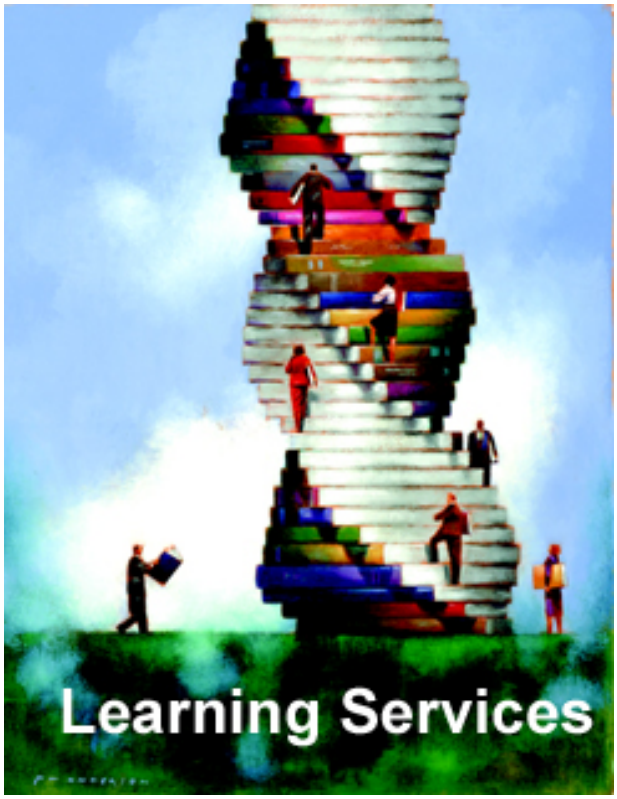


ObjectScript ガイド

（IRIS 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

IRIS はデータベースであると同時に、強力な開発用スクリプト言語を装備しています。
このスクリプト言語を ObjectScript と呼んでいます。
ObjectScript はオブジェクト指向プログラミングをサポートしており、また IRIS データベースへの SQL アクセス、多次元アクセス（ダイレクトアクセス）を行うことができます。
また、数値演算、文字列処理などのユーティリティ関数を豊富に用意しています。

2. Object Script とは

IRIS 用のスクリプト言語の一つである ObjectScript が採用している言語体系は ANSI/JIS で標準化された言語がベースになっており、言語としての完成度は非常に高いものになっています。
また、インタプリタの様な即時実行機能を備えているため、短期間で習得しやすいのが特徴です。

主な ObjectScript の用途には次のようなものがあります。
- サーバサイドアプリケーション記述
- データベース操作
- 埋め込み SQL のホスト言語としての記述
- クラスへのアクセス、クラスのメソッド記述
- Web アプリケーションのロジック記述

すなわち ObjectScript を習得すれば、サーバサイドのプログラミングで必要とされる技術を全てカバーできると言えるでしょう。

3. ターミナルを使った ObjectScript の実行

ここでは、ターミナルを使った ObjectScript 実行方法を解説します。
ターミナルは ObjectScript のインタラクティブな実行環境であり、プログラミング、デバッグ、運用関連のコマンドを実行する際等に使用します。

注）ここでは作業領域として既定で用意されているネームスペース”USER”を使用します。
本書を進めるためにネームスペースの概念等を特にご理解いただく必要はありません。
ここでは、データやプログラムの保管場所として”USER”という名前がつけられた場所を使用する、と理解していただければ結構です。
興味のある方はファーストステップガイドを参照してください。

