

付録 データプラットフォームの体験



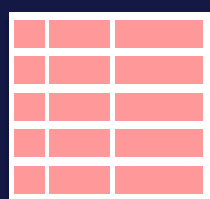
この資料の主な目標

- この資料では、InterSystems IRIS(またはIRIS for Health)で開発を行うために必要な基礎知識を操作を交えながら習得いただくため、以下の内容をご説明します。
 - データ基盤の概要
 - IRISへのアクセス
 - クラス定義の作成
 - オブジェクト操作によるデータ登録
 - SQLからのデータ参照／更新
 - レコードデータ／オブジェクト／グローバル変数の関係
 - メソッドの作成(クラスメソッド／ストアードプロシージャ)
 - テーブル定義／クラス定義／ルーチンの関係

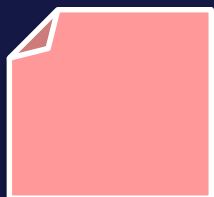


インターシステムズのデータプラットフォーム

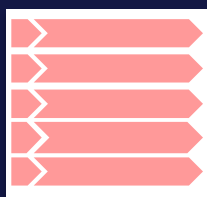
- 高性能で、スケーラビリティが非常に高く、安全性も高い
マルチモデルデータをサポートできるデータプラットフォームです。
 - 弊社全ての製品 (InterSystems IRIS / Caché / Ensemble / HealthShare) は、Cachéを基盤として開発されているため、データプラットフォームの基本操作は全製品共通です。
 - InterSystems IRISは、Cachéに含まれていない機能 (シャーディング、クラウドデプロイ用ツール) や、EnsembleやHealthShareの持つ相互運用性も含めた、包括的なデータプラットフォームです。
 - オブジェクト、リレーショナル、多次元、XMLを完全にサポートしています。



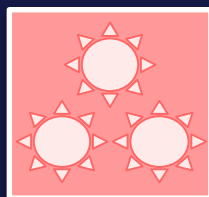
Table



Document

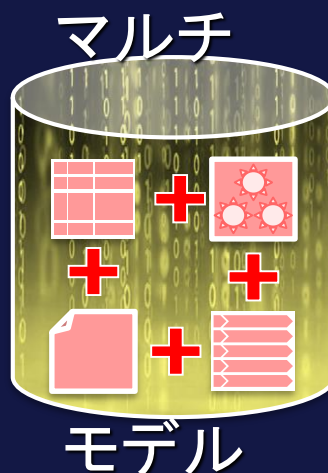


Global



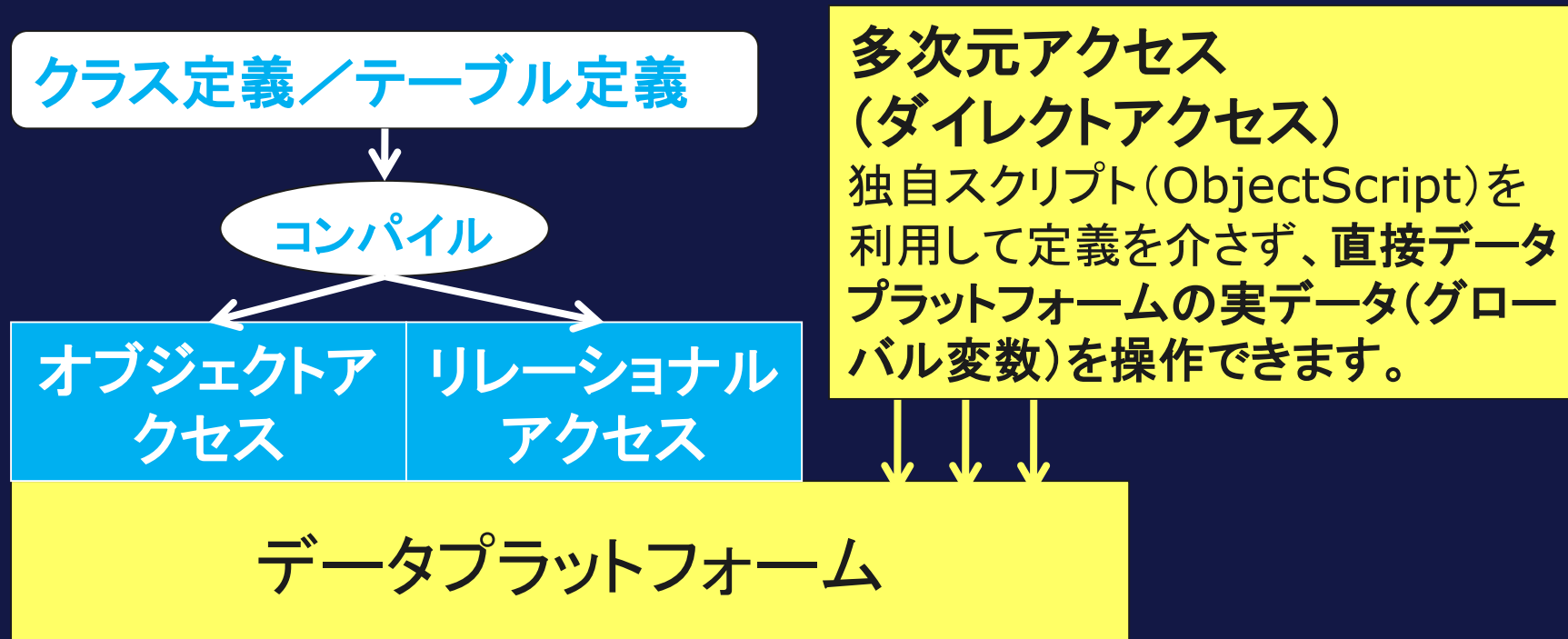
Object

...



データプラットフォームへのアクセス

- データプラットフォームにアクセスするために様々なツール、言語、方法を選択できます。
- シンプルに記述すると以下の図のようなアクセスがあります。



クラスとは？

- 最も簡単なクラスは、メソッドだけを定義するメソッドのコンテナとしたクラスです。
 - プロパティを持たないクラスでロジック(メソッド)だけを定義します。
 - アプリケーションロジックのユーティリティコードや、ストアドプロシージャ用ロジックを定義するクラスとして使用します。
- メソッドの他にデータ要素であるプロパティを定義するクラスもあります。
 - データベースに格納する機能を持つクラス(Persistent＝永続クラス)
 - 演習で作成します。
 - データベースに格納しないクラス(Registered＝メモリ上にインスタンスは作成できるけど、データベースには保存できないクラス)
 - 永続クラス(Persistent)のあるプロパティに埋め込みことが前提のクラス(Serial＝埋め込みクラス)
- メソッドは、特定の動作を実行するためのコードです。
 - メソッドコード中で使用する引数や変数は、プライベートスコープです。
- スタジオを使用して、クラスを作成します。
- メソッドの記述には、ObjectScript を使用します。



オブジェクト

- オブジェクトは、クラスのプロパティに特定のデータをセットしたクラス定義から構築される実体です。
 - オブジェクトの雛形がクラス定義とも表現できます。
- IRISのオブジェクトは、メモリ上でしか動作できないオブジェクト(Registered)とデータベースに保存できるオブジェクト(Persistent=永続オブジェクト)を作成できます。
 - クラス作成時「クラスタイプ」の項目で指定します。
- 永続オブジェクトは、データベースの新規保存時、ユニークなID番号が割り振られます。
 - テーブルのレコードID(ROWID)と同等です。

