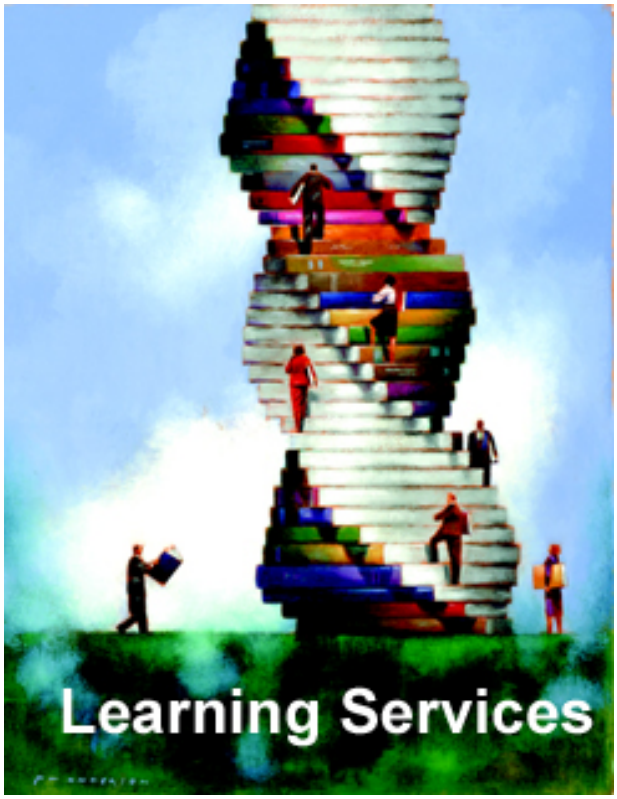


IRIS オブジェクト操作ガイド

（Version 2024.1 ベース）

V1.0

2024 年 12 月

インターシステムズジャパン株式会社

1. はじめに

この資料では、「IRIS ファーストステップ・ガイド」[1]をやり終えた方を対象に、IRIS オブジェクトの発展的な内容について説明します。
この資料は、実際に IRIS を操作しながら進められるよう書かれていますので、IRIS のコミュニティ版を実際にインストールしてお使い頂けば理解が深まると思います。

このガイドでは、IRIS オブジェクトを利用して、オブジェクト指向のアプリケーションを開発していきます。
IRIS のもつオブジェクト機能は、継承、関連など完全なオブジェクト指向をサポートしているので、オブジェクト指向開発による生産性向上の恩恵をフルに受けることができます。

このガイドは Java や.NET からのアクセスなど、クライアント側の説明を含みません。
IRIS オブジェクトを.NET などからアクセスする方法の説明は、該当するドキュメントを参照ください。
本ガイドでは主に、IRIS ターミナルから、サーバサイドでオブジェクトにアクセスする方法を説明します。

本ガイドは、Visual Studio Code をはじめ IRIS のツールに習熟していることが前提となります。
ツールについて知りたい場合は、「IRIS ファーストステップ・ガイド」やマニュアル等をご参照ください。

アプリケーションの概要

ここで、本ガイドで作成するアプリケーションの概要について説明します。
アプリケーションでは、クラス図で表されるデータを扱います。

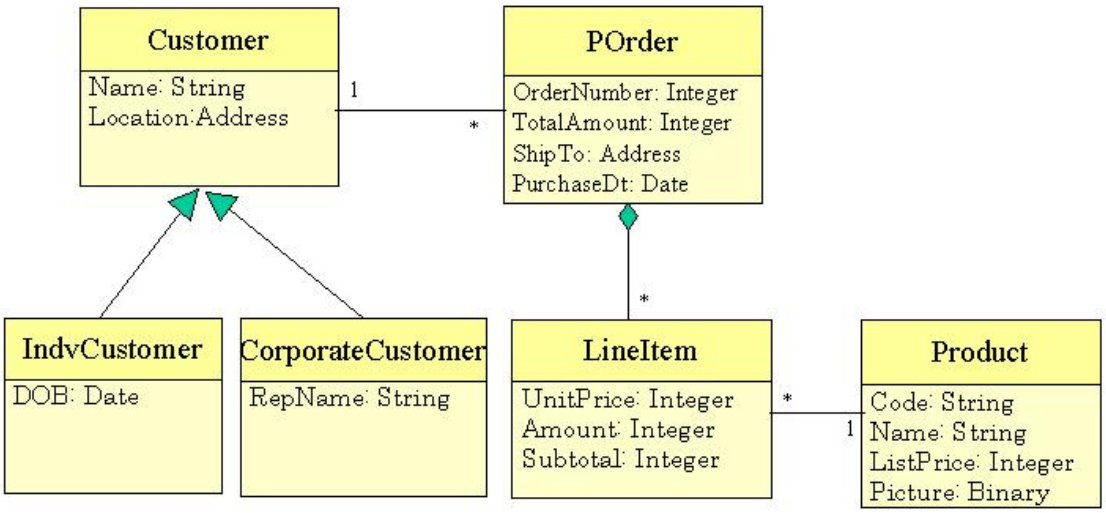


図 1　アプリケーションテーマのクラス図

このクラス図は、ある商品の購買のシステムを表しています。
顧客（Customer）と商品（Product）のクラスがあり、それらを結びつける注文（POrder, LineItem）のクラスがあります。
顧客（Customer）は、個人顧客（IndvCustomer）と法人顧客に分かれ、それらは継承によって表現されます。

