドは、指定された ID を確認します。指定されたオブジェクトがデータベース内に存在する場合は True (1) の値を返し、それ以外の場合は False (0) を返します。これは、**%Persistent** から継承されたすべてのクラスで使用できます。以下に例を示します。

ObjectScript

```
Write ##class(Sample.Person).%ExistsId(1),!   // should be 1
Write ##class(Sample.Person).%ExistsId(-1),!  // should be 0
```

ここでは、最初の行が 1 を返します。これは、Sample.Person が %Persistent を継承していて、SAMPLES データベースは、このクラスにデータを提供するためです。

%Exists() メソッドも使用できます。このメソッドには、ID ではなく OID が必要になります。OID はデータベースに対して一意の永続識別子で、オブジェクトの ID とクラス名の両方が含まれます。詳細は、"保存したオブジェクトの識別子：ID および OID" を参照してください。

12.2.2 SQL によるオブジェクトの存在のテスト

保存したオブジェクトの存在を SQL でテストするには、指定した ID と一致する %ID フィールドを持つ行を選択する SELECT 文を使用します (保存したオブジェクトの identity プロパティは、%ID フィールドとして投影されます)。

例えば、以下のように埋め込み SQL を使用します。

ObjectScript

```
&sql(SELECT %ID FROM Sample.Person WHERE %ID = '1')
Write SQLCODE,!  // should be 0: success

&sql(SELECT %ID FROM Sample.Person WHERE %ID = '-1')
Write SQLCODE,!  // should be 100: not found
```

ここでは、最初のケースは結果として 0 の SQLCODE を返します (成功を意味します)。これは、Sample.Person が %Persistent を継承していて、SAMPLES データベースは、このクラスにデータを提供するためです。

2 つ目のケースは、文の実行は成功したもののデータは返されないことを示す SQLCODE 100 になります。システムは 0 未満の ID を自動的には生成しないため、このように想定されます。

埋め込み SQL の詳細は、"埋め込み SQL の使用法" を参照してください。SQLCODE の詳細は、"InterSystems IRIS エラー・リファレンス" の "SQLCODE 値とエラー・メッセージ" を参照してください。

12.3 保存したオブジェクトのオープン

オブジェクトを開く (ディスクからメモリにオブジェクト・インスタンスをロードする) には、以下のように %OpenId() メソッドを使用します。

```
classmethod %OpenId(id As %String,
                    concurrency As %Integer = -1,
                    ByRef sc As %Status = $$$OK) as %ObjectHandle
```

以下はその説明です。

- id は、開くオブジェクトの ID です。

  この例では、ID は整数です。ID がオブジェクトの固有プロパティに基づくようにクラスを定義することもできます。

- concurrency は、オブジェクトを開くために使用する並行処理レベル (ロック) です。

- 参照渡しの sc は、%Status の値です。この値は、呼び出しが成功したか、失敗したかを表します。

指定のオブジェクトを開くことができる場合は、このメソッドが OREF を返します。オブジェクトが見つからない場合やオブジェクトが開けない場合は、NULL 値 ("") が返されます。

例：

ObjectScript

```
// Open person "10"
Set person = ##class(Sample.Person).%OpenId(10)

Write "Person: ",person,!    // should be an object reference

// Open person "-10"
Set person = ##class(Sample.Person).%OpenId(-10)

Write "Person: ",person,!    // should be a null string
```

%Open() メソッドも使用できます。このメソッドには、ID ではなく OID が必要になります。OID はデータベースに対して一意の永続識別子で、オブジェクトの ID とクラス名の両方が含まれます。詳細は、"保存したオブジェクトの識別子 : ID および OID" を参照してください。

オブジェクトが開くときに実行される追加の処理を行うには、%OnOpen() および %OnOpenFinally() のコールバック・メソッドを定義します。詳細は、"コールバック・メソッドの定義" を参照してください。

12.3.1 %OpenId() に対する複数の呼び出し

%OpenId() が 1 つの InterSystems IRIS プロセス内で同じ ID と同じオブジェクトに対して複数回呼び出された場合、メモリに作成されるオブジェクト・インスタンスは 1 つのみです。それ以降の %OpenId() の呼び出しはすべて、既にメモリにロードされているオブジェクトへの参照が返されます。

例えば、同じオブジェクトへの複数回の %OpenId() 呼び出しを特徴とする以下のクラスを見てみましょう。

Class Definition

```
Class User.TestOpenId Extends %RegisteredObject
{

ClassMethod Main()
{
   set A = ##class(Sample.Person).%OpenId(1)
   write "A: ", A.Name

   set A.Name = "David"
   write !, "A after set: ", A.Name

   set B = ##class(Sample.Person).%OpenId(1)
   write !, "B: ", B.Name

   do ..Sub()

   job ..JobSub()
   hang 1
   write !, "D in JobSub: ", ^JobSubD
   kill ^JobSubD

   kill A, B
   set E = ##class(Sample.Person).%OpenId(1)
   write !, "E:", E.Name
}

ClassMethod Sub()
{
   set C = ##class(Sample.Person).%OpenId(1)
   write !, "C in Sub: ", C.Name
}

ClassMethod JobSub()
{
   set D = ##class(Sample.Person).%OpenId(1)
   set ^JobSubD = D.Name
}

}
```

Main() メソッドを呼び出すと、以下のような結果になります。

1. 最初の %OpenId() 呼び出しは、インスタンス・オブジェクトをメモリにロードします。A 変数はこのオブジェクトを参照しており、この変数によって、ロードされたオブジェクトを変更できます。

ObjectScript

```
set A = ##class(Sample.Person).%OpenId(1)
write "A: ", A.Name
```

```
A: Uhles,Norbert F.
```

ObjectScript

```
set A.Name = "David"
write !, "A after set: ", A.Name
```

```
A after set: David
```

2. 2 つ目の %OpenId() 呼び出しは、データベースからオブジェクトを再ロードすることも、変数 A を使用して加えられた変更を上書きすることもありません。その代わり、変数 B が変数 A と同じオブジェクトを参照します。

ObjectScript

```
set B = ##class(Sample.Person).%OpenId(1)
write !, "B: ", B.Name
```

```
B: David
```

3. 3 つ目の %OpenId() 呼び出しは、Sub() メソッド内にあり、これも既にロードされているオブジェクトを参照します。このメソッドは、Main() メソッドと同じ InterSystems IRIS プロセスに含まれています。変数 C は、変数 A および変数 B と同じオブジェクトを参照します。ただし、変数 A と B はこのメソッドのスコープ内では使用できません。Sub() メソッドの最後に、プロセスは Main() メソッドに戻り、変数 C は破棄され、変数 A および B はスコープ内に戻ります。

ObjectScript

```
do ..Sub()
```

Class Member

```
ClassMethod Sub()
{
   set C = ##class(Sample.Person).%OpenId(1)
   write !, "C in Sub: ", C.Name
}
```

```
C in Sub: David
```

4. 4 つ目の %OpenId() 呼び出しは、JobSub() メソッド内にあり、JOB コマンドを使用して別個の InterSystems IRIS プロセスで実行されます。この新しいプロセスは、新しいインスタンス・オブジェクトをメモリにロードし、変数 D でこの新しいオブジェクトを参照します。

ObjectScript

```
job ..JobSub()
hang 1
write !, "D in JobSub: ", ^JobSubD
kill ^JobSubD
```

Class Member

```
ClassMethod JobSub()
{
   set D = ##class(Sample.Person).%OpenId(1)
   set ^JobSubD = D.Name
}
```

```
D in JobSub: Uhles,Norbert F.
```

5. 5 つ目の %OpenId() 呼び出しが実行されるのは、ロード済みのオブジェクトを元のプロセスで参照している変数 (A および B) をすべて削除し、それによってそのオブジェクトをメモリから削除した後です。直前の %OpenId() 呼び出しと同様に、この呼び出しも新しいインスタンス・オブジェクトをメモリにロードし、変数 E でこのオブジェクトを参照します。

   ObjectScript

   ```
   kill A, B
   set E = ##class(Sample.Person).%OpenId(1)
   write !, "E:", E.Name
   ```

   ```
   E: Uhles,Norbert F.
   ```

複数回ロードされるオブジェクトを強制的に再ロードするには、%Reload() メソッドを使用します。オブジェクトを再ロードすると、そのオブジェクトに対する未保存の変更は元に戻されます。以前に割り当てられていた変数は、再ロードされたバージョンを参照します。以下に例を示します。

ObjectScript

```
set A = ##class(Sample.Person).%OpenId(1)
write "A: ", A.Name // A: Uhles,Norbert F.

set A.Name = "David"
write !, "A after set: ", A.Name // David

set B = ##class(Sample.Person).%OpenId(1)
write !, "B before reload: ", B.Name // David

do B.%Reload()
write !, "B after reload: ", B.Name // Uhles,Norbert F.
write !, "A after reload: ", A.Name // Uhles,Norbert F.
```

12.3.1.1 並行処理

%OpenId() メソッドは、入力としてオプションの concurrency 引数を取ります。この引数は、オブジェクト・インスタンスを開くために使用する並行処理レベル (ロックのタイプ) を指定します。

%OpenId() メソッドがオブジェクトのロックを取得できない場合、メソッドは失敗します。

オブジェクトに対する現在の並行処理レベルの設定を上げ下げするには、そのオブジェクトを %OpenId() で再度開いて別の並行処理レベルを指定します。以下はその例です。

ObjectScript

```
set person = ##class(Sample.Person).%OpenId(6,0)
```

並行処理レベル 0 でperson を開き、以下のようにして並行処理レベルを事実上 4 まで引き上げます。

ObjectScript

```
set person = ##class(Sample.Person).%OpenId(6,4)
```

並行処理レベル 3 (共有保持ロック) または 4 (共有排他ロック) を指定すると、オブジェクトを開くときにそのオブジェクトが強制的に再ロードされます。可能なオブジェクト並行処理レベルの詳細は、"オブジェクト同時処理のオプション" を参照してください。

12.4 スウィズリング

永続オブジェクトのインスタンスを開いて(メモリにロードして)、そのインスタンスが参照するオブジェクトを使用する場合、この参照されるオブジェクトは自動的に開かれます。このプロセスをスウィズリングといいます。また、遅延ロードということもあります。

例えば、以下のコードは Sample.Employee オブジェクトのインスタンスを開き、ドット構文を使用して関連する Sample.Company オブジェクトを自動的にスウィズルします (開きます)。

ObjectScript

```
// Open employee "101"
Set emp = ##class(Sample.Employee).%OpenId(101)

// Automatically open Sample.Company by referring to it:
Write "Company: ",emp.Company.Name,!
```

オブジェクトをスウィズルすると、オブジェクトは、それをスウィズルするオブジェクトの並行処理値ではなく、それが属するクラスの既定の並行処理値を使用して開かれます。"オブジェクト同時処理のオプション" を参照してください。

スウィズルされたオブジェクトは、それを参照するオブジェクトや変数がなくなるとすぐに、メモリから削除されます。

オブジェクトを手動でスウィズルしたり、オブジェクトがスウィズルされているかどうか確認したい場合があります。これらのタスクのどちらにも、プロパティ・メソッド <prop>GetSwizzled() を使用できます。前述の例の Company プロパティの場合は、`CompanyGetSwizzled()` を呼び出すと、Company オブジェクトがスウィズルされ、オブジェクトの OREF が返されます。`CompanyGetSwizzled(1)` のように、1 を引数として渡すと、メソッドはオブジェクトの OREF を返しますが、オブジェクトが既にスウィズルされている場合に限られます。そうでない場合、NULL の OREF が返されます。

例として、ターミナルでは以下のようになります。

```
USER>set emp = ##class(Sample.Employee).%OpenId(102)

USER>set co = emp.CompanyGetSwizzled(1)

USER>if (co = "") {write "Company is not swizzled"} else {write co.Name}
Company is not swizzled
USER>set co = emp.CarGetSwizzled()

USER>if (co = "") {write "Company is not swizzled"} else {write co.Name}
Acme Company, Inc.
```

12.5 保存された値の読み取り

永続オブジェクトのインスタンスを開き、そのプロパティを変更したと仮定します。また、そのオブジェクトを保存する前に、データベースに保存された元の値を表示するとします。これを実行する最も簡単な方法は、生成されたプロパティ・メソッドの 1 つ、具体的には propertynameGetStored() を使用することです。以下に例を示します。

ObjectScript

```
// Open person "1"
Set person = ##class(Sample.Person).%OpenId(1)
Write "Original value: ",person.Name,!

// modify the object
Set person.Name = "Black,Jimmy Carl"
Write "Modified value: ",person.Name,!

// Now see what value is on disk
Set id = person.%Id()
Set name = ##class(Sample.Person).NameGetStored(id)
Write "Disk value: ",name,!
```

もう 1 つの簡単なオプションは、埋め込み SQL 文を使用することです (SQL は、メモリ内のオブジェクトではなく、常にデータベースに対して実行されます)。NameGetStored() を呼び出す代わりに、以下を使用します。

ObjectScript

```
// Now see what value is on disk
&sql(SELECT Name INTO :name
        FROM Sample.Person WHERE %ID = :id)

If (SQLCODE = 0) {Write "Disk value: ",name,!}
```

12.6 保存したオブジェクトの削除

永続インタフェースには、データベースからオブジェクトを削除するためのメソッドがあります。

12.6.1 %DeleteId() メソッド

%DeleteId() メソッドは、データベース内に保存されているオブジェクトを削除します。これには、オブジェクトに関連付けられているストリーム・データも含まれます。このメソッドのシグニチャは、以下のとおりです。

```
classmethod %DeleteId(id As %String, concurrency As %Integer = -1) as %Status
```

以下はその説明です。

- id は、開くオブジェクトの ID です。

  この例では、ID は整数です。ID がオブジェクトの固有プロパティに基づくようにクラスを定義することもできます。

- concurrency は、オブジェクト削除時に使用する並行処理レベル (ロック) です。

以下に例を示します。

```
Set sc = ##class(MyApp.MyClass).%DeleteId(id)
```

%DeleteId() は、オブジェクトが削除されたか否かを示す %Status 値を返します。

オブジェクトが削除される前に、%OnDelete() コールバック・メソッドが存在していれば、%DeleteId() はそれを呼び出します。%OnDelete() は %Status 値を返します。%OnDelete() がエラー値を返した場合、オブジェクトは削除されず、現在のトランザクションがロール・バックされ、%DeleteId() はエラー値を返します。

オブジェクトが削除された後に、%OnAfterDelete() コールバック・メソッドが存在していれば、%DeleteId() はそれを呼び出します。この呼び出しは、%DeleteData() がエラーを返さない限り、%DeleteData() が呼び出された直後に行われます。%OnAfterDelete() がエラーを返す場合、現在のトランザクションはロール・バックされ、%DeleteId() はエラー値を返します。

最後に、%DeleteId() は、%OnDeleteFinally() コールバック・メソッドが存在する場合は呼び出します。このコールバック・メソッドは、トランザクションが完了し、削除のステータスが確定した後に呼び出されます。

%DeleteId() メソッドは、メモリ内のオブジェクト・インスタンスには効果がありません。

%Delete() メソッドも使用できます。このメソッドには、ID ではなく OID が必要になります。OID はデータベースに対して一意の永続識別子で、オブジェクトの ID とクラス名の両方が含まれます。詳細は、"保存したオブジェクトの識別子：ID および OID" を参照してください。

12.6.2 %DeleteExtent() メソッド

%DeleteExtent() メソッドは、エクステント内のすべてのオブジェクト (およびオブジェクトのサブクラス) を削除します。具体的には、エクステント全体を反復処理して、オブジェクトの各インスタンスで %Delete() メソッドを起動します。既定では、すべてのインスタンスが正常に削除されると、%KillExtent() メソッドを呼び出します。

このメソッドのシグニチャは、以下のとおりです。

```
classmethod %DeleteExtent(concurrency As %Integer = -1, ByRef deletecount, ByRef instancecount,
pInitializeExtent As %Integer = 1, Output errorLog As %Status) as %Status
```

以下はその説明です。

- concurrency は、オブジェクト削除時に使用する並行処理レベル (ロック) です。
- deletecount は、削除されるインスタンスの数です。
- instancecount は、インスタンスの元の数です。
- pInitializeExtent は、エクステントが空の場合に %KillExtent() を呼び出すかどうかを決定します。
  - 0 の値は、%KillExtent() が呼び出されないことを意味します。一部の空のグローバルと ID カウンタ (存在する場合) は残る場合があります。
  - 1 の値は、%KillExtent() が呼び出されるが、既定のストリーム・グローバルは削除されないことを意味します。これが既定値です。
  - 2 の値は、%KillExtent() が呼び出され、既定のストリーム・グローバルが削除されることを意味します。

  注釈　エクステントにより使用されるグローバルがエクステントにより排他的に使用されない場合、一部のグローバルは %KillExtent() が呼び出されても残る場合があります。
- errorLog は、該当するエラーを返すために使用されます。

以下に例を示します。

```
Set sc = ##class(MyApp.MyClass).%DeleteExtent(-1, .deletecount, .instancecount, 2, .errorLog)
```

12.6.3 %KillExtent() メソッド

%KillExtent() メソッドは、クラスのエクステント、そのサブエクステント (サブクラス)、およびその子エクステント (そのクラスの子リレーションシップを持つクラスのエクステント) を格納するグローバルを直接削除します。既定のストリーム・ストレージ・グローバルはオプションで削除されます。%KillExtent() により %Delete() メソッドは呼び出されず、参照整合性アクションも実行されません。

このメソッドのシグニチャは、以下のとおりです。

```
classmethod %KillExtent(pDirect As %Integer = 1, killstreams As %Boolean = 0) as %Status
```

以下はその説明です。

- pDirect は、ファイリング・タイムスタンプも ^OBJ.DSTIME グローバルから削除されるかどうかを決定します。これは、InterSystems IRIS Business Intelligence で使用されます。"キューブの最新状態の維持" を参照してください。デフォルトは 1 です。
- killstreams は、既定のストリーム・グローバルが削除されるかどうかを決定します。デフォルトは 0 です。

以下に例を示します。

```
Set sc = ##class(MyApp.MyClass).%KillExtent(1, 0)
```

注意　%KillExtent() は、開発環境でのみ直接呼び出す必要があり、実際のアプリケーションでは使用しないでください。%KillExtent() は、制約やユーザ実装のコールバックを回避するため、データ整合性の問題を引き起こす可能性があります。アプリケーションでエクステント内のすべてのオブジェクトを削除する必要がある場合は、代わりに %DeleteExtent() を使用してください。

12.7 オブジェクト識別子のアクセス

オブジェクトが保存されている場合、そのオブジェクトには ID と OID があります。これらは、ディスクで使用する永久識別子です。オブジェクトの OREF があれば、その OREF を使用して、これらの識別子を取得できます。

OREF に関連付けられた ID を調べるには、オブジェクトの %Id() メソッドを呼び出します。以下に例を示します。

ObjectScript

```
write oref.%Id()
```

OREF に関連付けられた OID を調べる場合は、以下の 2 つの選択肢があります。

1. オブジェクトの %Oid() メソッドを呼び出します。以下に例を示します。

   ObjectScript

   ```
   write oref.%Oid()
   ```

2. オブジェクトの **%%OID** プロパティにアクセスします。このプロパティ名には % 文字が含まれているため、その名前を二重引用符で囲む必要があります。以下に例を示します。

   ObjectScript

   ```
   write oref."%%OID"
   ```

12.8 関連項目

- 永続オブジェクトの概要
- オブジェクト同時処理のオプション
- 永続クラスの定義
- 永続クラスのその他のオプション

13
オブジェクト同時処理のオプション

永続オブジェクトを開く場合や削除する場合には、適切に並行処理を指定することが重要です。

13.1 並行処理を指定する理由

以下のシナリオでは、オブジェクトの読み取り時または書き込み時に、適切に並行処理を制御することが重要になる理由についての例を示しています。以下のシナリオを考えてみます。

1. プロセス A は、並行処理を指定せずにオブジェクトを開きます。

```
SAMPLES>set person = ##class(Sample.Person).%OpenId(5)

SAMPLES>write person
1@Sample.Person
```

2. プロセス B は、同じオブジェクトを並行処理値 4 で開きます。

```
SAMPLES>set person = ##class(Sample.Person).%OpenId(5, 4)

SAMPLES>write person
1@Sample.Person
```

3. プロセス A がオブジェクトのプロパティを変更し、%Save() を使用してそれを保存しようして、エラー・ステータスを受け取ります。

```
SAMPLES>do person.FavoriteColors.Insert("Green")

SAMPLES>set status = person.%Save()

SAMPLES>do $system.Status.DisplayError(status)

ERROR #5803: Failed to acquire exclusive lock on instance of 'Sample.Person'
```

これは、適切な並行処理の制御がない場合の同時処理の例です。例えば、プロセス A がオブジェクトをディスクに保存する可能性がある場合、他のプロセスと競合せずにオブジェクトを保存できるように、並行処理 4 でオブジェクトを開く必要があります。(これらの値については次に説明します。)この場合、プロセス B はアクセスを拒否される (並行処理違反で失敗する)、またはプロセス A がオブジェクトを解放するまで待機する必要があります。

13.2 オプション

並行処理はいくつかの異なるレベルで指定できます。

1. 使用するメソッドの concurrency 引数を指定できます。

   %Persistent クラスの多くのメソッドで、この引数 (整数) を指定できます。この引数により、並行処理の制御に使用されるロックの方式が決定されます。並行処理の値は複数あります。

   メソッドの呼び出しで concurrency 引数の指定を省略すると、この引数はメソッドによって -1 に設定されます。つまり、操作するクラスに対して DEFAULTCONCURRENCY クラス・パラメータの値が使用されます。次の項目を参照してください。

2. 関連するクラスの DEFAULTCONCURRENCY クラス・パラメータを指定できます。すべての永続クラスは、プロセスに既定の並行処理を取得する式としてパラメータを定義する %Persistent から、このパラメータを継承します。次の項目を参照してください。

   クラス内で、このパラメータをオーバーライドして、ハードコードされた値を指定することも、独自のルールによって並行処理を決定する式を指定することもできます。どちらの場合も、パラメータの値は、指定可能な並行処理の値のいずれかにする必要があります。

3. プロセスの既定の並行処理モードを設定できます。そのためには、$system.OBJ.SetConcurrencyMode() メソッドを使用します (このメソッドは、%SYSTEM.OBJ クラスの SetConcurrencyMode() メソッドです)。

   その他の場合と同じように、指定可能な並行処理の値のいずれかを使用する必要があります。プロセスの並行処理モードを明示的に設定しない場合、既定値は 1 です。

   $system.OBJ.SetConcurrencyMode() メソッドは、DEFAULTCONCURRENCY クラス・パラメータに明示的な値を指定するクラスには効果がありません。

簡単な例を挙げてみましょう。永続オブジェクトを、その %OpenId() メソッドを使用して、並行処理の値を指定せずに (または値 -1 を明示的に指定して) 開いた場合、クラスで DEFAULTCONCURRENCY クラス・パラメータが指定されておらず、コードで並行処理モードを設定していないときは、オブジェクトの並行処理の既定値は 1 に設定されます。

注釈　システム・クラスによっては、DEFAULTCONCURRENCY クラス・パラメータが値 0 に設定されているものがあります。シャード・クラスでは常に並行処理の値 0 が使用されます。特定のオブジェクトの並行処理の値を確認するには、そのオブジェクトの %Concurrency プロパティを調べます。

13.3 並行処理の値

このセクションでは、有効な並行処理の値について説明します。まず、以下の詳細に注意してください。

- アトミック書き込みは、並行処理が 0 より大きいときに保証されます。
- InterSystems IRIS は、オブジェクトの保存や削除などの処理中にロックの取得と解放を実行します。詳細は、並行処理の値 (どのような制約があるのか、ロック・エスカレーションのステータス、およびストレージ構造) によって異なります。
- どのような場合でも、オブジェクトがメモリから削除されると、そのオブジェクトのロックも削除されます。

有効な並行処理の値は、以下のとおりです。このリストも示すように、それぞれの値には名前が付いています。

並行処理の値 0 (ロックなし)

ロックを使用しません。

並行処理の値 1 (アトミック読み込み)

ロックは必要に応じて取得および解放され、オブジェクトの読み込みがアトミック処理として実行されることを保証します。

新規オブジェクトの作成時に、InterSystems IRIS がロックを取得することはありません。

オブジェクトを開いている間は、InterSystems IRIS はオブジェクトに対する共有ロックを取得します (アトミック読み込みを保証するために必要な場合)。読み込み処理が完了すると、InterSystems IRIS はロックを解放します。

以下のテーブルに、各シナリオに現れるロックを示します。

	オブジェクトが作成されるとき	オブジェクトが開かれている間	オブジェクトが開かれた後	保存処理が完了した後
新規オブジェクト	ロックなし	N/A	N/A	ロックなし
既存のオブジェクト	N/A	共有ロック (アトミック読み込みを保証するために必要な場合)	ロックなし	ロックなし

並行処理の値 2 (共有ロック)

1 (アトミック読み込み) と同じですが、オブジェクトを開くときには常に共有ロックを取得します (アトミック読み込みを保証するために必要とされない場合でも、このロックを取得します)。以下のテーブルに、各シナリオに現れるロックを示します。

	オブジェクトが作成されるとき	オブジェクトが開かれている間	オブジェクトが開かれた後	保存処理が完了した後
新規オブジェクト	ロックなし	N/A	N/A	ロックなし
既存のオブジェクト	N/A	共有ロック	ロックなし	ロックなし

並行処理の値 3 (共有/保持ロック)

新規オブジェクトの作成時に、InterSystems IRIS がロックを取得することはありません。

既存のオブジェクトを開いている間は、InterSystems IRIS はオブジェクトの共有ロックを取得します。

新規オブジェクトの保存後、InterSystems IRIS はオブジェクトの共有ロックを保持します。

以下のテーブルは、シナリオのリストです。

	オブジェクトが作成されるとき	オブジェクトが開かれている間	オブジェクトが開かれた後	保存処理が完了した後
新規オブジェクト	ロックなし	N/A	N/A	共有ロック
既存のオブジェクト	N/A	共有ロック	共有ロック	共有ロック

並行処理の値 4 (排他/保持ロック)

既存のオブジェクトが開かれるとき、または新規オブジェクトが初めて保存されるときに、InterSystems IRIS は排他ロックを取得します。

以下のテーブルは、シナリオのリストです。

	オブジェクトが作成されるとき	オブジェクトが開かれている間	オブジェクトが開かれた後	保存処理が完了した後
新規オブジェクト	ロックなし	N/A	N/A	排他ロック
既存のオブジェクト	N/A	排他ロック	exclusive lock (排他ロック)	exclusive lock (排他ロック)

並行処理の値 -1 (既定の並行処理の値を使用)

並行処理の値を明示的に指定しない場合、値は -1 に設定されます。これは、既定の並行処理の値が使用されることを意味します。既定の並行処理の値の決定方法に関する詳細は、"オブジェクト同時処理のオプション" を参照してください。

13.4 並行処理とスウィズルされたオブジェクト

プロパティによって参照されるオブジェクトは、スウィズルされるオブジェクトのクラスで定義された既定の並行処理を使用してアクセス時にスウィズルされます。そのクラスの既定値が定義されていない場合、オブジェクトは、プロセスの既定の並行処理モードを使用してスウィズルされます。(詳細は、"オブジェクト同時処理のオプション" を参照してください。) スウィズルされるオブジェクトは、それをスウィズルするオブジェクトの並行処理値は使用しません。

スウィズルされるオブジェクトが既にメモリ内にある場合は、実際にスウィズルによってオブジェクトが開かれることはありません。それは、既存のメモリ内のオブジェクトを参照するだけであり、その場合、そのオブジェクトの現在の状態が保持され、並行処理は変更されません。

この既定の動作をオーバーライドするには、以下の 2 つの方法があります。

- スウィズルされたオブジェクトに対する同時処理を、新しい同時処理を指定する %Open() メソッドの呼び出しでアップグレードします。以下に例を示します。

  ObjectScript

  ```
  Do person.Spouse.%OpenId(person.Spouse.%Id(),4,.status)
  ```

  %OpenId() の最初の引数では ID を指定し、2 番目の引数では新しい並行処理を指定し、3 番目の引数 (参照渡し) はメソッドのステータスを受け取ります。

- オブジェクトをスウィズルする前に、プロセスの既定の並行処理モードを設定します。以下に例を示します。

  ObjectScript

  ```
  Set olddefault = $system.OBJ.SetConcurrencyMode(4)
  ```

  このメソッドは、その引数として新しい同時処理モードを取り、前の同時処理モードを返します。

  異なる並行処理モードが必要なくなった場合は、以下のように既定の並行処理モードをリセットします。

  ObjectScript

  ```
  Do $system.OBJ.SetConcurrencyMode(olddefault)
  ```

13.5 バージョン確認 (並行処理引数の代替)

オブジェクトを開いたり削除したりするときに、並行処理引数を指定する代わりに、バージョン確認を実装できます。そのためには、VERSIONPROPERTY というクラス・パラメータを指定します。すべての永続クラスに、このパラメータがあります。永続クラスを定義するときに、バージョン確認を有効にする手順は次のとおりです。

1. 更新可能なバージョン番号を保持した %Integer 型のプロパティを、クラスのインスタンスごとに作成します。
2. そのプロパティについては、InitialExpression キーワードの値を 0 に設定します。
3. クラスについては、そのプロパティの名前と等しくなるように、VERSIONPROPERTY クラス・パラメータの値を設定します。VERSIONPROPERTY の値は、サブクラスからは別のプロパティに変更できません。

これでバージョン確認が、クラス内のインスタンスへの更新に組み込まれます。

バージョン確認が実装されると、クラス内のインスタンスがオブジェクトまたは SQL で更新されるたびに VERSIONPROPERTY で指定したプロパティが自動的にインクリメントされます。プロパティがインクリメントされる前に、InterSystems IRIS によってメモリ内の値と保存されている値が比較されます。この値が異なる場合は、同時処理の競合が起こりエラーが表示されます。この値が同じ場合は、プロパティがインクリメントされて保存されます。

注釈　この機能一式を使用して、楽観的同時実行制御を実装できます。

VERSIONPROPERTY が InstanceVersion と呼ばれるプロパティを指す場合、クラスの SQL 更新文に同時処理の確認を実装するには、コードを次のように設定することになります。

```
SELECT InstanceVersion,Name,SpecialRelevantField,%ID
    FROM my_table
    WHERE %ID = :myid

    // Application performs operations on the selected row

UPDATE my_table
    SET SpecialRelevantField = :newMoreSpecialValue
    WHERE %ID = :myid AND InstanceVersion = :myversion
```

ここで、myversion は、元のデータで選択したバージョンのプロパティ値を指します。

13.6 関連項目

- 永続オブジェクトの概要
- 永続オブジェクトを使用した作業
- ロックと並行処理の制御

14

永続クラスの定義

永続クラスは、永続オブジェクトを定義するクラスです。ここでは、このようなクラスの作成方法について説明します。

このページで示されているサンプルは、Samples-Data (https://github.com/intersystems/Samples-Data) のものです。(例えば) SAMPLES という名前の専用ネームスペースを作成し、そのネームスペースにサンプルをロードすることをお勧めします。一般的な手順は、"サンプルのダウンロード" を参照してください。

14.1 永続クラスの定義

永続オブジェクトを定義するクラスを定義するには、そのクラスの プライマリ (最初の) スーパークラスを %Persistent にするかその他の永続クラスであることを確認してください。

例 :

Class Definition

```
Class MyApp.MyClass Extends %Persistent
{
}
```

14.2 パッケージからスキーマへのプロジェクション

永続クラスの場合は、パッケージが SQL スキーマとして SQL で表現されます。例えば、クラスが Team.Player (Team パッケージの Player クラス) である場合、対応するテーブルは Team.Player (Team スキーマの Player テーブル) です。

既定のパッケージは User です。これが SQLUser スキーマとして SQL で表現されます。したがって、User.Person という名前のクラスは、SQLUser.Person という名前のテーブルに対応します。

パッケージ名にピリオドが含まれる場合、ピリオドの代わりにアンダースコア (_) を使用します。例えば、MyTest.Test.MyClass クラス (MyTest.Test パッケージの MyClass クラス) は、MyTest_Test.MyClass テーブル (MyTest_Test スキーマの MyClass テーブル) になります。

SQL テーブル名がスキーマ名なしで参照される場合、既定のスキーマ名 (SQLUser) が使用されます。例えば、以下のコマンドがあります。

SQL

```
Select ID, Name from Person
```

上記は、以下と同じです。

SQL

```
Select ID, Name from SQLUser.Person
```

14.3 永続クラスのテーブル名の指定

永続クラスの場合は、短いクラス名がテーブル名になります (既定)。

別のテーブル名を指定する場合は、SqlTableName クラス・キーワードを使用します。以下に例を示します。

```
Class App.Products Extends %Persistent [ SqlTableName = NewTableName ]
```

InterSystems IRIS には、クラス名に対する制限はありませんが、SQL 予約語を SQL テーブルの名前に使用することはできません。そのため、予約語の名前を付けた永続クラスを作成すると、クラス・コンパイラによりエラー・メッセージが生成されます。この場合は、ここで説明する方法を使用して、クラス名を変更するか、クラス名とは異なるテーブル名をプロジェクションに指定する必要があります。

14.4 ストレージ定義とストレージ・クラス

%Persistent クラスは、データベース内のオブジェクト保存と検索用の、高レベルのインタフェースを提供します。オブジェクトの保存とロードの実際の作業は、ストレージ・クラスによって実行されます。

すべての永続オブジェクトおよびすべてのシリアル・オブジェクトはストレージ・クラスを使用して、データベースのオブジェクトの保存、ロード、および削除に使用される実際のメソッドを生成します。これらの内部メソッドは、ストレージ・インタフェースと呼ばれます。ストレージ・インタフェースには、%LoadData()、%SaveData()、および %DeleteData() などのメソッドが含まれます。これらのメソッドはアプリケーションから直接呼ばれることはなく、永続インタフェース (%OpenId() や %Save() など) のメソッドによって適宜呼び出されます。

永続クラスによって使用されるストレージ・クラスは、ストレージ定義によって指定されます。ストレージ定義には、ストレージ・インタフェースによって使用される追加のパラメータや、ストレージ・クラスを定義する、一連のキーワードや値が含まれます。

永続クラスには、1 つ以上のストレージ定義を含みますが、一度に 1 つしかアクティブになりません。アクティブなストレージ定義は、クラスの StorageStrategy キーワードを使用して指定されます。既定では、永続クラスは Default と呼ばれるストレージ定義を 1 つ持っています。

クラスのデータを格納するグローバルの名前の詳細は、"グローバル" を参照してください。

14.4.1 ストレージ定義の更新

クラスのストレージ定義は、クラスを初めてコンパイルしたときに作成されます。SQL などのためのクラス・プロジェクションは、コンパイル後に実行されます。クラスが正しくコンパイルされ、その後、プロジェクションに失敗した場合、InterSystems IRIS は、ストレージ定義を削除しません。また、クラスの変更がストレージ定義に影響を及ぼす可能性がある場合、アプリケーション開発者の責任により、ストレージ定義が更新されているか判断し、必要に応じて、ストレージ定義を変更して変更を反映します。"ストレージ定義の再設定" を参照してください。

14.4.2 %Storage.Persistent ストレージ・クラス

%Storage.Persistent は、永続オブジェクトから使用される既定のストレージ・クラスです。これは、永続クラスに対する既定のストレージ構造を自動的に作成し、保持します。

新しい永続クラスは、自動的に %Storage.Persistent ストレージ・クラスを使用します。%Storage.Persistent クラスでは、ストレージ定義のさまざまなキーワードを使用して、クラスに使用されるストレージ構造の特定の機能を制御できます。

さまざまなストレージ・キーワードに関する詳細は、"クラス定義リファレンス" を参照してください。

また、MANAGEDEXTENT クラス・パラメータの詳細は、"エクステント定義" を参照してください。

14.4.3 Storage.SQL ストレージ・クラス

%Storage.SQL クラスは、生成された SELECT、INSERT、UPDATE、および DELETE の各 SQL 文を使用してオブジェクトの永続性を提供する特別なストレージ・クラスです。

%Storage.SQL は一般的に、以下の用途で使用されます。

- 従来のアプリケーションによって使用された、以前に存在したグローバル構造へのオブジェクトのマッピング
- SQL ゲートウェイを使用した、外部リレーショナル・データベース内のオブジェクトの保存

%Storage.SQL は、%Storage.Persistent よりも制限されています。特に、複数のクラスのエクステントでのスキーマ進化の自動的なサポートを行いません。

14.5 スキーマ進化

%Storage.Persistent ストレージ・クラスは、自動的なスキーマ進化をサポートします。

既定の %Storage.Persistent ストレージ・クラスを使用する永続クラス (シリアル・クラス) をコンパイルするとき、クラス・コンパイラはクラスによって定義されたプロパティを解析し、自動的にそのプロパティを追加または削除します。

14.6 ストレージ定義の再設定

開発プロセス中は、永続クラスにプロパティの追加、変更、および削除などの変更をいくらでも加えることができます。クラス・コンパイラは互換性構造を維持しようと試みるため、ストレージ定義はより複雑になっていきます。すっきりしたストレージ構造がクラス・コンパイラで生成されるようにするには、クラスからストレージ定義を削除してから、そのクラスをリコンパイルします。

14.7 ID の生成を制御する方法

オブジェクトを初めて保存するときに、そのオブジェクトの ID はシステムによって生成されます。ID は永続的になります。

既定では、InterSystems IRIS は ID に整数を使用します。この整数は、最後に保存したオブジェクトから 1 だけ増分されます。

特定の永続クラスを定義する際に、以下のいずれかの方法で ID が生成されるように定義できます。

- ID は、クラスの特定のプロパティを基にすることができます (そのプロパティがインスタンスごとに一意である場合)。例えば、ドラッグ・コードを ID として使用することも可能です。この方法でクラスを定義するには、以下のようなインデックスをクラスに追加します。

  ```
  Index IndexName On PropertyName [ IdKey ];
  ```

  または (同様に)

  ```
  Index IndexName On PropertyName [ IdKey, Unique ];
  ```

  IndexName はインデックスの名前、PropertyName はプロパティの名前です。

  この方法でクラスを定義すると、InterSystems IRIS は最初にオブジェクトを保存するときに、そのプロパティの値を ID として使用するようになります。さらに、InterSystems IRIS はプロパティの値を要求して、そのプロパティの一意性を強制します。指定されたプロパティに同じ値を持つ別のオブジェクトを作成して、新しいオブジェクトを保存しようとすると、InterSystems IRIS は以下のエラーを発行します。

  ```
  ERROR #5805: ID key not unique for extent
  ```

  さらに、InterSystems IRIS により、そのプロパティは将来にわたって変更できなくなります。つまり、保存したオブジェクトを開き、そのプロパティ値を変更し、変更したオブジェクトを保存しようとすると、InterSystems IRIS は以下のエラーを発行するようになります。

  ```
  ERROR #5814: Oid previously assigned
  ```

  このメッセージは、ID ではなく OID について言及しています。これは、基礎となるロジックが OID の変更を防止しているためです。OID は ID を基にしています。

- ID は複数のプロパティに基づいたものにできます。この方法でクラスを定義するには、以下のようなインデックスをクラスに追加します。

  ```
  Index IndexName On (PropertyName1,PropertyName2,...) [ IdKey, Unique ];
  ```

  または (同様に)

  ```
  Index IndexName On (PropertyName1,PropertyName2,...) [ IdKey ];
  ```

  IndexName はインデックスの名前、PropertyName1、PropertyName2 などはプロパティの名前です。

  この方法でクラスを定義すると、InterSystems IRIS は最初にオブジェクトを保存するときに、以下のように ID を生成するようになります。

  ```
  PropertyName1||PropertyName2||...
  ```

  さらに、InterSystems IRIS はプロパティの値を要求して、プロパティの指定の組合せの一意性を強制します。さらに、それらのプロパティはいずれも将来にわたって変更できなくなります。

重要　リテラル・プロパティ (つまり、属性) に、連続する 1 対の垂直バー (||) が含まれている場合、そのプロパティを使用する IdKey インデックスは追加しないでください。この制限は、InterSystems SQL のメカニズムが動作するための方法に起因しています。IdKey プロパティ で || を使用すると、予測できない動作を起こす場合があります。

システムは OID も生成します。いずれの場合でも、OID は以下の形式になります。

```
$LISTBUILD(ID,Classname)
```

ID は生成された ID です。また、Classname はクラスの名前です。

14.8 サブクラスの SQL プロジェクションの制御

スーパークラス/サブクラスの階層内に複数の永続クラスが存在する場合、InterSystems IRIS は、それらのデータを 2 通りの方法で保存します。既定のシナリオは、極めて一般的です。

14.8.1 サブクラスの既定の SQL プロジェクション

クラス・コンパイラは、永続クラスを平坦化した表現を投影します。投影されたテーブルには、そのクラスの該当するフィールドのすべて (継承されたフィールドを含む) が含まれるようになります。したがって、サブクラスの SQL プロジェクションは、以下の項目で構成されたテーブルになります。

- スーパークラスのエクステントに含まれるすべての列
- サブクラスだけにあるプロパティに基づいたその他の列
- サブクラスの保存済みインスタンスを表す行

さらに、既定のシナリオでは、スーパークラスのエクステントには、スーパークラスおよびそのスーパークラスのすべてのサブクラスの保存済みオブジェクトごとに 1 つのレコードが格納されます。各サブクラスのエクステントは、スーパークラスのエクステントのサブセットです。

例えば、SAMPLES 内には、永続クラスの Sample.Person と Sample.Employee があるとします。Sample.Employee クラスは Sample.Person を継承し、いくつかのプロパティが追加されます。SAMPLES では、どちらのクラスもデータを保存しています。

- Sample.Person の SQL プロジェクションは、Sample.Person クラスの適切なプロパティをすべて含んでいるテーブルになります。Sample.Person テーブルには、Sample.Person クラスの保存済みインスタンスごとに 1 つのレコード、および Sample.Employee クラスの保存済みインスタンスごとに 1 つのレコードが格納されています。
- Sample.Employee テーブルには、Sample.Person と同じ列が含まれています。さらに、Sample.Employee クラスに固有の列も含まれています。Sample.Employee テーブルには、Sample.Employee クラスの保存済みインスタンスごとに 1 つのレコードが格納されています。

これを表示するには、以下の SQL クエリを使用します。まず、Sample.Person のすべてのインスタンスをリストして、それらのプロパティを表示します。

SQL

```
SELECT * FROM Sample.Person
```

2 番目のクエリでは、Sample.Employee のすべてのインスタンスと、それらのプロパティをリストします。

SQL

```
SELECT * FROM Sample.Employee
```

Sample.Person テーブルには、1 から 200 の範囲の ID を持つレコードが含まれることに注意してください。101 から 200 までの範囲に含まれる ID を持つレコードは従業員であり、Sample.Employee テーブルは同じ従業員 (同じ ID と同じ追加列を持つ従業員) を示します。Sample.Person テーブルには、2 つの見かけ上のグループのみが配置されています。これは、SAMPLES データベースが人為的な方法で構築されているためです。Sample.Person テーブルが移入された後で、Sample.Employee テーブルが移入されます。

一般的に、サブクラスのテーブルには親クラスよりも多数の列と、少数の行が含まれます。サブクラスが親クラスを拡張するとき、サブクラスは通常、プロパティを追加するので、サブクラス内にはより多くの列があります。サブクラスには親よりもサブクラスのインスタンスが少ないので、多くの場合行数も少なくなります。

14.8.2 サブクラスの代替の SQL プロジェクション

既定のプロジェクションは最も便利なものですが、場合によっては、代替の SQL プロジェクションを使用することが必要になります。このシナリオでは、各クラスに独自のエクステントがあります。この形式のプロジェクションを行うには、スーパークラスの定義に以下を含めます。

```
[ NoExtent ]
```

例：

Class Definition

```
Class MyApp.MyNoExtentClass [ NoExtent ]
{
//class implementation
}
```

このクラスの各サブクラスは、独自のエクステントを受け取ります。

この方法でクラスを作成し、その他のクラスのプロパティとして使用する場合は、"バリエーション：CLASSNAME パラメータ" を参照してください。

14.9 データを格納した永続クラスの再定義

開発プロセス中にクラスを再定義することはよくあることです。既にクラスのサンプル・データを作成している場合は、以下の点に注意してください。

- コンパイラはクラスのデータを格納するグローバルに影響しません。

  実際には、クラス定義を削除してもクラス定義のデータ・グローバルはそのまま残されます。これらのグローバルが不要になった場合は、手動で削除してください。

- クラスのプロパティを追加または削除して、クラスのストレージ定義は変更しない場合、そのクラスのデータにアクセスするコードはすべて、これまでと同様に機能します。"スキーマ進化" を参照してください。

- クラスのストレージ定義を変更する場合、データにアクセスするコードがこれまでと同様に機能するかしないかは、変更の種類によって異なります。

- シャード化されていないものからシャード化されているものに、またはその逆にクラスを変更した場合、既存のデータにアクセスできなくなることがあります。

- プロパティ検証がより制限的になるようにプロパティ定義を変更した場合、検証に合格しなくなったオブジェクト（またはレコード）で作業するとエラーが発生します。例えば、プロパティの MAXLEN パラメータを小さくすると、長すぎることになったそのプロパティの値を持っているオブジェクトで作業したときに、検証エラーが発生します。