山田太郎
yamada@mail.com

鈴木次郎
suzuki@mail.com

Training.Person

Name As %String
Email As %String

CreateEmail()

プロパティとは

- プロパティは、オブジェクトのデータ要素がどのような値であるかを定義する設定項目です。

- 定義文は以下の通りです。

Property Name As %String;

%Stringの正式名称は
%Library.String クラスで、
データタイプもクラス定義として実装されています(データを持たないタイプで定義されています)。

- As の後ろにデータタイプやそのプロパティを表現するためのクラス名を指定できます。
 - データタイプの例: %String、%Integer、%Dateなど
 - オブジェクト参照を行う場合は、参照先クラス名をAsの後ろの指定します。
 - %Stringに独自のチェック項目をつけたしたユーザ定義のデータタイプも指定できます。その場合は、作成したデータタイプクラス名を指定します。
- (ご参考)リスト、リレーションシップ定義など、このほかにもオブジェクトよりの定義も実装できます。



メソッドとは

- メソッドはクラスに関連したロジックを記述できる定義です。
 - ObjectScript を使用して記述します。
 - 2種類あります。
 - インスタンスメソッド
 - 実行時、メモリ上にメソッド実行対象であるインスタンスが存在していないと実行できないメソッド
例) オブジェクトを保存するときに使用する%Save()など
 - クラスメソッド
 - **##CLASS()**の文法を使用し、メモリ上のインスタンスの有無にかかわらずいつでも実行できるメソッド
 - SqlProc属性をTrueにすることでストアードプロシージャ化できるメソッド
 - メソッド内に記載するサーバ側コードには、オブジェクト／SQL／ダイレクト、全ての文法が記述できます。
 - ローカル変数のスコープは、ローカル変数を使用しているメソッド内のみです。



パッケージとは

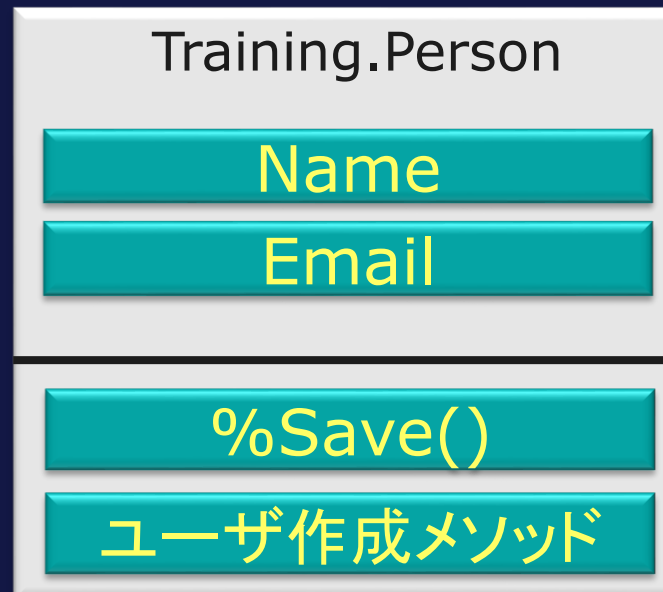
- パッケージはクラス定義にとってフォルダの役割です。
 - ネームスペース内のクラス定義をひとまとめにします。
- クラスの完全名は、パッケージ名.クラス名 です。
 - 大文字小文字の区別のある文字または、文字列で始まる数字を指定します。
- %付きパッケージは、InterSystemsが提供するシステムクラスを含むパッケージです。
 - %付きパッケージは、任意のネームスペースから自動的に利用可能です。
 - 様々な機能を持ったクラスが提供されていてクラスリファレンスやスタジオを使用して定義を確認できます。
- %Libraryパッケージは、クラスの“構築基盤”として使われるクラスを含んでいます。
 - %Library.Persistentや%Library.RegisterdObject、データタイプとして利用する%Library.Stringや%Library.Date
 - **%Library.** を省略でき、%Persistent や%String のように記述できます

クラスリファレンスの開き方
スタジオ→表示→クラスドキュメントの表示



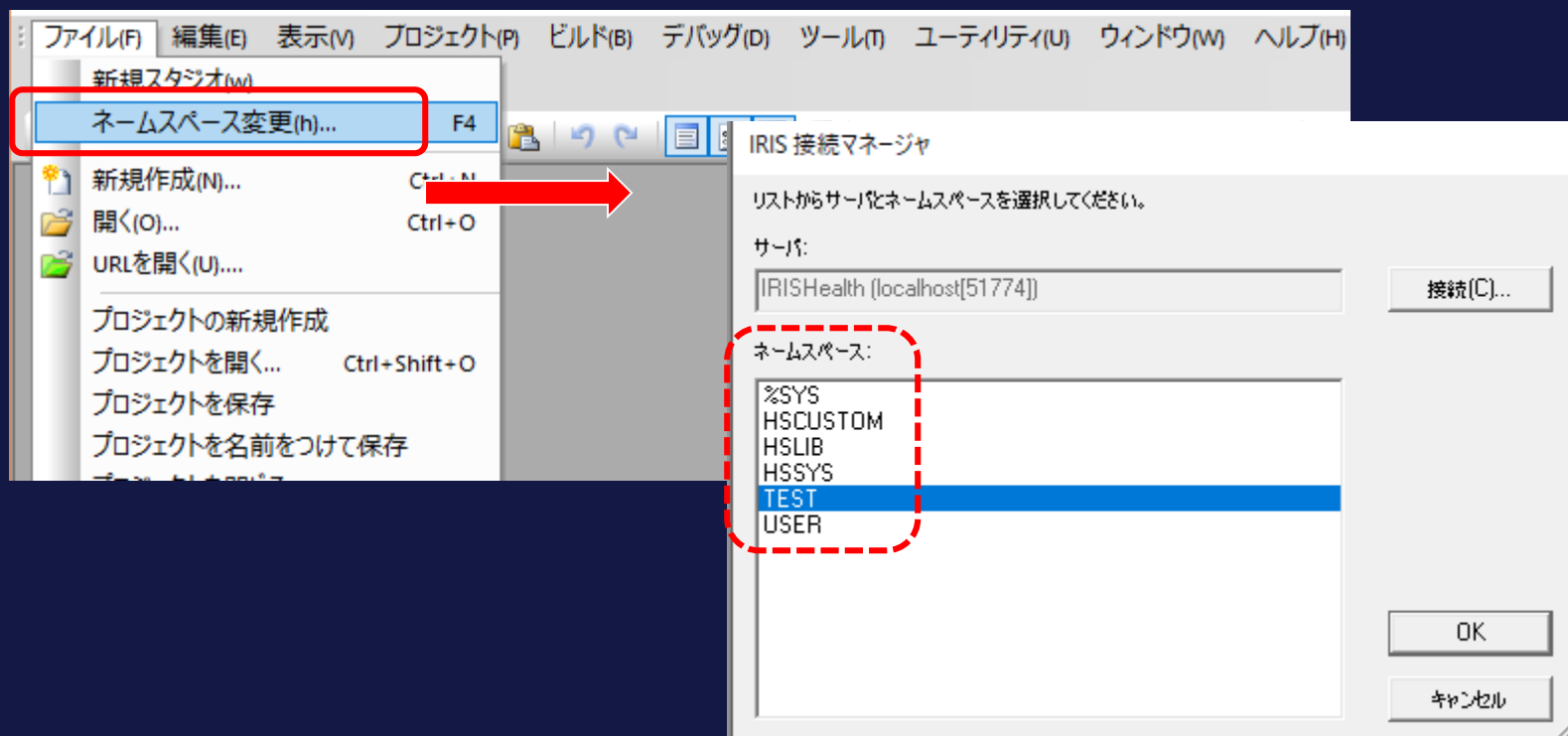
Training.Personクラスは 永続(Persistent)クラス

- 永続クラスとは、データベースに保存する機能を持ったクラスです。
 - データベースに保存されると、ユニークなID番号が付与されます。
- 永続クラス(クラスタイプ: Persistentを選択したクラス)は、%Library.Persistentクラスを継承します。



