2.5 制御構文

分岐やループなどの制御構文です。

分岐

if文

他の多くの言語と同じように、Dartではif文が利用できます。elseおよびelse ifは必須ではありません。中括弧（{ }）は実行処理が1文の場合に省略可能です。

```
final now = DateTime.now();
if (now.hour >= 6 && now.hour < 13)  {
  print('Good morning');
} else if (now.hour >= 13 && now.hour < 18) {
  print('Good afternoon');
} else {
  print('Good evening');
}
```

条件式はbool型以外をifなどの条件分岐に利用することはできません。

```
final number = 0;

if (number > 0) { // OK
  print('number is greater than 0');
}
if (number) { // => Error: A value of type 'int' can't be assigned to a variable
of type 'bool'.
  print('number is true');
}
```

if-case文

if-case文はパターンのマッチングと併せて変数へ分解する文法です（パターンと分解宣言については「2.6 パターン」で詳しく解説します）。以下の例はRecordのフィールドがnullでないことを判定し、変数messageと変数statusCodeに分解しています。いずれかがnullの場合にはelse節が実行されます。

```
final (String?, int?) response = ('OK', 200);
if (response case (String message, int statusCode)) {
  print('Response: message = $message, statusCode = $statusCode');
} else {
  print('No response received.');
}
// => Response: message = OK, statusCode = 200
```

さらに、whenキーワードに続けて条件式を記述することもできます。

```
final (String?, int?) response = ('OK', 200);
if (response case (String message, int statusCode) when statusCode == 200) {
  // messageが非nullかつ、statusCodeが200のときのみ、メッセージを出力
  print('Response: message = $message, statusCode = $statusCode');
} else {
  print('No response received.');
}
// => Response: message = OK, statusCode = 200
```

switch文

Dartのswitch文では、ケースに一致すると処理が実行されswitch文から抜けます。途中でswitch文を抜けるにはbreakが使え、switch文のあとの処理を続けて実行します。または、switch文が記述されたスコープから抜ける効果としてreturnやthrowを使うこともできます(throwと例外処理については「2.11 例外処理」で解説します)。ケースの処理が空の場合は次のケースに抜けます(フォールスルーと言います)。どのケースにも一致しない場合はdefaultが実行されます。

```
final String color = // 省略

switch (color) {
  case 'red':
    doSomethingIfRed();
  case 'blue':
    doSomethingIfBlue();
  case 'black':
    break; // switch文を抜ける
  case 'green':
  case 'yellow':
    doSomethingIfGreenOrYellow(); // greenとyellowのときに実行される
  default:
    throw 'Unexpected color';
}
```

continue文とラベルを使い、ケースの順に関係なくフォールスルーするこ

とが可能です。

```
final String color = // 省略
switch (color) {
  case 'red':
    doSomethingIfRed();
    continue other;
  case 'blue':
    doSomethingIfBlue();
  other:
  case 'black':
    throw 'Unexpected color'; // redとblackのときに実行される
}
```

switch文もwhenキーワードに続けて条件式を追加することができます。

```
final int? statusCode = null;
switch (statusCode) {
  case (int statusCode) when 100 <= statusCode && statusCode < 200:
    print('informational');
  case (int statusCode) when 200 <= statusCode && statusCode < 300:
    print('successful');
  case (int statusCode) when 300 <= statusCode && statusCode < 400:
    print('redirection');
  case (int statusCode) when 400 <= statusCode && statusCode < 500:
    print('client error');
  case (int statusCode) when 500 <= statusCode && statusCode < 600:
    print('server error');
  case (null):
    print('no response received.');
  default:
    print('unknown status code');
}
// => no response received.
```

式としてのswitch

switchを式として扱うことができます。以下の例は数値statusCodeの値に応じて、メッセージ文字列を生成しています。

```
final int statusCode = // 省略
final message = switch (statusCode) {
  >= 100 && < 200 => 'informational',
  >= 200 && < 300 => 'successful',
  >= 300 && < 400 => 'redirection',
  >= 400 && < 500 => 'client error',
  >= 500 && < 600 => 'server error',
```

```
  _ => 'unknown status code',
};
print(message);
```

式としてのswitchは次のような文法となります。

- **ケースはcaseキーワードではじまらない**
- **ケースの処理は単一の式**
- **空のケースは記述できない**
- **ケースのパターンと処理の区切りは : ではなく =>**
- **ケースは , で区切る**
- **デフォルトのケースは_（ワイルドカード表記）で記述する**

ループ

for文

他の多くの言語と同じように、Dartではfor文が利用できます。

```
for (int i = 0; i < 3; ++i) {
  print('index = $i');
}
// => index = 0
// => index = 1
// => index = 2
```

