NAL

プログラミング
リファレンス

Copyright © 1999-2011 ACT corporation.
All right reserved.

はじめに

ＮＡＬについて

NAL は（Network Abstractive Language）の頭文字をとったものです。
インターネットを含む、ネットワーク上の複数のコンピュータを連携してネットワーク・アプリケーションを構築する際に、その手続きおよび概念を簡略化して一元的な取り扱いを記述する目的で開発した言語です。
プログラミングの前提として、オブジェクト指向プログラミングについての基礎知識が必要となります。

著作権等について

本書に記述される内容について、そのすべての権利はアクト株式会社に属しています。
いかなる理由においても、権利者の許諾を得ない流用ならびに引用を禁じます。
詳しくは info@act.ne.jp 宛てお問合せください。

改訂履歴

99/08/05	初版
99/08/28	BREAK ＜レベル式＞、CONTINUE ＜レベル式＞に対応
99/09/07	#include を include と同じに扱う（C,C++互換）
00/04/14	isnull、isblank 演算子追加
00/06/23	組み込み関数に watchdog()を追加
00/09/10	inrange 演算子追加
01/06/03	extern function に VOID 型宣言を追加
01/06/11	%%%…%%%パス文追加、#if にて !演算子対応
01/07/06	extern function の返値書式を追記
01/11/16	継承メソッドでのスコープ解決優先度を変更
02/04/13	例外対応 try-catch 構文追加
02/12/08	カレントスコープ指定子 thishere（Ver4 以降 thisblock）を追加
03/02/13	Wrapper クラス追加、プロクシ型宣言追加、モジュール継承追加
03/04/10	enum オブジェクトを導入
03/10/27	import 文を追加
04/08/07	class コンストラクタ引数へのアクセス修飾指定に対応
05/01/24	“%{ }%”による埋め込み、define operator による演算子定義機能を追加
06/12/01	as 演算子（変数エイリアス）を追加
09/12/07	foreach 文の書式を変更

1 データ構造と表記 10
1.1 静的データの表現 10
1.2 動的データの表現 11
2 データの型と値 14
2.1 型の種類と特性 14
2.2 型宣言 15
2.3 演算による型の変換 16
3 定数 17
3.1 整定数 17
3.2 実定数 17
3.3 論理定数 17
3.4 文字列定数 17
3.5 配列定数、ハッシュ定数 18
3.6 NULL 定数 18
4 変数 19
4.1 ランタイム変数とスコープ 19
4.2 変数宣言リカバリ 19
4.3 未定義変数の扱い 20
5 メソッド 21
5.1 引数 21
5.2 リターン値 22
6 クラス 23
6.1 基本クラス 23
6.1.1 コンストラクタ引数並び 23
6.1.2 フィールド 24
6.1.3 メソッド 25
6.1.4 コンストラクタ 27
6.1.5 ローカル・クラス 30
6.2 派生クラス 31
6.2.1 基本クラス参照 31
6.2.2 基本コンストラクタ呼出し 32
6.2.3 継承とポリモーフィズム 33
6.2.4 継承インスタンス・メソッドにおけるスコープ解決 34
6.2.5 フィールドおよびメソッドの再定義 35
6.2.6 仮想メンバー 36
6.2.7 最終クラス 37
6.3 抽象クラス 37
6.3.1 基本クラスとしての抽象クラス 37
6.3.2 モジュールとしての抽象クラス 38
6.3.3 抽象フィールド 38
6.3.4 抽象メソッド 39
6.4 クラスフィールドとクラスメソッド 39
6.4.1 クラスフィールド 39
6.4.2 クラスメソッド 39
6.5 アクセス修飾一括宣言 40
6.6 クラスメンバー一括宣言 40
6.7 デストラクタ宣言 41
7 ラッパークラス 43

7.1 組込みラッパークラス........ 43
7.2 ラッパークラスの拡張........ 44

8 レコード........ 46

8.1 レコード定義........ 46
8.1.1 フィールド定義........ 46
8.1.2 メソッド定義........ 46
8.2 レコードの拡張........ 46
8.3 インスタンスの生成........ 48
8.4 構造化レコードの扱い........ 49
8.4.1 ハッシュフィールド........ 49
8.4.2 配列フィールド........ 50

9 モジュール........ 51

9.1 多重継承........ 51

10 配列とハッシュ........ 53

10.1 配列........ 53
10.1.1 配列定義........ 53
10.1.2 配列参照宣言........ 53
10.1.3 配列要素の参照と操作........ 54
10.1.4 配列の操作........ 54
10.1.5 配列に対する演算........ 55
10.1.6 メソッドの追加・削除........ 56
10.1.7 配列の拡張........ 57
10.1.8 インスタンス配列........ 57
10.2 ハッシュ........ 58
10.2.1 ハッシュ定義........ 58
10.2.2 ハッシュフィールドアクセス........ 59
10.2.3 予約フィールド........ 59
10.2.4 フィールド要素名の取り出し........ 59
10.2.5 インスタンス化........ 60
10.3 ハッシュと配列の混在........ 60
10.4 配列・ハッシュの消滅........ 60
10.5 配列・ハッシュの複製........ 61

11 オブジェクト........ 62

11.1 オブジェクトの種類........ 62
11.1.1 インスタンス........ 62
11.1.2 オブジェクト項........ 63
11.1.3 実行可能オブジェクト........ 64
11.1.4 静的オブジェクト........ 64
11.2 フィールド定義........ 66
11.2.1 メソッド........ 66
11.2.2 メソッド・フィールド........ 66
11.2.3 プロクシ・フィールド........ 67
11.2.4 Runtime 式・フィールド........ 69
11.2.5 オブジェクト・フィールド........ 70
11.3 オブジェクトの参照と操作........ 72
11.3.1 オブジェクトのアクセス........ 72
11.3.2 オブジェクト参照子........ 72
11.3.3 オブジェクト直接項........ 73
11.3.4 オブジェクト参照項........ 73
11.3.5 メンバーアクセス........ 74
11.3.6 直接メソッド呼び出し........ 74

11.3.7 間接メソッド呼び出し 75
11.3.8 メソッド・インターセプト 76
11.3.9 継承とオーバライド 77
11.3.10 class および super 参照子 77
11.3.11 メンバーの動的変更 78
11.3.12 オプショナル演算子 79
11.4 オブジェクトの共有 81
11.5 オブジェクトの消滅 82
11.5.1 参照解消による自動消滅 82
11.5.2 オブジェクトの強制削除 83
11.5.3 所有参照による連動消滅 83
11.5.4 コンポジション化による連動消滅 83
11.6 オブジェクトの複製 85
11.7 クラス・アサイン（実行時多重継承） 86

12 シリアライズ 87

12.1 オブジェクトのシリアライズ化と復元 87
12.2 クラスおよび関数のシリアライズ化と復元 87

13 スレッド 89

13.1 基本メソッド 89
13.2 同期機構 90
13.2.1 排他 90
13.2.2 イベント通知 90

14 ステートメント 92

14.1 if 文 92
14.2 select 文 92
14.3 switch 文 93
14.4 repeat 文 94
14.5 for 文 95
14.6 while 文 95
14.7 do・while 文 96
14.8 foreach 文 96
14.9 break 文 97
14.10 continue 文 98
14.11 return 文 98
14.12 with 文 99
14.13 eachof 文 100
14.14 create 文 100
14.15 delete 文 101
14.16 enum 文 102
14.16.1 クラス定義内に記述する enum 文 103
14.16.2 enum オブジェクトの比較 103
14.16.3 #if－#endif と enum の組み合わせ 103
14.16.4 enum オブジェクトのフィールド 104
14.16.5 enum オブジェクトのシリアライズ化と復元 104
14.16.6 enum オブジェクトの高度な利用例 104
14.17 exit 文 105

15 特殊構文 106

15.1 tag 文 106
15.2 output 文 106
15.3 pass 文 107
15.4 here-document 文 107

16 プログラム制御文 109
16.1 #using 宣言 109
16.2 #eval による文字列式解釈 110
16.3 include 文 110
16.3.1 エラー例外 111
16.3.2 HTML テンプレートとしての利用 112
16.3.3 #archive 制御文によるアーカイブファイル指定 113
16.4 import 文 113
16.4.1 #library 制御文によるライブラリ指定 113
16.4.2 ライブラリファイル 114
16.5 extern 文 114
16.5.1 extern 関数宣言 115
16.5.2 extern クラス宣言 115
16.5.3 extern オブジェクト宣言 116
16.5.4 extern content 宣言 118
16.5.5 url プロパティ 118
16.5.6 stat、response プロパティ 118
16.5.7 throwable プロパティ 118
16.5.8 redirect プロパティ 119
16.5.9 action プロパティ 119
16.5.10 呼出し時の action オプション 120
16.5.11 extern オブジェクトへの参照と型名 120
16.5.12 サーバプログラムの構成制御 121
16.6 その他のプログラム制御文 121
16.6.1 #if #elif #else #endif の構文制御 122
16.6.2 #charset による文字コード系指定 123
16.6.3 デバッグ支援出力 123
17 例外制御 124
17.1 基本構文 124
17.2 throw 文 124
17.3 try 文 124
17.4 catch 文 125
17.5 finally 文 125
17.6 システム既定例外 125
17.6.1 エラー例外 126
17.6.2 watchdog()関数によるタイムアウト監視 126
17.7 ユーザ定義例外 126
18 式と演算子 128
18.1 引数と arguments 配列 128
18.2 instance 項 128
18.3 object 項 129
18.4 method 参照項 130
18.5 method 直接項 130
18.6 current 項 130
18.7 this 項 131
18.8 thispart 識別子と entity 識別子 131
18.9 super 項 132
18.10 thisblock 項 133
18.11 with 項 133
18.12 root 項 134
18.13 文字列に対する演算 134
18.13.1 結合、削除、リピート演算 134

18.13.2 比較演算 134
18.13.3 文字列定数への式の埋め込み 135
18.14 代入式の扱い 135
18.15 論理式の評価 135
18.16 比較演算の特例 135
18.17 論理演算の特例 136
18.18 文字列評価演算子 136
18.19 swap 演算子 136
18.20 as 演算子（変数エイリアス） 137
18.21 in 演算子 137
18.22 inrange 演算子 138
18.23 instance 検査演算子 138
18.23.1 instanceof 演算子 138
18.23.2 isinstance 演算子 139
18.23.3 isextend 演算子 139
18.24 代替値演算子 139
18.24.1 isvalue 演算子 139
18.24.2 isnull 演算子 140
18.24.3 isblank 演算子 140
18.24.4 代替値演算子と例外 140
18.25 defined 演算子 141
18.26 typeof 演算子 141
18.27 その他の演算子 143
18.28 演算子の優先順位 144

19 ダイナミック・プログラミング 145

19.1 式の実行 145
19.2 ステートメントの実行 145
19.3 ダイナミック・メソッドの実行 145
19.4 ダイナミック・クラスの利用 146

20 エラーレベルとデバッグ 147

21 既定識別子とメソッド 148

21.1 識別子 148
21.2 ブロックのフィールド識別子とメソッド 148
21.3 インスタンスのフィールド識別子とメソッド 148

22 組込み関数 149

23 組込みメソッド 151

23.1 整数、実数 151
23.2 論理値 151
23.3 文字列 151
23.4 オブジェクト 152

24 組込み定数・変数・クラス 153

24.1 ABORT 定数 153

25 ＷＥＢアプリケーション対応 154

25.1 クライアントへのレスポンスヘッダ出力 154
25.2 クライアント・リクエスト解析 154

26 規約 156

26.1 識別名 156

26.2 コメント 156
26.3 予約語 156
26.4 データ表記規約 157

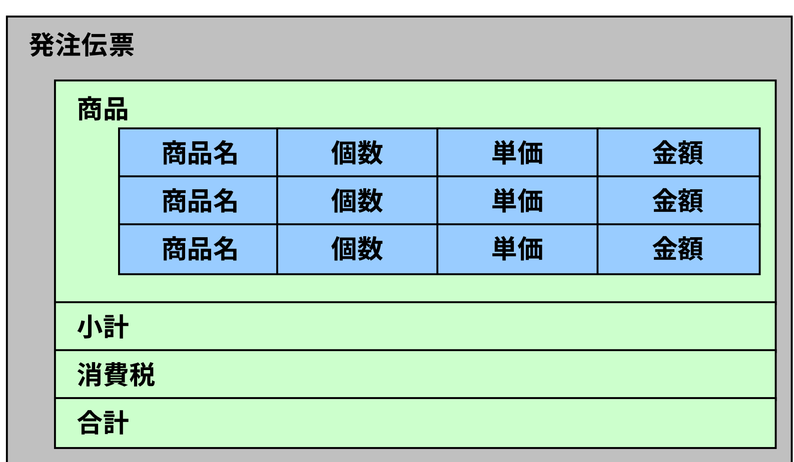
1 データ構造と表記

1.1 静的データの表現

現実の社会で普通に用いられるデータを記述する以下の表現について

発注伝票				
商品	商品名	個数	単価	金額
	品名１	20	1000	20000
	品名２	10	8000	80000
	品名３	2	4000	8000
小計				108000
消費税				5400
合計				113400

そのデータ構造は



としてとらえることができ、各々の矩形がデータとして値を持つ。

このようなデータ構造を表現する規定を以下のように定める。

- データは、≪名前：値≫という書式で表現する。（後述の配列要素を除く）
- まとまりをもったデータ群はコンマ（,）で区切り、｛ と対応する ｝ で括る。

この規定にしたがえば上記のデータはつぎのように表記される。

```
発注伝票: {
        商品: {
                {商品名:"品名１", 個数:20, 単価:1000, 金額:20000},
                {商品名:"品名２", 個数:10, 単価:8000, 金額:80000},
                {商品名:"品名３", 個数:2,  単価:4000, 金額:8000}
                },
        小計:   108000,
        消費税: 5400,
        合計:   113400
        }
```

各々のデータを性格で分けると、「商品名」「個数」「単価」など単独で値をなすものと、「金額」「小計」「消費税」「合計」など他のデータに依存するデータに分けられる。

「金額」の値は 「個数」と「単価」によって決定され、「金額＝単価×個数」という≪式≫として、「個数*単価」と表現する。

「小計」は商品並び（「並び」を「配列」という）各々の「金額」の合計であるが、商品は発注のたびに変動するはずのものであり、「小計は商品配列ごとの金額を合計したもの」として≪手続き≫を規定でき、これを以下のように表現する。

```
小計:   method{
        var 計 = 0;                     // 計を０にしておく
        eachof(商品) 計 += 金額;        // 商品配列要素各々につき、金額を計に加算する
        return 計;                      // 最終的に計の値をを小計の値とする
        }
```

「消費税」と「合計」については、「小計」に依存する≪式≫で表現できる。

以上をふまえて、上記データを表記しなおすと以下のようになる。

```
発注伝票: {
        商品: {
                {商品名:"品名１", 個数:20, 単価:1000, 金額:単価*個数},
                {商品名:"品名２", 個数:10, 単価:8000, 金額:単価*個数},
                {商品名:"品名３", 個数:2,  単価:4000, 金額:単価*個数}
                },
        小計: method {
                var 計 :int = 0;
                eachof (商品) 計 += 金額;
                return 計;
                },
        消費税: 小計 * 0.05,
        合計:   小計 + 消費税
        }
```

このように、静的データ（既にデータが存在するもの）を表現するための規約を NDSN（Nal Data Structure Notation）として規定する。

1.2 動的データの表現

フォーマットが決められている新しい伝票シートに記入していくイメージを考える。

発注伝票				
商品	商品名	個数	単価	金額
小計				
消費税				
合計				

まず最初に、商品データの構造を規定する。

商品名	個数	単価	金額

商品欄を構成する上表においては１行がひとつの商品を表すから、その商品の構成要素は上表の列要素であればよい。
商品名は表記として扱うから文字列、個数と単価は数値、金額は個数×単価として求められる。
これを商品データの仕様とすると、「商品データは、文字列値の商品名、整数値の個数と単価、ア

クセスごとに個数×単価を計算して値とする金額という４つの領域（フィールド）で構成される」となり、それを表すにはデータ構造を規定する≪レコード≫を用いて

```
record 商品データ{
        field   商品名 : string;
        field   個数 : int;
        field   単価 : int;
        field   金額 : method{ return 個数*単価};
        }
```

と記述する。

```
field 商品名 : string;
```

は、「領域 ≪商品名≫ は 文字列データ を保持する」という意味を表す。

次に、発注伝票データの構造を規定する。
発注伝票は、複数の商品データと、それらの金額を合計した小計、これを基にした消費税、そして小計と消費税を加算した合計、という要素で構成される。

発注伝票				
商品	商品名	個数	単価	金額
小計				
消費税				
合計				

商品データの詳細については先に決定したので、ここでは「商品データ」という括りでの配列として用意すればよい。また消費税は、税率が変更されればそれに伴い再計算されなければならない。
以上が、発注伝票というデータについての仕様となる。

発注伝票は、複数作成される場合でもその扱いは同じであり、そのような同じ性格を持ったまとまりを≪クラス≫といい、

```
class 発注伝票(){
        static field 消費税率 : real = 0.05;
        field 商品 : 商品データ[];
        field 小計 : method {
                var 計 : int = 0;
                eachof (商品) { 計 += 金額; }
                return 計;
                };
        field 消費税 : method { return (小計 * 消費税率).toIntValue() };
        field 合計 : method { return 小計 + 消費税 };
        }
```

と記述する。

実際に発注伝票を作るには以下のように記述する。

```
var 伝票 = new 発注伝票();
```

発注伝票に商品を記入するには、記入の仕方を規定した≪record 商品データ≫を利用し、商品の値として、「単価 1000 円の品名１という商品を 20 個」と指定する場合

```
伝票.商品 += new 商品データ{ 商品名:"品名１", 個数:20, 単価:1000 };
```

と記述する。
ここで金額は指定しないが、≪record 商品データ≫の記述に従って自動的に生成される。
同じように

```
伝票.商品 += new 商品データ{商品名:"品名２", 個数:10, 単価:8000};
```

```
伝票.商品 += new 商品データ{商品名:"品名３", 個数: 2, 単価:4000};
```

とすれば、併せて３つの商品について注文伝票に記入したことになる。

次に、作成されたデータの内容を表示する方法の記述を示す。
渡された発注伝票に対し、決められた欄の内容を表示する仕組みをメソッドといい、

```
method  伝票表示( 対象伝票 ){
        eachof( 対象伝票.商品 ) { writeln("%{ 商品名 }% : %{ 金額 }%円"); }
        writeln("合計：%{ 対象伝票.合計 }%円");
        }
```

「対象伝票の、商品欄に記載されている各々の商品データについて商品名と金額を表示し、その後で合計を表示する」という機能を表現する。
伝票の内容を表示するには、具体的な「伝票」というデータを引き渡して呼び出す。

```
伝票表示( 伝票 )
```

伝票自体に表示機能を実装する場合は以下のように記述する。

```
class 発注伝票{
        static field 消費税率 : real = 0.05;
        field 商品 : 商品データ[];
        ＜  中略  ＞
        field 合計 : method{ return 小計 + 消費税 };

        method  表示(){
                eachof( 商品 ){ writeln( "%{ 商品名 }% : %{ 金額 }%円" ); }
                writeln( "合計：%{ 合計 }%円" );
                }
        }
```

この「表示」という機能は個々の発注伝票内に作りつけられたものであり、「伝票の表示機能を呼び出す」という扱いとして実行されるため

```
伝票.表示( );
```

と記述される。

ここまで、静的データと動的データをどのように記述するかについて俯瞰的に解説した。
以下の節では、ここで記述した内容も含め更に詳細な解説をおこなう。

2 データの型と値

2.1 型の種類と特性

データ型を下表に示す。

	型	型名	例
プリミティブ	整数	int	10、-256、0x2AFD
	実数	real	0.0、-0.567、7.657E5、5.678E-3
	数	number	整数または実数
	論理値	bool	true 、false
	文字列	string	"abc123"
オブジェクト	静的オブジェクト	object	object myobject:{a:10, view:method{return a}}
	インスタンス	クラス名 レコード名	new <クラス名>() new ＜レコード名＞{}
	配列	array	{1,2,3,{"A","B","C"},5,6}
	ハッシュ	object	{abc:10 , xyz:"Hello"}
	クラス	class	class Myclass{ .. }
	レコード	record	record Myrecord{ .. }
	メソッド	method	method add(x,y){ .. }
	関数	function	function sub(x,y){ .. }
	NULL	なし	null

整数、実数、論理値、文字列はこれ自体を細分化できない単独のスカラー値であり、これをプリミティブ型という。各々のプリミティブは、その値を評価するためのメソッドを持つ。
例：
“ABC”.substring(0,1) は結果として”A”を返す
(3.14).toIntValue() は整数 3 を返す

静的オブジェクト、インスタンス、配列、ハッシュは内部に、状態値および機能を提供する複合されたデータを内包する。さらにインスタンスの雛型としてのクラス、レコードと機能を提供するメソッドおよび関数を、まとめてオブジェクト型もしくはオブジェクト[1]という。

すべてのデータは型を持った値として扱い、その取り扱いは代入と評価としておこなわれる。
代入は、代表的にはいわゆる左辺値と右辺値として述べる記述であり、“ a=b ”（比較ではない）という記述において “ a ” に対し “ 代入 ” という操作がおこなわれ、“ b ” に対しては “ 評価 ” がおこなわれる。
代入において、プリミティブは値そのものが、オブジェクトはその参照値が代入される。
ともに代入によって左辺値の型は右辺値の型に規定される。

整数の 0 と実数の 0.0 は数学上では同じであるが、別の型として扱われる。ただし、これを文字列化したり表示した場合にはともに “ 0 ” として表現される。
また整数と実数をまとめて数型という。

論理値は true と false の 2 値を取りうる。true を真、false を偽ともいう。

文字という単独の型は存在せず、したがって文字列は文字の配列としてではなく、単一のデータである。また、代入によってその内容が複製されてコピーされる。

[1] クラス・インスタンスに限らず、プリミティブおよび NULL でないものをすべてオブジェクトという。

静的オブジェクト、インスタンス、配列、ハッシュは、その構成要素がさらにプリミティブ値（もしくは他のオブジェクト）の集合であり、その構成要素を単独に操作できる。ただし、変数への代入操作によってその全体をコピーすることはできない。（変数へのオブジェクトの代入は、“参照”という扱いになる）
クラス型とレコード型は new 演算子の適用によってインスタンスを生成することができる。
メソッド型と関数型は実行可能なオブジェクトであり、結果として値を返す[2]。

NULL という型は、実際には何も値を持たないデータであり、いわゆる“空”である。
変数宣言において初期値を指定しない変数は NULL である。また、明示的に null を代入した場合も NULL となる。

2.2 型宣言

ワーニングレベル１以上の指定にて、型を特定する識別名を、変数および引数の宣言時に記述することで、代入実行時にデータ型の整合性を判断する。ただし型を指定しても、null 値はすべてに代入できる。null 値の代入を回避するには代替値演算子および既定値指定を使用すること。

■型宣言の書式と例

書式	記述例	解説
＜名前＞:＜型名＞	name:int	name は int 型
＜名前＞[]	name[]	name は任意型値の配列
＜名前＞[]:＜型名＞	name[]:int	name は int 型値の配列
＜名前＞:＜型名＞[]	name:int[]	name は int 型値の配列

※配列を示す[]内には要素数もしくは要素数範囲を記述できるが機能上は意味を持たない。
例：
name[**10**]
name:int[**0..***]

■型名の記述例と型判定

記述例	解説
all:**variant** all	動的に決定される全ての型を許可（任意型） 型指定しない場合は variant とみなされる
intval:**int**	整数のみ許可
pai:**real**	実数のみ許可
nums:**number**	整数と実数のみ許可
boolval:**bool**	論理値のみ許可
strval:**string**	文字列のみ許可
ary:**[]** ary:**array**;	任意型の要素で構成される配列のみ許可
obj:**object**	オブジェクトのみ許可（一般にはオブジェクト固有の型識別を使用）
cls:**class**	クラス参照値のみ許可
rec:**record**	レコード参照値のみ許可
callable:**method**	メソッド参照値のみ許可
callable:**function**	関数参照値のみ許可

さらに以下の型名も代替として使用できる。

all:**void**	variant の代替
intval:**integer**	int の代替
pai:**double**	real の代替
boolval:**boolean**	bool の代替

[2] 明示的に値を返さない場合も null を返すとみなす。

2.3 演算による型の変換

整数と実数の混合演算においては、整数は実数に変換された後に演算され、結果は実数となる。
整数、実数と文字列の演算（加算記号による結合のみ）においては数値は文字列化されたのちに結合され、結果は文字列となる。
比較演算の結果は論理値となる。

演算に伴う型変換	例	結果
実数と整数	`0.0 + 10`	実数 `10.0` となる
空文字列と整数	`"" + 14`	文字列 `"14"` となる
実数と文字列	`3.56 + "ABC"`	文字列 `"3.56ABC"` となる
比較演算	`100 > 10`	論理値 `true` となる

※整数同士の除算の商は整数であり、実数に型変換されない。もし実数としての商を期待するのであれば被除数あるいは除数を実数化した後に除算をおこなう必要がある。
例：

```
3/2 ⇒ 1
1.0*3/2 ⇒ 1.5
```

3 定数

種類	表記例
整定数	123、-876、0x0FD8、0b1011
実定数	1.23、-4.56e3、1.46e-4
論理定数	true、false
文字列定数	“ABCDE”、“123’ABC’45”、’123“abc”456’
配列定数	{10,10.0,”abc”}
ハッシュ定数	{ a:10,b:10.0,c:”abc”}
ＮＵＬＬ定数	null

3.1 整定数

符号付１０進数および符号なし１６進数、符号なし８進数、符号なし２進数があり、１６進数を記述する場合は”0x”で先導する１６進文字列（0|1|…|8|9|a|b|…|e|f）、８進数を記述する場合は”0o”で先導する８進数表記（0|1|…|6|7）、２進数を記述する場合は”0b”で先導する２進数表記（0|1）とする。
符号なし１６進数、８進数、および２進数は、評価された結果を符号付き整数として扱う。
例えば、表記 0xFFFFFFFF は符合付き１０進数の-1 となる。
１０進定数表記が絶対値で 2147483647 を越える場合は実数として扱われる。

3.2 実定数

符号付１０進少数表記および、＜少数表記＞・＜”E{+/-}整数”＞

3.3 論理定数

true もしくは false

true は論理値“真”、false は論理値“偽”として扱う。

3.4 文字列定数

文字＜"＞（ダブルクォーテーション）もしくは文字＜'＞（シングルクォーテーション）に囲まれた表記可能な文字列で指定する。
例："abc"、"xyz’123"、’abc"123"xyz’
はそれぞれ、“abc”、“xyz’123”、“abc"123"xyz”という文字列定数となる。

文字列定数表記が半角スペース、タブおよび復帰改行文字で区切られて連続する限り、連結したひとつの文字列定数として評価する。
例： "abc" "xyz" は "abcxyz" として評価する。

＜"＞で括る文字列中において文字＜"＞自身を表記する場合は＜¥"＞のように、同様に＜'＞で括る文字列中において文字＜'＞自身を表記する場合は＜¥'＞のようにエスケープ文字＜¥＞を先導する。したがって、文字＜¥＞を表すには常に＜¥¥＞とする必要がある。さらに＜%{＞は＜}%＞と対を成して埋め込みキー[3]として機能するため＜%{＞を文字列として解釈するには＜¥%{＞としなければならない。
エスケープ文字¥n は改行コード(0x0A)、¥r は復帰コード(0x0D)、¥t はタブコード(0x09)、また ¥a(0x07)、¥b(0x08)、¥f(0x0C)、¥v(0x0B)も解釈し、¥x に続く 2 文字までの 16 進表記を、¥に続く 3 文字までの 8 進表記を文字コードとして解釈する。

[3] 「18.13.3文字列定数への式の埋め込み」の節を参照

3.5 配列定数、ハッシュ定数

あるデータを文字“ ｛ ｝ ”で括った場合、そのデータを配列要素とする配列定数として生成する。また複数のデータを、＜,＞（コンマ）で連結して ｛ ｝ で括った場合は、その各々を記述順の配列要素として定義する。
例：{10 , 10.0 , ”10”}

あるデータをフィールド識別子（＜名前＞:）と共に記述した場合はハッシュ定数として生成する。
例：{ a:10 , b:10.0 , c:”10”}

3.6 NULL 定数

NULL 定数は特別な意味を持ち、変数を未設定としたり、変数が未設定かどうかを検査する目的に使用する。
例：

```
var     a = 10;
a = null;
```

は結果として変数 a を未設定とする。
a に格納されていた“値”（ここでは 10）は、この操作によって消滅する。なお、後述のオブジェクトの扱いにおいては重要な意味を持つ。

if(a == null){文}
は変数 a が未設定の時に真となり、文を実行する。

表記上の特例として、“a=;”のように代入式の右辺値を記述しない場合は“a=null;”と同じ扱いとなる。

※null に対する演算はおこなってはならない。

4 変数

変数は代入によって値を「保持」するものであり、フィールド変数とランタイム変数がある。

フィールド変数は、クラスあるいはレコードのメンバーとして規定されるものであり、インスタンス生成時に同時に生成され、インスタンス消滅とともに消滅する。

書式：field 変数名［:型］［=初期化式］;

ランタイム変数は、プログラムのブロック中の任意の場所に宣言され、ブロック内にて宣言を実行した時点で生成し、ブロックを抜け出す時点で消滅する。

書式：var 変数名［:型］［=初期化式］;

＜型＞を省略した場合は代入値に従った型に動的に規定される。
＜初期化式＞を指定しない場合の値は null である。

4.1 ランタイム変数とスコープ

ブロック内にて宣言された変数は、その宣言されたブロックを抜け出す時点で消滅し、スコープはそのブロック内に限られる。
例：

```
var     x = 10;             ①
var     y = x+10;           ②
if(y > 10){
        var     z = x;      ③
        var     p = z;      ④
        }
y = p + x;                  ⑤
```

① 変数 x は整数の 10 に設定される。
② 変数 x は評価され、その値 10 と定数 10 が加算されて変数 y に設定される。
③ if ブロックでは上位階層の変数 x のスコープが及ぶため、変数 z は整数 10 として設定される。
④ 変数 p は変数 z の設定値をコピーされ、その値は 10 となる。
⑤ 下位階層で宣言された変数 p（④）のスコープが及ばないため、変数 p は未定義となる。

スコープ範囲が異なれば同じ名前の変数を定義することができ、実行時のアクセスもスコープに従って解決される。但し、ワーニングレベルを３以上に設定した場合、同名の変数が上位ブロックに宣言されていれば警告する。
例：

```
var     x = 10;
if(true){
        var     x = 100;            // ※警告の対象となる
        writeln(x);                 // 100 を表示する
        }
```

4.2 変数宣言リカバリ

変数名の前に“？”を付加した場合は、その変数がすでにスコープ内に宣言されている場合は、ここでの変数宣言および初期値設定をおこなわない。この用法を変数宣言リカバリという。
例：

```
var     vars = 100;
if(true){
        var     ?vars = 200;        // ※上位に vars があるため無視される
```

```
        writeln(vars);                          // 100 を表示する
}
```

4.3 未定義変数の扱い

未定義の識別名に対し代入操作をおこなった場合、その識別名はグローバル変数（root のメンバー）として生成される。
但しワーニングレベルを 1 以上に設定した場合はワーニングが表示される。

5 メソッド

クラスあるいはレコード宣言内に定義されるメソッドは、インスタンス内にメンバーとして保持される<実行可能オブジェクト>となり、インスタンスを介して呼び出される。
ブロック内に定義されるランタイムメソッドは、ランタイム変数と同様に、定義されたブロック内でのみスコープされる<実行可能オブジェクト>を構成する。

書式： method メソッド名（ [引数並び] ） [:型] ブロック

例：

```
method div(arg1:int,arg2:int=1):int
{
        return  arg1/arg2;
}
```

引数宣言において、arg2 例のように既定値を宣言することにより、呼出し時に実引数を指定しなかった場合にこの既定値をもとにメソッドを実行する。

メソッド呼出し時の実引数は既定の arguments 配列にその順に格納され、仮引数は arguments 配列要素へのエイリアスとして機能する。

メソッド呼出しの結果値はメソッドブロック内で実行された return 式の結果であり、return 文を実行することなくメソッドを抜け出した場合は null となる。

ランタイムメソッドは

```
var div = method(arg1,arg2=1){ return arg1/arg2; };
```

のように、変数宣言における初期値としてメソッド項を代入する記述もおこなえる。

メソッドの型を指定した場合、ワーニングレベルが１以上の場合のみ return 値の型判定をおこなう。ただし、return 文を実行しない（結果が null となる）か null を返す場合は判定の対象とはならない。（null はすべての型に適合する）

ブロックは単独のステートメント、もしくは変数宣言および複数のステートメントを；で区切り全体を ｛｝ で括ったステートメント並びを記述する。
ただし、単一ステートメントで構成する場合は{}で括る必要はない。
例：

```
method div(arg1:int,arg2:int=1):int     return  arg1/arg2;
```

5.1 引数

メソッド呼出し時の実引数が仮引数に満たない場合、残りの引数には null が指定されたものとして扱う。

配列を引数として引き渡す場合、これを“＠”作用子によって引数並びに分解することができる。
例：

```
var     argary = {100,200,300};
var     ret = func(@argary);
```

上記は、

```
var     ret = func(100,200,300);
```

と記述した場合と同じ扱いとなる。

変数の内容ではなく、変数への参照値を引数に渡す場合は“＆”作用子を変数の直前に付加する。

例：

```
method  sigma(total,limit){
        do total += limit; while(limit--);
        }
var     sum = 0;
sigma(&sum,10);
writeln(sum);                                      // 55 を表示する
```

上記例で、メソッド sigma の仮引数 total に引き渡されるのは変数 sum の内容ではなく sum への参照値となる。したがって呼出し時の total は変数 sum へのエイリアスとなり、total へのアクセスは直接、sum へのアクセスとなる。

5.2 リターン値

メソッド呼出しの結果値は return ステートメントに続く式の値となる。
式を伴わない return ステートメントは return null と同じであり、また return ステートメントを実行しないでブロックを抜け出す場合もリターン値は null となる。
例：

```
method divider(x,y){
        if(y != 0)      return x/y;
        }
var     boo = divider(10,2);            // boo は 5 となる
var     foo = divider(10,0);            // foo は null となる
```

6 クラス

クラスには基本クラス、派生クラス、抽象クラスがある。

6.1 基本クラス

書式：
［アクセス修飾］class クラス名（コンストラクタ引数並び）
{
ローカルクラス定義｜フィールド定義｜メソッド定義｜コンストラクタ
}

または

［アクセス修飾］class クラス名
{
ローカルクラス定義｜フィールド定義｜メソッド定義｜コンストラクタ
}

例：

```
class   Car(name="atlus"){
        field   color;
        color = name;
        method  setColor(col){
                color = col;
        }
}
```

上記において、
Car はクラス名
name="atlus" はコンストラクタ引数とその既定値
field color はフィールド定義
color=name はコンストラクタ・ステートメント
method setColor(col){ .. } はメソッド定義
という。

6.1.1 コンストラクタ引数並び

書式：　［アクセス修飾］名前［:型］［=式］
　　　　{,［アクセス修飾］名前［:型］［=式］}

コンストラクタ引数はクラスから生成したオブジェクト（インスタンス）において、フィールドと同等に扱う。したがって、コンストラクタ引数はインスタンス生成時に評価ならびに操作されるのみならず、インスタンス・メソッドにおいても評価ならびに操作が可能である。

アクセス修飾は private、protected、public を指定でき、指定しない場合は public を指定した場合と同じである。
private は同一クラス内からの参照のみ許可し、protected は加えて派生クラス内のステートメントからの参照を許可する。public はすべてにおいて参照を許可する。
アクセス修飾指定に伴うアクセス警告はワーニングレベルを１以上に設定した場合に機能する。

また、単独でもしくはアクセス修飾と合わせて const を指定した場合、この引数は定数となり操作が禁止される。
例：

```
class   Person(const private sex){}
```

```
class   Person(private const sex){}
```

型を指定した場合、実引数渡し時点で型の整合性を検査し、該当しない場合はアボートする。
“＝<式>”として既定値を宣言することにより、呼出し元で実引数を指定しなかったり、また NULL 値を指定した場合、この既定値をもとにコンストラクタを実行する。
例：

```
class   Car(cname:string=”NoName”){
        field   name:string = cname;
}
var     s = new Car();
```

コンストラクタ引数を持たないクラスを定義する場合、コンストラクタ引数宣言部は省略可能である。
例：

```
class   Car{
        field   name;
        field   engine_power;
}
var     s = new Car();
```