クラス定義作成のための準備

- スタジオを開き、操作対象のネームスペースに移動します。
 - ファイル→ネームスペース変更



演習: Training.Personクラスの作成

- スタジオの **ファイル→新規作成→Cacheクラス定義** からクラスを作成します。

The screenshot shows the Studio IDE interface with the 'New Class Wizard' dialog open. The 'File' menu is open, and 'New' is selected. The 'New Class Wizard' dialog has two tabs: 'New Class Wizard' and 'Class Wizard'. The 'New Class Wizard' tab is active, showing the 'Package Name' and 'Class Name' fields. The 'Package Name' is 'Training' and the 'Class Name' is 'Person'. The 'Class Wizard' tab is also visible, showing the 'Class Type' section with '永続 (データベースに格納可能)' selected. The 'Extends' checkbox is checked, and the 'Superclass Name' field is empty. Red arrows and callouts provide instructions for each step.

IRISHealth/TEST@_SYSTEM - Default_system.prj - スタジオ

ファイル(F) 編集(E) 表示(M) プロジェクト(P) ビルド(B) デバッグ(D) ツー

新規スタジオ(w)
ネームスペース変更(h)... F4

新規作成(N)... Ctrl+N

開く(O)... Ctrl+O

URLを開く(U)....

新規作成

カテゴリ:

- 一般
- CSPファイル
- プロダクション
- メッセージ
- Zen
- カスタム

テンプレート:

- Cache ObjectScript ルーチン
- Cache クラス定義
- Cache Basic ルーチン
- MV

新規クラスウィザード

新規クラスウィザードへようこそ。
このウィザードは、新しいCacheクラスを作成するための手順に従い、次のページにいつでも「完了」を選択することができます。

パッケージ名を入力:
Training

クラス名を入力:
Person

新しいクラスの説明を入力してください(オプション):
このエリアには任意で説明文を記入できます。
後でクラスドキュメント(クラスリファレンス)画面で参照できます。

クラスウィザード

クラスタイプ

データベースに保存する「永続」を選択

どの種類のクラスを作成しますか?以下のクラスタイプを選択して下さい:

- ☒ 永続 (データベースに格納可能)
- ☐ Serial (永続オブジェクト内に埋め込むことが可能)
- ☐ Registered (データベース内には格納されない)
- ☐ 抽象
- ☐ データタイプ
- ☐ CSP (HTTPイベントの処理に使用)
- ☐ Extends

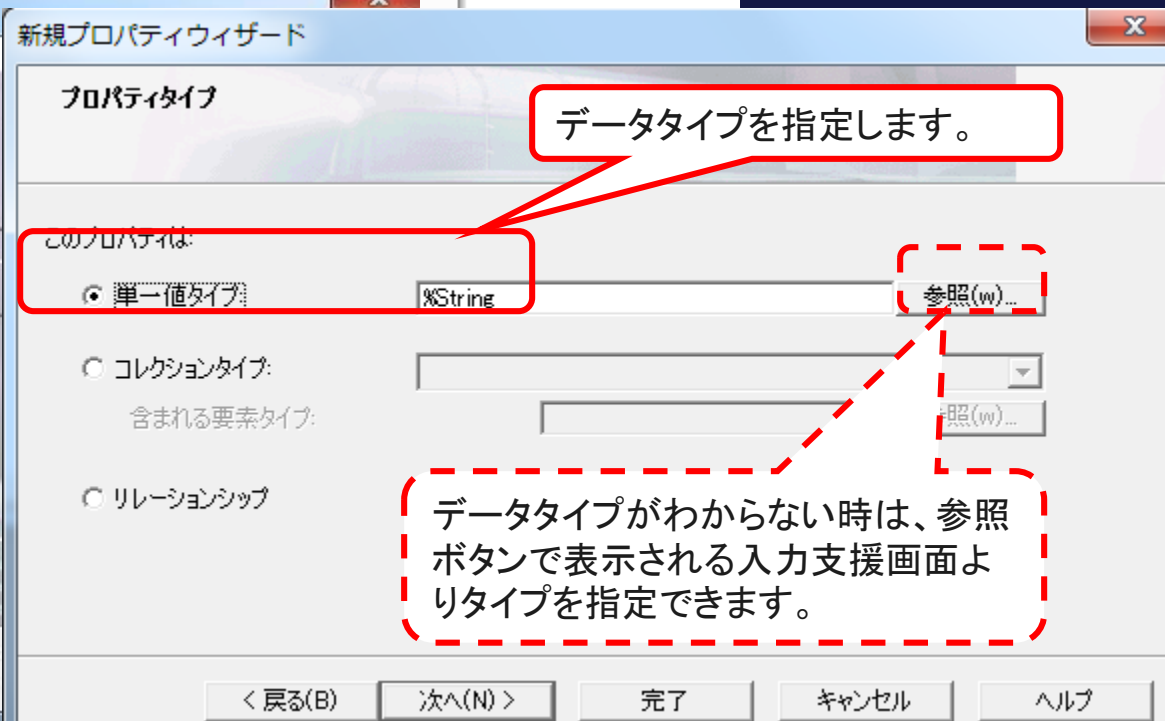
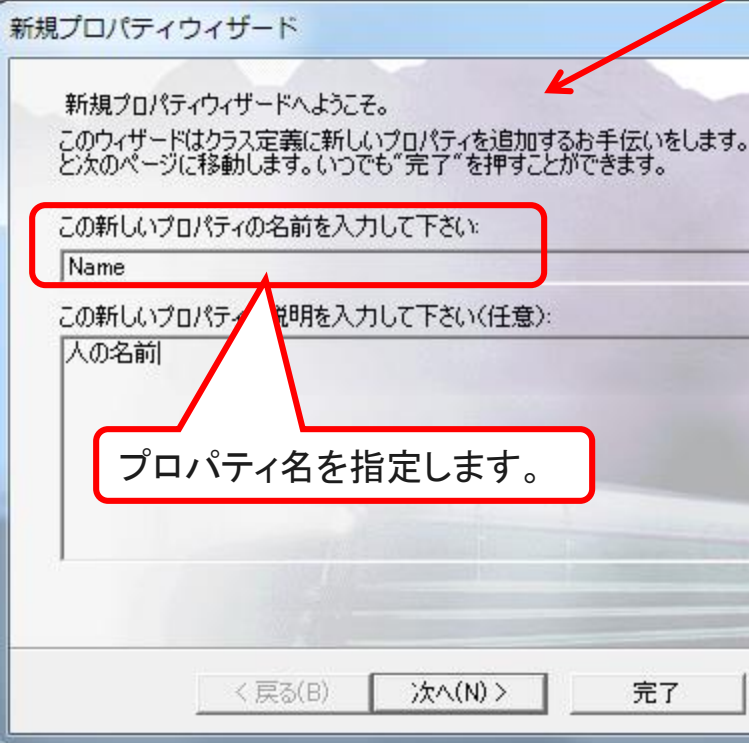
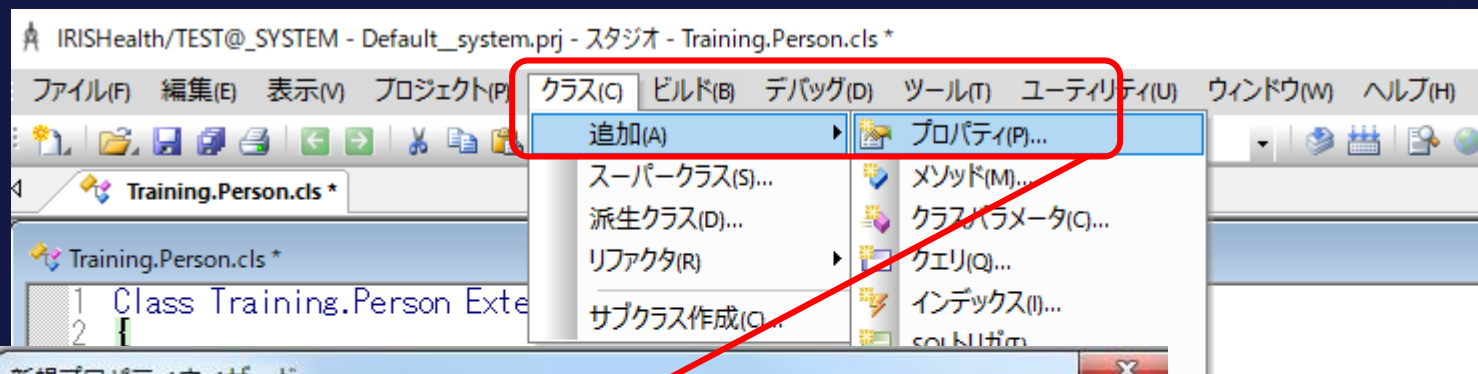
Extendsにチェックを入れるとスーパークラスを指定できます。