3.1. ターミナルでの即実行

まず、IRIS の環境が起動していることを確認します。

IRIS がインストールされている PC 上では Windows のタスクトレーにキューブ型のアイコンが表示されているはずです。

キューブが青色になっていれば IRIS は起動しています。

万一グレーアウトしていたら「IRIS の開始」をクリックしてください。

キューブのターミナルを選択し、ターミナルを起動してみてください。

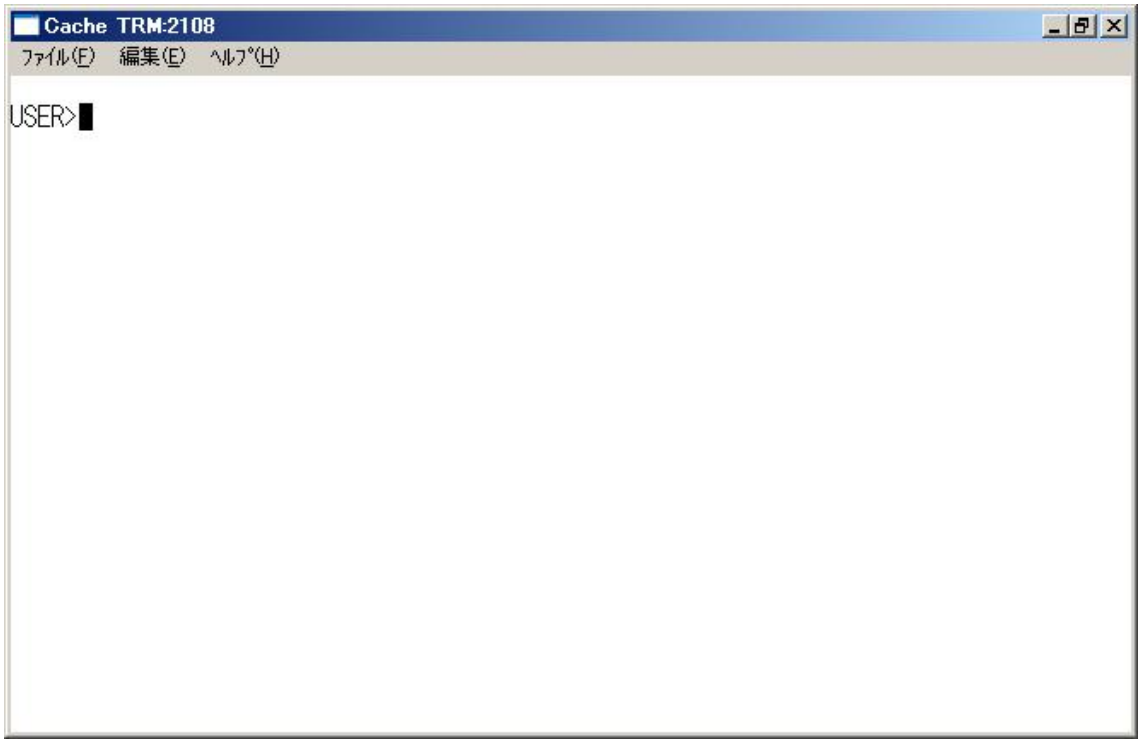


図 1 IRIS ターミナル

図のような表示になっていればターミナル起動は完了です。

Telnet 等と違い、ローカルマシンで稼動している IRIS に自動的に接続まで行います。

プロンプト"USER>"は現在、ネームスペース"USER"を使用中であることを意味します。

この状態で、IRIS のサーバ上で利用可能な機能を実行できます。
試しに下記の下線部のようにタイプしてみてください。
Write と 2 重引用符の間には半角スペースを 1 つ入れます。
タイプ完了後には改行を押下してください。

```
USER>Write "Hello"
Hello
USER>
```

2 重引用符で囲まれた文字列がターミナル上に出力されたことがご覧いただけたかと思います。
注意）これ以降、特に断りが無い限り、各行の終わりでは改行を押下してください

3.2. ターミナルでのプログラム実行

先程は、Write というコマンドを単体で実行しただけでしたが、ターミナルではシステム提供の既存のプログラムやユーザ作成のプログラムの実行も行えます。
以下のように入力してみてください。

```
USER>do ^%SS

                    Cache System Status:  2:41 pm 09 Oct 2008
 Process  Device       Namespace      Routine          CPU,Glob    Pr User/Location
     968                             CONTROL             0,0      7
    5752                             WRTDMN            62,433     8
    1748                             GARCOL              0,0      7
     744                             JRNDMN            6662,0     7
    2328                             EXPDMN              0,0      7
    5328  localhost      %SYS           %SYS.cspServer2
                                                  1818915,1467127   UnknownUser
    4296  //./nul        %SYS           MONITOR          16830,132   7
    2780  //./nul        %SYS           CLNDMN             510,42    7
     588  //./nul        %SYS           RECEIVE          35190,4847  7
    3412  //./nul        %SYS           ECPWork             0,0      7  ECPWORK
```

"^"記号はキャレットです。106 キーボード上では平仮名"へ"の位置にあります。
ここでは、IRIS の環境で起動しているプロセスの一覧を取得するシステムユーティリティである^%SS を実行しました。
ターミナルにはその結果が出力されています。

4. ObjectScript の要素

ここでは ObjectScript を構成する要素を解説します。

4.1. ステートメントとコマンド

多くのプログラミング言語と同様に、ObjectScript のプログラミングは複数のステートメントをロジックにのっとって記述することにあります。
ステートメントは命令とその引数(群)で構成されます。
例えば、先程実行したステートメントはコマンド(Write)と引数”Hello”から構成されています。
実行すると標準出力(この場合ターミナル画面上)に”Hello”と表示します。

```
USER>Write "Hello"
Hello
USER>
```

コマンドはケースインセンシティブ（大文字小文字を区別しない）です。
また、多くのコマンドが省略形を使用できます。
例えば、以下のステートメントは全て同じ結果をもたらします。

```
USER>WRITE "Hello"
USER>write "Hello"
USER>w "Hello"
```

省略形は、ターミナル上でコマンドを実行する際には便利ですが、プログラムを記述する際には、初めの例のように、コマンドは大文字で始める表現のほうが、可読性が増します。

コマンドはその処理内容により、下記のように大別できます。
個々のコマンドの詳細についてはオンラインマニュアルを参照してください。
https://docs.intersystems.com/iris20241/csp/docbookj/DocBook.UI.Page.cls?KEY=PAGE_objectscript

処理内容	コマンド群
プログラムの呼び出し	Do, Quit, Job, Xecute
割り当て	Set, Kill
フロー制御	If{} ElseIf{} Else{}, For{}, While{}, Do/While{}
入出力コマンド	Write, Read

トランザクション関連	Lock, TStart, TCommit, TRollBack
デバイス関連	Open, Use, Close
デバッグ関連	Break, GoTo

以下、ターミナル操作時に知っていると便利なコマンドを説明します。

コマンド名	用途・用例
Do 省略形 D	プログラムの呼び出し USER>Do ^MyProgram
ZLoad 省略形 ZL	プログラム実行コード及びソースコードをメモリに読み込む USER>Zload MyProgram
ZPrint 省略形 ZP	メモリ上のソースコードの画面出力 USER>ZPrint
ZWrite 省略形 ZW	ローカル変数(後述)の内容の画面へのダンプ USER>ZWrite MyLocalVar
ZZDump 省略形 無し	ローカル変数の内容の画面への 16 進数ダンプ USER>ZZDump MyLocalVar
Set 省略形 S	変数への値の代入 USER>Set MyLocalVar=1