コンストラクタ引数はそれ自体がインスタンスのフィールドとして組み込まれ、コンストラクタのみならず、インスタンス・メッソッドからもアクセスできる。また public であれば、インスタンスを参照する変数を介してもアクセスできる。
例：

```
class simple(arg1,arg2){}

var     obj = new simple(10,”Hello”);
writeln(obj.arg1 , obj.arg2);
```

またコンストラクタ引数にアクセス制限を付した場合は、フィールドおよびメソッドと同様隠蔽される。
例：

```
class simple(private arg1,protected arg2){}

var     obj = new simple(10,”Hello”);
writeln(obj.arg1,obj.arg2);                  // ※エラーとなる
```

6.1.2 フィールド

書式：　［アクセス修飾］field フィールド名［：型］［= 初期化式］；

フィールド定義は、フィールド名と必要に応じて初期化式を記述する。
フィールド名はコンストラクタ引数名および後述のメソッド名と重複してはならない。
型を指定した場合は、ワーニングレベルを１以上に設定すると代入がおこなわれる時点で整合性を検査する。初期化をおこなわない場合は null 値を持つ。
例：

```
class Car{
        field   name = "Hanako";
        field   age:int;
        }
```

アクセス修飾は private,protected,public があり、記述しない場合は public として扱う。

private は同一クラス内からの参照のみ許可し、protected は加えて派生クラス内からの参照を許可する。public はすべてにおいて参照を許可する。
final を public または protected と共に指定した場合は拡張クラスにおいて同名のメンバーを再定義するとエラーとなる。
アクセス修飾指定に伴うアクセス警告はワーニングレベルを１以上に設定した場合に機能する。
例：

```
class Car(setname){
        private field name = setname;
        method  getname(){
                return  name;
                }
        }

var     mycar=new car("atlas");
writeln(mycar.getname());               // “atlas”を表示する
writeln(mycar.name);                    // エラーとなる
```

6.1.3 メソッド

書式：　［アクセス修飾］method メソッド名 （［仮引数並び］）[：型] ＜ブロック＞

メソッド名はコンストラクタ引数名およびフィールド名と重複してはならない。
仮引数並びは、仮引数を”,”(コンマ)で区切って記述する。
仮引数の書式は　＜名前＞[: ＜型＞] [= ＜既定値式＞]　であり、型を指定した場合はワーニングレベル１以上に設定すると実引数渡し時点で型の整合性を検査する。
既定値式を宣言することにより、呼出し元で実引数を指定しなかったり、また null 値を指定した場合に、既定値が実引数値として採用される。

アクセス修飾は private,protected,public があり、記述しない場合は public として扱う。
private は同一クラス内からの参照のみ許可し、protected は加えて派生クラス内のステートメントからの参照を許可する。public はすべてにおいて参照を許可する。
final を public または protected と共に指定した場合、拡張クラスにおいて同名のメンバーを再定義するとエラーとなる。
アクセス修飾指定に伴うアクセス制限はワーニングレベルを１以上に設定した場合に機能する。
例：

```
class Car(setname){
        field   name:string = setname;
        private method getname(){
                return name;
                }
        method public_getname(){
                return  getname();
                }
        }

var     mycar=new Car("atlas");
writeln(mycar.public_getname());                ①結果は”atlas”となる。
writeln(mycar.getname());                       ②エラーとなる。
```

メソッドの型を指定した場合、ワーニングレベルを１以上に設定すると return 値の型整合性を検査する。
例：

```
class    Foo{
         method   getname():string{
                  return   100;                     実行時エラーとなる
                  }
         }
```

※注意：型指定に配列型を指定した場合、個々の要素の型判定はおこなわず、配列型であるかどうかの判定となる。

例：

```
method   arymethod(spec:bool):int[]{
         if(spec)         return   {1,"100"}        配列なのでエラーとはならない
         else     return   100;                     配列ではないのでエラーとなる
         }
```

ブロックには全体を“{ }”で括ったプログラム要素並びを記述する。
ただし、単一ステートメントで構成するメソッドはステートメントを“{ }”で括る必要がない。

例：

```
method   myName():string   return "NAL";
```

上記は

```
method myName():string{
         return   "NAL";
         }
```

と同じ構文である。

メソッド内のステートメントは、インスタンスフィールド、インスタンスメソッドならびにコンストラクタ引数、さらにクラスフィールド[4]とクラスメソッドを参照できる。

例：

```
class Car(setname,color){
         field name = setname;
         method getInformation():string{
                  return name + "("+ color + ")";
                  }
         }

var      mycar = new Car("atlas" , "RED");
writeln(mycar.getInformation());         結果は”atlas(RED)”となる。
```

メソッドからの return 値は、return ステートメントを記述しない場合は null となる。
return this は常に、実体インスタンスへの参照値を返す。

例：

```
class Car(setname){
         field name = setname;
         method   getname(){
                  name += "mate";
                  return this;
                  }
         }
```

[4]クラスフィールドとクラスメソッドについては該当する節（39ページ）を参照のこと

```
var     mycar = new Car("atlas");
writeln(mycar.getname().name);          ①結果は”atlas mate”となる。
mycar = new Car("play");
writeln(mycar.getname().name);          ②結果は”play mate”となる。
```

メソッド内にローカルなクラスを定義し、そのインスタンスをリターン値として外部に引き渡すと、インスタンスはクラスから切離され、独立したオブジェクトとなる。
例：

```
class   Car{
        method  powerModule():Engine{
                class   Engine{
                        field   power:int = 230;
                        }
                return  new Engine();
                }
        }
var     mycar = new Car();
var     engine = mycar.powerModule();
writeln(engine.class);                          // null となる
```

この場合、engine の参照するオブジェクトは Engine クラスのインスタンスではなく、Engine 型のオブジェクトとなるため、クラスをアクセスできない。ただし、power フィールドはメンバーとして含まれておりかつ値も保持している。

メソッド内にはさらにメソッド（ローカルメソッド）を定義でき、さらにその内部にもローカルメソッドを定義できる。
ローカルメソッドのスコープはそれを定義しているメソッド内に限定され、外部からアクセスすることはできない。内部メソッドは外部メソッドのスコープを引継ぐ。
例：

```
class   Car{
        field   power:int = 230;
        method  view(prompt:string ="'){        // 引数既定値は ""
                method  getPower():int{
                        return  power;          // インスタンスの power フィールド
                        }
                return  prompt + getPower() + "Ph";
                }
        }
var     mycar = new Car();
writeln(mycar.view("Power="));                  // "Power=230Ph"を表示する
writeln(mycar.view.getPower());                 // エラー
```

6.1.4 コンストラクタ

コンストラクタとは、インスタンス生成時に実行されるステートメントであり、クラス定義においてメソッド定義以外のステートメント全体を指す。これをデフォルト・コンストラクタといい、フィールドの初期化式もデフォルト・コンストラクタと位置付けられる。
さらに以下の書式に従ったオプショナル・コンストラクタをひとつだけ定義できる。

書式１：oncreate（ 仮引数並び ）＜ブロック＞
書式２：クラス名 （ 仮引数並び ）＜ブロック＞

オプショナル・コンストラクタはその定義位置にてデフォルト・コンストラクタとともにコンストラクタとして実行される。
オプショナル・コンストラクタの仮引数は、インスタンス生成時の new 演算子適用に伴う実引数に対応し、そのスコープはステートメント・ブロック内に限定される。※クラス名直後にコンストラクタ引数を定義する場合と異なりインスタンス・メンバーの一部とはならない。
ステートメント・ブロック内に return ステートメントを記述してはならない。

オプショナル・コンストラクタは、インスタンス生成時の引数をインスタンス内のメンバーとして組み込まない場合に使用するとよい。

コンストラクタ引数を定義しないクラスにおいても、インスタンス生成時にコンストラクタ実引数を指定することができる。コンストラクタは実引数を格納している arguments 配列を介して実引数を参照することができ、実引数の個数は arguments.length()によって、各々の引数は arguments[≪引数位置≫] によって参照できる。ここで引数位置の値は 0 が最初の引数に対応する。
※コンストラクト時の arguments 配列はコンストラクトの最後で削除されるためインスタンス・メソッドにおいて参照することはできない。（インスタンス・メソッドにとっての arguments 配列はメソッド呼出しに伴う実引数を保持する。）

以下のクラス定義はどれも同じである
例１： コンストラクタ引数をメンバーとして含む場合

```
class Car(private name:string = ""){
        private field   created:Date = new Date();      // フィールド定義と初期化式
        }
```

例２： arguments 配列を利用する場合

```
class Car{
        private field   name:string = arguments[0] isnull "";  // 第１引数を代入
        private field   created:Date = new Date();
        }
```

例３： コンストラクタ引数をメンバーとして含めない場合（書式１）

```
class Car{
        private field   name:string;
        private field   created:Date;
        oncreate(nm:string = ""){        // オプショナルコンストラクタ
                name = nm;               // field name に 引数 nm を代入
                }
        created = new Date();
        }
```

例４： C++,Java 文法に沿った書式とした場合（書式２）

```
class Car{
        private field   name:string;
        private field   created:Date;
        Car(nm:string = ""){
                name = nm;               // field name に 引数 nm を代入
                created = new Date();
                }
        }
```

上記各々の例示クラスに対して以下を実行して得られるインスタンスは同じである。

```
var    mycar = new Car("Atlas");
```
において、mycar.name の値は"Atlas" となる。
なお、例４はクラス名変更の際に追随してコンストラクタ名の変更を要する。

フィールド定義において初期コンストラクタ値を指定しなかった場合の値は null であり、コンストラクタは一般的に、そのステートメントによってこれら未設定値を決定する目的で記述される。

コンストラクタ・ステートメントは、コンストラクタ引数、クラス・フィールド、クラス・メソッドに加え、既出のインスタンス・フィールドおよびインスタンス・メソッドもアクセスできる。
例：

```
class Car(setname:string){
        field name;                     // 値は null
        method getname(){
                return setname.toUpperCase();
                }
        name = getname();               // name に値を設定する
        }
```

■ コンストラクタでの変数宣言
コンストラクタ部で宣言した変数はコンストラクタの終了と共に消滅し、インスタンス内には残らない。したがってインスタンス・メソッドからアクセスすることはできない。

■ コンストラクタ実行制御
シングルトン・デザインパターンなど、インスタンスを新たに生成せずに既成インスタンスを返す場合、明示的に return ステートメントを記述し値として既成インスタンスへの参照を返す。
例：

```
final class Singleton{
        static field instance:Singleton = null;

        if(instance == null)    instance = this;
        return  instance;
        }

var     inst1 = new Singleton();
var     inst2 = new Singleton();
```
上記例は２度目以降の new 演算子適用において最初のインスタンスを返すため、inst1 と inst2 がともに同じインスタンスを参照する。

return (null)を実行した場合は生成したインスタンスを破棄し、したがって new 演算子適用結果は null となる。
例：

```
final class Singleton{
        static field instanceCount:int = 0;

        if(instanceCount != 0)
                return  null;           // インスタンスを生成しない
        else    instanceCount++;        // 生成したインスタンス数を更新する
        }
```
上記は最初の new Singleton()のみインスタンスを生成し、以降インスタンスを生成しないで null を返す。

コンストラクタ内にて try-catch 節で処理されない throw を発行した場合、そのインスタンスを

生成しないで例外発生を通知する。
例：

```
class	Member{
	static	field	members[0..*]:Member = {};
	if(members.length >= 10)	throw	"Too many Member";
	members += this;
	}

var	ary[0..*]:Member;
try{
	for(var cnt=0; cnt<100; cnt++)	ary[cnt] = new Member();
	}
catch(err){
	writeln(err+":"+ary.length);
	}
```

上記の結果 "Too many Member:10"と表示する。

インスタンスを生成しない場合においても、デストラクタを指定していた場合はインスタンス破棄に伴いデストラクタを実行する。したがってデストラクタを実行したくない、あるいは専用のデストラクタを実行したい場合には下記のような対応が必要となる。またデストラクタ内で例外を発行した場合は、コンストラクタ内で発行した例外値は失われる。
例：

```
class	Member{
	private static	field	instanceCount:int = 0;
	private field	id:int = instanceCount;
	private method	destructorThrow()	writeln("limit="+instanceCount);
	private method	destructorNormal()	writeln("id="+id);
	if(instanceCounter >= 10){
		ondelete(destructorThrow);	// throw 時のデストラクタ
		throw	"Too many Member";
		}
	ondelete(destructorNormal);		// 通常時のデストラクタ
	instanceCount++;
	}

var	ary[0..*]:Member;
try{
	for(var cnt=0; cnt<20; cnt++)	ary[cnt] = new Member();
	}
catch(err){
	writeln(err+":"+ary.length);
	}
```

6.1.5 ローカル・クラス

クラス内に定義されたクラスをローカル・クラスといい、コンストラクタ実行時に局所的なクラス・インスタンスをフィールドに割り当てる用途で使用したり、メソッド内で使用する局所的なクラス・インスタンスを生成する用途で使用する。
例：

```
class	Car{
	class	Engine{
		field	出力:int;
```

```
                field   排気量:int;
                }
        private field   engine = new Engin();
        public  method  getPower(){
                return  engine.出力;
                }
        }
```

ローカル・クラス定義に際し、private もしくは protected キーワードを先導してアクセス制限を指定できる。無指定もしくは public を指定した場合はクラス外からの利用が可能である。
上記例では、Car クラス外から
new Car.Engine()
として Engin クラスのインスタンスを生成できる。

6.2 派生クラス

■書式１
［アクセス修飾］class クラス名(［コンストラクタ引数並び］）: 基本クラス参照
{
基本コンストラクタ呼び出し｜フィールド定義｜コンストラクタ｜メソッド定義
}

■書式２
［アクセス修飾］class クラス名(［コンストラクタ引数並び］）extends 基本クラス参照
{
基本コンストラクタ呼び出し｜フィールド定義｜コンストラクタ｜メソッド定義
}

派生クラス定義においては、基本クラス定義の場合に加え、基本クラス参照と基本コンストラクタ呼び出しの記述が必要である。

6.2.1 基本クラス参照

すでに定義されているクラスの派生クラスとしてクラスを定義する場合、クラス名およびコンストラクタ引数定義に続けて文字 ：（コロン） もしくは extends を記述し、続けて基本クラス名もしくは基本クラスを参照する式を記述する。

■静的な基本クラス宣言の例：

```
class Car(setname){                               // 基本クラス
        field name=setname;
}
class Supercar(setname,setspeed) extends Car{     // class Car の派生クラス
        super(setname);
        field maxspeed = setspeed;
}
```

■動的な基本クラス宣言の例：

```
class Car(setname){                               // 基本クラス
        field name=setname;
}
var     base:class = Car;
class Supercar(setname,setspeed) extends base{    //変数 base が参照するクラスの派生クラス
        super(setname);
```

```
        field maxspeed = setspeed;
}
```

6.2.2 基本コンストラクタ呼出し

書式：　super(引数並び);

派生クラス定義においては、そのコンストラクタ部において基本コンストラクタを実行しなければならない。（基本コンストラクタ呼び出しを明示しない場合の動作については後述する）
基本コンストラクタ呼出しは“super(引数並び)”とし、引数並びには基本クラスが要求する仕様に沿ったコンストラクタ引数を指定する。
基本コンストラクタ呼出しの結果、インスタンス内部に基本インスタンスがメンバーとして構成される。
構成された基本インスタンスは、基本クラスの型名をメンバー識別名とする参照値フィールドによって参照される。またこのフィールドは super という既定フィールド名でもアクセスできる。
例：

```
class Car(setname){
        field name=setname;
        }
class Supercar(name,speed) extends Car{
        super(name);                        // 基本コンストラクタ呼出し
        field maxspeed = speed;
        }
```

基本コンストラクタを実行するまで基本インスタンスおよびそのメンバーをアクセスできない。
基本コンストラクタは随時かつ多重に実行でき、実行ごとに基本インスタンスが再構築される。
例：

```
class freeDate() extends Date{
        super();                                  // new Date()に相当
        field this_week = toGMTString();
        super(getYear(),getMonth()+1,1);          // 今月１日とする
        field month_week = toGMTString ();

        method changeDate(y,m,d){
                super(y,m,d);                     // y 年(m+1)月 d 日とする
                return this.toGMTString();
                }
        }

var dates = new freeDate();
writeln(dates.this_week);
writeln(dates.month_week);
writeln(dates.changeDate(2000,1,1));
```

■コンストラクタ引数をそのまま基本コンストラクタに引き渡す場合
派生クラスにおいて、コンストラクタ引数をそのまま基本コンストラクタに引き渡す場合には @arguments を基本コンストラクタへの引数に指定することができる。
実引数指定における@演算子は配列に作用し、配列要素を順に引数として展開する。
例：

```
class   Person(name:string,addr:string);
class   Member extends Person{
        super(@arguments);
```

```
        }

var     member = new Member("田中","東京都");
```

上記は Member クラスのコンストラクタ実引数を保持する arguments 配列の要素を順に基本コンストラクタ実引数に指定する
super(arguments[0], arguments[1])
と記述した場合と同じである。

■基本コンストラクタ呼出しをおこなわない場合の動作
明示的な基本コンストラクタ呼出しをおこなわない場合は、当該クラスのコンストラクタ終了後に自動的に、コンストラクタ引数を伴った基本コンストラクタ呼出しをおこなう。
例：

```
class   Person(name:string);
class   Member extends Person{}

var     member = new Member("田中");
writeln(member.super.name);         // Person.name ⇒ "田中"を表示する。
```

この場合は

```
class   Member(name:string) extends Person{
        super(@arguments);
}
```

と、明示的に基本コンストラクタを呼び出した場合に相当する。
ただし、基本コンストラクタが呼び出されるのは当該クラスのコンストラクタ終了後であるため、当該クラスのコンストラクタにおいて基本インスタンスを参照することはできない。

6.2.3 継承とポリモーフィズム

派生クラスのコンストラクタとメソッドにおいて、基本クラスの protected もしくは public 宣言されたフィールドとメソッドを参照できる。また基本クラス内にて public 宣言されているフィールドとメソッドは、派生クラスのインスタンスを介して外部からも参照できる。
これを継承という。
※但し基本インスタンス中のメンバーについては、基本コンストラクタ実行後でなければアクセスできない。（基本コンストラクタ実行までは基本インスタンスが存在しないため）
例：

```
class Car{
        private field name;
        protected method setName(arg_name){
                name = arg_name;
                }
        public method getName(){
                return name;
                }
        }

class MyCar(name) extends Car{
        public method mycarName(){
                return getName();
                }
        super();
        setName(name);
        }
```

```
var mycar = new MyCar("Atlus");
writeln(mycar.mycarName());                    // “Atlus”を表示する
writeln(mycar.getName());                      // “Atlus”を表示する
```

派生クラスにおいては、基本クラスで定義されたフィールドおよびメソッドを再定義できる。
これをオーバライドといい、主にポリモーフィズムを実現する手段として用いる。
例：

```
class Vehcle{
        method getSpeed():int   return speed;   // 派生インスタンス内の speed メンバー
        }

class Car(speed:int) extends Vehcle{
        super();
        method getSpeed():string  return Vehcle.getSpeed()+"Km";
        }

class Ship(speed:int) extends Vehcle{
        super();
        method getSpeed():string  return super.getSpeed()+"ノット";
        }

var car = new Car(120);
var ship = new Ship(30);
writeln(car.getSpeed());               // “120Km”と表示する
writeln(ship.getSpeed());              // “30ノット”と表示する
```

メソッド内にて基本インスタンスをアクセスするには super フィールドを指定するかもしくは基本クラス名（例では Vehcle）を指定する。
オーバライドによるポリモーフィズムは主に抽象クラスから具体的なクラスを導出して扱う場合などに効果をもつ。

6.2.4 継承インスタンス・メソッドにおけるスコープ解決

派生クラスが基本クラスのインスタンス・メソッドあるいはクラス・メソッドを継承する場合、そのメソッド内でアクセスするメンバー（フィールドおよびメソッド）のスコープ優先順位を以下に示す。

インスタンス・メソッドを継承する場合、そのメソッド内でのスコープ解決優先順位は
メソッド定義クラス内メンバー　＞　派生クラス内メンバー　＞　基本クラス内メンバー
となる。
クラス・メソッドを継承する場合、そのメソッド内でのスコープ解決優先順位は
メソッド定義クラス内クラスメンバー　＞　基本クラス内クラスメンバー
となる。
例：

```
class Vihcle{
        field name = "noName";
        }
class Car extends Vihcle{
        method getName() return name;
        }
class Mycar extends Car{
        field name = "Atlus";
        }
```

```
class RentAcar extends Car{}

new Mycar().getName()                           // "Atlus" が返される
new RentAcar().getName()                        // "NoName" が返される
```

また、ワーニングレベル２では、abstract 宣言あるいは virtual 宣言を付して名前解決をおこなわない限り、自動的に派生メンバーをアクセスすることはできない。
すなわち、上記例で new Mycar().getName() の結果は"noName"が返される。

6.2.5 フィールドおよびメソッドの再定義

基本クラスにおいて public または protected 属性にて定義したフィールドあるいはメソッドを派生クラスにて再定義した場合、基本クラス内のフィールドあるいはメソッドは派生クラス内のものが優先される。これをオーバライドするという。
※基本クラス内のフィールドを派生クラス内で同名のメソッドとして定義しても、またメソッドをフィールドとして定義してもオーバライドされる。
例：

```
class   Car{
        protected field name;
        public method   getName(){
                return  name;
                }
        }
class   SuperCar extends Car{
        super();
        private field   name;           // Car クラス内の name をオーバライド
        public method   getName(){      // Car クラス内の getName()をオーバライド
                return  super.name;     // 基本クラスの name 値を参照
                }
        }
```

ワーニングレベル１以上では、基本クラス内で final public または final protected で修飾されたフィールドあるいはメソッドを派生クラスにおいて再定義するとエラーとなる。
すなわち、final 修飾子は拡張クラスでのオーバライドを禁止する。

ワーニングレベル２以上では、override プレフィックスをつけて再定義しないとワーニングが発せられる。
例：

```
#w1                                     // ワーニングレベルを１にする
class   Car{
        final protected field name;
        public method   getName(){
                return  name;
                }
        }
class   SuperCar extends Car{
        super();
        private field   name;           // final をオーバライドしたためエラーとなる
        override method getName(){      // オーバライドを明示
                return  super.name;     // 基本クラスの name 値を参照
                }
        }
```

6.2.6 仮想メンバー

メンバー定義において virtual プレフィックスを宣言した場合、このメンバーを仮想メンバーという。仮想メンバーはそれがアクセスされた際には、派生インスタンス内の同名のメンバーがアクセス対象となる。ただしこのメンバーも仮想メンバーである場合にはさらにその派生インスタンス内の同名メンバーをアクセス対象とする。どの派生クラスにも同名のメンバーがない場合にはこの仮想メンバーがアクセス対象となる。

書式： virtual field フィールド名 [：型][= 初期化式]
書式： virtual method メソッド名([引数並び])[：型] <ブロック>

例：

```
class Animal{
        virtual field sex:bool = false;
        method getSex():string{
                return  sex ? "オス" : "メス";
                }
        virtual method getKind():string{
                return  "動物";
                }
        method getName():string{
                return name();
                }
        virtual method information():string{
                return "%s%s %s".import(getSex(),getKind(),getName());
                }
        }
class Cat extends Animal{
        super();
        field sex:bool = true;
        virtual method name(){
                return "名はまだない";
                }
        virtual method getKind():string{
                return "猫";
                }
        }
class Noraneko(argname:string) extends Cat{
        super();
        method name(){
                return argname;
                }
        }

var neko = new Cat();
writeln(neko.getSex());                         // 結果は“オス”
writeln(neko.information());                    // 結果は“オス猫 名はまだない”
writeln(new Noraneko("我輩").information());    // 結果は“オス猫 我輩”
```

※クラス間での仮想メンバーとその実体決定は、名前のみでおこなわれる。

6.2.7 最終クラス

クラス定義に final プレフィックスを先導した場合、そのクラスの派生クラスを作ることができない。
例：

```
final class FinalCls{}
class   Derived extends FinalCls{        ⇒  エラーとなる
        }
```

6.3 抽象クラス

書式：
abstract class クラス名 {フィールド定義｜コンストラクタ｜メソッド定義 }

例：

```
abstract class Car{
        field   color;
        color = "";
        method  setColor(col){
                color = col;
                }
        }
```

上記において、
Car はクラス名
field color はフィールド定義
color="" はコンストラクタステートメント
method setColor(col){ .. } はメソッド定義
である。

抽象クラスは、通常のクラス（コンクリートクラスともいう）のようには単独でインスタンス生成をおこなうことはできないが、派生クラス定義において基本クラス宣言(extends)、もしくはモジュール宣言(includes)することによって、そのクラスのインスタンス生成時に構成要素の一部となる。
※抽象クラスを includes する場合､その抽象クラスの基本クラス(extends 指定したクラス)は無視される。よって、includes して利用する場合の抽象クラスは、単独クラスであるかあるいは他の抽象クラスを includes しているものでなければならない。

6.3.1 基本クラスとしての抽象クラス

派生クラス定義における基本クラスとして宣言した場合、定義するクラスの基本的な要素としての実装を実現する。その点では抽象クラスも通常のクラスと変わりがないが、<u>単独でインスタンスを生成できない</u>ということから、拡張クラスの部品としての性格をもつ。
例：

```
abstract class Animal{
        field   name:string;
        method  getName():string        return  name;
        method  setName(nm:string)      name = nm;
        }

class   Dog(name:string) extends Animal{
        super();
```

```
        setName(name);
        }

class   Cat(name:string) extends Animal{
        super();
        setName(name);
        }

var     dog = new Dog("Pochi");
var     cat = new Cat("Tama");
writeln(dog.getName());                         // “Pochi”を表示する
writeln(cat.getName());                         // “Tama”を表示する
```

上記例においては Dog も Cat も Animal であるという関係を構築する。
この場合、Dog も Cat も Animal で定義されたフィールドおよびメソッドを「使用することができる」（継承する）が、独自にこれを定義でき（オーバライドという）その場合は派生クラス内にて定義したフィールドならびにメソッドが優先される。

6.3.2 モジュールとしての抽象クラス

クラス定義において抽象クラスをモジュール・インクルード宣言（includes）した場合は、その抽象クラスのメンバーは、定義する「クラスの一部をなす」ものとしての性格をもつ。
例：

```
abstract class MusicPlayer{
        field   track;                  // 音楽トラック数
        method  PlayMusic(){}
        }

abstract class DataStorage{
        field   volume;                 // データ容量
        method  DataLoad(){}
        }

class CD_Drive includes MusicPlayer , DataStorage;
class HD_Drive includes DataStorage;
class CD_Player includes MusicPlayer;
```

この例は CD ドライブが音楽再生機器としての面とデータ記憶装置としての面をもっていることを示し、HD ドライブはデータ記憶装置としてのみ、また CD プレーヤは音楽再生機器としてのみ機能することを示す。
結果、CD_Drive インスタンスは MusicPlayer のメンバーと DataStorage のメンバーの両方を保有する。

※注意：includes された複数の抽象クラスに同名のメンバーが含まれる場合、最初に includes したクラスのものが採用される。さらに includes するクラスにて includes されるクラス内に定義されているメンバーを再定義する場合、override 修飾子をつけなければならず、またそのようにオーバライドした場合は includes されたクラスのメンバーを採用しないで、クラス内にて再定義したものを採用する。

6.3.3 抽象フィールド

書式： abstract field フィールド名;

抽象フィールドは、その定義されたクラス内に実体を持たず、派生クラスにて実体定義されるべきフィールドである。派生クラスのインスタンスから呼び出される基本クラス内のメソッドが抽象フィールドを参照する場合、派生クラス内に実体定義されたフィールドを参照する。
派生クラス内に実体定義がない場合は未定義エラーとなる。

6.3.4 抽象メソッド

書式： abstract method メソッド名（ ）;

抽象メソッドは、その定義されたクラス内に実装を持たず、派生クラスにて実装定義されるメソッドである。派生クラスのインスタンスから呼び出される基本クラス内のメソッドが抽象メソッドを呼び出す場合、派生クラス内に実装定義されたメソッドを呼び出す。
派生クラス内に実装定義がない場合は未定義エラーとなる。

6.4 クラスフィールドとクラスメソッド

クラス定義内にて static 修飾で先導する field、method 定義はインスタンス内に保持されずクラス内に形成される。したがって、同一クラスから生成されたすべてのインスタンスにとって、クラスフィールド、クラスメソッドは共有される唯一の資源となる。

6.4.1 クラスフィールド

書式： static field フィールド名［:型］［ = 式］

フィールド定義において、static 修飾をおこなった場合、このフィールドはインスタンス内ではなくクラス内に形成され、これをクラスフィールドという。
クラスフィールドはクラスに唯一つ存在し、クラスメソッドおよびインスタンスメソッドから共有アクセスされる。
初期値を設定する場合、その式はクラス定義時点で評価されるためコンストラクタ引数を参照することはできない。
例：

```
class List(setname){
        static field top = null;              // クラス変数の初期値は null
        field next = top;
        field name = setname;
        top = this;
        }

new List("Jon");
new List("Elza");
writeln(List.top.name);             // 結果は ” Elza ”
writeln(List.top.next.name);        // 結果は ” Jon ”
```

6.4.2 クラスメソッド

書式： static method メソッド名（［引数並び］）［:型］<ブロック>

メソッド定義において、static 修飾をおこなった場合、このメソッドはクラスに属し、これをクラスメソッドという。
クラスメソッドは、そのクラスから生成したインスタンスのフィールドをアクセスすることはできず、クラスフィールドもしくはクラスメソッドのみアクセスできる。
また public であれば、クラスの外部から<クラス名>.<クラスメソッド名>（引数）として、クラスを指定したメソッド呼び出しをおこなえる。
また、インスタンス自体からも、インスタンス・メソッド同様に呼び出すことができる。

例：

```
class Int{
        static method negate(value:int):int     return  -value;
        }
writeln(Int.negate(10));                  // 結果は-10 となる
```

※実装上の留意点
クラスメソッドは単にグローバルな関数と同じであり、上記の例のようにある概念の想定（Int が整数を扱うという概念）する範囲において、メソッド名が操作の概念をあらわすような場合（ユーティリティともいう）には、複数のクラスメソッドを集めて実装することはドキュメント性を高める上でも効果的である。しかしながら、クラス定義においてそのクラスがインスタンス生成を目的としている場合には、インスタンスとは直接関係しないクラスメソッドを収容するとかえってドキュメント性を損なうことになるため、インスタンス生成用のクラス宣言とユーティリティ収容クラス宣言を別におこなうように留意されたい。

6.5 アクセス修飾一括宣言

クラス定義において、各々のフィールド定義、メソッド定義に public、protected、private、virtual、final、abstract といったアクセス修飾子を設定する場合、以下の書式によって個別に、あるいは複数のアクセス修飾子を組み合わせて一括指定することができる。

書式：　アクセス修飾並び ： { ＜フィールド定義｜メソッド定義＞ }

＜アクセス修飾並び＞には、設定するアクセス修飾子をスペースもしくはタブで区切って記述する。
例：

```
private :{
        field   foo:int;
        method  boo(){ ... }
        }
```

上記は

```
private field     foo:int;
private method  boo(){ ... }
```

と同じである。

※ static 修飾は一括宣言内に記述してはならない。
例：

```
private:{
        static  field   foo:int;                ※許されない
        method  boo(){ ... }
        }
```

6.6 クラスメンバー一括宣言

書式：　static:{ ＜フィールド定義｜メソッド定義｜ステートメント＞ }

static: に続くブロックは、その内部に記述したフィールド定義およびメソッド定義を、それぞれクラスフィールドおよびクラスメソッドとして扱う。また、static と共にアクセス修飾を記述した場合は、ブロック内のフィールドおよびメソッドにアクセス修飾が適用される。
ブロック内のフィールド定義などには、さらに個別にアクセス修飾を追加できる。
例：

```
static :{
        private  field  foo:int;
```

```
        public    method boo(){ ... }
        }
```

またクラスメンバー一括宣言ブロック内に記述したステートメントは、クラス定義の時点で1度だけ実行されるブロックであり、クラスフィールドの初期化などに利用する。このステートメントはインスタンス生成時のコンストラクタ・ステートメントとしては扱われない。
例：

```
class Car{
        static:{
                field power;                    // クラス・フィールド
                power = 140;                    // ※クラス定義時に実行
                }
        }
```

6.7 デストラクタ宣言

書式： ondelete(メソッド参照)

デストラクタとは、インスタンス（オブジェクト）が消滅するタイミングで暗黙に実行されるメソッドであり、コンストラクタに限らず、クラス内の任意の位置で動的に宣言できる。
多重に宣言した場合は最後に宣言されたメソッドが有効となる。
また引数をもたない ondelete();によって、デストラクタを無効にすることができる。

注意：デストラクタとして宣言したメソッドは引数を持ってはならない。引数を定義しても呼出し時点で実引数は何も渡されない。
例：

```
class   Usrcls{
        method sweep(){
                writeln("End");
                }
        ondelete(sweep);
        }

var boo = new Usrcls();
boo = null;                             // ※ここで"End"と表示する
```

上記例では、コンストラクタ記述にてデストラクタを宣言しており、p = null;によって Usrcls インスタンスは参照解除されて破棄され、そのタイミングで sweep()メソッドが呼び出される。
なおデストラクタに指定するメソッドは、インスタンス・メソッドでもクラス・メソッドでもよいが、デストラクタの目的から考えればインスタンス・メソッドが望ましい。

注意：インスタンス消滅前にそのクラスが消滅した場合はデストラクタは実行されない。
例：

```
method  generator(){
        class   Usrcls{ // generator メソッド・ブロック内でしか存在しない Local クラス
                method sweep(){
                        writeln("End");
                        }
                ondelete(sweep);
                }
        return  new Usrcls();   // インスタンスをメソッド外に渡そうとするが
        }                       // ここで Usrcls が消滅しインスタンスではなくなる
```

```
var boo = generator();            // クラスから切り離されたインスタンスを受け取る
boo.sweep();                      // ただし sweep メソッドは存在し、"End"と表示する
boo = null;                       // ※デストラクタは実行されない
```

7 ラッパークラス

Wrapper クラスは、カプセル化された値（以下内包値という）を内部に保有するインスタンスを生成するクラスである。プリミティブをオブジェクト化して扱う場合などに利用する。

7.1 組込みラッパークラス

以下のラッパー基底クラスがあらかじめ用意されている。

クラス名	コンストラクタ引数型	備考
Integer	整数または Integer インスタンス	整数値を Wrap
Real	実数または Real インスタンス	実数値を Wrap
Number	整数、実数または Number インスタンス	整数または実数を Wrap
Boolean	論理値または Boolean インスタンス	論理値を Wrap
String	文字列または String インスタンス	文字列を Wrap
StringBuffer	文字列または String インスタンス	文字列編集機能を提供
Exception	任意の値	例外値として使用する
Variant	任意の値	別名として Wrapper

組込みラッパークラス・インスタンスは以下の共通フィールドおよびメソッドをもつ。

インスタンスフィールド	説明
Value:variant	カプセル化された値

インスタンスメソッド	説明
getObject():this	このオブジェクトへの参照を返す
getValue():variant	内包値を返す
setValue(val:variant)	内包値として val を格納する
equalTo(ref):bool	ref と、クラスおよび内包値が等しい場合 true
sameAs(ref):bool	ref と、内包値が等しい場合 true
toString():string	"wrapper ＜クラス名＞{＜内包値表記＞}"を返す

クラスに固有なメソッドについては「プリセットクラス　リファレンス」を参照のこと。

■equalTo()メソッドはインスタンスと、引数に与えたオブジェクトが同じクラスかつ同じ値を持つ場合のみ true を返す。
例：

```
class   Number extends Integer;
var     wrap1 = new Integer(100);
var     wrap2 = new Integer(100);
var     wrap3 = new Number(100);
writeln(wrap1.equalTo(wrap2));          // true を表示
writeln(wrap1.equalTo(wrap3));          // false を表示
writeln(wrap1 == wrap2);                // false を表示
```

■sameAs（）メソッドはインスタンスの内包する値と、引数に与えたオブジェクトの値が同じ値を持つ場合に true を返す。引数にプリミティブ値を指定してもよい。
例：

```
class   Number extends Integer;
var     wrap1 = new Integer(100);
var     wrap2 = new Integer(200);
var     wrap3 = new Number(100);
writeln(wrap1.sameAs(wrap2));           // false を表示
writeln(wrap1.sameAs(wrap3));           // true を表示
```

```
writeln(wrap1.sameAs(100));                // true を表示
```

※注意：class Variant およびその派生クラスのインスタンスにプリミティブ以外の値（オブジェクト）を内包している場合、equalTo()、sameAs()で true が返されるのは同じ参照値を持つ場合のみである。