スーパークラス名:

< 戻る(B) 次へ(N) > 完了 キャンセル

演習: プロパティの作成

- クラス定義を開いた状態でクラス→追加→プロパティを選択します。



クラス定義(完成)

```
IRISHealth/TEST@_SYSTEM - Default_system.prj - スタジオ - [Training.Person.cls]
ファイル(F) 編集(E) 表示(V) プロジェクト(P) クラス(C) ビルド(B) デバッグ(D) ツール(T) ユーティリティ(U) ウィンドウ(W) ヘルプ(H)
Training.Person.cls
1 Class Training.Person Extends %Persistent
2 {
3
4   /// 人の名前
5   Property Name As %String;
6
7   /// メールアドレス
8   Property Email As %String;
9
10  Storage Default
11  {
12    <Data name="PersonDefaultData">
13      <Value name="1">
14        <Value>%%CLASSNAME</Value>
15      </Value>
16      <Value name="2">
17        <Value>Name</Value>
18      </Value>
19      <Value name="3">
20        <Value>Email</Value>
21      </Value>
22    </Data>
23    <DataLocation>^Training.PersonD</DataLocation>
24    <DefaultData>PersonDefaultData</DefaultData>
25    <IdLocation>^Training.PersonD</IdLocation>
26    <IndexLocation>^Training.PersonI</IndexLocation>
27    <StreamLocation>^Training.PersonS</StreamLocation>
28    <Type>%Storage.Persistent</Type>
29  }
30
31 }
```



クラスを利用したプログラミング

- オブジェクト操作には **##class** 構文を使用します。

- 新規インスタンス生成

```
Set obj=##class(パッケージ名.クラス名).%New()
```

- 既存オブジェクトオープン

```
Set obj=##class(パッケージ名.クラス名).%OpenId(id)
```

- クラスメソッドの実行

```
Set modori=##class(パッケージ名.クラス名).メソッド名(引数n)
```

- インスタンスメソッドの実行

```
Set modori=obj.メソッド名(引数n)
```

- オブジェクトのメモリからの解放

```
Set obj="" または kill obj
```

- プロパティへのアクセス

```
Write obj.プロパティ名
```



永続メソッド

- 永続クラスに用意されている主なメソッドは以下の通りです。

メソッド	目的	成功時	失敗時
%New()	メモリー上に新規オブジェクトを作成	object	""
%Save()	オブジェクトを保存	1	エラーステータス
%Id()	object IDを返す	object ID	""
%OpenId(<i>id</i>)	保存済みオブジェクトを取得し、メモリーに展開する	object	""



ご参考: 永続メソッド(つづき)

メソッド	目的	成功時	失敗時
%DeleteId(<i>id</i>)	保存済みオブジェクトを削除	1	エラー ステータス
%DeleteExtent()	全ての保存済みオブジェクト(とその関連オブジェクト)を削除	1	エラー ステータス
%KillExtent() (開発中のみ使用)	全ての保存済みオブジェクトを削除	1	""
%ClassName(1)	オブジェクトの完全クラス名を表示	クラス名	



ご参考:

ObjectScriptの変数について

- サーバ側ロジックの記述に使用するObjectScriptの変数(ローカル／グローバル両方)は 型がなく(タイプレス)、変数タイプを宣言する必要がありません。
 - 内部的には、数字か文字列かのどちらかで保持されます。
 - 動的で弱い型に分類されます。
 - 変数自身は、タイプレスですが、クラス定義のプロパティには、データタイプを指定する必要があり、一般的なデータ型も含めて内部に用意されたデータタイプクラスを使用して定義します。



ご参考： ObjectScript よく使うコマンド

コマンド	内容	例文
SET	変数に値を割り当てる	Set val="あいうえお" 文字列は二重引用符で括ります。(SQL文では一重引用符です。)
WRITE	標準出力	Write val,! ターミナルでは！は改行を出力します。 引数無のWRITEはメモリ上の全変数の出力
KILL	変数の消去	Kill val 指定した変数を削除します。 引数無のKILLはメモリ上の全変数の消去
QUIT	プログラムの終了、FOR、WHILEのループの終了	QUIT retunVal 戻り値がある場合には引数を指定します。
DO	メソッド、プロシージャの実行	Do ##class(Training.Person).Method() クラスメソッドの実行
ZWRITE	Writeと似ているターミナル用コマンド	オブジェクトリファレンス、配列の全添え字情報出力ができる便利コマンド(デバッグ用)



演習:

Training.Personのオブジェクト作成

- オブジェクト生成／解放、データベースへの登録／オープンの流れは以下の通りです。

ユーザプロセス

指定のクラスに対する
メモリ上のインスタンス生成

↓
プロパティに値をセットする

↓
オブジェクトを保存
(永続化)

↓
オブジェクト
の解放

↓
オブジェクトのオープン

%New()メソッド実行

%Save()メソッド実行

Killコマンドでオブジェクト削除

%OpenId()メソッド実行

データベース



データの作成例(オブジェクト)

```
USER>set $namespace="TEST" // TESTネームスペースに移動

TEST>set person=# #class(Training.Person).%New() // インスタンス生成

TEST>set person.Name="山田太郎" // プロパティの設定

TEST>set person.Email="yamada@mail.com"

TEST>set st=person.%Save() // 保存

TEST>write st // 結果の確認: 1なら成功
1
TEST>kill person // メモリ内オブジェクトの削除

TEST>set person=# #class(Training.Person).%OpenId(1) // データベースからロード

TEST>write person.Name // プロパティ値の参照
山田太郎
TEST>set person.Name="山田たろう" //プロパティ値の更新(この時点ではメモリ内のみの変更)

TEST>set st=person.%Save() // データベースへの保存

TEST>write st
1
TEST>
```