このガイドでは、以上のクラスを IRIS で実装し、ObjectScript を用いて注文処理を行うところまで、順に解説していきます。

2. IRIS オブジェクト – 基本クラス

アプリケーションの作成に入る前に、IRIS のクラス階層の基本となるクラスについて説明します。

(1) %RegisteredObject

%RegisteredObject クラスは、メモリ上で扱われるオブジェクトの基本クラスです。
IRIS は、プロセスのメモリ内でオブジェクトを管理するために、OREF（オブジェクト・リファレンス）と呼ばれる、プロセス内で一意な ID を使用します。
%RegisteredObject クラスでは、OREF を扱うための機能が組み込まれています。

メモリ上にオブジェクトを新しく生成するには、次のようなコマンド（ObjectScript）を実行します。

```
Set obj = ##class(classname).%New()
```

%New メソッドは、新たにオブジェクト・インスタンスをメモリ上に生成するものです。
これにより、変数 obj に生成したオブジェクトの OREF が格納されます。

OREF は、システム内で特別な管理がなされています。
例えば、

```
Set obj2 = obj
```

として、OREF をコピーすると、メモリ上のオブジェクトが 2 箇所から参照されることになります。
このような「参照カウンタ」はシステムで自動的に管理されます。
そして、参照カウントが 0 になると、オブジェクトは自動的にメモリから削除されます。
上の例では、obj, obj2 の両方の変数がスコープから出たり、または、明示的にそれらの変数に""を代入したりすれば、自動的にオブジェクトが削除されます。
したがって、明示的にクローズする必要はありません。

(2) **%Persistent**

%Persistent は、ディスクに保存されるオブジェクトのクラスで、%RegisteredObject を継承しています。
したがって、メモリ上での振る舞いは、%RegisteredObject クラスの機能を引き継いでいます。

%Persistent は、%RegisteredObject の機能に加え、オブジェクトをディスクに保存したり、ディスクからロードするための機能が実装されています。
IRIS では、ディスク上のオブジェクトを識別するために、OID をオブジェクトに付与します。
OID は、システムで一意の ID で、既定では、クラス名とシーケンス番号により管理されています。

メモリ上のオブジェクト obj をディスクに保存する場合は、次のように%Save()メソッドを使用します。

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

status には保存が成功したかどうかのステータスが返ります。
また、ディスクからメモリにオブジェクトをロードする場合は、%OpenId メソッドを使用します（%Open という同様のメソッドもあります）。

```
Set obj = ##class(classname).%OpenId(id)
```

これにより、変数 obj にロードしたオブジェクトの OREF が代入されます。

(3) **%SerialObject**

%SerialObject も%RegisteredObject を継承して、メモリ上のオブジェクトの振る舞いを実装していますが、それに加え、他の埋め込まれるための機能を持ちます。
他のオブジェクトに埋め込まれるためには、オブジェクトの状態を「シリアライズ」し、親オブジェクトのプロパティとして扱われる必要があります。
%SerialObject は、自分自身を「シリアライズ」する機能を持っています。

3. 商品（Product）クラスの定義

それでは、アプリケーションに戻り、まず、商品を表す Product クラスの定義から始めます。
以後、本ガイドで作成するクラスは、パッケージ "Shop" 以下に作成するものとします。
また、クラスを定義・変更したらコンパイルが必要です。
本ガイドでは特に明示しませんが、クラスを定義・変更した場合は適宜コンパイルを行ってください。

(1) Persistent クラスの定義

Product クラスはディスクに保存するため、%Persistent を継承する形で実装します。
%Persistent を継承するクラスを定義する一つの方法は、Visual Studio Code の新規ファイル作成メニューで拡張子 cls を付加したファイルを作成することです。
その際、次の画面のように、Extend の後ろに%Persistent を記述します。