7.2 ラッパークラスの拡張

ユーザ定義のラッパークラスは、組込みラッパークラスあるいは他のユーザ定義ラッパークラスから派生して定義する。すなわち基底クラスに組込みラッパークラスがあればよい。
ラッパークラスから派生するクラスを定義する場合、一般にコンストラクタ引数は記述しない。
また、フィールドならびにメソッドを記述しない場合は定義ブロックも省略できる。
例：

```
class   整数 extends Integer;
または
class   整数:Integer;

var     p = new 整数(100);
writeln(p.Value);                          // 100 を表示する
p.setValue(200);
p.Value += 100;
writeln(p.getValue());                     // 300 を表示する
```

ラッパークラスからの派生クラスにおいてコンストラクタを記述する場合は、基本クラスのメンバーを参照する以前に super(@arguments) ステートメントを記述して基本クラスとしてのラッパーインスタンスを初期化しなければならない。
例：

```
class   MyString extends String{
        super(@arguments);
        field   charLen:int = length(); // ※length()は String のメソッド
        }

writeln(new MyString("こんにちは").charLen);    // 5 を表示する
```

ラッパークラスからの派生クラスにおいては独自のフィールドおよびメソッドを定義できる。
またその扱いは一般のクラス定義と同じである。
インスタンス・メソッドならびに外部からインスタンスを介して、内包する値のメソッドを利用できるが、クラス内にて独自メソッドを再定義（オーバライド）した場合は、オーバライドしたメソッドを呼び出す。
例：

```
class   整数:Integer{
        method  toString(){
                if(getValue() < 0)      return  "▲"+(-getValue());
                return  super.toString();
                }
        method  toChar(){
                return  "？";
                }
        }

var     value = new 整数(0x889F);
```

```
writeln(value.toChar());                    // 整数クラスの toChar()メソッド
writeln(value.Integer.toChar());            // プリミティブ int 値の toChar()

value = new 整数(-100);
writeln(value.toString());                  // 整数クラスの toString()メソッド
writeln(value.getValue().toString());       // プリミティブ int 値の toString()
writeln(value.Integer.toString());          // Integer クラスの toString()※
```

※toString()はラッパークラスに限らず、オブジェクトの既定メソッドとして、そのオブジェクトの構造を表示するためである。

ラッパークラスの定義において、内包する値固有のメソッドをオーバライドした場合、値本来のメソッドを利用するには getValue()によって一旦値を得てから適用するか、括弧で括って単独項として評価した後でメソッドを適用する。

例：

```
class   文字列:String{
        method  indexOf(str){ .. }
        };
var     name = new 文字列("山田花子");
writeln(name.getValue().indexOf("花子"));     // 2 を表示する
writeln((name).indexOf("花子"));              // 2 を表示する
writeln(name.indexOf("花子"));                // ※文字列クラスの indexOf()を呼び出す
```

Value フィールドへの演算は、直接インスタンス内の値に作用する。

例：

```
var     s = new String("ABC");
var     t = s.getObject();                  // t にはインスタンスの参照値が代入される
writeln(t);                                 // wrapper String{"ABC"}を表示
var     u = s.Value;                        // s の参照するインスタンス保有値を代入
s.Value += "XYZ";                           // s の参照するインスタンスの保有値を操作
writeln(t.getValue());                      // "ABCXYZ"を表示する
writeln(u);                                 // "ABC"を表示する
t.setValue("12345");                        // t の参照するインスタンス保有値を置換
writeln(s.Value);                           // "12345"を表示する
```

8 レコード

レコードはクラスと同様、インスタンスを生成するためのテンプレートとして機能する。
レコード定義には基本レコード定義と拡張レコード定義がある。

8.1 レコード定義

書式： record レコード名{ フィールド定義 [｜メソッド定義] }

8.1.1 フィールド定義

書式： field フィールド名 [：型] [= 式] ;

フィールド定義は、フィールド名と必要に応じて式を記述する。
フィールド名は既出のメンバー名（フィールド名、メソッド名）と重複してはならない。
型を指定した場合、ワーニングレベルが１以上に設定されている場合のみ代入がおこなわれるたびに整合性を検査する。

式は、インスタンス生成時にフィールドの既定値を定めるものであり定数もしくは定数項のみで構成される式でなければならない。

例：

```
record Car{
        field   name:string;                    // 規定値はnull
        field   color:string = "WHITE";         // 規定値は"WHITE"
        field   power:int = 100;
        }
```

8.1.2 メソッド定義

メソッドについての詳細はクラスのメソッドと同様である。
例：

```
record  Person{
        field   name:string = "";
        field   age:int = 0;
        field   sex:bool = true;

        method  isMale(){
                return  sex;
                }

        method  isAdult(){
                return  (age >= 20);
                }
        }

var     jon = new Person {name:"Jon", age:24, sex:true};
if(jon.isAdult() && jon.isMale()) writeln(jon.name,"は成人男性です");
else    writeln(jon.name,"は女性かまたは未成年です");
```

8.2 レコードの拡張

拡張レコードは基本レコードに対してフィールド追加を含む仕様の追加・変更を目的として利用されるものである。

拡張レコード定義は、≪record レコード名≫に続けて文字 ：（コロン）もしくは<includes>を記述し、続けて基本レコード名を記述する。

例：

```
record Car{                          // 基本レコード
        field   name:string;
        }
record Supercar includes Car{                // record car の拡張レコード
        field   maxspeed:int;
        }
```

上記は、以下の定義をおこなった場合と同じ扱いとなる。

```
record Supercar{
        field   name:string;
        field   maxspeed:int;
        }
```

拡張レコード定義において基本レコードのフィールドを再定義した場合、拡張レコードでの定義にリプレースするが、この場合<override>キーワードを明示的に付加しなければならない。
Ver3.0 以上では override キーワードを付加しない場合ワーニングレベル２以上にて警告を発する。また、基本レコードに定義のないフィールドに対し、override キーワードを付加した場合も警告を発する。
例：

```
record  Message{
        field   name:   string;
        }
record  ExtMessage includes Message{
        override field name: string = "未設定";
        }
```

拡張レコード定義において基本レコードで定義されたメソッドを再定義した場合、拡張レコードで定義した内容にリプレースされる。
例：

```
record  Base{                        // 基本レコード
        field   name:string = "";
        method  view(){
                writeln("name=",name);
                }
        }

record  Ext includes Base{                   // base の拡張レコード
        field   age:int;
        field   sex:bool;
        override method view(){
                writeln("name=",name," age=",age," sex=",sex ? "male":"femail");
                }
        }

var     tom = new base{name:"TOM"};
var     jon = new ext{name:"JON",age:18,sex:true};
tom.view();                                     // “ name=TOM ” と表示する
jon.view();                                     // “ name=JON  age=18  sex=male ” と表示する
```

Ver3.0 以上ではワーニングレベル２以上の場合に、オーバライド定義時に override キーワードを付加しないと警告を発する。また、基本レコードに定義されていないメソッドに対し override キーワードを付加した定義をおこなった場合も警告を発する。

8.3 インスタンスの生成

レコード・インスタンスは、レコード定義に対し new 演算子を適用して生成する。
書式： new レコード名 { フィールド初期化子並び }

<フィールド初期化子>≡<フィールド名:式>｜<フィールド名=式>｜<ハッシュ参照>

例：

```
record Car{
        field   name:string = "X";
        }

var     rec1:Car = new Car;                                    ①
var     recref:record = Car;
var     rec2:Car = new recref;                                 ②
var     rec3:Car = new Car { name:"atlus" };                   ③
```

①は field name の値が規定値”X”のレコード・インスタンスを生成する。
②は recref という変数の参照するレコード定義に従ったレコード・インスタンスを生成する。
①、②のように new <レコード名>; もしくは new <レコード参照>;とした場合は、生成したインスタンスの各フィールドは定義内の式の値で初期化される。

③は、インスタンス内のフィールド値を置換指定する例である。この例の場合、rec3.name は規定値"X"から”atlus”という値に置き換えられる。

インスタンスは先に定義に従ったインスタンスを生成し、その後にフィールド初期化子で示す同名のフィールド値を置換する。
複数のフィールドを同時に置換設定する場合は各々を<,>（コンマ）もしくは<;>（セミコロン）で区切る。フィールド順序はレコード定義の順序である必要はない。
例：

```
record Car{
        field   name:string="X";
        field   power:int=100;
        }

var     rec:Car = new Car{ power:150 , name:"atlus" };
```

または

```
var     rec:Car = new Car{
                power = 150;
                name = "atlus";
                };
```

また、ハッシュまたはオブジェクトを指定して対応するフィールドの値を取り込むこともできる。
例：

```
var     hash = {name:"atlas" , power:150};
var     rec = new Car{ hash };
```

拡張レコード生成の場合を以下に示す。
例：

```
record Car{
        field   name:string="X";
        field   power:int=100;
        }
record Supercar includes Car{
        override field power:int=200;
        field   maxspeed:int;
        }

var     rec:Supercar = new Supercar{name:"atlus" , maxspeed:170};
```

上記の例で、Supercar レコード定義において、基本レコードに定義してある field power を再定義しているが、この場合拡張レコードの定義が優先され、この例では rec.power の値は 200 となり、同名のフィールドが複数生成されるわけではない。
なお、フィールド再定義に際しては override キーワードを付さないとワーニングレベル２以上では警告する。

8.4 構造化レコードの扱い

8.4.1 ハッシュフィールド

レコードのフィールドがさらに複数のフィールドで構成される場合の定義方法を示す。
例：

name		"atlus"
engine	power	100
	type	"DOHC"
	cylinder	6
color		"RED"

に対応するレコード定義は、

```
record Car{
        field   name:string;
        field   engine : object{
                field   power:int;
                field   type:string = "DOHC";          // 既定値は"DOHC"
                field   cylinder:int = 4;              // 既定値は 4 気筒
                };
        field   color:string;
        }
```

とし、レコードの生成は、

```
var carrec = new Car{
        name: "atlas",
        engine.power: 100,                  // engine.power = 100, でもよい
        engine.cylinder: 6,                 // 既定値の 4 気筒を 6 気筒に変更
        color = "RED"
        };
```

のようにおこなう。

8.4.2 配列フィールド

レコードのフィールドが配列である場合の定義方法を示す。
例：

name		“atlus”
option	[0]	“RADIO”
	[1]	“Power Window”
	[2]	“Air Conditioner”

に対応するレコード定義は、

```
record Car{
        field   name:string;
        field   option[]:string;
        }
```

あるいは、レコード定義中に既定値を設定するには、

```
record Car{
        field   name:string;
        field   option[]={"Radio","Power Window","Air Conditioner"};
        }
```

のように初期化配列定数を記述する。

レコードの生成は

```
var carrec = new Car{
        name: "atlus",
        option: {"CD Audio","Power Window","Air Conditioner","Car Navi"}
        };
```

あるいは既定値を置き換えたり要素を追加して

```
var carrec = new Car{
        name:"atlus",
        option[0]: "CD Audio ";
        option[3]= "Car Navi";          // 要素追加は = 演算子を使用する
        };
```

とすれば、option フィールドを設定値で置換する。
var carrec = new Car{ name:"atlus" };
とした場合は、option フィールドにはレコード定義で指定した初期値が設定される。

9 モジュール

書式：
```
module モジュール名{
<フィールド定義>｜<メソッド定義>
}
```

モジュールは一意の概念として捉えられるものとしてまとめたものであり、値を保持するフィールドと機能を提供するメソッドにより構成される。

9.1 多重継承

クラス拡張による継承に関しては単一継承しか扱わないが、モジュール・インクルードによって多重継承と同様な機能拡張を実現できる。

書式：
```
class クラス名［extends 基本クラス］ includes モジュール名並び { <クラス定義> }
```

クラス宣言においてモジュールを includes 指定することによって、そのモジュールの構成要素（フィールドとメソッド）を、クラス内に記述した場合と同じ働きをする。
includes に続くモジュールはコンマ（,）によって複数指定でき、この場合をモジュール多重継承という。また、モジュールも更に他のモジュールを includes できる。
例：

```
module  デジタルカメラ{
        field   記録メディア;
        method  撮影(){・・・}
        method  電池残量(){・・・}
        }
module  携帯電話{
        method  通話(){・・・}
        method  電池残量(){・・・}
        }
class   カメラ付き携帯電話 includes 携帯電話,デジタルカメラ{・・・}
var     p_phone = new カメラ付き携帯電話();
p_phone.通話();
p_phone.撮影();
```

この例は、カメラ付き携帯電話が携帯電話としてもデジタルカメラとしても機能し、それぞれの性格を有しているものとして定義している。

ここで各々のモジュールで 電池残量()メソッドを定義しているが、includes によって取り込まれるメンバーは同名のメンバーが存在しない場合に限られる。したがって上記の場合は先に includes される module 携帯電話 内の 電池残量()メソッドが採用される。
※注意：モジュール・インクルードで取り込まれたメンバーをクラスでオーバライドする場合は、override キーワードを明示的に指定しないと多重定義となる。

上記の カメラ付き携帯電話 クラスは、以下の定義をおこなった場合と同じである。

```
class  カメラ付き携帯電話{
        method  通話(){・・・}
        method  電池残量(){・・・}
        field   記録メディア;
        method  撮影(){・・・}
        }
```

※ クラス継承とモジュール・インクルードを共に実装することもできるが、この場合はクラス継承を先に、モジュール・インクルードを後に指定しなければならない。

例：

```
class マルチーズ extends Dog includes Pet{ ... }
```

10 配列とハッシュ

配列とハッシュはともにオブジェクトとして扱われる。
複数の値を内部に保有する“箱”として理解すればよい。
配列は、値各々に一意な“番号”が付されており、その番号によって特定される。
ハッシュは、値各々に一意な“名前”が付されており、その名前によって特定される。
配列およびハッシュ内の値を「要素」という。

10.1 配列

10.1.1 配列定義

書式１：　{ 要素並び }　または　[要素並び]
書式２：　new 型[]{ 要素並び }
書式３：new Array(要素並び)

<要素並び>≡<要素値>｜<要素値>,<要素並び>

書式１は、任意の型をもった<要素値>を、その記述順に配列要素とする配列オブジェクトとなる。
例：

```
{ 1 , 2 , "3" , 4.56 }
[ 1 , 2 , "3" , 4.56 ]
```

書式２は、すべての要素が<型>で示す値であるか null であることを規定する。
この書式では、定義において要素の型が異なる場合、あるいは異なる型の値を要素として代入した場合に、ワーニングレベルが１以上のとき警告を発してアボートする。ただし型に variant を指定した場合はすべての型を許可する。
例：

```
new string[] { "ABC" , "XYZ" , "123" }
```

書式１は、書式２において new variant[]{ 要素並び } と記述した場合に同じである。

書式３は、Array クラスのインスタンスとして配列オブジェクトを生成する。

```
new Array( 1 , 2 , "3" , 4.56 )
```

引数に指定する要素の型は任意であり、また個数は可変である。

10.1.2 配列参照宣言

定義した配列を名前で識別するためには、配列を変数（もしくはフィールド）に割当てる。
この場合、参照対象が配列であることを明示的に指定して誤った代入操作を警告できる。
型名を明示した場合は、初期化式として記述する配列定義において new variant[]{ } のように要素の型を variant とした場合であっても指定した型の要素であるかどうか検査する。但し、個別に要素として代入する値の型については関与しない。

書式１：　var　変数名[] [:型] = 配列定義;
あるいは　field　フィールド名[] [:型] = 配列定義;

書式２：　var　変数名:型[] = 配列定義;
あるいは　field　フィールド名:型[] = 配列定義;
例：

```
var    ary[] = {1,2,”ABC”,4.5,true};
var    ary[] : string = new string[]{"ABC","XYZ"};
var    ary : string[] = new Array("ABC","XYZ");
```

型宣言しない配列の要素には任意のデータ型の値を指定できる。

var ary[]:string; のように配列定義を指定しない宣言は
var ary[]:string = **new string[]**{}; と同じであり、要素を持たない空配列を定義したことになり、null ではない。

10.1.3 配列要素の参照と操作

配列の要素は配列を参照する変数や、配列への参照値を返す関数などの配列参照に対し、０から始まる正整数で指定する要素番号による配列要素指定子によって特定される。配列要素指定子は整数式を[]で括って記述し、この式を要素番号式という。

書式： 配列参照[要素番号式]
例：

```
var     ary[] = { 1 , 2 , "ABC" , 4.5 , true };
var     element = ary[2];                          // element の値は"ABC"となる
method aryref():int[]{
        return  ary;
        }
element = aryref()[3];                             // element の値は 4.5 となる
```

要素番号式に負の値を指定した場合は配列の長さを加算するが、その結果が負になる場合はエラーアボートする。上記例では ary[3]と ary[-2]は同じ要素（4.5）となり、ary[-6]はエラーとなる。

多次元配列の扱いは、配列の階層を利用する。
また多次元配列の要素を取り出す際の記述は、階層ごとの要素番号を順に指定しておこなう。
例：

```
var     ary[] = { { 1 , 2 , 3 } , { 4 , 5 , 6 } , { 7 , 8 , 9 } };
var     x = ary[1][2];
```

上記は３行３列の配列例である。
この例では、x の内容は 6 となる。
上記において ary[1][2]という記述は(ary[1])[2]を意味している。

配列の要素には自由に値を追加できるため、初期状態においては空の配列を宣言するだけでもよい。
空の配列を宣言する場合は、
var ary[]; もしくは
var ary = {}; のように記述する。
この宣言をおこなった時点で要素を持たない配列を生成し、変数 ary は配列参照変数となる。
この後は、ary[10]=100; のような代入式によって新たな配列要素を構成できる。
また、多次元の空配列を宣言するには、 var ary = {{},{}}; のように指定する。

配列宣言においては、以下のような飛び越し設定をおこなうことができる。
例：

```
var     ary={1,2,,4,,,7};
writeln(ary[2]);                       // 結果は null となる
```

上記例では ary[2]は空要素であるため、結果としては null となる。

var ary[**0..***];
var ary[**10**];
のように配列指定子内に要素数を記述できるが、この記述は単にドキュメントとしての意味しかもたない。

10.1.4 配列の操作

変数に配列を代入する場合は、配列をコピーして代入するのではなく配列への参照が代入される。

（オブジェクトの操作を参照）
例：

```
var     ary={{1,2,3},{4,5,6},{7,8,9}};
var     x=ary;
x[1][2] = 0;
writeln(ary[1][2]);                     // 結果は０である。
```

すなわち、ある配列参照変数 ary を別の変数 x に代入することによって、２つの配列参照変数 ary、x が同じ配列の実体を指し示すことになる。
従って、一方の配列参照変数を通じて配列要素を操作した場合、他方の配列参照変数を通じて取り出すと変更された結果を取り出すことになる。

配列は既定フィールド length を持ち、その値は配列の伸長と共に自動的に更新する。
この値は配列内の最後の要素の要素番号＋１の値をもち、これを配列の長さという。
例：

```
var     ary={};
writeln(ary.length);                    // 結果は 0 となる
array[100000]=1;
writeln(ary.length);                    // 結果は 100001 となる
```

なお、length フィールドに整数を代入すると、その値が現在の length 値より小さい場合に限りその値で示す要素番号以降の要素を削除する。ただし、配列要素が全て埋まっていない場合は結果の length 値が指定した値になるとは限らない。
例：

```
var     ary={};
ary[100000]=1;
ary[1]=1;
writeln(ary.length);                    // 結果は 100001 となる
ary.length = 999999;
writeln(ary.length);                    // 結果は 2 となる
```

配列要素がさらに配列である場合の例を示しておく。
例：

```
var     ary={1,2,{1,2},3,4};
writeln(ary.length);                    // 結果は 5 となる
writeln(ary[2].length);                 // 結果は 2 となる
```

10.1.5 配列に対する演算

配列に対する演算として使用できる演算子は以下の５種類である。
このうち、“&=”、“|=”、“^=”は配列を集合として扱う演算子であり、演算後の各要素の順序については演算前と異なる場合がある。

演算子	機能	解説
+=	要素の追加	左辺値の配列に、右辺値の値もしくは配列の要素を追加する
-=	要素の削除	左辺値の配列から、右辺値の値もしくは配列要素を削除する
¦=	和集合	左辺値の配列に含まれない右辺値の配列要素を左辺値の配列に追加する
&=	積集合	左辺値の配列および右辺値の配列に含まれる要素で構成する
^=	排他集合	左辺値の配列および右辺値の配列に含まない要素で構成する

配列演算例

◆ “+=”演算子の例：

ary = {1,2,3} に対し、
ary += 10 の結果、ary は{1,2,3,10}となる。
ary += {2,3,4} の結果、ary は{1,2,3,2,3,4}となる。
ary += {4,{2,3,4},5} の結果、ary は{1,2,3,4,{2,3,4},5}となる。

右辺の値もしくは配列要素をその順に、左辺の配列要素として追加する。
配列そのものを１要素として追加する場合は、配列を要素とする配列（上記例：“{{2,3,4}}”）を指定しなければならない。

◆ “-=”演算子の例：
ary = {1,2,3,2} に対し
ary -= 2 の結果、ary は{1,null,3,2}となる。
ary -= {3,2,4} の結果、ary は{1,null,null,2}となる。

右辺の値もしくは配列要素各々について、左辺の配列要素のうち要素番号のより小さな同じ値の要素を null に置換する。左辺配列に同じ値の要素が複数あっても置換対象となるのは１要素のみであり、一致しない要素については影響を受けない。
※ -= 演算子は該当要素を null に置換するのみであり、この null 要素を除く必要がある場合には、Array.purge()メソッドを適用すること。

◆ “|=”演算子の例：
ary = {1,2,3} に対し
ary |= {2,3,4} の結果、ary は{1,2,3,4}となる。
ary |= 4 の結果、ary は{1,2,3,4}となる。

右辺の値もしくは配列要素各々について、左辺の配列要素に同値の要素がない場合はその値を要素として追加する。
※右辺の配列要素が配列である場合、同一参照値以外は同値とみなさないため追加する。

◆ “&=”演算子の例：
ary = {1,2,3}に対し
ary &= {2,3,4} の結果、ary は{2,3}となる。

左辺の配列要素のうち、右辺の配列要素のどれとも値が一致しない要素を削除して前詰めする。
※右辺の配列要素が配列である場合、同一参照値以外は同値とみなさない。また〝ary &= 2〞 のような単項オペランドは用意されていない。〝ary &= {2}〞とすること。

◆ “^=”演算子の例：
ary = {1,2,3} に対し
ary ^= {3,4,5} の結果、ary は{1,2,4,5}となる。
ary ^= 2 の結果、ary は{1,3}となる。
ary ^= 4 の結果、ary は{1,2,3,4}となる。

右辺の値もしくは配列要素各々と同値の、より要素番号の小さな左辺の配列要素を削除して前詰めし、左辺配列要素に同値のものがない右辺要素あるいは値を要素として追加する。

10.1.6 メソッドの追加・削除

配列にメソッドを追加することができる。
例：

```
var     ary = {1,2,3,4,5};

create  ary.sum = method(){                // メソッドを追加する
```

```
        var     s = 0;
        foreach(item in this) s += item;
        return  s;
        };

writeln(ary.sum());                     // 追加したメソッドを適用
delete  ary.sum;                        // メソッドを削除
```

10.1.7 配列の拡張

特定の機能をもった配列を構成する場合には、Array クラスを拡張する。
Array クラスを拡張する場合、実体としての配列要素は super もしくは Array メンバーとしてアクセスされ、これをアクセスするには、一旦変数にインスタンスの super あるいは Array を代入して、この変数を介して操作および参照しなければならない。
例：

```
class   ExArray extends Array{
        super(@arguments);                      // 要素を構築

        method  getValue():Array   return super;

        method  total():int{
                var     result:int = 0;
                foreach(item in Array)  result += item;
                return  result;
                }
        }

var     x:ExArray = new ExArray(1,2,3);         // 構築
var     y:Array = x.getValue();                 // y は配列要素へのアクセッサとなる
y += 100;                                       // 要素の追加
writeln("x.total()=", x.total());               // メソッド実行
```

10.1.8 インスタンス配列

コンストラクタ引数を持たないクラスもしくはレコードのインスタンスを要素とする配列の構築と、各々のインスタンスの初期化を一括しておこなうことができる。

書式：　new ＜クラスまたはレコード＞[] { ＜初期化指定子＞ [, ＜初期化指定子＞] }

＜初期化指定子＞::= { ＜フィールド初期値＞ [, ＜フィールド初期値＞] }

例：

```
record  Person{
        field   name:string;
        field   age:int;
        }
var     persons:Person[] = new Person[]{
        {"花子",18} , {"太郎",21} , {"次郎",17}
        };
```

上記は以下の記述と同じ結果となる。

```
var     persons:Person[] = {
        new Person{name = "花子"; age = 18},
        new Person{name = "太郎"; age = 21},
        new Person{name = "次郎"; age = 17}
        };
```

上記例のように配列宣言において new Person[] のようにコンストラクタ引数をもたないクラスあるいはレコードを指定した場合、その要素指定において new Person と記述する代わりに {"太郎",21} のように配列を指定すると、要素として Person インスタンスを自動生成し、記述された配列の要素値を順に Person レコード定義におけるフィールド定義の順に対応して"太郎"を name として、21 を age として初期化割付けしたインスタンス要素配列を構築する。

初期化できるフィールドは、値フィールド（プリミティブ値もしくはオブジェクトへの参照値をもつフィールド）に限られ、オブジェクト・フィールド、プロクシ・フィールド、メソッドなどフィールド自体がオブジェクトであるものは対象とはならない。
例えば上記のレコード定義を

```
reord Person{
        field   name:string;
        method  getName()       return  name;
        field   age:int;
        method  getAge()        return  age;
        };
```

としても同じ結果となる。

10.2 ハッシュ

※Ver 3.0 よりハッシュはオブジェクト項として統合された。

ハッシュとは、配列要素を指定する場合に要素番号ではなく識別子としての文字列を指定してアクセスする仕組みをいう。
例：
hash["name"];   //hash 配列の、フィールド名"name"で示す要素

10.2.1 ハッシュ定義

書式：　{<フィールド定義>[, <フィールド定義>]}
　　　　{<フィールド定義>[; <フィールド定義>]}

<フィールド定義>≡　　<フィールド識別名> : <式>　または
　　　　　　　　　　　<フィールド識別名> = <式>

例：

```
var     hash={ abc:10 , xyz:{ x1:"abc" , x2:"xyz" }};
```

複数のフィールド定義をおこなう場合、<,>（コンマ）または<;>（セミコロン）で区切る。
従って上記は以下のようにも記述できる。

```
var     hash = {
        abc = 10;
        xyz = {
                x1 = "abc";
                x2 = "xyz";
                }
        };
```

フィールド識別名は、識別名規約に従う文字列定数でもよい。（JSON 互換）
例：

```
var     hash={ "abc":10 , "xyz":{ "x1":"abc" , "x2":"xyz" }};
```

式として関数を記述する場合は function(引数){ステートメント・ブロック} とする。
例

```
var     hash={ boo:100, foo:function(arg){return arg+boo} };
writeln( hash.foo(10) );          // 110 を表示する
```

10.2.2 ハッシュフィールドアクセス

ハッシュのフィールドは、要素としてフィールド識別名を文字列で指定する方法と、オブジェクトのフィールド指定子＜.＞とフィールド識別名で指定する方法でアクセスする。
例：

```
var     hash={ abc:10, xyz:{ x1:"abc", x2:"ABCD" } };
var     x=hash["abc"];                    // 要素としてアクセス
var     y=hash.abc;                       // フィールドとしてアクセス
```

上記例では hash[“abc”]、hash.abc は同じフィールドを示し、よって x、y 共に 10 となる。
また、hash.xyz.x1 と hash["xyz"].x1 および hash["xyz"]["x1"]さらに hash["xyz.x1"] は共に同じ要素を指し、その値は "abc" である。
配列記述とフィールド指定子記述では、該当するフィールドが存在しない場合の扱いが異なる。
上記例で、hash["none"]は存在しない"none"というフィールド値をアクセスするため値は null となるが、hash.none としてアクセスするとワーニングレベルが 1 以上の場合にはエラーとして扱う。
配列記述書式での要素名は識別名として規定されたものでなくてはならず、hash["123"]のような指定はできない。
また、式が値としてフィールド識別名を示す文字列であるならば hash[＜式＞]として記述できる。
例：

```
var     fieldname:string = "abc";
hash[ fieldname ];               // hash["abc"]に同じ
```

10.2.3 予約フィールド

ハッシュのフィールドとして、以下のフィールド名は使用できない。

予約フィールド名	解説
length	ハッシュと配列が混在する場合に配列要素数を取り出すための組み込みフィールドもしくはメソッドとして予約されている。
getMember	ハッシュフィールド要素名を取り出すための組み込みメソッドとして予約されている。

10.2.4 フィールド要素名の取り出し

ハッシュには、組込みメソッドとして getMember()が用意されている。
getMember()は、ハッシュの要素識別名をその定義順に文字列の１次元配列として返す。
例：

```
var     foo = { arg1:100 , arg2:”ABC” };
var     member[]:string = foo.getMember();
// member は {"arg1","arg2"} という値を持つ配列への参照値となり、その要素値を利用して

foreach(item in member) writeln(item,":",foo[item]);
// 上記は foo の要素識別名とその値を":"で結合して表示する。
```

10.2.5 インスタンス化

ハッシュは Class.assign ()メソッドあるいは Record.assign()メソッドを適用してインスタンスとして振る舞うことができる。
第１引数にハッシュを、第２引数にクラスもしくはレコードを指定する。
例：

```
class   Child{
        field   name = "None";
        field   age;
        field   sex = "MALE";
        method  view()  writeln("I'm "+name+", "+age+" years old.");
        }

var     foo = {name:"Jon" , age:18};
Class.assign(foo,Child);
foo.view();
writeln(foo.sex);
```

上記は、foo が参照するハッシュが Child のインスタンスとなり、かつそのフィールド値はハッシュのもつ同名フィールドの値のままとなる。またクラスインスタンスのフィールドのうち、ハッシュ内にないフィールドはハッシュ内に追加される。
結果は″I’m Jon, 18 years old.″と、"MALE"と表示する。

10.3 ハッシュと配列の混在

ハッシュ内には配列要素を含むことができる。
例：

```
var     hash = {a1:100,a2:200,300,400};
var     hash = {a1:100,300,a2:200,400};
```

上記は論理的に同じオブジェクトを構成する。すなわちフィールド識別子のない要素は配列と同じ扱いとなり、そのアクセスは配列添え字式でおこなえる。
上記例では hash[0]で 300 を、hash[1]で 400 をアクセス対象とする。
ハッシュに配列要素を含むかどうかについては、hash.length 値もしくは hash.length()値で判断する。すなわち、これらの値が 0 であれば配列要素を含んでいない。

※注意：ハッシュと配列の混在はドキュメント性を損なうため極力使用しないほうが望ましい。

10.4 配列・ハッシュの消滅

配列宣言およびハッシュ宣言によって生成した配列は、どの参照変数からも参照されなくなった時点で消滅する。
例：

```
var     a={1,2,3};      ①配列を生成し、変数 a が参照する。
var     b=a;            ②変数 a が参照する配列を、変数 b も参照する。
        a=10;           ③配列は、変数 b からのみ参照されている
        b=a;            ④配列はどこからも参照されなくなり、ここで消滅する。
```

同様に、ブロック内で宣言された配列およびハッシュも、プログラム実行がブロックを脱する時点で消滅する。したがって、ループを繰り返すブロック内で配列およびハッシュを宣言した場合、ループ毎に生成と消滅が発生し実行速度低下の要因となる。
配列をループブロック内でプログラム定数として使用する場合は、ループブロックの外部で宣言して使用するか、for 文の初期化式で宣言することを奨める。

例：

```
for(var day=0; day<7; day++){
        writeln({"Sun","Mon","Tue","Wed","Thu","Fri","Sat"}[day]);
        }
```

上記は配列を７回生成破棄するが、以下は１回のみである。

```
const   wknm={"Sun","Mon","Tue","Wed","Thu","Fri","Sat"};
for(var day=0; day<7; day++){
        writeln(wknm[day]);
        }
```

10.5 配列・ハッシュの複製

配列またはハッシュは、Object.duplicate()または Array.duplicate()メソッドによって複製される。

例：

```
var     ary1 = {1,2,3,abc:100};
var     ary2 = Array.duplicate(ary1);
```

配列に限っては、pop()メソッドを使って複製ができる。

例：

```
var     ary1 = {1,2,3,abc:100};
var     ary2 = ary1.pop(,,true);
```

11 オブジェクト

オブジェクトには、クラスとレコードをもとに生成するインスタンス、およびオブジェクト項として動的に生成されるものと、オブジェクト宣言によって定義される静的オブジェクトがある。
また、クラス、ハッシュ、配列、メソッドならびに関数も広義のオブジェクトとして扱う。

11.1 オブジェクトの種類

11.1.1 インスタンス

クラスに対して new 演算子を適用するか、クラスを指定して既定の new()メソッドを呼び出す、あるいはクラスをメソッドのように呼び出すことによって、そのクラスのインスタンスを得る。

書式１：new ≪クラス参照≫(コンストラクタ引数並び)
書式２：≪クラス参照≫.new(コンストラクタ引数並び)
書式３：≪クラス参照≫(コンストラクタ引数並び)

≪クラス参照≫とは、式（クラス参照式）の値が特定のクラスを指すものである。
例：

```
class Car(setname){
        field name=setname;
        }
var     car1 = new Car("atlas");        ①
var     car2 = Car.new("atlas");        ②
var     car3 = Car("atlas");            ③
```

① はクラスに new 演算子を適用するスタイル
② はクラスの、みなしクラスメソッド new()の呼出しによるスタイル
③ はクラスを、関数のように呼び出すスタイル

例：

```
class Car(setname){
        field name=setname;
        }
var     carcls:class = Car;                     // Car クラスへの参照を代入
var     car1 = new carcls("atlas");
var     car2 = carcls.new("atlas");
var     car3 = carcls("atlas");
```

変数 carcls は Car クラスへの参照を保有するため、そのクラスを基にオブジェクトを生成する。

■ 実引数を持たないインスタンス生成
クラスがコンストラクタ引数を持たない場合、ならびに引数に既定値をもつ場合は、書式１に限り new クラス参照() 、または単に new クラス参照 としてインスタンスを生成できる。
例：

```
class ToDay{                                    //引数なし
        field today:Date = new Date();
        }

class DefaultDay(today:Date = new Date()){}     //既定値をもつ引数

var     date1 = new ToDay;
var     date2 = new ToDay();
var     date3 = new DefaultDay;
```

```
var     date4 = new DefaultDay();
```

ToDay クラスはコンストラクタ引数を持たない。DefaultDay クラスはコンストラクタ実引数を指定してもしなくてもよく、指定しない場合は new Date()が既定値となる。

■ コンストラクタ・バリエーション

インスタンス生成時に、目的に応じて引数の型を変えたり引数の個数を変えたい場合は以下の方法を用いると良い。

- クラス定義時にコンストラクタ引数は宣言しない。
- 実引数の参照は既定の arguments 配列を使用する。

例：

```
class   分数{
        field   分子:int, 分母:int;
        // ここからコンストラクタ
        select(typeof(arguments[0])){
        case    Type.null:      分子 = 分母 = 1;        // new 分数()
        case    Type.int:{      // 第１引数が整数       // new 分数(int[,int])
                分子 = arguments[0];
                分母 = arguments[1] isnull 1;
                }
        case    Type.object:{   // 第１引数がオブジェクト // new 分数(分数)
                if(arguments[0] instanceof 分数){
                        分子 = arguments[0].分子;
                        分母 = arguments[0].分母;
                        }
                else    throw   "不正な引数";
                }
        default:                throw "不正な引数";
        }
        // ここまでコンストラクタ
}

var     boo:分数 = new 分数(2,3);
var     foo:分数 = new 分数(boo);
```

上記例では、第１引数の型によって依頼元の意図を判断し、通常のインスタンス生成なのかあるいはコピー・コンストラクタ呼出しなのかを判断している。なお、arguments 配列はコンストラクタ中でしか参照できない。

※ コピー・コンストラクタを実現するだけなら、インスタンスの複製をおこなう duplicate()メソッドを利用するか、あるいはオーバライドした duplicate()メソッドを実装することを奨める。

11.1.2 オブジェクト項

object キーワードに続いてオブジェクトの実体を定義することによって、定義場所に、複合値をもつ単独のオブジェクトを生成する。（詳細は「object 項」を参照）

書式１：object ［型名］：｛フィールド定義｜メソッド定義｝
書式２：object ：［型名］ ｛フィールド定義｜メソッド定義｝
書式３：object ［型名］ ｛ NDSN 記述 ｝