4.2. 変数

IRIS には大別するとローカル変数とグローバル変数、プロセスプライベートグローバル変数が存在します。
ローカル変数は、使用中の IRIS プロセスのメモリ空間に格納され、プロセスの終了時に削除されます。
一方、グローバル変数は IRIS データベースに格納される永続性を持つ変数です。
プロセスプライベートグローバル変数は、データベースに格納されますが、プロセスの終了時に削除されます。
ローカル変数、グローバル変数、プロセスプライベートグローバル変数は全てケースセンシティブです。
また、変数は型を持たず、全て Variant 型として処理されます。
事前の宣言も必要ありません。
以下のローカル変数 A とローカル変数 a の値は別の領域に保持されます。

```
USER>Set A=1
USER>Set a=2
USER>Write A
1
```

また、いずれも任意の文字列あるいは数値を添え字にした配列を定義可能です。

```
USER>Set A(100, "Address")="Tokyo"
USER>Set A("任意の文字列")=1000
USER>Set A(100,200,300)= A("任意の文字列")
USER>zw A
A(100,200,300)=1000
A(100,"Address")="Tokyo"
A("任意の文字列")=1000
```

ZWrite（省略形 zw）コマンドは、引数で指定されたローカル変数をダンプするためのコマンドです。
引数無しの ZWRITE コマンドは定義されている全てのローカル変数をダンプします。

補足)その他に、IRIS 操作環境の特定状況を指定・指示する組み込み変数、オブジェクトのプロパティがあります。

4.3. 演算子

IRIS には多彩な演算子が用意されています。
算術演算子、論理比較演算子等の多くは他の高級言語における処理と同様の結果が得られます。

```
USER>Write 3*2
6                    （数値 3 と数値 2 の積が出力されます）
USER>Set a=1,b=2 Write a&b
1                    （1(真)と 2(真)の AND 結果 1(真)が出力されます）
USER>Set a=1,b=0 Write a&b
0                    （1(真)と 0(偽)の AND 結果 0(偽)が出力されます）
```

ただし、四則演算の際の演算の優先度が他言語と違い、必ず左から右に実行されるという特徴があります。
この場合、（）を使って明示的に優先度を指定することで同じ結果を得ることができます。

```
USER>Write 3+2*4
20                   （数値 3 と数値 2 の和と数値 4 との積が出力されます）
USER>Write 3+(2*4)
11                   （明示的に優先度を指定することで同じ結果が得られます）
```

また、IRIS 固有の演算子として文字列の演算子があります。
文字列に対しては、文字列の連結を行う_（アンダースコア）演算子の他に、通常の算出演算子を適用することができます。

```
USER> Set a="ABC",b="DEF" Write a_b
ABCDEF          (文字列”ABC”と”DEF”が連結されます)
USER> Set a=100,b="ABC" Write a+b
100             (文字列”ABC”は数値では 0 とみなされます)
USER> Set a=100,b="23ABC" Write a+b
123             (文字列”23ABC”は数値では 23 とみなされます)
```

この、文字列と数値との演算という処理は他言語ではあまり見られない特徴ですのでご注意ください。
文字列を演算から除外したい場合には、$number や$isvalidnum 演算子が有効です。
例えば$number はその引数が文字列の場合には NULL 文字列が、数字の場合にはその数字が返ります。
一方、$isvalidnum は引数が文字列の場合には 0 が、数字の場合には 1 が返ります。（$NUMBER、$ISVALIDNUM 関数について詳細はドキュメントをご参照ください。）

4.4. 関数・プロシージャ

関数とは、頻繁に実行する処理を記述するコードブロックのことです。
関数は戻り値を持ち、式として扱うことができます。
ユーザ作成の関数のことを特にプロシージャと呼ぶことがあります。
プロシージャの記述方法に関しては後述します。

4.5. サブルーチン

表記方法は関数と同じですが、戻り値を持たないものを特にサブルーチンと呼ぶことがあります。

4.6. ラベル

関数内の特定の位置につけられた名称です。
主にエラー処理時に使用します。
エラー処理に関しては後述します。

5. Visual Studio Code について

今までは、IRIS ターミナルを使用した、即時実行が可能な簡単な動作を見てまいりましたが、実際のアプリケーションを作成するためには、まとまったロジックを実行するための多くのコードを記述する必要があります。
Visual Studio Code はマイクロソフト社が開発したオープンソースですが IRIS の統合開発環境（IDE）として使用できます。
ここでは Visual Studio Code で実施する作業について簡単に触れてみたいと思います。

5.1. ルーチンファイルについて

IRIS のコードは、ルーチンファイルと呼ばれる単位で管理されています。
新規作成、保存、実行のためのメモリへのロードといった操作は全てこのルーチンファイル単位で行われます。
先程の関数と混同してしまいそうですが、ルーチンファイルはなんらかの共通点をもった関数の集合であり、通常、関数とは別の名称を持ちます。
C 言語で言うところの、ソースファイル(*.c)がルーチンファイルに、ソースファイル内に記述された関数が関数に相当します。
ルーチンファイルにはその形式に応じて拡張子を付与します。
Visual Studio Code はその拡張子に応じてルーチンファイルの形式を判別し、シンタックスチェックや適切な動作を行います。