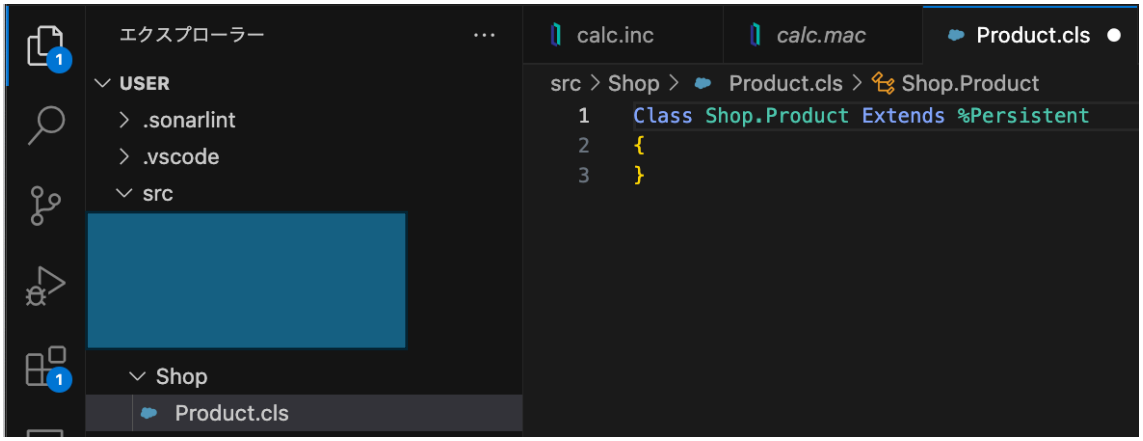


図 2 クラスタイプ設定画面

ウィザードにより定義したクラスは、次のようになります。

```
Class Shop.Product Extends %Persistent
{
}
```

%Persistent クラスを継承して定義されていることがお分かり頂けると思います。

次にプロパティを定義していきます。Product クラスは次のようなプロパティを持ちます。

Code （商品コード）
Name（商品名）
ListPrice（定価）
Picture（画像）

これらを Visual Studio Code で定義するには次のような宣言をコードウィンドウに追加します。

```
/// 商品コード
Property Code As %String;
/// 商品名
Property Name As %String;
/// 定価
Property ListPrice As %Integer;
/// 画像
Property Picture As %Stream.FileBinary;
```

%String や%Integer は、単純型で、それぞれ文字列、整数を表します。
それに対して、%Stream.FileBinary は、画像のようないわゆる BLOB（Binary Large Object）です。
IRIS では、このような BLOB データも容易に扱うことができます。
Picture プロパティにどのように BLOB データを設定するかについては、後で説明します。

(2) **IDKEY**

次に、OID について考えてみます。
デフォルトで OID は、オブジェクトが保存された順に番号が振られていきます。
通常はこれで問題ありませんが、特定プロパティの値を OID として扱いたい場合もあると思います。
IRIS では、このような場合、次のように IDKEY インデックスを定義します。

```
Index CodeIdx On Code [ IdKey, Unique ];
```

ここで、CodeIdx はインデックスの名前で、"On Code"句により Code プロパティを対象とすることを示します。

以上をコンパイル(ファイル>保存)すれば、商品クラスの定義が完了します。

(3) **オブジェクトの生成・保存**

では、IRIS ターミナルを開いて商品オブジェクトを一つ生成してみます。
以下は、ターミナルに対する入力です。

```
set p=##class(Shop.Product).%New()                    . . .  (*1)
set p.Code="P123456"
set p.Name="デスクトップ PC Pentium IV 2.6G"
set p.ListPrice=118000
set file=##class(%FileBinaryStream).%New()            . . .  (*2)
do file.LinkToFile("c:¥desktop.jpg")                  . . .  (*3)
do p.Picture.CopyFrom(file)                           . . .  (*4)
write p.%Save()                                       . . .  (*5)
```

(*1)では、%New() メソッドで Shop.Product クラスのインスタンスを（メモリ上に）生成し、その OREF が p に代入されます。
以後、変数 p に対して操作を行います。
p.Code という参照は、p が参照しているインスタンスの Code プロパティを表します。
このように、IRIS オブジェクトでは、オブジェクト指向言語で一般的な「ドット記法」を用いて、オブジェクトのプロパティやメソッドにアクセスします。
(*2),(*3)では、画像ファイルを"c:¥desktop.jpg"から読み込んで、メモリ上の Stream オブジェクトを生成しています。
そして、（*4）でそのオブジェクトの内容（すなわちファイルの内容）を、Picture プロパティにコピーしています。

このようにして、Picture プロパティに BLOB が設定されました。

最後に(*5)で、%Save() メソッドによりオブジェクトをデータベースに保存します。
1 が返ってくれば成功です。
万が一エラーが発生する場合に備えて、

```
set sc=p.%Save()
do $system.OBJ.DisplayError(sc)
```

とすれば、エラーが表示できます。

4. 注文クラス（Porder, LineItem）の定義

次に、注文を表すクラスを定義します。
ひとつの注文を表す POrder クラスと、注文の明細を表す LineItem クラスから構成されます。

Product クラスと同様に、Persistent クラスを新規に作成し、Shop.POrder という名前にします。
POrder クラスの基本プロパティは次の通りです。

```
Class Shop.POrder Extends %Persistent
{
/// オーダ番号
Property OrderNumber As %String;

/// 出荷先
Property ShipTo As Shop.Address;

/// 購入日
Property PurchaseDt As %Date;

/// 合計価格
Property TotalPrice As %Integer [ Calculated ];

/// 処理中フラグ
Property IsProcessing As %Boolean;
}
```

(1) 埋め込みオブジェクト

まず、出荷先を表す ShipTo プロパティを見てください。
型が Shop.Address となっています。
これは、ShipTo プロパティが、出荷先の住所を表す Shop.Address を持つということです。