14.10 関連項目

- クラスの定義
- 永続オブジェクトの概要
- 永続オブジェクトを使用した作業
- 永続クラスのその他のオプション

15
リテラル・プロパティの定義と使用

リテラル・プロパティは、あらゆるオブジェクト・クラスで定義できます。リテラル・プロパティは、リテラル値を保持し、データ型クラスに基づいています。ここでは、このようなプロパティを定義して使用する方法について説明します。

前述したように、トピックのいくつかは、他の種類のプロパティにも適用できます。

15.1 リテラル・プロパティの定義

リテラル・プロパティをクラス定義に追加するには、以下のいずれかのような要素をクラスに追加します。

Class Member

```
Property PropName as Classname;

Property PropName as Classname [ Keywords ] ;

Property PropName as Classname(PARAM1=value,PARAM2=value) [ Keywords ] ;

Property PropName as Classname(PARAM1=value,PARAM2=value) ;
```

以下はその説明です。

- PropName は、プロパティの名前です。
- Classname は、このプロパティが基づくクラスです。これを省略すると、Classname は **%String** であると見なされます。このプロパティをリテラル・プロパティとして定義するには、Classname を省略するか、Classname をデータ型クラスの名前として指定します。"一般的なデータ型クラス" を参照してください。カスタムのデータ型クラスも使用できます。
- Keywords はプロパティ・キーワードを表します。"コンパイラ・キーワードの概要" を参照してください。
- PARAM1、PARAM2 などはプロパティ・パラメータです。使用可能なパラメータは、プロパティで使用されるクラスによって異なります。

プロパティ・パラメータを含める場合は、パラメータを括弧で囲み、プロパティ・キーワードの前に置きます。プロパティ・キーワードを含める場合、キーワードは角括弧で囲みます。

15.1.1 例

例えば、クラスでは **%Integer** データ型クラスを使用して、**Count** プロパティを定義できます。

Class Member

```
Property Count As %Integer;
```

%Integer がデータ型クラスなので、Count はデータ型プロパティになります。

データ型パラメータを使用して、データ型プロパティの許容値を制限できます。例えば、タイプ %Integer のプロパティには、MAXVAL パラメータを指定できます。

Class Member

```
Property Count As %Integer(MAXVAL = 100);
```

データ型プロパティ値を空文字列に設定するには、以下の形式のコードを使用します。

ObjectScript

```
 Set object.Property = $C(0)
```

すべてのプロパティに照合タイプがあります。これにより、値を並べる方法（先頭文字の大文字化が有効かどうかなど）が決まります。文字列の既定の照合タイプは、SQLUPPER です。照合の詳細は、"データ照合" を参照してください。

15.2 プロパティの初期値式の定義

既定では、新しいオブジェクトを作成したときに、それぞれのプロパティは NULL になります。その代わりに、特定のプロパティの初期値として使用するように、ObjectScript 式を指定できます。式はオブジェクトが作成されるときに評価されます。

プロパティの初期値式を指定するには、プロパティ定義に InitialExpression キーワードを含めます。

```
Property PropName as Classname [ InitialExpression=expression ] ;
```

この expression は ObjectScript 式です（この式には、別の言語を使用できない点に注意してください）。詳細は、"Initial-Expression" を参照してください。

15.3 必須としてのプロパティの定義

このオプションは、永続クラスのプロパティにのみ適用されます。（このオプションはリテラル・プロパティだけでなく、任意の種類のプロパティに適用されます。）

既定では、新しいオブジェクトを保存するときに、InterSystems IRIS は、そのオブジェクトのプロパティが NULL 以外の値を持つことを必要としません。プロパティは、NULL 以外の値を必要とするように定義できます。そのためには、プロパティ定義に Required キーワードを含めます。

```
Property PropName as Classname [ Required ] ;
```

これにより、プロパティに NULL 値を持つオブジェクトを保存しようとすると、InterSystems IRIS は以下のエラーを発行するようになります。

```
ERROR #5659: Property required
```

Required キーワードは SQL アクセスを使用したこのクラスのデータの挿入または変更の方法にも影響を及ぼします。特に、レコードの挿入または更新をする際に値が欠落しているとエラーが発生します。

15.4 計算プロパティの定義

InterSystems IRIS では、計算プロパティを定義できます。計算プロパティの値は ObjectScript で計算されます (他のプロパティに基づく場合もあります)。一般的なフレーズの計算プロパティ (または計算フィールド) には、以下のバリエーションの両方が含まれます。

- Always computed (常に計算) ー このプロパティの値は、アクセスされるときに計算されます。インデックスを定義していない限り、データベースに格納されません。インデックスを定義していれば、そのインデックスが格納されます。
- Triggered computed (トリガによって計算) ー このプロパティの値は、トリガされたときに再計算されます (詳細は下記を参照)。

  プロパティが永続クラスで定義される場合、その値はデータベースに保存されます。

どちらの場合も、オブジェクト・アクセスや SQL クエリを使用するかどうかに関係なく、再計算が実行されます。

プロパティ・キーワードは 6 つあります (SqlComputed、SqlComputeCode、SqlComputeOnChange、Transient、Calculated、ComputeLocalOnly)。これらはプロパティ計算の是非と、計算方法を制御します。SqlComputeCode を指定する代わりに、*Property*Computation メソッドに計算コードを記述できます。Property は計算プロパティの名前です。詳細は、"計算プロパティ値" を参照してください。

以下のテーブルに可能な組合せをまとめます。

テーブル 15-1: プロパティが計算されるかどうかを判断する方法

		SqlComputed は true (SqlComputeCode は定義済み)	SqlComputed は false
Calculated は true	Transient は true	プロパティは常に計算されます	プロパティは計算されません
Calculated は true	Transient は false	プロパティは常に計算されます	プロパティは計算されません
Calculated は false	Transient は true	プロパティは常に計算されます	プロパティは計算されません
Calculated は false	Transient は false	プロパティはトリガによって計算されます (この場合、SqlComputeOnChange も指定できます)	プロパティは計算されません

計算プロパティを定義するには、以下の手順を実行します。

- プロパティ定義に SqlComputed キーワードを含めます。(つまり、SqlComputed キーワードを true に指定します。)
- プロパティ定義に SqlComputeCode キーワードを含めます。このキーワードの値には、SqlComputeCode に記載された規則に従い、プロパティの値を設定する ObjectScript コードの行を中括弧で囲んで指定します。例えば以下のようにします。

Class Definition

```
Class Sample.Population Extends %Persistent
{
Property FirstName As %String;

Property LastName As %String;

Property FullName As %String [ SqlComputeCode = {set {*}={FirstName}_" "_{LastName}}, SqlComputed
 ];
// ...

}
```

ObjectScript または Python などの他の言語を使用して *Property*Computation メソッドを指定し、プロパティの値を設定することもできます。例えば以下のようにします。

Class/ObjectScript

```
Class Sample.Population Extends %Persistent
{
Property FirstName As %String;

Property LastName As %String;

Property FullName As %String [ SqlComputed ];
// ...

ClassMethod FullNameComputation(cols As %Library.PropertyHelper) As %String
{
    return cols.getfield("FirstName")_" "_cols.getfield("LastName")
}
// ...
}
```

Class/Python

```
Class Sample.Population Extends %Persistent
{
Property FirstName As %String;

Property LastName As %String;

Property FullName As %String [ SqlComputed ];
// ...

ClassMethod FullNameComputation(cols As %Library.PropertyHelper) As %String [ Language = python ]
{
    return cols.getfield("FirstName") + " " + cols.getfield("LastName")
}
// ...
}
```

- プロパティが常に計算されるようにするには、プロパティ定義で Calculated キーワードを true に指定します。

  また、プロパティがトリガによって計算されるようにする場合は、Calculated および Transient キーワードをプロパティ定義に含めません。(つまり、必ず両方のキーワードを false にします。)

- プロパティがトリガによって計算される場合は、必要に応じて SqlComputeOnChange を指定します。

  このキーワードは 1 つ以上のプロパティを指定できます。これらのプロパティの値が変更されると、トリガされたプロパティは再計算されます。プロパティ名には、SqlFieldName で指定された名前 (後述) 以外を使用する必要があります。以下に例を示します (意図的に改行を入れてあります)。

```
Property messageId As %Integer [
SqlComputeCode = { set
{*}=$Select({Status}="":0,1:$List($List($Extract({Status},3,$Length({Status}))))) },
SqlComputed, SqlComputeOnChange = Status ];
```

  その他の例を示します (意図的に改行を入れてあります)。

```
Property Test2 As %String [ SqlComputeCode = { set {*}={Refprop1}_{Refprop2}}, SqlComputed,
SqlComputeOnChange = (Refprop1, Refprop2) ];
```

SqlComputeOnChange の値には、値 %%INSERT または %%UPDATE も含めることができます。詳細は、"SqlComputeOnChange" を参照してください。

- シャードまたはフェデレートされた環境で、計算フィールドに格納されたデータが、クエリの発行元のサーバからのみ返されるようにする場合は、追加で ComputeLocalOnly キーワードを指定できます。

このフィールドのインデックス付けを予定している場合は、非決定的コードではなく、決定的コードを使用します。非決定的コードは、古いインデックス・キー値を確実に削除することができないため、InterSystems IRIS は非決定的コードの結果に対するインデックスを保持できません。(決定的コードは、同じ引数を渡された場合はいつでも同じ値を返します。例えば、$h を返すコードは、$h が関数の制御外で変更されているため、非決定的です。)

"Calculated (プロパティ・キーワード)" も参照してください。後述する "計算プロパティの SQL プロジェクションの制御" も参照してください。

15.5 多次元プロパティの定義

多次元になるプロパティを定義できます。つまり、プロパティを多次元配列のように機能させるということです。これを行うには、プロパティ定義に MultiDimensional キーワードを含めます。

```
Property PropName as Classname [ MultiDimensional ] ;
```

このプロパティは、以下の点で他のプロパティとは異なります。

- InterSystems IRIS は、このプロパティにプロパティ・メソッドを提供しません ("プロパティ・メソッドの使用とオーバーライド" を参照)。
- オブジェクトを検証または保存するときに、無視されます。
- アプリケーションにそれを明示的に保存するコードが含まれていない場合、ディスクに保存されません。
- Java などのクライアントに公開することはできません。
- SQL テーブルに格納できず、SQL テーブルでも公開されません。

多次元プロパティはあまり使用されませんが、オブジェクトの状態に関する情報を一時的に格納する場合に有用な方法を提供します。

また、"多次元プロパティの値の指定" も参照してください。

15.6 列挙プロパティの定義

数多くのプロパティが、VALUELIST パラメータと DISPLAYLIST パラメータをサポートしています。これらを使用して、列挙プロパティを定義します。

プロパティに有効な値のリストを指定するには、VALUELIST パラメータを使用します。VALUELIST のフォームは、区切り文字で区切った論理値のリストで、区切り文字が最初の文字になります。以下はその例です。

Class Member

```
Property Color As %String(VALUELIST = ",red,green,blue");
```

この例で VALUELIST は、有効な値が red、green、blue であることを指定し、区切り文字としてコンマを使用します。同様に、

Class Member

```
Property Color As %String(VALUELIST = " red green blue");
```

上の文は同じリストを指定しますが、区切り文字としてスペースを使用しています。

プロパティはリストの値に限定され、データ型の妥当性検証コードは、その値がリストにあるかどうかだけを調べます。リストがなければ、値は特に限定されません。

DISPLAYLIST は、リストがある場合、プロパティの LogicalToDisplay() メソッドで返される対応する表示値を表す追加リストです。

表示値を取得する方法の例は、"プロパティ・メソッドの使用法" を参照してください。

15.7 リテラル・プロパティの値の指定

リテラル・プロパティに値を指定するには、以下に示すように SET コマンド、OREF、およびドット構文を使用します。

```
 SET oref.MyProp=value
```

oref は OREF、MyProp は対応するオブジェクトのプロパティです。value は、リテラル値に評価される ObjectScript 式です。以下に例を示します。

```
 SET patient.LastName="Muggles"
 SET patient.HomeAddress.City="Carver"
 SET mrn=##class(MyApp.MyClass).GetNewMRN()
 set patient.MRN=mrn
```

リテラル値は、プロパティ・タイプに対して有効な論理値にする必要があります。例えば、%Boolean に基づくプロパティには、1 または 0 を使用します。もう 1 つ例を挙げると、列挙プロパティの場合、値は、VALUELIST パラメータで指定された項目のいずれかにする必要があります。

15.7.1 多次元プロパティの値の指定

多次元プロパティについては、プロパティの添え字に応じて値を指定できます。以下に例を示します。

```
 set oref.MyStateProp("temp1")=value1
 set oref.MyStateProp("temp2")=value2
 set oref.MyStateProp("temp3")=value3
```

多次元プロパティは、オブジェクトで使用するための一時的な情報を保持する際に役立ちます。このようなプロパティはディスクに保存されません。

15.8 プロパティ・メソッドの使用法

それぞれのプロパティは、生成されたクラス・メソッドのセットをクラスに追加します。これらのメソッドには、propnameIsValid()、propnameLogicalToDisplay()、propnameDisplayToLogical()、propnameLogicalToOdbc()、propnameOdbcToLogical() などが含まれます。propname はプロパティの名前です。これらのメソッドの一部は、%Property クラスから継承され、その他のメソッドはプロパティが基づくデータ型クラスから継承されます。メソッドの詳細とリストは、"データ型クラスの定義" を参照してください。

InterSystems IRIS はこれらのメソッドを内部的に使用し、ユーザもこれらを直接呼び出すことができます。どの場合も、引数はプロパティ値です。例えば、Sample.Person に論理値 M と F (および表示値 Male と Female) を持つ Gender

という名前のプロパティがあるとします。次のように、指定されたレコードに対してこのプロパティの論理値と表示値を表示できます。

```
MYNAMESPACE>set p=##class(Sample.Person).%OpenId(1)

MYNAMESPACE>w p.Gender
M
MYNAMESPACE>w ##class(Sample.Person).GenderLogicalToDisplay(p.Gender)
Male
```

15.9 リテラル・プロパティの SQL プロジェクションの制御

永続クラスは、SQL テーブルとして投影されます。そのクラスでは、すべてのプロパティが SQL に投影されます。ただし、以下の例外を除きます。

- Transient プロパティ (ただし、"計算プロパティの SQL プロジェクションの制御" を参照すること)
- Calculated プロパティ (ただし、"計算プロパティの SQL プロジェクションの制御" を参照すること)
- プライベート・プロパティ
- 多次元プロパティ

このセクションでは、リテラル・プロパティについて詳しく説明します。

15.9.1 フィールド名の指定

既定では、プロパティ (SQL に投影される場合) は、そのプロパティと同じ名前の SQL フィールドとして投影されます。別のフィールド名を指定する場合は、プロパティ・キーワード SqlFieldName を使用します (プロパティ名が SQL 予約語の場合にも、このキーワードを使用できます)。例えば、"Select" という名前の %String プロパティがある場合、投影された名前を以下の構文で定義できます。

Class Member

```
Property Select As %String [ SqlFieldName = SelectField ];
```

注釈　SQL 予約語であるプロパティ名を使用する場合、SQL 文で使用するときに二重引用符を付けることで、対応するフィールド名をエスケープすることもできます。以下に例を示します。

```
SELECT ID, Name, "Select" FROM SQLUser.Person
```

詳細は、"区切り識別子" を参照してください。

15.9.2 列番号の指定

システムでは、各フィールドに一意の列番号が自動的に割り当てられます。列番号の割り当てを制御するには、以下の例で示すように、プロパティ・キーワード SqlColumnNumber を指定します。

Class Member

```
Property RGBValue As %String [ SqlColumnNumber = 3 ];
```

SqlColumnNumber に指定する値は 1 より大きい整数にする必要があります。SqlColumnNumber キーワードを引数なしに使用すると、InterSystems IRIS はテーブルで保持されていなくて固定位置を持たない列番号を割り当てます。

指定された SQL 列番号を持つプロパティがある場合、InterSystems IRIS は他のプロパティに列番号を割り当てます。割り当てられた列番号の開始値は、指定された SQL 列番号に続く最高値です。

SqlColumnNumber キーワードの値は継承されます。

15.9.3 データ型クラスとプロパティ・パラメータの影響

特定のプロパティで使用されるデータ型クラスは、SQL プロジェクションに影響を与えます。具体的には、データ型の SQL カテゴリ (SqlCategory キーワードで定義) により、プロパティを投影する方法を制御します。該当する場合は、プロパティ・パラメータも影響を与えます。

例えば、以下のようなプロパティ定義を考えてみます。

Class Member

```
Property Name As %String(MAXLEN = 30);}
```

このプロパティは、最大長が 30 文字の文字列フィールドとして投影されます。

15.9.4 計算プロパティの SQL プロジェクションの制御

InterSystems IRIS では、計算プロパティを定義できます。計算プロパティの値は ObjectScript で計算されます ("計算プロパティの定義" を参照してください)。以下のテーブルでは、どのバリエーションが SQL に投影されるかを示しています。

プロパティ定義	SQL へのプロジェクション
プロパティは計算されず、一時的ではありません (既定)	プロパティは SQL に投影されます
プロパティは計算されず、一時的です	プロパティは、SQL に投影されません
プロパティは常に計算されます	プロパティは SQL に投影されます
プロパティはトリガされる計算プロパティです	プロパティは SQL に投影されます

このテーブルでは、Calculated、Transient、SqlComputed、および SqlComputeCode キーワードの効果のみを考慮します。また、SQL プロジェクションを禁止するような別のキーワードがプロパティに使用されていないことを想定しています。例えば、プロパティはプライベートでないことを想定しています。

15.10 関連項目

- 一般的なデータ型クラス
- 一般的なプロパティ・パラメータ
- コレクションを使用した作業
- ストリームを使用した作業
- オブジェクト値プロパティの定義と使用
- リレーションシップの定義と使用
- プロパティ・メソッドの使用とオーバーライド

16

一般的なデータ型クラス

ここでは、InterSystems IRIS データ型クラスの概要について説明します。各データ型クラスは、いくつかのキーワードを指定します。これらのキーワードにより、InterSystems SQL でのデータ型の使用方法や、クライアント・システムでのデータ型の投影方法を制御します。

プロパティ定義で使用するデータ型を選択する際には、ニーズに応じた適切なクライアント・プロジェクションを持つクラスを選択する必要があります (該当する場合)。適切なクラスがない場合、"データ型クラスの定義" で説明するように、独自のデータ型クラスを作成できます。

16.1 概要

最も一般的に使用されるデータ型クラスは、以下のとおりです。

テーブル 16-1: 一般的なデータ型クラス

クラス名	保持する値	注
%BigInt	64 ビットの整数	このクラスは %Integer と類似しています (OdbcType および ClientDataType を除く)。
%Binary	バイナリ・データ	実際のバイナリ・データを Unicode (またはその他) に変換せずにクライアントへ送信、またはクライアントから受信します。
%Boolean	ブーリアン値	可能な論理値は 0 (false) および 1 (true) です。
%Char	固定長の文字フィールド	
%Counter	整数。固有のカウンタとしての使用を目的としています	タイプ・クラスが %Counter のプロパティには、新しいオブジェクトが保存されたとき、または SQL を使用して新しいレコードが挿入されたときに、値が割り当てられます (そのプロパティに値が指定されていない場合)。詳細は、"インターシステムズ・クラス・リファレンス" の "%Counter" を参照してください。
%Currency	通貨値	このクラスは、Sybase または SQL Server から InterSystems IRIS への移行のためにのみ定義されます。
%Date	日付	論理値は InterSystems IRIS $HOROLOG 形式になります。

クラス名	保持する値	注
%DateTime	日付と時間	このクラスは、主に T-SQL の移行と datetime datetime/smalldatetime の動作を %TimeStamp データ型にマップするために使用されます。この場合、DisplayToLogical() メソッドと OdbcToLogical() メソッドは、T-SQL アプリケーションでサポートされている不正確な datetime 値を処理するロジックを提供します。
%Decimal	固定小数点	論理値は小数点形式の数です。"インターシステムズ・アプリケーションでの数値の計算" を参照してください。
%Double	IEEE 浮動小数点数	論理値は IEEE 浮動小数点数です。"インターシステムズ・アプリケーションでの数値の計算" を参照してください。
%EnumString	文字列	%String の特殊サブクラスです。これにより、possible 値の列挙セットを (DISPLAYLIST および VALUELIST を使用して) 定義できます。%String とは異なり、このプロパティの表示値は、このタイプの列を ODBC 経由でクエリするときに使用されます。
%ExactString	文字列	EXACT の既定の照合を使用する %String のサブクラスです。
%Integer	整数	
%List	$List 形式のデータ	論理値は $List 形式のデータです。
%ListOfBinary	$List 形式のデータ (各リスト項目はバイナリ・データ)	論理値は $List 形式のデータです。
%Name	“姓と名” 形式の名前	%Name データ型は、%Storage.Persistent クラスと併用すると、特別なインデックス作成がサポートされます。詳細は、"インターシステムズ・クラス・リファレンス" の "%Name" を参照してください。
%Numeric	固定小数点	
%PosixTime	日付と時間の値	このデータ型の論理値は、64 ビット整数としてエンコードされた 1970 年 1 月 1 日 00:00:00 以降からの (または以前の) 秒数になります。%PosixTime は %TimeStamp よりもディスク領域とメモリの使用量が少なく、%TimeStamp よりもパフォーマンスが向上します。
%SmallInt	小さい整数値	このクラスは %Integer と同じです (OdbcType を除く)。
%Status	エラー・ステータス・コード	InterSystems IRIS クラス・ライブラリの多くのメソッドが、%Status タイプの値を返します。これらの値の使用の詳細は、"インターシステムズ・クラス・リファレンス" の "%Status" を参照してください。
%String	文字列	
%Time	時刻値	論理値は午前 0 時以降の経過秒数。
%TimeStamp	日付と時間の値	%TimeStamp データ型の論理値は、YYYY-MM-DD HH:MM:SS.nnnnnnnnn 形式で示されます。h が $H 形式の日付/時刻値の場合は、以下に示すように、$ZDATETIME を使用して %TimeStamp プロパティ: $ZDATETIME(h,3) の有効な論理値を取得できます。 %PosixTimeの説明も参照してください。
%TinyInt	非常に小さい整数値	このクラスは %Integer と同じです (OdbcType と、その最大値および最小値を除く)。

特殊な使用を目的とした追加のデータ型クラスが数多くあります。これらの型の大部分は、ここにリストしたクラスのサブクラスです。詳細は、インターシステムズ・クラス・リファレンスを参照してください。

16.2 SqlCategory でグループ化されたデータ型クラス

データ型クラスについては、SqlCategory クラス・キーワードで SQL カテゴリを指定します。InterSystems SQL は、そのタイプのプロパティの値を処理する際に、このカテゴリを使用します。SqlCategory で管理される演算には、比較演算があります (より大きい、より少ない、等しいなど)。他の演算でも使用することがあります。ここでリストされているデータ型の SqlCategory 値を以下のテーブルに示します。

テーブル 16-2: SqlCategory でグループ化されたデータ型クラス

値	InterSystems IRIS データ型
DATE	%Date
DOUBLE	%Double
INTEGER	%BigInt、%Boolean、%Counter、%Integer、%SmallInt、%TinyInt
NAME	%Name
NUMERIC	%Currency、%Decimal、%Numeric
POSIXTS	%PosixTime
STRING	%Binary、%Char、%EnumString、%ExactString、%List、%ListOfBinary、%Status、%String
TIME	%Time
TIMESTAMP	%DateTime、%TimeStamp

どのようにリテラル・プロパティが SQL タイプに投影されるかについての詳細は、"InterSystems SQL リファレンス" の “データ型” を参照してください。

16.3 OdbcType でグループ化されたデータ型クラス

データ型クラスについては、InterSystems SQL ODBC インタフェースで使用される値と論理データ値を InterSystems IRIS が相互に変換する方法を OdbcType クラス・キーワードで制御します。ここでリストされているデータ型の OdbcType 値を以下のテーブルに示します。

テーブル 16-3: OdbcType でグループ化されたデータ型クラス

値	InterSystems IRIS データ型
BIGINT	%BigInt
BIT	%Boolean
DATE	%Date
DOUBLE	%Double
INTEGER	%Counter、%Integer
NUMERIC	%Currency、%Decimal、%Numeric

値	InterSystems IRIS データ型
TIME	%Time
TIMESTAMP	%DateTime、%PosixTime、%TimeStamp
VARBINARY	%Binary
VARCHAR	%Char、%EnumString、%ExactString、%List、%ListOfBinary、%Name、%Status、%String

16.4 ClientDataType でグループ化されたデータ型クラス

データ型クラスについては、ClientDataType クラス・キーワードにより、InterSystems IRIS が (そのタイプの) プロパティを Java または Active X に投影する方法を制御します。ここでリストされているデータ型の ClientDataType 値を以下のテーブルに示します。

テーブル 16-4: ClientDataType でグループ化されたデータ型クラス

値	使用対象
BIGINT	%BigInt
BINARY	%Binary (または、データの Unicode 変換がないことを要求するすべてのプロパティ)
CURRENCY	%Currency
DATE	%Date
DOUBLE	%Double
DECIMAL	%Decimal
INTEGER	%Counter、%Integer、%SmallInt、%TinyInt
LIST	%List、%ListOfBinary
NUMERIC	%Numeric
TIME	%Time
TIMESTAMP	%DateTime、%PosixTime、%TimeStamp
VARCHAR	%Char、%EnumString、%ExactString、%Name、%String

16.5 関連項目

- リテラル・プロパティの定義と使用
- 一般的なプロパティ・パラメータ
- プロパティ・メソッドの使用とオーバーライド
- データ型クラスの定義

17

一般的なプロパティ・パラメータ

ここでは、一般的なプロパティ・パラメータについて説明します。

17.1 概要

プロパティによっては、その動作に作用するように、そのプロパティのパラメータを指定できます。例えば、パラメータでは、最大値と最小値、表示に使用する形式、照合、特定のシナリオに使用する区切り文字などを指定できます。クラス定義内でパラメータを指定できます。以下に例を示します。

```
Property MyProperty as MyType (MYPARAMETER="some value");
```

17.2 主要なプロパティ・パラメータ

一部のパラメータは、あらゆるタイプのすべてのプロパティに使用できます。これらのパラメータは、以下のとおりです。

- CALCSELECTIVITY ー テーブル・チューニング機能でプロパティの selectivity を計算するかどうかを制御します。通常は、このパラメータを既定 (1) のままにしておくことが最良です。詳細は、"SQL 最適化ガイド" の "テーブルのチューニング" を参照してください。
- CAPTION ー クライアント・アプリケーションで、このプロパティに使用するキャプションです。
- EXTERNALSQLNAME ーリンク・テーブルで使用します。このパラメータでは、このプロパティのリンク先の外部テーブルにあるフィールドの名前を指定します。リンク・テーブル・ウィザードは、クラス生成時にこのパラメータをプロパティごとに指定します。リモート・データベースの SQL フィールドの名前は、さまざまな理由 (リモート・データベースのフィールド名が InterSystems IRIS の予約語であるなど) で InterSystems IRIS サーバ上のプロパティ名と異なる必要がある場合があります。リンク・テーブルの詳細は、"リンク・テーブル・ウィザード : テーブルまたはビューへのリンク" を参照してください。

  プロパティ・パラメータ EXTERNALSQLNAME には、SQLFieldName コンパイラ・キーワードとは別の用途があり、それらの項目には別の値を保持できます。SQLFieldName では、InterSystems IRIS データベース内の投影された SQL フィールド名を指定します。また、EXTERNALSQLNAME はリモート・データベース内のフィールド名です。
- EXTERNALSQLTYPE ー リンク・テーブルで使用します。このパラメータでは、このプロパティのリンク先の外部テーブルにあるフィールドの SQL タイプを指定します。リンク・テーブル・ウィザードは、クラス生成時にこのパラメータをプロパティごとに指定します。"EXTERNALSQLNAME" を参照してください。
- JAVATYPE ー このプロパティの投影先になる Java データ型です。

ほとんどのプロパティ・パラメータは、データ型クラスで定義されるため、クラス固有のものになります。次のセクションを参照してください。

17.3 クラス固有のプロパティ・パラメータ

このページの大半の部分では、すべてのプロパティで利用可能なパラメータについて説明します。その他の使用可能なパラメータは、プロパティで使用されるクラスによって異なります。ここでリストしたデータ型クラスがサポートするパラメータを以下のテーブルにリストします。パラメータは、3 つの列にグループ分けされています。1) 多くのデータ型クラスに存在するパラメータ、または頻繁に見かけるパラメータ。2) XML および SOAP のコンテキストでのみ意味を持つパラメータ。および 3) 少数のデータ型クラスにのみ出現するパラメータ、またはほとんど見かけないパラメータ。

テーブル 17-1: システム・データ型クラスでサポートされるパラメータ

データ型クラス	一般的なパラメータ	XML と SOAP のパラメータ	あまり一般的でないパラメータ
%BigInt	DISPLAYLIST、FORMAT、MAXVAL、MINVAL、VALUELIST	XSDTYPE	
%Binary	MAXLEN、MINLEN	CANONICALXML、MTOM、XSDTYPE	
%Boolean		XSDTYPE	
%Char	COLLATION、CONTENT、DISPLAYLIST、ESCAPE、MAXLEN、MINLEN、PATTERN、TRUNCATE、VALUELIST	XMLLISTPARAMETER、XSDTYPE	
%Counter	DISPLAYLIST、FORMAT、MAXVAL、MINVAL、VALUELIST	XSDTYPE	
%Currency*	DISPLAYLIST、FORMAT、MAXVAL、MINVAL、SCALE、VALUELIST	XSDTYPE	
%Date	DISPLAYLIST、FORMAT、MAXVAL、MINVAL、VALUELIST	XSDTYPE	
%DateTime	DISPLAYLIST、MAXVAL、MINVAL、VALUELIST	XMLDEFAULTVALUE、XMLTIMEZONE、XSDTYPE	DATEFORMAT
%Decimal	DISPLAYLIST、FORMAT、MAXVAL、MINVAL、SCALE、VALUELIST	XSDTYPE	
%Double	DISPLAYLIST、FORMAT、MAXVAL、MINVAL、SCALE、VALUELIST	XSDTYPE	
%EnumString	COLLATION、CONTENT、DISPLAYLIST、ESCAPE、MAXLEN、MINLEN、PATTERN、TRUNCATE、VALUELIST	XSDTYPE	

データ型クラス	一般的なパラメータ	XML と SOAP のパラメータ	あまり一般的でないパラメータ
%ExactString	COLLATION、CONTENT、DISPLAYLIST、ESCAPE、MAXLEN、MINLEN、PATTERN、TRUNCATE、VALUELIST	XSDLISTPARAMETER、XSDTYPE	
%Integer	DISPLAYLIST、FORMAT、MAXVAL、MINVAL、VALUELIST	XSDTYPE	STRICT
%List	ODBCDELIMITER	XSDTYPE	
%ListOfBinary	ODBCDELIMITER	XSDTYPE	
%Name	COLLATION、MAXLEN	XSDTYPE	INDEXSUBSCRIPTS
%Numeric	DISPLAYLIST、FORMAT、MAXVAL、MINVAL、SCALE、VALUELIST	XSDTYPE	
%PosixTime	MAXVAL、MINVAL	XMLDEFAULTVALUE、XMLTIMEZONE、XSDTYPE	DATEFORMAT、INDEXNULLMARKER
%SmallInt	DISPLAYLIST、FORMAT、MAXVAL、MINVAL、VALUELIST	XSDTYPE	
%Status		XSDTYPE	
%String	COLLATION、CONTENT、DISPLAYLIST、ESCAPE、MAXLEN、MINLEN、PATTERN、TRUNCATE、VALUELIST	XMLLISTPARAMETER、XSDTYPE	
%Time	DISPLAYLIST、FORMAT、MAXVAL、MINVAL、VALUELIST	XMLTIMEZONE、XSDTYPE	PRECISION
%TimeStamp	DISPLAYLIST、MAXVAL、MINVAL、VALUELIST	XMLDEFAULTVALUE、XMLTIMEZONE、XSDTYPE	
%TinyInt	DISPLAYLIST、FORMAT、MAXVAL、MINVAL、VALUELIST	XSDTYPE	

* この特殊な目的のクラスは、InterSystems IRIS への移行にのみ使用します。

注釈　制約という用語は、プロパティ値に制限を適用するキーワードを指します。例えば、MAXVAL、MINVAL、DISPLAYLIST、VALUELIST、および PATTERN は、すべて制約です。

17.4 一般的なパラメータ

前述のテーブルの一般的なパラメータを以下に示します。

- COLLATION ー プロパティ値がインデックスに変換される方法を指定します。

  照合の許容値については、"SQL 入門" を参照してください。

- CONTENT ― 文字列が XML または HTML に変換される可能性があるコンテキストで使用される場合、その文字列のコンテンツを指定します。"STRING" (既定)、"ESCAPE"、または "MIXED" を指定します。

  詳細は、"オブジェクトの XML への投影" を参照してください。

- DISPLAYLIST ― 列挙 (複数選択) プロパティのために、VALUELIST パラメータと併用します。詳細は、"列挙プロパティの定義" を参照してください。

- ESCAPE ― 文字列が特定のコンテキストで使用される場合に、エスケープを行うタイプを指定します。"XML" (既定) または "HTML" のいずれかを使用します。

  既定では、小なり記号、大なり記号、およびアンド記号がそれぞれ <、>、および & に変換されます。"XML" の詳細は、"オブジェクトの XML への投影" を参照してください。

- FORMAT ― 表示値の形式を指定します。FORMAT の値については、$FNUMBER 関数の format 引数で指定した形式文字列を使用します。タイプ %Numeric または %Decimal のプロパティの場合は、オプション "AUTO" も使用できます。このオプションは末尾のゼロを抑制します。

- MAXLEN ― 文字列の最大文字数を指定します。既定値は 50 です。また、このパラメータが "" である場合、長さは無制限になります。他の多くのパラメータと同様に、このパラメータは、InterSystems IRIS によるデータの検証方法に影響します。これは、フィールドがデータベース・ドライバ上でどのように投影されるかにも影響することに注意してください。

- MAXVAL ― データ型の論理値の最大値を指定します。

- MINLEN ― 文字列の最小文字数を指定します。

- MINVAL ― データ型の論理値の最小値を指定します。

- ODBCDELIMITER ― ODBC を使用して投影されるとき、%List 値の構成に使用する区切り文字を指定します。

- PATTERN ― 文字列がマッチするパターンを指定します。PATTERN の値は、有効な InterSystems IRIS パターン・マッチング式である必要があります。パターン・マッチングの概要は、"パターン・マッチング (?)" を参照してください。

- SCALE ― 小数点以下の桁数を指定します。

- TRUNCATE ― 文字列を MAXLEN の長さまで切り捨てるかどうかを指定します。1 は True で、0 は False です。このパラメータは、Normalize() メソッドと IsValid() メソッドによって使用されますが、データベース・ドライバには使用されません。

- VALUELIST ― 列挙 (複数選択) プロパティに使用します。詳細は、"列挙プロパティの定義" を参照してください。

17.5 XML と SOAP のパラメータ

XML と SOAP のパラメータの列にあるパラメータの詳細は、"オブジェクトの XML への投影" を参照してください。"Web サービスおよび Web クライアントの作成" も参照してください。

17.6 あまり一般的でないパラメータ

前述のテーブルの、あまり一般的でないパラメータを以下に示します。

- STRICT (**%Integer** の場合) には、整数の値が必要になります。既定では、プロパティが **%Integer** 型の場合に、整数以外の数値を指定すると、InterSystems IRIS はその値を整数に変換します。プロパティに対して STRICT が 1 の場合には、InterSystems IRIS は値を変換せず、検証は失敗します。
- DATEFORMAT.
  - **%DateTime** の場合は、表示または ODBC 入力値に数値の日付形式が指定されているときに、このパラメータで日付部分の順序を指定します。有効なパラメータは mdy、dmy、ymd、ydm、myd、および dym です。既定の DATEFORMAT は mdy です。
  - **%PosixTime** の場合は、このパラメータで表示値の形式を指定します。許容される値については、$zdatetime 関数と $zdatetimeh 関数の fformat パラメータを参照してください。
- PRECISION (**%Time** の場合) では、保持する小数点以下の桁数を指定します。値が "" (既定) の場合、システムはソース値で入力された小数点以下桁数を保持します。値が 0 の場合、最も近い秒に丸められます。
- INDEXNULLMARKER (**%PosixTime** の場合のみ) では、**%PosixTime** 型プロパティのインデックスの添え字に使用する、既定の NULL 指標値を宣言します。既定値は -1E19 です。"NULL のインデックス付け" を参照してください。
- INDEXSUBSCRIPTS (**%Name** の場合) では、プロパティ値で区切り文字としてコンマを使用して、インデックス内のプロパティで使用される添え字の数を指定します。この数は、**%Storage.Persistent** クラスが使用します。2 の値は、プロパティ値の最初のコンマ部分は最初の添え字に、プロパティ値の 2 番目のコンマ部分は 2 番目の添え字に保存されることを示します。

17.7 関連項目

- リテラル・プロパティの定義と使用
- 一般的なデータ型クラス

18
コレクションを使用した作業

ここでは、InterSystems IRIS® データ・プラットフォームにおいてコレクション API (リストおよび配列) を使用する方法について説明します。コレクションは、オブジェクトのプロパティとして使用することも、スタンドアロンで使用することもできます。

18.1 コレクションの概要

コレクションは、同じタイプの要素が複数含まれるオブジェクト・クラスです。これらのタイプとしては、文字列や整数などのリテラル値、またはオブジェクトがあります。コレクション内の各要素は、キー (コレクション内でのその要素の位置) と値 (その位置に格納されるデータ) で構成されます。コレクションのメソッドとプロパティを使用して、以下のようなアクションを実行できます。

- キーに基づいて要素にアクセスし、値を変更する。
- 指定したキー位置に要素を挿入または削除する。
- コレクション内の要素の数を数える。

InterSystems IRIS® データ・プラットフォームには、リストと配列という 2 種類のコレクションの API が用意されています。

- リスト・コレクションでは、要素は順番に並べられます。リスト内のキーは、1 から N の整数で、N はリスト内の最後の要素です。以下の例では、リストの 2 番目の要素 (つまり、キーが 2 の要素) を文字列値 `red` に設定します。

  ObjectScript

  ```
  do myList.SetAt("red",2)
  ```

- 配列コレクションでは、要素の作成時に指定した任意のキー名に基づいて、要素にアクセスします。文字列または数値のキー名を指定できます。以下の例では、配列の `color` キーを文字列値 `red` に設定します。

  ObjectScript

  ```
  do myArray.SetAt("red","color")
  ```

  配列内の要素は、キーの順に並べられます。最初は数値キーで、小さい数値から大きい数値の順に並べられ、次に文字列キーで、アルファベット順に大文字の後に小文字という順に並べられます。例 : -2、-1、0、1、2、A、AA、AB、a、aa、ab。

18.2 コレクション・プロパティの定義

リストまたは配列のプロパティを定義するには、以下のプロパティ定義構文を使用します。

Class Member

```
Property MyProp as list of Type;
```

Class Member

```
Property MyProp as array of Type;
```

MyProp はプロパティ名です。Type は、データ型クラスまたはオブジェクト・クラスのどちらかです。

以下の例は、%String 値のリストのプロパティを定義する方法を示しています。

Class Member

```
Property Colors as list of %String;
```

以下のコードは、Doctor オブジェクトの配列のプロパティを定義する方法を示しています。

Class Member

```
Property Doctors as array of Doctor;
```

InterSystems IRIS では、コレクション・プロパティを %Collection パッケージ内にクラスのインスタンスとして格納します。これらのクラスには、コレクションの操作に使用できるメソッドとプロパティが含まれます。

以下の表は、コレクション・プロパティの定義方法と、その格納に使用する %Collection クラスの概要です。

コレクション・タイプ	%Collection クラス
文字列、整数、または他のデータ型のリスト	%Collection.ListOfDT
オブジェクトのリスト	%Collection.ListOfObj
文字列、整数、または他のデータ型の配列	%Collection.ArrayOfDT
オブジェクトの配列	%Collection.ArrayOfObj

注釈　プロパティのタイプとして %Collection クラスを直接使用しないでください。例えば、以下のようなプロパティ定義を作成してはいけません。

```
Property MyProp as %Collection.ArrayOfDT;
```

18.3 スタンドアロン・コレクションの定義

コレクション・プロパティを定義する別の方法として、メソッドの引数や戻り値として使用するためにスタンドアロン・コレクションを定義できます。スタンドアロン・コレクションを定義するために、InterSystems IRIS® データ・プラットフォームでは、%Library パッケージに、コレクション・プロパティの格納に使用する %Collection クラスと同様の機能を提供するクラスが用意されています。

スタンドアロン・コレクションを作成するには、適切なクラスの %New() メソッドを呼び出し、そのクラスのインスタンスを取得します。次に、そのインスタンスのメソッドを使用して、コレクションの操作を実行します。例えば、以下のコードは、%Library.ListOfDataTypes クラスを使用して文字列のリストを作成し、3 つの要素をリストに挿入してから、リスト内の要素の数を表示します。

ObjectScript

```
set mylist=##class(%ListOfDataTypes).%New()
do mylist.Insert("red")
do mylist.Insert("green")
do mylist.Insert("blue")
write mylist.Count()
```

以下の表は、スタンドアロン・コレクションの定義方法と、その作成に使用する %Library クラスの概要です。

コレクション・タイプ	%Library クラス	作成構文の例
文字列、整数、または他のデータ型のリスト	%ListOfDataTypes	`set myList = ##class(%ListOfDataTypes).%New()`
オブジェクトのリスト	%ListOfObjects	`set myList = ##class(%ListOfObjects).%New()`
文字列、整数、または他のデータ型の配列	%ArrayOfDataTypes	`set myArray = ##class(%ArrayOfDataTypes).%New()`
オブジェクトの配列	%ArrayOfObjects	`set myArray = ##class(%ArrayOfObjects).%New()`

18.4 リスト・コレクションの操作

以降のセクションの例は、リスト・プロパティを操作する方法を示していますが、同様の構文を使用してスタンドアロン・コレクションのリストを操作することができます。

18.4.1 リスト要素の挿入

要素をリストの末尾に挿入するには、Insert() メソッドを使用します。例えば、obj がオブジェクトへの参照であり、Colors が関連オブジェクトのリスト・プロパティであるとします。このプロパティの定義は以下のとおりです。

Class Member

```
Property Colors as list of %String;
```

以下のコードは、3 つの要素をリストに挿入します。リストがそれまで空だった場合、これらの要素は位置 1、2、および 3 にそれぞれ配置されます。

ObjectScript

```
do obj.Colors.Insert("Red") // key = 1
do obj.Colors.Insert("Green") // key = 2
do obj.Colors.Insert("Blue") // key = 3
```

要素をリスト内の特定の位置に挿入するには、InsertAt() メソッドを使用します。例えば、以下のコードは、obj の Colors プロパティの 2 つ目の位置に文字列 Yellow を挿入します。

ObjectScript

```
do obj.Colors.InsertAt("Yellow",2)
```

これにより、このリストの要素の順序は "Red" "Yellow" "Green" "Blue" になります。新しい要素は位置 2 に配置され、それまで位置 2 および 3 にあった要素 ("Green" および "Blue") が位置 3 および 4 に移動し、新しい要素の場所が作られます。

リスト要素の挿入は、オブジェクトに対しても同じように機能します。例えば、pat がオブジェクト参照であり、Diagnoses が関連オブジェクトのリスト・プロパティであるとします。このプロパティは、以下のように定義されています (PatientDiagnosis はクラスの名前です)。

Class Member

```
Property Diagnoses as list of PatientDiagnosis;
```

以下のコードは、オブジェクト参照 patdiag に格納される、PatientDiagnosis の新しいクラス・インスタンスを作成し、このオブジェクトを Diagnoses リストの末尾に挿入します。

ObjectScript

```
Set patdiag = ##class(PatientDiagnosis).%New()
Set patdiag.DiagnosisCode=code
Set patdiag.DiagnosedBy=diagdoc
Set status=pat.Diagnoses.Insert(patdiag)
```

18.4.2 リスト要素へのアクセス

リスト要素にアクセスするには、以下のコレクション・メソッドを使用します。

- GetAt(key) – key で指定された位置にある要素の値を返します。
- GetPrevious(key) – key の直前の位置にある要素の値を返します。
- GetNext(key) – key の直後の位置にある要素の値を返します。
- Find(value, key) – key の後から開始して、value に等しい次のリスト要素のキーを返します。

例えば、以下のコードは、リストを反復処理し、そのリストの要素を順番に表示します。コレクションの Count プロパティによって、反復処理する要素の数が決まります。

ObjectScript

```
set p = ##class(Sample.Person).%OpenId(1)
for i = 1:1:p.FavoriteColors.Count() {write !, p.FavoriteColors.GetAt(i)}
```