11.1.3 実行可能オブジェクト

静的な method および function は、その定義場所に実行可能オブジェクトを生成する。
クラスおよびレコードのインスタンス・メンバーとして定義される場合は、クラスおよびレコードのインスタンスが生成される際に、その内部にメンバー要素として生成される。

書式１：method 名前 （引数並び） { ブロック記述 }
引数並びで示す引数をその順に与えられ、ブロック記述のステートメントを実行し、そのうちの return 文で返される値を呼び出し元に返す、という<名前>で参照される【名前つき method オブジェクト】が生成される。
例：

```
method add(x:variant,y:variant):variant{
        return  x + y;
        }

writeln( add(100,200) );              // add を直接参照して実行
var     foo:method = add;             // 変数 foo に add への参照値を代入
writeln( foo("本日は","晴天なり") );   // foo が参照する method オブジェクトを実行
```

書式２：method （引数並び） { ブロック記述 }
機能は【名前つき method オブジェクト】と同じだが、名前を指定して参照することができない【無名 method オブジェクト】が生成される。
したがって、宣言時に参照変数への代入、もしくは他の method 呼び出しへの実引数として指定する必要がある。
例：

```
var     foo:method = method (x:variant,y:variant):variant{
        return  x + y;
        }

writeln( foo("本日は","晴天なり") );   // foo が参照する method オブジェクトを実行

// 引数に method オブジェクトを渡す仕様の method
method  executor(exec:method, arg1:variant, arg2:variant):variant{
        return  exec(arg1,arg2);
        }
// メソッド executor に無名 method と 100、200 を渡して結果を result に格納する
var     result = executor(method (x:variant,y:variant):bool{
        return  x > y;
        },
        100,200);
```

11.1.4 静的オブジェクト

独立した単一オブジェクトを定義位置に構築する。
定義されたオブジェクトは内部にメソッドを保有でき、メソッド呼び出しに対応する。

書式１：object 名前 ： { プログラム要素 }
書式２：const 名前 ： object{ プログラム要素 }
書式３：名前 ： { NDSN 記述 }

名前で示されたオブジェクトの実体を、その定義位置（スコープ）に生成するとともに、定義内にステートメントがあれば、これを実行する。

書式１の例：

```
object 分数:{
        field   分子:int = 1;
        field   分母:int = 3;
        method  toReal():real{
                return  分子/(分母*1.0);
                }
        };

var     obj = 分数;               // obj は分数を参照する変数
writeln(obj.toReal());          // 0.333333 と表示する
分数.分母 = 5;
writeln(obj.toReal());          // 0.2 と表示する
writeln(分数.toReal());          // 同じく 0.2 と表示する
```

書式２の例：

```
const   person: object{
        field name = "弥勒";
        field age = 140000000;
        method  view(){
                writeln("name=",name," age=",age);
                }
        };

person.view();                  // name=弥勒 age=140000000 と表示
person.name = "田中一郎";
person.age = 23;
person.view();                  // name=田中一郎 age=23 と表示
```

書式３の例：

```
発注伝票: {
        商品: {
                {商品名:"品名１", 個数:20, 単価:1000, 金額:method{return 単価*個数}},
                {商品名:"品名２", 個数:10, 単価:8000, 金額:method{return 単価*個数}},
                {商品名:"品名３", 個数:2,  単価:4000, 金額:method{return 単価*個数}}
                },
        小計: method {
                var 計 :int = 0;
                eachof (商品) 計 += 金額;
                return 計;
                },
        消費税: method{return 小計 * 0.05},
        合計:   method{return 小計 + 消費税}
        };

writeln(発注伝票.合計);          // 108000 と表示する
発注伝票.商品[2].個数 = 20;      // データを書き換え
writeln(発注伝票.合計);          // 合計は正しく計算され、180000 と表示する
```

11.2 フィールド定義

書式：field　名前［:型］［ = 初期化式］｛,名前［:型］［ = 初期化式］｝;

名前で識別するメンバーとして、値を保持する。
型を指定する場合は、初期化式を含め代入する値は型が同じでなければならない。
型を指定しない場合は、代入する値に応じて動的に型が規定される。
初期化式を指定しない場合の初期値は null である。

11.2.1 メソッド

書式：field　名前 : method（仮引数並び）［:返値型］{ ステートメント並び }

method　名前（仮引数並び）［:返値型］{ ステートメント並び }　の別書式。
例：

```
class   Simple{
        field   value:int;
        method  action1(adder:int):int{
                return  adder+value;
                }
        field   action2:method(adder:int):int{
                return  adder+value;
                }
        }
```

値フィールドにメソッドへの参照を代入することによっても、そのフィールドはメソッドと同様に機能する。
例：

```
class   Simple{
        field   value:int = 100;
        method  action1(adder:int):int{
                return  adder+value;
                }
        field   action2:method = action1;              // action1()を参照
        field   action3:method = method(adder:int):int{ // method 項を参照
                return  adder*value;
                }
        }

var     obj:Simple = new Simple();
writeln(obj.action2(100));                  // 200 を表示する
writeln(obj.action3(100));                  // 10000 を表示する
obj.action3 = obj.action2;                  // 参照先を変更
writeln(obj.action3(100));                  // 200 を表示する
```

この場合はメソッド参照フィールドといい、初期化式もしくは代入操作によってメソッドへの参照が動的におこなわれる。（詳細については「method 項」についての解説を参照のこと）

11.2.2 メソッド・フィールド

書式：　field　名前 : method{ ステートメント並び・return 値 };

フィールドが参照（アクセス）されるごとに、そのタイミングで method {...}で示すブロックを

実行し、その return 値を値とする。
例：

```
object  注文:{
        field   商品:object = {
                {商品名:"商品１", 単価:100, 数量:30},
                {商品名:"商品２", 単価:150, 数量:10},
                {商品名:"商品３", 単価:200, 数量:20}
                };

        field   合計: method{
                var     計:int = 0;
                foreach(item in 商品)  計 += item.単価 * item.数量;
                return  計;
                }
        };

writeln(注文.合計);              // 8500 を表示する
```

データを記述する場合にその扱い方とともに記述する例である。

11.2.3 プロクシ・フィールド

書式：　field　名前　：　proxy　{　<get アクセッサ>・<set アクセッサ>　};

プロクシ・フィールドはその内部に、外部に値を提供するための<get アクセッサ>、外部からの値を取り込むための<set アクセッサ>をもつ。

<get アクセッサ>≡　プロクシから値を読み出す際に呼出され
public method getValue()[:型] { ... return 値; }
として実装されなければならない。このメソッドは引数を持たず、呼び出しに際しては値を返す。

<set アクセッサ>≡　プロクシに値を書き込む際に呼出され
public method setValue(value[:型]) { ... }
として実装されなければならない。このメソッドはただ１つの引数をもち、呼び出し時にこの引数に与えられた値を内部に保持するように実装する。

構成書式：

```
field   名前 : proxy {
        private field inner_value:型 = 初期値;
        public method getValue():型     {return inner_value;}
        public method setValue(value:型) {inner_value = value;}
        }
```

public method getValue(){...}は#get{...}に、public method setValue(value){...}は #set{...}のように省略して記述できる。
なお、#set{}においては、実引数に与えられる値を arguments[0]で取り出すことになる。
更に#get{}および#set{}が各々１ステートメントで構成される場合は{}で括る必要もない。
また　名前:proxy{...} は 名前{...}　の書式でも良い。
これに従えば上記は以下のように簡潔に記述できる。

```
field   名前 {
        private field inner_value:型 = 初期値;
        #get    return inner_value;
        #set    inner_value = arguments[0];
```

```
        }
```

構成および使用例：

```
field   boo : proxy {
        private field value:int = 0;     // プロクシの初期値
        #get    return  value;
        #set    value = arguments[0];
        };
boo = 100;                   // ① プロクシ内の value に 100 が格納される
var foo = boo;               // ② プロクシ内の value 値 (==100) が foo に代入される
```

①がプロクシへの書き込みとなり、右辺値を setValue()メソッドの引数に渡して呼出す。
②はプロクシからの読み出しとなり、getValue()メソッドを呼出しその返値が右辺値となる。

プロクシ・フィールドは外部に公開するアクセス対象としては静的なフィールドと見えるが、実際は動的なアクセッサ・メソッドによって構成されるため、下記の例に示すように他のフィールド値の変化に依存するような値を、動的に算出するような場合に有用である。
例：

```
class   商品(商品名:string,仕入価格:int){
        static  private field   利益率 : real = 0.20;
        static  private field   税率 : real = 0.05;

        public  field   売価 : proxy {  // 正式書式
                public method   getValue():int{
                        return   (仕入価格*(1+利益率)).toIntValue();
                        }
                };

        public  field   消費税額 {      // 省略書式
                #get    return  (売価 * 税率).toIntValue();
                };
        }
```

商品の売価とか消費税額は商品の提供する「機能」ではなく、商品の持っている属性値であると考えるのが自然であり、これをメソッドとして提供するのは不自然である。
この例ではインスタンス生成時に値が決定できるので、初期化式として計算した値を値フィールドとして保有することでも良さそうである。
例：　public field 売価:int = (仕入価格*(1+利益率)).toIntValue();
しかし、値で保持した場合は外部アクセスによる書換えの可能性もあるしまた、インスタンス生成後に利益率とか税率を変更すると、その変更を受けて再計算が必要となり、そのためにメソッド呼出しにしなければならない。それでは利用するプログラムのすべてを、値フィールドのアクセスからメソッド呼び出しに変更しなければならず、良い方法とはいえない。
値フィールドをそのままプロクシ・フィールド化することにより、同じインターフェースを保持したまま、実体は処理の結果として提供することができる。
また setValue()メソッドを持たないようにすれば不用意な書換えからも保護できる。

プロクシ・フィールド内には、値を保持するフィールドや、アクセッサが呼び出すローカル・メソッドを定義でき、さらに初期化をおこなうなど定義時に実行するステートメント（イニシャライザ）を記述できる。
イニシャライザはプロクシ・フィールドの定義時点、すなわち一般フィールドの初期化式と同じタイミングで実行される。また定義順序さえ満たせば、イニシャライザからアクセッサを呼び出すこともできる。
例：

```
class   絶対値{
        public field absInt : proxy {
                private field value:number;                     // 値を保持
                private method toAbs(val:number):number{        // メソッド
                        return  Math.abs(val);
                        }
                // アクセッサ
                public method   getValue():number       return value;
                public method   setValue(val:number)    value = toAbs(val);

                // イニシャライザ
                value = 0;
                }; // end of proxy
        }

var     absobj = new 絶対値();
absobj.absInt = -100;
writeln(absobj.absInt);                                 // 100 と表示
absobj.absInt -= 150;
writeln(absobj.absInt);                                 // 50 と表示
```

プロクシ・フィールドがその値としてメソッドへの参照値を持つ場合、プロクシ・フィールドに対して呼出しをおこなって直接実行することはできない。
一旦変数に取り出してから実行するか、( )で括って値として評価したものを呼び出す必要がある。
例：

```
method  boo()   writeln("Hello");

field   foo: proxy{
        method  getValue():method       return  boo;    // boo()の参照値
        };

var woo = foo; woo();           // OK "Hello"を表示
(foo)();                        // OK "Hello"を表示
foo();                          // エラーとなる（foo 自身はメソッドではないため）
```

11.2.4 Runtime 式・フィールド

フィールドを読出しアクセスするごとに、記述されている式を評価しその結果を値とする。

書式１：`field　名前 :: 式;`
書式２：`field　名前 : expat{ 式 };`

例：

```
class   boo{
        field   info :: name+"は"+age+"歳です";
        field   name:string;
        field   age:int;
        }

var foo:boo = new boo();
foo.name = "太郎";
```

```
foo.age = 10;
writeln(foo.info);              // 太郎は 10 歳です と表示
foo.age = 20;
writeln(foo.info);              // 太郎は 20 歳です と表示
```

式に含まれるフィールド名等の識別子は、このフィールドがアクセスされた時点において名前解決されていればよい。またこのフィールドへの代入は無視される。
機能としては、メソッド・フィールド　　field 名前 : method { return 式;}
あるいは、プロクシ・フィールド　　field 名前 : proxy { #get return 式;}
と同じである。

11.2.5 オブジェクト・フィールド

内部に複数の構成要素を持つフィールドを構成する。
クラス・フィールドとして宣言した場合はクラス定義時に構築され、インスタンス・フィールドとして宣言した場合はインスタンス生成時に構築される。

書式： field 名前 : object{ フィールド定義｜メソッド定義};

例：

```
class   Person(性別:bool){
        static field 人数: object{
                field   男性:int = 0;
                field   女性:int = 0;
                method  view():string{
                        return  "男=%d 人、女=%d 人".import(男性,女性);
                        }
                };

        field   氏名:string = "";
        field   属性: object{
                field   年齢:int;
                field   職業:string;
                method  view( ):string{
                        return  {氏名,年齢,(性別 ? "男":"女"),職業}.join(":");
                        }
                };
        method  setValue(name:string,age:int,occupation:string){
                氏名 = name;
                属性.年齢 = age;
                属性.職業 = occupation;
                }

        if(性別)        人数.男性 += 1;
        else            人数.女性 += 1;
        }

var     person = new Person(false);             // 女性を生成
person.setValue("花子",19,"学生");
writeln(person.属性.view());                    // 花子:19:女:学生 と表示
writeln(Person.人数.view());                    // 男=0 人,女=1 人 を表示
```

オブジェクト・フィールドはその内部に定義されるメンバー全体を示す名前空間であり、オブジェ

クト・フィールドに対して直接代入操作をおこなうことはできない。
例：

```
person.属性.年齢 = 32;          ⇒  OK
person.属性 = "無職";            ⇒  NG（オブジェクト・フィールドには代入できない）
```

11.3 オブジェクトの参照と操作

11.3.1 オブジェクトのアクセス

オブジェクトは変数および引数への代入操作によって参照される。
オブジェクトを参照する変数に型を指定する場合、その型名を「object」とすれば、どのようなオブジェクトでも参照可能である。
例：

```
var     obj : object;
obj = new MyClass();
obj = {1,2,3};
```

インスタンスはクラス名を型名とし、静的オブジェクトとオブジェクト項は宣言時のオブジェクト名を型とする。
例：

```
class   Car{ }
var     mycar : Car = new Car();
var     mycar1: Car = mycar;

object  StaticObj:{     }
var     obj : StaticObj = StaticObj;
var     obj1:StaticObj = obj;

var     term :TermObj = object TermObj{ };
var     term1:TermObj = term;
```

11.3.2 オブジェクト参照子

キーワード this と、キーワード“．”（ピリオド）、キーワード super は各々単独で、インスタンス・メンバー指定において基準となるオブジェクトを既定する。
this 参照子は実体としてのオブジェクトすなわちエンティティ・オブジェクトを意味し、これを this 項という。
ピリオド参照子は常に、そのステートメントを記述してあるクラスのインスタンスすなわちカレント・オブジェクトを意味し、これを current 項という。
super 参照子は常にカレント･オブジェクトの基本インスタンスを意味し、これを super 項という。

以下にエンティティ・オブジェクトとカレント・オブジェクトの違いを示す。
例：

```
class Car{
        field   name;
        method  getName()       return  this.name;      // this 項
        method  getBaseName()   return  .name;          // current 項
        }

class SuperCar extends Car{
        super();
        field   name;                                   // override したフィールド
        method  getBase()       return  super;          // super 項
        }

var     base:Car = new Car();
var     mycar:SuperCar = new SuperCar();
```

変数 mycar が示すオブジェクトは SuperCar インスタンスであり、これをエンティティ・オブジェクトという。ところがこのオブジェクトは内部にサブ・オブジェクトとして Car を含んでおり、Car 部分について考える際には「mycar の Car 要素」といい、これを mycar.Car あるいは mycar.super といった表現で記述する。すなわち「mycar に関して」という部分をピリオドで区切るため、この段階で mycar がカレント・オブジェクトとなる。
さらに「mycar の Car 要素にある name 値は」といえば mycar.Car.name という表現となり、この段階でのカレント・オブジェクトは Car となる。
ここで、Car クラスで定義されている getBanseName()メソッドと getName()メソッドの構成ステートメントを考えてみる。
getBaseName()では .name として current 項の name メンバーを返すが、この位置でのカレント・オブジェクトは常に Car であり、したがってその返す対象は Car 内の name である。
一方、getName()メソッドでは this.name として this 項の name メンバーを返すが、this はエンティティ・オブジェクトを意味するため、new Car()とした場合と、new SuperCar()とした場合では実体としてのオブジェクトが異なる。
したがって、base.getName()では当然ながら Car 内の name を返すが、mycar.Car.getName()では SuperCar 内の name を返す。

11.3.3 オブジェクト直接項

値として直接オブジェクトを指定する場合、これをオブジェクト直接項という。
オブジェクト直接項はそれ自体が変数などに参照されない限り、その記述されたステートメント内でしか存在しない。すなわち、次のステートメント実行時にはすでに消滅している。
例：

```
writeln(new Date().toString());                              ①
writeln({a:10,b:20,add:method(){return a+b;}}.add());        ②
writeln({"SUN","MON","TUE","WED","THU","FRI","SAT"}[3]);     ③
```

① は、現在時刻をあらわす Date インスタンスを生成し、そのメソッド toString()を呼び出した結果を出力する。インスタンスは toString()メソッド呼出しから戻った時点で消滅し、writeln()に引数として渡されるのは toString()の結果として返される文字列である。
② は、{a:10,b:20,add:method(){return a+b;}} というオブジェクトを生成し、そのメソッド add()を実行した結果を出力する。オブジェクトは add()メソッド呼出しから戻った時点で消滅し、writeln()に引数として渡されるのは add()の結果として返される値である。
③ は、曜日名を意味する文字列要素をもつ配列オブジェクトからその要素を取り出して出力する。配列は要素を指定する添え字式（[3]）の実行直後に消滅し、writeln()に引数として渡されるのは要素値である文字列（"WED"）である。

11.3.4 オブジェクト参照項

変数にオブジェクトへの参照を割り当ててこの変数を式中に記述する場合、もしくはメソッドに実引数として渡されたオブジェクトをメソッド内で仮引数を介して参照する場合、各々をオブジェクト参照項という。オブジェクトは参照変数などに割付けられて参照が継続している間は消滅することなく存在する。

例：

```
var     todayNow = new Date();
writeln(todayNow.toGMTString(),todayNow.toLocalString());
```

変数 todayNow は Date インスタンスを参照するオブジェクト参照変数であり、これが式中に記述された場合は、変数が参照するインスタンスを指す。

```
var     weekname= {"SUN","MON","TUE","WED","THU","FRI","SAT"};
for(var cnt=0; cnt < weekname.length; cnt++) writeln(weekname[cnt]);
```

変数 weekname は文字列を要素とする配列を参照するオブジェクト参照変数。

```
var     obj = {a:10 , b:20 , add:method(){return a+b;}};
writeln(obj.add());
```

変数 obj はハッシュを参照するオブジェクト参照変数であり、obj.add()はそのハッシュ内フィールド add（メソッドとして実装されている）を実行呼び出しする。

11.3.5 メンバーアクセス

オブジェクトは値を保持するフィールドと、機能を提供するメソッドを持ち、これらをオブジェクトのメンバーという。
オブジェクトのメンバーをアクセスするには、以下の書式を使用する。

書式１：　　オブジェクト参照.メンバー名
書式２：　　オブジェクト参照["メンバー名"]

オブジェクトはフィールドとメソッドを持っており、フィールドの内容とメソッドの扱う対象はオブジェクトごとに異なる。
クラス・インスタンスのフィールドはコンストラクタによって初期化されその後の操作によって値が変化していくが、同じクラスから生成した他のインスタンスとは無関係である。
また、インスタンス・メソッド内でアクセスするメンバーは同じインスタンス内のメンバーもしくはクラス・メンバーに限られ、同じクラスから生成された他のインスタンス内のメンバーをアクセスすることはできない。

変数にオブジェクトが割り付けられていれば、変数が示すオブジェクトのメンバー名を指定してオブジェクトメンバーを参照、または操作できる。
また、オブジェクトがインスタンスの場合は、そのクラスメンバーもアクセスできる。
ただし、アクセス制御（private、protected）の規定に従う。

```
class Car(setname){
        field name=setname;
        }
class Supercar(setname,setspeed):Car{
        super(setname);
        field maxspeed = setspeed;
        }
var     mycar=new Supercar("atlas",260);
writeln(mycar.maxspeed);              書式１による記述
writeln(mycar["maxspeed"]);           書式２による記述
```

書式２によって多階層にまたがるメンバーをアクセスする場合、各々のメンバー名を「.」（ピリオド）で結合する。
例：

```
foreach(element in {"super.name","maxspeed"}){
        writeln(mycar[element]);        // element には順に"super.name"、"maxspeed"が入る
        }
```

11.3.6 直接メソッド呼び出し

書式１：オブジェクト参照.メソッド名(引数並び)
書式２：オブジェクト参照[メソッド名文字列値](引数並び)

変数が参照するオブジェクトのメソッド名と引数を指定してメソッドを実行する。
また、同一クラス内のメソッドについてはメソッド名と引数を指定して実行する。

例：

```
class Car(setname:string){
        field   name:string = setname;
        method  getname():string        return  name;
        method  view()                  writeln(getname());
        }
class Supercar(setname:string, setspeed:int):Car{
        field   maxspeed:int = setspeed;
        super(setname);
        method  speedset(speed:int)     maxspeed = speed;
        }

var     mycar:Supercar = new Supercar("atlas",260);
writeln(mycar.getname());
mycar.speedset(300);
あるいは
writeln(mycar["getname"]());    // writeln(mycar["super.getname"]()); でもよい
mycar["speedset"](300);
```

11.3.7 間接メソッド呼び出し

メソッド参照値を変数もしくはメソッドの引数に格納して、間接的にこれを呼び出すこともできる。メソッド m1(){...}が定義されている場合、m1()とすればメソッドの呼出し（実行）となるが、単に m1 とすればメソッドへの参照値が得られる。一般には変数にメソッドの参照値を代入したり、またコールバックなどの目的で参照値を引数として渡すなどが考えられる。変数や引数にメソッドの参照値を格納する場合、その型を指定するなら「method」とする。

例：

```
var     methodref:method = m1;          method m1(){}の参照値を得る
```

変数もしくは引数にインスタンス・メソッドへの参照を代入した場合、メソッド呼出しによる実行環境はそのメソッドの属すインスタンス環境となる。

例：

```
class Car(setname:string){
        field   name:string = setname;
        method  getname():string        return  name;
        }
class Supercar(setname:string, setspeed:int):Car{
        field   maxspeed:int = setspeed;
        super(setname);
        }

var     mycar:Supercar = new Supercar("atlas",260);
var     newcar:Car = new Car("polestar");

var     mycarmethod:method = mycar.getname;
var     newcarmethod:method = newcar.getname;
writeln(mycarmethod());         ⇒      "atlas" を表示
writeln(newcarmethod());        ⇒      "polestar" を表示
newcar = null;                          参照解除により Car インスタンス消滅
writeln(newcarmethod());        ⇒      "null" を表示
```

※ インスタンスが消滅した場合は、メソッド参照変数の内容は null となる。

11.3.8 メソッド・インターセプト

複数のメソッドに対して共通したインターセプト手続きを定義し、そのメソッドが呼び出された際にこれらに横断的な処理をまとめておこなうことができる。例えばメソッド実行に先立ち、もしくは実行後にメソッド名や引数のリストをログとして記録したい、あるいはメソッドの実行時間を測定したい場合などがある。
この場合、メソッドごとに処理を追加するのではなくインターセプターという共通の手続きを定義し、これに対象とするメソッドを登録して、そのメソッドが呼び出された際に個々のメソッドに固有な情報を統一的に扱うことができる。（アスペクト指向というらしい・・・）

書式： intercept 名前（メソッド名並び）＜ブロック＞

＜名前＞はクラス内でユニークでなければならない。
＜メソッド名並び＞に、インターセプトするメソッド名を、複数あれば「,」で区切って指定する。
例：
以下のメソッド

```
method  m1(){ ... }
method  m2(){ ... }
```

がクラス内に定義されているものとして、

```
intercept Logger(m1,m2){
        writeln("Start:", target.name());                ①
        excecute();
        writeln("End:", target.name());                  ②
        }
```

メソッド m1 あるいは m2 が呼び出された時点で、ここに示した Logger インターセプターに制御が移る。インターセプター内で execute()を実行した時にインターセプトしたメソッドを実行する。上記例ではメソッド実行前に①が実行され、メソッド実行後に②を実行する。

インターセプター内では既定された以下のフィールドおよびメソッドを使用する。

識別子	機能および内容
field target:method	インターセプトしたメソッド
field caller:method	インターセプトしたメソッドの呼び出し元
field arguments[0..*]:variant	インターセプトしたメソッドへの実引数配列
method execute():variant	インターセプトしたメソッドを実行し、その結果を返す
method deintercept():void	インターセプトしたメソッドのインターセプトを解除する

target は常にインターセプトしたメソッドを指し、caller はインターセプトしたメソッド（target）を呼び出したメソッド（コンストラクタ内から呼び出した場合はインスタンス）を指す。
arguments 配列には target メソッドに渡された実引数のコピーが引数の指定順に格納されている。
execute()を実行することによって、target メソッドが実行され、そのリターン値が execute()のリターン値として得られる。
インターセプター内での return ステートメントは execute()が返す target メソッドのリターン値に優先するが、インターセプター内で return をおこなわない場合は execute()の値すなわち target メソッドのリターン値が戻される。ただし execute()をおこなわない場合は target メソッドを実行しないし、呼出し元には null を返す。
deintercept()を実行すると target メソッドのインターセプトを解除し、次回以降 target が呼び出された際にインターセプトをおこなわない。
これらはインターセプト対象としたメソッドごとに独立している。

クラスメソッドをインターセプトするためには static intercept 名前(){ ... } と static を付けて定義しなければならない。

11.3.9 継承とオーバライド

private として宣言された基本クラスのメンバー以外は派生クラスから参照できる(継承という)。

派生クラスにおいて基本クラスのメンバーを再定義（オーバライドという）すると基本クラスのメンバーは隠蔽されるため、基本クラスのメンバーを明示的に参照する場合は、以下のように基本クラス名もしくは″super″を付加して指定する。

＜オブジェクト参照＞.＜基本クラス名＞.＜フィールド名＞
＜オブジェクト参照＞.＜基本クラス名＞.＜メソッド名＞(引数並び)
＜オブジェクト参照＞.super.＜フィールド名＞
＜オブジェクト参照＞.super.＜メソッド名＞(引数並び)

例：フィールドの継承とオーバーライド

```
class Car(setname){
        field name = setname;
        }
class Supercar(setname):Car{
        super(setname);
        field name = setname+" SPORTS";
        }
var     mycar=new Supercar("atlas");
writeln(mycar.name);                    //“atlas SPORTS”と表示する
writeln(mycar.Car.name);                //“atlas”と表示する
writeln(mycar.super.name);              //“atlas”と表示する
```

例：メソッドの継承とオーバーライド

```
class Car(argname){
        field name=argname;
        method  setname(newname)        name = newname;
        }
class Supercar(argname,setspeed):Car{
        super(argname);
        field maxspeed = setspeed;
        method  setname(option)         name += option;
        }
var     mycar=new Supercar("atlas",260);
writeln(mycar.setname("poseidon"));
writeln(mycar.Car.setname("Poseidon"));
writeln(mycar.super.setname("Poseidon"));
```

11.3.10 class および super 参照子

class および super という参照子は this あるいはピリオドで示すオブジェクト、さらにクラスを指定して以下に既定する対象を示す。

クラスに既定されているクラス・フィールドとして

フィールド名:型	記述例	解説

class:class	this.class	クラスメソッド内に記述した場合、その定義クラス[5]への参照値
	＜クラス名＞.class	＜クラス名＞で示すクラスへの参照値
super:class	this.super	基本クラスへの参照値

インスタンスに既定されているインスタンス・フィールドとして

フィールド名:型	記述例	解説
class:class	this.class	インスタンスメソッド内に記述した場合、その実クラス[6]への参照値
	.class	インスタンスメソッド内に記述した場合、その定義クラスへの参照値
	＜クラス名＞.class	＜クラス名＞で示すクラスへの参照値
super:object	this.super	基本インスタンスへの参照値

this.super は常に、実体インスタンスの直近基本インスタンスを参照するが、単に super もしくは .super とした場合はカレントインスタンス[7]に対する基本インスタンスとなる。

※ super の扱いがクラスメソッド内とインスタンスメソッド内で異なる点に注意されたい。
super.＜メンバー名＞ という指定はクラスメソッド内からは基本クラスメンバーを、インスタンスメソッド内からは基本インスタンスメンバーあるいは基本クラスメンバーを指す。

11.3.11 メンバーの動的変更

オブジェクトにフィールドおよびメソッドを追加する場合は create 文を使用する。
例：

```
class Car(argname){
        field name=argname;
        method  setname(newname)        name = newname;
        }

var     mycar = new Car("atlas");
create mycar.color=”RED”;                                            ①
create mycar.setColor=method(newcolor){color=newcolor;};             ②
```

①はオブジェクト内に存在しない“color”フィールドを新たに定義し、値“″RED″”を設定する。
②は以下のメソッドを追加する。

```
method setColor(newcolor){
        color = newcolor;
        }
```

フィールド値を変更する場合と同様、メソッドの実装を変更することができる。
上記で設定したメソッドを以下のメソッド置換によって機能を変更することができる。

```
mycar.setColor=method(newcolor){color=newcolor;return color;};
```

フィールドおよびメソッドの削除は delete 文でおこなう。
例：

```
delete  mycar.color,mycar.setColor;
```

delete 文実行後は、そのフィールドおよびメソッドはオブジェクト内に存在しない。

[5] ステートメントを定義しているクラス
[6] 派生クラスを構成するもっとも外側のクラス
[7] ステートメントが属すインスタンスをいう

※フィールドおよびメソッドの変更は該当するインスタンスに限られ、同じクラスから生成された他のインスタンスには影響を及ぼさない。

11.3.12 オプショナル演算子

インスタンスに対する２項演算子[8]を独自に定義し、メソッド呼出しよりも見通しの良いプログラムを記述できる場合がある。例えば下の例に示すように、分数に対する演算を、add()とか sub()というメソッドを定義してその引数に指定するより、「+」演算子や「-」演算子を用いて右辺値の値を左辺値に作用させるほうが自然である。
例：

```
class 分数(分子,分母);
var     valA = new 分数(1,3);          1/3
var     valB = new 分数(1,2);          1/2
var     valC = valA + valB;            1/3+1/2 を想定
```

オプショナル演算子は、以下の書式によるクラス内定義によって実現する。

書式： define operator 演算子識別 (右辺値名[：型])[：型] <ブロック>

演算子識別は「!#%&=~^¦+-*/<>」の各文字単独か、もしくはその文字を先頭に含み半角スペースあるいはタブもしくは右辺値定義部の開始文字である「(」（カッコ）が検出される範囲までとする。したがって、これらの終端文字を演算子内に含めることはできないし「{}()[]"'\$.,;:」の各文字も含んではならない。
演算子識別子には上記条件を満たせば任意の文字列を定義できる。
ただし、表記から想定される機能からかけはなれた演算子は定義すべきでない。
また、「=」演算子を定義した場合、オブジェクト参照変数への既定代入演算子がオプショナル演算子として解釈され、変数への代入ができなくなる点に注意が必要である。

定義した演算子は、言語レベルで既定している演算子よりも優先して解釈される。
似通った表記の演算子を複数定義した場合は、解釈時に最長一致演算子を優先する。

右辺値の書式は　<名前>[：<型>]　であり､型を指定した場合はワーニングレベル１以上の場合に実行時の右辺値の型を検査し、該当しない場合はアボートする。

ブロック内のステートメントはメソッドと同様、メンバーフィールド、メンバーメソッドならびにコンストラクタ引数、さらにクラスフィールドとクラスメソッドを参照できる。

リターン値の型を指定した場合、ワーニングレベルを１以上に設定した場合に限り型の整合性をチェックする。
※注意：ブロックに return を記述しないか null を返した場合は、インスタンス自身を返したものとして扱う。
例１：

```
class Light(color:int){
        define operator ¦= (col:int){
                color ¦= col;
                }
        }

var     light = new Light(0xFF0000);            // 赤い光
writeln((light ¦= 0x0000FF).color);             // 青の光を混ぜた結果を表示
```

例２：

[8] 単項演算子を定義することはできない

```
class Car{
        field option[]:string;
        define operator += (parts:string)      option.append(parts);
        }

var     mycar:Car = new Car();
mycar += "car audio";
mycar += "navigator";
```

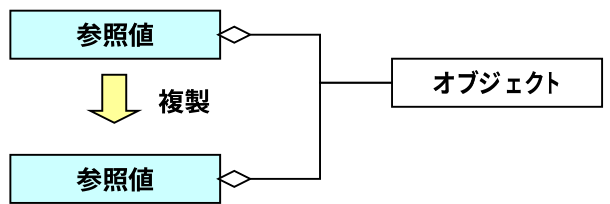
※注意：オプショナル演算子は、演算子を定義したクラスのインスタンスに対して右辺値を作用させる、最も優先度の高い２項演算子であり、右辺値は項でなければならない。
したがって mycar += "car"+" audio" は（mycar += "car"）+ " audio" と解釈する。もし結合された結果の文字列"car audio"**を作用させたいのであれば、**mycar += （"car"+" audio"）**とする必要がある。**

演算子の左辺値をなす被演算インスタンスと演算子の右辺値はいかなる場合も交換不能（例えば + **演算子を定義した場合、**A+B **は** B+A **とはできない）である。**

※オプショナル演算子の導入はプログラムの動作を直感的に捉えるには都合がよいが、間違った使用方法や実装、あるいは勘違いや思い込みによるミスを発見しにくい、という面も持っている。
あらかじめ同機能のメソッドを定義し、その動作を検証した上でオプショナル演算子の定義にてそのメソッドを呼び出すように実装することを推奨する。

11.4 オブジェクトの共有

オブジェクトは変数に代入することによって、その変数からの参照が確立する。
別の変数にオブジェクト参照変数を代入すると、この変数もオブジェクト参照変数となり、代入した変数と同じオブジェクトを参照することになる。
すなわちオブジェクト参照変数を複数の変数に代入することによって、参照のコピーが複数形成され、そのすべての変数がただ一つの実体オブジェクトを参照する。
※　オブジェクト自体のコピーがなされるわけではない点に注意。



例：

```
class   object{
        field   boo = 0;
        }
var     obj1 = new object();    // 新たなインスタンスを生成し、obj1 が参照
var     obj2 = obj1;            // obj2 も同じインスタンスを参照
        obj1.boo = 100;
        writeln(obj2.boo);      // 結果は 100 となる
```

あるオブジェクトを参照するオブジェクト参照変数に、別のオブジェクトへの参照を代入したりオブジェクト以外の値を代入することによって、そのオブジェクト参照変数からの参照は解除される。（後述のオブジェクトの削除を参照）

例：

```
class   object{}
var     obj = new object();     // オブジェクトを生成し変数に割り付ける
var     cp = obj;               // 参照のコピーを複製
var     cpx = cp;               // 参照コピーの更なるコピーを複製
        cp = 10;                        // 結果として obj、cpx がオブジェクトを参照
        cp = cpx;               // obj、cpx、cp がオブジェクトを参照
        obj= cpx = null;        // cp のみオブジェクトを参照
```

11.5 オブジェクトの消滅

オブジェクトの消滅は、オブジェクト参照変数からの参照がすべてなくなった時点で自動的に削除される。
また、deleteObject()メソッドの適用によっても強制削除される。
クラス・インスタンスの場合は、そのクラスに ondelete()によるデストラクタを宣言している場合、消滅のタイミングでデストラクタ・メソッドが呼び出される。

11.5.1 参照解消による自動消滅

オブジェクトは、そのオブジェクトを唯一参照しているオブジェクト参照変数の消滅などに伴う参照の解消によって自動的に消滅する。

```
var mycar=new Car("atlas");            ①新たなオブジェクトを生成する
var forsale=mycar;                     ②mycar と forsale の２個所から参照される
mycar=null;                            ③forsale からのみ参照される
forsale=null;                          ④参照されなくなったためオブジェクト消滅
```

また明示的な参照解除だけでなく、ブロックの終了に伴う参照変数の消滅によっても同様である。
例：

```
if(true){
        var obj = new Car("atlas");
        }←ここで変数 obj が消滅し、同時にその参照するオブジェクトも消滅する
```