順番にアクセスできるコレクションのIterableクラスがあります。ListやSetのスーパークラスです。このIterableはfor-inの形式や、

```
final list = [0, 1, 2];
for (final element in list) {
  print('element = $element');
}
// => element = 0
// => element = 1
// => element = 2
```

forEachメソッドが利用できます。

```
final list = [0, 1, 2];
list.forEach((element) {
  print('element = $element');
});
```

```
// => element = 0
// => element = 1
// => element = 2
```

while文

Dartはループの前に条件式を評価するwhile文、ループのあとに条件式を評価するdo-while文の両方が利用できます。

```
final flag = // 省略

while (flag) {
  doSomething();
}

do {
  doSomething();
} while (flag);
```

breakとcontinue

forやwhile文はbreakキーワードでループから抜けます。また、continueキーワードで次のループへスキップします。

```
for (int i = 0; i < 10; ++i) {
  if (i % 2 == 0) {
    continue;
  }
  if (i > 6) {
    break;
  }
  print('index = $i');
}
// => index = 1
// => index = 3
// => index = 5
```

2.6 パターン

Dartにパターンという構文があります。パターン構文にはオブジェクトのマッチングと分解宣言の2つの機能があります。

マッチングはオブジェクトが特定の形式であるかを判断する機能です。以下は値がリテラルと一致するかを判定するパターンです。

```
final name = // 省略
switch (name) {
  case 'john': // nameが'john'と一致するか判定
    doSomething();
}
```

分解宣言はオブジェクトをいくつかの変数に分解する機能です。以下はRecordのフィールドをそれぞれ変数に分解します。

```
final record = ('cake', 300);
final (name, price) = record; // recordをnameとpriceに分解
print('This $name is $price yen.');
// => This cake is 300 yen.
```

パターンはいくつか種類があり、使える場所も異なります。そこで、本書ではパターンを以下のように分類し、それぞれを解説します。

❶マッチング機能しか持たないパターン
❷マッチングと分解宣言の2つの機能を持つパターン
❸パターンを補助する構文

マッチング機能しか持たないパターン

マッチング機能しか持たないパターンはswitch文、または式としてのswitch、if-case文で利用できます。変数の宣言には利用できません。

論理演算子、比較演算子

and(&&)、or(||)、==などの演算子を使ってパターンを記述できます。これらのパターンはマッチング機能しか持たないため変数の宣言には利用できません。

```
final int statusCode = // 省略
final message = switch (statusCode) {
  >= 100 && < 200 => 'informational',
  >= 200 && < 300 => 'successful',
  >= 300 && < 400 => 'redirection',
  >= 400 && < 500 => 'client error',
  >= 500 && < 600 => 'server error',
  _ => 'unknown status code',
};
print(message);
```

一致判定

リテラルや定数との一致判定をパターンとして記述できます。このパターンはマッチング機能しか持たないため変数の宣言には利用できません。

リテラルであれば、数値、bool、文字列リテラルが利用可能です。

```
switch (variable) {
  case 123:
    print('123');
  case 'str':
    print('str');
  case false:
    print('false');
}
```

コレクションの一致判定ではリテラルに`const`修飾子を付与する必要があります。

```
switch (variable) {
  case const [0, 1, 2]:
    print('list');
  case const {0, 1, 2}:
    print('set');
  case const {'key': 0}:
    print('map');
}
```

`const`や`static`を付与した定数も利用できます。

```
switch (variable) {
  case double.maxFinite:
    print('maxFinite');
  case const SomeClass():
    print('SomeClass');
}
```

マッチングと分解宣言の2つの機能を持つパターン

マッチングと分解宣言の2つの機能を持ち、マッチした際に変数にバインドするパターンです。前項の「マッチング機能しか持たないパターン」と同様にswitch文、または式としてのswitch、if-case文で利用できます。さらに、変数の宣言やfor文やfor-in文でも利用できます。

List

Listのパターンはvarまたはfinalからはじまり、Listリテラルのように大括弧([ ])で囲い、中に分解宣言する変数を記述します。

List型を分解宣言する場合は要素数が一致している必要があります。ただし、...を記述すると任意の長さをマッチさせることができます。

```
final [a, b, ..., c] = [0, 1, 2, 3, 4, 5];
print('a = $a, b = $b, c = $c');
// => a = 0, b = 1, c = 5
```

ちなみに、Setはパターンが利用できません。

Map

Mapのパターンはvarまたはfinalからはじまり、Mapリテラルのように中括弧({ })で囲い、キーとバリューを:で区切ります。キーが一致するとバリューが変数にバインドされます。

Mapを分解宣言する場合はマップ全体と一致する必要はありません。パターンに存在しないキーは無視されます。

```
final { 200: successful, 404: notFound } = {
  200 : 'OK',
  404 : 'Not Found',
  500 : 'Internal Server Error',
};
print('200 -> $successful, 404 -> $notFound');
// => 200 -> OK, 404 -> Not Found
```

Record

Recordはすべての構造が一致する必要があります。名前付きフィールドはパターンにもフィールド名を含める必要があります。

```
final record = (name: 'cake', price: 300);
// パターンにもフィールド名を含めているのでマッチする
```