18.4.3 リスト要素の変更

指定したキーにある値を変更するには、以下の構文で示すように、SetAt() メソッドを使用できます。

ObjectScript

```
do oref.PropertyName.SetAt(value,key)
```

ここで、oref はオブジェクト参照であり、PropertyName はそのオブジェクトのリスト・プロパティの名前です。例えば、person.FavoriteColors は好きな色のリスト・プロパティであり、その要素として red、blue、および green があるとします。以下のコードは、リストの 2 つ目の色を yellow に変更します。

ObjectScript

```
do person.FavoriteColors.SetAt("yellow",2)
```

18.4.4 リスト要素の削除

リスト要素を削除するには、RemoveAt() メソッドを使用します。例えば、person.FavoriteColors は好きな色のリスト・プロパティであり、その要素として red、blue、および green があるとします。以下のコードは、位置 2 にある要素 (blue) を削除します。

ObjectScript

```
do person.FavoriteColors.RemoveAt(2)
```

これにより、このリストの要素の順序は red、green になります。それまで位置 3 にあった要素 (green) が位置 2 に移動し、削除された要素によって生じた空きが埋められます。

18.5 配列コレクションの操作

要素を配列に追加するには、SetAt() メソッドを使用します。例えば、myArray.SetAt(value,key) は、key の位置にある配列要素を、指定された value に設定します。以下のコードは、新しい色を RGB 値の配列に挿入します。

ObjectScript

```
do palette.Colors.SetAt("255,0,0","red")
```

palette はこの配列を格納するオブジェクトへの参照、Colors は配列プロパティの名前、"red" は値 "255,0,0" にアクセスするキーです。

重要　配列キーとして使用する値には、連続する 1 対の垂直バー (||) を含めないでください。この制限は、InterSystems SQL のメカニズムに起因しています。

SetAt() は、既存の要素の値を変更することもできます。例えば、以下のコードは、"red" キーに格納されている値を 16 進形式に変更します。

ObjectScript

```
do palette.Colors.SetAt("#FF0000","red")
```

配列内のコンテンツを反復処理するには、キーを GetNext() メソッドに参照渡しします。これにより、キーと値の両方をループで反復処理できます。例えば、以下のコードは、文字列配列のキーと値を順番に書き込みます。要素は、キーのアルファベット順に並べられます。

ObjectScript

```
set arr=##class(%ArrayOfDataTypes).%New()
do arr.SetAt("red","color")
do arr.SetAt("large","size")
do arr.SetAt("expensive","price")

set key=""
for  {set value=arr.GetNext(.key)  quit:key=""  write !,key," = ",value}

color = red
price = expensive
size = large
```

18.6 コレクション・データのコピー

一方のコレクションにある項目を他方のコレクションにコピーするには、コピー先のコレクションの設定を、コピー元のコレクションの設定と同じものにします。これにより、コピー元の内容が、コレクション自身の OREF ではなく、コピー先にコピーされます。以下は、このコマンドの例です。

ObjectScript

```
Set person2.Colors = person1.Colors
Set dealer7.Inventory = owner3.cars
```

ここで、person2、person1、dealer7、および owner3 は、すべてクラスのインスタンスで、Colors、Inventory、および cars は、すべてコレクションのプロパティです。このコードの 1 行目は、同じクラスにある 2 つのインスタンス間でデータをコピーし、2 行目は、互いに異なるクラスにあるインスタンスの間でデータをコピーします。

コピー先のコレクションがリストで、コピー元のコレクションが配列の場合、InterSystems IRIS では、配列のデータのみがコピーされます（キー値はコピーされません）。コピー先のコレクションが配列で、コピー元のコレクションがリストの場合、InterSystems IRIS では、コピー先の配列にキー値が生成されます。このキー値は、コピー元リストの項目の位置に基づいた整数になります。

注釈　　コレクションの間で OREF をコピーする方法はありません。データのコピーのみが可能です。

18.7 関連項目

- コレクション・プロパティのストレージと SQL プロジェクション
- リテラル・プロパティの定義と使用
- ストリームを使用した作業
- オブジェクト値プロパティの定義と使用
- リレーションシップの定義と使用
- プロパティ・メソッドの使用とオーバーライド

19

コレクション・プロパティのストレージとSQLプロジェクション

ここでは、リスト・プロパティと配列プロパティが既定で格納される方法と既定でSQLに投影される方法、およびこれらの詳細を変更する方法について説明します。

19.1 リスト・プロパティの既定のストレージとプロジェクション

既定では、リスト・プロパティはそれが属するオブジェクトと同じテーブルに格納され、複数の値を含む%List構造で構成される単一の列としてSQLに投影されます。この列内の項目を検索するには、FOR SOME %ELEMENT述語を使用します。例えば、以下のクエリは、FavoriteColors列（FavoriteColorsリスト・プロパティのプロジェクション）に要素'Red'が含まれる行を返します。

SQL

```
SELECT * FROM Sample.Person
WHERE FOR SOME %ELEMENT (FavoriteColors) (%VALUE = 'Red')
```

または、%INLIST述語を使用して、リスト内の特定の要素を検索したり、特定の要素が含まれないリストを検索したりすることもできます。例えば、以下のクエリは、FavoriteColors列にリスト要素'Red'が含まれない行を返します。

SQL

```
SELECT * FROM Sample.Person
WHERE NOT ('Red' %INLIST (FavoriteColors))
```

特定のインスタンスのリストが何も要素を含まない場合、これは空文字列として投影されます（SQL NULL文字列ではありません）。

19.2 配列プロパティの既定のストレージとプロジェクション

既定では、配列プロパティは子テーブルに格納され、子テーブルとして投影されます。この子テーブルは親テーブルと同じパッケージ内に存在します。この子テーブルの名前は、以下のようになります。

```
tablename_fieldname
```

以下はその説明です。

- tablename は親クラスの SqlTableName です (指定されている場合)。SqlTableName が指定されていない場合、親クラスの短い名前になります。
- fieldname は、配列プロパティの SqlFieldName です (指定されている場合)。SqlTableName が指定されていない場合、配列プロパティの名前になります。

例えば、Siblings という名前の配列プロパティが設定された Person クラスがあるとします。Siblings プロパティのプロジェクションは、Person_Siblings という名前の子テーブルです。

この子テーブルには、以下の列が含まれます。

1. 親クラスの対応するインスタンスの ID が含まれる列。この列は、配列が含まれる親クラスの外部キーとして機能し、そのクラスに対応する名前が付けられます。投影される Person_Child テーブルでは、この列の名前は Person です。
2. 各配列メンバの ID が含まれる element_key という名前の列。
3. 各配列メンバの値が含まれる列。この列には、配列プロパティに対応する名前が付けられます。投影される Person_Child テーブルでは、この列の名前は Siblings です。

以下の表は、サンプル・エントリと、Siblings 子テーブルの生成された列名を示しています。

テーブル 19-1: 配列プロパティのサンプル・プロジェクション

Person	element_key	Siblings
10	C	Claudia
10	T	Tom
12	B	Bobby
12	C	Cindy
12	G	Greg
12	M	Marsha
12	P	Peter

上の例で ID = 11 のインスタンスのように、親クラスのインスタンスが、空のコレクション (何も要素を含まないコレクション) を保持する場合、そのインスタンスの ID は子テーブルに表示されません。

親テーブルには、Siblings 列は含まれません。

配列メンバを含む列の数とコンテンツは、配列の種類によって異なります。

- データ型プロパティの配列は、1 列のデータとして投影されます。
- 参照プロパティの配列は、1 列のオブジェクト参照として投影されます。
- 埋め込みオブジェクトの配列は、子テーブル内の複数列として投影されます。これらの列の構造については、"埋め込みオブジェクト・プロパティ" を参照してください。

また、各インスタンスの ID と各配列メンバの識別子は、子テーブルの一意のインデックスを構成します。親インスタンスが関連する配列を持っていない場合、子テーブルには関連するエントリがありません。

注釈　　既定では、シリアル・オブジェクト・プロパティは、同じ方法で SQL に投影されます。

重要　コレクション・プロパティが配列として投影される際には、プロパティに追加する可能性がある任意のインデックスに対して固有の要件があります。"コレクションのインデックス作成"を参照してください。InterSystems IRIS 永続クラスのインデックスの概要は、"永続クラスのその他のオプション"を参照してください。

重要　配列コレクションによって投影された子テーブルの SQL トリガはサポートされていません。ただし、配列プロパティを更新した後、ObjectScript を使用して親オブジェクトを保存すると、該当するトリガが起動します。

19.3 コレクション・プロパティのストレージの制御

リスト・プロパティは子テーブルとして格納できます。また、配列プロパティは $LIST として格納できます。どちらの場合も、そのプロパティの STORAGEDEFAULT パラメータを指定します。

- リスト・プロパティの場合、既定では、STORAGEDEFAULT が "list" になります。STORAGEDEFAULT に "array" を指定すると、そのプロパティは子テーブルとして格納および投影されます。以下に例を示します。

  ```
  Property MyList as list of %String (STORAGEDEFAULT="array");
  ```

  結果のプロジェクションの詳細は、"配列プロパティの既定のストレージとプロジェクション"を参照してください。

- 配列プロパティの場合、既定では、STORAGEDEFAULT が "array" になります。STORAGEDEFAULT に "list" を指定すると、そのプロパティはリストとして格納および投影されます。以下に例を示します。

  ```
  Property MyArray as array of %String (STORAGEDEFAULT="list");
  ```

  結果のプロジェクションの詳細は、"リスト・プロパティの既定のストレージとプロジェクション"を参照してください。

重要　STORAGEDEFAULT プロパティ・パラメータは、コンパイラがクラスのストレージを生成する方法に影響します。クラス定義に指定のプロパティのストレージ定義が含まれていた場合、このプロパティ・パラメータはコンパイラによって無視されます。

19.4 SQL プロジェクションの制御

コレクション・プロパティが実際に格納される方法に関係なく、プロパティは親テーブル内の列として、子テーブルとして、またはその両方の方法で投影できます(リリース 2022.1 現在、これはリスト・プロパティと配列プロパティの両方に当てはまります)。これを制御するには、そのプロパティの SQLPROJECTION パラメータを指定します。このパラメータには、以下のいずれかの値を指定できます。

- "column" ― このプロパティを列として投影します。
- "table" ― このプロパティを子テーブルとして投影します。
- "table/column" ― このプロパティを列および子テーブルとして投影します。

例えば、以下のクラス定義について考えてみます。

Class Definition

```
Class Sample.Sample Extends %Persistent
{

Property Property1 As %String;

Property Property2 as array of %String(SQLPROJECTION = "table/column");

}
```

この場合、システムは、クラスで 2 つのテーブル (Sample.Sample および Sample.Sample_Property2) を生成します。

テーブル Sample.Sample_Property2 は、既定のシナリオどおり、配列プロパティ Property2 のデータを格納します。ただし既定のシナリオとは異なり、クエリは Sample.Sample テーブル内の Property2 フィールドを参照できます。以下に例を示します。

```
MYNAMESPACE>>SELECT Property2 FROM Sample.Sample where ID=7
13.     SELECT Property2 FROM Sample.Sample where ID=7

Property2
"1     value 12       value 23       value 3"
```

ただし、SELECT * クエリは、Property2 フィールドを返しません。

```
MYNAMESPACE>>SELECT * FROM Sample.Sample where ID=7
14.     SELECT * FROM Sample.Sample where ID=7

ID      Property1
7       abc
```

19.5 投影される子テーブルの名前の制御

コレクション・プロパティが子テーブルとして投影される場合は、そのテーブルの名前を制御できます。そのためには、そのプロパティの SQLTABLENAME パラメータを指定します。以下に例を示します。

```
Property MyArray as array of %String(SQLTABLENAME = "MyArrayTable");

Property MyList as list of %Integer(SQLTABLENAME = "MyListTable", STORAGEDEFAULT = "array");
```

SQLTABLENAME パラメータは、プロパティが子テーブルとして投影されていない場合には効果がありません。

19.6 関連項目

- 既定の SQL プロジェクションの概要
- コレクションを使用した作業

20
ストリームを使用した作業

ストリームは、大量のデータ（文字列長の制限値より長い）を保存する方法を提供します。ストリーム・プロパティは、あらゆるオブジェクト・クラスで定義できます。また、メソッドの引数や返り値として使用するなど、他の用途にスタンドアロンのストリーム・オブジェクトを定義することもできます。ここでは、ストリームおよびストリーム・プロパティについて説明します。

20.1 ストリーム・クラスの概要

InterSystems IRIS® Data Platform には、以下のストリーム・クラスが用意されています。

- %Stream.GlobalCharacter ー グローバル・ノードに文字データを格納するために使用します。
- %Stream.GlobalBinary ー グローバル・ノードにバイナリ・データを格納するために使用します。
- %Stream.FileCharacter ー 外部ファイルに文字データを格納するために使用します。
- %Stream.FileBinary ー 外部ファイルにバイナリ・データを格納するために使用します。
- %Stream.TmpCharacter ー 文字データを保持するためのストリームが必要だが、そのデータを保存する必要はない場合に使用します。
- %Stream.TmpBinary ー バイナリ・データを保持するためのストリームが必要だが、そのデータを保存する必要はない場合に使用します。

これらのクラスはすべて、共通のストリーム・インタフェースを定義する %Stream.Object を継承します。

%Library パッケージにもストリーム・クラスが含まれていますが、こちらは非推奨です。このクラス・ライブラリには追加のストリーム・クラスが含まれていますが、一般的な使用を目的としたものではありません。

ストリーム・クラスはオブジェクト・クラスです。したがって、ストリームはオブジェクトになります。

重要　これらのクラスのメソッドの多くは、ステータス値を返します。すべての場合において、詳細はクラス・リファレンスを参照してください。メソッドがステータス値を返す場合、コードで、戻り値をチェックして適切に処理する必要があります。同様に、%Stream.FileCharacter および %Stream.FileBinary に対して、Filename プロパティを設定した場合は、%objlasterror を調べてエラーをチェックする必要があります。

20.2 ストリーム・プロパティの定義

InterSystems IRIS は、バイナリ・ストリームと文字ストリームの両方をサポートします。バイナリ・ストリームは、%Binary タイプと同種のデータを含み、写真などの非常に大規模なバイナリ・オブジェクトを想定しています。同様に、文字ストリームは、%String タイプと同種のデータを含み、大量のテキストの保存を目的としています。文字ストリームは、文字列と同様に、クライアント・アプリケーション内で Unicode 変換を受ける可能性があります。

ストリーム・データは、ストリーム・プロパティの定義に応じて、外部ファイルまたは InterSystems IRIS グローバルに保存できます (一切保存しないようにすることもできます)。

- %Stream.FileCharacter クラスおよび %Stream.FileBinary クラスは、外部ファイルとして保存されたストリームに対して使用されます。
- %Stream.GlobalCharacter クラスおよび %Stream.GlobalBinary クラスは、グローバルとして保存されたストリームに対して使用されます。
- %Stream.TmpCharacter クラスおよび %Stream.TmpBinary クラスは、保存する必要のないストリームに使用します。

最初の 4 つのクラスは、オプションで LOCATION パラメータを使用して既定のストレージ場所を指定できます。

以下の例では、JournalEntry クラスに 4 つのストリーム・プロパティ (最初の 4 つのストリーム・クラスにつき 1 つずつ) が含まれ、そのうち 2 つについて既定のストレージ場所を指定しています。

Class Definition

```
Class testPkg.JournalEntry Extends %Persistent
{
Property DailyText As %Stream.FileCharacter;

Property DailyImage As %Stream.FileBinary(LOCATION = "C:/Images");

Property Text As %Stream.GlobalCharacter(LOCATION = "^MyText");

Property Picture As %Stream.GlobalBinary;
}
```

この例では、DailyImage のデータは C:/Images ディレクトリのファイル (ファイル名は自動的に生成されます) に保存されますが、Text プロパティのデータは ^MyText という名前のグローバルに保存されます。

20.3 ストリーム・インタフェースの使用法

すべてのストリームは、ストリームに含まれるデータの操作に使用されるメソッドおよびプロパティのセットを継承します。次のセクションでは多用するメソッドとプロパティの一覧を示し、それらの具体的な使用例をそれ以降のセクションで示します。項目は以下のとおりです。

- 一般的に使用されるストリーム・メソッドおよびストリーム・プロパティ
- ストリームをインスタンス化する方法
- ストリーム・データを読み取る方法と書き込む方法
- ストリーム間でコピーする方法
- ストリーム・データを挿入する方法
- ストリームのリテラル値を見つける方法
- ストリームを保存する方法

- オブジェクト・アプリケーションでストリームを使用する方法

重要　これらのクラスのメソッドの多くは、ステータス値を返します。すべての場合において、詳細はクラス・リファレンスを参照してください。メソッドがステータス値を返す場合、コードで、戻り値をチェックして適切に処理する必要があります。同様に、%Stream.FileCharacter および %Stream.FileBinary に対して、Filename プロパティを設定した場合は、%objlasterror を調べてエラーをチェックする必要があります。

20.3.1 一般的に使用されるストリーム・メソッドおよびストリーム・プロパティ

一般的に使用されるメソッドには以下のものがあります。

- Read() ー ストリームの現在の位置から、指定された数の文字を読み取ります。
- Write() ー 現在の位置から、ストリームにデータを追加します。この位置がストリームの末尾に設定されない場合は、既存のデータを上書きします。
- Rewind() ー ストリームの先頭に移動します。
- MoveTo() ー ストリーム内の特定の位置に移動します。
- MoveToEnd() ー ストリームの末尾に移動します。
- CopyFrom() ー コピー元のストリームの内容を、このストリームにコピーします。
- NewFileName() ー %Stream.FileCharacter または %Stream.FileBinary プロパティのファイル名を指定します。

一般的に使用されるプロパティには以下のものがあります。

- AtEnd ー Read がデータ・ソースの末尾に到達すると `true` に設定されます。
- Id ー %Location で指定したエクステントの範囲でストリームのインスタンスを特定する一意の識別子。
- Size ー ストリームの現在のサイズ (ストリームのタイプに応じて、バイト数または文字数で表されます)。

個々のストリーム・メソッドおよびプロパティの詳細は、このトピックの始めに示したクラスの "インターシステムズ・クラス・リファレンス" のエントリを参照してください。

20.3.2 ストリームのインスタンス化

ストリーム・クラスをオブジェクト・プロパティとして使用する場合、ストリームを含むオブジェクトをインスタンス化すると、ストリームが暗黙的にインスタンス化されます。

ストリーム・クラスをスタンドアロン・オブジェクトとして使用する場合は、%New() メソッドを使用してストリームをインスタンス化します。

20.3.3 ストリーム・データの読み取りと書き込み

ストリーム・インタフェースの中核は、メソッド Read()、Write()、Rewind() と、プロパティ AtEnd および Size です。

以下の例では、一度に 100 文字ずつ Person.Memo ストリームのデータを読み取り、それをコンソールに書き込みます。`len` の値は参照によって渡され、毎回 Read の前に 100 にリセットされます。Read メソッドは `len` で指定された数の文字を読み取ろうとし、次に `len` を実際に読み取った実際の文字数に設定します。

```
Do person.Memo.Rewind()
While (person.Memo.AtEnd = 0) {
  Set len = 100
  Write person.Memo.Read(.len)
}
```

同様に、ストリームにデータを書き込むことができます。

```
Do person.Memo.Write("This is some text. ")
Do person.Memo.Write("This is some more text.")
```

20.3.3.1 変換テーブルの指定

タイプ %Stream.FileCharacter のストリームをロケールのネイティブ文字セット以外の文字セットで読み取ったり書き込んだりする場合は、ストリームの TranslateTable プロパティを設定する必要があります。"変換テーブル" のリファレンス・ページを参照してください。

20.3.4 ストリーム間のデータのコピー

すべてのストリームは CopyFrom() メソッドを含み、1 つのストリームが別のストリームからのデータを取り入れることができます。これを使用して、例えばストリーム・プロパティにファイルのデータをコピーできます。その場合、一方では %Library.File クラスを使用します。これはオペレーティング・システム・コマンドのラッパであり、このクラスを使用することでファイルをストリームとして開くことができます。その場合のコードは以下のようになります。

```
// open a text file using a %Library.File stream
Set file = ##class(%File).%New("\data\textfile.txt")
Do file.Open("RU") // same flags as the OPEN command

// Open a Person object containing a Memo stream
// and copy the file into Memo
Set person = ##class(Person).%New()
Do person.Memo.CopyFrom(file)

Do person.%Save() // save the person object
Set person = ""   // close the person object
Set file = ""     // close the file
```

以下のように、Set コマンドを使用してデータをストリームにコピーすることもできます。

```
Set person2.Memo = person1.Memo
```

ここで、Person クラスの Memo プロパティはストリームの OREF を保持します。このコマンドは person1.Memo の内容を person2.Memo にコピーします。

20.3.5 ストリーム・データの挿入

一時ストリーム・クラス (そのデータは保存できない) とは別に、ストリームには一時的なストレージ場所と永続的なストレージ場所の両方があります。すべての挿入は一時的な場所に格納され、ストリームを保存する場合にのみ永続的に格納されます。ストリームへの挿入を開始した後でその挿入操作を中止すると、永続的な場所に保存されているデータは変更されません。

ストリームを作成する場合、挿入から開始し、MoveToEnd() を呼び出し、一時的なストリーム・データへの付加を継続します。ストリームを保存すると、データは永続的なストレージ場所に移動します。

以下に例を示します。

```
Set test = ##class(Test).%OpenId(5)
Do test.text.MoveToEnd()
Do test.text.Write("append text")
Do test.%Save()
```

この場合、test オブジェクトが保存されると、ストリームが永続ストレージに保存されます。

20.3.6 ストリームでのリテラル値の検索

ストリーム・インタフェースには、FindAt() メソッドが含まれています。このメソッドを使用すると、特定のリテラル値の場所を検索できます。このメソッドには、以下の引数があります。

```
method FindAt(position As %Integer, target, ByRef tmpstr, caseinsensitive As %Boolean = 0) as %Integer
```

以下はその説明です。

- position は、検索を開始する位置です。
- target は、検索するリテラル値です。
- tmpstr は参照によって渡され、FindAt() の次回呼び出しに使用できる情報を返します。これは、同じストリームを繰り返し (最後にターゲットが見つかった位置から開始して) 検索するときに使用します。このシナリオでは、position を -1 に指定し、呼び出しのたびに tmpstr を参照で渡します。その後の継続的な FindAt() の呼び出しは、前回の呼び出しの停止位置から開始されます。
- caseinsensitive は、大文字と小文字を区別する検索を実行するかどうかを指定します。既定では、このメソッドは大文字と小文字を区別しません。

このメソッドは、ストリームの先頭から開始したときの、この一致の位置を返します。一致する値が見つからない場合は、-1 を返します。

20.3.7 ストリームの保存

ストリーム・クラスをオブジェクト・プロパティとして使用する場合、ストリーム・データを含むオブジェクトを保存すると、ストリーム・データが保存されます。

ストリーム・クラスをスタンドアロン・オブジェクトとして使用する場合は、%Save() メソッドを使用してストリーム・データを保存します (一時ストリーム・クラス (%Stream.TmpCharacter および %Stream.TmpBinary) については、このメソッドはすぐに制御を返し、データを保存しません)。

20.3.8 オブジェクト・アプリケーションでストリームを使用する

ストリーム・プロパティは、そのストリーム・プロパティを所有するオブジェクトによって作成される一時的なオブジェクトを介して操作されます。ストリームは、リテラル値 (ストリームを大きな文字列と見なす) の役割を果たします。2 つのオブジェクト・インスタンスは同じストリームを参照できません。

以下のクラス定義では、Person クラスに、ストリーム・プロパティである Memo プロパティがあります。

```
Class testPkg.Person Extends %Persistent
{
Property Name As %String;

Property Memo As %Stream.GlobalCharacter;
}
```

以下の ObjectScript コードの断片は新しい Person オブジェクトを作成し、Memo ストリームを暗黙的にインスタンス化します。その後、いくつかのテキストをストリームに書き込みます。

```
    // create object and stream
    Set p = ##class(testPkg.Person).%New()
    Set p.Name = "Mo"
    Do p.Memo.Write("This is part one of a long memo. ")
    Do p.Memo.Write("This is part two of a long memo. ")
    Do p.Memo.Write("This is part three of a long memo. ")
    Do p.Memo.Write("This is part four of a long memo. ")
    Do p.%Save()
    Set id = p.%Id() // remember ID for later
    Set p = ""
```

以下のコードの断片はPersonオブジェクトを開いた後、ストリームの内容を書き込みます。オブジェクトを開くと、ストリーム・プロパティの現在の位置がストリームの先頭に設定されます。このコードでは、例示を目的として Rewind() メソッドを使用しています。

```
// read object and stream
Set p = ##class(testPkg.Person).%OpenId(id)
Do p.Memo.Rewind() // not required first time

// write contents of stream to console, 100 characters at a time
While (p.Memo.AtEnd = 0) {
    Set len = 100
    Write p.Memo.Read(.len)
}
Set p = ""
```

注釈　ストリーム・プロパティの内容を置き換える場合は、ストリームを巻き戻してから (ストリームの現在の位置が先頭になっていない場合)、Write() メソッドを使用して新しいデータをストリームに書き込みます。`set p.Memo = ##class(%Stream.GlobalCharacter).%New()` のように、%New() メソッドを使用して新しいストリーム・オブジェクトをインスタンス化し、それをストリーム・プロパティに割り当てないでください。この場合、古いストリーム・オブジェクトが孤立した状態でデータベース内に残ります。

20.4 gzip ファイルに使用するストリーム・クラス

%Stream パッケージでは、特殊なストリーム・クラス %Stream.FileBinaryGzip と %Stream.FileCharacterGzip も定義されています。このクラスは、gzip ファイルを対象とする読み込みと書き込みに使用できます。これらは前述の、同じインタフェースを使用します。以下の点に注意してください。

- これらのクラスについては、圧縮されていないサイズを Size プロパティが返します。Size プロパティにアクセスすると、InterSystems IRIS は、ファイルのサイズを計算するためにデータを読み込みます。これは、負荷の高い操作になることがあります。
- Size プロパティにアクセスすると、InterSystems IRIS はストリームを先頭の位置に巻き戻します。

20.5 SQL および ODBC へのストリーム・プロパティのプロジェクション

永続クラスは、SQL テーブルとして投影されます。このようなクラスについては、文字列ストリーム・プロパティとバイナリ・ストリーム・プロパティが、BLOB (binary large objects) として SQL (および ODBC クライアント) に投影されます。

ストリーム・プロパティは、ODBC タイプの LONG VARCHAR (または LONG VARBINARY) で投影されます。ODBC ドライバ、または ODBC サーバは、BLOB を読み取りまたは書き込むための特別なプロトコルを使用します。通常は、手動で BLOB アプリケーションを記述する必要があります。標準レポート・ツールは、それらをサポートしません。

後続のサブセクションでは、SQL でストリーム・プロパティを使用する方法について説明します。ここでは、以下のトピックについて説明します。

- 埋め込み SQL を使用してストリームを読み取る方法
- 埋め込み SQL を使用してストリームを書き込む方法

ストリーム・フィールドは、SQL では以下のような制限があります。

- WHERE 節の中では、いくつかの例外を除き、ストリーム値を使用できません。

- ストリームを含む複数の行を UPDATE (更新)、または INSERT (挿入) できません。一行ごとに操作を行う必要があります。
- 追加できる唯一のインデックスは、SQL Search ビットマップ・インデックスです。"SQL Search のためのソースのインデックス作成" を参照してください。

SQL でのストリームの使用に関する追加の情報について、"InterSystems SQL リファレンス" の "ストリーム・データ型" も参照してください。

20.5.1 埋め込み SQL によるストリームの読み取り

埋め込み SQL を使用して、以下のようにストリームを読み取ることができます。

1. 埋め込み SQL を使用して、ストリームの OID (オブジェクト ID) を選択します。

```
&sql(SELECT Memo INTO :memo FROM Person WHERE Person.ID = 12345)
```

   これはストリームの OID を取得して、memo ホスト変数に代入します。

2. その後、ストリームを開き、通常どおりに処理します。

20.5.2 埋め込み SQL によるストリームの書き出し

埋め込み SQL を使用してストリームを書き込むには、複数のオプションがあります。挿入する値については、ストリームのオブジェクト参照 (OREF)、ストリームの OREF の文字列バージョン、または文字列リテラルを使用できます。

以下の例で、これらすべての方法を示します。これらの例では、ストリーム値が予期される、Prop1 という名前の列を持つ Test.ClassWStream という名前のテーブルがあるものとします。

以下の例では、オブジェクト参照を使用します。

Class Member

```
///use an OREF
ClassMethod Insert1()
{
    set oref=##class(%Stream.GlobalCharacter).%New()
    do oref.Write("Technique 1")

    //do the insert; this time use an actual OREF
    &sql(INSERT INTO Test.ClassWStreams (Prop1) VALUES (:oref))
}
```

次の例では、オブジェクト参照の文字列バージョンを使用します。

Class Member

```
///use a string version of an OREF
ClassMethod Insert2()
{
    set oref=##class(%Stream.GlobalCharacter).%New()
    do oref.Write("Technique 2")

    //next line converts OREF to a string OREF
    set string=oref_""

    //do the insert
    &sql(INSERT INTO Test.ClassWStreams (Prop1) VALUES (:string))
}
```

最後の例では、文字列リテラルを Prop1 ストリームに挿入します。

Class Member

```
///insert a string literal into the stream column
ClassMethod Insert3()
{
    set literal="Technique 3"

    //do the insert; use a string
    &sql(INSERT INTO Test.ClassWStreams (Prop1) VALUES (:literal))
}
```

注釈　文字列リテラルの最初の文字は数字にすることはできません。数字の場合、SQL は、これを OREF として解釈し、OREF として保存しようとします。このストリームは OREF ではないため、これは SQL -415 エラーになります。

20.6 ストリームの圧縮

InterSystems IRIS では、グローバルに格納したストリーム・データを圧縮して、データベースでの占有容量とジャーナル・ファイルのサイズを削減できます。

圧縮は、クラス・パラメータ COMPRESS によってストリーム・クラスで制御されます。

%Stream.TmpCharacter クラスと %Stream.TmpBinary クラスでは、COMPRESS を 0 に設定するとストリーム・データが圧縮されません。

%Stream.GlobalCharacter クラスと %Stream.GlobalBinary クラスでは、COMPRESS を 1 に設定すると、データが圧縮に適していない以下の状況を除き、新しいストリーム・データが自動的に圧縮されます。

- ストリームが 1,024 文字未満の単一のチャンクに格納できる。
- ストリームが既に圧縮されている。つまり、データの最初のチャンクが、JPEG、MP3、ZIP などの一般的な圧縮ファイル形式である。
- ストリームの最初の 4KB を 20% 以上圧縮できない。

20.7 関連項目

- リテラル・プロパティの定義と使用
- オブジェクト値プロパティの定義と使用
- リレーションシップの定義と使用
- コレクションを使用した作業

21
オブジェクト値プロパティの定義と使用

ここでは、シリアル・オブジェクト・プロパティを含む、オブジェクト値プロパティを定義して使用する方法について説明します。

リレーションシップは、異なる永続クラスを関連付ける別の方法になります。"リレーションシップ" を参照してください。

21.1 オブジェクト値プロパティの定義

一般に、オブジェクト値プロパティとは、以下のように定義されたプロパティを指します。

```
Property PropName as Classname;
```

Classname は、コレクションまたはストリーム以外のオブジェクト・クラスの名前です。(コレクション・プロパティとストリーム・プロパティは特殊なケースです。これについては別途説明しています。)通常、Classname は、登録オブジェクト・クラス、永続クラス、またはシリアル・クラスのいずれかになります (次のセクションを参照)。

このようなプロパティを定義するには、プロパティが参照するクラスを定義してからプロパティを追加します。

21.1.1 バリエーション : CLASSNAME パラメータ

プロパティが永続クラスを基にしていて、そのクラスでサブクラスの代替のプロジェクションを使用している場合 ("永続クラスの定義" を参照) は、追加の手順が必要になります。この場合、そのプロパティの CLASSNAME プロパティ・パラメータに 1 を指定する必要があります。この手順は、このプロパティを InterSystems IRIS® データ・プラットフォームが保存する方法に影響し、そのプロパティがポイントしているオブジェクトを InterSystems IRIS が取得できるようにします。

例えば、MyApp.Payment では NoExtent を指定し、MyApp.CreditCard は MyApp.Payment のサブクラスだとします。MyApp.CurrencyOrder にはタイプ MyApp.CreditCard のプロパティが含まれているとします。このプロパティでは CLASSNAME を 1 に指定する必要があります。

Class Definition

```
Class MyApp.CurrencyOrder [ NoExtent ]
{
Property Payment as MyApp.CreditCard (CLASSNAME=1);

//other class members
}
```

SQL 矢印構文は、このシナリオでは機能しません (その代わりに、適切な JOIN を使用できます)。

重要　タイプがシリアル・クラスのプロパティに、CLASSNAME =1 を設定してはいけません。この使用法はサポートされていません。

21.2 シリアル・オブジェクトの概要

シリアル・クラスは %SerialObject を拡張します。このようなクラスの目的は、別のオブジェクト・クラスでプロパティとしての役割を果たすことです。シリアル・オブジェクトの値は、親オブジェクト内にシリアル化されます。シリアル・オブジェクトは、埋め込み (または、埋め込み可能) オブジェクトとも呼ばれます。InterSystems IRIS は、シリアル・オブジェクト・プロパティを、それ以外のオブジェクト・プロパティと異なる方法で処理します。以下に、2 つの相違点を示します。

- シリアル・オブジェクトのプロパティに値を割り当てる前に、シリアル・オブジェクトを作成する %New() を呼び出す必要はありません。
- シリアル・オブジェクト・プロパティが永続クラスに含まれている場合、シリアル・オブジェクトのプロパティは、永続クラスのエクステント内に保存されます。

シリアル・クラスを定義するには、%SerialObject を拡張するクラスを定義し、プロパティなどのクラス・メンバを必要に応じて追加します。以下に例を示します。

Class Definition

```
Class Sample.Address Extends %SerialObject
{

/// The street address.
Property Street As %String(MAXLEN = 80);

/// The city name.
Property City As %String(MAXLEN = 80);

/// The 2-letter state abbreviation.
Property State As %String(MAXLEN = 2);

/// The 5-digit U.S. Zone Improvement Plan (ZIP) code.
Property Zip As %String(MAXLEN = 5);

}
```

21.3 オブジェクトの可能な組み合わせ

以下のテーブルに、親クラスとそのクラスのオブジェクト値プロパティの可能な組み合わせを示します。

	プロパティが登録オブジェクト・クラス	プロパティが永続クラス	プロパティがシリアル・クラス
親クラスが登録オブジェクト・クラス	サポートされる	サポートされるが一般的ではない	サポートされる
親クラスが永続クラス	サポートされるが一般的ではない	サポートされる	サポートされる
親クラスがシリアル・クラス	サポートされない	サポートされない	サポートされる

21.3.1 オブジェクト値プロパティの用語

永続クラスには、オブジェクト値プロパティについての 2 つの用語があります。

- 参照プロパティ (別の永続オブジェクトに基づくプロパティ)
- 埋め込みオブジェクト・プロパティ (シリアル・オブジェクトに基づくプロパティ)

リレーションシップは、異なる永続クラスを関連付ける別の種類のプロパティです。"リレーションシップ" を参照してください。リレーションシップは双方向であり、このトピックで説明するプロパティとは異なります。

21.4 オブジェクト・プロパティの値の指定

オブジェクト値プロパティを設定するには、そのプロパティを適切なクラスのインスタンスの OREF と等しくなるように設定します。

ClassA には、ClassB を基にするプロパティ PropB が含まれているというシナリオについて考えてみます。ここでは、ClassB はオブジェクト・クラスです。

Class Definition

```
Class MyApp.ClassA
{
Property PropB as MyApp.ClassB;
//additional class members
}
```

さらに、ClassB には、独自のプロパティのセット (Prop1、Prop2、および Prop3) を持つ非シリアル・クラスが含まれています。

MyClassAInstance は、ClassA のインスタンスの OREF だとします。このインスタンスの PropB プロパティの値を設定するには、以下を実行します。

1. ClassB がシリアル・クラスでない場合は、最初に以下を実行します。
   a. ClassB のインスタンスの OREF を取得します。
   b. このインスタンスのプロパティを必要に応じて設定します。これは後で設定することもできます。
   c. その OREF と等しくなるように MyClassAInstance.PropB を設定します。

   ClassB がシリアル・クラスの場合は、この手順を省略できます。
2. 必要に応じて、カスケード・ドット構文を使用して、そのプロパティの各プロパティを設定します (つまり、MyClassAInstance.PropB の各プロパティを設定します)。

例 :

ObjectScript

```
set myclassBInstance=##class(MyApp.ClassB).%New()
set myClassBInstance.Prop1="abc"
set myClassBInstance.Prop2="def"
set myClassAInstance.PropB=myclassBInstance
set myClassAInstance.PropB.Prop3="ghi"
```

この例では、ClassB インスタンスの作成直後に、このインスタンスのプロパティを直接設定し、その後でカスケード・ドット構文によりさらに間接的に設定しています。

以下の手順では、同等の結果が得られます。

ObjectScript

```
 set myClassAInstance.PropB=##class(MyApp.ClassB).%New()
 set myClassAInstance.PropB.Prop1="abc"
 set myClassAInstance.PropB.Prop2="def"
 set myClassAInstance.PropB.Prop3="ghi"
```

これに対して、ClassB がシリアル・クラスの場合は、ClassB の %New() を呼び出すことなく、以下のように操作できます。

ObjectScript

```
 set myClassAInstance.PropB.Prop1="abc"
 set myClassAInstance.PropB.Prop2="def"
 set myClassAInstance.PropB.Prop3="ghi"
```

21.5 変更の保存

永続クラスを使用している場合は、格納オブジェクト（オブジェクト・プロパティを格納しているインスタンス）を保存します。オブジェクト・プロパティを直接保存する必要はありません。オブジェクト・プロパティは、格納オブジェクトが保存されるときに、自動的に保存されます。

以下の例は、これらの原則を示しています。以下の永続クラスを考えてみます。

Class Definition

```
Class MyApp.Customers Extends %Persistent
{

Property Name As %String;

Property HomeStreet As %String(MAXLEN = 80);

Property HomeCity As MyApp.Cities;

}
```

および、

Class Definition

```
Class MyApp.Cities Extends %Persistent
{

Property City As %String(MAXLEN = 80);

Property State As %String;

Property ZIP As %String;

}
```

この場合は MyApp.Customers のインスタンスを作成して、そのプロパティを以下のように設定できます。

ObjectScript

```
set customer=##class(MyApp.Customers).%New()
set customer.Name="O'Greavy,N."
set customer.HomeStreet="1234 Main Street"
set customer.HomeCity=##class(MyApp.Cities).%New()
set customer.HomeCity.City="Overton"
set customer.HomeCity.State="Any State"
set customer.HomeCity.ZIP="00000"
set status=customer.%Save()
if $$$ISERR(status) {
    do $system.Status.DisplayError(status)
}
```

これらの手順では、1 つの新しいレコードを MyApp.Customers に追加し、1 つの新しいレコードを MyApp.Cities に追加しています。

MyApp.Cities の %New() を呼び出す代わりに、以下のように既存のレコードを開くことができます。

ObjectScript

```
set customer=##class(MyApp.Customers).%New()
set customer.Name="Burton,J.K."
set customer.HomeStreet="17 Milk Street"
set customer.HomeCity=##class(MyApp.Cities).%OpenId(3)
set status=customer.%Save()
if $$$ISERR(status) {
    do $system.Status.DisplayError(status)
}
```

以下のバリエーションでは、新規顧客 (Customer) の追加プロセスの一環として、既存の市区町村 (City) を開いて、それを変更しています。

ObjectScript

```
set customer=##class(MyApp.Customers).%New()
set customer.Name="Emerson,S."
set customer.HomeStreet="295 School Lane"
set customer.HomeCity=##class(MyApp.Cities).%OpenId(2)
set customer.HomeCity.ZIP="11111"
set status=customer.%Save()
if $$$ISERR(status) {
    do $system.Status.DisplayError(status)
}
```

この変更は、当然、この出身地 (Home City) のどの顧客も参照できるようになります。

21.6 オブジェクト値プロパティの SQL プロジェクション

永続クラスは、SQL テーブルとして投影されます。このセクションでは、そのようなクラスの参照プロパティと埋め込みオブジェクト・プロパティが、どのように SQL に投影されるかについて説明します。

21.6.1 参照プロパティ

参照プロパティは、1 つのフィールドとして投影されます。このフィールドは、参照されるオブジェクトの OID の ID 部分を格納します。例えば、顧客オブジェクトには SalesRep オブジェクトを参照する Rep プロパティがあるとします。ある顧客の担当が ID 12 の営業員である場合、その顧客の Rep 列も 12 となります。この値は参照されているオブジェクトの ID 列の特定行の値と一致するので、この値を JOIN あるいはその他の処理を行うために使用できます。

InterSystems SQL では、JOIN を使用する代わりに特別な参照構文を使用して、このような参照を簡単に使用できます。以下に例を示します。

SQL

```
SELECT Company->Name FROM Sample.Employee ORDER BY Company->Name
```

21.6.2 埋め込みオブジェクト・プロパティ

埋め込みオブジェクト・プロパティは、親クラスのテーブルの複数列として投影されます。プロジェクションの 1 列に、すべての区切り文字と制御文字を含め、シリアル化した形式でオブジェクト全体が格納されます。他の各列は、オブジェクトの各プロパティに対応します。

オブジェクト・プロパティの列名は、そのオブジェクト・プロパティの名前と同じになります。他の列の名前は、オブジェクト・プロパティの名前 + アンダースコア + 埋め込みオブジェクト内のプロパティ名で構成されています。例えば、クラスが Address タイプの埋め込みオブジェクトを含む、Home プロパティを持っているとします。Home には、Street や Country などのプロパティが含まれます。その結果、この埋め込みオブジェクトのプロジェクションには Home_Street および Home_Country という列が存在します。(列名は、タイプ Address からではなく、プロパティ Home から得られます。)

例えば、サンプル・クラス Sample.Person には、タイプ Sample.Address の埋め込みオブジェクトである Home プロパティが含まれます。以下のように、SQL を経由して Home のコンポーネント・フィールドを使用できます。

SQL

```
SELECT Name, Home_City, Home_State FROM Sample.Person
WHERE Home_City %STARTSWITH 'B'
ORDER BY Home_City
```

埋め込みオブジェクトは、その他にも以下のような複雑な形式のデータを含むことができます。

- 参照プロパティのプロジェクションは、オブジェクト参照を含む読み取り専用フィールドを含みます。詳細は "参照プロパティ" を参照してください。
- 配列のプロジェクションは、テーブルの一部の編集できない単一の列になります。
- リストのプロジェクションは、投影したフィールドの 1 つであるリスト・フィールドになります。リスト・フィールドの詳細は、"リスト・プロパティの既定のプロジェクション" を参照してください。

21.7 関連項目

- リレーションシップ
- リテラル・プロパティの定義と使用
- コレクションを使用した作業
- ストリームを使用した作業
- プロパティ・メソッドの使用とオーバーライド

22
リレーションシップの定義と使用

ここでは、リレーションシップについて説明します。リレーションシップは永続クラスでのみ定義可能な、特殊なプロパティです。

22.1 リレーションシップの概要

リレーションシップとは、2 つの特定のタイプの永続オブジェクトの関連性です。2 つのオブジェクト間にリレーションシップを作成するには、それぞれのオブジェクトがリレーションシップの半分を定義するリレーションシップ・プロパティを持っている必要があります。InterSystems IRIS® データ・プラットフォームは、一対多と親対子の、2 種類のリレーションシップを直接サポートしています。

InterSystems IRIS のリレーションシップには、以下の特性があります。

- リレーションシップは二項です。つまり、リレーションシップは 2 つのみのクラス間で定義されるか、あるいは 1 つのクラスとそれ自体との間で定義されます。
- リレーションシップは、永続クラス間でのみ定義できます。
- リレーションシップは双方向です。リレーションシップは両側で定義する必要があります。
- リレーションシップは、参照整合性を自動的に提供します。これらは、SQL から外部キーとして見られます。"リレーションシップの SQL プロジェクション" を参照してください。
- リレーションシップは、メモリ内の振る舞いやディスク上の振る舞いを自動的に管理します。
- リレーションシップは、オブジェクト・コレクションでの優れたスケーリングや並行処理を提供します。

  一方、オブジェクト・コレクションには、オブジェクトの固有の順序があります。リレーションシップには同じことは当てはまりません。オブジェクト A、B、および C をこの順序でオブジェクトのリストに挿入すると、その順序が保持されます。オブジェクト A、B、および C をこの順序でリレーションシップ・プロパティに挿入しても、その順序は保持されません。

注釈　リレーションシップを追加するのではなく、永続クラス間に外部キーを定義することも可能です。外部キーにより、1 つのクラス内のオブジェクトが追加、更新、または削除されたときの動作内容を詳細に制御できるようになります。"トリガの使用法" を参照してください。