オブジェクトがメソッドの引数に引き渡されて呼出された場合は、メソッド内からオブジェクトへの参照が仮引数を通じて確立し、メソッドからリターンする時点で参照が解除される。
ただし、この引数がリターン値として戻され、かつ戻り先で変数に代入されたり、別のメソッドの引数として利用される場合は参照が継続保持されるため、最後の参照が解消するまでオブジェクトは存続しつづける。
また、変数などへの代入を伴わない場合であっても、リターン値としてのオブジェクト参照を通じたメンバー（フィールドあるいはメソッド）をアクセスする場合は、そのアクセスが完了するまで参照が継続される。
例：

```
method  dateNormalizer(date:Date):Date{
        date.setDate(1);                        // 月の初日（１日）に変更
        return  date;                           // 受け取った引数を返す
        }
var     result:Date=dateNormalizer(new Date()); // 代入により result が参照する
writeln(result.toString());                     // Date オブジェクトを表示する
result = null;                                  // ここで参照解消して消滅する
```

オブジェクトが、内部に保有するオブジェクトを外部から参照されている場合、親オブジェクトがスコープを失って消滅する際、外部から参照されている内包オブジェクトは削除されることなく、親オブジェクトから独立したオブジェクトとして残る。（外部からの参照解除時に削除される）
但し、内包オブジェクトが親オブジェクトを参照している場合は、親オブジェクト自体も残る。
例：

```
var     obj = null;
if(true){
        object  parent{
                object  child{
                        field   name = "花子";
                        }
```

```
                obj = child;            // obj が参照する
                }
        }←ここで parent オブジェクトが削除されるが、child オブジェクトは独立して残る
writeln(obj.name);                      // “花子”を表示
obj = null; ←ここで child オブジェクトが消滅
```

11.5.2 オブジェクトの強制削除

オブジェクトを強制的に削除するには、オブジェクトを参照している変数に対し delete 操作をおこなう。
例：

```
var     boo = {1,2,3};
var     foo = boo;
delete  boo;                    // 変数 boo および配列{1,2,3}が削除される
writeln(foo);                   // null を表示する
```

11.5.3 所有参照による連動消滅

オブジェクト参照変数への代入に際し“=”演算子の代わりに“:=”演算子を使って代入すると、そのオブジェクトへの所有権が変数に与えられる。これを所有参照という。
所有権を与えられた変数がスコープを失って削除されると、その参照するオブジェクトも連動して消滅する。
例：

```
class Car{}
var     mycar = new Car();              // mycar はインスタンスを参照する
if(true){
        var     owner := mycar;         // owner がインスタンスを所有する
        } // ←ここで owner が削除され、その所有する Car インスタンスが消滅する
writeln(mycar);                         // null を表示する
```

11.5.4 コンポジション化による連動消滅

コンポジション化は、オブジェクト（全体オブジェクト）の構成部品を成すオブジェクト（部分オブジェクト）のライフサイクルを全体オブジェクトと同じにする必要がある場合に利用する。
composition プレフィックスをつけたフィールドに参照代入されたオブジェクトはコンポジション化され、全体オブジェクトの消滅に伴い連動して消滅する。
例：

```
class Engine{...}
class Tire{...}
class Car{
        composition field engine = new Engine();
        composition field tire[4] = {new Tire(),new Tire(),new Tire(),new Tire()};
        ...
        }
var  mycar = new Car();
var  parts = mycar.engine;
delete mycar;                           // ここで Car インスタンスが消滅する
writeln(parts);                         // null を表示する
```

コンポジション化しない場合、上記例では Engine インスタンスが外部変数 parts によって参照されているため、Car インスタンスの消滅時に engine フィールドから切離されて独立する。

コンポジション化は、相互参照によるオブジェクトの滞留を回避するうえでも有効である。

相互参照となる例を以下に示す。
例：

```
class   Family(name:string){
        field   parent:Family = null;
        field   children[0..*]:Family = {};
        }
var     mama = new Family("Mama");
var     child = new Family("Baby");
mama.children += child;        // mama が child を参照
child.parent = mama;           // child が mama を参照
mama = null,child = null;      // 変数から切離されるだけで消滅しない
```

相互参照があると、変数からの参照がなくなっても参照解消しないためそのオブジェクトは独立してシステム内に滞留（ゾンビ化）する。プログラム終了時には強制的に破棄されるが、オブジェクトの期待する消滅シーケンス（デストラクタの実行）を踏まないことになる。
これを回避するには事前に相互参照を解消するか強制削除によってオブジェクトを削除しなければならないが、コンポジション化をおこなえば、相互参照の場合であっても連動消滅されるため、滞留は起きなくなる。
例：

```
class   Family(name:string){
        field parent:Family = null;
        composition field children[0..*]:Family = {};
        }
var     child;
if(true){
        var     mama := new Family("Mama");
        child = new Family("Baby");
        mama.children += child;         // child オブジェクトをコンポジション参照（所有する）
        child.parent = mama;            // child オブジェクトが mama オブジェクトを参照
        }                               // ここで mama が消滅し、所有するインスタンスも削除
writeln(child);                         // null を表示する
```

コンポジション化によって、オブジェクトを保有する側と保有される側のライフサイクルが同じになるように設計できる。
上記例で、mama 変数は new Famiry("Mama")を所有参照しており、これがブロック終端でスコープを失う（消滅する）時点で、その参照（保有）するオブジェクトも同時に消滅する。するとこのオブジェクトの children メンバーも消滅するが、通常であればその要素が削除される時点で各々の要素が参照しているオブジェクトは参照解除となるだけである。ところがこの例では、children 要素は composition であり、要素の消滅とともにその参照するオブジェクトも消滅するため、new Famiry("Baby")オブジェクトもこの時点で消滅する。

11.6 オブジェクトの複製

オブジェクトは duplicate()メソッドあるいは clone()メソッドによって複製する。
例：

```
var     mycar=new Car("atlas");
var     secondcar=mycar.duplicate();
secondcar.name="atlas2";
writeln(mycar.name,":",secondcar.name);
```

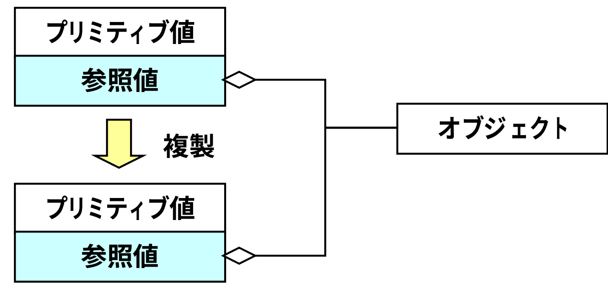
結果は“atlas:atlas2”となる。

複製は Object クラスのクラスメソッドとして記述してもよい。
その場合は、

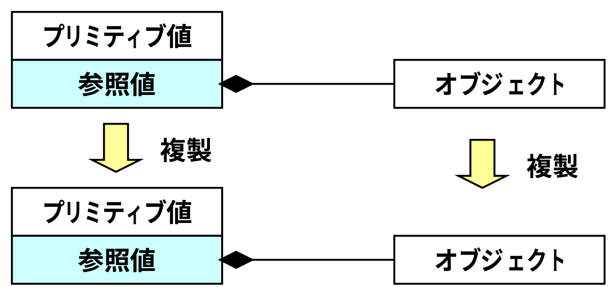
```
var     secondcar=Object.duplicate(mycar);
```

のように記述する。結果は同じである。

複製したオブジェクトのフィールドは複製元オブジェクトのフィールドをコピーしたものであり、フィールドが他のオブジェクト（あるいは自身でも）を参照している場合、そのオブジェクトの複製はおこなわない（shallow copy という）。したがってフィールドから参照されるオブジェクトは複製元ならびに複製先の両方から参照されることになる。



参照値が所有代入（:=演算子）による、あるいはコンポジション化されたオブジェクトの場合はそのオブジェクトについて複製し、それぞれが独立した所有（コンポジション）関係を維持する。



※注意　組み込みクラスおよびその派生クラスから生成したオブジェクトに duplicate()メソッドを適用することはできない。
ただし、コピー・コンストラクタを利用して同様の機能を実現できる。

11.7 クラス・アサイン（実行時多重継承）

オブジェクトは、Class.assign()メソッドによって、本来のクラス以外のクラスのインスタンスとして振る舞うように変更できる。この機能は多重継承を実現する場合にも有効である。

例：

```
class   orgcls(name,age){
        method  view()  writeln("My name is ",name,".");
        method  show()  view();
        }
class   othercls(age){
        method  view()  writeln("I’m ",age," years old.");
        }
var     obj = new orgcls("Jane",18);
obj.view();                                         ①
obj = Class.assign(obj,othercls);
obj.view();                                         ②
obj.show();                                         ③
```

結果は、

“My name is Jane.”	①の結果
“I’m 18 years old.”	②の結果
“I’m 18 years old.”	③の結果

となる。

①は本来の orgcls の view()メソッドを呼び出し、②では assign によって割り付けられた othercls クラスの view()メソッドを呼び出す。
③は show()メソッドがあるのは orgcls クラスであるため､そのクラスのメソッドを呼び出すが、そこでさらに呼び出される view()メソッドは othercls クラスのメソッドである。

クラス・アサインによって利用可能となるのはメソッドであり、field には影響しない。
アサイン先のメソッドはアサイン前のクラスと同名かつ同一の意味を持った field をアクセスするように配慮されていなければならない。
複数のクラスにおいて同名のメソッドを持つ場合、最後にアサインされたクラスのメソッドが有効となる。

12 シリアライズ

組込み関数および組み込みクラス、組込みオブジェクトを除き、すべてのクラス、関数、配列、ハッシュ、オブジェクト、基本データ型の値は、組み込みクラス Value の pack()メソッドを使用して文字列にシリアライズ化することができる。
シリアライズ化されたデータはその内容を操作してはならない。

文字列としてシリアライズ化されたデータは同じく組み込みクラス Value の unpack()メソッドによって復元できる。
ただし、クラスから生成されたオブジェクトをシリアライズ化したのち復元した場合は、内部にシリアライズ化時のメソッドを内包しているが、元クラスからは切り離された単独オブジェクトとなる点に注意する必要がある。

※注意：Date、Buffer、Wrapper 以外の組み込みクラス、あるいはその派生クラスのインスタンスおよび、これらのインスタンスを参照するオブジェクトをシリアライズ化してはならない。

12.1 オブジェクトのシリアライズ化と復元

オブジェクトを外部記憶に保存しておいて後で利用したり、オブジェクトをネットワークを通じて他のノードもしくはサーバに移送する場合はいったんシリアライズ化して、記憶したりまた転送してからこれを復元して利用する。
例：

```
class   Usrcls(a1,a2){
        method  add()   return  a1+a1;
        method  sub()   return  a1-a2;
        }
var     v1:Usrcls = new Usrcls(100,200);
var     p1:string = Value.serialize(v1);          ①
```

------ p1 の内容である文字列を他のネットワークノードへ転送 -------
↓
------ 他のネットワークノードから受信した文字列を p1 に格納 -------

```
var     r1:object = Value.eval(p1);               ②
Class.assign(r1,Usrcls);                          ③
writeln(r1.add());                                ④
```

① オブジェクト v1 をシリアライズ化した結果が p1 に格納される。（参照ではなく独立したデータである点に注目。
② シリアライズされた p1 を復元して生成したハッシュへの参照が r1 に格納される。
③ r1 の参照するハッシュを Usrcls クラスのインスタンスとして割り当てる。
④ r1 の示すオブジェクトに add()メソッドを作用する。

12.2 クラスおよび関数のシリアライズ化と復元

クラスをシリアライズ化してこれを他のノードもしくはサーバに移送する例を示す。
この例は、受け取り側でその処理方法がわからないオブジェクトを移送して利用を求める場合など、クラスを相手先に送って処理を依頼する場合（エージェントと呼ばれる）に有効である。
例：

```
class   Encript(data){
        field   agent:string;                     ①
        method  encode(){ … }                     ②
        method  decode(){ … }                     ③
        encode();                                 ④
        }
```

```
var     obj:Encript = new Encript(“元のデータ”);
        obj.agent = Value.serialize(Encript);              ★
var     senddata:string = Value.serialize(obj);            ⑤
```

----- senddata を通信などにより転送 -----

```
var     recvdata:object = Value.eval(senddata);            ⑥
var     agentcls:object = Value.eval(recvdata.agent);      ⑦
Class.assign(recvdata,agentcls);                           ⑧
recvdata.decode();                                         ⑨
var     data = recvdata.data;                              ⑩
```

①このクラス自体をシリアライズ化しこれを格納するフィールド
②暗号化メソッド（例）
③復号化メソッド（例）
④コンストラクタにて引数データを暗号化する
★クラスをシリアライズ化してオブジェクト内に格納する
⑤オブジェクト全体をシリアライズ化
⑥受信したシリアライズ化オブジェクトをハッシュに復元
⑦クラスを復元
⑧ハッシュを、復元したクラスに割り当てる（インスタンス化）
⑨クラスのメソッドを利用してデータを復号化
⑩複合化されたデータを取り出す

ここで示す例は暗号化アルゴリズムを逐次切り替えてデータを送る場合などに有用である。
ただし、盗聴者がその仕組みを知らないことが前提だが…。

関数のシリアライズ化と復元、また復元した関数の適用も同様である。
例：

```
function f(x){
        writeln(x);
        }
var     send:string = Value.serialize(f);
```

----- 通信による転送など -----

```
var     recv:object = Value.eval(send);
recv(“I’m original function”);        f(“I’m original function”)と同じ結果となる
```

上記は関数を他のノードなどに送る例である。

13 スレッド

クライアントサイド・インプリメントのみマルチ・スレッドに対応する

ユーザスレッドを構築する場合は、組込み基本クラス Thread の派生クラスとして構成する。
例：

```
class UserThread(message):Thread{
        field   mymessage = message;
        method  execute(){                          ①
                writeln(mymessage);
                }
        super(execute);                             ②
        }

var user_thread = new UserThread("Hellow");         ③
user_thread.Run();                                  ④
user_thread.waitExit();                             ⑤
```

① 実行プログラムを宣言
② スレッドを起動した場合に実行するプログラムを指定し、Thread 基本オブジェクトを構築。
③ 新しいスレッドオブジェクトを生成する。
④ 生成したスレッドを起動する。
⑤ スレッドが自己終了するまで待ち合わす。

Thread から派生するユーザ Thread クラスには実行プログラムとしてのメソッドを用意しなければならない。また、このメソッドは引数を持たない。
コンストラクタにて、実行メソッドを基本クラス(Thread)に引き渡すことによって、独立したスレッドが構成される。
実行メソッドの指定はコンストラクタ内の super()メソッドによって指定するため、コンストラクタ引数に応じて実行メソッドを選択するような実装も可能である。

13.1 基本メソッド

スレッド制御は以下のメソッドを介して行う。

メソッド	解説
Run()	super()コンストラクタに指定した実行プログラム（メソッド）を実行する。 スレッドは、他のスレッドおよびメインプログラムと非同期で実行される。
sleep(sec)	引数を指定しない場合および 0 を指定した場合は実行権の一時放棄として機能する。引数に 0 以外の値を指定した場合、その時間だけ実行を停止する。（精度は 0.001Sec）
waitEvent(event)	event で指定するイベントが発生するまで実行を停止する
postEvent(event)	event で指定するイベントをスレッドに通知する
waitExit()	指定したスレッドが実行状態（実行可能状態を含む）の間待ち合わせ、スレッドが終了する（実行プログラムとしてのメソッドから抜ける）まで制御を戻さない。
isExit()	isExit()メソッドは、指定したスレッドが実行状態であれば偽を、終了していれば真を返す。

※Thread クラスからの派生として実装せずに Thread クラスオブジェクトをそのまま使用する場合はメソッドの使用方法が異なるので注意。

13.2 同期機構

スレッド間の同期は、共有変数およびオブジェクトへの排他アクセスと、スレッド間イベント通知でおこなう。
排他アクセスはプログラムの安全性を目的として非自発的に排他をおこなう仕組みであり、イベント通知は自発的に排他を制御する目的で使用することを前提としている。
※デッドロックの問題についてはプログラマ自身によって明確に管理されなければならない。

13.2.1 排他

変数もしくはオブジェクトを占有宣言することによって、その占有を解除するまで後続スレッドが占有を完了するのを待たせることができる。
占有宣言は lock 変数名; ステートメントにておこなう。
占有できた場合、このステートメントを抜けて次のステートメントを実行できる。
もし、他のスレッドが先に占有していた場合、そのスレッドが占有を解除するまでこのスレッドの実行は停止する。

占有解除は unlock 変数名; ステートメントにておこなう。
占有を解除すると、その時点で占有を依頼している後続スレッドのリストのうち一番先に占有を依頼したスレッドの実行を開始する。

占有宣言した変数がオブジェクトを参照している場合は、占有は変数ではなくオブジェクトに対してなされる。
例：

```
class semaphore{}
var  obj1 = new semaphore();          // オブジェクト生成
var  obj2 = obj1;                     // その参照コピー

function thread1(){
        lock  obj1;                   // obj1 が示すオブジェクトを占有
        Thread.sleep(1.0);            // １秒間待ち合わせ
        unlock obj1;                  // 占有解除
        }

function thread2(){
        lock obj2;                    // obj2 が示すオブジェクトを占有
        unlock obj2;                  // 占有解除
        }

var     t1 = new Thread(thread1);     // thread1 スレッド生成
var     t2 = new Thread(thread2);     // thread2 スレッド生成
t1.Run();                             // スレッドを起動
t2.Run();
```

上記において、t2 が示す thread2 が obj2 を lock したとき、t1 が示す thread1 によって obj1 の lock が継続していれば、その unlock まで t2 で示す thread2 の実行は停止される。

lock および unlock の対象は、そのプログラム・ステートメントからアクセス可能な名前をもった変数であり、オブジェクト内のフィールドを指定することはできない。
ただし、オブジェクト内のメソッドからはオブジェクトのフィールドを個別に指定することができる。

13.2.2 イベント通知

スレッドはイベントの発生を待ち合わせたり、イベントの発生を検査したりできる。

また、他のスレッドに対してイベントを通知することもできる。

イベントは整数値で表される値として定義され、実際には２進数表記された場合の各桁を個別のイベントとして扱う。

イベントの発生を待つ場合、スレッド内において waitEventt(evnt);メソッドを呼び出す。
evnt に５を指定した場合、２進表記では 101 であるため、０番と２番のイベントを待ち合わせる、と解釈される。
※イベント番号は最下位ビットを０番とし、２進数の桁に応じて１番、２番、、、、と呼ぶ。
プログラムの読みやすさからは、waitEvent(5)よりも waitEvent(0b101);を薦める。

あるスレッドにイベントを通知する場合、postEvent(evnt);メソッドを使用する。
evnt で指定するイベントを対象スレッドに通知する。
もし、対象スレッドが該当するイベントを指定して waitEvent()を実行していれば、このスレッドはＷＡＩＴ状態を脱出する。
例：

```
function  thread1(){
        waitEvent(0b101);               // ０番か２番のイベント発生を待つ
        }

var     t1=new Thread(thread1);
        t1.Run();
        t1.postEvent(0b100);            // ２番イベントを通知する
```

14 ステートメント

14.1 if 文

書式１：if(式)ブロック
書式２：if(式)ブロック else ブロック

式を評価し、結果が真（あるいは真とみなせる）なら後続ブロックを実行する。偽なら後続ブロックを無視し、さらに else 節がある場合にはそのブロックを実行する。
例：

```
if(true) writeln("真です");
if(false) writeln("真です"); else writeln("偽です");
```

※記述上の注意：
else 節はそのもっとも近い if 節に対応するものとして解釈する。したがって、

```
if(式１) if(式２) ブロックＡ; else if(式３) ブロックＢ; else ブロックＣ;
```

は以下のように解釈される。

```
if(式１){
	if(式２) ブロックＡ;
	else{
		if(式３) ブロックＢ;
		else  ブロックＣ;
		}
	}
```

複雑な if 文を記述する場合は明示的なブロック指定（″{}″によって囲む）をおこなうべきである。

14.2 select 文

```
書式１：select( 式 ){
	case 文並び
	default 文
	}

書式２：select{
	case 文並び
	default 文
	}
```

＜case 文並び＞とは単独あるいは複数の≪case 式並び:ブロック≫
＜default 文＞とは単独の≪default:ブロック≫であり、省略もできる
＜式並び＞とは単独の式、もしくは複数の式をコンマで区切ったもの
＜ブロック＞とは単独のステートメント、もしくは複数のステートメントを{ }で括った複文
具体的な記述例は

```
書式１：select( 評価式 ){
		case 式: ブロック;
		case 式,式: {ブロック;ブロック}
		default: ブロック;
	}

書式２：select{
		case 式,式: ブロック;
		case 式: ブロック;
		default:{ブロック}
	}
```

となる。

書式１では順に、記述された case 文中の式並びの各々の式の結果と評価式の結果を比較し、いずれかと等しい場合はコロン（ ： ）に続くブロックを実行して select 文を抜け出す。
各々の case 文において、式並び中のどの式とも等しくない場合は次の case 文を評価する。
条件成立する case 文が複数ある場合でも、先に条件成立した case 文のみ実行する。
条件の成立する case 文がない場合のみ default 文を実行する。
例：

```
select(value){
case    "ABC":  writeln("value is ABC");

case    "X","Y": {
        writeln("value is X or Y");
        if(value == "X") writeln("value is X");
        else writeln("value is Y");
        }

default: writeln("Default");
}
```

複数の一致条件に対してブロックを実行する場合の case 文
case 式１,式２: ブロック;
を
case 式１: case 式２: ブロック;
と記述してもよい。

条件が一致する場合は何もおこなわない、とする場合は
case 式: ;
のようにブロックを記述せず case 文を終端するセミコロン（ ; ）のみを記述する。

書式２は書式１において以下の記述をした場合と同じである。

```
select(true){
  case  式並び: ブロック;
  case  式並び: ブロック;
}
```

参考：
if(式) ブロックＡ; else ブロックＢ; は
select{ case 式: ブロックＡ; default: ブロックＢ; }
と同じ制御となる。

14.3 switch 文

書式：

```
switch( 評価式 ){
    case 式並び: ブロック並び [break;]
    case 式並び: ブロック並び [break;]
    default: ブロック並び
}
```

評価式の結果を順に、列挙した case 文中の式の結果と比較し、いずれかと等しい場合はコロン（ ： ）に続くブロック並びを break 文を検出するか switch 文の終りまで実行する。この際後続の case 式並び、default ラベルは評価しない。
式並びとは、<式> {,<式>} のように複数の式をコンマ（ , ）で区切ったものである。

各々の case 文おいて、式並びのいずれの式も評価式と等しくない場合は次の case 文を評価する。どの case 文でも条件成立しない場合は default に続くブロック並びを、break 文を検出するか case 文を検出するか switch 文の終りまで実行する。
例：

```
switch(new Date().getMonth()+1){
case    2,4,6,9,11:     write("小の月"); break;
default:                write("大の月");
}
```

break が記述されていない場合は case 文の条件成立時に「小の月大の月」と続けて表示する。

ブロック並び中に break 文があれば switch 文を抜け出すが、レベルに２以上の break をおこなった場合は switch 文の抜け出しに１レベルを使用し、残りをその外側のループなどの break 機能に引き渡す。なおラベル付きの break の場合は該当ラベルのついた制御ブロックを抜け出す。
例：

```
for(var cnt=0; cnt < 10; cnt++){
        switch(cnt){
          case  6:      writeln("hit=",cnt);    break 2;
          default:      writeln(cnt);
          }
        }
```

上記は cnt 値が６になった時点で hit=6 と表示して for 文を抜け出す。

14.4 repeat 文

書式１：repeat ［:ラベル］ ブロック
書式２：repeat ［:ラベル］（繰返し数）ブロック
書式３：repeat ［:ラベル］（繰返し数 ; インデックス変数名 ［ = 初期値］ ）ブロック

書式１はブロック内で break 文を実行するまで、繰返しブロックを実行する。
書式２および書式３は＜繰返し数＞回ブロックを実行する。ただし繰返し数に達する以前にブロック内で break 文を実行した場合はそこで抜け出す。
書式３は、repeat に続く繰り返しブロック内にのみスコープされるインデックス変数を作成し、繰返しごとに１を加算する。
ラベルについては break 文 continue 文の節を参照のこと。

例：

```
var     text = "";
repeat(10)      text += "A";
writeln(text);                  // “AAAAAAAAAA”を表示する
```

繰返し数には整数または実数（整数部のみ採用）を値とする式を指定するが、その評価はループ開始以前に１回のみおこない、ブロック実行によって値が変わる変数を指定しても影響を受けない。
なおワーニングレベルが１以上で繰返し数に負の値を指定した場合は警告を表示する。
書式１と、その他の書式で繰返し数を指定しない、もしくは null を指定した場合は、ブロック内で break をおこなうまで繰返しブロックを実行する。

例：

```
repeat( ; index){                       // 書式３にて、繰返し数無指定（無限ループ）
        if(index >= 100 break;          // 条件 break （これがないと無限ループ）
        writeln(index);
        }
```

書式３で作成したインデックス変数のスコープはブロック内のみにスコープするローカル変数であり、初期値を指定しない場合は０に初期化する。
例：

```
var     idx = "ABC";                    // ※この変数はインデックス変数とは別
repeat(20; idx){
        writeln(idx);                   // インデックス変数の内容を表示する
        }
writeln(idx);                           // 変数 idx の内容（"ABC"）を表示する
```

上記例では、0,,,19 を表示した後 ABC を表示する。

14.5 for 文

書式： for [:ラベル] (前置文; 式; 後置文) ブロック

まず前置文を実行し、式の結果が真ならブロックを実行し、その後で後置文を実行し、再び式の評価をおこなう部分から繰り返す。
式の評価結果が偽ならブロックの次の文に制御を移す。
前置文、後置文は省略可能であるし、ブロックに空文を指定しても良い。
さらに式も省略でき、この場合は無条件に真として扱う。したがって、式を省略した場合は、ブロックにおいて break 文を記述してループを抜け出すようにしないと永久ループとなる。
ラベルについては break 文 continue 文の節を参照のこと。
例：

```
for(var a=1; a<100; a++) writeln(a);
for(var n=0,s=0; n<=10; n++,s+=n) writeln(s);
for(var i=0;;){
        if(i==10) break;
        writeln(i++);
        }
```

前置文内で宣言した変数は、for 文を抜けた後も変数として残り、かつ for 文内で変更された最終値を保持している。
例：

```
for(var a=1; a<100; a++) writeln(a);
writeln(a);                     // 100 を表示する
var a = 10;                     // 多重宣言エラーとなる
```

14.6 while 文

書式： while [:ラベル] (式) ブロック

式の結果が真の間ブロックを繰り返し実行する。
式が偽ならブロックを実行しないで次のステートメントに移る。
ラベルについては break 文 continue 文の節を参照のこと。
例：

```
var p = 10;
while(p--){
        writeln(p);
        }
```

14.7 do・while 文

書式１：　do［:ラベル］ブロック while(式);
書式２：　do［:ラベル］ブロック

書式１は、ブロックを実行し、while に続く式が真なら再び do ブロックを実行し、同様に while 式の評価を繰り返す。式の結果が偽なら制御は次のステートメントに移る。
ブロック内で break、continue、return 文を実行した場合は while 式を評価しない。
ラベルについては break 文 continue 文の節を参照のこと。
例：

```
var p = 10;
do writeln(p); while(p--);
```

書式２は、１回だけブロックを実行する。すなわち " do ブロック while(false); " の省略であり、" repeat(1) ブロック " と同じ動作をおこなう。
例：

```
var p = 10;
do{
	writeln(p);
	if(p--) continue;
}
```

14.8 foreach 文

書式：　foreach［:ラベル］(要素識別名 in 配列項［; 要素番号変数］) ブロック

配列項に指定する配列の要素を順に、要素識別（要素エイリアスという）に割付け、また要素番号変数を指定した場合には要素番号を変数の初期値（無指定時は 0）に加算した値を格納してブロックを実行する。要素識別名、要素番号変数は、続くブロック内でのみスコープを持つ。
ラベルについては break 文 continue 文の節を参照のこと。

要素番号変数を指定する場合は、配列項に続いてセミコロン（;）で区切り、
書式：　[var] 変数名［:int］［= 初期値］
初期値を指定しない場合は 0 とみなす。また要素番号変数は読み出しのみ可能である。
例：

```
var	arrays = {"A","B","C","D"};
foreach(element in arrays) writeln(element);
foreach(element in arrays; var elno) writeln(elno+1,"番目:",element);
foreach(element in arrays; var elno=1) writeln(elno,"番目:",element);
```

要素識別名で示す要素エイリアスは配列要素を直接参照するため、ブロック内にて要素エイリアスへの演算操作をおこなえば、対象としている配列要素に対して直接作用する。
また存在しない要素および null 値の要素はブロックの実行がスキップされる。
例：

```
var	arrays = {1, ,3,4};
foreach(element in arrays; var elno){
	write("arrays["+elno+"]=",element," ");
	if(element == 3) element = "ABC";
	}
writeln(); writeln("arrays=",arrays);
```

実行結果は、

```
arrays[0]=1 arrays[2]=3 arrays[3]=4
```

```
arrays={1,,"ABC",4}
```

と表示する。すなわち arrays[1]はスキップし、arrays[2]が"ABC"に置換される。

※ 注意：foreach ブロック内にて対象配列の要素を追加したり削除してはならない。
以前の書式 foreach [:ラベル]（要素識別名,要素番号変数名 in 配列項）ブロック　も可

14.9 break 文

書式１：break;
書式２：break　整数式;
書式３：break　論理式;
書式４：break　文字列式;
書式５：break ： ラベル;

repeat 文、for 文など繰返し文から抜け出し、制御をその文の次のステートメントに移す。また switch 文においては case ブロックから抜け出し、制御を switch 文の次のステートメントに移す。
例：

```
var  p = 10;
while(true){
        p--;
        if(p == 0) break;
        writeln(p);
        }
```

“break　整数式;”の形式は式の結果が 1 以上のとき意味を持ち、抜け出しレベル数を意味する。式の結果が 1 の場合は break;と同じ結果となり、結果が 0 および負の場合は break 文は機能しない。式の結果が論理値である場合、false は 0 として、true は 1 として扱う。
また、break null;は break;と同じ意味となる。
例１：

```
for(var s=0,c=0;;c++,s+=c) break(c >= 100);
```

上記は c が 100 に達した時点で for 文を抜け出す。

例２：

```
var     icnt=0 , jcnt=0;
for(icnt=0; icnt<10; icnt++){
        for(jcnt=0; jcnt < 10; jcnt++){
                if(icnt==5 && jcnt==5) break 2;
                writeln(icnt,jcnt);
                }
        }
writeln(icnt,jcnt);
```

上記は icnt,ｊcnt が共に５になった場合に外側の for ループを抜け出す。
なお、switch 文の case ブロック内に break 式;を使用する場合、この case ブロックの抜け出しに１レベル分使用することに注意のこと。

“break　文字列式;”の形式と“break : ラベル;”の形式は文字列もしくはラベル識別名に該当する直近のラベル付き繰返し文を抜け出す。
ラベルの書式は、繰返しステートメントに続き「:」に先導される識別名とする。
例：

```
repeat:exit1{
        repeat:exit2{
                if(...) break "exit1";  書式４
```

```
                if(...) break : exit2;  書式５
                }
        }
```

なお、break "";と break:;は共に break;と同じである。

14.10 continue 文

書式１：continue;
書式２：continue　整数式;
書式３：continue　論理式;
書式４：continue　文字列式;
書式５：continue　:　ラベル;

repeat 文、for 文など繰返し文のブロック内から抜け出し、再びループ条件判定をおこなう。
例：

```
var p = 10;
do{
        p--;
        writeln(p);
        if(p != 0) continue;
        } while(false);
```

“continue　整数式;” の形式は式の結果が 1 以上のとき意味を持ち、抜け出しレベル数を意味する。
式の結果が 1 の場合は continue;と同じ結果となり、結果が 0 および負の場合は continue 文は機能しない。式の結果が論理値である場合、false は 0 として、true は 1 として扱う。
また、continue null;は continue;と同じ。
例：

```
var     icnt , jcnt;
for(icnt=0; icnt<10; icnt++){
        for(jcnt=0; jcnt<10; jcnt++){
                if(icnt==5 && jcnt==5) continue 2;
                writeln(icnt,jcnt);
                }
        }
writeln(icnt,jcnt);
```

上記は icnt,jcnt が共に５になった場合 icnt=6,jcnt=0 から始める。

“continue　文字列式;” の形式と “continue　:ラベル;” の形式は文字列もしくはラベル識別名に該当する直近のラベル付き繰返し文まで抜け出し、この文を継続する。
ラベルの書式は、繰返しステートメントに続き「:」に先導される識別名とする。
例：

```
repeat:exit1(5){
        repeat:exit2(3; var i=0){
                if(...) continue "exit1";       書式４
                if(...) continue : exit2;       書式５
                }
        }
```

なお、continue "";と continue:;は共に continue;と同じである。

14.11 return 文

書式１：return;

書式2：return 式;

function もしくは method 内の任意の位置に記述でき、そこで指定した式の結果を function もしくは method の結果として返す。
式を伴わない場合の結果値は null となる。
例：

```
function f(arg){
	if(arg == null) return "データがありません";
	writeln(arg);
	return "OK";
	}
```

```
function div(a,b){
	if(b!=0) return a/b;
	}
writeln(div(10,2)," ",div(10,0));		// 結果は “5 null”となる。
```

```
function weekname(){
	const name={"Sun","Mon","Tue","Wed","Thu","Fri","Sat"};
	return name;
	}
```

この場合、return は関数内で宣言された配列オブジェクトへの参照を返す。
返された参照値を変数に代入しない場合は配列は消滅するが、参照が継続されている限り配列オブジェクトは存在している。

14.12 with 文

書式： with(オブジェクト参照値) ブロック

<オブジェクト参照値>（静的オブジェクト、インスタンス、配列、ハッシュ、クラス、レコード）を対象として、続くブロック内のステートメントを実行する。
オブジェクト参照値に上記種別オブジェクト以外（null 等）を指定するとブロックを実行しない。
また、ワーニングレベル1以上の場合でオブジェクトでも null でもない値を指定した場合は警告を表示して、同じくブロックを実行しない。
例：

```
class Boo(){
	field	a_value;
	static field	b_value;
	}
var obj = new Boo();
with(obj){
	a_value = 10;			// obj.a_value=10	と同じ
	this.a_value = 20;		// obj.a_value=20	と同じ※
	b_value = "ABC";			// Boo.b_value="ABC"	と同じ
	}
with(Boo) b_value = 100;		// Boo.b_value=100	と同じ

var ary = {1, 2, 3, {"A","B","C"}, 5, 6};
with(ary){
	writeln(this[2]);		// writeln(ary[2])と同じ
	writeln(this[3][2]);		// writeln(ary[3][2])と同じ
	}
```

```
with(ary[3]){
        writeln(this[1]);               // writeln(ary[3][1])と同じ
        }
```

※ with ブロック内での this は with 対象オブジェクトを指す。

注意：with ブロック内で with 文外の変数等をアクセスする場合、それと同名のメンバー（ユーザ定義および組込みメンバー）がオブジェクトに含まれる場合は、オブジェクト内メンバーが優先される。したがって安全を期す意味でも With ブロック内でのステートメントはオブジェクトのメンバーを対象とした操作に限定すべきである。

14.13 eachof 文

書式：eachof ［:ラベル］（オブジェクト配列）ブロック;

すべての要素がオブジェクトである配列の各要素に対し、ブロックを実行する。
foreach(element in オブジェクト配列) with(element) ブロック;
と等価。
例：

```
class   Boo(foo:int=0);
var     boos[]:Boo = new Boo[]{new Boo(), new Boo(10), new Boo(100)};
eachof(boos) foo *= 2;                  // 各々の要素の foo 値を 2 倍する
eachof(boos) writeln(this.foo);         // 0 20 200 と表示する
```

ブロック内での this は順次対象とされる個々のオブジェクトを指す。

※配列要素にオブジェクト以外が含まれている場合、ワーニングレベルが 1 以上ではアボートするが、ワーニングレベル 0 ではスキップする。

14.14 create 文

書式： create ＜変数名｜配列要素指定子｜メンバー名＞ ［ = 式］

create に続く変数名もしくは配列要素、オブジェクトのメンバーを生成する。
すでに存在する場合は新たに生成することはしない。
後続する代入文があればこれを評価し、値を設定する。
例：

```
var     hash = {a:10,b:"10"};
create hash.c = "ABC";
writeln(hash);                  // {a:10,b:"10",c:"ABC"}と表示
```

```
var     ary = {1,2,3};
create  ary[3] = 4;
writeln(ary);                   // {1,2,3,4}と表示
```

変数を生成する場合、生成される変数はグローバル変数として、root 直下に生成される。
with ブロック内で、対象オブジェクト内に新たなメンバーを生成する場合は
create this.メンバー名; あるいは with 項を指定してそのメンバーとする必要がある。
例：

```
var     hash = {a:10,b:"10"};
with(hash){
        create  this.c = "ABC";         // this は hash を指す
        create  hash.d = 100;
}
```