では、Shop.Address を定義します。
これまでと同様に VSCode で「新規ファイル」でクラス定義を作成します。
名前を Shop.Address とした後、コードの Extend の後ろを%SerialObject に変更し。後は、「保存」を押し、クラスの作成を終了します。

作成後のクラスにプロパティ定義を加え、以下のようにします。

```
Class Shop.Address Extends %SerialObject
{
Property Street As %String;
Property City As %String;
Property PostalCode As %String;
}
```

Shop.Address クラスは、%SerialObject クラスを継承していますので、それ自身ではデータベースに保存する機能を持っていません。
その代わりに、他のオブジェクトに埋め込まれるため、自分自身をシリアライズする機能を持っています。
ここでは、ShipTo プロパティとして Shop.Porder クラスのオブジェクトに埋め込まれることになります。

(2) 計算プロパティ

合計価格を表す"TotalPrice"プロパティに注意してください。
定義の最後に、[Calculated]と指定されています。
これは、計算プロパティと呼ばれ、プロパティの値をデータベースに保存せず、参照時に計算させることができます。
計算のロジックについては、後で説明します。

(3) 親子リレーションシップ

このガイドでは、注文と注文明細を分けて考えています。
注文クラスがあるひとつの注文を表すのに対し、注文明細クラスは、注文の中の商品とその数量を表しま

す。
従って注文クラス（POrder）と注文明細クラス（LineItem）は、一対多の関係で、かつ、注文明細の存在は、注文本体に依存することになります。
IRIS ではこのような関係を、親子リレーションシップで実装することができます。

まず、子クラスとなる LineItem を定義します。
通常の Persistent クラスとして以下のように定義します。

```
Class Shop.LineItem Extends %Persistent
{
/// 商品
Property Product As Shop.Product;

/// 単価
Property UnitPrice As %Integer;

/// 数量
Property Amount As %String;

/// 小計
Property Subtotal As %Integer [ Calculated ];

Method SubtotalGet() As %Integer
{
        Quit ..UnitPrice * ..Amount
}
}
```

注文明細は商品と関連付いています。
それが Product プロパティですが、Shop.Product 型として定義されていることに注意してください。
Shop.Product は先に定義してある Persistent クラスですので、この場合は埋め込みオブジェクトではなく、独立したオブジェクトへの片方向参照を表します。

関連付いている商品オブジェクトは、商品の定価（ListPrice）を持ちますが、値引きなどを考慮して注文明細自身に、単価（UnitPrice）を定義しておきます。

また、小計（Subtotal）は、この明細行の合計ですが、ここでも計算プロパティを使用しています。
先程説明したように、計算プロパティはデータベースに値を格納せずに、参照時に計算されます。
そして計算するロジックを定義する必要があります。
それが、ここでは、SubtotalGet()メソッドです。
一般に、*PropertyName*Get()という名前が、計算プロパティの計算メソッドになります。
%Integer 型のプロパティを計算しますので、%Integer を返すメソッドとして定義されています。
中身は単純で、自分自身の単価（UnitPrice）と数量（Amount）との積を求めます。
コード中、”..”はメソッドが呼び出された対象となる、自分自身のオブジェクトを表し、Java や C++の this に当たります。

では、親子リレーションシップの定義に戻ります。

POrder クラスに次のような、親子リレーションシップを追加します。

```
Relationship Items As Shop.LineItem [ Cardinality = children, Inverse = POrder ];
```

また、LineItem クラスにも次のような逆参照のプロパティが追加されています。

```
Relationship POrder As Shop.POrder [ Cardinality = parent, Inverse = Items ];
```

以上で、注文（POrder）と注文明細（LineItem）の親子リレーションシップの定義が完成しました（POrder クラスを保存するのを忘れないでください）。

最後に、POrder クラスの計算プロパティ、TotalPrice を計算するメソッドを次の通り定義します。

```
Method TotalPriceGet() As %Integer
{
        Set total = 0
        For i=1:1:..Items.Count() {
                Set total =total + ..Items.GetAt(i).Subtotal
        }
        Quit total
}
```

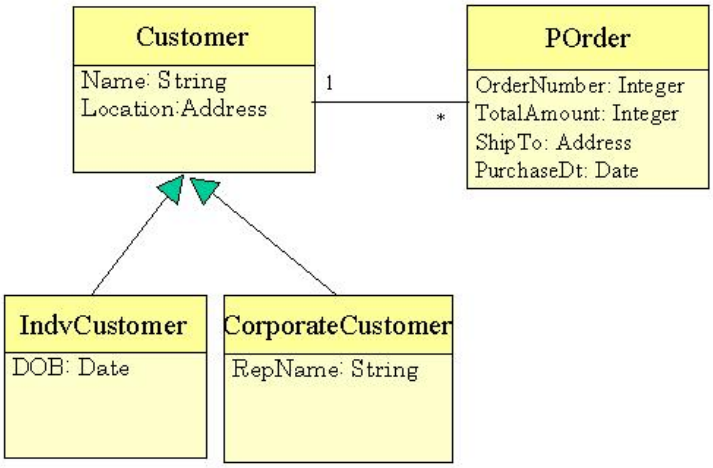
TotalPrice は、注文の合計額ですから、各注文明細の小計の和が求めるものとなります。
上のコードでは、注文明細（Items プロパティ）を順にループして、合計を求めています。
Items は、先に定義した子クラスを表していますが、ここで、Count()メソッドは、関連付いている子オブジェクトの数を返し、GetAt(n)は、n 番目の子オブジェクトへの参照を返すメソッドです。