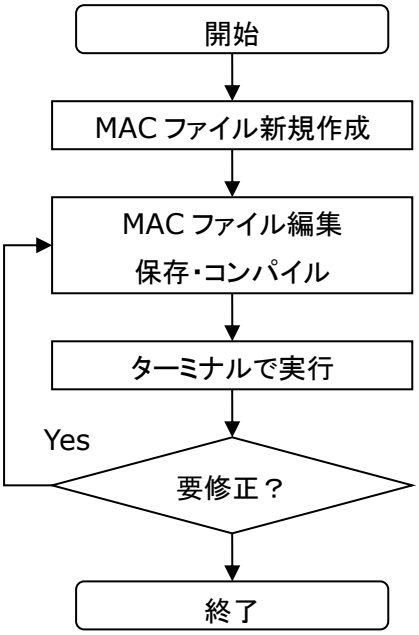
以下の表は拡張子とその用途の概要です。

拡張子	用途
MAC	マクロソースファイル。 MAC ルーチンファイルは、マクロや埋め込み SQL 文、埋め込み HTML の記述が可能です。Caché はコンパイル時にマクロ・プロセッサによって MAC に対して前処理を行い、INT を生成します。
INC	インクルードファイル。 MAC ルーチンファイルにインクルードすることができます。 C 言語のヘッダファイルに相当します。
INT	中間ソースファイル。 MACの前処理の結果により生成されるほか、プログラマが直接作成、編集することも可能です。 このファイルはコンパイラにより OBJ ファイルに変換されます。 単にルーチンと呼んだ場合、この形式（あるいは OBJ）を意味することが多いようです。
OBJ	実行コード。 実行可能な形式（バイトコード）のファイルです。 スタジオでこの形式を操作することはありません。また多くのユーティリティでは既定では表示されないようになっています。

Visual Studio Code でのプログラミングは、新規作成するルーチンの形式の選択から始まります。

5.2. Visual Studio Code を使ったプログラミングから実行までの流れ

以下に Visual Studio Code を使った ObjectScript のプログラミングからターミナルによる実行までの流れを示します。



ここでは、MAC ファイルを新規作成（あるいは既存のファイルをオープン）し、編集・保存・コンパイルした結果である OBJ ファイルを、ターミナルで実行するという大まかな流れをご理解いただければと思います。

6. プロシージャの記述方法

ObjectScript によるプログラム記述スタイルは、大別して 2 つ存在します。
ひとつは旧来の記述スタイルで、これは主に下位互換性確保のためのものです。
今後は、以下のプロシージャ記述スタイルを使うことを強く推奨します。

```
        I $D(NJOB) S NJOB=NJOB+1 ;count of total jobs
        S JOB(J)=Info F X=1:1:11,"P","D" S JOB(J,X)=Info(X)
        ;
        I $D(WID(5)) DO
        . N X S X=$L(Info(5)) I X>WID(5) S WID(5)=X ;width
        Q
```

一方、構造化プログラミングやクラスのメソッド記述等に適した別の記述スタイル（プロシージャ）が存在します。

```
fact(number) PUBLIC
 ; compute factorial
 {
 if number < 1 {quit "Error!"}
 if (number = 1) || (number = 2) {quit number}
 set x = number * $$fact( number - 1 )
 quit x
 }
```

本章ではこのプロシージャによる記述を取り扱います。

6.1. ObjectScript の関数作成及び呼び出し

ここで、「引数として渡された 2 つの値を掛け算した結果を返す関数」を作成してみましょう。
繰り返しとなりますが、ObjectScrip ではプログラムの編集・保存単位をルーチン、戻り値のあるコードブロックを関数、戻り値の無いコードブロックをサブルーチンと呼んでいます。

Visual Studio Code を利用して ObjectScript でプログラミングを開始するためには事前のセットアップが必要です。

詳細は、以下を参照してください。

Visual Studio Codeの使い方

左ペインに表示されるエクスプローラーの src フォルダー（フォルダー階層は、最初にフォルダーを指定する際に図のような構成に必ずしもする必要はありませんが、慣例として src ディレクトリ配下にファイルを置くことが推奨されています。）の所で右クリック新しいファイルをクリックして、名前を以下のように calc.mac とします。