注釈　リレーションシップは、シャード・クラスではサポートされていません。

22.1.1 一対多リレーションシップ

クラス A とクラス B の間の一対多リレーションシップでは、クラス A の 1 つのインスタンスが、クラス B のゼロ個以上のインスタンスに関連付けられます。

例えば、company (会社) クラスは、employee (従業員) クラスと一対多のリレーションシップを定義することがあります。この場合、それぞれの company オブジェクトに関連付けられた employee オブジェクトが 0 個以上あることを意味します。

これらのクラスは、以下に示すように相互に独立しています。

- どちらかのクラスのインスタンスを作成したときに、そのインスタンスは、もう一方のクラスのインスタンスに関連付けても、関連付けなくてもかまいません。
- クラス B のインスタンスが、クラス A の特定のインスタンスに関連付けられている場合、この関連付けは、削除することも変更することも可能です。クラス B のインスタンスは、クラス A の別のインスタンスに関連付けることができます。クラス B のインスタンスは、クラス A のインスタンスとの関連付けが必要なわけではありません (その逆も同じです)。

1 つのクラス内での一対多リレーションシップもあり得ます。そのクラスの 1 つのインスタンスを、そのクラスのゼロ個以上の別のインスタンスに関連付けることができます。例えば、Employee クラスでは、ある従業員と、その従業員に直接報告を行う別の従業員とのリレーションシップを定義することもできます。

22.1.2 親子リレーションシップ

クラス A とクラス B との間の親子リレーションシップの場合は、クラス A の 1 つのインスタンスが、クラス B のゼロ個以上のインスタンスに関連付けられます。また、以下に示すように、子テーブルは親テーブルに依存します。

- クラス B のインスタンスを保存するとき、そのインスタンスは、クラス A のインスタンスと必ず関連付けられます。インスタンスを保存しようとしたときに、その関連付けが定義されていないと、保存アクションは失敗します。
- この関連付けは変更できません。そのため、クラス B のインスタンスをクラス A の別のインスタンスに関連付けることはできません。
- クラス A のインスタンスが削除されると、それに関連付けられたクラス B のすべてのインスタンスも削除されます。
- クラス B のインスタンスは削除できます。クラス A は、クラス B のインスタンスとの関連付けを必要としません。

例えば、invoice (送り状) クラスでは、line item (明細) クラスとの親子リレーションシップを定義できます。この場合、送り状はゼロ個以上の明細で構成されます。このような明細は、別の送り状に移動することはできません。それらが独自に意味を持つこともありません。

重要　　また、子テーブル (クラス B) 内の ID は、純粋な数字ではありません。結果として、このクラスのリレーションシップ・プロパティにビットマップ・インデックスを追加することは不可能になります。ただし、別の形式のインデックスは追加できます (インデックスの追加は、ここで後述するように役立ちます)。

22.1.2.1 親子リレーションシップとストレージ

クラスのコンパイル前に親子リレーションシップを定義すると、両方のクラスのデータは同一のグローバルに保存されます。以下に示すような構造で、子のデータは親のデータの下位に配置されます。

```
^Inv(1)
^Inv(1, "invoice", 1)
^Inv(1, "invoice", 2)
^Inv(1, "invoice", 3)
...
```

これにより、InterSystems IRIS は、このような関連オブジェクトをより高速に読み書きできるようになります。

22.1.3 一般的なリレーションシップ用語

このセクションでは、リレーションシップに関する説明を円滑に進めるために、一般に使用されている語句について例を挙げて説明します。

会社と従業員との間の一対多リレーションシップについて考えてみます。つまり、1 つの会社は複数の従業員を抱えているということです。このシナリオでは、会社は一側と呼ばれ、従業員は多側と呼ばれます。

同様に、会社と製品との間の親子リレーションシップについて考えてみます。つまり、会社は親であり、製品が子であるということです。このシナリオでは、会社は親側と呼ばれ、製品は子側と呼ばれます。

22.2 リレーションシップの定義

2 つのクラスのレコード間にリレーションシップを作成するには、補完的なリレーションシップ・プロパティを 1 組 (それぞれのクラスに 1 つずつ) 作成します。同一クラスのレコード間にリレーションシップを作成するには、そのクラスに補完的なリレーションシップ・プロパティを 1 組作成します。

以下のサブセクションでは、一般的な構文について説明してから、一対多リレーションシップと親子リレーションシップを定義する方法について説明します。

22.2.1 一般的な構文

リレーションシップ・プロパティの構文は、以下のとおりです。

```
Relationship Name As classname [ Cardinality = cardinality_type, Inverse = inverseProp ];
```

以下はその説明です。

- classname は、このリレーションシップが参照するクラスです。これは、永続クラスである必要があります。
- cardinality_type (必須) は、こちら側からリレーションシップがどのように見えるかを定義すると同時に、独立リレーションシップ (一対多) か、依存リレーションシップ (親子) かを定義します。cardinality_type は、`one` (一)、`many` (多)、`parent` (親)、または `children` (子) のいずれかです。
- inverseProp (必須) は、もう一方のクラスで定義される、補完的なリレーションシップ・プロパティの名前です。

補完的なリレーションシップ・プロパティでは、cardinality_type キーワードは、ここの cardinality_type キーワードを補完するものにする必要があります。値 `one` と `many` は、相互に補完するものです。同様に、値 `parent` と `children` は、相互に補完します。

リレーションシップはプロパティの一種であるため、その他のプロパティ・キーワード (Final、Required、SqlFieldName、Private などを含む) も使用できます。MultiDimensional など、一部のプロパティのキーワードは、適用されません。詳細は、"クラス定義リファレンス" を参照してください。

22.2.2 一対多リレーションシップの定義

このセクションでは、classA と classB との間に、一対多リレーションシップを定義する方法について説明します。ここでは、classA の 1 つのインスタンスが classB のゼロ個以上のインスタンスに関連付けられます。

注釈　1 つのクラスに含まれるレコード間で一対多リレーションシップを持つことができます。そのため、以下の説明の classA と classB は、同じクラスでもかまいません。

classA には、以下の形式のリレーションシップ・プロパティを含める必要があります。

Class Member

```
Relationship manyProp As classB [ Cardinality = many, Inverse = oneProp ];
```

oneProp は、補完的なリレーションシップ・プロパティの名前です。このプロパティは、classB で定義されます。

classB には、以下の形式のリレーションシップ・プロパティを含める必要があります。

Class Member

```
Relationship oneProp As classA [ Cardinality = one, Inverse = manyProp ];
```

manyProp は、補完的なリレーションシップ・プロパティの名前です。このプロパティは、classA で定義されます。

重要　一側 (classA) では、リレーションシップはクエリを使用してリレーションシップ・オブジェクトを生成します。ほとんどの場合、このクエリのパフォーマンスは、補完的なリレーションシップ・プロパティにインデックスを追加する (つまり、多側の class B にインデックスを追加する) ことで向上できます。

22.2.3 親子リレーションシップの定義

このセクションでは、classA と classB との間に、親子リレーションシップを定義する方法について説明します。ここでは、classA の 1 つのインスタンスが classB のゼロ個以上のインスタンスの親になります。これらは、同一のクラスにすることはできません。

classA には、以下の形式のリレーションシップ・プロパティを含める必要があります。

Class Member

```
Relationship childProp As classB [ Cardinality = children, Inverse = parentProp ];
```

parentProp は、補完的なリレーションシップ・プロパティの名前です。このプロパティは、classB で定義されます。

classB には、以下の形式のリレーションシップ・プロパティを含める必要があります。

Class Member

```
Relationship parentProp As classA [ Cardinality = parent, Inverse = childProp ];
```

childProp は、補完的なリレーションシップ・プロパティの名前です。このプロパティは、classA で定義されます。

重要　親側 (classA) では、リレーションシップはクエリを使用してリレーションシップ・オブジェクトを生成します。ほとんどの場合、このクエリのパフォーマンスは、補完的なリレーションシップ・プロパティにインデックスを追加する (つまり、子側の class B にインデックスを追加する) ことで向上できます。

22.2.3.1 親子リレーションシップとコンパイル

親子リレーションシップの場合、InterSystems IRIS では、前述のとおり、親オブジェクトと子オブジェクトのデータを 1 つのグローバルに保存するストレージ定義を生成できます。こうしたストレージ定義により、これらの関連オブジェクトにアクセスする速度を向上できます。

クラスのコンパイル後にリレーションシップを追加すると、InterSystems IRIS は、この最適化されたストレージ定義を生成しません。そのような場合は、すべてのテスト・データを削除し、2 つのクラスのストレージ定義を削除してからリコンパイルします。

22.3 例

このセクションでは、一対多リレーションシップと、親子リレーションシップの例を示します。

22.3.1 一対多リレーションシップの例

この例は、会社 (company) と従業員 (employee) との間の一対多リレーションシップを表しています。会社クラスは、次のようになります。

Class Definition

```
Class MyApp.Company Extends %Persistent
{

Property Name As %String;

Property Location As %String;

Relationship Employees As MyApp.Employee [ Cardinality = many, Inverse = Employer ];

}
```

また、従業員クラスは、次のようになります。

Class Definition

```
Class MyApp.Employee Extends (%Persistent, %Populate)
{

Property FirstName As %String;

Property LastName As %String;

Relationship Employer As MyApp.Company [ Cardinality = one, Inverse = Employees ];

Index EmployerIndex On Employer;

}
```

22.3.2 親子リレーションシップの例

この例は、送り状 (invoice) と明細 (line item) との間の親子リレーションシップを示しています。送り状クラスは以下のようになります。

Class Definition

```
Class MyApp.Invoice Extends %Persistent
{

Property Buyer As %String;

Property InvoiceDate As %TimeStamp;

Relationship LineItems As MyApp.LineItem [ Cardinality = children, Inverse = Invoice ];

}
```

明細クラスは以下のようになります。

Class Definition

```
Class MyApp.LineItem Extends %Persistent
{

Property ProductSKU As %String;

Property UnitPrice As %Numeric;

Relationship Invoice As MyApp.Invoice [ Cardinality = parent, Inverse = LineItems ];

Index InvoiceIndex On Invoice;

}
```

22.4 オブジェクトの接続

リレーションシップは双方向です。具体的には、一方のオブジェクトのリレーションシップ・プロパティの値を更新すると、その影響は、関連するオブジェクトの対応するリレーションシップ・プロパティの値に即座に現れます。したがって、一方のオブジェクトでリレーションシップ・プロパティの値を指定して、両方のオブジェクトに反映させることができます。

リレーションシップ・プロパティの性質は、2 つのクラスで異なります。そのため、リレーションシップの更新には、一般的なシナリオが 2 つ存在します。

- シナリオ 1：リレーションシップ・プロパティは単純な参照プロパティです。プロパティを、該当するオブジェクトと等しくなるように設定します。
- シナリオ 2：リレーションシップ・プロパティは %RelationshipObject のインスタンスです。このインスタンスには配列のようなインタフェースがあります。このインタフェースのメソッドを使用して、リレーションシップにオブジェクトを挿入します。リレーションシップ内のオブジェクトは、順序付けされないことに注意してください。リレーションシップでは、それにオブジェクトを挿入した際の順序は保持されません。

次のサブセクションで詳細を説明します。3 番目のサブセクションでは、シナリオ 1 のバリエーションについて説明します。このバリエーションは、リレーションシップに多数のオブジェクトがある場合に特に適しています。

ここでは、オブジェクトをリレーションシップに追加する方法について説明します。オブジェクトの変更プロセスはほぼ同じですが、親子リレーションシップの場合には、重要な（設計上の）例外があります。特定の親オブジェクトに関連付けられた（その後で保存された）子オブジェクトは、別の親に関連付けられなくなります。

22.4.1 シナリオ 1：多または子側の更新

多側または子側（ObjA）では、リレーションシップ・プロパティは、ObjB を指す単純な参照プロパティです。こちら側からオブジェクトに接続するには、以下の手順を実行します。

1. もう一方のクラスのインタンスに向けた OREF（ObjB）を取得します。（適宜、新しいオブジェクトを作成するか、既存のオブジェクトを開きます。）
2. ObjA のリレーションシップ・プロパティを ObjB と等しくなるように設定します。

例えば、前述した親子クラスの例について考えてみます。以下の手順では、リレーションシップを MyApp.LineItem 側から更新します。

ObjectScript

```
 //obtain an OREF to the invoice class
set invoice=##class(MyApp.Invoice).%New()
//...specify invoice date and so on

set item=##class(MyApp.LineItem).%New()
//...set some properties of this object such as the product name and sale price...

//connect the objects
set item.Invoice=invoice
```

item オブジェクトの %Save() メソッドを呼び出すと、システムにより、両方のオブジェクト (item および invoice) が保存されます。

この手法のバリエーションについては、最後のサブセクションも参照してください。

22.4.2 シナリオ 2 : 一または親側の更新

一側または親側では、リレーションシップ・プロパティは、%RelationshipObject のインスタンスです。こちら側では、オブジェクトに接続するために、以下の手順を実行できます。

1. もう一方のオブジェクトのインタンスに向けた OREF を取得します。(適宜、新しいオブジェクトを作成するか、既存のオブジェクトを開きます。)
2. こちら側のリレーションシップ・プロパティの Insert() メソッドを呼び出して、引数として OREF を渡します。

前述した親子クラスの例について考えてみます。それらのクラスの場合は、以下の手順により、リレーションシップを MyApp.Invoice 側から更新することになります。

ObjectScript

```
set invoice=##class(MyApp.Invoice).%OpenId(100034)
//set some properties such as the customer name and invoice date

set item=##class(MyApp.LineItem).%New()
//...set some properties of this object such as the product name and sale price...

//connect the objects
do invoice.LineItems.Insert(item)
```

invoice オブジェクトの %Save() メソッドを呼び出すと、システムにより、両方のオブジェクト (item および invoice) が保存されます。

重要　InterSystems IRIS は、オブジェクトがリレーションシップに追加される順序に関する情報を維持しません。そのため、以前に保存したオブジェクトを開いて、GetNext() などのメソッドを使用して、リレーションシップに繰り返し処理を実行すると、そのリレーションシップ内のオブジェクトの順序は、オブジェクトを作成したときの順序とは異なります。

22.4.3 オブジェクト接続の近道

比較的多数のオブジェクトをリレーションシップに追加する必要がある場合は、シナリオ 1 で示した手法のバリエーションを使用します。このバリエーションの内容は、以下のようになります。

1. クラス A の OREF (ObjA) を取得します。
2. クラス B のインスタンスの ID を取得します。
3. ObjA のリレーションシップ・プロパティのプロパティ・セッター・メソッドを使用します。このメソッドには、引数として ID を渡します。

   リレーションシップ・プロパティの名前が MyRel の場合、プロパティ・セッター・メソッドの名前は MyRelSetObjectId() になります。

プロパティ・セッター・メソッドの詳細は、"プロパティ・メソッドの使用とオーバーライド" を参照してください。

シナリオ 1 で説明した例のクラスを考えてみます。それらのクラスの場合は、以下の手順により、多数の送り状項目を 1 つの送り状に挿入することになります (当該のセクションで説明した手法よりもすばやく実行できます)。

ObjectScript

```
set invoice=##class(MyApp.Invoice).%New()
//set some properties such as the customer name and invoice date
do invoice.%Save()
set id=invoice.%Id()
kill invoice  //OREF is no longer needed

for index = 1:1:(1000)
 {
   set Item=##class(MyApp.LineItem).%New()
   //set properties of the invoice item

   //connect to the invoice
   do Item.InvoiceSetObjectId(id)
   do Item.%Save()
 }
```

22.5 リレーションシップの削除

一対多リレーションシップでは、2 つのオブジェクト間のリレーションシップを削除できます。これを行う方法の 1 つを以下に示します。

1. 子オブジェクト (または多側のオブジェクト) のインスタンスを開きます。
2. このオブジェクトの該当するプロパティを null に設定します。

例えば、Sample.Company と Sample.Employee との間に一対多リレーションシップが存在します。ID 101 の従業員 (employee) が、ID 5 の会社 (company) に勤めている場合の例を以下に示します。この会社には、4 人の従業員がいることに注目してください。

```
MYNAMESPACE>set e=##class(Sample.Employee).%OpenId(101)

MYNAMESPACE>w e.Company.%Id()
5
MYNAMESPACE>set c=##class(Sample.Company).%OpenId(5)

MYNAMESPACE>w c.Employees.Count()
4
```

次に、この従業員について、Company プロパティを null に設定します。この会社の従業員が 3 人になったことに注目してください。

```
MYNAMESPACE>set e.Company=""

MYNAMESPACE>w c.Employees.Count()
3
```

別のオブジェクトに変更を加えることで、リレーションシップを削除することもできます。ここでは、コレクション・プロパティの RemoveAt() メソッドを使用します。例えば、会社の ID は 17、最初の従業員の従業員 ID は 102 の場合の例を以下に示します。

```
MYNAMESPACE>set e=##class(Sample.Employee).%OpenId(102)

MYNAMESPACE>w e.Company.%Id()
17
MYNAMESPACE>set c=##class(Sample.Company).%OpenId(17)

MYNAMESPACE>w c.Employees.Count()
4
MYNAMESPACE>w c.Employees.GetAt(1).%Id()
102
```

この会社とこの従業員との間のリレーションシップを削除するには、RemoveAt() メソッドを使用し、引数として値 1 を渡して、最初のコレクション項目を削除します。これを実行すると、この会社の従業員は 3 人になることに注目してください。

```
MYNAMESPACE>do c.Employees.RemoveAt(1)

MYNAMESPACE>w c.Employees.Count()
3
```

親子リレーションシップの場合、2 つのオブジェクト間のリレーションシップを削除することはできません。ただし、子オブジェクトを削除することは可能です。

22.6 リレーションシップからのオブジェクトの削除

一対多リレーションシップの場合は、以下の規則により、オブジェクトを削除しようとしたときの動作が制御されます。

- 一側のオブジェクトは、そのオブジェクトを参照する多側のオブジェクトが存在していると、リレーションシップによって削除が妨げられます。例えば、会社を削除しようとしたときに、従業員テーブルにその会社を指しているレコードが存在していると、削除操作が失敗します。

  そのため、多側のレコードを先に削除しておく必要があります。

- 多側 (従業員テーブル) のオブジェクトの削除が、リレーションシップによって妨げられることはありません。

親子リレーションシップの場合は、規則が異なります。

- リレーションシップにより、親側の削除が子側に影響します。具体的には、親側のオブジェクトを削除すると、それに関連付けられた子側のオブジェクトが自動的に削除されます。

  例えば、送り状と明細との間の親子リレーションシップがある場合は、送り状を削除すると、その明細が削除されます。

- 子側 (明細テーブル) のオブジェクトの削除が、リレーションシップによって妨げられることはありません。

一対多および親子の両方のリレーションシップについて、OnDelete プロパティ・キーワードを使用して、一側または親側のオブジェクトを削除する既定の動作を変更することができます。

22.7 リレーションシップを使用した作業

リレーションシップは、プロパティです。一または親のカーディナリティを持つリレーションシップは、アトミック (非コレクション) の参照プロパティのように動作します。多または子のカーディナリティを持つリレーションシップは、配列のようなインタフェースを持つ %RelationshipObject クラスのインスタンスです。

例えば、以下の方法で、上記で定義された Company および Employee オブジェクトを使用できます。

ObjectScript

```
// create a new instance of Company
Set company = ##class(MyApp.Company).%New()
Set company.Name = "Chiaroscuro LLC"

// create a new instance of Employee
Set emp = ##class(MyApp.Employee).%New()
Set emp.LastName = "Weiss"
Set emp.FirstName = "Melanie"

// Now associate Employee with Company
Set emp.Employer = company

// Save the Company (this will save emp as well)
Do company.%Save()

// Close the newly created objects
Set company = ""
Set emp = ""
```

リレーションシップは、メモリ内で完全に双方向です。一方での操作を、他方ですぐに見ることができます。したがって、上記のコードは以下の company に対して操作するコードと同じです。

ObjectScript

```
Do company.Employees.Insert(emp)

Write emp.Employer.Name
// this will print out "Chiaroscuro LLC"
```

リレーションシップは、ディスクからロードすることができ、他の多様なプロパティを使用するように使用できます。リレーションシップの一側のオブジェクトを多側の任意のオブジェクトから参照すると、一側のオブジェクトは、参照 (オブジェクト値) プロパティと同じ方法で自動的にメモリへスウィズルされます。リレーションシップの多側の任意のオブジェクトを一側のオブジェクトから参照しても、多側のオブジェクトはすぐにはスウィズルされず、一時的な %RelationshipObject コレクション・オブジェクトが作成されます。メソッドがこのコレクションに呼び出されるとすぐに、リレーションシップ内のオブジェクト ID 値を持っているリストを構築します。これは、実際に関連するオブジェクトが、メモリにスウィズルされるコレクション内のオブジェクトの 1 つを参照するときに適用されます。

以下は、特定の Company に関連するすべての Employee オブジェクトを表示する例です。

ObjectScript

```
// open an instance of Company
Set company = ##class(Company).%OpenId(id)

// iterate over the employees; print their names
Set key = ""

Do {
   Set employee = company.Employees.GetNext(.key)
   If (employee '= "") {
       Write employee.Name,!
   }
} While (key '= "")
```

この例では、company を閉じることで、メモリから Company オブジェクトや、それに関連するすべての Employee オブジェクトを削除します。しかし、ループが終了するまでにリレーションシップに含まれるすべての Employee オブジェクトが (同時に) メモリ内にロードされることに注意してください。このオペレーションが使用する (多くの Employee オブジェクトが存在する) メモリの容量を削減するために、名前を表示した後に %UnSwizzleAt() メソッドを呼び出すことによって、Employee オブジェクトをアンスウィズルするようにループを変更します。

ObjectScript

```
Do {
   Set employee = company.Employees.GetNext(.key)
   If (employee '= "") {
       Write employee.Name,!
       // remove employee from memory
       Do company.Employees.%UnSwizzleAt(key)
   }
} While (key '= "")
```

重要　リレーションシップは、リスト・インタフェースをサポートしません。これは、関連するオブジェクト数をカウントしたりオブジェクトの繰り返し処理を行うのに、ポインタを 1 から 1 ずつインクリメントする方法は使用できないことを意味します。代わりに、配列コレクション・スタイルの繰り返しを使用する必要があります。リレーションシップ内のオブジェクトの繰り返し処理の詳細は、%Library.RelationshipObject の参照ページを参照してください。

22.8 リレーションシップの SQL プロジェクション

永続クラスは、SQL テーブルとして投影されます。このセクションでは、それらのクラスのリレーションシップがどのように SQL に投影されるかについて説明します。

注釈　関連するクラスの他のプロパティのプロジェクションは変更できますが、リレーションシップの SQL プロジェクション自体は変更できません。例えば、リレーションシップの CLASSNAME プロパティ・パラメータを指定することはサポートされていません。このパラメータについては、"オブジェクト値プロパティの定義" で説明されています。

22.8.1 一対多リレーションシップの SQL プロジェクション

このセクションでは、一対多リレーションシップの SQL プロジェクションについて説明します。一例として、前述した一対多クラスの例について考えてみます。この場合、クラスは以下のように投影されます。

- 一側 (会社クラス内) には、リレーションシップを表すフィールドがありません。会社テーブルには、その他のプロパティ用のフィールドはありますが、従業員を保持するフィールドはありません。
- 多側 (従業員クラス内) では、リレーションシップは単純な参照プロパティであり、そのプロパティは他の参照プロパティと同じ方法で SQL に投影されます。従業員テーブルには、Employer (雇用主) という名前のフィールドがあり、このフィールドが会社テーブルを指しています。

これらのテーブルに対して同時にクエリを実行するには、以下の例に示すように、従業員テーブルに対してクエリを実行し、矢印構文を使用します。

SQL

```
SELECT Employer->Name, LastName,FirstName FROM MyApp.Employee
```

または、以下の例に示すように、明示的な結合を実行します。

SQL

```
SELECT c.Name, e.LastName, e.FirstName FROM MyApp.Company c, MyApp.Employee e WHERE e.Employer = c.ID
```

また、このリレーションシップ・プロパティのペアは、暗黙的に外部キーを従業員テーブルに追加しています。この外部キーには、UPDATE および DELETE が含まれていて、どちらも NOACTION が指定されます。

22.8.2 親子リレーションシップの SQL プロジェクション

同様に、前述したように親子クラスの例について考えてみます。この例には、送り状と、その明細との間に親子リレーションシップがあります。この場合、クラスは以下のように投影されます。

- 親側 (送り状クラス内) には、リレーションシップを表すフィールドがありません。送り状テーブルには、その他のプロパティ用のフィールドはありますが、明細を保持するフィールドはありません。
- 子側 (明細クラス内) では、リレーションシップは単純な参照プロパティであり、そのプロパティは他の参照プロパティと同じ方法で SQL に投影されます。明細テーブルには、`Invoice` (送り状) という名前のフィールドがあり、このフィールドが送り状テーブルを指しています。
- さらに、子側の ID には、常に親レコードの ID が含まれます。その子にのみ基づいて IDKey を作成しようとしても同じことになります。また、子クラスの IDKey の定義に明示的に親のリレーションシップを設定しておくと、これがコンパイラで認識されるので、親のリレーションシップが重複して追加されることがありません。これにより、生成したグローバル参照内の添え字として親の参照が使用される順序を変更できます。

  その結果として、このプロパティにはビットマップ・インデックスを追加できなくなります。ただし、別の形式のインデックスは追加できます。

これらのテーブルに対して同時にクエリを実行するには、以下の例に示すように、送り状テーブルに対してクエリを実行し、矢印構文を使用します。

SQL

```
SELECT
Invoice->Buyer, Invoice->InvoiceDate, ID, ProductSKU, UnitPrice
FROM MyApp.LineItem
```

または、以下の例に示すように、明示的な結合を実行します。

SQL

```
SELECT
i.Buyer, i.InvoiceDate, l.ProductSKU,l.UnitPrice
FROM MyApp.Invoice i, MyApp.LineItem l
WHERE i.ID = l.Invoice
```

また、子側のクラスの場合、投影されたテーブルは、他方のテーブルの子テーブルとして採用されます。

22.9 多対多リレーションシップの作成

InterSystems IRIS では、多対多リレーションシップを直接はサポートしていませんが、このセクションでは、そのようなリレーションシップを間接的にモデル化する方法について説明します。

クラス A とクラス B との間に多対多リレーションシップを確立するには、以下の手順を実行します。

1. それぞれのリレーションシップを定義する中間クラスを作成します。
2. そのクラスとクラス A との間に、一対多リレーションシップを定義します。
3. そのクラスとクラス B との間に、一対多リレーションシップを定義します。

その後、クラス A のインスタンスとクラス B のインスタンスとの間のリレーションシップごとに、中間クラスにレコードを作成します。

例えば、クラス A で doctor (医者) を定義し、このクラスでプロパティ Name (名前) と Specialty (診療科目) を定義するとします。クラス B で patient (患者) を定義し、このクラスでプロパティ Name (名前) と Address (住所) を定義します。医者と患者との間の多対多リレーションシップをモデル化するため、以下に示すように中間クラスを定義できます。

Class Definition

```
/// Bridge class between MN.Doctor and MN.Patient
Class MN.DoctorPatient Extends %Persistent
{

Relationship Doctor As MN.Doctor [ Cardinality = one, Inverse = Bridge ];

Index DoctorIndex On Doctor;

Relationship Patient As MN.Patient [ Cardinality = one, Inverse = Bridge ];

Index PatientIndex On Patient;
}
```

次に、医者クラスは以下のようになります。

Class Definition

```
Class MN.Doctor Extends %Persistent
{

Property Name;

Property Specialty;

Relationship Bridge As MN.DoctorPatient [ Cardinality = many, Inverse = Doctor ];

}
```

さらに、患者クラスは以下のようになります。

Class Definition

```
Class MN.Patient Extends %Persistent
{

Property Name;

Property Address;

Relationship Bridge As MN.DoctorPatient [ Cardinality = many, Inverse = Patient ];

}
```

医者と患者の両方に対してクエリを実行する最も簡単な方法は、中間テーブルに対してクエリを実行することです。以下に例を示します。

```
SELECT top 20 Doctor->Name as Doctor, Doctor->Specialty, Patient->Name as Patient
FROM MN.DoctorPatient order by doctor

Doctor  Specialty       Patient
Davis,Joshua M. Dermatologist   Wilson,Josephine J.
Davis,Joshua M. Dermatologist   LaRocca,William O.
Davis,Joshua M. Dermatologist   Dunlap,Joe K.
Davis,Joshua M. Dermatologist   Rotterman,Edward T.
Davis,Joshua M. Dermatologist   Gibbs,Keith W.
Davis,Joshua M. Dermatologist   Black,Charlotte P.
Davis,Joshua M. Dermatologist   Dunlap,Joe K.
Davis,Joshua M. Dermatologist   Rotterman,Edward T.
Li,Umberto R.   Internist       Smith,Wolfgang J.
Li,Umberto R.   Internist       Ulman,Mo O.
Li,Umberto R.   Internist       Gibbs,Keith W.
Li,Umberto R.   Internist       Dunlap,Joe K.
Quixote,William Q.      Surgeon Black,Charlotte P.
Quixote,William Q.      Surgeon LaRocca,William O.
Quixote,William Q.      Surgeon Black,Charlotte P.
Quixote,William Q.      Surgeon Smith,Wolfgang J.
Quixote,William Q.      Surgeon LaRocca,William O.
```

```
Quixote,William Q.      Surgeon LaRocca,William O.
Quixote,William Q.      Surgeon Black,Charlotte P.
Salm,Jocelyn Q. Allergist       Tsatsulin,Mark S.
```

バリエーションとして、一対多リレーションシップのうちの 1 つの代わりに親子リレーションシップを使用します。これにより、データの物理クラスタリングを実現できます。ただし、これは、そのリレーションシップにはビットマップ・インデックスが使用できないことを意味します。

22.9.1 外部キーによるバリエーション

中間クラスとクラス A および B との間にリレーションシップを定義するのではなく、参照プロパティと外部キーを使用すると、中間クラス `MN.DoctorPatient` は前述のバージョンの代わりに次のようになります。

Class Definition

```
Class MN.DoctorPatient Extends %Persistent
{

Property Doctor As MN.Doctor;

ForeignKey DoctorFK(Doctor) References MN.Doctor();

Property Patient As MN.Patient;

ForeignKey PatientFK(Patient) References MN.Patient();

}
```

外部キーの詳細は、"外部キーの使用法" および "外部キー定義" を参照してください。

単純な外部キー・モデルを使用する 1 つの利点は、不注意による大量のオブジェクトのスウィズリングが発生しなくなることです。1 つの欠点は、自動的なスウィズリングが使用できないことです。

22.10 関連項目

- 永続オブジェクトの概要
- 永続オブジェクトを使用した作業
- プロパティの構文とキーワード

23

永続クラスのその他のオプション

ここでは、永続クラスに使用できる、その他のオプションについて説明します。

23.1 シャード・クラスの定義

データ・ストレージの水平方向の拡張にシャーディングを使用している場合、クラス定義で Sharded クラス・キーワードを使用してシャード・クラスを定義できます。シャード・クラスは永続クラスの 1 つで、データはシャード・クラスタのデータ・ノード間で分散されますが、アプリケーションはローカルの場合と同じようにそのデータにアクセスできます。

シャード・クラスを作成するには、以下の例のようにキーワード `Sharded = 1` を設定します。

```
Class MyApp.Person Extends %Persistent [ Sharded = 1 ]
```

シャード・クラスが完全に実装されるまでは、オブジェクト側からではなく SQL からシャード・テーブルを作成することをお勧めします。

シャードの詳細は、"シャーディングによるデータ量に応じた InterSystems IRIS の水平方向の拡張" を参照してください。

23.2 読み取り専用クラスの定義

オブジェクトを開くことはできても、保存または削除できない永続クラスを定義できます。これを行うには、クラスの READONLY パラメータを 1 に設定します。

Class Member

```
Parameter READONLY = 1;
```

これは、既存のストレージ (既存のグローバルや外部データベースなど) にマップされたオブジェクトを持っている場合にのみ便利です。読み取り専用オブジェクトで %Save() メソッドを呼び出す場合、常にエラー・コードが返されます。

23.3 列指向ストレージの使用法

重要　列指向ストレージは、リリース 2022.2 の試験的機能として利用できます。つまり、この機能は実稼働環境ではサポートされていません。今後のリリースへアップグレードした場合は、列指向テーブルのデータを再読み込みする必要があります。

InterSystems IRIS でデータを保存する永続クラスを使用する場合、通常はデータが行単位で保存されます。このストレージ・レイアウトは、データの挿入、更新、削除が頻繁に発生するオンライン・トランザクション処理に適しています。一方、オンライン分析処理の場合は、データを列単位で保存する方が適しています。このような処理ではデータベース全体で特定列のデータを集約しますが、リアルタイムの挿入、更新、削除の頻度は高くありません。例えば、行ストレージを使用している場合、指定したプロパティのすべての値の平均値を計算するには、そのデータベースにあるすべての行を読み込む必要があります。列指向ストレージを使用すれば、そのプロパティの値を収めた列のみを読み込むだけで平均値を計算できます。

クラスで列指向のストレージを使用するには、パラメータ `STORAGEDEFAULT = "columnar"` を設定します。

```
Class Sample.BankTransaction Extends %Persistent [ DdlAllowed, Final ]
{
   Parameter STORAGEDEFAULT = "columnar";
   .
   .
   .
}
```

列指向を既定のストレージ・レイアウトとして使用するクラスでは、`Final` クラス・キーワードまたは `NoExtent` クラス・キーワードを指定する必要があります。また、そのクラスに直下のサブクラスがあれば、それを明示的に `Final` として定義する必要があります。

トランザクションベースの処理を使用していても、頻繁に分析クエリの実行対象となるプロパティがわずかな場合は、そのプロパティにのみ列指向ストレージを使用できます。

クラスで単一のプロパティに列指向ストレージを使用するには、そのプロパティにパラメータ `STORAGEDEFAULT = "columnar"` を設定します。

```
Class Sample.BankTransaction Extends %Persistent [ DdlAllowed ]
{
   // Line below is optional
   Parameter STORAGEDEFAULT = "row";

   Property Amount As %Numeric(SCALE = 2, STORAGEDEFAULT = "columnar");

   Index BitmapExtent [ Extent, Type = bitmap ];
   .
   .
   .
}
```

あるクラスに行ストレージを使用して、頻繁にクエリの対象となるプロパティに列指向インデックスを作成することもできます。

クラスの特定のプロパティに列指向インデックスを作成するには、そのインデックスにキーワード type = columnar を使用します。

```
Class Sample.BankTransaction Extends %Persistent [ DdlAllowed ]
{
   // Line below is optional
   Parameter STORAGEDEFAULT = "row";

   Property Amount As %Numeric(SCALE = 2);

   Index AmountIndex On Amount [ type = columnar ];
   .
   .
   .
}
```

一般的なインデックスの詳細は、以下の "インデックスの追加" を参照してください。

列指向のストレージの詳細は、"高性能スキーマの定義" の "SQL テーブルのストレージ・レイアウトの選択" を参照してください。

23.4 インデックスの追加

インデックスは、永続クラスのインスタンス全体にわたる検索を最適化するメカニズムを提供します。クラスに関連付けられたデータのうち、要求されることが多いデータの一部をソートしてインデックスが定義されます。これらは、パフォーマンス・クリティカルな検索のオーバーヘッドを削減するのに非常に役立ちます。

インデックスは、インデックスが定義されるクラスの全範囲を自動的にカバーします。Person クラスにサブクラス Student がある場合、Person で定義したすべてのインデックスでは、Person オブジェクトと Student オブジェクトの両方が対象になります。Student クラスで定義したインデックスでは、Student オブジェクトのみが対象になります。

インデックスは、そのクラスに属する 1 つまたは複数のプロパティでソートできます。これは、返される結果の順序を制御するのに大変便利です。

またインデックスは、ソートされたプロパティに基づいて、クエリで頻繁に要求される他のデータを格納できます。インデックスの一部として他のデータを含めることによって、インデックスを使用するクエリの性能が大きく向上します。メイン・データ・ストレージにアクセスしなくても、クエリがインデックスを使用して結果セットを生成できます。(以下の Data キーワードを参照してください。)

注釈　インデックス定義をクラスに追加すると同時に、そのインデックスにデータを構築するには、ALTER TABLE 文、CREATE INDEX 文、または DROP INDEX 文を使用する以外に方法はありません。クラス定義にインデックスを追加する場合は、インデックスを手動で構築する必要があります。インデックスを手動で構築する方法の詳細は、"インデックスの構築" を参照してください。

インデックスの追加情報は、"インデックスの定義と構築" の、特に "インデックスを付けることができるプロパティ" のセクションを参照してください。"インデックス定義" も参照してください。

重要　インデックスは、プライマリ・スーパークラスからのみ継承されます。

23.5 外部キーの追加

対応する永続クラスの外部キーを定義して、テーブル間の参照整合性を強制できます。外部キー制約を持つテーブルを変更する際に、外部キー制約が確認されます。外部キーを追加する方法の 1 つは、クラス間にリレーションシップを追

加することです。"リレーションシップの定義と使用"を参照してください。クラスには、明示的な外部キーも追加できます。詳細は、"外部キーの使用法" および "外部キー定義" を参照してください。

23.6 トリガの追加

InterSystems SQL はトリガの使用をサポートしているので、永続クラスに関係するすべてのトリガは、クラスの SQL プロジェクションの一部として含まれています。

トリガは、特定のイベントが InterSystems SQL で発生するときに実行されるコード・セグメントです。InterSystems IRIS® データ・プラットフォームは INSERT コマンド、UPDATE コマンド、および DELETE コマンドの実行をベースにしたトリガをサポートします。トリガ定義により、指定されたコードは関連するコマンドが実行される直前、または直後に実行されます。各イベントは、実行順序が指定されていれば複数のトリガを持つことができます。

`Foreach = row/object` を使用してトリガが定義されている場合、トリガはオブジェクト・アクセスの特定の時点でも呼び出されます。"トリガとトランザクション" を参照してください。

トリガは、従来のストレージ・クラス **%Storage.SQL** によって使用されていた永続メソッドによっても呼び出されます。これは、永続の動作を内部的に実装するために SQL 文を使用しているためです。

トリガの詳細は、"トリガの使用法" および "トリガ定義" を参照してください。

23.7 ObjectScript からのフィールドの参照

クラス定義内には、SQL で使用される ObjectScript コードを含むことのできる場所が複数あります。例えば、SQL 計算フィールド・コードおよびトリガ・コードは、SQL 内から実行されます。これらの場合、現在のオブジェクトという概念はないので、ドット構文を使用して、特定のインスタンス内のデータにアクセスしたり、データを設定することはできません。代わりに、フィールド構文を使用して現在の行内のフィールドとして同じデータにアクセスできます。

現在の行の特定のフィールドを参照するには、`{fieldname}` 構文を使用します (fieldname は、フィールド名です)。

例えば、以下のコードは、従業員の給与が、50000 未満かどうかを確認します。

```
If {Salary} < 50000 {
   // actions here...
}
```

注釈　UPDATE トリガ・コードでは、`{fieldname}` は、更新済みのフィールド値を示します。DELETE トリガ・コードでは、`{fieldname}` は、ディスク上のフィールドの値を示します。

SQL 計算フィールドの現在のフィールドを参照するには、`{*}` 構文を使用します。

例えば、以下のコードは、Compensation フィールドのための計算コードに表示され、Salary フィールドおよび Commission フィールドの値に基づいてその値を計算します。

```
Set {*} = {Salary} + {Commission}
```

トリガ固有の構文については、"特殊なトリガ構文" を参照してください。

23.8 行レベル・セキュリティの追加

一般的なセキュリティに加えて、InterSystems IRIS では 1 行ごとのきめ細かい SQL セキュリティを提供します。これは、行レベル・セキュリティと呼ばれます。行レベル・セキュリティでは、各行に、表示操作を承認されたユーザまたはロールのリストが保持されます。詳細は、"ユーザ" と "ロール" を参照してください。

通常、SQL セキュリティは、テーブルまたはビューに対してユーザまたはロールに SELECT 特権を許可することによって制御します。ロールを使用することにより、セキュリティ・ロールの数がユーザの数を大幅に下回る場合にアクセス制御が単純化します。ほとんどの場合、各ユーザがどの行を選択できるかを制御するのにはビューレベル・セキュリティで十分です。ただし、目的の制御を実現するのに必要なビューの数が非常に多くなる場合は、きめ細かいアクセス制御を行うために別の方法が必要となります。

例えば、病院では各患者が患者固有のデータをオンラインで入手できるようにする場合があります。各患者に個別のビューを作成する方法は、実用的ではありません。代わりに、きめ細かいアクセス制御と InterSystems IRIS のロールベースの認証モデルを組み合わせることにより、行レベル・セキュリティでこのタイプのアプリケーションを効率的に、より安全に作成できます。

以下は、行レベル・セキュリティの使用に関する制約です。

- 行レベル・セキュリティは、永続クラスでのみ利用可能です。
- 行レベル・セキュリティは、InterSystems IRIS サーバ上でインスタンス化されたテーブルでのみ利用可能です。リンク・テーブル (つまり、外部サーバ上でインスタンス化されたテーブル) では利用できません。
- 行レベル・セキュリティは、SQL から行にアクセスする際にのみ適用されます。グローバルに直接アクセスする場合、またはオブジェクト・インタフェースを介してグローバルにアクセスする場合には適用されません。

23.8.1 行レベル・セキュリティの設定

テーブルの行レベル・セキュリティを有効にするには、そのテーブルの投影元のクラスの定義を編集します。

1. クラス定義コードでは、ROWLEVELSECURITY の値を 1 に設定します。以下は、その例です。

   ```
   ROWLEVELSECURITY = 1;
   ```

   このパラメータの定義は、行レベル・セキュリティが有効であり、このクラスは生成された %READERLIST プロパティを使用して、行へのアクセスが認証されたユーザおよびロールについての情報を格納することを示しています。

   または、パラメータを以下のように定義できます。

   ```
   ROWLEVELSECURITY = rlsprop;
   ```

   rlsprop は、同じクラス内のプロパティ名です。この代替手段では、行レベル・セキュリティが有効であり、クラスは指定されたプロパティを使用して、行へのアクセスが認証されたユーザおよびロールに関する情報を格納することを示しています。この場合、次のようにインデックスもクラスに追加します。

   ```
   Index %RLI On rlsprop;
   ```

2. %SecurityPolicy() クラス・メソッドを定義します。このクラス・メソッドは、ビューおよびテーブルの SELECT 特権を前提とし、行の選択を許可されているロールおよびユーザ名を指定します。

   %SecurityPolicy() メソッドの構造は以下のとおりです。

   ```
   ClassMethod %SecurityPolicy() As %String [ SqlProc ]
   {
       QUIT ""
   }
   ```

その特性は以下のとおりです。

- %SecurityPolicy という名前のクラス・メソッドです。
- 文字列 (タイプ **%String**) を返します。
- ゼロ個以上の引数を指定できます。このメソッドに引数がある場合は、各引数がクラス内のプロパティ名と一致する必要があり、すべて互いに異なる必要があります。
- SqlProc キーワードは、メソッドをストアド・プロシージャとして呼び出すことができることを示します。
- メソッドの QUIT 文は、行を表示できるユーザまたはロールを返します。複数のユーザまたはロールがある場合、QUIT はそれらの名前をコンマ区切りリストにして返す必要があります。例に示すように、Null 文字列を返す場合は、テーブルで SELECT 特権を持つすべてのユーザが行を表示できることを示します。

重要　**%All** ロールに割り当てられたユーザには、行レベル・セキュリティによって保護されているテーブルの行に対するアクセス権が自動的には付与されません。**%All** ユーザがこれらの行にアクセスできるようにするには、%SecurityPolicy() メソッドで明示的にそのアクセス権の付与を指定する必要があります。

3. クラスおよび依存クラス (存在する場合) をコンパイルします。

23.8.2 既存のデータを含むテーブルへの行レベル・セキュリティの追加

行レベル・セキュリティを既存のデータを含むテーブルに追加するには、まず、"行レベル・セキュリティの設定" で説明する手順を実行します。次に、以下の操作を行います。

1. テーブルのインデックスを再構築します。
2. 各行を表示できるユーザおよびロールを一覧表示するプロパティの値を更新します。

23.8.2.1 インデックスの再構築

注意　ユーザがテーブルのデータにアクセスしている間は、そのテーブルのインデックスを再構築しないでください。これを行うと、クエリ結果が不正確になる場合があります。

テーブルのインデックスを再構築する手順は以下のとおりです。

1. WITH CHECK OPTION 節を持つ定義済みのビューがテーブルにある場合は、DROP VIEW コマンドを使用してこれらのビューを削除します (各行にどのユーザがアクセス権を持つかという設定を更新した後、これらのビューを再作成できます)。
2. 管理ポータルのホーム・ページで、[SQL] ページ ([**システムエクスプローラ**]→[SQL]) に移動します。
3. テーブルを含むネームスペースを選択します。
4. [**テーブル**] でテーブルの名前を選択します。これにより、テーブルの [**カタログの詳細**] が表示されます。
5. [**アクション**] ドロップダウン・リストで、[**テーブルのインデックスを再構築**] をクリックします。