14.15 delete 文

書式：　delete [&¦@]<対象> [, [&¦@]<対象>]
<対象>≡<変数｜配列要素｜メンバー｜変数エイリアス>

<対象>に指定した変数もしくは配列要素、オブジェクトのメンバーを削除する。
ただし、<対象>に変数エイリアスを指定した場合は変数エイリアスが指す実体、すなわち変数が<対象>となる。
<対象>変数もしくは配列要素がオブジェクトを参照している場合、そのオブジェクトも同時に削除される。
例：

```
var     a = {1,2};
delete  a;
```

上記は、変数 a とその参照する配列{1,2}が共に削除される。

```
var     a = {1,2};
var     b = a;
delete  b;
```

上記は、変数 b とその参照する配列{1,2}が共に削除されるとともに、変数 a は null となる。
変数 a およびその参照するオブジェクトに影響を与えずに、変数 b のみを削除するには delete b に先立ち、b = null として参照解除すること。

配列の最後の要素を delete した場合は、配列の length 値が-1 される。
ただし、最後の要素以外を delete した場合、該当要素は null になるが length 値は変わらない。
例：

```
var     a = {1,2,3};
writeln(a.length);              ①3を表示する
delete  a[a.length-1];
writeln(a.length);              ②2を表示する
delete a[0];
writeln(a.length);              ③2を表示する
```

オブジェクトのメンバー（フィールドおよびメソッド）を削除することもできる。
例：

```
class   Cls{
        field   ab;
        method  setup(x){
                ab = x;
                }
        }
var     obj = new Cls();
delete  obj.setup;
```

結果として obj オブジェクト内から setup()メソッドが削除される。

delete @<対象>の場合は、<対象>がオブジェクトを参照している場合のみ、その参照しているオブジェクトを削除する。対象の値は null となるが対象自体は存続する。

delete &<対象>の場合は、<対象>そのものを削除し、<対象>（変数等）がオブジェクトを参照していてもそのオブジェクトは削除しない。ただし、オブジェクトが<対象>のみから参照されている場合はオブジェクトも同時に削除される。（参照解除による自動消滅）
<対象>が変数エイリアスの場合は、エイリアスそのものが削除されエイリアスが指していた実体の変数は影響をうけない。

※ foreach 文中の配列要素エイリアスに対して & を指定した delete はおこなってはならない。
例：

```
foreach(element in {1,2,3}){
        delete  @element;               // OK
        delete  element;                        // OK
        delete  &element;               // NG
}
```

14.16 enum 文

書式： enum グループ名{ 識別名並び }
識別名並び≡ 識別名宣言 {,識別名宣言}
識別名宣言≡ 識別名｜識別名=値｜識別名:値

enum で示すグループは、識別子で規定する個々のオブジェクト（enum オブジェクトという）に値を割り当てた、オブジェクトの集合となる。
＜識別名＞のみを宣言する場合、初期値０から順に正整数（順序数）を割り当てる。
＜識別名＞=＜値＞もしくは＜識別名＞:＜値＞の書式において、値に整数を宣言した場合、以降に続く識別名のみの宣言でこの整数+1 の値を起点として順次割り当てる。またこの書式の場合のみ、次の＜識別名宣言＞との区切りとして（,）の代わりに（;）を使用できる。
※値には式も指定できるが結果として null となる場合は＜識別名＞のみを指定した場合と同じ扱いとなる。
例：

```
enum Week{SUN,MON,TUE,WED,THU,FRI,SAT}                  ①
enum Post{社長=1,部長,課長,主任=課長,係長}              ②
enum 成果{成功=true,失敗=false,どちらともいえない}      ③
```

① は曜日を表す識別名に順次順序数を割り当てる。すなわち SUN には 0 を SAT には 6 が割り当てられる。
② は役職の序列を数値化する例である。筆頭順位の社長には 1 を以下順に序列が下がるほど値は大きくなるとする例である。主任=課長という宣言は、課長と主任は同じ序列であるという表現となる。
③ は識別子を順序数以外の値に割り付ける例である。この場合の「どちらともいえない」という値には整数 2 が割り当てられる点に注意。

enum で宣言した識別名は、プログラム中においてオブジェクト定数として評価され、割り当てられた値を明示的に得るには value フィールドを参照する。また、enum オブジェクトは定数であるため、値を代入することはできない。
例：

```
writeln(TUE);                   // “enum Week.TUE{2}”と表示する
writeln(TUE.value);             // 2 を表示する
```

上記例において“enum Week.TUE{2}”という表示は、Week グループの TUE という enum オブジェクトであり、値として整数 2 を割り付けられていることを示す。

enum で宣言した識別子のスコープは定義したブロック内に限定されかつ宣言順になされる。
同じ識別名を enum にて複数宣言した場合は、目的ごとにグループを指定して記述する必要がある。
例：

```
enum friend{太郎,次郎,花子,あゆみ};
enum family{次郎,明子,裕太};
writeln({次郎.value , friend.次郎.value , family.次郎.value});
```

結果は{1,1,0}と表示する。

グループを指定しない識別子は宣言順になされるため、“次郎”に対しては friend グループ内の“次郎”が採用される。これに対し、グループを指定した“family.次郎”は曖昧性がない。

enum オブジェクトを参照する変数を宣言する場合の型名にはグループ名を指定する。
例：

```
var wkval:Week = TUE;          // 変数 wkval は TUE オブジェクトを参照
writeln(wkval);                // “enum Week.TUE{2}”と表示する
writeln(wkval.value);          // 2 を表示する
```

ワーニングレベルを１以上に設定してある場合のみ、変数への代入時に、型で指定するグループと enum オブジェクトの属するグループが一致するかどうかを検査する。

14.16.1 クラス定義内に記述する enum 文

クラス定義内に enum 文を記述した場合、その enum オブジェクトはクラスメンバーと同様クラス・メソッドおよびインスタンス・メソッドのどちらからも参照できる。
例：

```
class   MyClass{
        enum    state {OFF,ON};
        static  method  view_static(){
                writeln(ON);
                }
        method  view(){
                writeln(OFF);
                }
        }

var     boo = new MyClass();
boo.view();                             // enum state.OFF{0} を表示する
MyClass.view_static();                  // enum state.ON{1} を表示する
```

14.16.2 enum オブジェクトの比較

enum オブジェクト同士の比較は割り当てられた順序数の比較をおこなう。
例：

```
enum    曜日{日,月,火,水,木,金,土}
enum    惑星{水,金,地,火,木,土,天,海,冥}
var     week:曜日 = 金;              // 変数 week は曜日.金を参照する
writeln(week == 金);                 // “true”を表示する
writeln(week == 5);                  // “true”を表示する
writeln(week > 木);                  // “true”を表示する
writeln(week == 惑星.金);            // “false”を表示する
```

14.16.3 #if－#endif と enum の組み合わせ

enum 文を利用してプログラムの構成を変更することができる。
例：

```
enum    選択肢{本番,デバッグ}
enum    構成{
        仕様 = 本番;     ※「本番」が選択肢グループ以外にないため「選択肢」を省略可能
        }
#if (仕様 == 本番)
```

```
        本番用ステートメント
#else
        デバッグ用ステートメント
#endif
```

上記の例は、プログラム全体にわたって本番用とデバッグ用の記述を切替えて使用したい場合に、enum 構成グループの仕様という識別値を変更するだけでよい。
デバッグ用プログラムを残したままで本番運用が容易におこなえるし、問題が発生した際にプログラムを大きく書き直すことなくデバッグ用コードを有効にできる。

14.16.4 enum オブジェクトのフィールド

enum オブジェクトには以下のフィールドが用意されている。
記述例には以下の enum 宣言を用いた例を示す。
enum　曜日{日,月,火,水,木,金,土}

フィールド	記述例	値の意味
ident	曜日.月.ident	識別名を文字列（例では"月"）として得る
prev	曜日.月.prev	前の enum オブジェクト（曜日.日）を参照
next	曜日.月.next	次の enum オブジェクト（曜日.火）を参照
top	曜日.月.top	先頭の enum オブジェクト（曜日.日）を参照
last	曜日.月.last	最後の enum オブジェクト（曜日.土）を参照
group	曜日.月.group	enum 曜日グループ全体を参照
value	曜日.月.value	曜日.月の割付値を示す（この場合順序数の 1）

prev と next フィールドについては、該当するオブジェクトがない場合は null である。

14.16.5 enum オブジェクトのシリアライズ化と復元

enum オブジェクトを参照する変数をシリアライズすると、enum オブジェクトのまま書き出す。
アンシリアライズをおこなう場合、その場所で同じ enum グループが参照可能であれば同じく enum オブジェクトとして復元されるが、参照できない場合は割り付けられた値のみ復元する。
また、復元時に参照するグループが本来のグループとは別のグループであっても、グループ名と enum オブジェクトの識別名が一致すれば、そのオブジェクトとして復元するため、割当順序数が異なることにもなる点には注意が必要である。

14.16.6 enum オブジェクトの高度な利用例

enum オブジェクトは定数オブジェクトであり、階層化したりエイリアス化して利用することができる。
例：

```
enum    曜日{日,月,火,水,木,金,土}
enum    惑星{水,金,地,火,木,土,天,海,冥}
enum    動物{人,サル,犬:enum{小型犬,大型犬},猫,鳥}                ①
enum    火{曜日:曜日.火, 惑星:惑星.火,無関係:動物}                ②
```

① 犬という enum オブジェクトはさらに小型犬と大型犬に細分化されたグループオブジェクトであることを示している。階層が異なるグループを定義した場合、割当順序数は 0 からおこなわれる。したがって「動物.犬.大型犬.value」 は 1 である。
② 「火.曜日」が「曜日.火」オブジェクトを、「火.惑星」が「惑星.火」オブジェクトを参照し、「火.その他」が動物グループを参照していることを示している。これは「火とは曜日の場合は火曜日であり、惑星の場合は火星、無関係なのは動物である」という記述を表現しているとみなすことができる。

enum グループ内の enum オブジェクトを識別名称として取り出すには getMember() メソッドを利用する。また、通常のオブジェクトもしくはハッシュと同様、配列要素参照記述によっても enum オブジェクトを評価できる。
例：

```
enum    SWITCH{ON,OFF}
var     boo:string[] = SWITCH.getMember();             // 「{"ON","OFF"}」が返される。
writeln(SWITCH.ON);                                     // enum SWITCH.ON{0} と表示する。
writeln(SWITCH["OFF"]);                                 // enum SWITCH.OFF{1} と表示する。
foreach(element in boo) writeln(SWITCH[element]);       // 順次取り出して表示する
```

14.17 exit 文

書式： exit

AbortException 例外を発行する。

15 特殊構文

15.1 tag 文

書式：< 記述 >

アングルブラケット（“<”と“>”の組）に囲まれた文字列は、これを tag 文とみなす。tag 文内では改行に続く連続するタブ文字を無視する。

tag 文内に“%%”で括ったＮＡＬステートメントがある場合はこれを解釈実行し、その出力結果を合成して全体をタグとする。

例：

```
var     imgs = {"img1.gif","img2.gif"};
foreach(spec in imgs) <IMG src="%% write(spec) %%" /> ;
```

上記は

```
<IMG src=”img1.gif” /><img src=”img2.gif” />
```

と出力する。

なお、tag 文直後に“;”を付加すると、該当タグ出力に続いて改行文字を出力する。

tag 文も単独のステートメントであり、文法上ステートメント区切りとして“;”を記述する必要があってもタグにつづいて改行を期待しない場合は tag 文と ; の間に１文字以上のスペースもしくはタブコードを挿入すること。

tag 文内に“%{”と“}%”で括った単独の式の値が null でない場合、この値を文字列化して合成する。

※この場合は式のみを記述し、“%%”で括る場合のように出力命令を記述してはいけない。

メソッド呼び出しを記述する場合、メソッドのリターン値が null 以外だとその値を埋め込む点に注意。

例：

```
var     imgs[]:string = {"img1.gif","img2.gif"};
foreach(spec in imgs) <IMG src="%{ spec }%" /> ;
```

上記は

```
<IMG src="img1.gif" /><img src="img2.gif" />
```

と出力する。

15.2 output 文

書式１：式 $

書式２：%{ 式 }%

＜式＞に続いて“$”を記述するか、＜式＞を“%{”と“}%”で括った場合、式の内容が null でない場合に、これを文字列化し標準出力に出力する。

例：

```
10+1.345$                              // 11.345 が表示される
%{ "abcd"+324 }%                       // abcd324 が表示される
```

ドキュメント構成例：

```
var     optionItem:string = 〝日月火水木金土〟;
<SELECT>
repeat(optionItem.length; var choice){
<OPTION value=〟%{choice}%〟>%{optionItem.charAt(choice)+ 〝曜日〟}%</OPTION>
```

```
}
</SELECT>
```

※上記例で≪ %{choice}% ≫については TAG 文の解説を参照のこと。

15.3 pass 文

書式　%%＜任意テキスト＞%%

＜任意テキスト＞部分を標準出力に出力する。
NAL プログラムと連携してドキュメントを構成する際に、固定文字列ならびに HTML タグをそのまま出力する用途で使用する。
＜任意テキスト＞中に"%{"と"}%"[9]で囲んだ範囲に式を記述すると、式の値が null でない場合のみ文字列化して埋め込む。
例：

```
var     host = "www.act.ne.jp";
var     title = "サンプル";
%%<a href="http://%{host}%/nal/samle.html">%{title}%</a>%%
write("<a href=\"http://"+host+"/nal/samle.html\">"+title+"</a>");
```

上記はともに同じ内容を出力する。

pass 文の開始を表す %% に続く、連続する復帰改行文字（\r\n）およびそれに続く連続するタブコード(\t)は出力対象とはならず、タブコード検出後はタブコード以外の文字が出現した位置から出力対象となる。この位置までにカウントされたタブコード文字数を行頭タブとして、任意テキスト中の改行文字直後に続く行頭タブコードを削除して出力する。
例：

```
%%<改行>
<TAB><TAB>この文は<改行>
<TAB><TAB>以下のように出力されます<改行>
<TAB><TAB><TAB>インデントを揃えるのに便利です<改行>
%%
```

<TAB><TAB>が行頭タブの指定であり、結果は<改行>直後の<TAB><TAB>が除かれて

```
この文は<改行>
以下のように出力されます<改行>
<TAB>インデントを揃えるのに便利です<改行>
```

となる。

※"%%"および"%{"をテキスト文字列として出力するには各々を文字列"%%"および"%{"[10]として記述する必要がある。

15.4 here-document 文

書式　%<< 変数名 :＜任意テキスト＞>>%

"%<<" に続く名前で示す変数に、続く":" 直後から ">>%" 直前までの＜任意テキスト＞部分を文字列として格納する。
＜任意テキスト＞中に"%{"と"}%"[11]で囲んだ範囲に式を記述すると、式の値が null でない場合のみこれを文字列化して埋め込む。
例：

```
var     name = "田中一郎";
```

[9] Rel-20050111 にて実装
[10] "%{"パターンは文字列表記中にあっても埋め込み記号であるから実体表現を使用する必要がある

```
%<< 挨拶 :こんにちは、私は%{name}%です。>>%
var 挨拶 = "こんにちは、私は田中一郎です。";
```

上記はともに同じ結果となる。

任意テキストの開始を表す : に続く、連続する復帰改行文字（\r\n）およびそれに続く連続するタブコード(\t)以外の文字以降が対象任意テキストとなる。この位置までにカウントされた連続するタブコード文字数を行頭タブとして、任意テキスト中の改行文字直後に連続する行頭タブコードは任意テキスト中には含まれない。
例：

```
%<< strvar :<改行>
<TAB><TAB>この文は<改行>
<TAB><TAB>以下のように出力されます<改行>
<TAB><TAB><TAB>インデントを揃えるのに便利です<改行>
>>%
```

<TAB><TAB>が行頭タブの指定であり、結果は<改行>直後の<TAB><TAB>が除かれて

```
この文は<改行>
以下のように出力されます<改行>
<TAB>インデントを揃えるのに便利です<改行>
```

となる。

※">>%"および"%{"をテキスト文字列として扱うには各々を文字列">>\%%"および"\%{"として、これを"%{"と"}%"で括って式として埋め込むか、">>%"および"%{"とする必要がある。
また if 文など制御文のブロックとして記述する場合は{ }で括る必要がある。

[11] Rel-20050111 にて実装

16 プログラム制御文

16.1 #using 宣言

書式：#using　＜識別名＞ {,＜識別名＞} ;

識別名にモジュール名、クラス名もしくはオブジェクト参照変数名を指定し、その識別名を通じてしかアクセスできないメンバーに対し、直接アクセスできるようにスコープを拡張する。
例：

```
class   Engine{
        field   power:int = 120;
        field   torque:int = 15;
        }

class   Car(name:string){
        field   engine:Engine = new Engine();           // Car の構成部品
        method  view(){
                #using  engine;
                writeln("%s %dKw %dKg/m".import(name,power,torque));
                }
        }
```

上記例において、method view()中のステートメントで参照する power と torque は field engine が参照するインスタンス内のメンバーであり、本来は engine.power あるいは engine.torque としてしかアクセスできない。#using engine として宣言した場合、例えば識別名 power がスコープ内に見つからない場合には#using 指定された識別名 engine を検索対象とし、engine 内の power へのアクセスを試みる。
#using に指定した識別名を検索対象とするのはスコープ内での名前解決ができない場合であり、検索の優先度を変更するものではない。
例：

```
var     foo = 100;
if(true){
        #using  boo;
        module  boo{
                field   foo = 200;
                }
        writeln(foo);                   // 上位変数 foo（100）を表示する
        }
```

#using に複数の識別名を指定した場合、先にアクセス可能なものが対象となる。
例：

```
#using  module1,module2;
module  module1{
        method  foo()   writeln(100);
        field   boo_i:int = 10;
        }
module  module2{
        method  foo()   writeln("ABC");
        field   boo_s:string = "abc";
        }
foo();                  // より前方に定義されている module1.foo()を実行
writeln(boo_s);         // module2.boo_s を表示
```

16.2 #eval による文字列式解釈

書式： #eval ＜文字列式＞

文字列式をステートメント並びとして解釈し実行する。
eval()関数は文字列を解釈した結果の値を返すが、#eval は文字列をステートメントとして解釈実行しプログラム中での作用を及ぼす。すなわち#eval の位置に文字列式の内容をステートメントとするプログラム記述がなされていることと同じ効果をもつ。
例：

```
var     s = 0;
var     stmt =  "repeat(10; var idx=1){"
                        "s += idx;"
                "}"
                "writeln(s);";
#eval stmt;     ①
#eval stmt;     ②
```

上記は、stmt 値が "repeat(10; var idx=1){s += idx;}writeln(s);" であるから

```
var s = 0;
repeat(10; var idx=1){s += idx;}writeln(s);          // ①に相当
repeat(10; var idx=1){s += idx;}writeln(s);          // ②に相当
```

として解釈実行される。

#eval が解釈する文字列は構文として完結していなくてはならない。

```
#eval "if(false) writeln(100+200)";
else writeln("error");                  // ここで SYNTAX ERROR が検出される
```

なお、文字列を解釈し、その結果値を式中で使用する場合は eval(文字列式)関数を使用する。
eval()は文法上、文字列全体を"{}"で括ったものを解釈実行し、最終ステートメントとしてのブロックの値を返す。従って、"var foo = 100"という文字列を#eval で解釈させた場合は同じブロック内に foo という変数を生成してその値が 100 となるが、eval()の場合は {var foo = 100} と解釈するため、記述したブロック内に変数 foo は生成されない。（下位ブロックに生成されるが eval()からのリターン時に消滅する）
例：

```
var     exp = "100+200";
writeln(10+eval(exp));                  // 310 を表示する
```

例：

```
var     stmt:string = "Hello";
#eval   "var "+stmt+":string = ' world.'";
//↑ <var Hello:string = ' world.'> として解釈実行
writeln(stmt+eval(stmt));               // "Hello world."を表示する
//↑ 変数 stmt の内容 "Hello" と 変数 Hello の内容 " world." を結合した結果を writeln()
```

※ **注意：#eval でクラス定義など、ステートメント以外を実行することはできない**

16.3 include 文

書式１：include path 文字列式;
書式２：include path 文字列式 : プロトタイプ並び;

プロトタイプ並び≡プロトタイプ[,プロトタイプ並び]

プロトタイプ≡ ［<function｜method｜class｜record>・］ 名前

path 文字列式はスクリプトソースのパスを基準とする相対パスもしくは HTTP プロトコル URI を指定する。
例：

```
include “lib/module1.nlb”;
include “http://xxxxx/lib/module2.nlb”;
```

なお、HTTP プロトコル URI を指定する場合、その対象がファイルである必要はなく、レスポンスとして返されるドキュメントがスクリプトソースであれば CGI 等の結果であってもよい。

書式１の場合は、path 文字列式で示すソースファイルが一度もローディングされていなければ、これを直ちにローディングし評価するが、既にローディングされていればローディングはおこなわない。
include 指定は、path 文字列式で指定するソースファイル内容が、そのままこの位置に展開されたように機能する。

書式２の場合は、プロトタイプもしくはプロトタイプ並びで指定する名前がプログラム内から最初に参照された時点で path 文字列式で示すソースファイルをローディングする。したがって include 宣言されていても、プロトタイプに該当する名前が参照されない限りローディングはおこなわれない。また既にローディングされていれば再度ローディングすることはない。
例：

```
include  “lib/gui.nal”:function funcs,class classes;
var newcls=new classes();          // ここでローディングされる
var funcret=funcs();               // 既にローディングされているため再ロードしない
```

※#include”キーワードも include 文として扱う。
例：

```
#include “sub_block.nal”;
```

16.3.1 エラー例外

include 対象として指定したソースファイルが存在しない、もしくはローディングの過程でエラーを生じた場合（多くの場合 http://～ によるネットワーク・インクルードで発生）には、RuntimeException 例外を発行する。
利用例：

```
try     include "http://" + main_server + "/nalsrc/xxxxx.nlb";
catch(err:RuntimeException)
        include "http://" + alt_server + "/nalsrc/xxxxx.nlb";
```

上記は、main_server からのローディングに失敗した場合に代替の alt_server からのローディングを試みる。
※注意：上記を

```
try{
        include "http://" + main_server + "/nalsrc/xxxxx.nlb";
}catch(err:RuntimeException){
        include "http://" + alt_server + "/nalsrc/xxxxx.nlb";
}
```

のように try-catch ブロック内に記述した場合、try-catch 節の外にインクルードしたソースのスコープが及ばなくなり、後続プログラムから利用できない。
include 文によって発行される可能性のある例外は RuntimeException のみであるため、ほとんどの場合以下のように記述してよい。

```
try     include "http://" + main_server + "/nalsrc/xxxxx.nlb";   // メインから
catch() try include "http://" + alt_server + "/nalsrc/xxxxx.nlb";// 代替から
catch() include "http://" + alt_no2_server + "/nalsrc/xxxxx.nlb";// さらに別の代替から
```

16.3.2 HTML テンプレートとしての利用

プログラムロジックと表現を分離する手法として HTML テンプレートを使用する場合にも include 文を使用する。テンプレートとして使用する HTML 形式ファイルは HTML エディタなどのツールを使用してデザインする。こうして出来上ったテンプレート"template.htm"を例に示す。

```
<html>
<body>
<table border="1">
<tr>
<!--make cell%% foreach(item in arg){ %%-->
<td>%{item}%</td>
<!--%% } %%end-->
</tr>
</table>
</body>
</html>
```

※path 文字列式の拡張子部分に".html"もしくは".htm"を指定した場合は、そのファイルを HTML ドキュメントとしてインクルードする。この場合は、文法的に include 文の前後に"%%"（パス文）を自動挿入して解釈するため、include されるファイルの先頭と最後に"%%"を記述する必要はない。

このテンプレートを利用するプログラム例は

```
var    arg = {"ABC","123"};
include  "template.htm";
```

とする。
ここで、

```
<!--make cell%% foreach(item in arg){ %%-->
<td>%{item}%</td>
<!--%% } %%end-->
```

の部分がテンプレートとしての記述である。
HTML エディタでデザインすべき点は

```
<td>%{item}%</td>
```

であり、テーブルセルの内容として item という値を表示することを示している。
※ %{}%は PASS 文中への値の埋め込み機能であり、PASS 文の解説を参照のこと。

```
<!--make cell%% foreach(item in arg){ %%-->
```

は%%によって PASS 文を終了し、続く記述が NAL ステートメントであることを示す。
また NAL ステートメントが終了すれば再び PASS 文を有効とするために%%を記述する。

これを実行すると。

```
<html>
<body>
<table border="1">
<tr>
<!--make cell-->
<td>ABC</td>
<!---->
<td>123</td>
<!--end-->
</tr>
</table>
</body>
</html>
```

と出力する。

16.3.3 #archive 制御文によるアーカイブファイル指定

書式： #archive アーカイブパス文字列式;

#archive 制御文を指定することによって、インクルードするファイルをアーカイブファイルにまとめることができる。
この場合、include で指定するファイルが存在しない場合、#archive に続くパス名（文字列値）で示すアーカイブファイル内に収容されている場合は、これをローディングする。
例：

```
#archive "application.zip";
include "module1.nlb";
include "module2.nlb";
```

と記述した場合、例えば"module2.nlb"ファイルが同一ディレクトリ内に無い場合には"application.zip"ファイル内にアーカイブされている"module2.nlb"を使用する。

この機能は、インクルード対象としてアーカイブファイルの使用を前提として、一時的にモジュールの機能を変更したい場合に使用したり、アプリケーションをパッケージ化してまとめる場合などに使用する。
#archive の指定をおこなわなかった場合は"nalib"もしくは"nalib.zip"を検索する。
アーカイブファイルは zip 形式ファイルでなければならず、圧縮の有無を問わない。

16.4 import 文

書式１：import パッケージ名[,パッケージ名];
書式２：import パッケージ名[,パッケージ名] : ライブラリパス文字列式;

ライブラリ内のパッケージ名を import に続く名前に指定する。
パッケージ名は具体的な名前を指定するか、「グループ名.*」のように、グループを指定することもできる。
例：
ライブラリ内に"util.sample.sampleMethod"として
method sampleMethod(){ ... }
が収容されている場合、

```
import util.sample.*;
sampleMethod();
```

と記述することによって、sampleMethod()が呼び出される時点すなわち sampleMethod の名前解決ができない場合にライブラリ内の"util.sample.sampleMethod"パッケージに記述されている method sampleMethod()を呼び出す。また一度呼出されたパッケージはプログラム内に組み込まれ、次回の sampleMethod()呼出しからはライブラリからではなく組み込まれたモジュールを利用する。

書式２では検索対象として、ライブラリ・パス文字列式で示すライブラリファイルを使用する。ライブラリのパス記述はスクリプトソースのパスを基準とする相対パスでなければならない。

16.4.1 #library 制御文によるライブラリ指定

書式： #library ライブラリパス文字列式;

以降に出現する import 文で指定する名前のモジュールが収容されているライブラリ・ファイルのパスを指定する。パス記述はスクリプトソースのパスを基準とする相対パスでなければならない。

例：
“debuglib”と”releaselib”というふたつのライブラリファイル内に、ともに”util.sample.sampleMethod”というパッケージが含まれていることを前提に、

```
#if defined(DEBUG)
    #library "debuglib";
#else
    #library "releaselib";
#endif

import util.sample.*;
sampleMethod();
```

とした場合、DEBUG が定義されていれば”debuglib”内の”util.sample.sampleMethod”パッケージを使用し、定義されていなければ”release”内の”util.sample.sampleMethod”パッケージを使用する。

16.4.2 ライブラリファイル

ライブラリファイルは“nalib”あるいは“nalib.zip”という zip 圧縮形式ファイルを既定とする。import 文で指定するパッケージがこのファイル内にない場合、および既定のライブラリファイルがスクリプトと同じディレクトリ内にない場合はライブラリパス指定つきの import 文もしくは #library による指定をおこなわなければならない。
ライブラリパスに指定するファイルは指定ファイルがない場合には拡張子として“.zip”を付して検索する。
ライブラリ化するスクリプトソースをライブラリファイルとして構成するには
1. パッケージとして指定するパスに対応するファイル名のファイルとしてこれを zip ファイルに格納する。
2. パッケージとして指定するパスごとにフォルダ分けしたファイルをフォルダパスつきで zip ファイルに格納する。

のどちらかをおこなう。
例：
import abc.xyz : usrlib;
としてアクセスされる場合、“abc.xyz.nlb”というファイルを usrlib.zip に格納するか、“abc/xyz.nlb”（abc ディレクトリ内の xyz.nlb というファイル）をディレクトリごと usrlib.zip ファイルに格納する。

注意：
import abc.xyz とした場合、abc.xyz.nlb でも abc.xyz.any.nlb でも対象となり、ライブラリファイル内で先に見つかったものを対象とするため、このような場合は import すべきパッケージパスについては曖昧であってはならない。

16.5 extern 文

extern 文は、ネットワークにある他のサーバ上のプログラムの実行結果、もしくはコンテンツをデータとして利用するためのステートメントである。
サーバプログラムを関数として使用する extern 関数宣言と、抽象クラスとして使用する extern クラス宣言、オブジェクトとして使用する extern object 宣言、さらにサーバプログラムの出力もしくはコンテンツを Buffer 型オブジェクトとして得る extern content 宣言がある。
ここでいうサーバとは、extern で指定する対象をいい、あるサーバが別のサーバのプログラムを利用する場合においても、利用する側をクライアント、利用される側をサーバという。
ネットワーク上の複数のコンピュータが、互いに相手のプログラムを呼び出すことによってデータ交換をおこなうような場合、その各々のコンピュータは相互にクライアントにもサーバにもなりう

る。
※現在利用できるネットワーク上のプログラムもしくはコンテンツは http プロトコルでアクセスされる WEB 上の CGI および静的コンテンツである。

16.5.1 extern 関数宣言

書式：　extern function 名前(引数並び)： url 文字列式;
　　　　extern method 名前(引数並び)： url 文字列式;

プログラム内にて<名前>で指定した関数を呼び出した場合、<url 文字列式>に指定するネットワーク上のサーバにあるＮＡＬプログラムを、関数とみなして呼び出しアクセスする。
呼び出されるＮＡＬプログラムが return 文によって値を返すことによって、呼び出した側からその値を利用できる。

例：

```
extern  function  search(name:string): "http://boo.foo.com/search.nal";
var     result:string = search("鈴木");
```

search(〝鈴木〞)の関数呼び出しを、http://boo.foo.com/search.nal で指定するネットワーク・アプリケーション呼び出しに置き換え、その実行結果が result に格納される。

http://boo.foo.com/ 内に配置する search.nal のプログラムソースを

```
return  name+"様";
```

とした場合、〝search.nal〞はクエリーとして〝name〞キーセットの値として〞鈴木〞を受け取るので name の内容が〞鈴木〞となる。その値に〞様〞を連結して return で返すと呼出し側でこれを受け取り、result には〞鈴木様〞が得られる。

呼出し元が結果として受け取るデータの型はネットワーク・アプリケーション側が返す値の型になる。すなわち上記例では文字列型であるが、search.nal が

```
return  name.length();
```

とした場合は返されるデータは int で 2 となる。ただし、この場合は呼び出し側で string を受取るように指定しているので型不整合となる。

注意：<引数並び>に指定する仮引数名および型は、extern 関数側で規定した名前および型でなければならない。ただし extern 関数側が variant で受取る仕様に構成するのであれば、呼び出し側の型指定は必要もなくまた extern 関数側で処理可能な値であれば自由な値を引き渡すことが出来る。

16.5.2 extern クラス宣言

書式：　extern class 名前[:LIB]:url 文字列式;

<名前>で指定したクラスが、<url 文字列式>にて指定するネットワーク上のサーバにおいて抽象クラスとして定義されていることを意味する。

“:LIB”を指定しない場合は、url 式で示すサーバ側プログラム記述全体を暗黙のクラスであるとみなし、そのプログラム中の第１階層メソッドをクラスメソッドとみなす。

“:LIB”を指定する場合は、url 式で示すサーバ側プログラム記述中に、<名前>で示すクラス宣言が明に記述されかつ、そのメソッドは public かつ static として定義されている必要がある。またこのクラスは抽象クラスとしてのみ利用され、利用側にはメソッド呼出しのみ許される。
サーバ側プログラムに複数のクラスを宣言（クラスライブラリ）し、そのクラスメソッドを選択利用する場合に使用する。

暗黙クラスの利用例を示す。
例：

```
extern  class  Base: "http://〜〜〜/base.nal";
class   Ext extends Base{
        method  foo()   return  base_foo();
        }

var     obj:Ext = new Ext();
writeln(obj.foo());                                     ①
writeln(obj.base_foo());                                ②
writeln(Ext.base_foo());                                ③
writeln(Base.base_foo());                               ④
```

対応するサーバ側プログラム base.nal は

```
method  base_foo()      return 100;
```

のように記述する。これは呼び出し側にとって、文法上

```
abstract class Base{
        static  method  base_foo()      return  100;
        }
```

と記述されていることと同じにみなされる。

上記①から④はいずれも、指定した url のサーバにて宣言されているスタティックメソッド base_foo()を呼び出し、その結果を表示する。
なお、指定したメソッドの実体が extern クラスにない場合は結果として null が返される。
また、var mptr:method = Base.base_foo; のように参照変数に代入してから mptr();のように間接呼び出しをおこなうようなことはできない。

次にクラスライブラリとしての使用例を示す。

サーバ側プログラム base.nal を

```
class base1{
        public static method base_foo() return 100;
        }
class base2{
        public static method base_foo() return  200;
        }
```

のように記述する。
対するクライアント側のプログラムは

```
extern  class  base1:LIB: "http:// 〜〜〜/base.nal";
extern  class  base2:LIB: "http:// 〜〜〜/base.nal";
writeln(base1.base_foo());                  100 と表示する
writeln(base2.base_foo());                  200 と表示する
```

のように記述する。

16.5.3 extern オブジェクト宣言

書式：　extern object 名前[:LIB]:url 文字列式;

<url 文字列式>にて指定するネットワーク上のサーバに、<名前>で指定したオブジェクトがあることを意味する。
“LIB”を指定しない場合は、url 式で示すサーバ側プログラム記述全体を暗黙のオブジェクトであるとみなす。extern で宣言した<名前>はこのプログラム内における識別としての意味しかもたな

い。
“LIB”を指定する場合は、url 式で示すサーバ側プログラム記述中に、<名前>で示すオブジェクト宣言かモジュール宣言が明に記述されかつ、そのメソッドは public として定義されている必要がある。利用側には指定したオブジェクト内のメソッド呼出しのみ許される。

extern オブジェクトの利用はプログラム内において、url 文字列式で指定したネットワーク上の NAL プログラムの実行結果を暗黙オブジェクトとみなし、その内部メソッドの呼び出しをおこなう。メソッドの実体が extern オブジェクト側にない場合は結果として null が返され、またサーバ側では undefined エラーが発生する。

例：

```
extern  object  base: "http:// ～～～/base.nal";
writeln(base.base_foo());
```

これはプログラムとしては

```
object base{
include "http:// ～～～/base.nal";
}
writeln(base.base_foo());
```

と同じように機能する。

サーバ側プログラム base.nal のプログラムを

```
private field:int sum = 0;
method  base_foo():int{
        return sum;
        }
for(var cnt=1; cnt<=100; cnt++) sum += cnt;
```

のように記述した場合、このプログラムは、クライアントにとっては

```
object base{
private field:int sum = 0;
method base_foo():int{
        return sum;
        }
for(var cnt=1; cnt<=100; cnt++) sum += cnt;
}
writeln(base.base_foo());
```

と記述されたと同じである。（但し、object base のコンストラクタを実行するのは自身ではなくサーバである）
実際のサーバ側の動作は

```
private field:int sum = 0;
method  base_foo():int{
        return sum;
        }
for(var cnt=1; cnt<=100; cnt++) sum += cnt;
⇒ここで base_foo()を呼び出し、結果をクライアントに返す
```

呼出しごとに、サーバ側プログラムがコンストラクタ部分（for(var cnt=0; cnt<=100; cnt++) sum += cnt;）を実行し、すべての処理が終わった時点でメソッド base_foo()を呼び出し、その結果をクライアント側に返す。

機能としては extern class 宣言と同じであるが、url 文字列式で示す外部 NAL プログラム全体をオブジェクトであるとみなす使い方であり、プログラミングの実態に近い概念である。
サーバ側プログラムは base_foo()メソッド呼出しが発生する以前に、そのプログラム全体を実行