5. 顧客クラスの定義

次に顧客を表すクラスを定義していきます。

0.アプリケーションの概要　のクラス図で示したように、継承を利用して、個人顧客と法人顧客を扱います。

（関連部分のクラス図を再掲します）



まずは、継承の元となる Customer クラスの定義を行います。

Customer には、個人顧客、法人顧客、共通のプロパティを定義します。

コードは以下のようになります。

```
Class Shop.Customer Extends %Persistent
{
Property Name As %String;
Property Location As Shop.Address;
Property TotalOrderAmount As %Integer [ Calculated ];
Method determinePrice(listprice As %Integer) As %Integer
{
        Quit 0
}
}
```

Customer は、データベースに保存しますので、Persistent クラスとして定義します。
また、次のようなプロパティを持ちます。

- 名前（Name）
- 住所（Location）
- 合計注文金額（TotalOrderAmount）：計算プロパティ

Location プロパティは、Shop.Address クラスを埋め込みオブジェクトとして扱います。
また、計算プロパティである TotalOrderAmount の計算メソッドについては、後程 POrder クラスとの関連を定義したときに説明します。

determinePrice()メソッドは、定価を渡して実際の購入価格を返すメソッドです。
本アプリケーションでは、個人顧客では定価で販売するのに対し、法人顧客ではこれまでの購入金額によって値引きを行うものとします。
この determinePrice()メソッドは、そのロジックを記述します。
ロジックは、Customer クラスを継承して、個人顧客、法人顧客に対応するサブクラスを定義したときに、併せて定義します。
詳しくは、P22 の「継承による個人顧客クラス、法人顧客クラスの定義」をご覧ください。
親クラスである Customer では、単にこのメソッドを定義しているだけでロジックは実装しません（単に 0 を返すだけです）。

(1) POrder との一対多リレーションシップ

1 人の顧客は複数の注文を行うことができます。
これをクラス図で表すと、顧客と注文は一対多の関連をもつことになります。
IRIS では、このような関連を一対多リレーションシップで実装することができます。

親子リレーションシップと似ている概念ですが、若干異なります。
注文と注文明細の関連では、注文明細は注文の存在が前提であるため、親子リレーションシップでしたが、顧客と注文はそれぞれ独立して存在できるため、ここでは、一対多リレーションシップを用います。

顧客と注文の関連は、顧客が個人か法人かには依らないため、親クラスである Customer クラスで実装します。

Shop.Customer クラスには、次の定義を追加します。

```
Relationship POrders As Shop.POrder [ Cardinality = many, Inverse = Customer ];
```

また、Shop.POrder には、次の定義を追加します。

```
Relationship Customer As Shop.Customer [ Cardinality = one, Inverse = POrders ];
Index CustomerIndex On Customer;
```

(2) 計算プロパティ TotalOrderAmount

POrder クラスの TotalPrice プロパティと同様に TotalOrderAmount プロパティを参照時に計算するロジックを実装します（TotalOrderAmountGet()メソッド）。

```
Method TotalOrderAmountGet() As %Integer
{
        Set total = 0
        For i=1:1:..POrders.Count() {
                Set o = ..POrders.GetAt(i)

                // 処理中のオーダを除いて計算する
                If o.IsProcessing = 0 { Set total = total + o.TotalPrice }
        }
        Quit total
}
```

先程定義した、POrder へのリレーションシップを走査して、この顧客が行った注文金額の合計を計算します。
リレーションシップを走査するためのメソッドは、一対多と親子で同じであることに注意してください。

(3) 継承による個人顧客クラス、法人顧客クラスの定義

では、まず個人顧客を表す IndvCustomer クラスから定義していきます。

IndvCustomer クラスは Shop.Customer を継承するように以下のコードを記述します。

```
Class Shop.IndvCustomer Extends Shop.Customer
{
}
```

Extends Shop.Customer という部分で、Shop.Customer クラスを継承していることを宣言しています。
Shop.Customer が Persistent クラスなので、必然的に Shop.IndvCustomer も Persistent になります。
また、IndvCustomer クラスは、この時点で、Customer クラスが持つプロパティ・メソッドはすべても持ちます。

次に、個人顧客に固有の実装を行っていきます。
まずは、プロパティの追加です。
個人顧客の場合、誕生日（DOB）を持つとします。
次のようなコードを追加します。

```
Property DOB As %Date;
```

