



ソースコードの構造と値

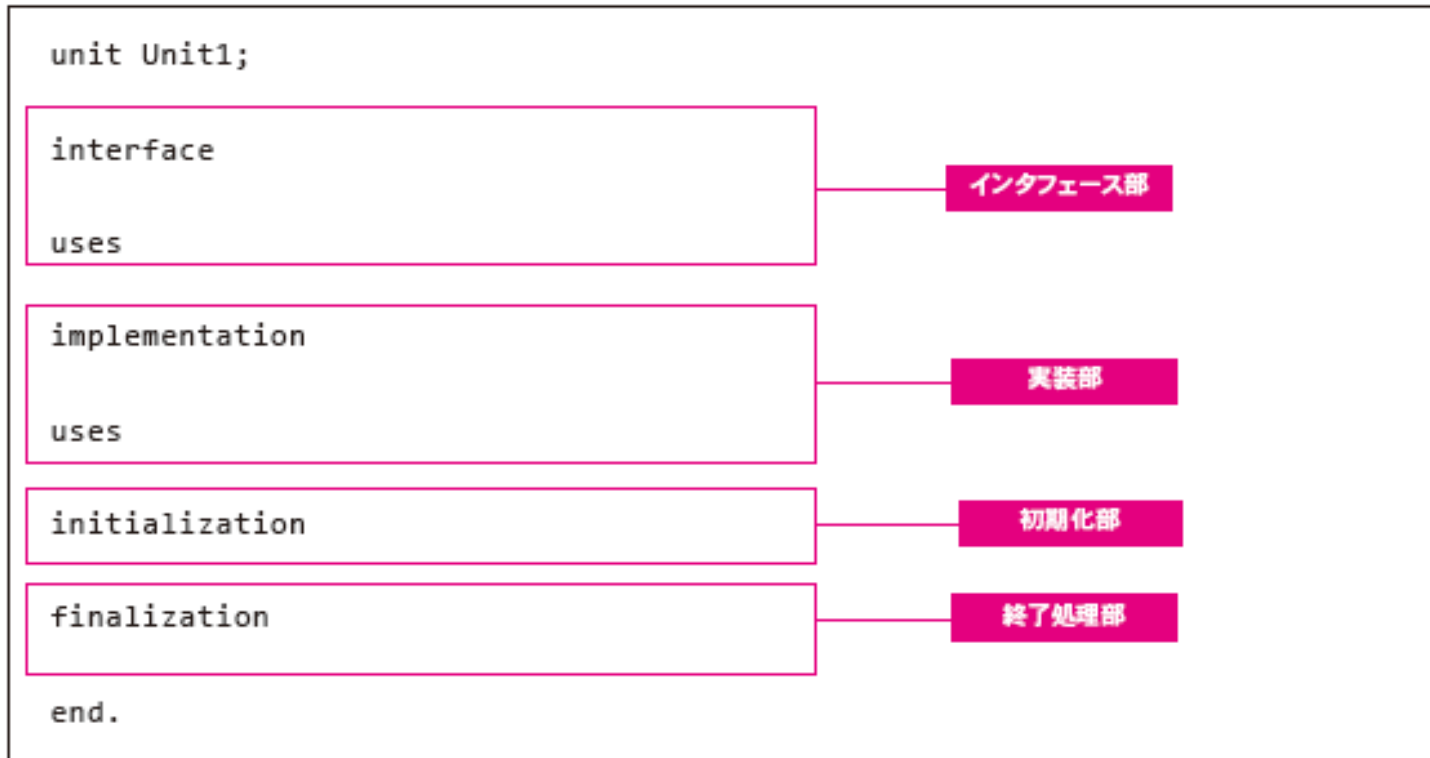
■ Delphiのソースコードを構成するファイルの種類

- プロジェクトファイル(.dpr)
プロジェクトのメインとなるファイル
通常はDelphiが内容を自動処理
- ユニットファイル(.pas)
実際に処理を記述するファイル
主にここにソースコードを記述
- フォームファイル(.dfm)
画面デザインなどの情報が記述されるファイル
通常はDelphiが内容を自動処理

ユニットファイルの構造

■ ユニットファイルの構造

ユニットは大きく 4 つに分けられます。



ユニットファイルの構造

- ユニットファイルはunitヘッダで始まる
- unit の横はファイル名となり、IDEのエディタ上で直接書き換えてはいけない
- ユニット名は、別ディレクトリにあっても同じプロジェクト内で重複することはできない
プロジェクトファイル名とも同様