し終わっており、その時点で変数 sum は実行結果を保持している。すなわち、メソッド呼出し以前に、そのプログラムをオブジェクトとみなした場合のコンストラクタが実行されたものとしてとらえることが出来る。
なお、var mptr=base.base_foo; のように参照変数に代入してから mptr();のように間接呼び出しをおこなうことはできない。

16.5.4 extern content 宣言

書式： extern content 名前(): url 文字列式 ;

<名前>で指定した関数を呼び出した場合、<url 文字列式>に指定する WEB 上の静的なコンテンツ（HTML ファイルや画像など）または CGI スクリプトの標準出力結果を取り出し、Buffer 型オブジェクトとして返す。エラーが発生した場合は null が返される。
例：

```
extern  content  Image(): "http:// ～～～ /image.jpg";
var     result:Buffer = Image();
```

結果として result には JPEG 画像のバイナリーデータが格納される。

16.5.5 url プロパティ

extern 宣言において、<url 文字列式>で与えた値は url プロパティとして参照ならびに設定できる。
例：

```
extern  function getname():"http:// ～～～ /getname.nal";
writeln(getname.url);                                    ①
getname.url = "http://www1.act.ne.jp/getname.nal";       ②
writeln(getname());
```

① は extern 関数宣言した getname()関数の url プロパティ値を得て表示する。
② は getname()関数の url プロパティ値を代入置換する。

<url 文字列式>には、絶対パス（http://で始まる書式）、ルートパス（/で始まる書式）および相対パス（このプログラム自身のパス基準）を指定できる。
例：

```
extern function x():"http://nal.act.ne.jp/lib/class.nal";     絶対パス
extern function x():"/lib/class.nal";                         ルートパス
extern function x():"class.nal";                              相対パス
```

※注意：url で示すホストは httpd 1.0 以上に対応している必要がある。

16.5.6 stat、response プロパティ

extern 宣言されたオブジェクトへのアクセス結果を保持する。
stat プロパティに整数値 0 以外の値が格納されている場合はエラーを意味する。
response プロパティにはサーバから返されたレスポンス・ヘッダ文字列が格納される。

16.5.7 throwable プロパティ[12]

extern 呼び出し前に true に設定すると、呼び出し実行時のエラー発生を例外として発行する。
extern 宣言時点で<URL>文字列値に続いて throwable を記述しない場合、この値は false であり、例に示すように明示的に true を指定しない限り例外発行をおこなわない。例外を発行しない場合であっても stat プロパティにはエラー値が格納されるため、stat プロパティの値によってエラーかどうかの判断をすることができる。

[12] Rel-20050120 にて追加された

例：

```
extern function ExtFunc(query:string):<URL>;
ExtFunc.throwable = true;              // エラー例外の発行を指示
try{
        var     result = ExtFunc("query strings");      // 例外発生の可能性あり
        }
catch(err:IOException){
        ①
        }
catch(err:RuntimeException){
        ②
        }
```

ExtFunc()の呼出しに際し、Socket 通信に関するエラーが検出された場合は①の catch 節に制御が移り、HTTP ステータスがエラーとなった場合は②の catch 節に制御が移る。

① の場合 err 値は Socket に関するエラーを示す IOException 例外となっている。
　err.getValue() のリターン値は
　"Socket:Open error"、"Socket:Write error"、"Socket:Read error"の何れかが返される。

② の場合 err 値は HTTP に関するエラーを示す RuntimeException 例外となっている。
　err.getValue() のリターン値は
　"HTTP Error:<URL> status=<HTTP status>"書式の文字列が返される。
　<HTTP status>の値については RFC を参照のこと。

throwable プロパティは、以下のように宣言時に<URL>文字列値に続いて throwable キーワードを付加することによっても true に設定できる。
例：

```
extern function ExtFunc(query:string):<URL> throwable;
```

16.5.8 redirect プロパティ[13]

extern 宣言時点で<URL>文字列値に続いて noredirect を記述しない場合この値は true であり、明示的に false を指定しない限り extern の対象とするＵＲＬのリダイレクトに対応する。
redirect プロパティ値に false を設定するとリダイレクトには対応しない。
例：

```
extern function ExtFunc(query:string):<URL> noredirect;
```

はりダイレクトに対応しない。
throwable と noredirect を共に指定する場合は、

```
extern function ExtFunc(query:string):<URL> throwable noredirect;
```

のように指定しなければならない。

16.5.9 action プロパティ

extern 関数などの呼出しに先立って、″VOID″,″HTTPDOC″,″CGI″,″LIB″文字列を代入することにより、以後の動作および機能として設定できる。また、文字列をコンマ（″,″）で連結し、″CGI,HTTPDOC″のようにも指定でき、この場合は各々指定した機能が論理和として設定される。但し、″VOID″はもっとも優先される。

■“VOID″を設定した場合、extern に対するアクセスは呼出し後直ちにリターンし、その応答を待ち合わせしない（非同期呼出）。″VOID″を含まない値に設定すると同期型呼出となる。
例：

[13] Rel-20050120 にて追加された

```
extern object netobj:"url";
netobj.action = "VOID";
netobj.method();                                ①
netobj.action = "";
netobj.method();                                ②
```

①では method()の返す値を待ち合わせしないが、②では結果のリターンを待ち合わせる。

■"HTTPDOC"を設定した場合、呼出しの結果として extern 側のプログラムの標準出力を以下に示すハッシュ・オブジェクトとして得る。

```
object httpdoc{
        field   HEAD:string;
        field   BODY:Buffer;
        }
```

HEAD にはサーバからのレスポンス・ヘッダ文字列を含む。
BODY にはサーバからのレスポンス・エンティティ・ボディをバイナリーデータとして保有する。

■"CGI"を設定した場合、呼出し対象は一般の CGI プログラムであるとみなし、呼出しに際してのクエリー文字列は CGI 規約に従う。すなわち関数として宣言した場合など引数にどのような型の値を指定してもすべてそれを文字列化した値に変換しキーセットとして構成してからリクエストをおこなう。したがってクエリーキーセットの値の型については呼び出された側での対応が求めらる。

■"LIB"を設定した場合、extern で示す＜名前＞がサーバ側のプログラムで明に宣言されていることを規定する。また、呼出しに際しては＜名前＞で示すクラス、メソッドが直接アクセスされる。

16.5.10 呼出し時の action オプション

呼出しに先立つ action プロパティへのアクション指定と同様な機能を、呼出しごとに"as"キーワードに続けて指定できる。ただし、アクションキーワードは文字列としてではなく識別名として指定し、さらにコンマで区切って複数指定できる。但し"LIB"は指定できない。
例：

```
extern class extClass:"http://extHost/docs.nal";
var     doc:httpdoc = extClass.extMethod() as HTTPDOC,CGI;
```

この場合、変数 doc には httpdoc 型のオブジェクト、すなわち呼び出した先のプログラムの標準出力が格納される。なお、as キーワードに続くアクション指定は呼出し単位で機能し、action プロパティの内容に影響を与えないし、また action プロパティに影響されない。

16.5.11 extern オブジェクトへの参照と型名

extern 文で宣言した関数およびクラスなどはグローバル・スコープを持った extern オブジェクトとして構築保持される。
したがって変数もしくはフィールドに対し、参照として割り当てることができ、この場合の型名は宣言名と同じとなる。
例：

```
extern function ExtFunc():"http://xxxxxxxx";
var     ext:ExtFunc = ExtFunc;
ext();
```

※ 注意：同名の extern オブジェクトはプログラム内に唯一であり、同名の extern 宣言をおこなうごとに先のオブジェクト内容を上書きする。

16.5.12 サーバプログラムの構成制御

extern 文によって呼び出される側の NAL プログラムには、extern の種別に応じて以下のグローバル変数があらかじめ定義される。
extern function で定義された場合、 _FUNCTION_ が定義される。
extern object もしくは extern class で定義されかつメソッド呼び出しがおこなわれた場合、 _METHOD_ が定義される、その値は各々文字列で、呼び出すメソッド名を保持する。
各々defined 演算子によって定義の有無を調べることはできるが、値を操作したり delete によって削除してはならない。

以下にプログラム例を示す。

```
#if     (defined(_METHOD_))
method  view(){
        return  new Date();
        }
#elif   (defined(_FUNCTION_))
return  new Date();
#else
%%現在時刻を Date 型のオブジェクトで返します%%
#endif
```

このプログラムは様々な呼出し形式に対応する。
一般的には、ドキュメントとして呼び出す場合はプログラムドキュメントを表示するように構成するとよい。

※注意：
extern object もしくは extern class として使用される場合のメソッドを

```
if  defined(_METHOD_){
method  view(){
        return  new Date();
        }
}
```

のように構成してはならない、これは_METHOD_が定義された場合には、プログラムの終了時に呼び出しもとから指定されたメソッド呼び出しがおこなわれるが、上記例では method view()は if ブロック内の Runtime メソッドであり if ブロック終了時に消滅しているため、呼び出した時点では undefined である。

16.6 その他のプログラム制御文

#if 項	項の評価結果が true または 0 以外の整数のとき、対を成す#elif または #else,#endif 直前までのステートメント群を実行し、false の場合はそのステートメント群を無視する
#elif 項	先行する#if もしくは#elif にて false になった場合に評価され、項の評価結果が true または 0 以外の整数のとき、次の#elif または#else,#endif 直前までのステートメント群を実行し、false の場合はそのステートメント群を無視する
#else	先行する#if または#elif で false または 0 となった場合に、対応する #endif 直前までのステートメント群を実行する
#endif	#if {#elif}[#else]#endif の組み合わせで使用し、プログラム制御をおこなう。（単独では何もしない）
#charset 識別	識別には「Shift_JIS」 もしくは 「EUC-JP」を記述し、スクリプトソース

	ファイルの文字コードを指定する。
!w0 !w1 !w2 !w3	ワーニングレベルを設定する。既定値はワーニングレベル０（!w0） 但し、http://localhost/としてアクセスされるスクリプトでの既定値はワーニングレベル１
!s+	ソーストレースをＯＮする
!s-	ソーストレースをＯＦＦする（既定値）
!e+	エラーログを〝nal.log〞というファイルに出力する
!e-	エラーログを標準出力デバイスに出力する（既定値）
!e++	!e+ およびエラー発生時、ErrorException 例外を発行する
!e--	!e- およびエラー発生時、Abort する（既定値）
!+	トレース機能をＯＮする
!-	トレース機能をＯＦＦする（既定値）
!?	トレース機能がＯＮの場合に限り、そのソース行数を表示して、続くブロックを実行する。トレース機能がＯＦＦの場合は、ソース行数も表示せず続くブロックを無視する。（!?以降のブロック全体を/**/で囲んだことに等しい）
!{ 式並び }	デバッグ情報の表示を目的に、{}内に記述された式の表記とその値を表示する。ワーニングレベル０では{}内の記述は無視される。

16.6.1 #if #elif #else #endif の構文制御

例：

```
const   VER = 20001213;
const   REL = 3.0;
#if !defined(VER)
        writeln(“オリジナルバージョンです”);         // 実行されない
#elif (VER >= 20000101)
        #if (VER < 20010101)
                writeln(“ミレニアムバージョンです”);  // 実行される
        #else
                writeln(“21 世紀バージョンです”);      // 実行されない
        #endif
        !+                                           // トレースモードＯＮ
        !? writeln(“デバッグ中です”);                 // 実行される
#endif
!?{                                                  // 実行される
        !{ VER , REL }
}
```

上記結果は

```
ミレニアムバージョンです
<行数:>デバッグ中です
<行数:>{ VER=20001213,REL=3 }
```

と表示される。

#if および#elif に続く項に式を記述する場合は必ず“（”と“）”で括った評価式形式とする必要がある。（if 文と同じ形式）
#if 等は行頭から始まる必要はない。
例：

```
#if !defined(DBG) writeln(“正規の処理”); #else writeln(“デバッグ中です”); #endif
```

16.6.2 #charset による文字コード系指定

通常は.nal ファイルのマルチバイトコードを自動判定するが、この自動判定が正しくない場合に #charset 制御文で指定することを薦める。
例：
#charset="Shift_JIS";
以後、マルチバイトコードはシフトＪＩＳとして扱われる。
#charset="EUC-JP";
以後、マルチバイトコードは EUC として扱われる。

#charset 制御文は動的に作用する。すなわちソース中の位置ではなく、解釈実行の流れに対して作用する。したがって include 文で文字コード系の異なるソースファイルを取り込む場合、その前後で指定する必要がある。また、文字コード系は WEB アプリケーションとしてして動作する場合のレスポンスヘッダ内の charset 指定にも連動する。したがって文字コード系の異なるスクリプトソースを混在させた場合には、<META>タグによる明示的な指定が必要になる場合もある。

16.6.3 デバッグ支援出力

ワーニングレベルが１以上の場合のみ、プログラムの途中経過などを表示したい用途に !{ 式 } 記述を使用できる。機能としては、式の表記とその値を組として表示する。

書式： !{ ＜式並び＞ } [;]

式並びとは単独の式、あるいは式と式を半角スペースかコンマ（,）かセミコロン（;）で区切ったもの。
式の前にコロン（:）を記述すると、＜式表記＞:＜式の値＞と表示し、コロンとイコール(:=)を記述すると、＜式表記＞=＜式の値＞と表示し、指定のない場合は＜式の値＞のみを表示する。
式並びをコンマで区切ると、式表記と式の値の組あるいは式の値を', 'で区切り、セミコロンで区切ると改行する。
!{ 式並び }; のように、最後にセミコロンをつけると、最後に改行を出力する。

例：

```
var     boo = 100;
var     foo = "ABC";
!{ boo , :foo ; :=boo+foo };
```

結果は「100, foo:"ABC"＜改行＞boo+foo="100ABC"＜改行＞」と表示する。
なお、式の前に HTML タグを記述してもよいが、式の後にタグを記述する場合はコンマもしくはセミコロンで区切らないと比較演算子と誤認するので注意する。
例： !{ <B> :boo ;</B> }

17 例外制御

プログラム実行中に発生する例外を捉えて的確な処理をおこなうことができる。

17.1 基本構文

書式：
try ブロック catch(例外引数) ブロック {catch(例外引数)ブロック} [finally ブロック]

try に続くブロック内に、例外が発生する可能性のあるステートメントを記述する。
ステートメント実行の過程で例外が発生した場合は、ステートメントの実行を中断し制御は続く、例外値に対応する catch ブロックに移行する。

catch 文は例外値を受け取る変数名（例外引数）を宣言し、続くブロック内に例外引数に渡された例外値を処理するステートメントおよび例外発生に対応するステートメントを記述する。

finally 文は省略可能であり、記述した場合は例外が発生したかどうかにかかわらず、ブロック内のステートメントを実行する。すなわち catch ブロック内でさらに例外を発生させる場合においても実行する。

17.2 throw 文

書式：throw　式

＜式＞には例外値として任意の型の値を記述でき、これを例外値という。
try ブロック内にて直接、もしくは関数、メソッド呼び出しにおいて throw を実行した場合、throw 文に続くステートメントは実行されずに、catch ブロックに制御が移る。
try ブロック外で throw 文を実行した場合ならびに例外をキャッチする catch ブロックがない場合は、例外発生を通知してプログラムを終了する（例外終了）。この場合、エラーレベルが１以上であれば例外の発生したステートメント位置をレポートするがエラーレベルが０の場合は単に終了するのみである。
return に続いて throw 文を記述した場合、および return ＜式＞での式評価中に例外が発行された場合は一旦呼び出しもとに戻った後で例外を発行する。
したがって、try ブロック内で return throw した場合および return ＜式＞の式評価中に例外が発行された場合においては、対応する catch ブロックにではなく、戻り先の try-catch 節で例外処理がなされる。

17.3 try 文

書式：try ブロック

try に続くブロック内に例外を発生する可能性のあるステートメントを記述する。
記述された複数のステートメントのうちあるステートメントで例外が発生した場合、続くステートメントは実行されず、制御は直ちに catch ブロックに移行する。
どのステートメントにおいても例外が発生しなかった場合は、catch ブロックはスキップし、finally ブロックが記述されていれば、そのブロックを実行し、さらに try-catch 構文に続くステートメントの実行を継続する。
try ブロック内に return 文がある場合、例外が発生しなければそのステートメントは有効であるが、実際にリターンするタイミングは finally ブロックの終了後である。また、finally ブロックにおいても return 文を実行した場合は、try ブロック内の return 文は無視され finally ブロック内の return 文が実行される。

try ブロック内にさらに try-catch 構文を記述する場合、内部の try-catch 構文内で例外が発生した場合には、その catch ブロック内にて throw をおこなわない限り外側の catch ブロックに制御が移ることはない。そんため外側の catch ブロックに制御を移す必要がある場合は、内側の

catch ブロック内で throw 文を実行して例外を発生する必要がある。

17.4 catch 文

書式：catch(例外引数) ブロック

catch に続くブロック内に、try ブロック内にて例外が発生した場合に処理すべきステートメントを記述する。
＜例外引数＞は＜名前＞［:＜型名＞］であり、型名を省略した場合はシステム既定例外を除くすべての例外値を捕捉する。逆にシステム既定例外を捕捉する場合は、variant 型もしくはその型を明示しなければならない。variant 型を指定した場合にはすべてのシステム既定例外を catch する。
throw null あるいは throw のみを catch する場合は catch(){ ... }のように例外引数を記述しない。
型名を指定した場合は、例外値が型に対応する場合のみこの catch ブロックを実行し、対応しない場合は後続の catch ブロックを順次検査するが、対応する catch ブロックがない場合は finally ブロックがあればこれを実行した後、上位の try-catch 節に例外を通知する。
例外値に配列を用いる場合、これを catch で受け取るには、＜名前＞[]:＜型＞もしくは＜名前＞:＜型＞[]とする。
なお、異なる型の要素が混在する配列もしくはハッシュを例外値として catch するには例外引数に型指定のない catch でなければならない。したがって、例外引数型を指定する catch ブロックを先に記述し、最後に例外引数の型を指定しない catch ブロックを記述する。
catch ブロックに制御が移った場合、例外引数には例外値（throw 値）が格納されており、この値もしくは型を判断することによって例外の種類を特定するなどの判定に使用する。

catch ブロック内でさらに例外を発生する（throw 文の実行など）場合、続く finally ブロックは実行するが try-catch 構文に続くステートメントは実行されないで上位の catch ブロックに制御が移る。
結果的にどの try-catch 節によっても捕捉されない例外はアボートする。
ただしワーニングレベル１以上の場合、アボート前にエラー通知をおこなう。

17.5 finally 文

書式：finally　ブロック

finally に続くブロック内に try-catch 構文の処理を実行した場合に必ず実行すべきステートメントを記述する。主に、catch ブロック内にて break、return などの制御文を記述する場合に、その制御に従った処理がおこなわれる前に処理すべきステートメントを記述する目的で使用する。
finally ブロック内で return 文を実行した場合は、try ブロックあるいは catch ブロック内の return 値は無効となり finally 内の return 文が実行される。

17.6 システム既定例外

以下の例外がシステムで既定されている。

例外名	例外発生理由
AbortException	ABORT 定数のアクセス
ErrorException	続行不能なエラー発生（下記解説参照）
ExitException	exit ステートメント実行
IOException	IO に関わるエラーの発生（プリセットクラス・リファレンス参照）
RuntimeException	正常な実行が困難と思われるエラー発生
TimeoutException	watchdog()監視でのタイムアウト発生（下記解説参照）

これらの例外はその発生時にシステムが Exception クラスを拡張してその例外インスタンスを生成するため、プログラムにおいて throw の例外値として使用することはできない。

ExitException 以外のこれら例外を catch しないと、例外発生をエラー通知する。
また、catch するには必ず型指定（例：catch(e:IOException){}）しなければならず、型指定のない catch 節で捕捉することはできない。

17.6.1 エラー例外

プログラム制御文 !e++もしくは !e-+を指定することにより、続行不能なエラー（syntax error など）が発生した際に ErrorException 例外を発生させ、プログラムでこれを捕捉することができる。
ErrtorException 例外を catch するには catch 節の例外引数の型を ErrorException として明示しなければならない。（例： catch(err:ErrorException){}）
既定および !e+- もしくは !e-- 制御文を指定した場合は、エラー発生時に ErrorException 例外を発行することなくアボートする。

17.6.2 watchdog()関数によるタイムアウト監視

watchdog()組み込み関数の引数に０以外の数（整数もしくは実数で監視秒数を意味する）を指定すると、引数に指定した経過秒以内に新たに watchdog()によって監視秒数を再設定するか watchdog(0)によって監視を停止しない場合に TimeoutException 例外が発生する。
なお、CGI エンジンとして利用しかつ URI の host 名に localhost を指定した場合には既定として watchdog(30)が設定される。
例：

```
watchdog(10);                              // １０秒の監視を設定
try{
        ・・・・
        }
catch(exept_other){ // ※注意：引数の型指定がないとシステム既定例外を捕捉しない
        writeln("例外が発生しました:",except);
        }
catch(except:TimeoutException){ // try ブロック処理に１０秒以上要した場合
        writeln(except);
        }
watchdog(0);                               // 監視を停止
```

上記例では try ブロック内での処理に１０秒以上を要した場合に catch(except:TimeoutException)ブロックに制御が移るが、この際の例外引数 except には、TimeoutException オブジェクトが渡され、その Exception フィールドに Wrapper 文字列 ”WatchDog TimeOver (rr.rrSec)” が格納される。rr.rr 部は実数表記にて watchdog()設定からの経過秒数を表す。
この文字列を取り出すには、上記例の場合 except.getValue()をおこなう。

watchdog()によるタイムアウト監視は 100 ステートメント実行ごとにおこなわれる。

17.7 ユーザ定義例外

組込み Wrapper クラスである Exception クラスを拡張して独自例外クラスを定義できる。
ただしシステム既定例外と同名の例外をユーザ定義してはならない。

例：

```
class  MyException extends Exception;
```

また、独自にフィールドを定義したり、メソッドを定義する場合は

```
class  MyException extends Exception{フィールド定義｜メソッド定義}
```

として定義する。

利用例：

```
try{
        throw   new MyException("メッセージ");
} catch(except:MyException){
        ※except.getValue()は throw 時の引数"メッセージ"を取り出す。
}
```

18 式と演算子

18.1 引数と arguments 配列

クラスのコンストラクタ引数ならびにメソッドおよび関数呼び出しにおける実引数は、各々 arguments という既定識別名の配列にも引数の指定順に格納される。
この配列はクラス定義時の引数宣言、メソッドなどの引数宣言に依存しない。
コンストラクタおよびメソッド・関数において明示的な引数の代わりに arguments 配列を使用して実引数を扱うことができる。また、仮引数以上の実引数が指定された場合も arguments 配列には格納され、arguments 配列の長さを得ることによって実引数の個数を調べることもできる。
このことを利用して可変引数を扱うことができる。
例：

```
class   分数{
        field   分子:int = arguments[0];;
        field   分母:int = 1;

        if(arguments.length() > 1)      分母 = arguments[1];
        }

var     整数 = new 分数(100);
var     実数 = new 分数(2,3);
```

※ 注意：インスタンス生成時の arguments 配列はコンストラクタ実行時のみ使用でき、メンバーとしてインスタンスに留まることはない。したがって、インスタンス・メソッド内から this.arguments としてコンストラクタ引数をアクセスすることはできない。
またメソッド・関数のリターン値として arguments 配列自身を渡すことはできない。

18.2 instance 項

instance 項はクラス・インスタンスもしくはレコード・インスタンスへの参照値をいい、クラスあるいはレコード・コンストラクタ呼出しにより生成された値である。

■クラス・インスタンス
例：

```
class   Car(name:string=""){ ... }
var     myCar:Car = new Car("Atlas");
```

■レコード・インスタンス
例：

```
record  Person{
        field   name:string = "";
        field   address:string = "";
        }

var     tanaka:Person = new Person{
                name="田中"; address="東京都";
                };
```

例の Person のように、レコード定義において既定された初期値をフィールドに設定しており、インスタンス生成時に明示的なフィールドの初期化をおこなわない場合は

```
var     kimura:Person = new Person{};
var     suzuki:Person = new Person;
```

のように、フィールド初期化を省略する。

18.3 object 項

object キーワードに続いてオブジェクトの実体を定義することによって、定義場所にオブジェクトを生成する。

書式１：object ［型名］ : ｛フィールド定義｜メソッド定義｝
書式２：object : ［型名］ ｛フィールド定義｜メソッド定義｝
書式３：object ［型名］ ｛ NDSN 記述 ｝

例：

```
var     obj:分数 = object 分数 : {
                field   分子 : int = 1;
                field   分母 : int = 3;
                method  toReal() : real {
                        return 分子/(分母*1.0);
                        }
                };
```

型名を省略した object 項を受け取る側（変数、引数、リターン値など）で型を明示する場合には、広義のオブジェクトを示す「object」でなければならない。
例：

```
method  fraction(nume:int,denom:int=1):object{
        return  object : {
                field   分子 : int = nume;
                field   分母 : int = denom;
                method  実数化() : real {
                        return 分子/(分母*1.0);
                        }
                };
        }

var     fract:object = fraction(1,3);
writeln( fract. 実数化() );
```

書式３の NDSN 記述は以下の規定にしたがう。
NDSN 記述≡　　メンバー記述 ｛, メンバー記述｝
メンバー記述≡　フィールド記述｜メソッド記述
フィールド記述≡フィールド名 : 式
メソッド記述≡　メソッド名 : method （仮引数並び）ブロック
例：

```
var     fract:分数 = object 分数 {
        分子 : 1,
        実数化 : method(){
                return 分子/(分母*1.0);
                },
        分母 : 4
        };

writeln( fract.実数化() );                          // 0.25 を表示する
```

18.4 method 参照項

method 参照項は、その場所で method 型のオブジェクトを生成し、その参照値を値とする。

書式： method(仮引数並び) ブロック

例：

```
var     sum:method = method(limit){
                var     s:int = 0;
                for(var id = 0; id < limit; id++) s += id;
                return  s;
                };
writeln(sum(10));                       // 55 を表示する
```

上記は、変数 sum に method 参照項の値、すなわち動的メソッドへの参照値が格納される。
従って、sum()は method 参照項で生成されたメソッドの呼出しとなる。

18.5 method 直接項

method 直接項は、その項が参照（アクセス）された時点で、ブロック内に記述されたステートメント並びを実行し、return 値を項の値とする。

書式： method { ブロック }

例：

```
var     value:string = "value は" + (
        method{
                var     sum:int = 0;
                for(var id = 0; id <= 10; id++) sum += id;
                return  sum;
                } + 100         // method 項の値 + 100
        ) + "です";
writeln(value);                 // value は 155 です　を表示する
```

18.6 current 項

メソッド内で記述される単独の ．（ピリオド）は常に、メソッドが定義されているクラスのインスタンスを指すオブジェクト参照項である。
例：

```
class   Person{
        field   name;
        method  mySelf(){
                return  .;              // Person インスタンスを返す
                }
        method  myName(){
                return  .name;          // Peroson インスタンス内の name を返す
                }
        static  method  myCls(){
                return  .;              // Person クラスを返す
                }

        .name = "AnyOne";               // Person インスタンス内の name フィールドに代入
        }
```

```
class   Vip extends Person{
        field   name = "Vip";           // Person クラスの name フィールドを override
        }

var vip = new Vip();
writeln(vip.mySelf());                  // Person インスタンスを表示
writeln(vip.mySelf().name);             // "AnyOne"を表示
writeln(vip.myName());                  // "AnyOne"を表示
writeln(vip.myCls());                   // Person クラスを表示
```

18.7 this 項

コンストラクタ部およびインスタンス・メソッド内で記述される this は常に、実体インスタンスを指すオブジェクト参照項である。
例：

```
class   Person{
        field name:string;
        method setName(nm:string){
                this.name = nm;         // ※下記参照
                }
        method setMyName(nm:string){
                name = nm;              // Person 内の name メンバーに設定する
                }
        }
```

上記の例で class Person が他のクラスの基本クラスとして利用される場合、インスタンスメソッド内で使用される this は実体インスタンスすなわち拡張クラスのインスタンスを指す。つまり上記クラスを拡張したクラス内に name メンバーが override されている場合、this.name は拡張クラスの name メンバーが対象となる。
したがって、

```
class   Member extends Person{
        field name:string;              // Person の name フィールドを override
        }
var     member = new Member();
member.setName("太郎");
member.setMyName("花子");
```

とした場合、member.name は "太郎"、member.Person.name は "花子"となる。

with 文内で参照される this 項は、with で指定したオブジェクト自身を指す。
例：

```
var     ary = {"Jon","Tom","Bill"};
with(ary){
        for(var c = 0; c < this.length; c++){
                writeln(this[c]);
                }
        }
```

18.8 thispart 識別子と entity 識別子

識別子 thispart はコンストラクタ部あるいはインスタンス・メソッド内において記述され、その

記述されているクラスのインスタンスを指す。すなわち current 項と同じである。
単独の識別子 entity は常に実体インスタンスを指し、すなわち this 項と同じである。
また thispart.entity のように記述した場合も同様に実体インスタンスを指す。

例：

```
class   Person{
        field name:string;
        method setName(nm:string){
                entity.name = nm;        // ※下記参照
                }
        method setMyName(nm:string){
                thispart.name = nm;      //Person 内の name メンバーに設定する
                }
        }
```

new Person()インスタンスにおいては、Person クラス内に記述された entity も thispart も共に Person クラスインスタンスを指すが、Person クラスが他のクラスの基本クラスとして利用される場合、entity は最終的に拡張されたクラスのインスタンスを指す。したがって拡張したクラス内に name メンバーが override されている場合、entity.name は実体クラスの name メンバーが対象となる。
例：

```
class   Member extends Person{
        field name:string;               // Person の name フィールドを override
        }

var     member = new Member();
member.setName("太郎");
member.setMyName("花子");
```

とした場合、member.name は "太郎"、member.super.name は "花子"となる。

18.9 super 項

識別子 super は、クラス・メソッド内においては基本クラス（親クラス）を指す。クラスが派生クラスでない場合は組み込みクラス Object を指す。
コンストラクタおよびインスタンス・メソッド内においては基本インスタンスを指す。クラスが派生クラスでない場合は組込みクラス Object のインスタンスを指す。
例：

```
class   Person{
        method  mySuper(){
                return  super;           // Object インスタンスを返す
                }
        static  method  superCls(){
                return  super;           // Object クラスを返す
                }
        }

class   Member extends Person{
        method  mySuper(){
                return  super;           // Person インスタンスを返す
                }
        static  method  superCls(){
                return  super;           // Person クラスを返す
                }
```

```
        }
```

18.10 thisblock 項

thisblock は、その記述されるブロック自身を参照する object 型のオブジェクト参照項である。
例：

```
method  consume(env:object){
        env.x += env.y;          // env 環境の変数をアクセス
        env.y -= 100;
        env.view();              // env 環境の view()メソッドを実行する
        }

method  product(){
        method  view()  writeln("x=%d,y=%d".import(x,y));
        var     x = 10;
        var     y = 200;
        consume(thisblock);      // 自身のブロックを consume に引き渡す
        }

product();                       // x=210,y=100 と表示する
```

例：

```
method  boo(limit:int){
        var     sigma:int = 0;
        repeat(limit) sigma += limit--;
        return  thisblock;
        }
writeln(boo(10).sigma);          // boo の実行オブジェクト内の sigma 変数値 55 を表示
```

上記は、メソッドの return 値として thisblock を返した場合、その参照が続いている限りメソッドの実行オブジェクトが消滅しないことを示す例である。
※Ver 4 以前のキーワード thishere も互換のため利用できる。

18.11 with 項

式内で使用される with は、続くオブジェクト参照値を項の値とし、このオブジェクトを対象にブロックを実行する。

書式： with （オブジェクト参照値）ブロック
例：

```
var     date = with(new Date()){
                setDate(1);              // 当月 1 日とする
                };
```

ブロック内に return 文がないか this 値を返した場合の with 項の値は対象オブジェクト参照値であり、他の値を返した場合にはその値が with 項の値となる。

例：

```
var     date = with( new Date() ){
                if(getDate() == 1)       return  null;
                setHours(0);
                setMinutes(0);
```

```
                return  this;
                };
writeln(date);                  // 結果は、毎月１日の場合は null を表示する
```

例：

```
var     dataset:int[] = with( new int[]{} ){    // 整数要素に規定した空の配列を用意
        this += 100;            // 配列要素[0]として 100 を追加
        this[2] = 200;          // 配列要素[2]に 200 を格納
        };
writeln(dataset);               // {100,,200} と表示する
```

18.12 root 項

識別子 root は、プログラム中において常にプログラムの最上位ブロックを指す。
最上位ブロックに宣言された変数ならびに関数はグローバル変数、グローバル関数としてすべてのプログラム階層からスコープされる。
例：

```
var     x = 100;                        // グローバル変数
function  lower(){              // グローバル関数
        var     x = 200;                // ローカル変数
        writeln({x , root.x});
        }
lower();                                // function lower()呼出し
```

上記を実行した場合、“{200,100}”と出力される。
すなわち、root.x は関数内に宣言された変数 x ではなく、グローバル変数 x を指している。

18.13 文字列に対する演算

文字列に対しては加減演算子と比較演算子が適用できる。

18.13.1 結合、削除、リピート演算

文字列を対象とする演算子は以下の機能を果たす。

演算子	機能	例	
+	A+B は文字列 A と文字列 B を結合し、B が文字列以外なら文字列化の後に結合する	"ABC" + "XYZ"	"ABCXYZ"となる
		"ABC" + 100	"ABC100"となる
		"ABC" + 0.12	"ABC0.12"となる
		"ABC" + true	"ABCtrue"となる
-	A-B は文字列 A に文字列 B を含む場合のみ、文字列 A から文字列 B を抜き取った文字列となる。文字列中 A に文字列 B を完全に含まない場合は文字列 A の値を保持する。	"ABCD" - "BC"	″AD″となる。
		"ABCD" - "BCDE"	"ABCD"のまま変わらない。
*	A*n は n>0 の整数の場合のみ文字列 A を n 回繰り返して結合する。 Ver4 以降では n<1 の場合″″となる	"ABC" * 2	"ABC"を 2 回繰返して結合した文字列、"ABCABC"となる。

文字列を値として持つ変数に対する +=、-=、*= 演算においても同様に機能する。
※結合およびリピート演算子適用後の文字列長が 2^{31} バイトを超えないこと。

18.13.2 比較演算

文字列を対象とする比較は、以下に示す文字列との比較演算のみ可能である。
比較の判定は先頭からの構成文字各々について code()メソッドで返される値を大小比較した結果

である。

演算子	記述例	解説
==	str == ref	"ABC" == "ABC" のみ true
!=	str != ref	"ABC" != "ABC" のみ false
<	str < ref	"ABC" < "ABCD" は true
>	str > ref	"ABC" > "ABCD" は false
<=	str <= ref	"ABC" <= "ABCD" は true
>=	str >= ref	"ABC" >= "ABCD" は false

18.13.3 文字列定数への式の埋め込み

式を %{ と }% で括って以下のように記述して、値を文字列中に埋め込むこともできる。
例：

```
var     name = "田中";
writeln("私の名前は%{ name }%です");
```

これは

```
writeln("私の名前は" + name + "です");
```

と同じ結果となる。
式の値が null の場合は""として扱う。

"%{"を埋め込みキーとしてではなく文字列として解釈させたい場合は"\%{"のように記述する。
例：

```
writeln("文字列埋め込み記号'\%{'を文字列として表現するには\\\%{とします");
```

18.14 代入式の扱い

代入式は、その全体をひとつの項としてあつかう。
例：
s=10*v=1+2　は　s=10*(v=1+2)　と見なされる。したがって s は 30、v は 3 となる。
"+="、"-="、"*="、"/="、"<<="、">>="　も同様である。
例：
return s+=10; は変数 s に 10 を加え、その結果の s の値をリターン値として返す。

18.15 論理式の評価

論理積（"&&"）は、左辺値が真でなければ、右辺式をすべて評価しない。
論理和（"||"）は、左辺値が真のとき右辺式をすべて評価しない。
したがって、複数の論理積および論理和を使用する式においてはカッコで括って２項間演算に集約すべきであるし、評価されることを期待した副作用を伴う式を記述すべきでない。
例：

```
s = (false && true || true);
```

s は false であるし、||の前後の true は評価されない。
上記を s = ((false && true) || true);　のように記述すると、s は true となる。
s = (true && false || true); とした場合は s は true となり、かつすべての項が評価される。

18.16 比較演算の特例

比較演算子、"=="、"!="はオブジェクト同士もしくは配列同士を比較する場合、両方の参照が同じ（同一のオブジェクト実体を参照している）場合のみ true となる。
従って、オブジェクトのフィールド内容が同じであったとしても、その実体が別であれば false となる点に注意が必要である。
例：

```
class   Car(name){
```

```
        field   carname = name;
        }
var     c1 = new Car(“Atlus”);
var     c2 = new Car(“Atlus”);
var     c3 = c1;
write(c1==c2);          // “false”を表示する
write(c1==c3);          // “true”を表示する
```