これにより、IndvCustomer クラスは、Customer から継承した Name、Location と、今定義した DOB のプロパティを持つことになります。

次にメソッド determinePrice()をオーバライドします。
オーバライドとは、親クラスから継承したメソッドの実装を変えるために、子クラスでメソッドを再定義することです。

```
Method determinePrice(listprice As %Integer) As %Integer
{
        Quit listprice
}
```

個人顧客には割引をしないという仕様に基づき、ここでは与えられた定価をそのまま返すような実装になっています。

同様に、法人顧客を表す CorporateCustomer クラスを定義します。
定義の内容は次の通りです。

```
Class Shop.CorporateCustomer Extends Shop.Customer
{
Property RepName As %String;

Method determinePrice(listprice As %Integer) As %Integer
{
        Set price = 0
        // これまでの購入金額に応じて割引
        If ..TotalOrderAmount > 1000000 {
                Set price = listprice * 0.8
        } Else {
                Set price = listprice
        }
        Quit price
}
}
```

プロパティとしては、担当者名（RepName）を追加しています。
また、IndvCustomer と同様、determinePrice メソッドをオーバライドしています。

CorporateCustomer の determinePrice()メソッドは、少々込み入っています。
これまでの注文金額を計算し、それが 1,000,000 円を超えていれば定価の 2 割引とし、そうでなければ定価のまま（値引きなし）というものです。

(4) **顧客インスタンスの作成**

一通り顧客クラスの定義が終わりましたので、動作を確認するために簡単なインスタンスを作成してみます。
IRIS ターミナルを対話的に使用します。

まず、個人顧客（IndvCustomer）のインスタンスを作成し、保存します。

```
Set ic=##class(Shop.IndvCustomer).%New()
Set ic.Name="日本 太郎"
Set ic.Location.Street="丸の内"
Set ic.Location.City="東京"
Set ic.Location.PostalCode="100-0001"
Set ic.DOB=$zdh("2/4/2003")
Set st=ic.%Save()
```

Name,Location は、Customer クラスから継承しているプロパティです。
Location は Shop.Address クラスの埋め込みプロパティですので、ドットを連ねて指定しています。
DOB は、IndvCustomer で定義されたプロパティです。
日付を指定する場合、

$zdh(“mm/dd/yyyy”)

という IRIS 関数により、文字列から内部フォーマットに変換しています。

同様に、法人顧客のほうも次のように作成します。

```
Set cc=##class(Shop.CorporateCustomer).%New()
Set cc.Name="インターシステムズジャパン"
Set cc.Location.Street="西新宿"
Set cc.Location.City="東京"
Set cc.Location.PostalCode="160-0023"
Set cc.RepName="速井 一郎"
Set st=cc.%Save()
```

CorporateCustomer の場合、RepName プロパティを指定できることを確認してください。

ここで determinePrice()メソッドを、それぞれのオブジェクトに対して実行すると、同じ結果が得られるはずです。

```
> write ic.determinePrice(10000)
10000
> write cc.determinePrice(10000)
10000
```

これは、IndvCustomer の方は常に与えられた値を返すのに対し、CorporateCustomer は、合計注文金額がまだ 1,000,000 以下なので、与えられた値をそのまま返すためです。
後程、顧客に注文を関連付けた際、IndvCustomer と CorporateCustomer の振る舞いの違いを確認します。

6. POrder クラスのメソッド

以上で、クラス定義の大枠が完成しました。
ここでは、POrder クラスに注文を作成して、発注を行うメソッドを定義していきます。
流れは次の通りです。

1. 顧客情報と出荷先を入力として、注文を作成
2. 商品と数量を指定（必要に応じて繰り返し）
3. 発注

1. は、注文インスタンスを作成するメソッドですので、クラスメソッドとして実装します。
2. 、3.は、1.で作成されたインスタンスに対する操作ですので、インスタンスメソッドとして実装します。

以下で、POrder クラスにメソッドを追加していきます。

(1) 注文の作成

注文の作成のメソッドです。

```
ClassMethod create(cust As Shop.Customer, shipto As Shop.Address) As Shop.POrder
{
        Set po = ##class(Shop.POrder).%New()
        Set po.Customer = cust
        Set po.ShipTo = shipto
        Set po.IsProcessing = 1
        Quit po
}
```

このメソッド（create()）は、引数として、注文を発行しようとしている顧客（Shop.Customer）と、出荷先（Shop.Address）を取ります。

ロジックは単純で、まず、Shop.POrder クラスのインスタンスを%New()で生成し、あとはそのプロパティを与えられたパラメータから設定します。
Customer プロパティは一対多リレーションシップです。
リレーションシップは両方向関連ですので、このように POrder 側から設定した場合、Customer 側にも反映されます。

IsProcessing プロパティを 1 にして、注文が生成され処理中（発注確定前）であることを示します。
最後に、作成した POrder クラスのインスタンスを返します。