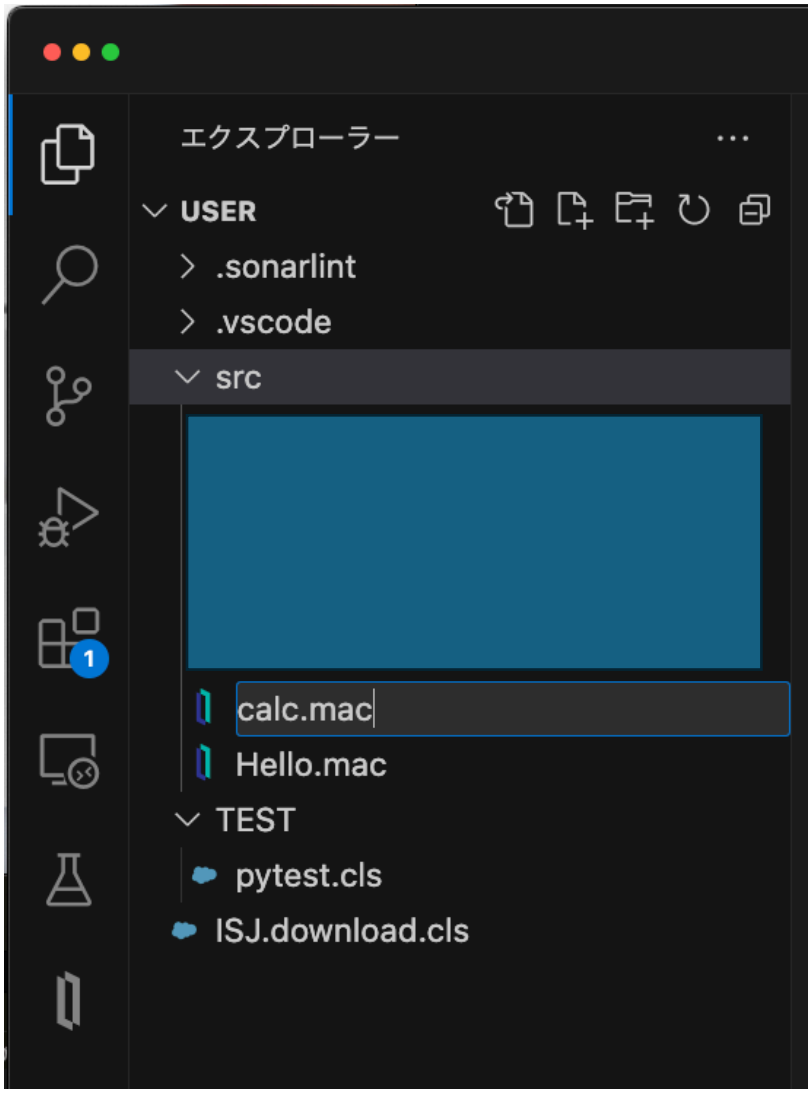


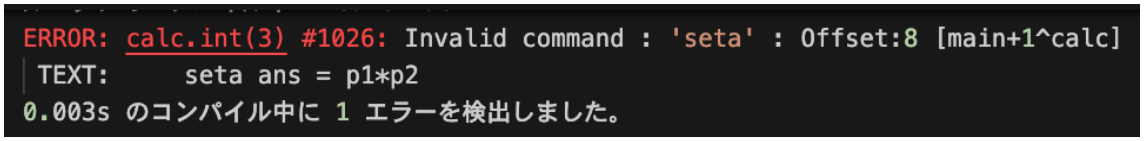
図 2 Visual Studio Code の起動

空白のエディット用ウィンドウが表示されますので、下記のコードを入力してください。

```
main(p1,p2) public {
        //与えられた引数を使った計算をします
        Set ans=p1*p2
        Quit ans
}
```

//はコメント行です。
シンタックスエラーがある場合は、その箇所が赤い波線で強調表示されます。
入力が完了したら、ファイル>保存を押下してください。
これでルーチン calc.MAC が作成されました。

と表示されれば、コンパイルは成功です。



この例は Set コマンドを Seta と打ち間違えた場合の出力です。
御注意いただきたいのは、シンタックスエラーの発生箇所は編集中の MAC ではなく、プリコンパイルにより生成される INT での箇所がレポートされている点です。

ERROR #1026: Invalid command : 'Seta' : Offset:5 [main+2^calc]

と表示されていますが、ここで main+2^calc はエラーの発生箇所を示しており、ルーチン calc 内の関数 main の+2 行目にエラーがあることを示しています。

右クリック View Other で INT を表示できます。
エラー箇所の確認のために INT を表示することがありますが、編集を行う際には忘れずに MAC 表示に切り替えてから、MAC を編集することにご留意ください。

これでルーチンが実行可能な状態になりました。ターミナルを起動して
USER>プロンプトに対し、下記のコマンドを入力してください。$$はインタプリタにプロシージャ呼び出しを式として扱うことを通知するための特殊文字です。

```
USER>set x=$$main^calc(2,3)
USER>w x
6
```

このように表示されるはずです。

ここでは、第 1 引数で数値の 2、第 2 引数で数値の 3 を与えた結果である数値 6 が変数 ans に格納され、それが Quit 命令により戻り値として返されています。前章で ObjectScript は文字列に対して算術演算の実行が可能であるというお話をしましたが、試しに下記のように呼び出しますと

```
USER>set x=$$main^calc(2,"A")
USER>w x
0
```

となります。これは文字列"A"が演算の対象となった場合、数値の 0 として扱われるためです。

6.2. プロシージャのスコープ指定

個々の関数にスコープを指定することができます。
public 指定された関数は他のプロシージャからの実行が可能です。
private 指定された関数は同一ルーチン内からのみ実行が可能です。
省略時は private 扱いになります。

6.3. 変数のスコープ

ここで、現在定義されている全ローカル変数を表示するコマンド zw を再実行してみてください。

```
USER>zw
x=0
USER>
```

インタプリタ上での実行と違い、関数の中で定義された変数のスコープは既定では
その関数の内部のみとなります。
よって変数 ans は表示されません。
ここで、前述のルーチンを下記のように修正してください。

```
main(p1,p2) public {
        Set ans=p1*p2
        Set %ans=ans
        Set a=ans
        Set b=ans
        Quit ans
}
```

同様に実行後、zw を実行しますと、今回は次のように表示されます。

```
USER>zw
%ans=6
x=6
USER>
```

%ans という変数が関数内で定義されたにも関わらず出力されています。
これは%で始まる変数はグローバルスコープを持つ特別な変数として扱われるためです。

さらに、特定の変数をグローバルスコープとして扱うことを指示することも可能です。
対象となる変数を[]記号で指示します。
先ほどの%で始まる変数は暗黙にこの[]指定がなされています。
注意）ここで言うグローバルスコープを持つローカル変数とグローバル変数は全く別のものです

```
main(p1,p2) [a,b]  public {
        Set ans=p1*p2
        Set %ans=ans
        Set a=ans
        Set b=ans
        Quit ans
}
```

同様に実行後、zw を実行しますと、今回は次のように表示されます。

```
USER>zw
%ans=6
a=6
b=6
x=6
USER>
```

以上をまとめますと次のようになります。

変数	スコープ
通常の変数	そのプロシージャ内だけで可視。
%で始まる変数	全プロシージャで可視。
[]指定された変数	[]指定を行っているプロシージャでのみ可視。

6.4. プロシージャの呼び出し

プロシージャ内から他のプロシージャを呼び出すことができます。

```
main(p1,p2) public {
        Set ans=p1*p2 Set msg=$$log("掛け算",ans)
        Quit msg
}
log(header,str) private {
        Quit header_":計算結果は"_str_"です"
}
```