ユニットファイルの構造

インタフェース部

- interface以降、implementationの前までがインタフェース部
型、定数、変数、手続きおよび関数を宣言
- ここで宣言した内容は別のユニットからも参照可能
C言語であれば、ヘッダファイルに相当
- インタフェース部には手続きや関数のルーチンの宣言だけを含む
実際の手続きや関数のソースコードはimplementation部に記述

次に、インタフェース部内のuses節が続く

インタフェース部で定義する内容に関連するユニット名を羅列
別ユニットで定義されている型やクラス、手続きや関数を使用する
場合、それが定義されているユニットをカンマ区切りで並べていく

ユニットファイルの構造

実装部

- 実際に動作する手続きや関数の中身のコードは、ここから下の部分に書く
- implementationからinitializationまでの間
(initializationがない場合はユニットの末尾まで) が実装部

次に、実装部内のuses節が続く

実装部で使用する内容に関連するユニット名を列記
インタフェース部のuses節と同様に、別ユニットで定義されている型やクラス、手続きや関数を使用する場合、それが定義されているユニットをカンマ区切りで並べていく

実装部のみの定義でよいusesのユニットは、**こちらに定義する**方がよい

ユニットファイルの構造

実装部－循環参照（１）

インタフェース部は他ユニットからも参照できる内容であるため、以下の記述例のような問題が発生する可能性がある
この例では、Unit1がUnit2を参照し、Unit2がUnit1を参照しており、コンパイル時にエラーが発生する

```
unit Unit1;  
  
interface  
uses Unit2;  
...  
  
end.
```

```
unit Unit2;  
  
interface  
uses Unit1;  
...  
  
end.
```

Unit1のインタフェース部を見てUnit2を参照しに行き、Unit2のインタフェース部でUnit1を参照しているため、堂々巡りとなってしまう
これをユニットの循環参照という

ユニットファイルの構造

実装部－循環参照（２）

どちらかの参照をimplementation部に移動すればコンパイルエラーを回避できる

```
unit Unit1;  
  
interface  
uses Unit2;  
...  
...  
...  
end.
```

```
unit Unit2;  
  
interface  
...  
  
implementation  
uses Unit1;  
...  
end.
```


ユニットファイルの構造

実装部

- 実装部のuses節の次に、インタフェース部で宣言された手続きや関数を定義する
- そのユニット内でのみ使用する手続きや関数、定数、変数、クラスや型を定義することも可能
- 実装部でのみ定義したものは、他のユニットからは参照できないが、このユニット内でしか処理されないものであると判断できるので、後々ソースをメンテナンスする場合にも分かりやすくなる

ユニットファイルの構造

初期化部

- initialization以降、finalizationの前までが初期化部
(finalizationがない場合はユニット末尾まで)
- プログラムの起動時に実行させたい内容を記述する
たとえば、プログラムを動かすにあたって必要なデータをセットしておきたい時などに、初期化部でデータをセット可能
- initialization部はオプションなので、基本的には無くても良い

プログラム起動段階での処理が必要となった場合に、初期化部で対応できるということだけ覚えておくとよい

ユニットファイルの構造

終了処理部

- finalizationからユニットの末尾までの間が終了処理部
- 終了処理部は、初期化部がある場合のみ定義が可能
- 初期化部で確保したリソースを解放するときに使う
- finalization部はオプションのため、基本的には無くてもなくても良い
initialization部でプログラム終了時に解放しなければならないリ
ソースを確保している場合のみ使用する

変数と定数

- プログラムで値を扱うには、変数または定数を使用する
- 変数や定数は値を入れておく入れ物のようなもの
- 数学の場合の式とは異なり、以下のような書き方ができる
 $A := A + 1;$
- 変数はプログラムの実行時に値を変更できる
- 定数はプログラムの実行中は一定の値のまま
定数に対して値を変更しようとするエラーとなる

データ型

- 型はデータの種類の名前
- 変数や定数を使用するには型を指定する必要がある
- 指定した型によって変数に格納できる内容と実行できる操作が決まる
- Delphiは型のチェックに厳密な言語

データ型

基本的なデータの型

整数型

型	範囲	書式
Integer	-2147483648 ~ 2147483647	符号付き 32bit
Cardinal	0 ~ 4294967295	符号なし 32bit
Shortint	-128 ~ 127	符号付き 8bit
Smallint	-32768 ~ 32767	符号付き 16bit
Longint	-2147483648 ~ 2147483647	符号付き 32bit
Int64	$-2^{63} \sim 2^{63}-1$	符号付き 64bit
Byte	0 ~ 255	符号なし 8bit
Word	0 ~ 65535	符号なし 16bit
LongWord	0 ~ 4294967295	符号なし 32bit