(2) **商品と数量の指定**

次に、注文オブジェクトに、注文明細を追加していきます。
注文明細は、商品とその数量からなりますので、Shop.Product のインスタンスと数量を表す整数を引数に取ります。

次のコードがその実装です（addItem()メソッド）。

```
Method addItem(p As Shop.Product, amt As %Integer)
{
        Set item = ##class(Shop.LineItem).%New()
        Set item.Product = p
        Set item.UnitPrice = ..Customer.determinePrice(p.ListPrice)
        Set item.Amount = amt
        Do ..Items.Insert(item)
}
```

まず、注文明細（LineItem）クラスのインスタンスを作成します。
そしてそれに対して、Product、UnitPrice、Amount の各プロパティを設定します。

UnitPrice プロパティは、注文した商品の単価を表しますが、ここでは単価は、定価から顧客に応じた値引きを行って得られたものです。
したがって、注文に関連付いている顧客（Customer プロパティ）の determinePrice()メソッドに、与えられた商品の定価を渡すことで求めています。
ここで、このメソッドは、値引きのロジックについては一切触れていないことに注意してください。
値引きは、実行時に関連付いている顧客オブジェクトによって計算されます。
すなわち、関連付いているのが、個人顧客か法人顧客かによって、単価が決定されます。
このように、実行時のオブジェクトのクラスによって動作を決定できることをポリモフィズムといい、ソフトウェアの柔軟性を高めるためオブジェクト指向においては非常に重要な概念です。

メソッドの最後に、注文の Items プロパティに、作成した注文明細を追加しています。
このように、親子リレーションシップの親側では、Insert メソッドによって子クラスのオブジェクトを追加することができます。

(3) **発注**

注文を確定させる place()メソッドです。

```
Method place() As %Status
{
        Set ..PurchaseDt = +$h
        Set ..OrderNumber = "O"_$tr($Justify(..getSeqNum(), 9)," ",0)
        Set ..IsProcessing = 0
        Quit ..%Save()
}
```

まず、PurchaseDt プロパティに、現在の日付を代入します。
$h 特殊変数は、1841 年 1 月 1 日を基準とした経過日数と、その日 0 時 0 分からの経過時間を秒数で表し、

経過日数,秒数

とカンマで区切った値を返すものです。
ここでは、それに+を先頭に付けることで、経過日数部分だけを取り出し、それを PurchaseDt プロパティに設定しています。
ちなみにここで$tr と$Justify を組み合わせることにより、数字を右詰にして 0 を埋め込んでいることになります。
$tr と$Jusitify の詳細については ObjectScript マニュアルをご参照ください。

次に、OrderNumber プロパティに、オーダ番号を採番して代入します。
その中で呼び出している、getSeqNum()メソッドは、次の通りです。

```
ClassMethod getSeqNum() As %Integer [ Private ]
{
        Quit $Increment(^Shop.Order)
}
```

Shop.Order は、IRIS のダイレクトアクセスモードでの、グローバル変数です。
この変数は、データベースに格納されます。
この変数を、$Increment に渡すことにより、排他的に ^ Shop.Order を 1 ずつ増分することができます。
このように、IRIS では、オブジェクトアクセスでのメソッドの中に、高速で柔軟なダイレクトアクセスを混在させることができ、開発生産性の向上に寄与しています。

また、このメソッドは、[Private] と宣言されていることに注意してください。
この宣言により、このメソッドはこのクラス（Shop.POrder）以外から呼び出すことができなくなります。

(4) **例: 注文の流れ**
では、IRIS ターミナルを起動して、以上の流れを、順を追って実行してきます。
（以下のターミナルに対する入力の例で、”>”はプロンプトを示しています。）
まず、購入対象の商品をオープンします。
ここでは、これまでに作ってある、ID＝P123456 のものをオープンし、そのプロパティの一部を確認します。

```
> Set p=##class(Shop.Product).%OpenId("P123456")
> Write p.Name
デスクトップ PC Pentium IV 2.6G
> Write p.ListPrice
118000
```

次に発注を行う顧客をオープンします。ここでは、ID=1 の個人顧客をオープンします。

```
> Set c=##class(Shop.Customer).%OpenId(1)
> Write c.Name
日本 太郎
```

次に、出荷先のオブジェクトをメモリ上に作成します。

```
> Set addr=##class(Shop.Address).%New()
> Set addr.Street="西中島"
> Set addr.City="大阪"
> Set addr.PostalCode="532-0011"
```

最後に、注文を作成し、商品を追加して、発注を行います。

```
>Set po=##class(Shop.POrder).create(c,addr)
>Do po.addItem(p,1)
>Write po.place()
1
```

1 が返ってくれば成功を表します。
po.addItem の呼出しでは、商品オブジェクト p が表す商品を 1 つ注文しています。

以上で、注文が作成されデータベースに保存されました。
例えば、

```
>w po.TotalPrice
118000
```

のように、POrder クラスのオブジェクトの TotalPrice プロパティを見ると、注文した商品の価格の合計、118,000 円が表示されます。

もう一つの例として、法人顧客が注文を行い、determinePrice()メソッドで実装している値引きのロジックが動作するかを確認しています。
まず、法人顧客をオープンします（Customer クラスの ID=2 が法人顧客のはずです）。

```
>Set cc=##class(Shop.Customer).%OpenId(2)
>Write cc.Name
インターシステムズジャパン
```