補足）ObjectScript はマルチステートメント記述ができますので、このような可読性を重視したコーディングも可能です。

6.5. 変数のスコープとプロシージャ呼び出しの関係

変数のスコープのルールは、プロシージャ呼び出しが繰り返されたり、ネストされた場合でも有効です。

```
main() [A,B] public {
        Set A=1
        Set B=1
        Set %P=1
        Do subA()
        Do subB()
}
subA() [B] {
        Set X=1
        Set Y=1
}
subB() [P] {
        Set X=2
        Set Y=2
        Set P=1
}
```

以下に、上記の例のようにプロシージャ main()が procA(),procB()を相次いで呼び出した場合の各変数のスコープを図示します。

補足）サブルーチン（戻り値の無いユーザ作成関数）呼び出しには Do コマンドを使用します。

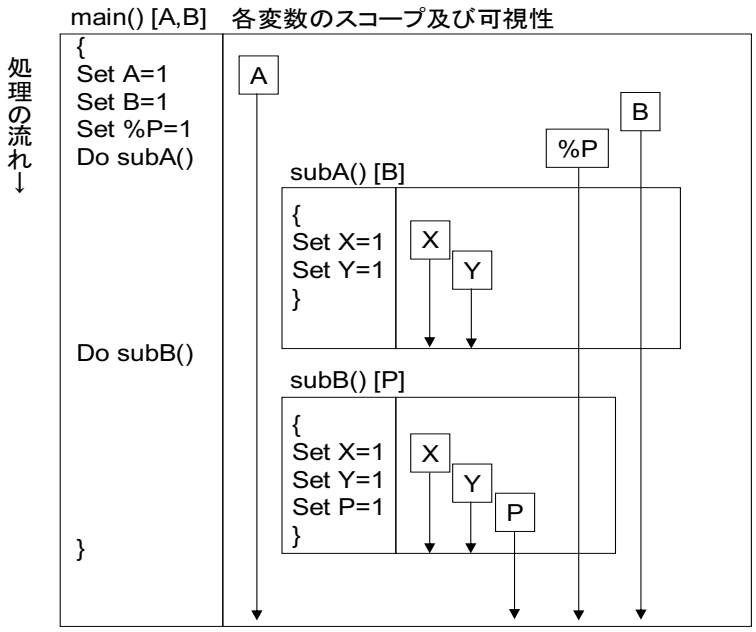


図 3 変数のスコープ

IRIS のローカル変数のスコープはアニメーションのセル画のようなものをイメージするとわかりやすいかもしれません。

背景（public の変数領域）の上にセル画（プロシージャ呼び出し）を重ねて、絵（変数）を書き込んでいくのですが、public 指定した場合は、そのセル画に穴があいていて、実は背景を直接書き換えているような感じです。

背景に書いているのでセル画をとりはずしても、その内容は保持され続けます。

一方、public 指定無しの通常の変数は、セル画を取り外した時に、一緒に取り除かれます。

%付きの変数は常に背景に書き込まれます。

6.6. マクロ定義及びインクルードファイル

MAC ファイルではマクロ定義及びマクロ定義のみを記述したファイル（インクルードファイル）の取り込み指示を行うことができます。

マクロ定義及びその使用は下記のように行います。

calc.mac と同様にエクスプローラーの所でファイル名には calc.inc と入力します。

```
#define TRUE 1
#define FALSE 0
#define DISP(%x) Write %x,!
```

先程の calc.mac を次のように修正して、コンパイルを実行してください

```
#include calc

main(p1,p2,result) [a,b] public {
        Set ans=p1*p2
        Set %ans=ans
        Set a=ans
        Set b=ans
        Set result=$$$TRUE
        $$$DISP("DONE!")
        Quit ans
}
```

ここでは、新たに第 3 引数を追加しています。

IRIS では引数は既定では値渡しですが、呼び出し元で.(ピリオド)を付けて変数を指定することで、参照渡しが実行できます。

計算結果が有効である場合は、この第 3 引数で指定された変数に真($$$TRUE)が設定されます。

```
USER> Set r="",x=$$main^calc(2,3,.r)
DONE!
USER>w r
1
```

また、マクロ定義を使用すれば、デバッグバージョンとリリースバージョンのビルドの切り替えも容易に行えます。

```
#define DEBUG 1
#if $$$DEBUG
        Write "DEBUG",!
#endif
```

7. ObjectScript で良く使う機能

ここでは良く使うコマンドや関数を紹介しています。

	用途・用例
Set	変数への値の割り当て Set myVar=1
Write	カレントデバイスへの出力 Write "ABC",! //!は改行コード出力指示
Do	プロシージャ、サブルーチンの呼び出し Do main^MyProg()
Quit	プロシージャ、サブルーチンを終了 Quit Quit 値
If…ElseIf…Else	条件分岐 If (x=1) { 処理 } ElseIf (x=2) { 処理 } Else { 処理 }

For…Continue
Do…While
	繰り返し
	For i=1:1:100 {
	処理…
	If (条件) Continue //次のループ値から処理を継続
	If (条件) Quit //ループを直ちに終了
	}
	
	Do {
	処理…
	If (条件) Quit //ループを直ちに終了
	} While (条件)

$Length	変数が保持する値の文字数を返す USER>Set x= "ABC" USER>Write $Length(x) 3 USER>Set x= "あいうえお" USER>Write $Length(x) 5

	用途・用例
$Extract	文字列の一部を返す USER>Set x= “あいうえお” USER>Write $Extract(x,3,4) //変数 x の 3 文字目から 4 文字目まで うえ
$H	インターナル表記の現在時刻を返す USER>Write $H 59562,65658
$ZDATETIME	インターナル表記の現在時刻を外部表記に変換 USER>Write $ZDATETIME($H) 01/28/2004 18:48:11
$ZTRAP	エラーハンドラを指定する特殊変数 次章を参照
$ZERROR	最後に発生したエラー情報を保持する特殊変数 次章を参照