テーブル

- 永続オブジェクトは、リレーショナルテーブルでは、行になります。
 - オブジェクトid番号は、IDカラムです。
 - パッケージ名は、スキーマ名になります。
 - 一部例外で、*User*パッケージは、*SQLUser*スキーマになります。
- Training.Personテーブルでは以下の通りです。
 - Training.Personクラスのデータと同等です。

ID	Name	Email
1	山田太郎	taro@mail.com
2	鈴木一郎	ichiro@mail.com
3	佐々木小次郎	kojiro@mail.com



管理ポータル SQL画面の使い方 ネームスペースの切り替え

- 管理ポータル→システムエクスプローラ→SQL

The screenshot shows the InterSystems IRIS Data Platform management portal. The main interface displays the 'SQL' view with a sidebar containing links for 'テーブル' (Tables), 'ビュー' (Views), 'プロシージャ' (Procedures), and 'クエリキャッシュ' (Query Cache). A red box highlights the 'ネームスペース %SYS 変更' (Namespace %SYS Change) link. A red arrow points from this link to a 'ネームスペース選択' (Namespace Selection) dialog box. The dialog box shows a list of available namespaces: '%SYS', 'HSCUSTOM', 'HSLIB', 'HSSYS', 'TEST', and 'USER'. The 'TEST' namespace is highlighted with a red box. A red arrow points from this box to a text box explaining the next step.

InterSystems™
IRIS Data Platform

管理ポータル

サーバ 6f45803d364c

ネームスペース %SYS 変更 ユーザ _SYSTEM

接続先ネームスペースを確認します。

システム > SQL

フィルタ 適用先 すべて

システム ☐ スキーマ

> テーブル

> ビュー

> プロシージャ

> クエリキャッシュ

ネームスペース選択

ネームスペース選択

利用可能なネームスペース

%SYS
HSCUSTOM
HSLIB
HSSYS
TEST
USER

「変更」のリンクを押下し、ネームスペース一覧から接続先を切り替えます。

キャンセル OK

管理ポータル SQL画面の使い方 テーブルの操作

- 管理ポータル > システムエクスプローラ > SQL

The screenshot shows the SQL management portal interface. Red annotations highlight key features:

- ①** Points to the "スキーマ" (Schema) dropdown menu, which is currently set to "Training".
- ②** Points to the "Training.Person" table selected in the "テーブル" (Table) list.
- ③** Points to the "テーブルを開く" (Open Table) button in the top navigation bar.

The interface includes tabs for "カタログの詳細" (Catalog Details), "クエリ実行" (Query Execution), "参照" (Reference), and "SQLステートメント" (SQL Statements). The "クエリ実行" tab is active, showing a table view for "Training.Person".

Below the table view, there is a text area for SQL queries:

```
SELECT  
ID, Email, Name  
FROM Training.Person
```

A red dashed box highlights the "クエリ実行" tab and the SQL text area, with a note:

クエリ実行タブでは、任意のSQL文を実行できます。

The "テーブルを開く" (Open Table) dialog box is shown. It includes a "更新" (Update) button and a "ウインドウを閉じる" (Close Window) button. The dialog displays the name space "TEST 中の Training.Person" and the last update time "最終更新: 2021-11-17 16:46:10.131".

#	ID	Email	Name
1	1	yamada@majorcorp.com	山田たろう

完了

クラス定義／テーブル定義 言葉の対応

オブジェクト指向プログラミング (OOP)	構造化クエリー言語 (SQL)
パッケージ	スキーマ
クラス	テーブル
プロパティ	カラム
メソッド	ストアド・プロシジャ
2つのクラス間のリレーションシップ	外部キー制約、組み込みJOIN
オブジェクト（メモリー上とディスク上）	行（ディスク上）



データとグローバル変数

- データプラットフォームへのアクセス方法は3種類(SQL／オブジェクト／ダイレクト)ありますが、実体のデータは「グローバル変数」と呼ばれる永続多次元配列で格納されます。
 - SQLの場合はテーブルとして操作しているデータも、グローバル変数として格納されます。
- オブジェクト／リレーショナルアクセスでは、クラス／テーブル定義に合わせ、初回コンパイル時に格納するグローバル変数の構造を決定しています。
 - ストレージ定義を初回コンパイル時に自動的に作成します。

コンパイルにより自動生成

SQL
オブジェクト

生成コード

クラス／テーブル定義

ストレージ定義

グローバル変数

コンパイルにより自動生成

ダイレクト

データの参照・更新＝グローバル変数の参照・更新

どのアクセス方法で操作してもデータの実体はグローバル変数と呼ばれる永続多次元配列に格納されます。

補足:オブジェクト／レコードとストレージ定義

```
1 Class Training.Person Extends (%Persistent, %XML.Adaptor)
2 {
3     // 人の名前
4     Property Name As %String;
5     // メールアドレス
6     Property Email As %String;
7     // Nameプロパティに対するインデックス
8     Index NameIndex On Name;
9     // Emailアドレスを作成するクラスメソッド<br>
10    // SQLのINSERT文で利用する例<br>
11    // insert into Training.Person (Name,Email) values(
12    ClassMethod CreateEmail(account As %String) As %String
13    {
14        // 引数が指定されなかったら空を返す
15        if $get(account)="" quit
16        quit account_"@mail.com"
17    }
18
19    Training.PersonD=1
20    Training.PersonD(1)=$lb("山田","yamada@mail.com")
21    Training.PersonD(2)=$lb("鈴木","suzuki@mail.com")
22    Training.PersonD(3)=$lb("佐藤","sato@mail.com")
23    Training.PersonD(4)=$lb("川田","kawata@mail.com")
24    Training.PersonD(5)=$lb("斉藤","saito@mail.com")
25    Training.PersonD(6)=$lb("川上","kawakami@mail.com")
26    Training.PersonD(7)=$lb("山本","yama@mail.com")
27    Training.PersonD(8)=$lb("佐々木","sasaki@mail.com")
28    Training.PersonD(9)=$lb("田中","tanaka@mail.com")
29    Training.PersonD(10)=$lb("田原","tahara@mail.com")
30
31    Storage Default
32    {
33        <Data name="PersonDefaultData">
34            <Value name="1">
35                <Value>%%CLASSNAME</Value>
36            </Value>
37            <Value name="2">
38                <Value>Name</Value>
39            </Value>
40            <Value name="3">
41                <Value>Email</Value>
42            </Value>
43        </Data>
44        <DataLocation>^Training.PersonD</DataLocation>
45        <DefaultData>PersonDefaultData</DefaultData>
46        <IdLocation>^Training.PersonD</IdLocation>
47        <IndexLocation>^Training.PersonI</IndexLocation>
48        <StreamLocation>^Training.PersonS</StreamLocation>
49        <Type>%Library.CacheStorage</Type>
50    }
```

ID=1のデータを参照

```
>write $LIST(^Training.PersonD(1),2)
山田
>write $LIST(^Training.PersonD(1),3)
yamada@mail.com
```

27 | データ ID=1のデータを更新 set \$LIST(^Training.PersonD(1),2)="やまだたろう"