インデックスの再構築の詳細は、"インデックスの定義と作成" を参照してください。

23.8.2.2 各行を表示できるユーザおよびロールの更新

これを実行する手順は以下のとおりです。

1. 管理ポータルのホーム・ページで、[SQL] ページ ([**システムエクスプローラ**]→[SQL]) に移動します。
2. テーブルを含むネームスペースを選択します。

3. [クエリ実行] をクリックします。
4. 編集可能領域で、テーブルを更新する文を発行します。その形式は以下のようになります。

```
UPDATE MySchema.MyClass SET rlsprop =
               MySchema.SecurityPolicy(MySQLColumnName1, ...)
```

   各項目の内容は次のとおりです。

   - MySchema は、そのクラスを含むスキーマ (パッケージ) です。
   - MyClass はそのクラスの名前です。
   - rlsprop は、行を読み取ることができるユーザおよびロールのリストを含むフィールドです。これは、既定では %READERLIST であり、それ以外は ROWLEVELSECURITY パラメータの宣言で指定されるプロパティ名です。
   - SecurityPolicy は、%SecurityPolicy() メソッドの定義で SqlName 値により指定される値です。%SecurityPolicy() メソッドに明示的な SQL 名がなく、そのクラスが MySchema.MyClass である場合、既定の名前は myClass_sys_SecurityPolicy (MySchema.MyClass_sys_SecurityPolicy の完全修飾形式) です。
   - MySQLColumnName1, ... は、%SecurityPolicy() クラス・メソッドで定義される、引数に対応する一連の SQL 列名 (存在する場合) です。

5. [実行] をクリックします。
6. 必要に応じて、最初に削除したビューを再作成します。

23.8.3 パフォーマンスのヒントおよび情報

%READERLIST プロパティは計算フィールドであり、その値は %SecurityPolicy() メソッドによって決まります。INSERT または UPDATE が実行されるたびに、その行に対して %SecurityPolicy() が呼び出され、%READERLIST の値が自動的に入力されます。

%READERLIST プロパティのコレクション・インデックスが定義され、これにより、クエリ・オプティマイザで使用されて、行レベル・セキュリティが有効になったときのパフォーマンス上の影響を最小化できます。

通常、セキュリティ・ポリシーは複数のコンマ区切りのロールまたはユーザ名を返す可能性があるため、既定では、ROWLEVELSECURITY を 1 に設定すると、コレクション・インデックスは %READERLIST プロパティ (列) に対して定義されます。セキュリティ・ポリシーが複数のユーザまたはロール名を返すことがない場合は、ROWLEVELSECURITY パラメータをオーバーライドし、%RLI インデックスを通常の (非コレクション) ビットマップ・インデックスとして明示的に定義できます。通常は、これによりパフォーマンスが最適化されます。

23.8.4 セキュリティのヒントおよび情報

行レベル・セキュリティを使用する際は、以下のセキュリティ要素に注意してください。

- 行レベル・セキュリティは、テーブルレベル・セキュリティに加えて処理を行います。SELECT、INSERT、UPDATE、または DELETE 文を実行するには、ユーザは該当する行に対してテーブルレベル・アクセスと行レベル・アクセスの両方を許可されている必要があります。
- ユーザ特権は、実行時 (SQL コマンドの実行を試行する際) に動的に確認されます。
- 更新可能なビューを作成し、WITH CHECK OPTION を指定した場合、そのビューの INSERT 処理は、挿入される行が、ビューで指定された WHERE 句を満たすかどうかを確認します。さらに、行レベルのセキュリティを持つテーブルからビューを作成している場合、ビューで SELECT * FROM のコマンドを発行した際、ビューの WHERE 句または暗黙の行レベルのセキュリティ述語により行を表示できないと、INSERT は失敗します。

- 行へアクセスできる場合は、%READERLIST フィールド（または、行を表示できるユーザおよびロールのリストを保持する任意のフィールド）の値を変更できます。これは、行に対する自分自身のアクセス権を直接的または間接的に削除するアクションを実行できることを示します。
- 行の挿入を試みた際に、その挿入が仮に行レベル・セキュリティが定義されていない状態で UNIQUE 制約に違反するようなものであれば、行レベル・セキュリティが有効なことによりたとえ制約のエラーを引き起こす行が更新トランザクションからは参照できないとしても、行の挿入は制約に違反することになります。

23.9 関連項目

- 永続オブジェクトの概要
- 永続オブジェクトを使用した作業
- 永続クラスの定義
- オブジェクト同期化機能の使用法

24

メソッド・ジェネレータとトリガ・ジェネレータの定義

メソッド・ジェネレータは、独自の実行時コードを生成する特殊なメソッドです。同様に、トリガ・ジェネレータは、独自の実行時コードを生成するトリガです。

ここでは、主にメソッド・ジェネレータについて説明しますが、トリガ・ジェネレータの場合も詳細は同様のものになります。

24.1 概要

InterSystems IRIS® データ・プラットフォームには、メソッド・ジェネレータを定義できるという強力な機能があります。メソッド・ジェネレータは、メソッドの実行時コードを生成する小さいプログラムであり、クラス・コンパイラによって呼び出されます。同様に、トリガ・ジェネレータもクラス・コンパイラによって呼び出され、トリガの実行時コードを生成します。

メソッド・ジェネレータは、InterSystems IRIS クラス・ライブラリ内で広範囲に使用されます。例えば、%Persistent クラスのメソッドの多くは、メソッド・ジェネレータとして実装されます。これにより、効率が少し下がる汎用コードの代わりに、カスタマイズされたストレージ・コードを各永続クラスに提供できます。多くの InterSystems IRIS データ型クラス・メソッドも、メソッド・ジェネレータとして実装されます。またこれにより、これらのクラスが使用される内容により、カスタム実装を行えるようになります。

メソッド・ジェネレータとトリガ・ジェネレータは、ユーザ独自のアプリケーション内で使用できます。メソッド・ジェネレータの一般的な使用方法は、1 つまたは複数の ユーティリティ・スーパークラスを定義し、使用するサブクラスの専用メソッドを提供することです。これらのユーティリティ・クラス内のメソッド・ジェネレータは、それらを使用するクラスの定義 (プロパティ、メソッド、パラメータ値など) に基づいて特別なコードを作成します。この技術の例としては、InterSystems IRIS ライブラリで提供されている %Populate クラスと %XML.Adaptor クラスがあります。

24.2 基本

メソッド・ジェネレータは、InterSystems IRIS クラスのメソッドで、CodeMode キーワードに `objectgenerator` が設定されたものです。

Class Definition

```
Class MyApp.MyClass Extends %RegisteredObject
{
Method MyMethod() [ CodeMode = objectgenerator ]
    {
        Do %code.WriteLine(" Write """ _ %class.Name _ """")
        Do %code.WriteLine(" Quit")
        Quit $$$OK
    }
}
```

クラス MyApp.MyClass がコンパイルされるとき、最終的には以下の実装を持った MyMethod メソッドが生成されます。

ObjectScript

```
 Write "MyApp.MyClass"
 Quit
```

トリガ・ジェネレータを定義することもできます。これを行うには、トリガの定義内で CodeMode = objectgenerator を使用します。トリガ内で使用できる値は、メソッド・ジェネレータ内で使用できる値と多少異なります。

24.3 ジェネレータの機能

メソッド・ジェネレータはクラスをコンパイルするときに有効になります。メソッド・ジェネレータの操作は簡単です。クラス定義がコンパイルされるとき、クラス・コンパイラは以下のことを行います。

1. クラスの継承を解決します (継承されたすべてのメンバのリストを構築します)。
2. (各メソッドの CodeMode キーワードから) メソッド・ジェネレータとして指定されているすべてのメソッドのリストを作成します。
3. すべてのメソッド・ジェネレータからコードを集め、それを 1 つまたは複数の一時的なルーチンにコピーし、コンパイルします (これによってメソッド・ジェネレータ・コードを実行することができます)。
4. コンパイルされるクラスの定義を示す、一時的な一連のオブジェクトを作成します。これらのオブジェクトは、メソッド・ジェネレータ・コードで使用できます。
5. メソッド・ジェネレータのすべてのコードを実行します。

   コンパイラは、各メソッドに GenerateAfter キーワードが存在すればその値を参照して、それぞれのメソッドを呼び出す順序を調整します。このキーワードによって、メソッド間にコンパイラのタイミングの依存性がある場合の制御を提供します。
6. 各メソッド・ジェネレータの結果を (コード行と他のメソッド・キーワードに対するすべての変更)、コンパイルされたクラス構造にコピーします (クラスの実際のコードを生成するのに使用)。

   元のメソッド・シグニチャ (引数と返りタイプ) は、すべてのメソッド・キーワード値と同様に、生成されたメソッドに使用されます。%Integer の返りタイプを持たせるようにメソッド・ジェネレータを指定する場合、実際のメソッドは %Integer の返りタイプを持ちます。
7. メソッド・ジェネレータによって生成されたコードと、メソッド・ジェネレータではないメソッドで生成されたコードを結合させることによって、実行可能なクラスのコードを生成します。

詳細は、トリガ・ジェネレータについても同じです。

24.4 メソッド・ジェネレータで使用できる値

メソッド・ジェネレータの実装には、メソッド・ジェネレータ・コードが実行されるコンテキストを理解することが重要です。前のセクションで説明したように、クラス・コンパイラはメソッド・ジェネレータ・コードを、クラス継承が解決された後、クラスのコードを生成する前の時点で呼び出します。メソッド・ジェネレータ・コードが呼び出されるときに、クラス・コンパイラは以下の変数をメソッド・ジェネレータ・コードで使用できるようにします。

テーブル 24-1: メソッド・ジェネレータで使用できる変数

変数	説明
%code	%Stream.MethodGenerator クラスのインスタンス。これは、メソッドのコードを出力するストリームです。
%class	%Dictionary.ClassDefinition クラスのインスタンス。クラスの元の定義を含みます。
%method	%Dictionary.MethodDefinition クラスのインスタンス。メソッドの元の定義を含みます。
%compiledclass	%Dictionary.CompiledClass クラスのインスタンス。コンパイルされるクラスの、コンパイルされた定義を含みます。つまり、継承が解決された後のクラスに関する情報を含むということです (スーパークラスから継承されたものを含む、すべてのプロパティやメソッドのリストなど)。
%compiledmethod または %objcompiledmethod	コンパイルされるメソッドの %Dictionary クラスのインスタンス (例 : %Dictionary.CompiledMethod、%Dictionary.CompiledPropertyMethod、%Dictionary.CompiledIndexMethod)。生成されるメソッドの、コンパイルされた定義を含みます。
%parameter	パラメータ名によってインデックスされたすべてのクラス・パラメータの値を含む配列。例えば、%parameter("MYPARAM") は、現在のクラスに対する MYPARAM クラス・パラメータの値を含みます。この変数は、%class オブジェクト経由で使用できる、パラメータ定義のリストを使用する簡単な方法として提供されています。
%kind または %membertype	メンバ・メソッドの場合、このメソッドに関連するクラス・メンバの種類 (例 : プロパティ・メソッドの場合は a、インデックス・メソッドの場合は i)。
%mode	メソッドのタイプ (例 : `method`、`propertymethod`、`indexmethod`)。
%pqname または %member	メンバ・メソッドの場合、このメソッドに関連するクラス・メンバの名前。

24.5 トリガ・ジェネレータで使用できる値

メソッドと同様に、トリガもジェネレータとして定義できます。つまり、トリガの定義内で CodeMode = `objectgenerator` を使用できます。トリガ・ジェネレータで使用できる変数を以下に示します。

テーブル 24-2: トリガ・ジェネレータで使用できる追加の変数

変数	説明
%code、%class、%compiledclass、および %parameter	前のセクションを参照してください。
%trigger	%Dictionary.TriggerDefinition クラスのインスタンス。トリガの元の定義を含みます。
%compiledtrigger または %objcompiledmethod	%Dictionary.CompiledTrigger クラスのインスタンス。生成されるトリガのコンパイル済みの定義を含みます。
%kind または %membertype	トリガの場合、これは値 t です。
%mode	トリガの場合、これは値 trigger です。
%pqname または %member	このトリガの名前。

24.6 メソッド・ジェネレータの定義

メソッド・ジェネレータを定義するには、以下の手順を実行します。

1. メソッドを定義し、その CodeMode キーワードを objectgenerator に設定します。
2. メソッドのボディで、クラスがコンパイルされるときに、実際のメソッド・コードを生成するコードを記述します。このコードは、コードを出力するために %code オブジェクトを使用します。これは、使用できる他のオブジェクトをインプットとし、どのようなコードを生成するかを決定します。

以下は、クラスが属するすべてのプロパティの名前をリストにするメソッドを生成する、メソッド・ジェネレータの例です。

Class Member

```
ClassMethod ListProperties() [ CodeMode = objectgenerator ]
{
    For i = 1:1:%compiledclass.Properties.Count() {
        Set prop = %compiledclass.Properties.GetAt(i).Name
        Do %code.WriteLine(" Write """ _ prop _ """,!")
    }
    Do %code.WriteLine(" Quit")
    Quit $$$OK
}
```

このジェネレータは、以下に類似した実装を持つメソッドを生成します。

ObjectScript

```
 Write "Name",!
 Write "SSN",!
 Quit
```

メソッド・ジェネレータ・コードについて、以下のことに注意してください。

1. %code オブジェクトの WriteLine メソッドを使用して、メソッドの実際の実装を含むストリームにコード行を書き込みます。(Write メソッドを使用して、行末の文字なしでテキストを出力することもできます)。
2. 生成されたコードの各行の文頭には、スペース文字があります。ObjectScript が行の最初のスペースにはコマンドを許可しないので、これは必須です。使用しているメソッド・ジェネレータが Basic、または Java コードを作成している場合は必要ありません。

3. 作成されたコード行は文字列に表示されるので、引用符文字を二重にすることで(””)エスケープすることに注意する必要があります。
4. クラスのプロパティのリストを検索するには、%compiledclass オブジェクトを使用します。%class オブジェクトを使用することもできますが、その場合は継承されたプロパティはリストにせず、コンパイルされているクラス内で定義されたプロパティのみをリストにします。
5. $$$OK のステータス・コードを返します。これは、メソッド・ジェネレータが正常に実行したことを示します。この返り値は、メソッドの実際の実装には関係ありません。

24.6.1 メソッド・ジェネレータ内で CodeMode を指定する

既定で、メソッド・ジェネレータは code メソッドを作成します (つまり、生成されたメソッドに対する CodeMode キーワードが code に設定されるということ)。これは、%code オブジェクトの **CodeMode** プロパティを使用して変更できます。

例えば、以下のメソッド・ジェネレータは ObjectScript 式のメソッドを生成します。

Class Member

```
Method Double(%val As %Integer) As %Integer [ CodeMode = objectgenerator ]
{
    Set %code.CodeMode = "expression"
    Do %code.WriteLine("%val * 2")
}
```

24.7 ジェネレータおよび INT コード

メソッド・ジェネレータとトリガ・ジェネレータでは、クラスのコンパイル後に、統合開発環境 (IDE) に対応する INT コードを表示すると非常に便利になる場合があります。InterSystems Studio を使用している場合は、**[表示]** メニューから **[他のコードの表示]** を選択します。Visual Studio Code に InterSystems ObjectScript 拡張機能を使用している場合は、エディタでクラス・ファイルを右クリックして、**[他を表示]** を選択します。

ジェネレータがカーネルで実装されるほど十分に単純である場合、.INT コードは生成されないことに注意してください。

24.8 サブクラスへの影響

ここでは、ジェネレータ・メソッドを定義したクラスのサブクラスに含まれるジェネレータ・メソッドに特有のトピックについて説明します。

当然のことですが、サブクラスはスーパークラスのコンパイル後にコンパイルする必要があります。

24.8.1 サブクラス内のメソッド生成

ジェネレータ・メソッドを定義するクラスをサブクラス化するとき、InterSystems IRIS は、前述したものと同じコンパイル・ルールを使用します。ただし、生成されるコードの外見がスーパークラス生成コードと同じになる場合、InterSystems IRIS はサブクラスのメソッドをリコンパイルしません。このロジックでは、両方のクラスのインクルード・ファイルが同じかどうかについては考慮しません。インクルード・ファイルで定義されたマクロをメソッドが使用する場合、別のインクルード・ファイルをサブクラスで使用しても、InterSystems IRIS はサブクラスのメソッドをリコンパイルしません。ただし、どのクラスのジェネレータ・メソッドも強制的にリコンパイルすることは可能です。そのためには、そのメソッドの ForceGenerate メソッド・キーワードを指定します。このキーワードは、別のシナリオでも必要になることがあります。

24.8.2 スーパークラスのメソッドの呼び出し

サブクラスでは、ローカルに生成されたメソッドではなく、スーパークラスのために生成されたメソッドを使用することが必要になる場合があります。その場合は、$$$OK を返すだけのジェネレータ・メソッドをサブクラスで定義します。以下に例を示します。

Class Member

```
ClassMethod Demo1() [ CodeMode = objectgenerator ]
{
    quit $$$OK
}
```

24.8.3 生成されたメソッドの削除

生成されたメソッドをサブクラスから削除して、そのクラスのメソッドの呼び出しを不可能にすることができます。そのためには、スーパークラスのジェネレータ・メソッドを定義するときに、現在のクラスの名前を検証して目的のシナリオでのみコードを生成するロジックを組み込みます。以下に例を示します。

```
ClassMethod Demo3() [ CodeMode = objectgenerator ]
{
    if %class.Name="RemovingMethod.ClassA" {
        Do %code.WriteLine(" Write !,""Hello from class: " _ %class.Name _ """")
    }
    quit $$$OK
}
```

このメソッドをサブクラスから呼び出そうとすると、<METHOD DOES NOT EXIST> エラーを受け取ることになります。

このロジックは、前のセクションで説明したものとはわずかに違う点に注意してください。特定のクラスにジェネレータ・メソッドが存在していても、メソッドが NULL 実装である場合は、スーパークラスのメソッドが代用されます（存在する場合）。ただし、特定のクラスのジェネレータ・メソッドが、特定のサブクラスにはコードを生成しない場合、そのサブクラスにメソッドは存在しなくなり、そのメソッドは呼び出しできなくなります。

24.9 関連項目

- メソッドの定義と呼び出し
- トリガの追加

25
クラス・クエリの定義と使用

ここでは、クラス・クエリについて説明します。クラス・クエリは、クラス構造の一部である名前付きクエリとして機能し、ダイナミック SQL を通じてアクセスできます。

25.1 クラス・クエリの概要

クラス・クエリとは、ダイナミック SQL と共に使用することを目的として、クラスに含まれているツールです。これにより、指定した条件を満たすレコードを検索します。クラス・クエリを使用すると、アプリケーションに応じて事前に定義した検索を作成できます。例えば、名前でレコードを検索したり、特定の条件のセット (パリからマドリッドまでのすべての航空便など) を満たすレコードのリストを提供したりできます。

クラス・クエリを作成することで、特定のオブジェクトの内部 ID を使用してそのオブジェクトを検索せずに済むようになります。その代わりに、必要とするクラス・プロパティに基づいてレコードを検索するクエリを作成できます。これは、実行時にユーザによって指定することも可能です。

カスタム・クラス・クエリを定義すると、検索ロジックに ObjectScript を使用することも、検索ロジックを任意の複合式にすることもできます。

クラス・クエリには、以下の 2 種類があります。

- 基本クラス・クエリ。クラス **%SQLQuery** と SQL SELECT 文を使用します。
- カスタム・クラス・クエリ。クラス **%Query** と、カスタム・ロジック (クエリを実行、フェッチ、およびクローズするためのロジック) を使用します。これらについては別途説明します。

クラス・クエリはどのクラス内でも定義できます。クラス・クエリは永続クラス内に含める必要はありません。

重要　　別のクラス・クエリの結果に依存するクラス・クエリを定義しないでください。このような依存関係はサポートされません。

25.2 クラス・クエリの使用法

クラス・クエリを定義する方法について説明する前に、その使用方法を知っておくと便利です。サーバ側のコードでは、以下のようにクラス・クエリを使用できます。

1. %New() を使用して、**%SQL.Statement** のインスタンスを作成します。

2. そのインスタンスの %PrepareClassQuery() メソッドを呼び出します。引数として、以下をこの順番で使用します。

   a. 使用するクエリを定義しているクラスの完全修飾名。

   b. そのクラスに含まれるクエリの名前。

   このメソッドは %Status 値を返すので、その値を確認する必要があります。

3. %SQL.Statement インスタンスの %Execute() メソッドを呼び出します。これにより %SQL.StatementResult のインスタンスが返されます。

4. %SQL.StatementResult のメソッドを使用して、結果セットからデータを取得します。詳細は、"ダイナミック SQL の使用" を参照してください。

以下に、Sample.Person の ByName クエリを使用する簡単な例を示します。

ObjectScript

```
// classquerydemo
set statement = ##class(%SQL.Statement).%New()
set status = statement.%PrepareClassQuery("Sample.Person","ByName")
if $$$ISERR(status) {
   write "%Prepare failed:" do $SYSTEM.Status.DisplayError(status)
   quit
}

set rset = statement.%Execute()
if (rset.%SQLCODE '= 0) {
  write "%Execute failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message
  quit
}

while rset.%Next()
{
  write !, rset.%Get("Name")
}
if (rset.%SQLCODE < 0) {
  write "%Next failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message
  quit
}
```

ストアド・プロシージャとしてクエリを呼び出して、それを SQL コンテキストから実行することもできます。"ストアド・プロシージャの定義と使用" を参照してください。

25.3 基本クラス・クエリの定義

基本クラス・クエリを定義するには、以下のようにクエリを定義します。

- (単純なクラス・クエリの場合) タイプは %SQLQuery にする必要があります。
- 引数リストに、クエリが受け入れる引数を指定します。
- 定義の本文に、SQL SELECT 文を記述します。

  この文では、引数名の先頭にコロン (:) を付けて、引数を参照します。

  この SELECT 文には、INTO 節を含めないでください。
- クエリ定義に SqlProc キーワードを含めます。
- オプションとして、ストアド・プロシージャに既定の名前とは別の名前を付ける場合は、クエリ定義内で SqlName キーワードを指定します。

  これらは、コンパイラ・キーワードであるため、クエリ・タイプ (%SQLQuery) の後のパラメータに続く角括弧の内側に含めます。

25.4 例

以下に簡単な例を示します。

Class Member

```
Query ListEmployees(City As %String = "")
   As %SQLQuery (ROWSPEC="ID:%Integer,Name:%String,Title:%String", CONTAINID = 1) [SqlProc,
SqlName=MyProcedureName]
{
SELECT ID,Name,Title FROM Employee
 WHERE (Home_City %STARTSWITH :City)
 ORDER BY Name
}
```

25.5 文字列パラメータの最大長

ADO.NET、ODBC、または JDBC を使用してクラス・クエリを呼び出す場合、既定では、文字列パラメータが 50 文字に切り捨てられます。パラメータの最大文字列長を長くするには、以下の例のようにシグニチャで MAXLEN を指定します。

```
Query MyQuery(MyParm As %String(MAXLEN = 200)) As %SQLQuery [SqlProc]
```

管理ポータルまたは ObjectScript からクエリを呼び出す場合、この切り捨ては行われません。

25.6 関連項目

- カスタム・クラス・クエリの定義
- クラスの定義

26

カスタム・クラス・クエリの定義

基本的なクラス・クエリは、ユーザ向けの結果セットの管理はすべて実行しますが、特定のアプリケーションにとっては十分ではありません。そのような場合、カスタム・クラス・クエリを作成できます。このクエリは、連携する一連のメソッドにより定義されます。これらのメソッドは既定で ObjectScript に書き込まれます。

注釈　　カスタム・クラス・クエリは、シャード・クラスではサポートされていません。

26.1 カスタム・クラス・クエリの定義

カスタム・クエリを定義するには、基本クラス・クエリの定義手順を使用しますが、その手順に以下の変更を加えます。

- クエリ・タイプには、**%Query** を指定します。
- クエリの ROWSPEC パラメータを指定します (クエリ・タイプの後の括弧内で指定します)。このパラメータは、クエリの結果セットの各行に含まれるフィールドの名前、データ型、ヘッダ、および順序に関する情報を提供します。"ROWSPEC パラメータ" を参照してください。
- 必要に応じて、クエリの CONTAINID パラメータを指定します (クエリ・タイプの後の括弧内で指定します)。このパラメータでは、特定の行の ID を格納するフィールドの列番号を指定します (フィールドが存在する場合)。既定値は 1 です。"CONTAINID パラメータ" を参照してください。

  ROWSPEC パラメータと CONTAINID パラメータをまとめて、クエリ仕様と呼びます。
- クエリ定義の本体は空のままにしておきます。以下に例を示します。

  Class Member

```
Query All() As %Query(CONTAINID = 1, ROWSPEC = "Title:%String,Author:%String")
{
}
```

- 同一クラス内に、以下のクラス・メソッドを定義します。
  - querynameExecute ー このメソッドでは、1 回限りの任意の設定を実行する必要があります。
  - querynameFetch ー このメソッドでは、結果セットの行を返す必要があります。後続の呼び出しごとに、その次の行を返します。
  - querynameClose ー このメソッドでは、クリーンアップ処理をすべて実行する必要があります。

  queryname はクエリの名前です。

これらのメソッドは、それぞれ 1 つの引数 (qHandle) を受け入れます。この引数は、参照で渡します。この引数は、これらのメソッド間で情報を渡すために使用できます。

これらのメソッドでクエリを定義します。次のセクションで詳細を説明します。

26.2 メソッドの定義

基本的なデモンストレーションのために、このセクションでは、基本クラス・クエリとして実装することも可能な簡単な例を示します。これらのメソッドは、以下のクエリのコードを実装します。

Class Member

```
Query AllPersons() As %Query(ROWSPEC = "ID:%String,Name:%String,DOB:%String,SSN:%String")
{
}
```

より複雑な例は、次のセクションを参照してください。その他の使用例について、"カスタム・クエリの使用法" も参照してください。

26.2.1 querynameExecute() メソッドの定義

querynameExecute() メソッドでは、必要とされる設定ロジックをすべて提供する必要があります。メソッドの名前は、querynameExecute にする必要があります。queryname はクエリの名前です。このメソッドには、以下のシグニチャが必要です。

```
ClassMethod queryNameExecute(ByRef qHandle As %Binary,
                             additional_arguments) As %Status
```

以下はその説明です。

- qHandle は、このクエリを実装する他のメソッドとの通信に使用します。

  このメソッドでは、querynameFetch メソッドの要求に応じて、qHandle を設定する必要があります。

  qHandle は、正式には %Binary タイプですが、任意の値 (OREF や多次元配列を含む) を保持できます。
- additional_arguments は、クエリで使用できる任意の実行時パラメータです。

このメソッドの実装では、以下の一般的なロジックを使用します。

1. 1 回限りの任意の設定手順を実行します。

   SQL コードを使用するクエリの場合、このメソッドには、一般的にカーソルの宣言とオープンも含めます。
2. querynameFetch メソッドの要求に応じて、qHandle を設定します。
3. ステータス値を返します。

次に示す単純な例は、前述の AllPersons クエリの AllPersonsExecute() メソッドです。

Class Member

```
ClassMethod AllPersonsExecute(ByRef qHandle As %Binary) As %Status
{
    set statement=##class(%SQL.Statement).%New()
    set status=statement.%PrepareClassQuery("Sample.Person","ByName")
    if $$$ISERR(status) { quit status }
    set resultset=statement.%Execute()
    set qHandle=resultset
    Quit $$$OK
}
```

このシナリオでは、このメソッドで qHandle が OREF になるように設定します。具体的には、この OREF は %SQL.StatementResult のインスタンスであり、%Execute() メソッドから返された値です。

前述したように、このクラス・クエリは、カスタム・クラス・クエリではなく、基本クラス・クエリとして実装することもできます。ただし、カスタム・クラス・クエリには、開始点としてダイナミック SQL を使用するものもあります。

26.2.2 querynameFetch() メソッドの定義

querynameFetch() メソッドは、単一のデータ行を $List 形式で返す必要があります。メソッドの名前は、querynameFetch にする必要があります。queryname はクエリの名前です。このメソッドには、以下のシグニチャが必要です。

```
ClassMethod queryNameFetch(ByRef qHandle As %Binary,
                           ByRef Row As %List,
                           ByRef AtEnd As %Integer = 0) As %Status [ PlaceAfter = querynameExecute ]
```

以下はその説明です。

- qHandle は、このクエリを実装する他のメソッドとの通信に使用します。

  このメソッドの実行を InterSystems IRIS® データ・プラットフォームが開始するときに、qHandle は querynameExecute メソッドで設定された値か、このメソッドの前回の呼び出し (ある場合) で設定された値を保持しています。このメソッドでは、それ以降のロジックの要求に応じて、qHandle を設定する必要があります。

  qHandle は、正式には %Binary タイプですが、任意の値 (OREF や多次元配列を含む) を保持できます。

- Row は、返されるデータの行を表す値の %List にするか、Null 文字列 (返す値がない場合) にする必要があります。
- AtEnd は、最後のデータの行に到達したときに 1 にする必要があります。
- PlaceAfter メソッド・キーワードでは、生成されたルーチン・コード内で、このメソッドの位置を制御します。querynameExecute は、具体的な querynameExecute() メソッドの名前で置き換えます。これは、クエリで SQL カーソルを使用する場合には必ず含めます。(この順序を制御する機能は高度な機能であり、慎重に使用する必要があります。インターシステムズは、このキーワードの一般的な使用をお勧めしていません。)

このメソッドの実装では、以下の一般的なロジックを使用します。

1. 返す必要のある結果が、まだあるかどうかを判断するための確認を行います。
2. 適切な場合は、データの行を取得し、%List オブジェクトを作成して、Row 変数に格納します。
3. qHandle は、このメソッドの以降の呼び出し (ある場合) で要求される場合や、querynameClose() メソッドで要求される場合に設定します。
4. 以降の行が存在しない場合は、Row を Null 文字列に設定して、AtEnd を 1 に設定します。
5. ステータス値を返します。

AllPersons の例では、AllPersonsFetch() メソッドは以下のようになります。

Class Member

```
ClassMethod AllPersonsFetch(ByRef qHandle As %Binary, ByRef Row As %List, ByRef AtEnd As %Integer = 0)
 As %Status
[ PlaceAfter = AllPersonsExecute ]
{
  set rset = $get(qHandle)
  if rset = "" quit $$$OK

  if rset.%Next() {
    set Row=$lb(rset.%GetData(1),rset.%GetData(2),rset.%GetData(3),rset.%GetData(4))
    set AtEnd=0
  } else {
    if (rset.%SQLCODE < 0) {write "%Next failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message
quit}
    set Row=""
    set AtEnd=1
  }
    quit $$$OK
}
```

このメソッドでは qHandle 引数を使用している点に注目してください。この引数は、%SQL.StatementResult オブジェクトを提供します。その後で、そのクラスのメソッドを使用してデータを取得しています。このメソッドでは、$List を構築して、それを Row 変数に格納しています。これは、単一のデータ行として返されます。さらに、このメソッドには、取得できるデータがなくなったときに、AtEnd 変数を設定するロジックが含まれている点にも注目してください。

前述したように、このクラス・クエリは、カスタム・クラス・クエリではなく、基本クラス・クエリとして実装することもできます。この例は、Row 変数と AtEnd 変数の設定についての実例を示すことを目的としています。

26.2.3 querynameClose() メソッド

querynameClose() メソッドでは、データ取得の完了後に必要になるクリーンアップをすべて実行する必要があります。メソッドの名前は、querynameClose にする必要があります。queryname はクエリの名前です。このメソッドには、以下のシグニチャが必要です。

```
ClassMethod queryNameClose(ByRef qHandle As %Binary) As %Status [ PlaceAfter = querynameFetch ]
```

以下はその説明です。

- qHandle は、このクエリを実装する他のメソッドとの通信に使用します。

  このメソッドの実行を InterSystems IRIS が開始するときに、qHandle は querynameFetch メソッドの最後の呼び出しで設定された値を保持しています。

- PlaceAfter メソッド・キーワードでは、生成されたルーチン・コード内で、このメソッドの位置を制御します。querynameFetch は、具体的な querynameFetch() メソッドの名前で置き換えます。これは、クエリで SQL カーソルを使用する場合には必ず含めます。(この順序を制御する機能は高度な機能であり、慎重に使用する必要があります。インターシステムズは、このキーワードの一般的な使用をお勧めしていません。)

このメソッドの実装では、メモリからの変数の削除、SQL カーソルのクローズ、またはその他のクリーンアップを必要に応じて実行します。このメソッドはステータス値を返す必要があります。

AllPersons の例では、AllPersonsClose() メソッドは以下のようになります。

例えば、ByNameClose() メソッドのシグニチャは以下のようになります。

Class Member

```
ClassMethod AllPersonsClose(ByRef qHandle As %Binary) As %Status [ PlaceAfter = AllPersonsFetch ]
{
        Set qHandle=""
        Quit $$$OK
}
```

26.2.4 カスタム・クエリ用に生成されるメソッド

システムは、querynameGetInfo() と querynameFetchRows() を自動的に生成します。ユーザのアプリケーションは、上記のメソッドを直接呼び出すことはありません。

26.3 クラス・クエリのパラメータ

ここでは、カスタム・クラス・クエリ内で指定するパラメータの詳細について説明します。

26.3.1 ROWSPEC パラメータ

クエリの ROWSPEC パラメータでは、各行に含まれるフィールドの名前、データ型、ヘッダ、および順序に関する情報を提供します。これらの変数名は、引用符付きのコンマ区切りのリストで、データ型の形式は以下のとおりです。

```
ROWSPEC = "Var1:%Type1,Var2:%Type2[:OptionalDescription],Var3"
```

ROWSPEC は、コンマ区切りのリストとしてフィールドの順序を指定します。各フィールドの情報は、そのフィールドの名前、データ型（対応するプロパティのデータ型と異なる場合）、およびヘッダ（オプション）をコロンで区切ったリストで構成されています。

ROWSPEC パラメータの要素数は、クエリのフィールド数と一致している必要があります。数が異なる場合、InterSystems IRIS® データ・プラットフォームは Cardinality Mismatch エラーを返します。

例えば、Sample.Person クラスの ByName クエリは以下のようになります。

Class Member

```
Query ByName(name As %String = "")
    As %SQLQuery(CONTAINID = 1, ROWSPEC = "ID:%Integer,Name,DOB,SSN", SELECTMODE = "RUNTIME")
   [ SqlName = SP_Sample_By_Name, SqlProc ]
{
        SELECT ID, Name, DOB, SSN
            FROM Sample.Person
            WHERE (Name %STARTSWITH :name)
            ORDER BY Name
}
```

ここでは、最初のフィールドが行 ID（既定）であることを CONTAINID パラメータで指定しています。SELECT 文で指定されている最初のフィールドも ID である点に注意してください。ROWSPEC パラメータでは、各フィールドが ID（整数として処理される）、Name、DOB、および SSN であることを指定しています。同様に、SELECT 文にも ID、Name、DOB、および SSN の各フィールドが、この順序で含まれています。

26.3.2 CONTAINID パラメータ

CONTAINID は、ID を返す列の番号（既定では 1）に設定する必要があります。ID を返す列がない場合は 0 に設定します。

注釈　システムは CONTAINID の値を検証しません。ユーザがこのパラメータに対して無効な値を指定した場合でも、エラー・メッセージは発行されません。つまり、ユーザのクエリ処理ロジックがこの情報に基づいている場合で、CONTAINID パラメータが不正に設定されているときは、不整合が生じる可能性があります。

26.3.3 クエリ・クラスのその他のパラメータ

ROWSPEC と CONTAINID に加え、以下に示すクエリのパラメータも指定できます。これらは、%SQLQuery のクラス・パラメータです。

- SELECTMODE
- COMPILEMODE

詳細は、%Library.SQLQuery と、そのスーパークラス %Library.Query のクラスリファレンスを参照してください。

26.4 カスタム・クエリのパラメータの定義

カスタム・クエリでパラメータを受け入れる必要がある場合は、以下の手順に従います。

- それらをクエリ・クラス・メンバの引数リストに含めます。以下の例は、MyParm という名前のパラメータを使用しています。

  Class Member

  ```
  Query All(MyParm As %String) As %Query(CONTAINID = 1, ROWSPEC = "Title:%String,Author:%String")
  {
  }
  ```

- 同じパラメータを、クエリ・クラス・メンバの順序と同じ順序で、querynameExecute メソッドの引数リストに含めます。
- querynameExecute メソッドの実装で、必要に応じたパラメータを使用します。

注釈　ADO.NET、ODBC、または JDBC を使用してクラス・クエリを呼び出す場合、既定では、文字列パラメータが 50 文字に切り捨てられます。パラメータの最大文字列長を長くするには、以下の例のようにシグニチャで MAXLEN を指定します。

```
Query MyQuery(MyParm As %String(MAXLEN = 200)) As %Query [SqlProc]
```

管理ポータルまたは ObjectScript からクエリを呼び出す場合、この切り捨ては行われません。

26.5 カスタム・クエリを使用する場合

以下のリストでは、カスタム・クエリが適切になる、いくつかのシナリオを示しています。

- 返されるデータに特定の行を含めるかどうかを決定するために、非常に複雑なロジックを使用する必要がある場合。querynameFetch() メソッドには、任意の複雑なロジックを含めることができます。
- 現在の使用事例には不便な形式でデータを返す API がある場合。このようなシナリオでは、Row 変数の要求に応じて、その形式のデータを $List に変換する querynameFetch() メソッドを定義することになります。
- クラス・インタフェースを備えていないグローバルに、データが格納されている場合。
- データにアクセスするために、ロール・エスカレーションが必要になる場合。このシナリオでは、querynameExecute() メソッド内でロール・エスカレーションを実行します。

- データにアクセスするために、ファイル・システムへの外部呼び出しが必要になる場合 (例えば、ファイルのリストを構築する場合)。このシナリオでは、querynameExecute() メソッド内からコールアウトを実行して、その結果を qHandle か、グローバルに格納します。
- データを取得する前に、セキュリティ・チェック、接続のチェック、またはその他の特別な設定作業を実行する必要がある場合。そのような作業は、querynameExecute() メソッド内で実行することになります。

26.6 SQL カーソルとクラス・クエリ

クラス・クエリで SQL カーソルを使用する場合は、以下の点に注意してください。

- **%SQLQuery** タイプのクエリから生成されたカーソルは、自動的に Q14 などの名前を持ちます。

  カーソルに固有の名前が与えられていることを確認してください。
- エラー・メッセージは、通常、桁を 1 桁余分に持っている内部カーソル名を参照します。したがって、カーソル Q140 に対するエラー・メッセージは、おそらく Q14 を参照しています。
- クラス・コンパイラは、カーソルを使用する前に、カーソル宣言を見つける必要があります。そのため、カーソルを使用するカスタム・クエリを定義する際には、特別な注意を払う必要があります。

  DECLARE 文 (通常は querynameExecute() メソッド内) は、Close や Fetch と同じ MAC ルーチン内にあることが必要です。また、Close や Fetch よりも先に来る必要があります。このようにするには、ここで前述したように、querynameFetch() と querynameClose() の両方のメソッド定義で、メソッド・キーワード PlaceAfter を使用してください。

26.7 関連項目

- クラス・クエリの定義と使用
- クラスの定義

27

XData ブロックの定義と使用

XData ブロックは、コンパイル後のクラスで使用するためにクラス定義に含めるデータの名前とユニットで構成されるクラス・メンバです。

27.1 基本

XData ブロックは、クラス定義に含めるデータの名前付きユニットです。通常、クラスのメソッドで使用します。一般的に、これは適格な XML ドキュメントですが、JSON や YAML など、他の形式のデータで構成することもできます。

統合開発環境 (IDE) で直接入力することによって XData ブロックを作成できます。

XData ブロックは、名前付きクラス・メンバ です (プロパティ、メソッドなどと同様)。使用できる XData ブロック・キーワードは以下のとおりです。

- SchemaSpec – オプションで、XData を検証できる XML スキーマを指定します。
- XMLNamespace – オプションで、XData ブロックが属する XML ネームスペースを指定します。また、当然ながら、XData ブロック自体の中にもネームスペース宣言を含めることができます。
- MimeType – XData ブロックのコンテンツの MIME タイプ (正式には、インターネット・メディア・タイプ)。既定値は `text/xml` です。

XML の格納に使用する場合、XData ブロックは、1 つのルート XML 要素と任意の有効なコンテンツで構成される必要があります。

27.2 XML の例

プログラム的に任意の XData ブロック内の XML ドキュメントにアクセスするには、%Dictionary.CompiledXData および %Dictionary パッケージのその他のクラスを使用します。

少量のシステム・データを定義する場合は、XData ブロックが役に立ちます。例えば、`EPI.AllergySeverity` クラスに、プロパティ `Code` (内部使用) とプロパティ `Description` (ユーザへの表示用) があるとします。このクラスに、次のように XData ブロックを記述できます。

Class Member

```
XData LoadData
{
<table>
 <row>1^Minor</row>
 <row>2^Moderate</row>
 <row>3^Life-threatening</row>
 <row>9^Inactive</row>
 <row>99^Unable to determine</row>
</table>
}
```

同じクラスに、次のように、この XData ブロックを読み取ってテーブルを生成するクラス・メソッドを記述することもできます。

Class Member

```
ClassMethod Setup() As %Status
{
   //first kill extent
   do ..%KillExtent()

   // Get a stream of XML from the XData block contained in this class
   Set xdataID="EPI.AllergySeverity||LoadData"
   Set compiledXdata=##class(%Dictionary.CompiledXData).%OpenId(xdataID)
   Set tStream=compiledXdata.Data
   If '$IsObject(tStream) Set tSC=%objlasterror Quit

   set status=##class(%XML.TextReader).ParseStream(tStream,.textreader)
   //check status
   if $$$ISERR(status) do $System.Status.DisplayError(status) quit

   //iterate through document, node by node
   while textreader.Read()
   {
       if (textreader.NodeType="chars")
       {
           set value=textreader.Value
           set obj=..%New()
           set obj.Code=$Piece(value,"^",1)
           set obj.Description=$Piece(value,"^",2)
           do obj.%Save()
       }
   }
}
```

これから以下の点がわかります。

- XData に記述する XML は必要最小限で済みます。つまり、独自の要素や属性を持つ XML 要素としてアレルギー反応程度を記述する代わりに、単なるデータの行を区切り文字列として XData ブロックに記述します。これにより、見やすい形式で設定データを記述できます。
- `EPI.AllergySeverity` クラスは、XML 対応ではなく、また XML 対応にする必要もありません。

27.3 JSON の例

クラスには、次のように JSON コンテンツが含まれる XData ブロックも記述できます。

Class Member

```
XData LoadJSONData [MimeType = "application/json"]
{
   {
      "person":"John",
      "age":30,
      "car":"Ford"
   }
}
```

同じクラスに、次のように、この XData ブロックを読み取ってダイナミック・オブジェクトを生成するクラス・メソッドを記述することもできます。

Class Member

```
/// Reads a JSON XData block
ClassMethod SetupJSON() As %Status
{

   // Get a stream of JSON from the XData block contained in this class
   Set xdataID="Demo.XData||LoadJSONData"
   Set compiledXdata=##class(%Dictionary.CompiledXData).%OpenId(xdataID)
   Set tStream=compiledXdata.Data
   If '$IsObject(tStream) Set tSC=%objlasterror Quit

   // Create a dynamic object from the JSON content and write it as a string
   Set dynObject = {}.%FromJSON(tStream)
   Write dynObject.%ToJSON()

}
```

27.4 YAML の例

クラスには、以下の Swagger API 仕様のように、YAML コンテンツが含まれる XData ブロックも記述できます。

```
XData SampleAPI [mimetype = "application/yaml"]
{
swagger: "2.0"
info:
  title: Sample API
  description: API description in Markdown.
  version: 1.0.0
host: api.example.com
basePath: /v1
schemes:
  - https
paths:
  /users:
    get:
      summary: Returns a list of users.
      description: Optional extended description in Markdown.
      produces:
        - application/json
      responses:
        200:
          description: OK
}
```

このクラス・メソッドは XData ブロックを読み取り、そのコンテンツを 1 行ずつ書き込みます。

Class Member

```
/// Reads a YAML XData block
ClassMethod SetupYAML() As %Status
{

   // Get a stream of YAML from the XData block contained in this class
   Set xdataID="Demo.XData||SampleAPI"
   Set compiledXdata=##class(%Dictionary.CompiledXData).%OpenId(xdataID)
   Set tStream=compiledXdata.Data
   If '$IsObject(tStream) Set tSC=%objlasterror Quit

   // Write the content from the stream, line by line
   While 'tStream.AtEnd {
      Write tStream.ReadLine(,.sc,.eol)
      If eol { Write ! }
   }
}
```

28
クラス・プロジェクションの定義

ここでは、クラスをコンパイルまたは削除したときの動作をカスタマイズする手段を提供するクラス・プロジェクションについて説明します。

28.1 概要

クラス・プロジェクションは、クラスをコンパイルまたは削除したときの動作をカスタマイズする手段を提供します。クラス・プロジェクションは、クラス定義をプロジェクション・クラスと関連させます。プロジェクション・クラス (%Projection.AbstractProjection クラスから派生) は、InterSystems IRIS® データ・プラットフォームが自動的に追加のコードを生成するために、次の 2 つの時点で使用するメソッドを提供します。