データ型

基本的なデータの型

実数型

型	範囲	有効桁数	サイズ (Byte 数)
Real	$-5.0 \times 10^{324} \sim 1.7 \times 10^{308}$	15 ~ 16	8
Single	$-1.5 \times 10^{45} \sim 3.4 \times 10^{38}$	7 ~ 8	4
Double	$-5.0 \times 10^{324} \sim 1.7 \times 10^{308}$	15 ~ 16	8
Extended	$-3.6 \times 10^{4951} \sim 1.1 \times 10^{4932}$	19 ~ 20	10
Comp	$-2^{63} \sim 2^{63}-1$	19 ~ 20	8
Currency	-922337203685477.5808 ~ 922337203685477.5807	19 ~ 20	8

文字型

型	用途	サイズ (Byte 数)
Char	WideChar と同じ	2
AnsiChar	ANSI 文字	1
WideChar	Unicode 文字	2

文字列型

型	用途	最大長
ShortString	下位互換性のため	255 文字
AnsiString	8 ビット (ANSI) 文字、DBCS ANSI、MBCS ANSI 等	~ 2 ³¹ 文字
WideString	Unicode 文字、マルチユーザーサーバーと多言語アプリケーション	~ 2 ³⁰ 文字

データ型

基本的なデータの型

論理型

型	用途	サイズ (Byte 数)
Boolean	True(真) または False(偽) の値の格納	1

基本的には次の 4 つの型を覚えておけば十分

- 整数型 • • • Integer
- 実数型 • • • Double
- 文字列型 • • • String
- 論理型 • • • Boolean

データ型

- 整数型は整数を扱う際に使用
a:=100;
- 実数型は実数を扱うのに使用
b:=123.456;
- 文字列型は文字列を扱うのに使用
c:='こんにちは';
文字列をシングルクォートで括って表現する
- 論理型は真偽値を代入する場合に使用
真偽値とはtrueとfalseで表現できる値
判定結果や制御文の判断に使用

文とコメント行

- 文とはプログラムを構成する要素 1 文はセミコロンで区切られる
- コメント文は、プログラムの動作には影響しない
- 注釈としてソースコード内に記述しておきたい内容を記述するためのもの1行のコメントを記述する場合

// 1行のコメント

- 複数行のコメントを記述する場合

```
{  
複数行のコメント  
}
```

手続きと関数

- 手続きと関数は、それ自体で完結した文の集合
- プログラム内の他の場所から呼び出すことができる
- 手続きと関数はルーチンとも呼ばれる
- 関数は実行すると値を返すルーチンで、手続きは値を返さないルーチン

手続きと関数

■ 手続きの宣言

```
procedure 手続き名(パラメータ1; パラメータ2);  
  begin  
  end;
```

procedure が手続きの宣言
beginとend;の間に行いたい処理を記述する

■ 関数の宣言

```
function 関数名(パラメータ1; パラメータ2): 戻り値の型;  
  begin  
  end;
```

functionが関数の宣言
begin と end; の間に行いたい処理を記述する
関数には戻り値の型も一緒に宣言する

パラメータ

- 多くの手続きと関数にはパラメータリストがある
- パラメータリストは、パラメータ宣言をセミコロンで区切って並べ、全体をカッコで囲んだもの
- 宣言の前には、var、const、out のいずれかの予約語を付けられる

varは変数パラメータ

constは定数パラメータ

outは出力パラメータ

予約後を付けない場合は値パラメータとなり、値が手続きや関数に渡される

変数パラメータ (var) は、渡されるパラメータに手続きや関数内で値を変更することができ、手続きや関数を呼び出した側でその値を受け取ることができる

定数パラメータ (const) は、手続きや関数内で値を変更することができず、読み取り専用のようなもの

出力パラメータ (out) は、手続きや関数からの結果を受け取ることしかできない

パラメータ

◆関数の場合

```
function SquareRectangle(const height, width : Double): Double;  
begin  
  result := height * width;  
end;
```

長方形の縦と横の値をパラメータとして関数に渡し、長方形の面積を返す。
戻り値は変数resultに代入する。

◆手続きの場合

```
procedure SquareRectangle(const height, width : Double; out res : Double);  
begin  
  res := height * width;  
end;
```

呼び出す側の処理

```
height := 2;  
width := 2;  
SquareRectangle(height, width, res);
```

パラメータ

◆手続きの場合（outを付けない場合）

```
procedure SquareRectangle(const height, width : Double; const res : Double);
```

呼び出す側の処理

```
height := 2;  
width := 2;  
res := 0;  
SquareRectangle(height, width, res);
```