ここからは・・・

- 独自スクリプト「ObjectScript」を使用したプログラミングを体験します。
- ObjectScriptは、クラス／テーブル定義を作成した時点で自動生成されるコードに使用されていますが、それ以外にも、ビジネスロジックの記述に利用できます。
- 以降のページでは、Training.Personクラスにクラスメソッドを追加しながら、スクリプトの記述方法を練習します。



ObjectScriptとは

- インターシステムズ製品の独自スクリプトです。
 - ANSI/JISで標準化された言語をベースとし、完成度としては非常に高い言語です。(Mumps言語と互換性があります。)
 - データ操作以外にも一般言語と同等の操作ができます。
- スクリプトのほとんど全てに大文字小文字の区別があります。
 - 変数、メソッド、ルーチン、パッケージ、クラスは大文字小文字の区別があります。
 - SQLは区別がありません。(テーブル名、カラム名、ストアド名)
 - コマンドは、大文字小文字の区別はありません。
 - クラスの操作に使用する `##class` 指示文は、大文字小文字の区別はありません。
- 最初のうちは、全てに大文字小文字の区別があると考えておきましょう。



まずはターミナルで Hello world !

- ObjectScriptは、ターミナルでインタラクティブにコマンド実行とその結果の確認が行えます。
- ターミナルを開き、WRITEコマンドを利用して、文字列「Hello world!」を出力してみましょう。

```
USER>write "Hello world!"  
Hello world!  
USER>
```

- 書き方:
 - コマンドに引数を指定するときはスペースを1つ入れます。
 - コマンドは大文字小文字どちらも対応できます。
 - 文字列は二重引用符で括ります。



スタジオでプログラミング

- ターミナルは、コマンド実行の確認だけでなく、作成したプログラムのテスト実行にも最適です。
- プログラムの記述はターミナルでは無理なので、スタジオで記述します。
- スタジオは統合開発環境のため作成する用途に合わせエディタが異なります。
- プログラミングはルーチン単位で作成することもできますが、ストアドプロシージャなど外部からの呼び出しに備え、本資料ではクラスメソッドの作成を中心にご説明します。
 - 既存ルーチンをストアドプロシージャやメソッドとして外部から実行したい場合は、クラスメソッドを作成し、その中からルーチンを呼び出すことで実装できます。



スクリプトの記述ルール

- ObjectScript の記述ルールは以下の通りです。
- メソッドにコードを追加する際、行頭にタブを挿入してからコマンドを書き始めます。
 - 行頭から書き始めた文字はラベルと認識されます。
- シンタックスは以下の通りです。
command argument1, argument2, ..., argumentN
 - コマンドと引数の間には、1つのスペースを記入します。
 - 複数の引数を持てるコマンドの場合、引数をカンマで区切ります。
 - Set, Do, Job, Lock など複数引数を持てるコマンドがありますが、基本は `command arg1, arg2` は `command arg1 command arg2` と同等です。
- スクリプトには **\$** から始まる様々な関数があります。
 - 文字列操作関数
 - `$piece()` : 区切り文字のデータを操作する関数
 - `$length()` : 文字の長さを調べる関数
 - `$extract()` : 指定個所の部分抽出を行う関数
 - 変数チェック用
 - `$Get()` : 変数が存在しない場合、空("")を返します。
 - `$Data()` : 変数の存在や配列変数に下位のノードが存在するかチェックできます。
 - レコードデータをダイレクトアクセスで操作するときに使用する関数
 - `$LIST()` / `$LISTBUILD()` / `$LISTLENGTH()`



ObjectScript

よく使うコマンド

コマンド	内容	例文
SET	変数に値を割り当てる	Set val="あいうえお" 文字列は二重引用符で括ります。(SQL文では一重引用符です。)
WRITE	標準出力	Write val,! ターミナルでは！は改行を出力します。 引数無のWRITEはメモリ上の全変数の出力
KILL	変数の消去	Kill val 指定した変数を削除します。 引数無のKILLはメモリ上の全変数の消去
QUIT	プログラムの終了、FOR、WHILEのループの終了	QUIT retunVal 戻り値がある場合には引数を指定します。 ※次のループへジャンプするにはCONTINUEを使います。
DO	メソッド、プロシージャの実行	Do ##class(Training.Visit).Method() クラスメソッドの実行
ZWRITE	Writeと似ている ターミナル用コマンド	オブジェクトリファレンス、配列の全添え字情報 出力ができる便利コマンド(デバッグ用)

コメントの記述方法

```
ClassMethod comment()
```

```
{
```

```
    // 1行コメント
```

```
    ; 1行コメント
```

```
    write 1,!
```

```
    /*
```

```
    複数行コメント
```

```
    */
```

```
    write 2,!
```

```
    #; 1行コメント+中間コードに記載されないコメント文
```

```
    write 3,!
```

```
}
```

```
870 zcomment() public {  
871     // 1行コメント  
872     ; 1行コメント  
873     write 1,!  
874     /*  
875     複数行コメント  
876     */  
877     write 2,!  
878     write 3,! }
```

生成された中間コード



ループの記述

- FORの記述は以下の通りです。

```
for カウンタ用変数=開始値:増(減)分値:終了値 {}  
for カウンタ用変数=開始値,増(減)分値,終了値 {}
```

- FORのオプション指定

- 終了値のみ指定なしでも、FOR文は増分または減分します。
- 引数を何も指定しない場合は、無限ループになります。
- 終了値の指定がない場合は、QUITでループを終了します。

- WHILEの記述は以下の通りです。

```
while (条件) { コード } または、do { コード } while (条件)
```

- do-whileは、1回は実行されます。

- FORとWHILE共通で、ある条件で次のループに移動する場合は、CONTINUEコマンドを使用します。



ループの記述 ContinueとQuitの違い

```
ClassMethod loopContinue()
```

```
{  
    for i=1:1:5 {  
        if i=3 {  
            write "i=3 のとき continue → ループをスキップ",!  
            continue  
        }  
        write "i=",i,!  
    }  
    write "メソッドの終わり",!  
}
```

```
USER>do ##class(Script.Sample).loopContinue()  
i=1  
i=2  
i=3 のとき continue → ループをスキップ  
i=4  
i=5  
メソッドの終わり
```

```
ClassMethod loopQuit()
```

```
{  
    for i=1:1:5 {  
        if i=3 {  
            write "i=3 のとき quit → ループを停止",!  
            quit  
        }  
        write "i=",i,!  
    }  
    write "メソッドの終わり",!  
}
```

```
USER>do ##class(Script.Sample).loopQuit()  
i=1  
i=2  
i=3 のとき quit → ループを停止  
メソッドの終わり
```

ループの記述 QuitとReturnの違い

ClassMethod loopQuit()

```
{  
    for i=1:1:5 {  
        if i=3 {  
            write "i=3 のとき quit → ループを停止",!  
            quit  
        }  
        write "i=",i,!  
    }  
    write "メソッドの終わり",!  
}
```

ObjectScript ではプログラムの終了にQuitまたはReturnを利用できますが動作が異なります (Returnは2013.1以降のバージョンで利用可)。

```
USER>do ##class(Script.Sample).loopQuit()  
i=1  
i=2  
i=3 のとき quit → ループを停止  
メソッドの終わり
```

ClassMethod loopReturn()

```
{  
    for i=1:1:5 {  
        if i=3 {  
            write "i=3 のとき return → メソッド停止",!  
            return  
        }  
        write "i=",i,!  
    }  
    write "メソッドの終わり",!  
}
```

```
USER>do ##class(Script.Sample).loopReturn()  
i=1  
i=2  
i=3 のとき return → メソッド停止
```



メソッドの作成

- クラス→追加→メソッド よりウィザードを起動します。

メソッド名を指定します。

メソッド名を指定します。

戻り値がある場合はタイプを指定します。

引数リストの設定

新規メソッドウィザード

メソッドシグニチャ

この新メソッドの戻りタイプ(任意)と引数リスト(任意)を入力

戻りタイプ(T): %String

引数リスト:

名前	タイプ	デフォルト値	渡し方
account	%String		Value

名前: account

タイプ: %String

デフォルト:

値 (選択済み) / 参照

OK / キャンセル(O)

参照(w)...