そして、先程と同様に注文オブジェクトを作成し、注文を発行します。

```
>Set po=##class(Shop.POrder).create(cc,addr)
>Do po.addItem(p,10)
>Write po.place()
1
```

ここでは、create()メソッドに渡す顧客が cc（先程オープンした法人顧客）になっている点、また、addItem()メソッドで、商品の 10 個追加している点に注意してください。

ここで、注文金額を確認すると、1,180,000 円となり今回は値引きされていませんが、この注文で、合計注文金額が 100 万円を超えたことが分かります。

```
> Write po.TotalPrice
1180000
```

そこで、再度この顧客に対して注文を作成します。

```
>Set po=##class(Shop.POrder).create(cc,addr)
>Do po.addItem(p,10)
>Write po.place()
1
>Write po.TotalPrice
944000
```

そして、注文の合計金額を求めると、944,000となり定価の２割引で注文ができたことが確認できます。これまでに説明したように、実行時に注文を行う顧客の種類（個人顧客、法人顧客）によって、異なった determinePrice()メソッドの実装が呼び出され、法人顧客に対する値引きのロジックがうまく実装されていることがお分かりいただけたと思います。

7. クエリによる注文の検索

以上の例では、顧客や商品をオープンする際、%OpenId メソッドに直接 OID を指定して行っていました。
しかし、オブジェクトをオープンするとき、常に OID が分かるわけではありません。
例えば注文であれば、注文日付をキーにした検索をしたい場合もあるでしょう。

IRIS ではこのような場合に、クラスに対してクエリを定義することができます。
クエリは、クラスに対して定義された SELECT 文で、サーバサイドで検索が実行されます。
また、ODBC や JDBC に対しては、クエリをストアドプロシージャとして公開することもできます。

では、VScode を使用して、POrder クラスに対して、注文日付で注文を検索するクエリを定義します。
POrder をオープンした状態にしてください。
（なお、ここでは取り上げませんが、注文日付による検索を高速化するためには、通常 PurchaseDt プロパティに対してインデックスを設定します。）

以下のクエリ定義を追加します。

```
Query ByDate(dt As %Date) As %SQLQuery(CONTAINID = 1)
{
SELECT %ID FROM POrder
 WHERE (PurchaseDt = :dt)
}
```

ここでコードに修正を加えます。
今、PurchaseDt プロパティと与えられた引数:dt とは直接比較していますが、PurchaseDt は IRIS の内部形式で日付を保持しているため、”1/26/2004”のような日付を表す文字列と比較することができません。
そこで、次のように、TO_DATE 関数を使用します。

```
Query ByDate(dt As %Date) As %SQLQuery(CONTAINID = 1)
{
SELECT %ID FROM POrder
 WHERE (PurchaseDt = TO_DATE(:dt,'MM/DD/YYYY'))
}
```

TO_DATE(:dt,'MM/DD/YYYY')

とすることで、与えられた:dt を’月/日/年’の形式として解釈し、IRIS の内部形式に変換しています。

(1) クエリの実行

では、定義したクエリを実行する方法について説明します。
クエリを実行するには、%ResultSet クラスを使用します。

ここでは、次のようなクラスメソッドを定義して、%ResultSet の代表的な使用方法について説明します。

```
ClassMethod printOrderByDate(dt As %Date) {
	Set rs=##class(%ResultSet).%New("Shop.POrder:ByDate")
	Do rs.Execute(dt)
	While rs.Next() {
		Set
po=##class(Shop.POrder).%OpenId(rs.GetDataByName("ID"))
		Write "【注文番号】"_po.OrderNumber,!
		Write "【顧客名】"_po.Customer.Name,!
		Write "【合計額】"_po.TotalPrice,!

		Set itms=po.Items
		For i=1:1:itms.Count() {
			Write "	【商品名】"_itms.GetAt(i).Product.Name,!
			Write "	【数量】"_itms.GetAt(i).Amount,!
		}

		Write "----------",!
	}
	Do rs.Close()
}
```

%ResultSet クラスのオブジェクトは、%New("*classname*:*queryname*") という書式で作成することにより、与えられたクエリの実行を行えるようになります。

そして、Execute()メソッドに入力パラメータである日付を渡します。
そうすると、Next()メソッドが 1 を返す間、条件にマッチするレコードがありますので、GetDataByName()メソッドで ID を取り出します。
そして、その ID に対して%OpenId()で、POrder クラスのインスタンスをオープンし、その後は、そのインスタンスに対してアクセスすることでデータを取得しています。

このメソッドの実行例は以下の通りです。

（引数： ”MM/DD/YYYY” の形式の日付を指定）

```
do ##class(Shop.POrder).printOrderByDate("11/09/2012")
```