8. エラー処理

ここではエラー処理について説明します。
ObjectScript はトラップによるエラー処理を実行できます。
プログラマはエラーハンドラ（エラー処理のための専用の関数）を作成し、各呼び出しレベルに応じてエラーハンドラを定義することで、可読性の高いプログラムを記述することができます。

IRIS には 3 種類のエラートラップ機能が備わっていますが、ここでは$ZTRAP と呼ばれる機能を使用した例をご紹介します。

他のオブジェクト指向言語（Java など）と同様のエラー処理（Try/Catch 構文もサポートしています。現在では、この構文を使用したエラー処理を推奨しています。

8.1. $ZTRAP 特殊変数とエラーハンドラ

$ZTRAP は特殊変数と呼ばれる変数の一種で、この変数にエントリを指定することで、エラー発生時にはそのエントリが呼び出される仕組みを提供しています。
まずはエラー発生時の既定の動作を見るために下記のコードを作成し、コンパイルしてください。

```
main(a,b) public {
        Write a/b,!
        Quit
}
```

これを下記のように実行すると 0 除算エラー(<DIVIDE>)が発生します。

```
USER>Do main^calc2(4,0)
Write a/b,!
^
<DIVIDE>main+1^calc2
USER 2d1>
```

このように、既定ではエラーが発生した場合、実行がそこで中断され、デバッグモードと呼ばれる状態に切り替わります。
ここで
<DIVIDE>main+1^calc2
はエラーの種類とエラーの発生箇所を示しています。
この場合、<DIVIDE>エラーが calc2 という MAC ファイルに含まれるプロシージャ main の+1 行目で発生したことがわかります。
USER 2d1>というプロンプトは現在デバッグモードにあることを示しています。

注意）実際のアプリケーションは、このようなターミナルからの実行ではなく、バックグラウンドジョブとして起動されるのが通常です。
その場合、デバッグモードにはならずプロセスは直ちに終了します。

次にこのプロシージャにエラー処理を追加した例を示します。

```
main(a,b) public {
        Set $ZTRAP="Error"
        Write a/b,!
        Quit
Error
        Set $ZTRAP=""
        Write "STACK@Error=",$STACK," ERROR=",$ZE," at main()",!
        Quit
}
```

ここで Error はコードの開始位置につけられた名前で、ラベルと呼ばれます。

```
        Set $ZTRAP="Error"
```

この定義により、プロシージャ main 内部(及び main から呼び出される他のプロシージャが存在すれば、その内部)でエラーが発生した場合には、ラベル Error で特定されるコードに制御を渡すように指定できます。
先程と同様に実行してみますと

```
USER>Do main^calc2(4,0)
STACK@Error=1 ERROR=<DIVIDE>main+2^calc6 at main()
USER>
```

となりました。ターミナルには Error ラベル以降の処理結果が表示されています。
また、今回はデバッグモードにはならず、通常の USER>プロンプトに戻りました。

```
        Set $ZTRAP=""
```

エラーハンドラ内でこのように$ZTRAP をヌルで再定義しているのは、エラーハンドラ内でのエラー発生を想定してのことです。
通常エラーハンドラ内でのエラーを誘発しうるような処理の記述は避けるべきですが、万一エラーが発生すると、$ZTRAP が再度作用するので、結果として無限ループ状態に陥ってしまいます。
それを避けるために、エラーハンドラの先頭では習慣的に、$ZTRAP のクリアを行います。

$STACK 特殊変数は参照された時点での呼び出しのネストレベルを返します。

8.1.1. エラーハンドラ完了後の動作

それでは、$ZTRAP の動作をより深く理解するために、プロシージャを次のように修正してください。

```
main(a,b) public {
        Set $ZTRAP="Error"
        Write a/b,!
        Write "End of main",!
        Quit
Error
        Set $ZTRAP=""
        W "STACK@Error=",$STACK," ERROR=",$ZE," at main()",!
        Quit
}
```

これを再度実行しますと、先程と同じ結果となります。

```
USER>Do main^calc2(4,0)
STACK@Error=1 ERROR=<DIVIDE>main+2^calc6 at inside main()
USER>
```

追加した”End of main”を出力する命令はなぜ実行されなかったのでしょうか？

これは、エラーが発生し$ZTRAP に制御が渡ると、そのエラーハンドラ終了(Quit 実行)時には、エラー発生箇所(この場合 Write a/b,!)の次の行ではなく、最後に$ZTRAP が定義されたプロシージャ（この場合 main）を呼び出したレベルにまで制御が戻るためです。

main は端末から Do コマンドで呼び出しましたので、結果として処理が全て完了し、USER>プロンプトに戻ります。

8.1.2. 複数の$ZTRAP 定義時の動作

この動作を理解するために、main を main と Div という 2 つのプロシージャに分割してみましょう。

```
main(a,b) public {
        Set $ZTRAP="Error"
        Do Div(a,b)
        Write "End of main",!
        Quit
Error
        Set $ZTRAP=""
        Write "STACK@Error=",$STACK," ERROR=",$ZE," at main()",!
        Quit
}
Div(a,b) public {
        Set $ZTRAP="Error"
        Write a/b,!
        Write "End of Div",!
        Quit
Error
        Set $ZTRAP=""
        Write "STACK@Error=",$STACK," ERROR=",$ZE," at Div()",!
        Quit
}
```