- クラスがコンパイルされるとき
- クラスが削除されるとき

このメカニズムは、Java プロジェクション (プロジェクションの語源) によって、クラスがコンパイルされるときは常に、必要なクライアント・バインディング・コードを自動的に生成するために使用されます。

28.2 クラスにプロジェクションを追加する

クラス定義にプロジェクションを追加するには、クラス定義内の Projection 文を使用します。

Class Definition

```
class MyApp.Person extends %Persistent
{
Projection JavaClient As %Projection.Java(ROOTDIR="c:\java");
}
```

この例では、%Projection.Java プロジェクション・クラスを使用する JavaClient というプロジェクションを定義しています。プロジェクション・クラスのメソッドが呼び出されるとき、そのメソッドは ROOTDIR パラメータの値を受け取ります。

クラスには、一意に名付けられた複数のプロジェクションを含めることができます。複数のプロジェクションの場合、各プロジェクション・クラスのメソッドは、クラスがコンパイルされるとき、または削除されるときに呼び出されます。複数のプロジェクションが処理される順序は、未定義です。

InterSystems IRIS には、以下のプロジェクション・クラスが用意されています。

クラス	説明
%Projection.Java	Java からのクラスへのアクセスを可能にする、Java クライアント・クラスを生成します。
%Projection.Monitor	このクラスを、ログ・モニタで使用するルーチンとして登録します。メタデータが Monitor.Application、Monitor.Alert、Monitor.Item、および Monitor.ItemGroup に書き込まれます。Monitor.Sample という新規の永続クラスが作成されます。
%Projection.MV	MV からクラスへのアクセスを可能にする MV クラスを生成します。
%Projection.StudioDocument	このクラスをスタジオで使用するルーチンとして登録します。
%Studio.Extension.Projection	XData ・ブロックをメニュー・テーブルに投影します。

また、独自のプロジェクション・クラスを生成し、組み込みのプロジェクション・クラスと同様に使用することもできます。

28.3 新規プロジェクション・クラスの作成

新規のプロジェクション・クラスを作成するには、%Projection.AbstractProjection クラスのサブクラスを作成し、プロジェクション・インタフェース・メソッドを実装して (サブセクションを参照)、必要なクラス・パラメータを定義します。例えば、以下のようになります。

Class Definition

```
Class MyApp.MyProjection Extends %Projection.AbstractProjection
{

Parameter MYPARAM;

/// This method is invoked when a class is compiled
ClassMethod CreateProjection(cls As %String, ByRef params) As %Status
{
    // code here...
    QUIT $$$OK
}

/// This method is invoked when a class is 'uncompiled'
ClassMethod RemoveProjection(cls As %String, ByRef params, recompile as %Boolean) As %Status
{
    // code here...
    QUIT $$$OK
  }
}
```

28.3.1 プロジェクション・インタフェース

すべてのプロジェクション・クラスは、クラスのライフ・サイクル中に発生する特定のイベントに応じて呼び出される一連のメソッド、プロジェクション・インタフェースを実装します。このインタフェースは、以下のメソッドで構成されています。

CreateProjection()

CreateProjection() メソッドは、クラス定義のコンパイルが完了した後に、クラス・コンパイラによって呼び出されるクラス・メソッドです。このメソッドは、プロジェクション用に定義されたパラメータ値 (パラメータ名によって添え字にされている) を含む配列と同様、コンパイルされたクラスの名前を渡されます。

RemoveProjection()

RemoveProjection() メソッドは、以下のいずれかの場合に呼び出されるクラス・メソッドです。

- クラス定義が削除されたとき
- クラスのリコンパイルの開始時

このメソッドは、削除されるクラスの名前、プロジェクション用に定義されたパラメータ値を含む配列 (パラメータ名によって添え字にされている)、メソッドがリコンパイルの一部として呼び出されているかを示す、またはクラス定義が削除されていることを示すフラグを渡されます。

プロジェクションを含むクラス定義がコンパイルされるとき、以下のイベントが発生します。

1. クラスが以前にコンパイルされている場合、新規のコンパイルが開始される前に、そのクラスはアンコンパイルされます (つまり、以前のコンパイルのすべての結果は削除されます)。ここで、コンパイラはリコンパイルが発生することを示したフラグ付きで、すべてのプロジェクションに対して RemoveProjection() メソッドを呼び出します。

   この時点でクラスは存在しないので、RemoveProjection() メソッド内部から関連するクラスのメソッドを呼び出すことができないことに注意してください。

   また、以前に (プロジェクションなしで) コンパイルされたクラスに対する新規のプロジェクション定義を追加する場合、CreateProjection() メソッドが今までに呼び出されたことがなくても、コンパイラは次のコンパイル時に RemoveProjection() メソッドを呼び出します。このメソッドを実装するときは、この可能性に備えて計画するようにしてください。

2. クラスが完全にコンパイルされた後 (つまり、使用準備ができたということ)、コンパイラはすべてのプロジェクションに対して CreateProjection() メソッドを呼び出します。

クラス定義が削除されたとき、すべてのプロジェクションに対して、RemoveProjection() メソッドが削除フラグを付けて呼び出されます。

28.4 関連項目

- クラスの定義
- 投影の構文とキーワード

29

コールバック・メソッドの定義

コールバック・メソッドは、システム・メソッドによって呼び出され、ユーザが提供する追加処理を行うことができます。コールバック・メソッドは、%On または On で始まり、通常、それらを呼び出すメソッドの名前が続く名前を持つことによって識別できます。

コールバック・メソッドを実装しているシステム・メソッドを実行すると、そのメソッドによりコールバック・メソッドが呼び出されます。例えば、%OnDelete() が実装されている場合、%Delete() が %OnDelete() を呼び出します。

重要　　コールバック・メソッドは、直接実行しないでください。

テーブル 29-1: コールバック・メソッド

コールバック名	実装対象	メソッド・タイプ	プライベート・メソッドかどうか
%OnAddToSaveSet()	%RegisteredObject	インスタンス	はい
%OnAfterBuildIndices()	%Persistent	クラス	はい
%OnAfterDelete()	%Persistent	クラス	はい
%OnAfterPurgeIndices()	%Persistent	クラス	はい
%OnAfterSave()	%Persistent	インスタンス	はい
%OnBeforeBuildIndices()	%Persistent	クラス	はい
%OnBeforePurgeIndices()	%Persistent	クラス	はい
%OnBeforeSave()	%Persistent	インスタンス	はい
%OnClose()	%RegisteredObject	インスタンス	はい
%OnConstructClone()	%RegisteredObject	インスタンス	はい
%OnDelete()	%Persistent	クラス	はい
%OnDeleteFinally()	%Persistent	クラス	いいえ
%OnNew()	%RegisteredObject	インスタンス	はい
%OnOpen()	%Persistent, %SerialObject	インスタンス	はい
%OnOpenFinally()	%Persistent	インスタンス	いいえ
%OnReload	%Persistent	インスタンス	はい
%OnRollBack	%Persistent	インスタンス	はい

コールバック名	実装対象	メソッド・タイプ	プライベート・メソッドかどうか
%OnSaveFinally()	%Persistent	インスタンス	いいえ
%OnValidateObject()	%RegisteredObject	インスタンス	はい
%OnDetermineClass()	%Storage.Persistent	クラス	いいえ

注釈　プライベート・メソッドのコールバックのドキュメントは、[クラスリファレンス] の右上隅の **[プライベート]** チェック・ボックスにチェックが付いている場合にのみ、クラスリファレンスに表示されます。

29.1 コールバックおよびトリガ

SQL とオブジェクト・アクセスの両方を使用するアプリケーションでトリガを実装する場合、通常はオブジェクト・アクセスの同じ場所で同じロジックを呼び出すことをお勧めします。例えば、行を削除するときに監査レコードを挿入する場合は、オブジェクトを削除するときにも監査レコードを挿入します。

`Foreach = row/object` を使用してトリガが定義されている場合、トリガはオブジェクト・アクセスの特定の時点でも呼び出されます。"トリガとトランザクション" を参照してください。

ただし、そのようなトリガが作成できない場合に、SQL とオブジェクトの動作を前述のように同期させるには、1 つ以上のコールバックを実装する必要があります。これらの実装では、トリガ定義で使用したロジックと同等のロジックを使用します。以下のコールバック・メソッドには、SQL トリガの該当機能と同等の機能があります。

- %OnBeforeSave() ― BEFORE INSERT、BEFORE UPDATE
- %OnAfterSave() ― AFTER INSERT、AFTER UPDATE
- %OnDelete() ― BEFORE DELETE

トリガの詳細は、"トリガの使用法"、または "InterSystems SQL リファレンス" の "CREATE TRIGGER" ページを参照してください。

29.2 %OnAddToSaveSet()

このインスタンス・メソッドは、%AddToSaveSet() によって現在のオブジェクトが SaveSet に追加されるときに必ず呼び出されます。%AddToSaveSet() は、以下によって呼び出すことができます。

- **%Persistent** のインスタンスに対する %Save()
- **%SerialObject** のインスタンスに対する %GetSwizzleObject()
- 参照オブジェクトの %AddToSaveSet()

%OnAddToSaveSet() が他のオブジェクトを修正する場合、%OnAddToSaveSet() は修正されたオブジェクトで %AddToSaveSet() を呼び出す必要があります。%AddToSaveSet() を %OnAddToSaveSet() から呼び出す際、depth を最初の引数として渡し、1 (リテラルの 1) を 2 番目の引数として渡します。

例えば、MyPerson オブジェクトで %Save() を呼び出すとき、システムは MyPerson が参照するオブジェクトのリストを生成します。SaveSet は保存されたオブジェクトと、それを参照するすべてのオブジェクトで構成される、オブジェクトのリストです。この例では、SaveSet は参照されたオブジェクト MySpouse や MyDoctor などを含みます。詳細は、"オブジェクトの保存" を参照してください。

%OnAddToSaveSet() のシグニチャは以下のとおりです。

Class Member

```
Method %OnAddToSaveSet(depth As %Integer,
                       insert As %Integer,
                       callcount As %Integer)
                        As %Status [ Private, ServerOnly = 1 ]
{
    // body of method here...
}
```

以下はその説明です。

depth

SaveSet 構文の内部状況を表す %AddToSaveSet() から渡される整数値。%OnAddToSaveSet() を使用して SaveSet に他の任意のオブジェクトを追加する場合、この値を変更せずに %AddToSaveSet() に渡します。

insert

保存されているオブジェクトが、エクステントに現在挿入されている場合は (1) を、既にエクステントの一部ある場合は (0) を示すフラグ。

callcount

このオブジェクトに対して %OnAddToSaveSet が呼び出された回数。オブジェクト・リファレンスはネットワーク化されているという特徴があるため、%AddToSaveSet は同じオブジェクトで複数回呼び出されることがあります。

メソッドは、**%Status** コードを返します。失敗ステータスの場合、保存は失敗してトランザクションはロールバックされます。

%AddToSaveSet() を呼び出すことで、オブジェクトの更新や、新規オブジェクトの生成、オブジェクトの削除、およびオブジェクト自身を現在の SaveSet に含めるかをそのオブジェクトに尋ねることができます。現在のインスタンスや、その下位のいずれかを変更する場合は、変更を行ったことをシステムに伝える必要があります。これを行うには、変更を行ったインスタンスの %AddToSaveSet() を呼び出して、Refresh 引数を 1 に指定します。

%OnAfterSave()、%OnBeforeSave()、および %OnValidateObject() で定義された変更制約は、どれも %OnAddToSaveSet() には適用されません。

%OnAddToSaveSet() を使用してオブジェクトを削除する場合、ダングリング参照をすべて削除するために、%RemoveFromSaveSet() を呼び出す必要があります。

このメソッドは、**%Library.RegisteredObject** の任意のサブクラスでオーバーライドできます。

29.3 %OnAfterBuildIndices()

このクラス・メソッドは、%BuildIndices() メソッドがインデックスを構築し、$SortEnd を実行した後、そして拡張ロックをリリースする直前に (要求されている場合)、%BuildIndices() メソッドによって呼び出されます。

そのシグニチャは、以下のとおりです。

```
ClassMethod %OnAfterBuildIndices(indexlist As %String(MAXLEN="") = "") As %Status [ Abstract, Private,
 ServerOnly = 1 ]
{
    // body of method here...
}
```

各項目の内容は次のとおりです。

indexlist

インデックス名の $List。

29.4 %OnAfterDelete()

このクラス・メソッドは、指定したオブジェクトが削除された直後 (%DeleteData() メソッドの呼び出しに成功した直後) に %Delete() または %DeleteId() メソッドによって呼び出されます。このメソッドを使用すると、保存するオブジェクトのスコープ外のアクション (後で発生する通知アクションのキューなど) を実行できます。

そのシグニチャは、以下のとおりです。

Class Member

```
ClassMethod %OnAfterDelete(oid As %ObjectIdentity) As %Status [ Private, ServerOnly = 1 ]
{
    // body of method here...
}
```

各項目の内容は次のとおりです。

oid

削除するオブジェクト。

このメソッドは %Status コードを返します。この場合は、失敗のステータスによって %Delete() または %DeleteId() が失敗し、アクティブなトランザクションがある場合は、そのトランザクションのロールバックが発生します。%Delete() または %DeleteId() からエラー (このメソッド自身のエラー、または %DeleteData() で発生したエラー) が返ると、%OnAfterDelete() は呼び出されません。

%Library.Persistent のサブクラスには、このメソッドをオーバーライドするオプションがあります。

29.5 %OnAfterPurgeIndices()

このクラス・メソッドは、%PurgeIndices() メソッドが処理をすべて完了した後、%PurgeIndices() メソッドによって呼び出されます。

そのシグニチャは、以下のとおりです。

```
ClassMethod %OnAfterPurgeIndices(indexlist As %String(MAXLEN="") = "") As %Status [ Abstract, Private,
 ServerOnly = 1 ]
{
    // body of method here...
}
```

各項目の内容は次のとおりです。

indexlist

インデックス名の $List。

29.6 %OnAfterSave()

このインスタンス・メソッドは、オブジェクトが保存された直後に %Save() メソッドによって呼び出されます。このメソッドを使用すると、保存するオブジェクトのスコープ外のアクション (後で発生する通知アクションのキューなど) を実行できます。例えば、ある特定の金額以上の預金を使用している銀行が、顧客に預金規定の説明を送付する場合などです。

そのシグニチャは、以下のとおりです。

Class Member

```
Method %OnAfterSave(insert as %Boolean)As %Status [ Private, ServerOnly = 1 ]
{
    // body of method here...
}
```

各項目の内容は次のとおりです。

insert

保存されているオブジェクトが、エクステントに現在挿入されている場合は (1) を、既存のオブジェクトの更新の場合は (0) を示すフラグ。

このメソッドは、%Status コードを返します。失敗ステータスの場合、%Save() は失敗して、最終的にトランザクションをロールバックします。

%Library.Persistent のサブクラスには、このメソッドをオーバーライドするオプションがあります。

29.7 %OnBeforeBuildIndices()

このクラス・メソッドは、%BuildIndices() メソッドが拡張ロックを取得した後 (要求されている場合)、インデックスの構築を開始する前に、%BuildIndices() メソッドによって呼び出されます。

そのシグニチャは、以下のとおりです。

```
ClassMethod %OnBeforeBuildIndices(ByRef indexlist As %String(MAXLEN="") = "") As %Status [ Abstract,
Private, ServerOnly = 1 ]
{
    // body of method here...
}
```

各項目の内容は次のとおりです。

indexlist

インデックス名の $List。このパラメータは参照によって渡されます。%OnBeforeBuildIndices() の実装でこの値を変更した場合、%BuildIndices() は変更された値を受け取ります。

29.8 %OnBeforePurgeIndices()

このクラス・メソッドは、%PurgeIndices() メソッドが動作を開始する前に、%PurgeIndices() メソッドによって呼び出されます。このメソッドがエラーを返した場合、%PurgeIndices() はインデックス構造を削除せずに、%PurgeIndices() の呼び出し元にエラーを返し、直ちに終了します。

そのシグニチャは、以下のとおりです。

```
ClassMethod %OnBeforePurgeIndices(ByRef indexlist As %String(MAXLEN="") = "") As %Status [ Abstract,
Private, ServerOnly = 1 ]
{
    // body of method here...
}
```

各項目の内容は次のとおりです。

indexlist

インデックス名の $List。このパラメータは参照によって渡されます。%OnBeforePurgeIndices() の実装でこの値を変更した場合、%PurgeIndices() は変更された値を受け取ります。

29.9 %OnBeforeSave()

このインスタンス・メソッドは、オブジェクトが保存される直前に %Save() メソッドによって呼び出されます。このメソッドを使用すると、アクションを完了してからインスタンスをディスクに保存する前に、ユーザに確認を求めることができるようになります。

重要　%OnBeforeSave() で現在のオブジェクトを更新することは有効ではありません。保存する前にオブジェクトを修正する場合は、代わりに %OnAddToSaveSet() コールバックを実装し、そのメソッドにロジックを含めます。

そのシグニチャは、以下のとおりです。

Class Member

```
Method %OnBeforeSave(insert as %Boolean) As %Status [ Private, ServerOnly = 1 ]
{
    // body of method here...
}
```

各項目の内容は次のとおりです。

insert

保存されているオブジェクトが、エクステントに現在挿入されている場合は (1) を、既にエクステントの一部ある場合は (0) を示すフラグ。

このメソッドは、%Status コードを返します。失敗ステータスの場合、保存が失敗します。

%Library.Persistent のサブクラスには、このメソッドをオーバーライドするオプションがあります。

29.10 %OnClose()

このインスタンス・メソッドは、オブジェクトが破棄される直前に呼び出されるので、ユーザはロック解除や一時的データ構造の削除などの補助項目の処理を実行する機会を得られます。

そのシグニチャは、以下のとおりです。

Class Member

```
Method %OnClose() As %Status [ Private, ServerOnly = 1 ]
{
    // body of method here...
}
```

このメソッドは %Status コードを返しますが、この場合、失敗のステータスは単なる情報的なものであり、オブジェクトの破棄を阻止するものではありません。

%Library.RegisteredObject のサブクラスには、このメソッドをオーバーライドするオプションがあります。

29.11 %OnConstructClone()

このインスタンス・メソッドは、オブジェクトクローン化のためにオブジェクト構造が割り当てられ、すべてのデータがそこにコピーされた直後に、%ConstructClone() メソッドによって呼び出されます。このメソッドを使用して、ロックの取得や、クローンのプロパティ値のリセットなど、クローンのオブジェクトに関連する追加のアクションを実行できます。

そのシグニチャは、以下のとおりです。

Class Member

```
Method %OnConstructClone(object As %RegisteredObject,
                         deep As %Boolean,
                         ByRef cloned As %String)
                         As %Status [ Private, ServerOnly = 1 ]
{
    // body of method here...
}
```

各項目の内容は次のとおりです。

object

クローン化されたオブジェクトの OREF。

deep

クローン処理の“深さ”。0 では、クローンが関連するオブジェクトはオリジナルと同じものをポイントします。1 では、クローンされるオブジェクトに関連するオブジェクトもクローンされ、クローンは独自の関連オブジェクトを持つようになります。

cloned

使用法が %OnConstructClone() の呼び出し方によって変わる引数。詳細は、%Library.RegisteredObject のクラス・ドキュメントを参照してください。

このメソッドは、%Status コードを返します。失敗ステータスの場合、クローンは作成されません。

%Library.RegisteredObject のサブクラスには、このメソッドをオーバーライドするオプションがあります。

29.12 %OnDelete()

このクラス・メソッドは、オブジェクトが削除される直前に %Delete() または %DeleteId() メソッドによって呼び出されます。このメソッドは、オブジェクトを削除してもデータの整合性を破壊しないために使用されます。例えば、他のオブジェクトを含むように設計されたオブジェクトは、空になったときだけ削除されることを確実にします。

そのシグニチャは、以下のとおりです。

Class Member

```
ClassMethod %OnDelete(oid As %ObjectIdentity) As %Status [ Private, ServerOnly = 1 ]
{
    // body of method here...
}
```

各項目の内容は次のとおりです。

oid

削除されるオブジェクトに対するオブジェクト識別子。

このメソッドは、%Status コードを返します。失敗ステータスの場合、削除は中止されます。

削除されるオブジェクトのプロパティにアクセスするには、渡される OID を ID に変換して、その ID を持つオブジェクトを開きます。以下に例を示します。

```
ClassMethod %OnDelete(oid As %ObjectIdentity) As %Status [ Private, ServerOnly = 1 ]
{
    set id = $$$oidPrimary(oid)
    set obj = ..%OpenId(id)
    write "Deleting object with name ", obj.Name, !
    return 1
}
```

%Library.Persistent のサブクラスには、このメソッドをオーバーライドするオプションがあります。

29.13 %OnDeleteFinally()

このクラス・メソッドは、すべての処理の完了後、呼び出し元に戻る直前に %Delete() または %DeleteId() メソッドによって呼び出されます。%Delete() メソッドまたは %DeleteId() メソッドがトランザクションを開始していた場合、そのトランザクションはこのコールバックを呼び出す前に完了されます。このメソッドは値を返さず、メソッドを呼び出したメソッドの結果を変更することはできません。

そのシグニチャは、以下のとおりです。

Class Member

```
ClassMethod %OnDeleteFinally(oid As %ObjectIdentity, status As %Status) [ ServerOnly = 1 ]
{
    // body of method here...
}
```

各項目の内容は次のとおりです。

oid

削除されるオブジェクトに対するオブジェクト識別子。

status

ユーザに報告されるステータス。このコールバック・メソッドでステータスを変更することはできません。

%Library.Persistent のサブクラスには、このメソッドをオーバーライドするオプションがあります。

29.14 %OnNew()

このインスタンス・メソッドは、オブジェクトに対するメモリが割り当てられ、プロパティが初期化されるときに、%New() メソッドによって呼び出されます。

そのシグニチャは、以下のとおりです。

Class Member

```
Method %OnNew(initvalue As %String) As %Status [ Private, ServerOnly = 1 ]
{
    // body of method here...
}
```

各要素の内容は以下のとおりです。

initvalue

次のメモで説明するように、オーバーライドされない限り、オブジェクトの設定時にメソッドが使用する文字列。

重要 %OnNew() の引数は、%New() の引数と一致する必要があります。このメソッドをカスタマイズする際、引数を %New() から受け取る変数およびタイプでオーバーライドします。例えば、%New() が、誕生日を示す dob および姓名を示す name の 2 つの引数を受け入れる場合、%OnNew() のシグニチャは、以下のようになります。

Class Member

```
Method %OnNew(dob as %Date = "", name as %Name = "") as %Status [ Private, ServerOnly =
 1 ]
{
 // body of method here...
}
```

このメソッドは、%Status コードを返します。失敗ステータスの場合、オブジェクトの作成は中止されます。

例えば、インスタンスが Name プロパティの値を持つ必要があるクラスの場合、コールバックは、以下の形式のものになります。

Class Member

```
Method %OnNew(initvalue As %String) As %Status
{
    If initvalue="" Quit $$$ERROR($$$GeneralError,"Must supply a name")
    Set ..Name=initvalue
    Quit $$$OK
}
```

%Library.RegisteredObject のサブクラスには、このメソッドをオーバーライドするオプションがあります。

29.15 %OnOpen()

このインスタンス・メソッドは、オブジェクトがオープンされる直前に %Open() または %OpenId() メソッドによって呼び出されます。関連エンティティと比較したインスタンスの状態を確認できます。

そのシグニチャは、以下のとおりです。

Class Member

```
Method %OnOpen() As %Status [ Private, ServerOnly = 1 ] {
    // body of method here...
}
```

このメソッドは、%Status コードを返します。失敗ステータスの場合、オブジェクトのオープンは中止されます。

%Library.Persistent および %SerialObject のサブクラスには、このメソッドをオーバーライドするオプションがあります。

29.16 %OnOpenFinally()

このクラス・メソッドは、すべての処理の完了後、呼び出し元に戻る直前に %Open() または %OpenId() メソッドによって呼び出されます。このメソッドは値を返さず、メソッドを呼び出したメソッドの結果を変更することはできません。

そのシグニチャは、以下のとおりです。

Class Member

```
ClassMethod %OnOpenFinally(oid As %ObjectIdentity, status As %Status) [ ServerOnly = 1 ]
{
    // body of method here...
}
```

各項目の内容は次のとおりです。

oid

オープンされるオブジェクトのオブジェクト識別子。

status

ユーザに報告されるステータス。このコールバック・メソッドでステータスを変更することはできません。

%Library.Persistent のサブクラスには、このメソッドをオーバーライドするオプションがあります。

29.17 %OnReload()

このインスタンス・メソッドは、oid によって指定されたオブジェクトが再読み込みされたことを通知するために、%Reload メソッドによって呼び出されます。oid によって識別されるオブジェクトが既にメモリ内にある場合、%Open は %Reload を呼び出します。このメソッドがエラーを返す場合は、オブジェクトが開かれていません。

そのシグニチャは、以下のとおりです。

Class Member

```
Method %OnReload() As %Status [ Private, ServerOnly = 1 ]
  {
    // body of method here...
  }
```

このメソッドは、%Status コードを返します。失敗ステータスの場合、ロールバック処理は中止されます。

%Library.Persistent のサブクラスには、このメソッドをオーバーライドするオプションがあります。

29.18 %OnRollBack()

既に SaveSet の一部として正常にシリアル化されているオブジェクトをロールバックするときに、InterSystems IRIS® データ・プラットフォームはこのインスタンス・メソッドを呼び出します ("オブジェクトの保存" を参照)。

永続オブジェクトまたはストリームの %Save() を呼び出したとき、またはシリアル・オブジェクトの %GetSwizzleObject() を呼び出したとき、システムは保存トランザクションを開始します。このトランザクションは、すべてのオブジェクトを SaveSet に含めます。%Save() が失敗すると (例えば、プロパティが検証に合格しなかったため)、InterSystems IRIS は、それまでに SaveSet の一部として正常にシリアル化しているオブジェクトをすべてロールバックします。つまり、それらのオブジェクトごとに、InterSystems IRIS は %RollBack() を呼び出すことになります。このメソッドは、%OnRollBack() を呼び出します。

InterSystems IRIS は、正常にシリアル化されていないオブジェクト (つまり、無効なオブジェクト) に対しては、このメソッドを呼び出しません。

%RollBack() は、そのオブジェクトのデータのディスク上の状態をトランザクション前の状態にリストアします。ただし、そのオブジェクトの ID 割り当てとは別に設定したプロパティのメモリ内の状態には効果がありません。(詳細は、"オブジェクトの保存" を参照してください。)メモリ内の変更を元に戻す場合は、%OnRollBack() でそのように実行します。

このメソッドのシグニチャは以下のとおりです。

Class Member

```
Method %OnRollBack() As %Status [ Private, ServerOnly = 1 ]
{
    // body of method here...
}
```

このメソッドは、%Status コードを返します。失敗ステータスの場合、ロールバック処理は中止されます。

%Library.Persistent のサブクラスには、このメソッドをオーバーライドするオプションがあります。

29.19 %OnSaveFinally()

このクラス・メソッドは、すべての処理の完了後、呼び出し元に戻る直前に %Save() メソッドによって呼び出されます。%Save() メソッドがトランザクションを開始していた場合、そのトランザクションはこのコールバックを呼び出す前に完了されます。このメソッドは値を返さず、メソッドを呼び出したメソッドの結果を変更することはできません。

そのシグニチャは、以下のとおりです。

Class Member

```
ClassMethod %OnSaveFinally(oref As %ObjectHandle, status As %Status) [ ServerOnly = 1 ]
{
    // body of method here...
}
```

各項目の内容は次のとおりです。

oid

保存されるオブジェクトのオブジェクト識別子。

status

ユーザに報告されるステータス。このコールバック・メソッドでステータスを変更することはできません。

%Library.Persistent のサブクラスには、このメソッドをオーバーライドするオプションがあります。

29.20 %OnValidateObject()

このインスタンス・メソッドは、すべての検証が発生した直後に %ValidateObject() メソッドによって呼び出されます。このメソッドを使用することで、あるプロパティに対する有効な値が、他のプロパティの値によって変化する場合など、カスタムの検証を実行できます。

そのシグニチャは、以下のとおりです。

Class Member

```
Method %OnValidateObject() As %Status [ Private, ServerOnly = 1 ]
{
    // body of method here...
}
```

このメソッドは、%Status コードを返します。失敗ステータスの場合、検証は失敗します。

%Library.RegisteredObject のサブクラスには、このメソッドをオーバーライドするオプションがあります。

29.21 %OnDetermineClass()

%OnDetermineClass() クラス・メソッドは、そのオブジェクトの最も適切なタイプのクラスを返します。(最適なタイプのクラスの詳細は、"%ClassName() および最も適切なタイプのクラス (MSTC)" を参照してください。)%OnDetermineClass() は、既定のストレージ・クラスによって実装されます。カスタム・ストレージまたは SQL ストレージを使用する場合、このメソッドの既定の実装はありませんが、メソッドの実装は可能です。

そのシグニチャは、以下のとおりです。

```
ClassMethod %OnDetermineClass(
        oid As %ObjectIdentity,
        ByRef class As %String)
    As %Status [ ServerOnly = 1 ]
```

各要素の内容は以下のとおりです。

oid

オブジェクトのオブジェクト識別子。

class

oid で特定するインスタンスの最も適切なタイプのクラス。オブジェクトがクラスのインスタンスであり、さらにそのクラスのどのサブクラスのインスタンスにもなっていない場合、そのクラスはそのオブジェクトの最も適切なタイプのクラスとなります。

返り値は、成功または失敗を示す状態値です。

%Library.SwizzleObject のサブクラスは、このメソッドをオーバーライドできます。

29.21.1 %OnDetermineClass() の呼び出し

%OnDetermineClass() は、以下の 2 つのいずれかの方法で呼び出すことができます。

```
Set status = ##class(APackage.AClass).%OnDetermineClass(myoid, .myclass)
Set status = myinstance.%OnDetermineClass(myoid, .myclass)
```

ここで、myoid は、最も適切なタイプのクラスを特定しようとしているオブジェクトであり、myclass は、特定されたクラスです。APackage.AClass はメソッドの呼び出し元のクラスであり、myinstance はメソッドの呼び出し元のインスタンスです。

この場合、メソッドは、myoid の最も適切なタイプのクラスを計算し、myclass をその値に等しくなるよう設定しています。myoid が現在のクラスのインスタンスではない場合、エラーが返されます。

%OnDetermineClass() を Sample.Person のサブクラスである Sample.Employee と共に使用する例を考えてみます。次の形の呼び出しがあり、

```
Set status = ##class(Sample.Employee).%OnDetermineClass(myoid, .class)
```

最も適切なタイプのクラスが Sample.Person であるオブジェクトを myoid が参照する場合、呼び出しはエラーを返します。

29.21.2 %OnDetermineClass() に対する呼び出し結果の例

MyPackage.Person クラスを MyPackage.Student クラスが拡張し、このクラスを MyPackage.GradStudent クラスが拡張するとします。以下に、%OnDetermineClass() を呼び出し、その最も適切なタイプのクラスが MyPackage.Student であるオブジェクトの OID を渡した結果を示します。

- `##class(MyPackage.Person).%OnDetermineClass(myOid,.myClass)`
  - 返り値：$$$OK
  - myClass は、MyPackage.Student に設定されます。
- `##class(MyPackage.Student).%OnDetermineClass(myOid,.myClass)`
  - 返り値：$$$OK
  - myClass は、MyPackage.Student に設定されます。
- `##class(MyPackage.GradStudent).%OnDetermineClass(myOid,.myClass)`
  - 返り値：エラー・ステータス
  - myClass は、"" に設定されます。

29.22 関連項目

- クラスの定義
- 登録オブジェクトを使用した作業
- 永続オブジェクトの概要

30

プロパティ・メソッドの使用とオーバーライド

ここでは、プロパティ・メソッドについて説明します。OREF を使用してオブジェクトのプロパティを操作するとき、プロパティ・メソッドは InterSystems IRIS® データ・プラットフォームによって使用される実際のメソッドです。これらのメソッドは、SQL 経由でデータにアクセスするときは使用されません。

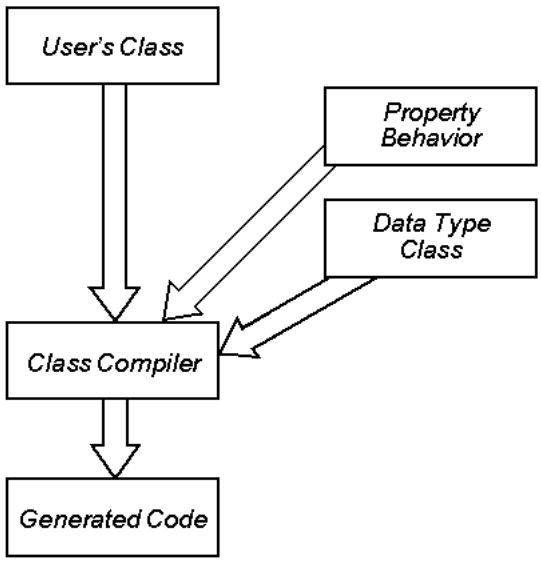
30.1 プロパティ・メソッドの概要

プロパティには、自動的に関連付けられた多くのメソッドがあります。これらのメソッドは、標準の継承を介して継承されません。その代わりに、各プロパティに対する一連のメソッドを作成するために、特別なプロパティの振る舞いメカニズムを使用します。

各プロパティは、2 つの場所からの一連のメソッドを継承します。

- Get()、Set()、および確認コードなど、組み込みの動作を提供する %Property クラス。
- プロパティによって使用されるデータ型クラス(該当する場合)。これらのメソッドの多くは、メソッド・ジェネレータです。

図 30-1: プロパティの振る舞い



プロパティの振る舞いクラスは、システム・クラスです。プロパティの振る舞いは、指定、または修正できません。

例えば、3 つのプロパティでクラス Person を定義する場合、以下のようになります。

Class Definition

```
Class MyApp.Person Extends %Persistent
{
Property Name As %String;
Property Age As %Integer;
Property DOB As %Date;
}
```

コンパイルされた Person クラスは、各プロパティに対して自動的に生成された一連のメソッドを持ちます。これらのメソッドは、プロパティと関連するデータ型クラスからも、システムの Property クラスからも継承されます。生成されたこれらのメソッド名は、プロパティ名に継承クラスからのメソッド名を連結した名前です。例えば、DOB プロパティに関連付けられたメソッドでは以下のようになります。

ObjectScript

```
 Set x = person.DOBIsValid(person.DOB)
 Write person.DOBLogicalToDisplay(person.DOB)
```

IsValid() はプロパティ・クラスのメソッドで、LogicalToDisplay() は %Date データ型クラスのメソッドです。

30.2 リテラル・プロパティのプロパティ・アクセサ

オブジェクト・プロパティを参照する InterSystems IRIS のドット構文は、値を取得および設定するアクセサ・メソッドのセットへのインタフェースです。それぞれの非計算プロパティについては、コードから oref.Prop (oref はオブジェクト、Prop はプロパティ) を参照すると、常にシステム提供の PropGet() メソッド、または PropSet() メソッドが呼び出されたかのように実行されます。以下に例を示します。

ObjectScript

```
 Set person.DOB = x
```

上記のコードは、以下のメソッドが呼び出されたかのように振る舞います。

ObjectScript

```
 Do person.DOBSet(x)
```

以下のコードは、

ObjectScript

```
 Write person.Name
```

以下のように振る舞います。

ObjectScript

```
 Write person.NameGet()
```

多くの場合、実際には PropGet() メソッドや PropSet() メソッドは存在しません。最善のパフォーマンスを得るために、単純なプロパティへのアクセスは InterSystems IRIS 仮想マシン内で直接実装されます。ただし、プロパティがオブジェクトタイプまたは多次元でない限り、特定のプロパティのために PropGet() メソッドと PropSet() メソッドを用意できます。これらのメソッドを定義すると、そのメソッドは実行時に、システムによって自動的に呼び出されます。以下のセクションでは、

これらのアクセッサ・メソッドを定義する方法を説明します。カスタム・メソッド内では、アプリケーションに必要な任意の特別な処理を実行できます。

スタジオの [**新規プロパティ・ウィザード**] の最後の画面には、カスタム Get() メソッド、Set() メソッド、またはその両方を生成するオプションが用意されています。これらのオプションを使用すると、スタジオは適切なシグニチャと共にスタブ・メソッドを定義します。

PropGet() メソッドおよび PropSet() メソッドを使用してオブジェクトのプロパティにアクセスするには、オブジェクトをメモリにロードする必要があります。一方、PropGetStored() メソッドを使用すると、保存されているオブジェクトのプロパティ値をディスクから直接取得できます。オブジェクト全体をメモリにロードする必要はありません。例えば、ID が 44 の個人の名前を書き込むには、以下のコードを使用できます。

ObjectScript

```
Write ##class(MyApp.Person).NameGetStored(44)
```

30.3 オブジェクト値プロパティのプロパティ・アクセサ

すべての参照プロパティに対しては、SetObject() (OID 値を使用する) メソッドと SetObjectId() (ID 値を使用する) メソッドがあります。例えば、特定の保存された Person オブジェクトを Car オブジェクトの持ち主として割り当てるには、以下のコードを使用します。

ObjectScript

```
Do car.OwnerSetObjectId(PersonId)
```

car は Car オブジェクトの OREF、PersonId は保存された Person オブジェクトの ID です。

GetObject() メソッドと GetObjectId() メソッドも使用できます。これらのメソッドは、それぞれ参照プロパティに関連付けられている OID または ID を取得します。まとめると、以下のメソッドがあります。

- GetObject() ー プロパティに関連付けられている OID を取得します。プロパティの名前が prop の場合、メソッド名は propGetObject() です。
- GetObjectId() ー プロパティに関連付けられている ID を取得します。プロパティの名前が prop の場合、メソッド名は propGetObjectId() です。
- SetObject() ー プロパティに関連付けられている OID を設定します。プロパティの名前が prop の場合、メソッド名は propSetObject() です。
- SetObjectId() ー プロパティに関連付けられている ID を設定します。プロパティの名前が prop の場合、メソッド名は propSetObjectId() です。

30.4 プロパティ・ゲッター・メソッドのオーバーライド

プロパティのゲッター・メソッドをオーバーライドするには、そのプロパティを含むクラスを変更して、以下のようなメソッドを追加します。

- PropertyName`Get` という名前にする必要があります。PropertyName は、対応するプロパティの名前です。
- 引数を取りません。
- 返りタイプは、プロパティのタイプと同じにする必要があります。

- プロパティの値を返す必要があります。
- このプロパティの値を参照するには、このメソッドで変数 i%PropertyName を使用する必要があります。この名前では大文字と小文字が区別されます。

  重要　特定のプロパティに対するこのゲッター・メソッド内では、そのプロパティの値を参照するために、..PropertyName 構文を使用してはいけません。そのようにすると、再帰的な参照の連続により <FRAMESTACK> エラーを招くことになります。ただし、その他のプロパティを参照する場合には ..PropertyName を使用できます。このようにしても、再帰が発生することがないためです。

  変数 i%PropertyName はインスタンス変数です。インスタンス変数の詳細は、"i%PropertyName" を参照してください。

注釈　オブジェクトタイプ・プロパティまたは多次元プロパティのアクセサ・メソッドに対するオーバーライドはサポートされていません。また、メソッド名の最大長が 220 文字なので、長さが 218 文字、219 文字、または 220 文字の名前を持つプロパティのアクセサ・メソッドは作成できません。

以下に、HasValue という名前のプロパティ (タイプ %Boolean) のセッター・メソッドの例を示します。

Class Member

```
Method HasValueGet() As %Boolean
{
  If ((i%NodeType="element")||(i%NodeType="")) Quit 0
  Quit 1
}
```

30.5 プロパティ・セッター・メソッドのオーバーライド

プロパティのセッター・メソッドをオーバーライドするには、そのプロパティを含むクラスを変更して、以下のようなメソッドを追加します。

- PropertyNameSet という名前にする必要があります。PropertyName は、対応するプロパティの名前です。
- 1 つの引数を受け取ります。この引数にはプロパティの値を格納します。

  具体的には、SET コマンドで指定された値です (プロパティが設定されている場合)。
- %Status 値を返す必要があります。
- このプロパティの値を設定するには、このメソッドで変数 i%PropertyName を設定する必要があります。この名前では大文字と小文字が区別されます。

  重要　特定のプロパティに対するこのセッター・メソッド内では、そのプロパティの値を参照するために、..PropertyName 構文を使用してはいけません。そのようにすると、再帰的な参照の連続により <FRAMESTACK> エラーを招くことになります。ただし、その他のプロパティを参照する場合には ..PropertyName を使用できます。このようにしても、再帰が発生することがないためです。

  変数 i%PropertyName はインスタンス変数です。インスタンス変数の詳細は、"i%PropertyName" を参照してください。

注釈　オブジェクトタイプ・プロパティまたは多次元プロパティのアクセサ・メソッドに対するオーバーライドはサポートされていません。また、メソッド名の最大長が 220 文字なので、長さが 218 文字、219 文字、または 220 文字の名前を持つプロパティのアクセサ・メソッドは作成できません。

例えば、MyProp のタイプは %String だとします。以下のセッター・メソッドを定義できます。

Class Member

```
Method MyPropSet(value as %String) As %Status
{
    if i%MyProp="abc" {
        set i%MyProp="corrected value"
    }
    quit $$$OK
}
```

以下に示す別の例は、DefaultXmlns という名前のプロパティのセッター・メソッド (タイプ %String) です。

Class Member

```
Method DefaultXmlnsSet(value As %String) As %Status
{
    set i%DefaultXmlns = value
    If ..Namespaces'="" Set ..Namespaces.DefaultXmlns=value
    quit $$$OK
}
```

この例では、同じオブジェクトの Namespaces プロパティを参照するために、..PropertyName 構文を使用しています。この使用法は、再帰が発生しないため、エラーになりません。

30.6 カスタム・アクセサ・メソッドによるオブジェクト値プロパティの定義

前述したように、オブジェクトタイプ・プロパティのアクセサ・メソッドに対するオーバーライドはサポートされていません。オブジェクト値を保持するプロパティを定義する必要があり、カスタム・アクセサ・メソッドを定義する必要もある場合は、%RawString. タイプでプロパティを定義します。これは、オブジェクト・クラスではなく汎用クラスであるため、このプロパティのアクセサ・メソッドはオーバーライドできます。このプロパティを使用する場合は、目的のクラスのインスタンスと等しくなるように設定します。

例えば、以下のクラスにはプロパティ Zip が含まれます。その公式の型は %RawString です。プロパティの説明は、そのプロパティが Sample.USZipCode のインスタンスであることを示します。このクラスでは、ZipGet() および ZipSet() プロパティ・メソッドも定義されます。

Class Definition

```
Class PropMethods.Demo Extends %Persistent
{

/// Timestamp for viewing Zip
Property LastTimeZipViewed As %TimeStamp;

/// Timestamp for changing Zip
Property LastTimeZipChanged As %TimeStamp;

/// When setting this property, set it equal to instance of Sample.USZipCode.
/// The type is %RawString rather than Sample.USZipCode, so that it's possible
/// to override ZipGet() and ZipSet().
Property Zip As %RawString;

Method ZipGet() As %RawString [ ServerOnly = 1 ]
{
    // get id, swizzle referenced object
    set id = i%Zip
    if (id '= "") {
        set zip = ##class(Sample.USZipCode).%OpenId(id)
        set ..LastTimeZipViewed = $zdt($zts)
    }
    else {
```

```
        set zip = ""
    }
    return zip
}

Method ZipSet(zip As %RawString) As %Status [ ServerOnly = 1 ]
{
    // set i% for new zip
    if ($isobject(zip) && zip.%IsA("Sample.USZipCode")) {
        set id = zip.%Id()
        set i%Zip = id
        set ..LastTimeZipChanged = $zdt($zts)
    }
    else {
        set i%Zip = ""
    }
    quit $$$OK
}

}
```

以下のターミナル・セッションは、このクラスの使用法を示しています。

```
SAMPLES>set demo=##class(PropMethods.Demo).%New()

SAMPLES>write demo.LastTimeZipChanged

SAMPLES>set zip=##class(Sample.USZipCode).%OpenId(10001)

SAMPLES>set demo.Zip=zip

SAMPLES>w demo.LastTimeZipChanged
10/14/2015 19:21:08
```

31
データ型クラスの定義

ここでは、データ型クラスがどのように機能するかについて説明します。また、それらのクラスを定義する方法も説明します。

31.1 データ型クラスの概要

データ型クラスの目的は、オブジェクト・クラスのリテラル・プロパティのタイプとして使用されることです。データ型クラスには、以下の機能があります。

- SQL 論理操作、クライアント・データ型、および変換情報の提供による、SQL およびクライアントの相互運用性
- データ型クラス・パラメータを使用して拡張、またはカスタマイズできるリテラル・データ値の妥当性検証
- これらは、(ディスク上に) 格納された、論理 (メモリ内) 形式、および表示形式に対するリテラル・データの変換を管理します。