この場合は、res の値が手続きを呼び出す前の値 **0 のまま**となる

関数や手続き内で値をパラメータとして返したい場合は、**out**か**var**を付けること

制御文

条件によって処理を分けたり、繰り返し処理をしたいという場合がある

プログラムを通常の順序通りに実行するのではなく、実行順序を変化させる命令文のことを制御文、制御構文、構造化文などと呼ぶ

制御文

■ if文

- if文は、条件式が真 (true) であるか偽 (false) であるかで実行する文を決める
- if文には「if 条件式 then文;」と「if 条件式 then 文 1 else 文 2;」の2つの形式がある
- 真と偽の処理の文は1文のみ記述することが可能
- elseの前にはセミコロンを付けてはいけない
- 複数の文を処理したい場合はブロックを使用する
- ブロックはbeginとendで囲まれたものとなり、1ブロックは1文とみなされる

制御文

if文の例

例 1) 処理が 1 文の場合

```
if a = 0 then
    ShowMessage('処理できません')
else
    b := c/a;
```

例 2) ブロックを使う場合

```
if a = 0 then
    begin
        result := false;
        ShowMessage('処理できません');
    end
else
    begin
        b := c/a;
        result := true;
    end;
```

制御文

■ case文

- case文は、順序のある条件によって、複数の処理に分ける場合に使用する
- 記述方法は、caseの次に変数または式の値（セクタ）、続いてofという形になる
- 各分岐条件の書き方は、分岐条件である値（ケースリスト）、次に：（コロン）そしてセクタとケースリストが一致した時に処理したい文
- どのケースリストにも一致しない場合に処理したい文がある場合には、elseが使用可能
- if文と同様に複数の文を処理したい場合はブロックを使用する

制御文

case文の例

```
Case セレクタ of  
  ケースリスト1 : 文1 ;  
  ケースリスト2 : 文2 ;  
  ケースリスト3 : 文3 ;  
else  
  文4 ;  
end;
```

記述例)

```
Case a of  
  1 : str := '小学生';  
  2 : str := '中学生';  
  3 : str := '高校生';  
  4 : str := '大学生';  
else  
  str := '該当なし';  
end;
```

制御文

■ for文

for文は処理を一定回数繰り返したい場合に使用する

```
for カウンタ変数 := カウンタの初期値 to カウンタの最終値 do  
  文;
```

又は

```
for カウンタ変数 := カウンタの初期値 downto カウンタの最終値 do  
  文;
```

- カウンタ変数は処理回数を制御する変数
- カウンタの初期値からカウンタの最終値まで文を繰り返す
- 複数の文を処理したい場合はブロックを使用
- for ... to の文では、カウンタ変数は1ずつ増え、for ... downto の文では、カウンタ変数は1ずつ減る
- カウンタ変数をループ内で変更することはできない
- ループ終了した後のカウンタ変数の値は不定

制御文

■ while文

while文は制御条件が真（true）の間だけ処理を繰り返す

```
while 制御条件 do  
    文;
```

- 制御条件が偽（false）になるとループを抜ける
- 制御条件が初めから偽(false)の場合は、1度もループせずに終了
- 複数の文を処理したい場合はブロックにする
- breakとcontinueを使用してループの制御が可能
 - ループの途中でループを抜きたい場合にはbreak
 - ループ内の処理の途中で次の反復に移行したい場合はcontinue

演算子

- 演算子は、Delphi 言語に組み込まれた定義済み関数のように機能する

- 演算子の種類

- 1.算術演算子（計算に使用する）
- 2.関係演算子・論理演算子（判定に使用する）

演算子

■ 算術演算子

- 算術演算子は計算に使用する
- 割り算については実数と整数で使う演算子が異なる
- $x \text{ div } y$ の値は、 x/y の値を 0 の方向に向かって最も近い整数に丸めた値
- `mod` は整数同士の割り算の余りを計算する

演算子	オペレーション
+	加算
-	減算
*	乗算
/	実数除算
div	整数除算
mod	余り

演算子

■ 代入演算子

- 代入演算子は、変数に値を代入する時に使用する
- 右辺の値を左辺に代入する

演算子	オペレーション
<code>:=</code>	代入

演算子

■ 関係演算子

- 関係演算子は 2 つの値の比較に使用される
- 結果は論理型となる

演算子	オペレーション
=	左辺と右辺が等しい
<>	左辺と右辺が等しくない
<	左辺が右辺より小さい
>	左辺が右辺より大きい
<=	左辺が右辺以下
>=	左辺が右辺以上

演算子

■ 論理演算子

- 論理演算子は2つの論理型の値の判定に使用する
- 結果は論理型となる

演算子	オペレーション
not	否定
and	論理積 (2つとも論理値が真の時、真になる)
or	論理和 (どちらかの論理値が真の時、真になる)
xor	非他的論理和 (どちらか片方の論理値だけが真の時、真になる)