18.17 論理演算の特例

論理演算において、ワーニングレベル１以下の場合は論理値に加え、null は偽（false）、数値は 0（整数および実数）のみ偽、文字列値は空文字列のみ偽として扱う。またオブジェクト参照値はすべて真（true）として扱う。
但しワーニングレベル２では論理値と数値以外では警告アボートし、ワーニングレベル３以上の場合は論理値以外では警告アボートする。
例：

```
!w1                                     // ワーニングレベルを１にする
const   a = 1, b = "ABC", c = {1,2,3};
const   x = 0, y = "";
if(a)   writeln("a isn't 0");
if(b)   writeln("b isn't blank");
if(c)   writeln("c is object");
if(!x)  writeln("x is 0");
if(!y)  writeln("y is blank");
```

上記はいずれも if 文に続く writeln() を実行する。

18.18 文字列評価演算子

書式：　#文字列項

“#”演算子を文字列項に適用すると、文字列中に記述された NAL 文法に従った式を評価する。
例：

```
var     source = "100+200";
var     val1 = #source;
var     val2 = #"source";
```

結果は、val1 に 300 が、val2 には”100+200”が格納される。
式の評価は、文法的に有効な範囲で打ち切られることに注意すること。
文字列以外の項に“#”演算子を適用した場合は一切の作用を及ぼさない。

変数にメソッド名もしくは関数名を格納し、その関数を呼び出す場合にも＃演算子は有用である。
例：

```
function foo(a,b)       return  a+b;
var     fc = "foo";
var     s = (#fc)(10,20);
```

※ #＜文字列＞ は eval()と同様、ステートメント記述であっても解釈実行できるが、その結果はあくまで値である。したがって、#"boo" = 200 は無効であるが #"boo = 200" は変数 boo に 200 を代入し、この項としての値も 200 である。

18.19 swap 演算子

書式：　変数 swap 変数名

＜変数＞の内容と＜変数名＞で示す変数の内容を交換し、式全体の値は交換後の＜変数＞の値となる。＜変数＞には一意の変数名、配列要素、ハッシュ要素名、インスタンス・フィールド名を指定できる。
例：

```
var     a=100, b="ABC";
writeln({a, b, a swap b, a, b});
```

結果は“{100,”ABC”,”ABC”,”ABC”,100}”と表示される。

18.20 as 演算子（変数エイリアス）

書式： 変数 as 識別名

＜変数＞の別名として＜識別名＞を割当てる。＜識別名＞を変数エイリアスという。
以後、＜識別名＞への演算は＜変数＞への演算と同じ効果をもつ。
＜変数＞ as ＜識別名＞ = ＜値＞ とすれば、エイリアス化とともに変数にも値を代入する。
＜変数＞ as ＜識別名＞++ とすれば、エイリアス化とともに変数を+1 する。
＜変数＞は var 変数、配列要素、インスタンス・フィールドなど＜値＞を保持するものであれば制限はなく、また単独のステートメントだけではなく、if 文内など広範囲に適用できる。
識別名のスコープは、ローカル変数と同様、ブロック内に限定される。
例：

```
var     ary={ 1, { 2, {3,4} } };
ary[1][1][1] as elm += 100;                   ①
writeln("ary="+ary," : elm="+elm);            ②
elm += 100;                                   ③
writeln("ary="+ary);                          ④
```

① は、ary[1][1][1] を elm という名前に割付け、ary[1][1][1]に 100 を加算したことになる。
② の結果は“ary={1,{2,{3,104}}} : elm=104”と表示される。
③ は、ary[1][1][1] += 100 と同じ効果をもたらす。
④ の結果は“ary={1,{2,{3,204}}}”と表示される。

18.21 in 演算子

書式： 項 in 配列項

＜項＞が＜配列項＞の要素として含まれていれば一致する要素のうち、より小さい要素番号+1（要素順序数[14]という）、含まれていなければ 0。＜配列項＞に配列以外を指定した場合は 0 となる。
例：

```
if (value in {null , ""} != 0)  writeln("value は NULL か空文字列です");

上記は、整数値を論理判定として利用できるため
if (value in {null , ""})  writeln("value は NULL か空文字列です");
とできる。（あまり推奨はしないが）
```

例：

```
var     names:string[] = { "ABC" , "XYZ" };
var     detect:int = value in names;
if (detect != 0) writeln("value の値は",names[detect-1]);
```

in 演算子と同様な機能は配列メソッド indexOf()を利用することもできる。
例：

[14] 要素順序数とは配列の要素番号+1 の整数値をいう。配列 ary の要素順序 m の要素とは ary[m-1]である。

```
var     names:string[] = { "ABC" , "XYZ" };
var     detect:int = names.indexOf( value );
if (detect != -1) writeln("value の値は",names[detect]);
```

※注意：in 演算子による値と indexOf()メソッドでは意味する結果の値が異なる。

18.22 inrange 演算子

書式：　項 inrange 上下限配列

＜上下限配列＞とは、要素 0 を下限値とし、要素 1 を上限値とする配列

＜項＞が＜下限値＞以上＜上限値＞以下の場合 true を値とする。
＜項＞、＜下限値＞、＜上限値＞の型は数（整数、実数）または文字列で、かつ同じ型でなければならない。

例：

```
if((value>=0 && value<=100)||(value>=200 && value<=300)){ ... }
```

のように、value の値域を判定する場合に煩雑となりやすい記述を、

```
if(value inrange{0,100} || value inrange{200,300}){ ... }
```

もしくは

```
const    range = {{0,100},{200,300}};
if(value inrange range[0] || value inrange range[1]){ ... }
```

と記述できる。

例：

```
var     check = "ABC";
var     result = check inrange {"A","Z"};
```

上記例では result には true が格納される。

※注意：上下限配列に３要素以上の配列を指定しても第１要素を下限、第２要素を上限とし、その他の要素には関知しない。

18.23 instance 検査演算子

オブジェクトとクラスの関連を調べる instance 検査演算子として instanceof,isinstance,isextend がある。
書式：　項　＜instance 検査演算子＞　クラス参照項

instance 検査演算子は、左辺項と右辺のクラス参照項の関係を検査し、その結果は論理値となる。

例：

```
class Animal{・・・}          基本クラス
class Dog:Animal{・・・}      Animal の派生クラス
var     obj = new Dog();      Dog クラスのインスタンスを生成
```

において
(obj intanceof Animal) も (obj instanceof Dog) も共に true となる。
(obj isinstance Animal) は false、(obj isinstance Dog) は true となる。
(obj isextend Animal) は true、(obj isextend Dog) は false となる。

18.23.1 instanceof 演算子

書式：　項 instanceof クラス参照項

instanceof 演算子は、<項>が、<クラス参照項>が参照するクラスもしくはその派生クラスのインスタンスの場合に true となる。
例：

```
class   Sup{...}
class   Derive extends Sup{...}
class   another_cls{...}
var     x = new Derive();

write(x instanceof Sup);                // true を表示する
write(x instanceof Derive);             // true を表示する
write(x instanceof another_cls);                // false を表示する
```

18.23.2 isinstance 演算子

書式：　項 isinstance クラス参照項

isinstance 演算子は、<項>が、<クラス参照項>が参照するクラスのインスタンスの場合に true となる。
例：

```
class   Sup{...}
class   Derive extends Sup{...}
class   another_cls{...}
var     x = new Derive();

write(x isinstance Sup);                // false を表示する
write(x isinstance Derive);             // true を表示する
write(x isinstance another_cls);                // false を表示する
```

18.23.3 isextend 演算子

書式：　項 isextend クラス参照項

isextend 演算子は、<項>が、<クラス参照項>が参照するクラスの派生クラスのインスタンスの場合に true となる。
例：

```
class   Sup{...}
class   Derive extends Sup{...}
class   another_cls{...}
var     x = new Derive();

write(x isextend Sup);                  // true を表示する
write(x isextend Derive);               // false を表示する
write(x isextend another_cls);          // false を表示する
```

18.24 代替値演算子

18.24.1 isvalue 演算子

書式：　項 isvalue(比較演算子　式) 代替式

isvalue 演算子は、<項>の値と<式>を<比較演算子>に従って比較し、結果が true なら<代替式>を評価して値として採用し、false なら<代替式>を評価せずに<項>をそのまま採用する。
例：

```
var     a = new Date().getHours() isvalue( >= 12) 11;
```

この例は

```
var     a = new Date().getHours();
if (a >= 12) a = 11;
```

と同じ結果であるが、変数 a への代入は１回だけである点が異なる。

※より複雑な判定に基づく代替値を扱う場合は method 項の利用を勧める。

18.24.2 isnull 演算子

書式： 項 isnull 代替式

isnull 演算子は、<項>が NULL であった場合、<代替式>の結果を値として採用する。
<項>が NULL でなければ<項>が採用され、<代替式>は評価しない。
例：

```
var     a = null;
var     b = a isnull 100+200;
var     c = b isnull 500;
```

結果は b が 300、c が 300 となる。

複数の isnull 演算子を使用する式においてはカッコで括って２項間演算に集約すべきであるし、評価されることを期待した副作用を伴う式を記述すべきでない。
例：
writeln(a isnull 100+b isnull 400) は
writeln(a isnull (100+(b isnull 400))) と等価であり、
writeln((a isnull 100)+(b isnull 400)) とは等価ではないことに注意。

18.24.3 isblank 演算子

書式： 項 isblank 代替式

isblank 演算子は、<項>が文字列でないか空文字列であった場合、<代替式>の結果を値として採用する。
<項>が文字列でかつ空文字列でなければ<項>が採用され、<代替式>は評価しない。
例：

```
var     a = "";
var     b = a isblank "ABC";
var     c = b isblank "300";
```

結果は b が″ABC″、c が″ABC″となる。

複数の isblank 演算子を使用する式においてはカッコで括って２項間演算に集約すべきであるし、評価されることを期待した副作用を伴う式を記述すべきでない。

18.24.4 代替値演算子と例外

代替演算子に続く代替式に throw 文を記述した場合、代替条件成立に伴い例外を発行する。
例：

```
function 割り算(被除数,除数){
    return  被除数 /( 除数 isvalue( == 0) throw "0 で割りました");
}
```

関数の引数、除数に 0 を指定した場合に例外を発行する。

例：

```
var     x = null;
try{
        var     y = x isnull throw "NULL";
        writeln("ABC");
        }
catch(err){
        writeln(err);
        }
```

変数 y に x を代入するがその際 x の値が null であるため､続く throw〝NULL″ 文が評価される。したがって例外が発生し、catch 節に制御が移る。

18.25 defined 演算子

書式１： defined(識別名)
書式２： defined(#文字列式)

書式１は<識別名>について、定義の有無を論理値で返す関数と同等な働きをする。
識別名が定義されている場合は true、定義されていない場合は false となる。
識別名には、引数名、変数名、関数名、クラス名、enum 要素名を指定する。
例：

```
var     abc;
writeln(defined(abc),":",defined(xyz));
```

abc は変数として定義されているが xyz は定義されていないので､結果は 〝true:false″となる。

書式２では<文字列式>の値に対応する識別名について、定義の有無を論理値で返す関数と同等な働きをする。
文字列式の値を識別名とする変数等が定義されていれば true となる。
例：

```
var     abc = "xyz";
var     xyz = "ABC";
writeln(defined(#abc),":",defined(#xyz));
```

変数 abc の値は"xyz"であり、これを xyz という識別名とみなすと変数 xyz は定義されている。一方、変数 xyz の値は"ABC"であり、これを ABC という識別名にみなすと ABC は定義されていないので、結果は 〝true:false″となる。

18.26 typeof 演算子

書式： typeof(式)

typeof 演算子は<式>の値について、その型を示す文字列値を返す関数と同等な働きをする。
typeof 演算子を適用した結果を基に式の型検査をおこなうには Type クラス内に用意してある組込み定数と比較すること。
例：

```
var     a = 1.2;
if(typeof(a) == Type.real) writeln("Not a integer");
if(typeof(a) in {Type.int,Type.real}) writeln("Number");
```

値 value に対し typeof(value)を適用した場合の結果に対応する Type クラス定数

value の型	typeof()値に対応する組込み定数	値（参考）
整数型	Type.int	"INT"

実数型	Type.real	"REAL"
文字列型	Type.string	"STRING"
論理型	Type.bool	"BOOL"
配列	Type.array	"ARRAY"
オブジェクト	Type.object	"OBJECT"
クラス	Type.class	"CLASS"
レコード	Type.record	"RECORD"
メソッド	Type.method	"METHOD"
関数	Type.function	"FUNCTION"
ＮＵＬＬ	Type.null	"NULL"

18.27 その他の演算子

演算子	例	解説	適応型
+	A + B	加算	数値、文字列
-	A - B	減算	数値、文字列
*	A * B	乗算	数値、文字列
/	A / B	除算	数値
=	A = B	代入（objectは参照代入）	
:=	A := B	代入（objectは所有代入）	
+=	A += B	A = A + B　に同じ	数値、文字列
-=	A -= B	A = A - B　に同じ	数値、文字列
*=	A *= B	A = A * B　に同じ	数値、文字列
/=	A /= B	A = A / B　に同じ	数値
==	A == B	同じなら真	
!=	A != B	異なれば真	
>	A > B	大きければ真	数値、文字列
<	A < B	小さければ真	数値、文字列
>=	A >= B	大きいか同じなら真	数値、文字列
<=	A <= B	小さいか同じなら真	数値、文字列
&&	A && B	論理積	論理値
¦¦	A ¦¦ B	論理和	論理値
!	!A	NOT	論理値

演算子	例	解説	適応型
~	~A	補数	整数
&	A & B	ビットＡＮＤ	整数
¦	A ¦ B	ビットＯＲ	整数
^	A ^ B	ビットＸＯＲ	整数
&=	A &= B	A = A & B に同じ	整数
¦=	A ¦= B	A = A ¦ B に同じ	整数
^=	A ^= B	A = A ^ B に同じ	整数
+	+A	正符号	数値
-	-A	負符号	数値
++	++A　A++	インクリメント	整数
--	--A　A --	デクリメント	整数
>>	A >> B	右シフト	整数
<<	A << B	左シフト	整数
>>>	A >>> B	符号なし右シフト	整数
>>=	A >>= B	A = A >> B　に同じ	整数
<<=	A <<= B	A = A << B　に同じ	整数
>>>=	A >>>= B	A = A >>> B　に同じ	整数
%	A % B	剰余	整数
%=	A %= B	A = A % B に同じ	整数

18.28 演算子の優先順位

演算子の優先順位を示す。上段欄にあるものが上位優先順位である。

<table>
<tr><td>() [] {} .</td></tr>
<tr><td># defined new with</td></tr>
<tr><td>in inrange swap instance 検査演算子 代替値演算子</td></tr>
<tr><td>! ~ - ++ -- typeof</td></tr>
<tr><td>* / %</td></tr>
<tr><td>+ -</td></tr>
<tr><td><< >> >>></td></tr>
<tr><td>< <= > >=</td></tr>
<tr><td>== !=</td></tr>
<tr><td>&</td></tr>
<tr><td>^</td></tr>
<tr><td>|</td></tr>
<tr><td>&&</td></tr>
<tr><td>||</td></tr>
<tr><td>?:</td></tr>
<tr><td>= := += -= *= /= %= <<= >>= >>>= &= |= ^=</td></tr>
<tr><td>,</td></tr>
</table>

19 ダイナミック・プログラミング

変数もしくは定数にＮＡＬステートメントを記述し､これを実行する場合は eval()関数を使用する。
eval()関数は、引数として与えられた文字列をＮＡＬプログラムとして実行し、その結果を返す機能を持つ。

プログラムの流れに応じて、ステートメントもしくはブロック、メソッドなどをダイナミックに生成し、これを実行する場合に使用する。

本機能はインタープリタにのみ実装される。

19.1 式の実行

文字列として式を指定した場合、eval()関数はこれを解釈し、その結果を返す。
例：

```
var     r=10;
var     面積 = “r*r*Math.PI”;
var     円周 = “2*r*Math.PI”;
writeln(“円周=”,eval(円周),”面積=”,eval(面積));
```

結果としてはそれぞれ、100*3.14 ,20*3.14 と同じ結果となる。

プログラム文字列に含まれる変数､関数などは eval()を呼び出す位置でのスコープ解決ができるものであれば制限はない｡

19.2 ステートメントの実行

文字列としてステートメントを指定した場合、それを実行し、最後に評価された式の値を結果として返す。
例：

```
var     sw=false;
writeln(eval("if(sw) 100; else 200;"));
```

結果は 200 が表示される。

上記例は、以下のような記述のほうが理解しやすい。

```
writeln(eval("if(sw) return 100; else return 200;"));
```

もっと複雑な例では、

```
writeln(eval("var s=0;for(var I=0;I<100;I++) s+=I;"));
```

のように複合ステートメントを記述して実行することもできる。
この場合は最終的な変数 s の内容を結果として返す。

eval()の引数に与える文字列は結果としてＮＡＬプログラムであればよく､以下の例のようにプログラムの流れに従った構成を記述することもできる。
例：

```
var     s=10,e=100;
writeln(eval("var s=0;for(var I=" + s + ";I<" + e + ";I++) s+=I;"));
```

これは数と文字列の結合規則から､

```
writeln(eval("var s=0;for(var I=10;I<100;I++) s+=I;"));
```

と同じ結果となる。

19.3 ダイナミック・メソッドの実行

ＮＡＬプログラムが全体でオブジェクトを構成するように､eval()メソッドに評価させる文字列も完結したＮＡＬプログラムとして記述できる。
その上で、内包するメソッドを呼び出すことができる。

例：

```
var     p = "method a(x) return x*x; method b(x) return x+x;";
var     q = "var s=100; method b(x) return x+s";
var     v = eval(p);                                ①
writeln(v.a(100));                                  ②
writeln(v.b(100));                                  ③
writeln(eval(q).b(200));                            ④
writeln(eval(q));                                   ⑤
```

① は変数 p に与えられたメソッド定義を含むオブジェクト（subbody オブジェクトと呼ぶ）を結果として返す。すなわち変数 v は、もはや文字列ではなくオブジェクトそのものを参照するオブジェクト参照変数となっている。もちろん、このオブジェクトはオブジェクトの寿命規則に従って、どこからも参照されなくなった時点で消滅する。
② は変数 v が参照するオブジェクトのメンバーとしてのメソッド a()を呼び出している。メソッド a()の引数 x として 100 を与えているため、結果は当然 10000 が得られる。
③ は同様にメンバーメソッド b()を呼び出している。
④ は同様な仕組みではあるが、メソッド b()内で参照する変数 s がオブジェクト内に存在する例である。このようにオブジェクトはその内部にメンバーとして値もしくはオブジェクトを内包しており、それをそのまま利用できる。この仕組みは静的なクラス変数を利用する場合と同様な仕組みである。
⑤ は結果として subbody オブジェクトを返してくる。①も同様であるが、プログラム文字列中に値を生成するステートメントもしくは式が存在しない点が重要である。逆に、**ダイナミック・メソッドを利用する場合、プログラム文字列中に値を生成するステートメントもしくは式を記述してはならない。**

19.4 ダイナミック・クラスの利用

ダイナミック・メソッドと同じ概念によって、ダイナミック・クラスを利用することができる。
例：

```
var     p = "class x(y){field s=100; method b(x) return x+s+y;}";
var     v = eval(p);
writeln(new (v.x)(10).b(100));
```

上記で v.x はクラス x を示していることは理解いただけるはずである。

記述上の注意点としては、writeln(new v.x(10).b(100));としてはいけないことである。
このように記述した場合、ＮＡＬは((new v).x(10)).b(100)と解釈するため、シンタックスエラーとなる。すなわち v はクラスではなく、subbody オブジェクトであるからである。

20 エラーレベルとデバッグ

!w＜数値＞によってエラーレベルを設定できるが、WEB アプリケーションとして動作する場合、URL 指定が localhost の場合は既定としてエラーレベルが１に設定される。
エラーレベルが０以外の場合、型判定をはじめ実行時エラーならびにプログラマの見落としによる値の整合性および値による制御の流れを、より厳密にレポートする。
プログラムの動作が意図したものでない場合、積極的に型判定をおこなうことでデバッグの助けとなる。

エラーレベル０の場合の型判定については例え明示的になされていたとしても、本来が実行時型に基づく処理であるため、構文上の型規定を除いてはエラー報告しない。また、未定義の変数についても自動生成となる。
プログラミング上特に注意すべきは引数の順序ならびに引数の数の不一致による誤動作であるが、引数の数については既定値を指定してリカバリーするなどの対策も重要である。

WEB アプリケーションの開発においてはできるだけ明確な型判定をおこない、localhost 環境におけるエラーレベル既定値を利用して、より正確なプログラミングを心がけられたい。
但し、小規模かつプロトタイプとして構成する場合などは型指定をおこなわずに作成することがあってもよく、その場合にはより迅速なプログラミングも可能となる。

プログラミングの品質と開発性に関わるこれらの問題は、プログラマーのスキルならびにポリシーにしたがって決定されるべきである。

21 既定識別子とメソッド

21.1 識別子

root	最上位ブロックを指す
this	このステートメントの記述されるブロックを基準として、それが属す実体オブジェクトを指す
entity	実体インスタンスを指す
. thispart	このステートメントの記述されるメソッドが定義されているクラスのインスタンスを指す
super	基本インスタンスを指す
thisblock	このステートメントの記述されるブロックを指す

21.2 ブロックのフィールド識別子とメソッド

thisblock.root	このブロックの上位ブロックを指す
thisblock.caller	このメソッドを呼び出したブロックを指す
thisblock.getName()	ブロックの名称を返す
thisblock.member()	ブロックに属する変数、メソッドなどの名称を１次元文字列配列として返す
thisblock.hereInfo	このステートメントが記述される″ファイル名：行番号″
thisblock.callerInfo	このメソッドを呼び出した″ファイル名：行番号″

21.3 インスタンスのフィールド識別子とメソッド

<インスタンス>.entity	インスタンスの実体インスタンスを指す
<インスタンス>.super	インスタンスの基本インスタンスを指す
<インスタンス>.class	インスタンスのクラスを指す
<インスタンス>.getMemeber()	インスタンス内のフィールドおよびメソッドの名称を１次元文字列配列として返す
<インスタンス>.getName()	インスタンスの名称を文字列として返す
<インスタンス>.toString()	インスタンスを文字列化した結果を返す
<インスタンス>.duplicate()	インスタンスを複製した結果を返す
<インスタンス> .deleteObject()	インスタンスを強制的に削除する （オブジェクトの消滅に関する節を参照）

22 組込み関数

以下にあらかじめ組み込まれている関数を示す。
これらの関数と同名の関数およびクラス変数等をユーザ定義した場合はユーザ定義が優先する。
ユーザ定義によって組み込み関数が隠蔽された場合は、組み込みクラス Util のクラスメソッドとしても利用できるため、Util.関数名(引数)として実行する。
例：
writeln(“ABC”) は Util.writeln(“ABC”) としても同じ。

インターフェースおよび解説	
deleteObject(obj:object):bool	obj に指定したオブジェクト（プリミティブ以外）を強制削除する。obj がプリミティブの場合は false を返す。
escape(str:string):string	JavaScript の escape()と同じ変換をおこなう
unescape(str:string):string	JavaScript の unescape()と同じ変換をおこなう
toIntValue(value:variant):int	value で指定するを値を整数化した結果を返す。 実数値を指定した場合は整数部を返す。 論理値を指定した場合は true なら 1 を false なら 0 を返す。 文字列を指定した場合はこれを数として解釈した後に整数化した値を返す。※廃止予定
toRealValue(value:variant):real	value で指定するを値を実数化した結果を返す。 論理値を指定した場合は true なら実数の 1.0 を false なら実数の 0 を返す。 文字列を指定した場合はこれを数として解釈した後に実数化した値を返す。※廃止予定
toString(value:variant, format:string=null):string	value を文字列化した結果を返す。 string クラスの import()メソッドに規定するフォーマット指定文字列を format に指定した場合は、そのフォーマットに従って整形する。ただし、値とフォーマットが整合しない場合の動作は保障しない。※廃止予定
isBool(value:variant):bool	value が論理値なら true を返す
isString(value:variant):bool	value が文字列なら true を返す
isNumber(value:variant):bool	value が整数か実数なら true を返す
isInt(value:variant):bool	value が整数なら true を返す
isReal(value:variant):bool	value が実数なら true を返す
isArray(value:variant):bool	value が配列なら true を返す
isObject(value:variant):bool	value がオブジェクトなら true を返す
parseInt(str:string):int	str の整数表記とみなせる範囲までを整数化した結果を返す。
parseReal(str:string):real parseFloat(str:string):real	str の実数表記とみなせる範囲までを実数化した結果を返す。
isNaN(value:variant):bool	value が数値（整数または実数）でない場合 true を返す。
onWEB():bool	WEB アプリケーションとして実行中は true を返す
write(value,,,) writeln(value,,,) print(value,,,) println(value,,,)	value 列を文字列化し、標準出力に出力する。 write()と print()、writeln()と println()は同じ機能を持つ別名関数。 writeln(),println()は最後に改行コードを付加する
sleep(secs:number)	secs 秒間実行を停止する（精度：1mSec）
getenv(id:string):string	文字列 id で示す環境変数を返す。

system(cmd:string, import:bool=false):Buffer	文字列 cmd で示すコマンドおよび引数をシェルに引き渡す 例：system(“ls -l”); import に true を指定した場合、cmd の標準出力を Buffer オブジェクトとして返す。（スタンドアロンのみ） 例：var buf=Util.system(“ls -l”,true); ※USER_ID()関数によって正規の許諾情報が指定された場合（特権モード）にのみ機能する。
watchdog(sec:number=null) :real	sec で指定する秒数の watch dog タイマーを設定し、先の設定からの経過秒を得る。watch dog タイマーを設定すると、新たな watchdog を指定するか watchdog(0)で停止するまでにタイムアウトとなれば TimeoutException 例外を発行する。(※例外の項を参照) sec を指定しない場合はプログラムの開始時点もしくは watchdog(数値)実行時点からの経過秒を得る。sec に 0 を指定すると watch dog を停止する。
tryNest():int	try-catch 構文の内部にある場合、ネスト数を返す。構文外にある場合は 0 を返す。
USER_ID(id:string):string	id に許諾情報を通知し、特権モードに移行する。 許諾情報はライセンス単位に発行される文字列値である。 特権モードに移行できた場合は許諾情報と同じ値を返し、許諾情報が間違っている場合は null を返す。

23 組込みメソッド

変数もしくは値に対しては値の型に応じた、あらかじめ組み込まれているメソッドを利用できる。
以下に、型ごとに利用可能なメソッドの抜粋と概要を示す。
その他のメソッドならびに詳細は、「プリセットクラス・リファレンス」を参照のこと。

23.1 整数、実数

toIntValue()	実数の場合、整数化した値を返す（少数部切り捨て）
toRealValue()	整数の場合、実数化した値を返す
toChar()	実数の場合は整数化し、下位 16 ビットにつき、その値に相当する文字を文字列として返す。 8 ビットを超える場合は、第 1 バイトに上位 8 ビットを、第 2 バイトに下位 8 ビットを格納する。例えば 0x889F.toChar()は“亜”となる。
isEven()	整数かつ偶数の場合のみ true を返す。
toString()	文字列化した結果を返す。

23.2 論理値

toIntValue()	true は 1、false は 0 とした整数返す
toString()	true は”true”、false は”false”とした文字列を返す

23.3 文字列

文字列はオブジェクトではないため、その操作メソッドはもとの文字列には影響を与えない。
以下に例を示す。

```
var     str = "abcdefg";
var     res = str.toUpperCase();
```

の結果 res は”ABCDEFG”となるが、str は依然”abcdefg”のままである。
なお、マルチバイト文字についてはシフトＪＩＳを標準とする。

substring(start,end)	start 文字目（インデックス[15]）から end 文字目までの部分文字列を返す。
extract(lead,trail)	lead と trail 文字列パターンに挟まれる部分文字列を返す。
includes(key)	key 文字列を含んでいれば true を返す。
indexOf(key)	key 文字列を含む場合、その位置を先頭からのインデックス値で返す。含まない場合は-1 を返す。
matchOf(key)	key 文字列中の何れかの文字を含む場合、その位置を先頭からのインデックス値で返す。含まない場合は-1 を返す。
length()	文字数を返す。
trim(str)	先頭および末尾に、str に含むいずれかの文字が連続する場合、これを取り除いた文字列を返す。
replace(str,to)	str 文字列パターンを to 文字列パターンにすべて置き換えた文字列を返す。
toUpperCase()	文字列中の半角英字を大文字化した文字列を返す。
toLowerCase()	文字列中の半角英字を小文字化した文字列を返す。
split(term)	term 区切り文字列で分割した文字列の配列を返す。
charAt(index)	index 文字目の文字を取り出す。

[15] 半角全角各々表記単位に１文字とカウントする、先頭文字を０とする文字位置

23.4 オブジェクト

getMember()	内包するメンバー名を文字列配列として返す。
length()	内包する配列の要素数を返す。
toString()	オブジェクトをを文字列化した結果を返す。

24 組込み定数・変数・クラス

24.1 ABORT 定数

ABORT という定数は、これをアクセスした場所で AbortException 例外を発生する。
ABORT はステートメントのように単独で記述しても、また式中に記述しても構わない。
例：

```
method div(x:int, y:int = ABORT){ ・・・ }
```

div(100) のように引数が不足した場合は ABORT 定数が引数リカバリー（既定値）として採用され、このタイミングで AbortException 例外が発生する。

プログラムミスを検出する目的で､スタブ内に ABORT 定数を利用すると問題の発生したステートメントを検出することができる。
例：

```
//var    value:int;              // debug 目的で以下のスタブ構成に変更
var      value: proxy{           // スタブを実現するためプロクシ・オブジェクト化
         field   value:int;                        // 実際の値を保持
         method  getValue():int  return  value;   // get
         method  setValue(val:int){                // set
                 if(val < 0)     return  ABORT;    // 負値を代入されたら Abort 例外を通知
                 value = val;
                 }
         };

value = 100;
value = -1;      // ←この場所で AbortException 例外が発生する
```

上記は、変数 value に値を書き込むステートメントがプログラム内の任意の位置に複数あるものとして、変数に負の値を代入したステートメント位置を検出する例である。
ポイントは例外通知する際に return ABORT とすることである。
return 値の式中で発生した例外は呼び出し元に委譲されるため、例外発生位置は ABORT を記述した位置ではなく、それを含むメソッドなどを呼び出した位置となる。
したがって、 value = -1 というステートメント位置が ABORT 例外発生位置となり、問題のステートメントを特定することができる。
また、先の例のように引数既定値に ABORT を使う場合も、実引数を渡す位置、すなわち method div()を呼び出しているステートメント位置で例外が発生する。

25 ＷＥＢアプリケーション対応

サーバサイドＮＡＬでは、ＷＥＢアプリケーション・エンジンとして利用する場合の利便性を考慮して、レスポンスヘッダ送出とクエリー解析を自動化および体系化することにより、より容易にＷＥＢアプリケーションを構築する仕組みが組み込まれている。

25.1 クライアントへのレスポンスヘッダ出力

サーバサイドＮＡＬでは Head という既定 Wrapper オブジェクトをもち、ドキュメント出力に際して WEB クライアントへのレスポンスヘッダ出力を自動的におこなう。

既定値では

```
Status: 200 OK
Content-Type: text/html; charset=Shift_JIS[16]
Content-Language: ja
Pragma: no-cache
Cache-control: no-cache
X-Server-ID: NAL
```

という文字列を、ドキュメント送出に先立って自動出力する。

一般にＨＴＭＬドキュメントを出力する場合はこの自動出力を利用すればよいが、例えば画像バイナリコードをスクリプトのレスポンスとして返すような場合などでは、自動出力のままではクライアント側で誤動作する。
この場合は、Head.putValue()によってふさわしいレスポンスヘッダを用意する必要がある。

25.2 クライアント・リクエスト解析

Query という既定 Wrapper オブジェクトに、WEB クライアントから POST あるいは GET メソッドによって送り出されたクエリー文字列を保持する。

クエリー文字列は以下の書式で示す文字列となっている。

```
＜識別名＞=＜値文字列＞ {&＜識別名＞=＜値文字列＞}
```

＜識別名＞=＜値文字列＞の組をキーセットといい、&によって複数のキーセットが連結される。
＜識別名＞および＜値文字列＞は通常のＷＥＢ環境においてはＵＲＬエンコードされている。
また、キーセットは場合によっては＜識別名＞だけキーのみの場合もある。

ＮＡＬにおいては以下に示すように、クエリーは自動的に解析される。
クライアントが GET xxxx.nal?arg1=value1&arg2=value2 というリクエストを行った場合、クエリー文字列の内容は“arg1=value1&arg2=value2”となる。
このとき、xxxx.nal プログラム実行に先立って上記クエリー文字列をＵＲＬデコードし、かつ

```
const arg1:string="value1";
const arg2:string="value2";
```

という記述がプログラムコード以前になされているように準備する。
したがって、xxxx.nal プログラムではクエリーとして送られたキーセットを上記文字列定数として扱ってよい。

一方、 GET xxxx.nal?arg1=value1&arg1=value2 というリクエスト、すなわち同じ arg1 という識別名に対して別の値、value1、value2 を指定する場合（HTML では複数項目選択タグ <SELECT name='arg1' multiple></SELECT>が該当する）は、ＮＡＬの文法上の規則から最後の arg1=value2 を採用する。
このようなケースでは自動解析機能を使用しないで Query 文字列をもとに処理しなければならない。

[16] EUC エンコード環境では"EUC-JP"となる

手順として以下の方法を推奨する。
例：

```
var arrayQuery[]:string = {};
var divideQuery[]:string = Query.getValue().split("&"); // Query 文字列を"&"で分割
foreach(keyset in divideQuery){
    var keysetAry[]:string = keyset.split("=");         // キーセット文字列を"="で分割
    keysetAry[0] = URI.decode(keysetAry[0]);            // キーを URL デコード
    keysetAry[1] = URI.decode(keysetAry[1]);            // 値を URL デコード
    arrayQuery.append(keysetAry);                       // キーセットを格納
    }
```

ここで writeln(arrayQuery)を実行すれば {{"arg1","value1"},{"arg1","value2"}}と表示される。すなわち、arrayQuery[0][1]の値は"value1"となっている。

26 規約

26.1 識別名

変数名をはじめ、すべての識別名は半角英字もしくは半角カナや漢字など２バイト系文字ならびに半角記号“ _ ”（アンダースコア）で先導しなければならない。また、２文字目以後は半角記号（スペースおよび制御文字含む）以外のすべての文字を使用できる。

例：a abc _abc 変数

26.2 コメント

“ // ”から改行文字までを行コメントとし、“ /* ”から“ */ ”までをブロックコメントとする。

26.3 予約語

変数名、メソッド名などに使用してはならない語

abstract	arguments				
break					
case	catch	class	const	continue	create
define	delete	do			
eachof	else	enum	entity	eval	exit
expat	extern				
false	field	final	finally	for	foreach
forindex	function				
if	import	include	intercept		
lock					
method	module				
new	null				
object	oncreate	ondelete	override		
private	protected	proxy	public		
record	repeat	return	root		
select	static	super	switch		
this	thisblock	thispart	throw	true	try
typeof					
unlock					
var	virtual				
while	with				

変数名、メソッド名への使用を避けるべき語

action	array	as			
bool	Boolean				
caller	callerInfo				
default	defined	derived	double		
entry	extends				
hereInfo					
in	includes	inrange	instanceof	int	integer
isblank	isextend	isinstance	isntblank	isntnull	isnull

isvalue					
length					
number					
parent					
readonly	real	redirect	response		
stat	string	swap			
throwable					
url					
value	variant	void			

26.4 データ表記規約

NDSN(Nal Data Structure Notation)として、以下に規定する。

≪データ≫::=　≪識別名≫:≪値≫

データは、識別名と値をコロンで結合したもの。

≪値≫::=　≪数値≫|≪文字列≫|≪論理値≫|≪式≫|
{ ≪値≫,≪値≫ } |
{ ≪データ≫,≪データ≫ } |
method{ ≪プログラム記述≫・≪return 式≫} |
method(引数並び){ ≪プログラム記述≫・≪return 式≫}

値は、数値、文字列、論理値、あるいはこれらを演算子で結合した式、
０個以上の値をコンマで区切って並べ、{}で括ったもの（配列）、
０個以上のデータをコンマで区切って並べ、{}で括ったもの（ハッシュ）、
method{}内のプログラム記述にて実行された≪return 式≫ステートメントの式の値。
method(引数並び){}で定義されたメソッドオブジェクト。