データ型クラスを使用した、コンパイラによるプロパティのコード生成の詳細は、"プロパティ・メソッド" を参照してください。

データ型は他のクラスとは異なる点がいくつかあります。

- 独立してインスタンス化されたり、格納することはできません。
- プロパティを持つことはできません。
- 以下に示すような特定のメソッド (データ型インタフェース) に対応します。

内部の詳細をいくらかでも知っていると役に立つので、このセクションでは、データ型クラスがどのように機能するかについて、手短に説明します。

前述のように、データ型クラスの目的はプロパティのタイプとして (特に重要なオブジェクト・クラスの 1 つを拡張するクラス内で) 使用されることです。3 つのプロパティを持っているオブジェクト・クラスの例を以下に示します。各プロパティは、プロパティのタイプとしてデータ型クラスを使用します。

Class Definition

```
Class Datatypes.Container Extends %RegisteredObject
{

Property P1 As %String;

Property P2 As %Integer;

Property P3 As %Boolean;

}
```

31.1.1 プロパティ・メソッド

クラスにリテラル・プロパティを追加してクラスをコンパイルすると、InterSystems IRIS® データ・プラットフォームによって、そのクラスにプロパティ・メソッドが追加されます。参考として、プロパティを含むクラスをコンテナ・クラスという用語を使用して表します。プロパティ・メソッドは、コンテナ・クラスがそれらのプロパティのデータを処理する方法を制御します。このシステムは、以下のように実行されます。

- 各データ型クラスは、メソッドのセット（具体的にはメソッド・ジェネレータ）を提供します。このセットは、メソッドを使用するクラスをコンパイルするときに、InterSystems IRIS によって使用されます。メソッド・ジェネレータは、独自の実行時コードを生成するメソッドです。

  ここに示す例では、**Datatypes.Container** をコンパイルするとき、コンパイラは **%String**、**%Integer**、および **%Boolean** データ型クラスのメソッド・ジェネレータを使用します。これらのメソッド・ジェネレータは各プロパティのメソッドを作成し、作成したメソッドをコンテナ・クラスに追加します。前述したように、これらのメソッドはプロパティ・メソッドと呼ばれます。それらの名前は、適用先のプロパティの名前で始まります。例えば、**P1** プロパティの場合、コンパイラは P1IsValid()、P1Normalize()、P1LogicalToDisplay()、P1DisplayToLogical() などのメソッドを生成します。

- コンテナ・クラスは処理中の適切な段階で、自動的にプロパティ・メソッドを呼び出します。例えば、前述のクラスのインスタンスの %ValidateObject() インスタンス・メソッドを呼び出すとき、メソッドは P1IsValid()、P2IsValid()、および P3IsValid() を順番に呼び出します。つまり、メソッドは各プロパティの IsValid() メソッドを呼び出します。もう 1 つ例を挙げると、コンテナ・クラスが永続で、InterSystems SQL を使用して関連テーブルのすべてのフィールドにアクセスする場合、SQL の実行時モードが ODBC なら、InterSystems IRIS は各プロパティの LogicalToOdbc() メソッドを呼び出します。これにより、クエリは ODBC 形式で結果を返します。

プロパティ・メソッドはクラス定義には表示されないので、注意が必要です。

重要　これらのメソッドはすべてクラス・メソッドで、そのすべてに、妥当性のチェック、正規化、変換を行う値などの引数が必要です。

31.1.2 データ形式

プロパティ・メソッドの多くは、データをある形式から別の形式に変換します。例えば、データを人が読める形式で表示するときや、ODBC 経由でデータにアクセスするときなどです。形式は以下のとおりです。

- Display ー データの入力および表示形式。例えば、“April 3, 1998” や “23 November, 1977.” のフォームの日付。
- Logical ー データのメモリ内形式（オペレーションが実行される形式）。日付には上記で説明したような表示形式を持ちますが、論理形式は整数です。上記のサンプルの日付では、論理形式のそれらの値は、それぞれ 57436 と 50000 です。
- Storage ー ディスク上のデータ形式（データベースに格納されるデータの形式）。一般的に、これは論理形式と同じものです。
- ODBC ー データを ODBC または JDBC 経由で表すことのできる形式。この形式は、データが ODBC/SQL に公開されるときに使用されます。使用可能な形式は ODBC で定義したものに対応します。

- XSD – SOAP でエンコードされた形式。この形式は、XML へのエクスポートまたは XML からのインポートの際に使用します。これは、XML 対応のクラスにのみ適用されます。

31.1.3 データ型クラスのパラメータ

クラス・パラメータは、データ型クラスで使用されるときに、特別な振る舞いをします。データ型クラスでは、クラス・パラメータはデータ型に基づいてプロパティの動作をカスタマイズする方法を提供するために使用されます。

例えば、%Integer データ型クラスは、クラス・パラメータ MAXVAL を持ち、%Integer 型のプロパティに対して最大有効値を指定します。プロパティ NumKids を持つクラスを、以下のように定義した場合、

Class Member

```
Property NumKids As %Integer(MAXVAL=10);
```

これは、%Integer クラスの MAXVAL パラメータを、NumKids プロパティに対して 10 に設定することを指定します。

内部的には、標準データ型クラスに対するすべての検証メソッドは、メソッド・ジェネレータとして実装され、さまざまなクラス・パラメータを使用して、これらの検証メソッドの生成を制御します。この例では、NumKids の値が 10 を超えるか否かをテストする NumKidsIsValid() メソッドに対するコンテンツをこのプロパティ定義で生成します。クラス・パラメータを使用せずに、この機能を作成するには IntegerLessThanTen クラスの定義が必要です。

31.2 データ型クラスの定義

データ型クラスを簡単に定義するには、まず自分の要求に近い既存のデータ型クラスを特定します。このクラスのサブクラスを作成します。サブクラスで以下を実行します。

- キーワード SqlCategory、ClientDataType、および OdbcType に適切な値を指定します。
- 必要に応じて、クラス・パラメータをオーバーライドします。例えば、MAXLEN パラメータをオーバーライドして、プロパティの長さを無制限にすることもできます。

  必要に応じて、ユーザ独自のクラス・パラメータを追加します。
- 必要に応じて、データ型クラスのメソッドをオーバーライドします。必要に応じて、実装時にこのクラスのパラメータを参照します。

データ型クラスが既存のデータ型クラスに基づいていない場合は、必ずクラス定義に [ClassType=datatype] を追加してください。クラスが別のデータ型クラスに基づく場合、この宣言は必要ありません。

注釈　データ型クラスにプロパティを含めることはできないため、データ型クラスを定義する際に、%Persistent や %RegisteredObject などのオブジェクト・クラスを拡張しないでください。

31.3 データ型クラスのクラス・メソッドの定義

要求の内容に応じて、以下に示すデータ型クラスのクラス・メソッドの一部または全部を定義します。

- IsValid() ー 必要に応じてプロパティ・パラメータを使用して、プロパティのデータの検証を行います。前述したように、オブジェクト・クラスの %ValidateObject() インスタンス・メソッドは、各プロパティの IsValid() メソッドを呼び出します。このメソッドには、以下のシグニチャがあります。

  ```
  ClassMethod IsValid(%val) As %Status
  ```

  %val は検証される値です。このメソッドは、値が無効の場合にはエラー・ステータスを返し、それ以外の場合には $$$OK を返します。

  注釈　NULL 値に対しては検証ロジックを呼び出さないのが InterSystems IRIS での標準的なやり方です。

- Normalize() ー プロパティのデータを標準フォームまたは標準形式に変換します。オブジェクト・クラスの %NormalizeObject() インスタンス・メソッドは、各プロパティの Normalize() メソッドを呼び出します。このメソッドには、以下のシグニチャがあります。

  ```
  ClassMethod Normalize(%val) As Type
  ```

  %val は検証される値、Type は適切なタイプのクラスです。

- DisplayToLogical() ー 表示値を論理値に変換します。(形式の詳細は、"データ形式" を参照してください。)このメソッドには、以下のシグニチャがあります。

  ```
  ClassMethod DisplayToLogical(%val) As Type
  ```

  %val は変換される値、Type は適切なタイプのクラスです。

  その他の形式変換メソッドには、同様の一般的な形式があります。

- LogicalToDisplay() ー 論理値を表示値に変換します。
- LogicalToOdbc() ー 論理値を ODBC 値に変換します。

  ODBC の値は、データ型クラスの OdbcType クラス・キーワードで指定された ODBC のタイプと一致している必要があります。

- LogicalToStorage() ー 論理値をストレージ値に変換します。
- LogicalToXSD() ー 論理値を SOAP でエンコードされた適切な値に変換します。
- OdbcToLogical() ー ODBC 値を論理値に変換します。
- StorageToLogical() ー データベースのストレージ値を論理値に変換します。
- XSDToLogical() ー SOAP でエンコードされた値を論理値に変換します。

重要　これらのメソッドはすべてクラス・メソッドで、そのすべては、妥当性のチェック、正規化、変換を行う値などの引数を受け入れる必要があります。

データ型クラスに DISPLAYLIST パラメータと VALUELIST パラメータが含まれる場合、これらのメソッドは、最初にこれらのクラスのパラメータが存在するかを確認し、存在する場合は、これらのリストを処理するコードを含める必要があります。このロジックは、その他のメソッドと類似しています。

ほとんどの場合、これらのメソッドの多くはメソッド・ジェネレータです。

注釈　データの書式設定と変換のメソッドには、埋め込み SQL を含めることはできません。このロジックで埋め込み SQL を呼び出す必要がある場合は、埋め込み SQL を別個のルーチンに配置すると、このルーチンをメソッドで呼び出せるようになります。

31.4 データ型クラスのインスタンス・メソッドの定義

データ型クラスには、インスタンス・メソッドを追加することもできます。これらのメソッドは、プロパティの現在の値を含む変数 %val を使用できます。コンパイラはこれらを使用して、データ型クラスを使用するクラスの関連プロパティ・メソッドを生成します。例えば、以下のデータ型クラスの例を考えてみます。

Class Definition

```
Class Datatypes.MyDate Extends %Date
{

Method ToMyDate() As %String [ CodeMode = expression ]
{
$ZDate(%val,4)
}
}
```

以下に示す、別のクラスがあるとします。

Class Definition

```
Class Datatypes.Container Extends %Persistent
{

Property DOB As Datatypes.MyDate;

/// additional class members
}
```

これらのクラスをコンパイルすると、InterSystems IRIS によって DOBToMyDate() インスタンス・メソッドがコンテナ・クラスに追加されます。このメソッドは、後でコンテナ・クラスのインスタンスを作成するときに呼び出すことができます。以下に例を示します。

```
TESTNAMESPACE>set instance=##class(Datatypes.Container).%New()

TESTNAMESPACE>set instance.DOB=+$H

TESTNAMESPACE>write instance.DOBToMyDate()
30/10/2014
```

31.5 関連項目

- クラスの定義
- 登録オブジェクトを使用した作業
- リテラル・プロパティの定義と使用
- プロパティ・メソッド

32
動的ディスパッチの実装

ここでは、InterSystems IRIS® データ・プラットフォームのクラスの動的ディスパッチについて説明します。

32.1 動的ディスパッチの概要

InterSystems IRIS クラスには、動的ディスパッチに対するサポートを含めることができます。動的ディスパッチが使用中で、プログラムがクラス定義の一部でないプロパティまたはメソッドを参照している場合、ディスパッチ・メソッドと呼ばれるメソッドが呼び出され、未定義メソッドまたは未定義プロパティを解決しようと試みます。例えば動的ディスパッチを使用すると、定義されていないプロパティ値を返したり、実装されていないメソッドのためにメソッドを呼び出すことができます。ディスパッチ先はクラス記述子に表示されないという点で動的であり、実行時まで解決されません。

InterSystems IRIS は、実装可能なディスパッチ・メソッドを多数作成します。各メソッドは、さまざまな状況で見つからない要素を解決しようと試みます。

ディスパッチ・メソッドを実装すると、次のような効果があります。

- アプリケーションの実行時、コンパイル済みクラスの一部でない要素が見つかると、ディスパッチ・メソッドを呼び出してその要素を解決しようと試みます。
- この解決には、そのクラスを使用中のアプリケーション・コードは使用されません。InterSystems IRIS が自動的にディスパッチ・メソッドの存在をチェックし、メソッドが存在すればそのディスパッチ・メソッドを呼び出します。

32.2 動的ディスパッチを実装するメソッドのコンテンツ

アプリケーション開発者は、ディスパッチ・メソッドのコンテンツを制御することができます。そのクラスが解決しようと試みるメソッドまたはプロパティを実装するのに必要なものを、ディスパッチ・メソッドのコンテンツ内のコードに指定できます。

動的ディスパッチのコードには、同一エクステント、同一パッケージ、同一データベース、同一ファイルシステムや、その他のあらゆる基準からメソッドを特定するような処理を含めることができます。ディスパッチ・メソッドが一般的なケースを提供する場合は、この一般的な解決が含まれる継続した処理が記録されるように、メソッドでこの操作を記録するログも作成することをお勧めします。

例えば次のような %DispatchClassMethod() を実装すると、アプリケーション・ユーザがどのような操作を行う際にもメソッドを呼び出せるようになります。

Class Member

```
ClassMethod %DispatchClassMethod(Class As %String, Method As %String, args...)
{
    WRITE "The application has attempted to invoke the following method: ",!,!
    WRITE Class,".",Method,!,!
    WRITE "This method does not exist.",!
    WRITE "Enter the name of the class and method to call",!
    WRITE "or press Enter for both to exit the application.",!,!


    READ "Class name (in the form 'Package.Class'): ",ClassName,!
    READ "Method name: ",MethodName,!

    IF ClassName = "" && MethodName = "" {
        // return a null string to the caller if a return value is expected
        QUIT:$QUIT "" QUIT
    } ELSE {
        // checking $QUIT ensures that a value is returned
        // if and only if it is expected
        IF $QUIT {
            QUIT $CLASSMETHOD(ClassName, MethodName, args...)
        } ELSE {
            DO $CLASSMETHOD(ClassName, MethodName, args...)
            QUIT
        }
    }
}
```

アプリケーションの全クラスのセカンダリ・スーパークラスであるクラスにこのメソッドを含めておくと、存在しないクラス・メソッドに対してアプリケーション全体の呼び出し処理を構築できます。

32.2.1 返り値

どのディスパッチ・メソッドにも、指定の返り値はありません。これは、各ディスパッチ・メソッドは、最初にディスパッチの必要性が生じた呼び出しと同じタイプの出力を提供する必要があるためです。

ディスパッチ・メソッドがメソッドまたはプロパティを解決できない場合は、$SYSTEM.Process.ThrowError() を使用して <METHOD DOES NOT EXIST> エラーまたは <PROPERTY DOES NOT EXIST> エラーなどを返すことができます。

32.3 動的ディスパッチ・メソッド

不明メソッドと不明プロパティを解決するのに実装できるメソッドは、次のとおりです。

- %DispatchMethod()
- %DispatchClassMethod()
- %DispatchGetProperty()
- %DispatchSetProperty()
- %DispatchSetMultidimProperty()

32.3.1 %DispatchMethod()

このメソッドは、不明なメソッド呼び出しを実装します。構文は、以下のとおりです。

```
Method %DispatchMethod(Method As %String, Args...)
```

ここで、最初の引数は参照メソッド名を指し、次の引数は元のメソッドに渡される全引数を保持した配列を指します。引数の数とそのタイプは解決対象となるメソッドによって異なるため、%DispatchMethod() 内のコードで正しく処理する必要が

あります (クラス・コンパイラでは、タイプに関する想定がまったくできません)。Args... 構文でこれを柔軟に処理することができます。

%DispatchMethod() は、そのクラスに関連付けられている不明なインスタンス・メソッドを解決しようと試みますが、返り値が指定されていないため、解決できた場合は、呼び出し元が返り値を予期するかどうかにかかわらず、解決対象のメソッドによって決定されたタイプの値が返されます。

%DispatchMethod() を使用すると、不明な多次元プロパティ参照も解決できます。つまり、プロパティの値も取得できます。ただし、多次元プロパティ参照は動的ディスパッチでのみサポートされています。$DATA, $ORDER、および $QUERY はサポートされていません。また、変数リストを用いる SET コマンドもサポートされていません。

32.3.2 %DispatchClassMethod()

このメソッドは、不明なクラス・メソッド呼び出しを実装します。構文は、以下のとおりです。

```
ClassMethod %DispatchClassMethod(Class As %String, Method As %String, Args...)
```

最初の 2 つの引数は参照クラス名と参照メソッド名を指します。3 番目の引数は、元のメソッドに渡される引数全体を保持する配列を指します。引数の数とそのタイプは解決対象となるメソッドによって異なるため、%DispatchClassMethod() 内のコードで正しく処理する必要があります (クラス・コンパイラでは、タイプに関する想定がまったくできません)。Args... 構文でこれを柔軟に処理することができます。

%DispatchClassMethod() は、そのクラスに関連付けられている不明なインスタンス・メソッドを解決しようと試みますが、返り値が指定されていないため、解決できた場合は、呼び出し元が返り値を予期するかどうかにかかわらず、解決対象のメソッドによって決定されたタイプの値が返されます。

32.3.3 %DispatchGetProperty()

このメソッドは、不明プロパティの値を取得します。構文は、以下のとおりです。

```
Method %DispatchGetProperty(Property As %String)
```

ここで、引数は参照されるプロパティを指します。%DispatchGetProperty() は、そのクラスに関連付けられている不明なインスタンス・メソッドを解決しようと試みますが、返り値が指定されていないため、解決できた場合は、解決対象のプロパティ・タイプの値が返されます。

32.3.4 %DispatchSetProperty()

このメソッドは、不明プロパティの値を設定します。構文は、以下のとおりです。

```
Method %DispatchSetProperty(Property As %String, Value)
```

この引数には、参照先のプロパティ名とそのプロパティに設定する値を指定します。

32.3.5 %DispatchSetMultidimProperty()

このメソッドは、不明な多次元プロパティの値を設定します。構文は、以下のとおりです。

```
Method %DispatchSetMultidimProperty(Property As %String, Value, Subs...)
```

最初の 2 つの引数には、参照先のプロパティ名とそのプロパティに設定する値を指定します。3 番目の引数 Subs は、添え字の値を含む配列を指します。 Subs には添え字の数を指定する整数値が、Subs(1) には最初の添え字の値が、Subs(2) には 2 番目の添え字の値が入り、以下同様に続きます。添え字を指定しないと、Subs が未定義になります。

多次元プロパティ参照は、動的ディスパッチでのみサポートされています。$DATA, $ORDER、および $QUERY はサポートされていません。また、変数リストを用いる SET コマンドもサポートされていません。

注釈　%DispatchGetMultidimProperty() ディスパッチ・メソッドはないことに注意してください。これは、多次元プロパティ参照がメソッド呼び出しと同じためです。したがって、この種の参照は %DispatchMethod() を呼び出すため、メソッド名と多次元プロパティ名を区別するためのコードを含めておく必要があります。

32.4 関連項目

- クラスの定義

A

オブジェクト特有の ObjectScript の機能

ここでは、クラスおよびオブジェクトの操作に特有の ObjectScript の機能についてまとめます。

A.1 相対ドット構文 (..)

相対ドット構文 (..) は、現在のコンテキストのメソッドやプロパティを参照するメカニズムを提供します。インスタンス・メソッドまたはプロパティのコンテキストは、現在のインスタンスです。クラス・メソッドのコンテキストは、メソッドが実装されているクラスです。クラス・メソッドで相対ドット構文を使用してプロパティやインスタンス・メソッドを参照することはできません。これらを参照するにはインスタンス・コンテキストが必要です。

例えば、**%Integer** タイプの Bricks プロパティがあるとします。

Class Member

```
Property Bricks As %Integer;
```

CountBricks() メソッドは、相対ドット構文を使用して Bricks を参照できます。

Class Member

```
Method CountBricks()
{
    Write "There are ",..Bricks," bricks.",!
}
```

同様に、WallCheck() メソッドは CountBricks() および Bricks を参照できます。

Class Member

```
Method WallCheck()
{
    Do ..CountBricks()
    If ..Bricks < 100 {
        Write "Your wall will be small."
    }
}
```

A.2 ##class 構文

##class 構文により、以下が可能になります。

- クラスの既存のインスタンスがないとき、または開かれているインスタンスがないときにクラス・メソッドを呼び出します。
- 別のクラスからのメソッドとして、あるクラスからメソッドをキャストします。
- クラス・パラメータへのアクセス

注釈　##class は、大文字と小文字を区別しません。

A.2.1 クラス・メソッドの呼び出し

クラス・メソッドを呼び出すには、以下のどちらかの構文を使用します。

```
>Do ##class(Package.Class).Method(Args)
>Set localname = ##class(Package.Class).Method(Args)
```

以下のように、式の一部に ##class を使用することもできます。

ObjectScript

```
 Write ##class(Class).Method(args)*2
```

変数を、返り値と同じ値に設定する必要はありません。

新規のインスタンスの生成において、この構文を頻繁に使用します。

```
>Set LocalInstance = ##class(Package.Class).%New()
```

A.2.2 メソッドのキャスト

あるクラスのメソッドを、別のクラスのメソッドとしてキャストするには、以下のどちらかの構文を使用します。

```
>Do ##class(Package.Class1)Class2Instance.Method(Args)
>Set localname = ##class(Package.Class1)Class2Instance.Method(Args)
```

クラス・メソッドとインスタンス・メソッドの両方ともキャストできます。

例えば、2 つのクラス MyClass.Up と MyClass.Down が、両方とも Go() メソッドを持つとします。MyClass.Up の場合は、このメソッドが以下のようになります。

Class Member

```
Method Go()
{
    Write "Go up.",!
}
```

MyClass.Down の場合は、Go() メソッドが以下のようになります。

Class Member

```
Method Go()
{
    Write "Go down.",!
}
```

ユーザは、MyClass.Up のインスタンスを生成することができ、これを使用して MyClass.Down.Go メソッドを呼び出します。

```
>Set LocalInstance = ##class(MyClass.Up).%New()

>Do ##class(MyClass.Down)LocalInstance.Go()
Go down.
```

以下のように、式の一部に ##class を使用することもできます。

ObjectScript

```
Write ##class(Class).Method(args)*2
```

変数を、返り値と同じ値に設定する必要はありません。

さらに一般的なメソッド参照方法として、インスタンス・メソッドを参照する $METHOD 関数およびクラス・メソッドを参照する $CLASSMETHOD 関数があります。"$CLASSNAME およびその他の動的アクセス関数" を参照してください。これらは、プログラム的にパッケージ、クラス、メソッドを参照するメカニズムを提供します。

A.2.3 クラス・パラメータへのアクセス

クラス・パラメータにアクセスするには、以下の式を使用します。

```
##class(Package.Class).#PARMNAME
```

Package.Class はクラスの名前、PARMNAME はパラメータの名前です。以下に例を示します。

ObjectScript

```
w ##class(%XML.Adaptor).#XMLENABLED
```

この例では、XML アダプタによって生成されたメソッドが XML 対応かどうかが表示されます。既定の設定は 1 です。

$PARAMETER 関数も使用できます。この関数については、"$CLASSNAME およびその他の動的アクセス関数" を参照してください。

A.3 $this 構文

$this 変数は、現在のインスタンスの OREF へのハンドルを提供します。例えば、現在のインスタンスを別のクラスに渡したり、別のクラスが現在のインスタンスのプロパティやメソッドを参照するために使用します。インスタンスが、そのインスタンスのプロパティやメソッドを参照する場合は、相対ドット構文のほうが高速であるために推奨されています。

注釈　$this は大文字と小文字を区別しません。このため、$this、$This、$THIS、およびその他の変異形は同じ値を持ちます。

例えば、Accounting.Order クラスと Accounting.Utils クラスを持つアプリケーションがあるとします。Accounting.Order.CalcTax() メソッドは、Accounting.Utils.GetTaxRate() メソッドおよび Accounting.Utils.GetTaxableSubtotal() メソッドを呼び出し、現在のインスタンスの "city" と "state" の値を GetTaxRate() メソッドに渡し、注文された品目と関連する税金関係の情報のリストを GetTaxableSubtotal() に渡します。CalcTax() は、返された値を使用して、注文に対する消費税を算出します。従って、コードは以下のようになります。

Class Member

```
Method CalcTax() As %Numeric
{
    Set TaxRate = ##Class(Accounting.Utils).GetTaxRate($this)
    Write "The tax rate for ",..City,", ",..State," is ",TaxRate*100,"%",!
    Set TaxableSubtotal = ##class(Accounting.Utils).GetTaxableSubTotal($this)
    Write "The taxable subtotal for this order is $",TaxableSubtotal,!
    Set Tax = TaxableSubtotal * TaxRate
    Write "The tax for this order is $",Tax,!
}
```

メソッドの最初の行では、##Class 構文 (前述) を使用して、クラスの別のメソッドを呼び出しています。ここでは $this 構文を使用して、そのメソッドに現在のオブジェクトを渡します。メソッドの 2 行目では、相対ドット構文を使用して、City プロパティと State プロパティの値を取得します。3 行目の動作は、1 行目と同様です。

Accounting.Utils クラスの GetTaxRate() メソッドは、渡されたインスタンスのハンドルを使用して、さまざまなプロパティへのハンドルを (値を取得したり設定したりするために) 取得できます。

Class Member

```
ClassMethod GetTaxRate(OrderBeingProcessed As Accounting.Order) As %Numeric
{
    Set LocalCity = OrderBeingProcessed.City
    Set LocalState = OrderBeingProcessed.State
    // code to determine tax rate based on location and set
    // the value of OrderBeingProcessed.TaxRate accordingly
    Quit OrderBeingProcessed.TaxRate
}
```

GetTaxableSubtotal() もインスタンスのハンドルを使用し、そのプロパティを確認して TaxableSubtotal プロパティの値を設定します。

したがって、Accounting.Order クラスの MyOrder インスタンスに対して CalcTax() メソッドを呼び出すと、ターミナルに以下のような出力が表示されます。

```
>Do MyOrder.CalcTax()
The tax rate for Cambridge, MA is 5%
The taxable subtotal for this order is $79.82
The tax for this order is $3.99
```

A.4 ##super 構文

サブクラス・メソッドは、スーパークラス・メソッドをオーバーライドするとします。サブクラス・メソッド内から、オーバーライドされたスーパークラス・メソッドを呼び出すには ##super() 構文を使用できます。

注釈　##super は、大文字と小文字を区別しません。

例えば、MyClass.Down クラスが MyClass.Up のサブクラスで、Simple クラス・メソッドをオーバーライドするとします。MyClass.Up.Simple() のコードが以下のとおりで、

Class Member

```
ClassMethod Simple()
{
    Write "Superclass.",!
}
```

MyClass.Down.Simple() のコードが以下のとおりである場合、

Class Member

```
ClassMethod Simple()
{
    Write "Subclass.",!
    Do ##super()
}
```

サブクラス・メソッド MyClass.Down.Simple() の出力は、以下のようになります。

```
>Do ##Class(MyClass.Down).Simple()
Subclass.
Superclass.
>
```

さらに一般的なメソッド参照方法として、インスタンス・メソッドを参照する $METHOD 関数およびクラス・メソッドを参照する $CLASSMETHOD 関数があります。"$CLASSNAME およびその他の動的アクセス関数" を参照してください。これらは、プログラム的にパッケージ、クラス、メソッドを参照するメカニズムを提供します。

A.4.1 ##super が作用する呼び出し

##super は、現在のメソッド・コールにのみ作用します。そのメソッドで他の呼び出しを実行する場合、それらの呼び出しは、スーパークラスに対してではなく、現在のオブジェクトまたはクラスに対して実行されます。例えば、以下のように、MyClass.Up に MyName() メソッドと CallMyName() メソッドがあるとします。

```
Class MyClass.Up Extends %Persistent
{

ClassMethod CallMyName()
{
    Do ..MyName()
}

ClassMethod MyName()
{
    Write "Called from MyClass.Up",!
}

}
```

また、以下のように、MyClass.Down で上記のメソッドをオーバーライドするとします。

```
Class MyClass.Down Extends MyClass.Up
{

ClassMethod CallMyName()
{
    Do ##super()
}

ClassMethod MyName()
{
    Write "Called from MyClass.Down",!
}

}
```

ここで、CallMyName() メソッドを呼び出すと、以下のような結果が返されます。

```
USER>d ##class(MyClass.Up).CallMyName()
Called from MyClass.Up

USER>d ##class(MyClass.Down).CallMyName()
Called from MyClass.Down
```

MyClass.Down.CallMyName() の出力は、MyClass.Up.CallMyName() とは異なっています。これは、MyClass.Down.CallMyName() の CallMyName() メソッドには ##super が含まれており、MyClass.Up.CallMyName() メソッドを呼び出した後、キャストされない MyClass.Down.MyName() メソッドを呼び出しているからです。

A.4.2 ##super と メソッドの引数

##super は、引数を許可するメソッドでも機能します。サブクラス・メソッドで引数の既定値を指定していない場合は、そのメソッドがスーパークラスへの参照で引数を渡していることを確認します。

例えば、スーパークラス (MyClass.Up.SelfAdd()) のメソッドのコードが以下のとおりだとします。

Class Member

```
ClassMethod SelfAdd(Arg As %Integer)
{
    Write Arg,!
    Write Arg + Arg
}
```

出力は、以下のようになります。

```
>Do ##Class(MyClass.Up).SelfAdd(2)
2
4
>
```

サブクラス (MyClass.Down.SelfAdd()) のメソッドでは、##super を使用し、引数を参照で渡します。

Class Member

```
ClassMethod SelfAdd(Arg1 As %Integer)
{
    Do ##super(.Arg1)
    Write !
    Write Arg1 + Arg1 + Arg1
}
```

出力は、以下のようになります。

```
>Do ##Class(MyClass.Down).SelfAdd(2)
2
4
6
>
```

MyClass.Down.SelfAdd() では、引数名の前にあるピリオドに注目してください。これを省略して、引数を指定せずにメソッドを呼び出すと、<UNDEFINED> エラーを受け取ることになります。

A.5 $CLASSNAME およびその他の動的アクセス関数

InterSystems IRIS® データ・プラットフォームは、オブジェクトに対する一般的な処理をサポートするいくつかの関数を提供します。このサポートは、クラスおよびそのメソッド、プロパティへの参照を実行時に操作することで実現しています。これを Java ではリフレクションといいます。これらの関数は以下のとおりです。

- $CLASSNAME ー クラス名を返します。
- $CLASSMETHOD ー クラス・メソッドへの呼び出しをサポートします。
- $METHOD ー インスタンス・メソッドへの呼び出しをサポートします。
- $PARAMETER ー 指定されたクラスのクラス・パラメータの値を返します。
- $PROPERTY ー インスタンスの特定のプロパティへの参照をサポートします。

ここでは、関数名はすべて大文字で表記されていますが、実際には大文字と小文字は区別されません。

A.5.1 $CLASSNAME

この関数はクラス名を返します。このシグニチャは以下のようになります。

```
$CLASSNAME(Instance)
```

Instance は OREF です。

詳細は、"$CLASSNAME" を参照してください。

A.5.2 $CLASSMETHOD

この関数は、指定されたクラスにある指定されたクラス・メソッドを実行します。このシグニチャは以下のようになります。

```
$CLASSMETHOD (Classname, Methodname, Arg1, Arg2, Arg3, ... )
```

以下はその説明です。

Classname

既存のクラス。

Methodname

最初の引数で指定したクラスのメソッド。

Arg1、Arg2、Arg3、...

指定したメソッドへの引数を置き換える一連の式。

詳細は、"$CLASSMETHOD" を参照してください。

A.5.3 $METHOD

この関数は、指定されたクラスの指定されたインスタンスで、指定されたインスタンス・メソッドを実行します。このシグニチャは以下のようになります。

```
$METHOD (Instance, Methodname, Arg1, Arg2, Arg3, ... )
```

以下はその説明です。

Instance

クラスのインスタンスの OREF。

Methodname

最初の引数のインスタンスで指定したクラスのメソッド。

Arg1、Arg2、Arg3、...

指定したメソッドへの引数を置き換える一連の式。

詳細は、"$METHOD" を参照してください。

A.5.4 $PARAMETER

この関数は、指定されたクラスのクラス・パラメータの値を返します。このシグニチャは以下のようになります。

```
$PARAMETER(Instance,Parameter)
```

以下はその説明です。

Instance

クラスのインスタンスの OREF。

Instance

クラスの完全修飾名、またはクラスのインスタンスの OREF のどちらか。

Parameter

指定されたクラスのパラメータ。

詳細は、"$PARAMETER" を参照してください。

A.5.5 $PROPERTY

この関数は、指定されたクラスにあるインスタンスで、プロパティの値を取得または設定します。プロパティが多次元の場合、プロパティの値にアクセスするときのインデックスとして、プロパティ名に続く引数を使用します。このシグニチャは以下のようになります。

```
$PROPERTY (Instance, PropertyName, Index1, Index2, Index3... )
```

各要素の内容は以下のとおりです。

Instance

クラスのインスタンスの OREF。

PropertyName

最初の引数のインスタンスで指定したクラスのプロパティ。

Index1、Index2、Index3...

多次元プロパティの場合、プロパティが表す配列へのインデックス。

詳細は、"$PROPERTY" を参照してください。

A.6 i%<PropertyName> 構文

このセクションでは、インスタンス変数の追加情報について説明します。このような変数は、プロパティのアクセサ・メソッドをオーバーライドしていない場合は参照する必要がありません。これについては、"プロパティ・メソッドの使用とオーバーライド" を参照してください。

どのようなクラスのインスタンスを作成するときでも、システムは、そのクラスの非計算プロパティごとに 1 つのインスタンス変数を作成します。インスタンス変数は、プロパティの値を保持します。プロパティ PropName の場合、インスタンス変数には i%PropName という名前が付けられます。この変数名は、大文字と小文字が区別されます。このような変数は、クラスのインスタンス・メソッド内で使用できます。

例えば、あるクラスにプロパティ Name と DOB がある場合は、そのクラスのあらゆるインスタンス・メソッド内で、インスタンス変数 i%Name と i%DOB を使用できます。

InterSystems IRIS では、r%PropName や m%PropName という名前の付いた追加のインスタンス変数も内部的に使用しますが、このような変数の直接使用はサポートされていません。

インスタンス変数は、その変数に割り当てられたプロセス・プライベートのメモリ内ストレージを保持します。このような変数は、ローカル変数シンボル・テーブルに保持されないため、Kill コマンドの影響は受けません。

A.7 ..#<Parameter> 構文

..#<Parameter> 構文を使用すると、同じクラスのメソッド内からクラス・パラメータを参照できます。

例えば、クラス定義に以下のパラメータおよびメソッドが含まれているとします。

Class Member

```
Parameter MyParam = 22;
```

そして、以下のメソッドも含まれているとします。

Class Member

```
ClassMethod WriteMyParam()
{
    Write ..#MyParam
}
```

この場合、WriteMyParam() メソッドは、引数として MyParam パラメータの値を含む Write コマンドを呼び出します。

B

Populate ユーティリティの使用

InterSystems IRIS® データ・プラットフォームは、ここで説明するように、永続クラスで使用する擬似ランダム・テスト・データを作成するユーティリティを備えています。

そのようなデータの作成は、データ生成として知られています。このユーティリティは InterSystems IRIS Populate ユーティリティとして知られ、実際のアプリケーションに永続クラスを配置する前に、永続クラスをテストするのに非常に便利です。特に、大量のデータを扱っているときに、アプリケーションのさまざまな部分がどのように機能するかをテストする場合に役立ちます。

Populate ユーティリティは、InterSystems IRIS クラス・ライブラリの一部である、主要な **%Populate** クラスからその名前を取っています。**%Populate** から継承したクラスには、有効なデータを持つクラス・インスタンスの生成と保存に使用できる Populate() メソッドがあります。また、必要に応じたデータを提供するように、**%Populate** クラスの動作をカスタマイズすることもできます。

%Populate クラスと共に、Populate ユーティリティは **%PopulateUtils** を使用します。**%Populate** は、ユーティリティへの継承を提供し、**%PopulateUtils** はヘルパー・クラスです。

Samples-Data サンプル (https://github.com/intersystems/Samples-Data) では Populate ユーティリティが使用されています。(例えば) **SAMPLES** という名前の専用ネームスペースを作成し、そのネームスペースにサンプルをロードすることをお勧めします。一般的な手順は、"InterSystems IRIS で使用するサンプルのダウンロード" を参照してください。

B.1 データ生成の基本

Populate ユーティリティを使用するには、以下の操作を実行します。

1. データを生成する永続クラスごと、および各シリアル・クラスごとに変更を加えます。具体的には、スーパークラスのリストの末尾に **%Populate** を追加して、クラスがインタフェース・メソッドを継承するようにします。例えば、クラスが直接 **%Persistent** を継承する場合、その新規のスーパークラスのリストは、以下のようになります。

   Class Definition

   ```
   Class MyApp.MyClass Extends (%Persistent,%Populate) {}
   ```

   %Populate をプライマリ・スーパークラスとして使用しないでください。つまり、スーパークラスのリスト内で最初のクラスとしてリストしないでください。

2. このようなクラスでは、各プロパティの POPSPEC パラメータと POPORDER パラメータを必要に応じて指定して、Populate ユーティリティが、それらのプロパティにデータを生成する方法を制御します (既定のデータではなく、カスタム・データを生成する必要がある場合)。これについては、次のセクションを参照してください。

3. クラスをリコンパイルします。

4. データを生成するには、それぞれの永続クラスの Populate() メソッドを呼び出します。既定では、このメソッドは、クラスごとに 10 個のレコードを生成します (クラスが参照するシリアル・オブジェクトを含む)。

   ObjectScript

   ```
   Do ##class(MyApp.MyClass).Populate()
   ```

   作成するオブジェクトの数を指定することもできます。

   ObjectScript

   ```
   Do ##class(MyApp.MyClass).Populate(num)
   ```

   num は希望するオブジェクトの数です。

   これは、手動で各クラスにレコードを追加する場合と同じ順序で実行します。つまり、Class A に Class B を参照するプロパティがある場合は、以下のテーブルを使用して、最初にデータを生成するクラスを判断します。

Class A のプロパティの形式	Class B の継承元	最初にデータ生成するクラス
以下のいずれかの形式 : ・ Property PropertyName as ClassB; ・ Property PropertyName as List of ClassB; ・ Property PropertyName as Array of ClassB;	%SerialObject	ClassA (これにより、自動的に ClassB にデータを生成する)
以下のいずれかの形式 : ・ Property PropertyName as ClassB; ・ Property PropertyName as List of ClassB; ・ Property PropertyName as Array of ClassB;	%Persistent	ClassB
以下のいずれかの形式 : ・ Relationship PropertyName as ClassB [Cardinality = one ...]; ・ Relationship PropertyName as ClassB [Cardinality = parent ...];	%SerialObject または %Persistent	ClassB
以下のいずれかの形式 : ・ Relationship PropertyName as ClassB [Cardinality = many...]; ・ Relationship PropertyName as ClassB [Cardinality = child ...];	%SerialObject または %Persistent	ClassA

生成されたデータを削除するときには、永続インタフェースの %DeleteExtent() メソッド (安全性を重視)、または %KillExtent() メソッド (速度を重視) を使用します。詳細は、"保存したオブジェクトの削除" を参照してください。

Tipヒント 実際には、多くの場合、コードを変更するたびに、繰り返してクラスにデータを生成することが必要になります。そのため、正しい順序でクラスにデータを生成するメソッドまたはルーチンと、生成したデータを削除するメソッドまたはルーチンを作成しておくと作業に役立ちます。

B.1.1 Populate() の詳細

正式には、Populate() クラス・メソッドのシグニチャは以下のようになります。

```
classmethod Populate(count As %Integer = 10,
                     verbose As %Integer = 0,
                     DeferIndices As %Integer = 1,
                     ByRef objects As %Integer = 0,
                     tune As %Integer = 1,
                     deterministic As %Integer = 0) as %Integer
```

以下はその説明です。

- count は、作成する予定のオブジェクト数です。
- verbose では、進行状況メッセージを現在のデバイスに出力するかどうかを指定します。
- DeferIndices では、データの生成後にインデックスを並べ替える (True) か、データの生成中に並べ替えるかを指定します。
- objects は、生成されたオブジェクトを格納する参照渡しの配列です。
- tune では、データの生成後に $SYSTEM.SQL.TuneTable() を実行するかどうかを指定します。これが 0 の場合、このメソッドは $SYSTEM.SQL.TuneTable() を実行しません。これが 1 (既定) の場合、メソッドは、このテーブルに対して $SYSTEM.SQL.TuneTable() を実行します。これが 1 より大きい値の場合、メソッドは、このテーブルと、このクラスの永続スーパークラスによって投影されたテーブルに、$SYSTEM.SQL.TuneTable() を実行します。
- deterministic は、メソッドを呼び出すたびに同じデータを生成するかどうかを指定します。既定では、メソッドを呼び出すたびに異なるデータが生成されます。

Populate() は、実際にデータを生成したオブジェクトの数を返します。

ObjectScript

```
 Set objs = ##class(MyApp.MyClass).Populate(100)
 // objs is set to the number of objects created.
 // objs will be less than or equal to 100
```

最小長や最大長などの定義された制約の場合、生成されたデータの一部には検証に合格しないものもあるので、個別のオブジェクトは保存されません。この場合、Populate() は、指定されたオブジェクト数よりも少ない数を作成します。

エラーによりオブジェクトが保存されない場合は、これが 1,000 回連続して (正常な保存なしに) 発生すると、Populate() は終了します。

B.2 既定の動作

このセクションでは、Populate() メソッドが、以下の種類のプロパティに既定でデータを生成する方法について説明します。

- リテラル・プロパティ
- コレクション・プロパティ
- シリアル・オブジェクトを参照するプロパティ

- 永続オブジェクトを参照するプロパティ
- リレーションシップ・プロパティ

Populate() メソッドは、ストリーム・プロパティを無視します。

B.2.1 リテラル・プロパティ

このセクションでは、Populate() メソッドが、以下の形式のプロパティに既定でデータを生成する方法について説明します。

```
Property PropertyName as Type;
Property PropertyName;
```

Type は、データ型クラスです。

このようなプロパティの場合、Populate() メソッドは、最初に名前を調べます。以下に示すように、一部のプロパティ名は特別に処理されます。

任意の大文字小文字の組み合わせで、プロパティ名が以下の場合	データを生成するために Populate() が呼び出すメソッド
NAME	Name()
SSN	SSN()
COMPANY	Company()
TITLE	Title()
PHONE	USPhone()
CITY	City()
STREET	Street()
ZIP	USZip()
MISSION	Mission()
STATE	USState()
COLOR	Color()
PRODUCT	Product()

プロパティの名前が上記のいずれにも当てはまらない場合、Populate() メソッドはプロパティ・タイプを調べて、それに適した値を生成します。例えば、プロパティ・タイプが %String の場合、Populate() メソッドはランダムな文字列を生成します (プロパティの MAXLEN パラメータが考慮されます)。別の例として、プロパティ・タイプが %Integer の場合、Populate() メソッドはランダムな整数を生成します (プロパティの MINVAL および MAXVAL パラメータが考慮されます)。

プロパティにタイプがない場合、InterSystems IRIS は、そのプロパティ・タイプを文字列であると見なします。つまり、Populate() メソッドは、その値にランダムな文字列を生成します。

B.2.1.1 例外

Populate() メソッドは、プロパティがプライベート、多次元、または計算の場合や、プロパティに初期値式がある場合はデータを生成しません。

B.2.2 コレクション・プロパティ

このセクションでは、Populate() メソッドが、以下の形式のプロパティに既定でデータを生成する方法について説明します。

```
Property PropertyName as List of Classname;
Property PropertyName as Array of Classname;
```

これに該当するプロパティは、以下のようになります。

- 参照されるクラスがデータ型クラスの場合、Populate() メソッドは値のリストまたは配列を (適宜) 生成します。データの生成には、前述したデータ型クラスのロジックが使用されます。
- 参照されるクラスがシリアル・オブジェクトの場合、Populate() メソッドはシリアル・オブジェクトのリストまたは配列を (適宜) 生成します。データの生成には、前述したシリアル・オブジェクトのロジックが使用されます。
- 参照されるクラスが永続クラスの場合、Populate() メソッドは、参照されるクラスに含まれるエクステントのランダム・サンプルを実行し、そのサンプルからランダムに値を選択します。その値を使用してリストまたは配列を (適宜) 生成します。

B.2.3 シリアル・オブジェクトを参照するプロパティ

このセクションでは、Populate() メソッドが、以下の形式のプロパティに既定でデータを生成する方法について説明します。

```
Property PropertyName as SerialObject;
```

SerialObject は、**%SerialObject** を継承するクラスです。

これに該当するプロパティは、以下のようになります。

- 参照されるクラスが **%Populate** を継承している場合、Populate() メソッドはクラスのインスタンスを作成し、前述のセクションで説明したようにプロパティ値を生成します。
- 参照されるクラスが **%Populate** を継承していない場合、Populate() メソッドは、そのプロパティの値を生成しません。

B.2.4 永続オブジェクトを参照するプロパティ

このセクションでは、Populate() メソッドが、以下の形式のプロパティに既定でデータを生成する方法について説明します。

```
Property PropertyName as PersistentObject;
```