今回は、$ZTRAP の定義が main 呼び出しレベルと Div 呼び出しレベルの 2 箇所で定義されています。

この場合、エラーが発生した時点から見て直近の$ZTRAP 定義値（この場合 Div での定義）が有効になります。

注意）プロシージャ Div 終了時には main で定義された$ZTRAP の定義が有効になります。

また、Error というラベルが 2 箇所で使用されていますが、ラベルのスコープはそのプロシージャ内のみとなりますので、各々の Error は別のコードとして認識されます。

それでは実行してみてください。

```
USER>d main^calc2(4,0)
STACK@Error=2 ERROR=<DIVIDE>Div+2^calc2 at Div()
End of main
USER>
```

今度は”End of main”が出力されました。

これは、プロシージャ Div 内の Error のエラーハンドラ終了時には、エラー発生箇所(この場合 Write

a/b,!)の次の行ではなく、最後に$ZTRAP が定義されたプロシージャ（この場合 Div）を呼び出した箇所(この場合 main の Do Div(a,b))の次の行に制御が戻るためです。
次の行は”End of main”の出力処理なので、結果としてそれ以降の処理が継続します。

8.2. Try-Catch 構文でのエラー処理

Try-Catch を利用してエラー処理を行うこともできます。
構文の構造は以下の通りです。

```
TRY {
        // 通常処理
}
CATCH errorhandle {
        // エラー処理
}
//Try-Catch　以外の処理
```

エラー発生時、例外ハンドラ（エラー用オブジェクト）が生成され、処理は Catch ブロックへ移動します。
例外用ハンドラは%Exception.SystemException クラスのインスタンスで、以下プロパティを利用して、エラー情報にアクセスすることができます。

プロパティ名	内容
Name	<UNDEFINED>などのエラー名
Code	エラー番号
Location	エラー発生場所の情報 ラベル名+行数^ルーチン名　でエラー発生場所を示します。
Data	エラーによって報告される任意の追加データ

※　%Exception.SystemException クラス詳細については、クラスリファレンスもご参照ください。

例外ハンドラ（エラー用オブジェクト）は、Catch ブロックで指定する変数に格納されます。
Catch ブロックでは、指定した変数を利用して、以下の例文のように、エラー情報にアクセスします。

```
main(a,b) public {
        try{
                // b=0 の場合、エラーになる
                write a/b
        }
        catch err{
                write "ErrorName - ",err.Name,!
                write "ErrorCode - ",err.Code,!
                write "ErrorLocation - ",err.Location,!
                write "ErrorData - ",err.Data,!
        }
        quit
}
```

実行結果は以下の通りです。

```
USER>do main^calc3(1,0)
ErrorName - <DIVIDE>
ErrorCode - 18
ErrorLocation - main+3^calc3
ErrorData -
```

catch ブロックで指定した変数を利用して

変数.プロパティ名

でエラー情報を取得しています。

次のページでは、8.1.2[複数の$ZTRAP 定義時の動作] で確認した 2 つのプロシージャを利用して Try-Catch 構文でのエラー処理方法をご紹介します。

```
main(a,b) public {
        try{
                do Div(a,b)
                write "●  End of main  ●",!
   }
        catch ex{
                write "◆  main で発生したエラー情報  ◆",!
                write "ErrorName - ",ex.Name,!
                write "ErrorCode - ",ex.Code,!
                write "ErrorLocation - ",ex.Location,!
                write "ErrorData - ",ex.Data,!
        }
        quit
}
Div(a,b) public {
        try {
                Write a/b,!
                Write "●  End of Div  ●",!
        }
        catch ex {
                write "◆  Div で発生したエラー情報  ◆",!
                write "ErrorName - ",ex.Name,!
                write "ErrorCode - ",ex.Code,!
                write "ErrorLocation - ",ex.Location,!
                write "ErrorData - ",ex.Data,!
        }
        Quit
}
```

```
USER>do main^calc3(1,0)
◆  Div で発生したエラー情報  ◆
ErrorName - <DIVIDE>
ErrorCode - 18
ErrorLocation - Div+2^calc3
ErrorData -
●  End of main  ●
```

8.2.1. Throw コマンド

Try-Catch 構文では、明示的な例外を発生するための Throw コマンドがあります。
このコマンドを利用すると、Try ブロックから Catch ブロックへ処理を移動させることができます。
Throw コマンドを使用するためには、例外オブジェクトを引数として渡す必要があります。
例外オブジェクトの生成には、%Exception.AbstractException を継承するクラスを使用します。
なお、先ほど説明した例外ハンドラ用クラス：%Exception.SystemException は、%Exception.AbstractException を継承しています。
以下、**%Exception.StatusException** クラスを使用した Throw コマンドをご紹介します。

```
main2(a,b) public {
        try {
         if $get(b)=""!($get(b)=0) {
                set myerror=##class(%Exception.StatusException).%New()
                set myerror.Name="第 2 引数が """" または 0 が指定されています"
                Throw myerror
         }
         write a/b,!
         write "● End of main2 ●",!
        }
        catch ex{
         write "◆ main2 で発生したエラー情報 ◆",!
         write "ErrorName - ",ex.Name,!
         write "ErrorCode - ",ex.Code,!
         write "ErrorLocation - ",ex.Location,!
         write "ErrorData - ",ex.Data,!
        }
        quit
}
```

実行結果は以下の通りです。

```
USER>do main2^calc3(1,0)
◆ main2 で発生したエラー情報 ◆
ErrorName - 第 2 引数が ""または 0 が指定されています
ErrorCode -
ErrorLocation -
ErrorData -
```

Throw コマンドによって、try から catch ブロックへ移動していることがわかります。

9. デバッガについて

ここでは IRIS のデバッガについて説明します。

IRIS のデバッガは Visual Studio Code から起動します。

詳細は、以下のコミュニティサイトの記事をご参照ください。

https://jp.community.intersystems.com/post/vscode-を使ってみよう！ - 5