< 戻る(B) / 次へ(N) > / 完了

メソッドの作成 つづき

- クラスメソッドとして定義したメソッドは、ストアードプロシージャ化することができます。(SqlProc属性の利用)

新規メソッドウィザード

メソッド属性

このメソッドの追加の性質を選択することができます:

- ☐ Private このメソッドはこのクラスに対してprivateです
- ☐ Final このメソッドはfinalです
- ☒ クラスメソッド このメソッドはクラスメソッドです
- ☒ SQLストアードプロシージャ このメソッドはSQLストアードプロシージャです

言語:
cache

戻る(B) 次へ(N) > 完了

新規メソッドウィザード

インプリメンテーション

この新しいメソッドのソースコードを入力できます:

```
%String(account:%String)  
// 引数が指定されなかったら空を返す  
if $get(account) = "" quit ""
```

先頭文字にタブを1つ挿入してからコマンドの記述を開始します。

< 戻る(B) 次へ(N) > 完了 キャンセル ヘルプ

メソッドコードはスタジオの主画面で修正もできます。

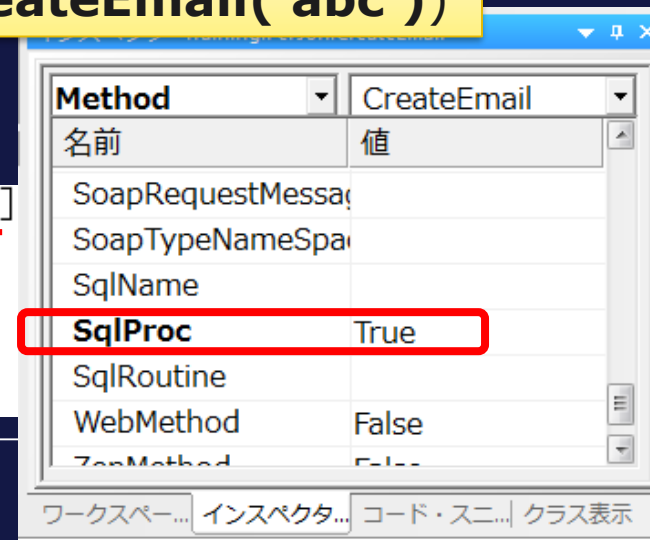
クラスメソッドにチェックを入れると「SQLストアードプロシージャ」にチェックができるようになります。

ストアドプロシージャ 基本

- ストアドプロシージャを作成したい場合は、クラスメソッドを作成し、**SqlProc属性をTrue**に設定します。
- 定義したストアドプロシージャは、以下命名規則で実行します。
スキーマ名.テーブル名_メソッド名
 - 別名に変更することもできます(メソッド定義のSqlName属性に別名を登録します)。
- ストアドプロシージャの呼び出しには、CALL文を使用します。
例) **call Training.Person_ProcName()**
- 結果セットを返さない場合、関数実行のようにSQL文中に指定できます。

```
insert into Training.Person (Name,Email)
values('適当な名前',Training.Person_CreateEmail('abc'))
```

```
16 ClassMethod CreateEmail(account As %String) As %String [ SqlProc ]
17 {
18     // 引数が指定されなかったら空を返す
19     if $get(account)="" quit
20     quit account_">@mail.com"
21 }
22
```



補足：外部I/Fからメソッドを実行したい！ という場合の一例

- データプラットフォームに定義したクラスメソッドは、SQLベースの外部I/Fからでは、直接実行することができません。
- 演習で体験したように「ストアードプロシージャ」属性をクラスメソッドに付与することで、外部I/Fに対してストアードプロシージャとして公開され、どこからでも呼び出すことができます。



クラスコンパイルによる生成コード

- データプラットフォームでは、クラス定義＝テーブル定義として取り扱えます。
- クラス／テーブル定義コンパイル時、オブジェクトとSQLそれぞれのアクセスに対応した実行コードを生成しています。

スキーマ名.テーブル名.番号

```
02/24/2014 12:02:55 に修飾子 'cukb /checkuptodate=expandedonly' でコンパイルを開始しました。  
クラスのコンパイル中 Training.Person  
テーブルのコンパイル中 Training.Person  
ルーチンのコンパイル中 Training.Person.1  
コンパイルが正常に終了しました (所要時間: 0.303秒)。
```

スタジオ出力画面

- クラスのコンパイルごとに生成ルーチンは、削除／再作成されます。
 - 小さな変更では、差分コンパイルが行われ、既存ルーチンを削除することなく小さな追加のルーチンを生成します。
- クラス操作の中でエラーが発生した場合は、生成ルーチンの行番号でエラー発生個所を示します。
 - 表示 → 他のコードを表示 をクリックすると、クラスの生成コードをスタジオで開くことができます。(参照のみ)



演習内容のイメージ

Objectで更新

Training.Person

Name As %String
Email As %String

ID	Name	Email
INTEGER	VARCHAR	VARCHAR

CreateEmail() = ストアド: Training.Person_CreateEmail()

生成コード

グローバル変数

^Training.PersonD、^Training.PersonI

ID	Email	Name
1	Heloisa@GyroAismL.net	山田
2	Elmo@IsmlesT.org	鈴木
3	Yan@lonOps.com	佐藤
4	Roberta@IsoOn.net	川田
5	Andrew@MiliGeoCo.com	斉藤
6	Mario@AmOpAn	川上
7	Usha@Two	山本
8	Samantha@Abl.org	佐々木
9	Christine@IcOpAnTw.net	田中
10	Jeff@Vi.com	田原

SQLで更新

生成コード＋
ユーザコード

```
^Training.PersonD = 10
^Training.PersonD(1) = $Ib("", "山田", "Heloisa@GyroAismL.net")
^Training.PersonD(2) = $Ib("", "鈴木", "Elmo@IsmlesT.org")
^Training.PersonD(3) = $Ib("", "佐藤", "Yan@lonOps.com")
^Training.PersonD(4) = $Ib("", "川田", "Roberta@IsoOn.net")
^Training.PersonD(5) = $Ib("", "斉藤", "Andrew@MiliGeoCo.com")
^Training.PersonD(6) = $Ib("", "川上", "Mario@AmOpAn.net")
^Training.PersonD(7) = $Ib("", "山本", "Usha@TwoLookG.com")
^Training.PersonD(8) = $Ib("", "佐々木", "Samantha@Abl.org")
^Training.PersonD(9) = $Ib("", "田中", "Christine@IcOpAnTw.net")
^Training.PersonD(10) = $Ib("", "田原", "Jeff@Vi.com")
```

```
^Training.PersonI("NameIndex", "佐々木", 8) = ""
^Training.PersonI("NameIndex", "佐藤", 3) = ""
^Training.PersonI("NameIndex", "山本", 7) = ""
^Training.PersonI("NameIndex", "山田", 1) = ""
^Training.PersonI("NameIndex", "川上", 6) = ""
^Training.PersonI("NameIndex", "川田", 4) = ""
^Training.PersonI("NameIndex", "斉藤", 5) = ""
^Training.PersonI("NameIndex", "田中", 9) = ""
^Training.PersonI("NameIndex", "田原", 10) = ""
^Training.PersonI("NameIndex", "鈴木", 2) = ""
```