PersistentObject は、**%Persistent** を継承するクラスです。

これに該当するプロパティは、以下のようになります。

- 参照されるクラスが **%Populate** を継承している場合、Populate() メソッドは参照されるクラスに含まれるエクステントのランダム・サンプルを実行し、そのサンプルからランダムに 1 つの値を選択します。

  そのため、参照されるクラスのデータを最初に生成しておく必要があります。または、他の何らかの方法でクラスのデータを作成します。
- 参照されるクラスが **%Populate** を継承していない場合、Populate() メソッドは、そのプロパティの値を生成しません。

リレーションシップの詳細は、次のセクションを参照してください。

B.2.5 リレーションシップ・プロパティ

このセクションでは、Populate() メソッドが、以下の形式のプロパティに既定でデータを生成する方法について説明します。

```
Relationship PropertyName as PersistentObject;
```

PersistentObject は、%Persistent を継承するクラスです。

これに該当するプロパティは、以下のようになります。

- 参照されるクラスが %Populate を継承している場合 :
  - リレーションシップのカーディナリティが one または parent の場合、Populate() メソッドは参照されるクラスに含まれるエクステントのランダム・サンプルを実行し、そのサンプルからランダムに 1 つの値を選択します。

    そのため、参照されるクラスのデータを最初に生成しておく必要があります。または、他の何らかの方法でクラスのデータを作成します。
  - リレーションシップのカーディナリティが many または children の場合、Populate() メソッドは、このプロパティを無視します。これは、このプロパティの値が、このクラスのエクステントには保存されないためです。
- 参照されるクラスが %Populate を継承していない場合、Populate() メソッドは、そのプロパティの値を生成しません。

B.3 POPSPEC パラメータの指定

%Populate を拡張しているクラスの特定のプロパティについては、そのプロパティに Populate() メソッドでデータを生成する方法をカスタマイズできます。そのためには、以下を実行します。

- このプロパティに適したランダムな値を返すメソッドを見つけるか、作成します。

  %PopulateUtils クラスには、そのようなメソッドが大量に用意されています。詳細は、クラスリファレンスを参照してください。
- このプロパティの POPSPEC パラメータを指定して、このメソッドを参照するようにします。最初のサブセクションで詳細を説明します。

POPSPEC パラメータは、リストおよび配列プロパティに追加のオプションを提供します。これについては後続のサブセクションで説明します。

コレクション以外のリテラル・プロパティの場合は、このプロパティに使用する値を格納している SQL テーブル列を特定するという別の方法もあります。その後で、POPSPEC パラメータを指定して、このプロパティを参照するようにします。最後のサブセクションを参照してください。

注釈　また、クラス全体のデータ生成を制御する、クラス・レベルで定義された POPSPEC パラメータも存在します。これは、プロパティ特有の POPSPEC パラメータによって置き換えられた、旧式のメカニズムです (互換性のために含められています)。このトピックでは、これ以上の詳しい説明はしません。

B.3.1 非コレクション・プロパティの POPSPEC パラメータを指定する方法

コレクション以外のリテラル・プロパティには、以下のいずれかのバリエーションを使用します。

- POPSPEC="MethodName()" — この場合、Populate() は %PopulateUtils クラスのクラス・メソッド MethodName() を呼び出します。

- `POPSPEC=".MethodName()"` ー この場合、Populate() は生成されるインスタンスのインスタンス・メソッド `MethodName()` を呼び出します。
- `POPSPEC="##class(ClassName).MethodName()"` ー この場合、Populate() は `ClassName` クラスのクラス・メソッド `MethodName()` を呼び出します。

例：

Class Member

```
Property HomeCity As %String(POPSPEC = "City()");
```

指定のメソッドに引数として文字列値を渡す必要がある場合は、その文字列を囲む先頭の引用符と末尾の引用符を 2 つ続けます。以下に例を示します。

Class Member

```
Property PName As %String(POPSPEC = "Name(""F"")");
```

また、指定したメソッドが返す値に文字列を追加することもできます。以下に例を示します。

Class Member

```
Property JrName As %String(POPSPEC = "Name()_"" jr."" ");
```

その文字列を囲む先頭の引用符と末尾の引用符は 2 つ続ける必要がある点に注意してください。先頭に文字列を追加することはできません。POPSPEC は、メソッドが先頭になることを想定しているためです。

別の方法については、"SQL テーブル経由で POPSPEC パラメータを指定する方法" も参照してください。

B.3.2 リスト・プロパティの POPSPEC パラメータを指定する方法

プロパティがリテラルまたはオブジェクトのリストである場合は、以下のバリエーションを使用できます。

```
POPSPEC="basicspec:MaxNo"
```

以下はその説明です。

- basicspec は、前のセクションに示した基本バリエーションの 1 つです。プロパティがオブジェクトのリストの場合は、basicspec を空のままにしておきます。
- MaxNo は、リストに含まれる項目の最大数です。既定値は 10 です。

例：

Class Member

```
Property MyListProp As list Of %String(POPSPEC = ".MyInstanceMethod():15");
```

basicspec は省略できます。以下に例を示します。

Class Member

```
Property Names As list of Name(POPSPEC=":3");
```

以下の例では、複数のデータ型のリストがあります。Colors は文字列のリスト、Kids は永続オブジェクトへの参照リスト、Addresses は埋め込みオブジェクトのリストです。

```
Property Colors As list of %String(POPSPEC="ValueList("",Red,Green,Blue"")");
Property Kids As list of Person(POPSPEC=":5");
Property Addresses As list of Address(POPSPEC=":3");
```

Colors プロパティのデータを生成するには、Populate() メソッドで、PopulateUtils クラスの ValueList() メソッドを呼び出します。この例では、引数として、このメソッドにコンマ区切りリストを渡している点に注意してください。Kids では、自動的に参照を生成する指定されたメソッドはありません。Addresses プロパティでは、シリアル Address クラスが %Populate から継承され、データがそのクラスのインスタンスに自動的に入力されます。

B.3.3 配列プロパティの POPSPEC パラメータを指定する方法

リテラルまたはオブジェクトの配列であるプロパティには、以下のバリエーションを使用できます。

```
POPSPEC="basicspec:MaxNo:KeySpecMethod"
```

以下はその説明です。

- basicspec は、前出の基本バリエーションの 1 つです。プロパティがオブジェクトの配列の場合は、basicspec を空のままにしておきます。
- MaxNo は、配列内の項目の最大数です。既定値は 10 です。
- KeySpecMethod は、配列のキーに使用する値を生成するメソッドの仕様です。既定は、`String()` です。これは、InterSystems IRIS が %PopulateUtils の String() メソッドを呼び出すことを意味します。

以下の例では、数種類のデータのタイプと異なる種類のキーの配列を示します。

```
Property Tix As array of %Integer(POPSPEC="Integer():20:Date()");
Property Reviews As array of Review(POPSPEC=":3:Date()");
Property Actors As array of Actor(POPSPEC=":15:Name()");
```

Tix プロパティのデータは、PopulateUtils クラスの Integer() メソッドを使用して生成されています。このキーは、PopulateUtils クラスの Date() メソッドを使用して生成されています。Reviews プロパティには、指定されたメソッドがないので、自動的に参照が生成されることになります。またそのキーも、Date() メソッドを使用して生成されます。Actors プロパティには、指定されたメソッドはなく、自動的に参照が生成されます。またそのキーも、PopulateUtils クラスの Name() メソッドを使用して生成されます。

B.3.4 SQL テーブル経由で POPSPEC パラメータを指定する方法

POPSPEC には、ランダムな値を返すメソッドを指定する以外にも、使用する SQL テーブル名と SQL 列名を指定できます。このようにすると、Populate() メソッドは、そのテーブルの列から重複しない列値を返すダイナミック・クエリを作成します。このバリエーションの POPSPEC には、以下の構文を使用します。

```
POPSPEC=":MaxNo:KeySpecMethod:SampleCount:Schema_Table:ColumnName"
```

以下はその説明です。

- MaxNo と KeySpecMethod はオプションです。これらは、コレクション・プロパティにのみ適用されます (リストおよび配列についてのサブセクションを参照)。
- SampleCount は、指定の列から取得する非重複値の数です。これを開始点として使用します。この数が、その列に含まれる既存の非重複値の数よりも大きい場合は、すべての値が使用される可能性があります。

- Schema_Table は、テーブルの名前です。
- ColumnName は、列の名前です。

例：

Class Member

```
Property P1 As %String(POPSPEC=":::100:Wasabi_Data.Outlet:Phone");
```

この例では、プロパティ P1 は、Wasabi_Data.Outlet テーブルから取得した 100 個の電話番号のリストから、ランダムな値を受け取ります。

B.4 生成されたプロパティを別のプロパティのベースにする方法

場合によっては、あるプロパティ (A) に適した値のセットが、別のプロパティ (B) に存在する値に応じて決まることがあります。このような場合は、以下のようにします。

- プロパティ A の値を生成するインスタンス・メソッドを作成します。このメソッドでは、インスタンス変数を使用して、プロパティ B (および考慮に入れる必要のあるその他のプロパティ) の値を取得します。以下に例を示します。

  Class Member

  ```
  Method MyMethod() As %String
  {
      if (i%MyBooleanProperty) {
          quit "abc"
      } else {
          quit "def"
      }
  }
  ```

  インスタンス変数の詳細は、"i%PropertyName" を参照してください。

  該当するプロパティの POPSPEC パラメータで、このメソッドを使用します。"POPSPEC パラメータの指定" を参照してください。

- 特定の順序でデータを生成する必要のあるプロパティの POPORDER パラメータを指定します。このパラメータは整数にする必要があります。InterSystems IRIS は、POPORDER の値が小さいプロパティにデータを生成してから、POPORDER の値が大きいプロパティにデータを生成します。以下に例を示します。

  ```
  Property Name As %String(POPORDER = 2, POPSPEC = ".MyNameMethod()");
  Property Gender As %String(POPORDER = 1, VALUELIST = ",1,2");
  ```

B.5 %Populate の動作

このセクションでは、%Populate が内部的にどのように動作するかを説明します。%Populate クラスには、Populate() と PopulateSerial() の 2 つのメソッド・ジェネレータがあります。%Populate を継承する各永続クラス、またはシリアル・クラスには、これら 2 つのメソッドの一方、または他方が (必要に応じて) 含まれます。

ここでは Populate メソッドだけを説明します。Populate() メソッドはループであり、要求されたオブジェクトの個数だけ繰り返されます。

ループの内部では、コードは以下のように動作します。

1. 新規オブジェクトを作成します。
2. このプロパティに対する値を設定します。
3. オブジェクトを保存し、閉じます。

POPSPEC パラメータをオーバーライドしない単純なプロパティは、以下の形式のコードを使用して生成された値を持っています。

ObjectScript

```
Set obj.Description = ##class(%PopulateUtils).String(50)
```

また、Name:Name() を指定して、%PopulateUtils のライブラリ・メソッドを使用すると、以下が生成されます。

ObjectScript

```
Set obj.Name = ##class(%PopulateUtils).Name()
```

埋め込み Home プロパティは、以下のようなコードを作成します。

ObjectScript

```
Do obj.HomeSetObject(obj.Home.PopulateSerial())
```

ジェネレータはクラスのすべてのプロパティにわたってループし、一部のプロパティに対して以下のようなコードを作成します。

1. プロパティがプライベート、計算、多次元であるか、プロパティに初期値式があるかをチェックします。上記のいずれかが当てはまる場合、ジェネレータは終了します。
2. プロパティに POPSPEC オーバーライドがある場合、ジェネレータはそれを使用して終了します。
3. プロパティが参照の場合、最初にループするとき、ジェネレータはランダムな ID のリストを作成し、リストから 1 つ取って終了します。次のループからは、ジェネレータは単にリストから ID を取って終了します。
4. プロパティの名前が、特別に処理される名前のいずれかである場合、ジェネレータは、それに対応するライブラリ・メソッドを使用してから終了します。
5. ジェネレータがプロパティ・タイプに基づいたコードを生成できる場合は、コードを生成してから終了します。
6. それ以外の場合、ジェネレータはプロパティを空の文字列に設定します。

利用可能なメソッドのリストは、%PopulateUtils を参照してください。

B.6 カスタム生成のアクションと OnPopulate() メソッド

生成されたデータを詳細に制御するには、OnPopulate() メソッドを定義します。OnPopulate() メソッドを定義しておくと、Populate() メソッドでオブジェクトを生成するたびにそのメソッドが呼び出されます。このメソッドは、プロパティに値を割り当てた後、オブジェクトがディスクに保存される前に呼び出されます。Populate() メソッドを呼び出すと、OnPopulate() メソッドがあるかどうかが確認され、Populate() メソッドで生成したオブジェクトごとに OnPopulate() が呼び出されます。

このインスタンス・メソッドは、プロパティに値を割り当てた後で、オブジェクトがディスクに保存される前に、Populate メソッドによって呼び出されます。このメソッドは、生成されたデータの追加の制御を提供します。OnPopulate() メソッドが存在する場合、Populate メソッドは、オブジェクトを生成するたびにこのメソッドを呼び出します。

そのシグニチャは、以下のとおりです。

Class Member

```
Method OnPopulate() As %Status
{
    // body of method here...
}
```

注釈　　これは、プライベート・メソッドではありません。

このメソッドは、%Status コードを返します。失敗ステータスの場合、生成されているインスタンスは破棄されます。

例えば、Memo というストリーム・プロパティを持っていて、生成時に値をそれに割り当てたい場合、OnPopulate() メソッドを提供します。

Class Member

```
Method OnPopulate() As %Status
{
    Do ..Memo.Write("Default value")
    QUIT $$$OK
}
```

このメソッドは、%Library.Populate のサブクラスでオーバーライドできます。

B.7 代替手段 : ユーティリティ・メソッドの作成

%Populate クラスと %PopulateUtils クラスのメソッドを使用する方法は、もう 1 つあります。スーパークラスとして %Populate を使用するのではなく、目的のクラスに応じたデータを生成する、ユーティリティ・メソッドを作成します。

このコードでは、各クラスに対して目的の回数だけ繰り返し処理が実行されます。繰り返し処理ごとに、以下の手順を実行します。

1. 新しいオブジェクトを作成します。
2. 適切なランダム値 (または、ランダムに近い値) を使用して各プロパティを設定します。

   プロパティのデータを生成するには、%Populate または %PopulateUtils のメソッドを呼び出すか、独自に作成したメソッドを使用します。
3. オブジェクトを保存します。

標準的な方法の場合と同様に、独立クラスのデータを生成してから、依存クラスのデータを生成する必要があります。

B.7.1 データに構造を構築するためのヒント

場合によっては、全体に対して特定の割合にのみ、特定の値を含めることが必要になります。$RANDOM 関数を使用してこれを行うことができます。例えば、この関数を使用して、引数として指定した切り捨てパーセンテージに応じて、True と False をランダムに返すメソッドを定義します。したがって、例えば、10% の場合や 75% の場合に True を返すことができます。

プロパティのデータを生成する場合、このメソッドを使用して、値を割り当てるかどうかを判別します。

ObjectScript

```
 If ..RandomTrue(15) {
    set ..property="something"
 }
```

ここに示す例では、レコードの約 15% がこのプロパティに対して指定された値を持つようになります。

この他の場合は、ディストリビューションのシミュレートが必要なこともあります。そのためには、くじ引きシステムをセットアップし、使用します。例えば、値の 1/4 を A に、1/4 を B に、1/2 を C にする必要があるとします。このくじ引きのロジックは、以下のようにすることができます。

1. 1 ～ 100 (両端の値を含む) から 1 つの整数を選択します。
2. その数字が 25 より小さい場合は、値 A を返します。
3. その数字が 25 ～ 49である場合 (両端の値を含む) は、値 B を返します。
4. それ以外の場合は、値 C を返します。

C

%Dictionary クラスの使用

ここでは、クラス定義クラスについて説明します。このクラスは、すべてのクラス定義にオブジェクト・アクセスと SQL アクセスを提供する永続クラスのセットです。

C.1 クラス定義クラスの概要

クラス定義クラスは、すべてのクラス定義へのオブジェクト・アクセスと SQL アクセスを提供します。これらのクラスを使用して、クラス定義の調査、クラス定義の修正、新規クラスの生成をプログラムでできます。ドキュメントを自動生成するプログラムを記述することもできます。これらのクラスは、%Dictionary パッケージに含まれます。

注釈　%Library パッケージには、従来の一連のクラス定義クラスがあります。これらは既存のアプリケーションとの互換性のために保存されています。新規のコードでは、%Dictionary パッケージ内のクラスを使用する必要があります。これらのクラスを使用するときは、正しいパッケージ名を指定してください。

クラス定義クラスには、定義クラスとコンパイル・クラスの 2 つの並列セットがあります。

定義クラス定義は、特定のクラスの定義を表します。それは、そのクラスによって定義された情報のみを含み、スーパークラスから継承された情報は含みません。ディクショナリのクラスに関する情報を提供するだけでなく、これらのクラスを使用して、プログラム的にクラス定義を変更したり新規作成できます。

コンパイル・クラス定義は、スーパークラスから継承されるクラス・メンバをすべて含みます。コンパイル・クラス定義オブジェクトは、コンパイルされたクラスからインスタンスを生成するだけです。コンパイル・クラス定義は保存できません。

この付録では、定義クラス定義のみ説明します。コンパイル・クラス定義のオペレーションも類似しています。

定義クラスを表すクラス定義のファミリには、以下のものが含まれます。

クラス	説明
%Dictionary.ClassDefinition	クラス定義を表します。クラス・メンバ定義を含むコレクションと、クラス・キーワードを含みます。
%Dictionary.ForeignKeyDefinition	クラス内の外部キー定義を表します。
%Dictionary.IndexDefinition	クラス内のインデックス定義を表します。
%Dictionary.MethodDefinition	クラス内のメソッド定義を表します。
%Dictionary.ParameterDefinition	クラス内のパラメータ定義を表します。
%Dictionary.PropertyDefinition	クラス内のプロパティ定義を表します。
%Dictionary.QueryDefinition	クラス内のクエリ定義を表します。
%Dictionary.TriggerDefinition	クラス内の SQL トリガ定義を表します。

重要

繰り返しますが、コンパイルされていないクラス定義 (%Dictionary.ClassDefinition のインスタンスとして) の内容は、コンパイル済みクラス定義 (%Dictionary.CompiledClass のインスタンスとして) の内容と必ずしも同一ではありません。%Dictionary.ClassDefinition クラスは、クラスの定義を調査または変更する API を提供しますが、継承を解決したコンパイル済みクラスを表すことはありません。一方、%Dictionary.CompiledClass は、継承を解決したコンパイル済みクラスを表します。

例えば、クラス定義内の特定のキーワードの値を特定しようとする場合は、%Dictionary.ClassDefinition の keywordnameIsDefined() メソッド (OdbcTypeIsDefined() や ServerOnlyIsDefined() など) を使用します。このブーリアン・メソッドが false を返す場合、キーワードはそのクラスに対して明示的に定義されていません。クラス定義のキーワードの値を調べると、それは既定値です。ただし、コンパイル (継承の解決を含む) の後では、キーワードの値は、継承によって決定され、定義されていた値とは異なることがあります。

C.2 クラス定義のブラウズ

管理ポータルの SQL ページを使用すると、クラス定義クラスを参照できます。

同様に、クラス定義全体をプログラムで参照することもできます。これは、その他の種類のデータを参照するために使用するのと同じ手法を使用します。ダイナミック SQL を使用したり、特定のクラス定義を表す永続オブジェクトをインスタンス化したりできます。

例えば、InterSystems IRIS® データ・プラットフォームのプロセスから、以下の %Dictionary.ClassDefinition:Summary() クエリを使用して、現在のネームスペースのディクショナリ内で定義されているすべてのクラスのリストを取得できます。

ObjectScript

```
  set stmt=##class(%SQL.Statement).%New()
  set status = stmt.%PrepareClassQuery("%Dictionary.ClassDefinition","Summary")
  if $$$ISERR(status) {write "%Prepare failed:" do $SYSTEM.Status.DisplayError(status) quit}

  set rset=stmt.%Execute()
  if (rset.%SQLCODE '= 0) {write "%Execute failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message
 quit}

  while rset.%Next()
  {
    write rset.%Get("Name"),!
  }
  if (rset.%SQLCODE < 0) {write "%Next failed:", !, "SQLCODE ", rset.%SQLCODE, ": ", rset.%Message
quit}
```

このサンプル・メソッドは、現在のネームスペース内の表示可能なすべてのクラス (システム・ライブラリ・クラスを含む) の名前を書き込みます。%Dictionary.ClassDefinition:Summary() クエリによって返されたさまざまな列を使用して、望まないクラスをフィルタできます。

特定のクラス定義に関する詳細は、そのクラスの %Dictionary.ClassDefinition オブジェクトを開き、そのプロパティを表示することで取得できます。%Dictionary.ClassDefinition オブジェクトを格納するための ID は、そのクラス名です。

ObjectScript

```
Set cdef = ##class(%Dictionary.ClassDefinition).%OpenId("Sample.Person")
Write cdef.Name,!

// get list of properties
Set count = cdef.Properties.Count()
For i = 1:1:count {
    Write cdef.Properties.GetAt(i).Name,!
}
```

クラス名とそのパッケージ名から成る完全記述名を指定しない場合、%OpenId() への呼び出しが失敗することに注意してください。

C.3 クラス定義の修正

%Dictionary.ClassDefinition オブジェクトを開き、任意の変更を行い、%Save() メソッドを使用してそれを保存することで、既存のクラス定義を修正できます。

新規の %Dictionary.ClassDefinition オブジェクトを生成し、そのプロパティを入力して保存することで、新規のクラスを生成できます。%Dictionary.ClassDefinition オブジェクトを生成するときには、%New() コマンドでクラスの名前を渡す必要があります。クラス (プロパティやメソッド) にメンバを追加するとき、("class_name.member_name" を含む文字列を %New() コマンドに渡して) 対応する定義クラスを作成し、オブジェクトを %Dictionary.ClassDefinition オブジェクト内の該当コレクションに追加する必要があります。

例えば以下のようになります。

ObjectScript

```
Set cdef = ##class(%Dictionary.ClassDefinition).%New("MyApp.MyClass")
If $SYSTEM.Status.IsError(cdef)  {
    Do $system.Status.DecomposeStatus(%objlasterror,.Err)
    Write !, Err(Err)
}
Set cdef.Super = "%Persistent,%Populate"

// add a Name property
Set pdef = ##class(%Dictionary.PropertyDefinition).%New("MyClass:Name")
If $SYSTEM.Status.IsError(pdef)  {
    Do $system.Status.DecomposeStatus(%objlasterror,.Err)
    Write !,Err(Err)
}

Do cdef.Properties.Insert(pdef)

Set pdef.Type="%String"

// save the class definition object
Do cdef.%Save()
```

C.4 関連項目

- クラスの定義

- 最上位レベル・クラスの構文とキーワード

D

オブジェクト同期化機能の使用法

ここでは、オブジェクト同期化機能について説明します。この機能を使用すると、不定期的に接続されるシステム上のデータベースにある特定のテーブルを同期できます。

D.1 オブジェクトの同期化の概要

オブジェクトの同期化は、InterSystems IRIS® データ・プラットフォームのオブジェクトに使用できるツールのセットです。オブジェクトの同期化により、アプリケーション開発者は、不定期に接続されるシステム上のデータベースを同期化するためのメカニズムをセットアップできます。このプロセスを経て、各データベースのオブジェクトが更新されます。オブジェクトの同期化によって、高可用性を提供する InterSystems IRIS システム・ツールに機能が補足されます。これは、リアルタイムの更新をサポートするためのものではなく、不定期に更新する必要があるシステムで最も役立つ機能です。

例えば、中央のサーバにデータベースのマスタ・コピーがあり、クライアント・マシンにセカンダリ・コピーがある環境に、オブジェクトの同期化の標準アプリケーションがあるとします。各営業員がノート・パソコンに該当のデータベースのコピーを持っている営業用データベースの場合を考えてみましょう。営業員の Mary は、ネットワークに接続していないとき、データベースの自分用のコピーを更新します。彼女が自分のコンピュータをネットワークに接続すると、中央とリモートのデータベースのコピーが同期化されます。これは、時間単位、1 日単位、あるいは任意の間隔で実行できます。

2 つのデータベース間でオブジェクトを同期化するには、それぞれのデータベースをもう一方のデータベースのデータで更新する必要があります。しかし、InterSystems IRIS では、このような双方向での同期はサポートされていません。正確には、一方のデータベースの更新がもう一方のデータベースに送信された後、後者のデータベースの更新が前者のデータベースに送信されます。標準アプリケーションでは、(前述の営業データベースの例のように) 1 つのメイン・データベースと 1 つ以上のローカル・データベースがある場合、更新はまずローカル・データベースからメイン・データベースに送信され、次に、メイン・データベースからローカル・データベースに送信されるようにすることをお勧めします。

オブジェクトの同期化の場合、クライアントとサーバの概念は、慣例に従うという理由のみで存在します。任意の 2 つのデータベースでは、双方向の更新が可能です。2 つ以上のデータベースがある場合、すべてのデータベース (メイン・データベースと個別に同期化するローカル・データベースなど) の更新に使用する方法を選択できます。

この章では、以下の項目について説明します。

- GUID
- 更新の機能
- SyncSet および SyncTime オブジェクト

D.1.1 GUID

更新が正確に実行されるには、データベースの各オブジェクトを一意に区別できる必要があります。この機能のために、InterSystems IRIS では、各オブジェクト・インスタンスに GUID (Globally Unique ID) を割り当てます。この GUID によって各オブジェクトは例外なく一意に区別可能になります。

GUID は、GUIDENABLED パラメータ値に基づいてオプションで作成されます。GUIDENABLED の値が 1 である場合は、新しい各オブジェクト・インスタンスに GUID が割り当てられます。

以下の例を考えてみます。2 つのデータベースが同期化されて、それぞれのデータベースに同じオブジェクトのセットがあるとします。同期化の後、各データベースに新しいオブジェクトが 1 つ追加されています。オブジェクトの同期化では、この 2 つのオブジェクトが共通の GUID を共有している場合、これらのオブジェクトは状態が異なる同一のオブジェクトと見なされます。各オブジェクトに固有の GUID が割り当てられている場合、これらのオブジェクトは異なるオブジェクトと見なされます。

D.1.2 更新の機能

1 つのデータベースから別のデータベースへの各更新は、トランザクションのセットとして送信されます。これにより、相互依存するオブジェクトすべてが一緒に更新されます。各トランザクションの内容は、ソース・データベースのジャーナルの内容によって異なります。更新には、最後の同期化以降に発生したすべてのトランザクションを限度として、1 つ以上のトランザクションを含めることができます。

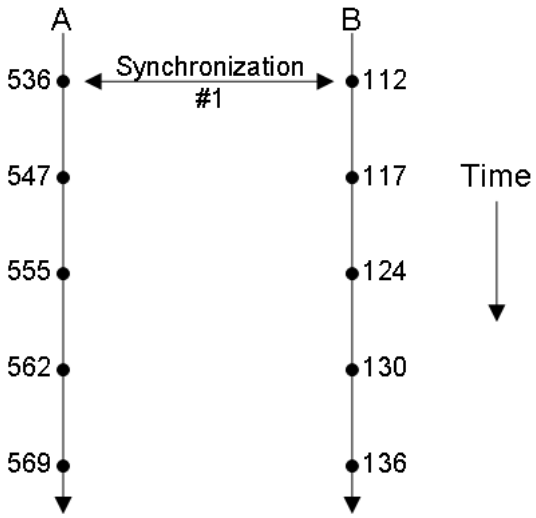
以下の状況はアプリケーションで解決します。

- 一意のキーを共有する 2 つのインスタンスの GUID が異なる場合。この 2 つのレコードが単一のオブジェクトを表しているのか、または 2 つの固有のオブジェクトを表しているのかを判別する必要があります。
- 2 つの変更を調整する必要がある場合。この 2 つの変更が共通のプロパティに対して行われたのか、または接点のないプロパティのセットに対して行われたのかを判別する必要があります。

D.1.3 SyncSet および SyncTime オブジェクト

2 つのデータベースを同期化するときは、それぞれが、他方のデータベースにはないトランザクションを含んでいます。以下はその図です。

図 IV-1: 2 つの同期化されていないデータベース



データベース A とデータベース B は、データベース A のトランザクション 536 とデータベース B のトランザクション 112 で同期化されています。各データベースの後続のトランザクションは、他方のデータベースで更新される必要があります。これを行うために、InterSystems IRIS では SyncSet オブジェクトが使用されます。このオブジェクトには、データベースの更新に使用するトランザクションのリストが含まれています。例えば、データベース A からデータベース B に同期化する場合、SyncSet オブジェクトの既定の内容はトランザクション 547、555、562、および 569 です。同様に、データベース B からデータベース A に同期化する場合、SyncSet オブジェクトの既定の内容は、117、124、130、および 136 です (トランザクションでは連続した番号は使用されません。これは、各トランザクションに複数の挿入、更新、および削除がカプセル化されており、そのそれぞれがその間の番号を使用するためです)。

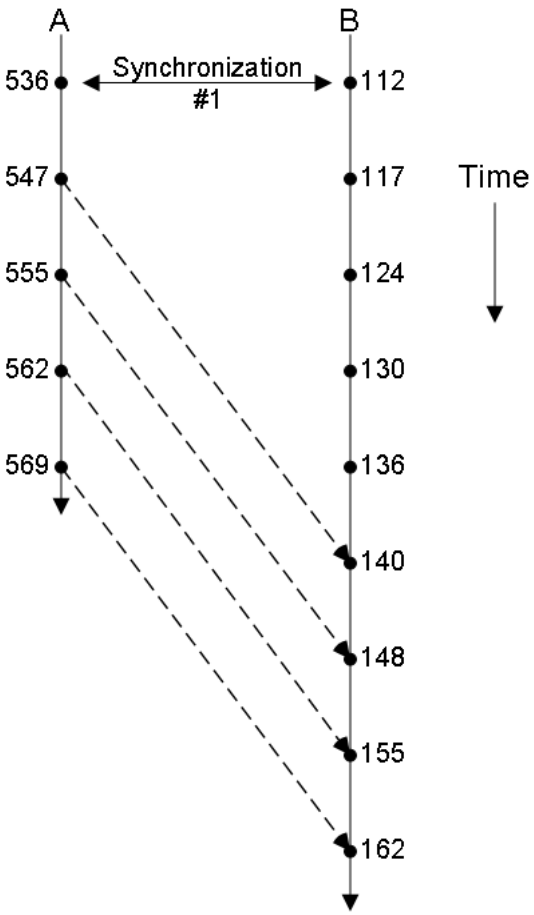
各データベースには、他方のデータベースとの同期化の履歴があります。このレコードを SyncTime テーブルと呼びます。データベースの場合、次の形式になっています。

```
Database     Namespace     Last Transaction Sent     Last Transaction Received
-------------------------------------------------------------------------------
B            User          536                       112
```

注釈　各トランザクションと対応している番号はタイム・スタンプではありません。個々のデータベース内でトランザクションを報告した順序を示しています。

データベース A からデータベース B への同期化が完了すると、2 つのデータベースは次のようになります。

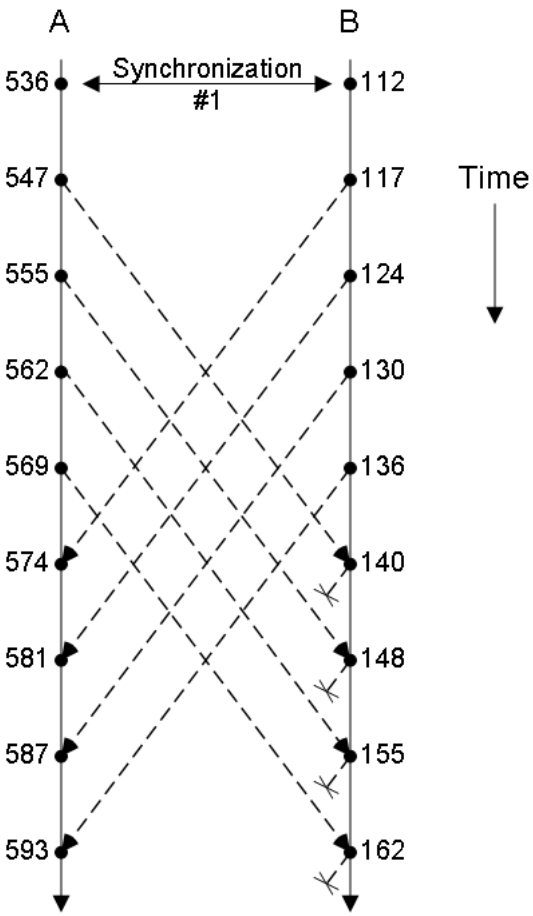
図 IV-2: 一方のデータベースが他方のデータベースと同期化された後の 2 つのデータベース



トランザクションがデータベース B に追加されるため、そのデータベースで新しいトランザクション番号が付けられます。

同様に、データベース B からデータベース A への同期化によって、データベース A に 117、124、130、および 136 が追加され、新しいトランザクション番号が付きます。

図 IV-3: 2 つの同期化されたデータベース



データベース A が発端になるデータベース B のトランザクション (140 ～ 162) により、データベース A が更新されることはありません。これは、B から A への更新には、同期化機能の一部である特別な機能が使用されるためです。これは、以下のように実行されます。

1. データベース内の各トランザクションには、元のデータベースを示すラベルが付けられます。この例では、データベース B のトランザクション 140 にはデータベース A が元のデータベースであることを示すマークが付けられ、トランザクション 136 にはデータベース B が元のデータベースであることを示すマークが付けられています。
2. 同期化のための一連のトランザクションを 1 つにまとめた SyncSet.AddTransactions() メソッドでは、特定のデータベースが元となっているトランザクションを除外できます。したがって、B から A に更新する場合、データベース A が元になっているすべてのトランザクションは、既にデータベース B のトランザクション・リストに追加されているために、AddTransactions() によって除外されます。

この機能によって、2 つのデータベースが、互いに同じセットのトランザクションを更新し続けるという無限ループを防止できます。

D.2 同期をサポートするためのクラスの変更

オブジェクトの同期化には、サイト間に同じ GUID のデータがある必要があります。既存のデータベースを使用する場合でそのレコードに GUID が割り当てられていないとき、そのデータベース内の各インスタンス (レコード) に GUID を割り当て、各サイトにそのデータベースと一致するコピーがあるようにします。詳細なプロセスは以下のとおりです。

1. 同期対象のクラスごとに、OBJJOURNAL パラメータの値を 1 に設定します。

   Class Member

   ```
   Parameter OBJJOURNAL = 1;
   ```

   これにより、トランザクションごとのファイリング操作 (挿入、更新または削除) のログ機能を有効化します。この情報は、^OBJ.JournalT グローバルに格納されます。OBJJOURNAL の値に 1 を指定しておくと、ファイリング操作で変更したプロパティ値はシステムのジャーナル・ファイルに格納されます。同期化の際は、同期化を必要とするデータがそのファイルから取得されます。

   注釈　OBJJOURNAL では、値を 2 に設定することもできます。ただし、この値は特別な場合に限り使用可能です。既定のストレージ・メカニズムを使用するクラス (%Storage.Persistent) に対して使用することはできません。この値が 2 の場合、ファイリング操作で変更したプロパティ値は ^OBJ.Journal グローバルに格納されます。同期化の際は、同期化を必要とするデータがそのグローバルから取得されます。また、グローバルに情報を保存すると、データベースのサイズが急速に増加します。

2. オプションで、JOURNALSTREAM パラメータの値を 1 に設定します。

   Class Member

   ```
   Parameter JOURNALSTREAM = 1;
   ```

   既定で、オブジェクトの同期化は、ファイル・ストリームの同期化をサポートしません。JOURNALSTREAM パラメータは、OBJJOURNAL が真のときに、ストリームがジャーナル化されるかどうかを制御します。

   - JOURNALSTREAM が偽で、OBJJOURNAL が真の場合、オブジェクトはジャーナル化されますが、ストリームはジャーナル化されません。
   - JOURNALSTREAM が真で、OBJJOURNAL が真の場合、ストリームはジャーナル化されます。参照オブジェクトが処理されるときに、オブジェクト同期化ツールが、ジャーナル化されたストリームを処理します。

3. 同期化する各クラスの GUIDENABLED パラメータを 1 に設定します。これによって、InterSystems IRIS によってクラスに GUID を付けて格納することが許可されます。

   Class Member

   ```
   Parameter GUIDENABLED = 1;
   ```

   この値が設定されていない場合、同期化が正確に機能しないことに注意してください。また、シリアル・クラスに対しては GUIDENABLED を設定する必要がありますが、埋め込みオブジェクトには設定しないでください。

4. クラスをリコンパイルします。

5. 同期化する各クラスに対して、AssignGUID() メソッドを実行して、各オブジェクト・インスタンスにその GUID を割り当てます。

   ObjectScript

   ```
   Set Status = ##Class(%Library.GUID).AssignGUID(className,displayOutput)
   ```

   各項目の内容は次のとおりです。

   - className は、GUID (`"Sample.Person"` など) が割り当てられているインスタンスを含むクラスの名前です。
   - displayOutput は、整数です。ゼロは、出力が表示されないことを示し、ゼロ以外の値は出力が表示されることを示します。

   このメソッドは %Status 値を返すので、その値を確認する必要があります。

6. 各サイトにデータベースのコピーを配置します。

D.3 同期の実行

このセクションでは、同期を実行する方法について説明します。更新を提供するデータベースはソース・データベースと呼ばれ、更新を受け取るデータベースはターゲット・データベースと呼ばれます。実際の同期化の方法は、以下のとおりです。

1. 2 つのデータベースを同期化するたびに、ソース・データベースのインスタンスに移動します。ソース・データベースで、**%SYNC.SyncSet** クラスの %New() メソッドを使用して新しい SyncSet を作成します。

   ObjectScript

   ```
   Set SrcSyncSet = ##class(%SYNC.SyncSet).%New("unique_value")
   ```

   %New() の整数引数である unique_value は、簡単に識別できる一意の値である必要があります。これにより、各サイトのトランザクション・ログに追加したものが区別可能になります。

2. **SyncSet** インスタンスの AddTransactions() メソッドを呼び出します。

   ObjectScript

   ```
   Do SrcSyncSet.AddTransactions(FirstTransaction,LastTransaction,ExcludedDB)
   ```

   以下はその説明です。

   - FirstTransaction は同期化する最初のトランザクション番号です。
   - LastTransaction は同期化する最後のトランザクション番号です。
   - ExcludedDB は、トランザクションが SyncSet から除外されるデータベース内のネームスペースを示します。

   このメソッドは、同期データを収集して、それをグローバルに格納し、エクスポートできるようにします。

   または、前回の同期以降のトランザクションをすべて同期するために、1 番目と 2 番目の引数を省略します。

   ObjectScript

   ```
   Do SrcSyncSet.AddTransactions(,,ExcludedDB)
   ```

   これによって、最初の同期化されていないトランザクションから最新のトランザクションまでのすべてのトランザクションが取得されます。このメソッドでは、**SyncTime** テーブルの情報を使用して値が決定されます。

   ExcludedDB は $LIST です。以下のようにして作成します。

   ObjectScript

   ```
   Set ExcludedDB = $ListBuild(GUID,namespace)
   ```

   以下はその説明です。

   - GUID は、ターゲット・システム上のシステム GUID です。この値は、%SYS.System.InstanceGUID クラス・メソッドを通じて使用できます。このメソッドを呼び出すには、`##class(%SYS.System).InstanceGUID()` 構文を使用します。
   - namespace は、ターゲット・システム上のネームスペースです。

3. ErrCount() メソッドを呼び出して、発生したエラー数を確認します。エラーが発生していた場合、SyncSet.Errors クエリを実行すると、詳細情報が得られます。
4. ExportFile() メソッドを使用して、そのデータをローカル・ファイルにエクスポートします。

   ObjectScript

   ```
   Do SrcSyncSet.ExportFile(file,displaymode,bUpdate)
   ```

   以下はその説明です。

   - file は、トランザクションのエクスポート先のファイルで、名前には相対パスまたは絶対パスが含まれています。
   - displaymode では、このメソッドが現在のデバイスに出力を書き込むかどうかを指定します。出力する場合は d を指定します。出力しない場合は -d を指定します。
   - bUpdate は、SyncTime テーブルが更新されるかどうかを指定するブーリアン値です (既定値は True を意味する 1 です)。この時点ではこれを明示的に 0 に設定し、ターゲットが実際にデータを受信し、同期化が実行されたことの確認をソースが受信した後に、これを 1 に設定すると便利です。
5. エクスポートしたファイルを、ソース・マシンからターゲット・マシンに移動します。
6. SyncSet.%New() メソッドを使用して、ターゲット・マシンに SyncSet オブジェクトを作成します。%New() の引数に、ソース・マシンと同じ値を使用します。これは、同期化されるトランザクションのソースを識別する値です。
7. Import() メソッドを使用して、SyncSet オブジェクトを、ターゲット・マシンの InterSystems IRIS インスタンスに読み込みます。

   ObjectScript

   ```
   Set Status = TargetSyncSet.Import(file,lastSync,maxTS,displaymode,errorlog,diag)
   ```

   以下はその説明です。

   - file はインポートするデータを含むファイルです。
   - lastSync は、最後に同期化されたトランザクション番号 (既定値は synctime テーブルから取得) です。
   - maxTS は、SyncSet オブジェクトの最後のトランザクション番号です。
   - displaymode では、このメソッドが現在のデバイスに出力を書き込むかどうかを指定します。出力する場合は d を指定します。出力しない場合は -d を指定します。
   - errorlog は、エラー情報のリポジトリです (アプリケーションに情報を提供するために参照によって呼び出されます)。
   - diag は、インポート時に発生した詳細な診断情報です。

   このメソッドにより、データがターゲット・データベースに格納されます。これは、以下のように動作します。

   a. 最後の同期化以降、ソース・データベースとターゲット・データベースの両方でオブジェクトが変更されたことをメソッドが検出すると、%ResolveConcurrencyConflict() コールバック・メソッドが呼び出されます。他のコールバック・メソッドと同様、%ResolveConcurrencyConflict() のコンテンツはユーザが提供します (この 2 つの変更が共通のプロパティに対して行われた場合、またはこの 2 つの変更がそれぞれ接点のないプロパティのセットに対して行われた場合のいずれかに、このコールバック・メソッドが呼び出されます)。%ResolveConcurrencyConflict() メソッドが実装されない限り、この競合は解決されません。

   b. Import() メソッドの実行後、解決できない競合がある場合、これらは未解決の項目として SyncSet オブジェクトに残ります。残っている競合には、適切に対処してください。例えば、競合を解決する、その項目を未解決の状態にしておくなどの処置が考えられます。

重要　Import() メソッドはステータス値を返しますが、そのステータス値は、SyncSet の処理を妨げるエラーが発生せずにこのメソッドが完了したことを示すだけです。SyncSet 内の各オブジェクトがエラーの発生なしに正常に処理されたことを示すわけではありません。同期エラー報告の詳細は、%SYNC.SyncSet のクラス・リファレンスの "Import()" を参照してください。

8. 最初のデータベースが 2 番目のデータベースを更新したら、2 番目のデータベースが最初のデータベースを更新できるように、2 番目のデータベースから最初のデータベースに対して同じプロセスを実行します。

D.4 GUID と OID との変換

オブジェクトの OID から、そのオブジェクトの GUID を判断する (またはその逆を判断する) ために、以下の 2 つのメソッドを使用できます。

- %GUIDFind() は、**%GUID** クラスのクラス・メソッドで、オブジェクト・インスタンスの GUID を取得して、そのインスタンスに関連付けられた OID を返します。
- %GUID() は、**%Persistent** クラスのクラス・メソッドであり、オブジェクト・インスタンスの OID を取得して、そのインスタンスに関連付けられた GUID を返します。このメソッドは、対応するクラスの GUIDENABLED パラメータが TRUE の場合にのみ実行できます。このメソッドは多様な形式でディスパッチし、OID にクラス情報が含まれない場合、最も適切なタイプのクラスを決定します。インスタンスに GUID が割り当てられていない場合は空文字列を返します。

D.5 SyncTime テーブルの手動更新

データベースの SyncTime テーブルを手動で更新するには、最後のトランザクション番号を設定する SetlTrn() メソッドを呼び出します。

ObjectScript

```
Set Status=##class(%SYNC.SyncTime).SetlTrn(syncSYSID, syncNSID, ltrn)
```

以下はその説明です。

- syncSYSID は、ターゲット・システム上のシステム GUID です。この値は、%SYS.System.InstanceGUID クラス・メソッドを通じて使用できます。このメソッドを呼び出すには、`##class(%SYS.System).InstanceGUID()` 構文を使用します。
- syncNSID は、ターゲット・システム上のネームスペースであり、$Namespace 変数に保持されています。
- ltrn は、インポートされたことがわかっているトランザクション番号の最大値です。この値は、SyncSet の GetLastTransaction() メソッドを呼び出すことで取得できます。

SetlTrn() メソッドは、既定の動作 (ソース・システムからエクスポートされたトランザクションの番号の最大値を設定) とは異なって、ターゲット・システム上の同期化されたトランザクションの番号の最大値を設定します。どちらの方法を使用しても問題ありません。アプリケーションの開発中にいずれかを選択できます。