おまけ: 機能の追加 (JSONアダプタ)

- クラス定義は多重継承が行えます。
- %JSON.Adapterを追加して動作を確認してみましょう。
 - プロパティ名を利用してJSONオブジェクトのインポート、エクスポートが行えます。

```
Training.Person.cls
1 Class Training.Person Extends (%Persistent, %JSON.Adapter)
2 {
```

```
TEST>set person=##class(Training.Person).%OpenId(1)
```

```
TEST>do person.%JSONExport()
```

```
{"Name":"山田たろう","Email":"yamada@majorcorp.com"}
```

```
TEST>do person.%JSONExportToString(.moji)
```

```
TEST>write moji
```

```
{"Name":"山田たろう","Email":"yamada@majorcorp.com"}
```

```
TEST>do person.%JSONExportToStream(.stream)
```

```
TEST>write stream.Read()
```

```
{"Name":"山田たろう","Email":"yamada@majorcorp.com"}
```

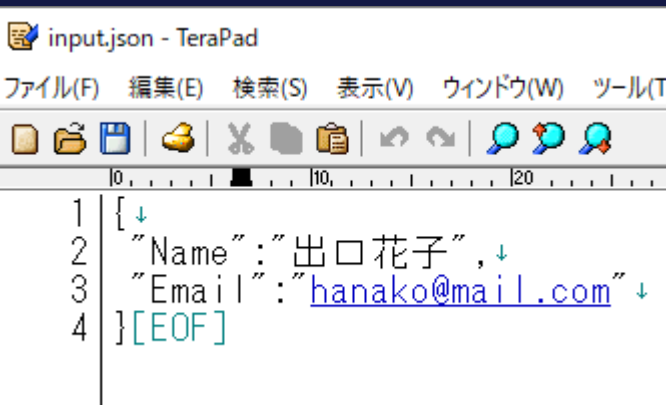
```
TEST>
```

エクスポート例



おまけ: 機能の追加(JSONアダプタ): インポートの例

- プロパティ名に合わせたJSONオブジェクトをインポート
(UTF8で保存したファイルを使用している例)。



The screenshot shows a TeraPad editor window titled 'input.json - TeraPad'. The menu bar includes 'ファイル(F)', '編集(E)', '検索(S)', '表示(V)', 'ウィンドウ(W)', and 'ツール(T)'. The toolbar contains icons for file operations and editing. The editor content is as follows:

```
1 {  
2   "Name": "出口花子",  
3   "Email": "hanako@mail.com"  
4 }[EOF]
```

input.jsonの中身

```
TEST>set person=##class(Training.Person).%New()  
  
TEST>do person.%JSONImport("C:¥temp¥input.json")  
  
TEST>zwrite person  
person=1@Training.Person ; <OREF>  
+----- general information -----  
|   oref value: 1  
|   class name: Training.Person  
| reference count: 2  
+----- attribute values -----  
|   %Concurrency = 1 <Set>  
|   Email = "hanako@mail.com"  
|   Name = "出口花子"  
+-----
```



エラーステータス(%Status)について

- システム提供クラスでは、処理の成功／失敗のステータス(%Status)をメソッドの戻り値や引数で返すクラスが多数あり、実行結果の成功可否の判断に使用します。
 - %Statusが設定されている場合、成功したときは1、エラー時はオブジェクトでエラー情報が渡されます。 例) set status = human.%Save()
 - メソッドの実行結果を取得し損ねた場合は、デフォルトで用意される%objlasterror変数にステータスが格納されます。
 - %objlasterrorは、直近で発生したオブジェクトのエラーステータスを格納するローカル変数です(明示的に中身をクリアしない限り情報は残ります)。
- エラーステータスの解析には**%SYSTEM.Statusクラス**を利用します。

\$SYSTEM.Status.メソッド名()
で実行できます。

\$system.Status.xxx	用途
IsError(st)	引数のステータスコードがエラーであるとき1、それ以外は0を返す。
GetErrorText(st)	引数のステータスコードのエラーテキストを返す。複数エラーが発生した場合は、エラーテキストをCR+LFで区切って返す。
GetOneStatusText(st,#)	複数エラーが発生した場合、第2引数に指定した番号のエラーテキストのみを返す。
DecomposeStatus(st,.err)	エラーステータスを分解して第2引数の変数に代入する。
StatusToSQLCODE(st,.msg)	エラーステータスからSQLCODEを返します。第2引数はエラーメッセージが設定される出力引数です。

エラー処理: 便利なマクロ

- クラス／テーブル定義では、オブジェクトのステータス確認に利用するマクロなど、**%occStatus.inc** から自動的に提供されます。
 - 何も継承しないクラス定義(メソッドのコンテナ)では提供されません。

例

```
ClassMethod A() As %Status
{
  set st=$$$OK // 戻り値の初期値設定
  // 入力アシスト機能を利用するための指定
  #dim ex As %Exception.AbstractException
  try {
    // コードの記述
    /* %Statusを戻すメソッドの呼び出しなど */
    $$$THROWONERROR(ex,st)
  }
  catch ex {
    set st=ex.AsStatus() //エラー時の戻り値を設定
    /* 任意のエラー処理 */
  }
  quit st //戻り値を持つ場合に記述
}
```

%occStatus内マクロ(一部)	内容
\$\$\$ISERR()	%Statusコードがエラーの場合1を返します。
\$\$\$THROWONERROR(err,st)	第2引数の%Statusコードがエラーである場合、第1引数の変数にステータスから例外オブジェクトに変換したものを格納し、CATCHブロックへTHROWします。
\$\$\$OK	%StatusでOK(=1)を返すときに使用します。

クラスリファレンス(クラスドキュメント)

- 保存したクラスのクラス定義ドキュメントを自動的に作成します。
- スタジオでは、以下メニューから起動できます。
 - ワークスペースウィンドウのクラス名を右クリックし、クラスドキュメントの表示をクリックします。
 - クラスを編集中に表示→クラスドキュメントの表示をクリックします。
- ドキュメンテーションホームページから、クラスリファレンス情報をクリックします。

%付きクラス詳細
も確認できます。

INTERSYSTEMS クラスリファレンス Training.Person

ドキュメント | 検索

[kiso] > [Training] > [Person]

作成したクラス参照できます。

persistent class **Training.Person** extends [%Persistent](#)

▼ Inventory

Parameters	Properties	Methods	SystemMethods	Queries	Indices	ForeignKeys
	2	4			1	

▼ Summary

プロパティ

Email	Name
-------	------

メソッド

%AcquireLock	%AddToSaveSet	%AddToSyncSet	%E
%CheckConstraints	%CheckConstraintsForExtent	%ClassIsLatestVersion	%C
%ComposeOid	%ConstructClone	%ConstructCloneInit	%L
%DeleteData	%DeleteExtent	%DeleteId	%L
%DispatchGetModified	%DispatchGetProperty	%DispatchMethod	%L