第2章 Dartの言語仕様

Flutterの開発にはDart[注1]というプログラミング言語を用います。SwiftやKotlin、Javaと同じように静的型付け言語、クラスベースのオブジェクト指向言語です。2023年にメジャーアップデートされたDart 3ではすべてのコードがnull安全になったほか、Recordやパターンマッチングなどの新機能が追加されました。これらの新機能も解説します。

本章はできるだけDartの言語仕様を広く網羅することを目指しましたので分量が多くなっています。これは「新しいフレームワークに触れるときは、使用言語を頭にたたき込んでから」という筆者のスタイルを反映しています。とはいえ、勉強スタイルは人それぞれですので、本章は必要に応じて読み飛ばしていただいてもかまいません。

2.1 変数宣言

Dartの変数宣言の記述方法はいくつかあります。一つずつ見ていきましょう。

変数と型推論

```
int age = 0;
```

この例ではint型のageという変数を整数リテラル0で初期化しました。ここでのintのように変数の型を宣言する部分をDartでは**型注釈**(*Type Annotation*)と呼びます。

Dartは型推論の機能があります。型注釈を省略し代わりにvarと記述することで、変数の型を推論させることができます。

```
var age = 0;
```

定数 —— finalとconst

変更する予定のない変数は、定数を利用することが推奨されています。定

注1 https://dart.dev/

数として宣言するにはfinal修飾子を付けます。するとその変数への再代入はコンパイルエラーとなります。

```
final int age = 37;
age = 38;
// => Error: Can't assign to the final variable 'num'.
```

[注2]

また、この場合も型注釈を省略して型推論させることができます。

```
final age = 37;
```

finalのほか、constという修飾子でも定数を宣言できます。

```
const int age = 37;
const age = 37; // constも型推論可能
```

こちらはコンパイル時定数として扱われます。そのため、クラス変数などはconst宣言することはできません(静的なクラス変数であれば可能)。

また、finalで宣言されたクラスのフィールドは変更可能ですが、constで宣言されたクラスのフィールドは変更不可です。クラスについては「2.12 クラス」で詳しく解説します。

いろいろな初期値の与え方

変数は必ずしも宣言時に初期化される必要はありません。利用時までに初期化されていればOKです。初期化済みかどうかはDartコンパイラが判断してくれます。

```
final flag = DateTime.now().hour.isEven;

final int number; // 宣言時に初期化しない（この場合もfinalで宣言可能）
if (flag) {
    number = 0;
} else {
    number = 255;
}
print(number); // 必ず初期化されているのでOK
```

以下のように利用時までに初期化が保証されていないコードはコンパイルエラーとなります。

注2 本書では、コードやコマンドのコンパイル結果や実行結果などを「// =>」というコメント表記で示しています。

```
final userName = 'steve';

int number; // 宣言時に初期化しない
if (userName == 'joe') {
  number = 0;
} else if (userName == 'john') {
  number = 255;
} // else ケースがない

print(number);
// => Error: Non-nullable variable 'num' must be assigned before it can be used.
```

遅延初期化

変数の初期化をDartコンパイラが必ずしも正しく判断できない場合もあります。たとえば、グローバル変数の初期化などがそれにあたります。

そのようなときは**late**修飾子を付与することでコンパイラのチェックを回避できます。

```
late String globalVariable; // 宣言時に初期化しない

void main() {
  globalVariable = 'initialized';
  print(globalVariable);
  // => initialized
}
```

final lateのように**late**修飾子と**final**修飾子を併用し、一度初期化されたら変更不可にすることもできます。

また**late**修飾子は、宣言時に初期化処理を記述すると、変数にアクセスされるまで初期化処理を遅延することができます。以下の例では変数**variable**にアクセスするまで、初期値を計算する**highCostFunction**は実行されません。

```
late String variable = highCostFunction();
```

使用されるかどうかわからない変数や、初期化処理の実行コストが高い場合に用いると効果的です。

late修飾子を使う場合は、未初期化の変数にアクセスすると実行時エラーとなりますので利用には注意が必要です。

2.2 組み込み型

Dartの代表的な組み込み型を紹介します。

数値型

数値型を表現する型は整数型としてintクラス、浮動小数型としてdoubleクラス、以上の2つが用意されています。どちらも共通のスーパークラスnumを継承しています。

int —— 整数型

符号付整数型としてintクラスが提供されています。bitサイズはプラットフォームごとに異なります。昨今のiOSとAndroidを対象とするなら64bitのみと考えて差し支えないでしょう。

以下はintクラスとして推論される整数リテラルです。

```
final x = 1;
final hex = 0xFF; // 16進数リテラル
final exponent = 1e5; // 指数リテラル
```

double —— 浮動小数型

64bit浮動小数型としてdoubleクラスが提供されています。

以下はdoubleクラスとして推論される小数リテラルです。

```
final y = 1.1;
final exponents = 1.42e5; // 指数表記も可
```

String —— 文字列型

文字列型としてStringクラスが提供されています（その他、本書では詳しく解説しませんが、UTF-16コードポイントのコレクションとしてRunesクラス、（書記素クラスタによる）部分文字のコレクションとしてCharactersクラスがあります）。

Stringクラスとして推論される文字列リテラルは、ダブルクオートとシン

グルクオートどちらも対応しています。

```
final str1 = 'Hello, Dart!';
final str2 = "Hello, Dart!";
```

文字列リテラルに変数の値を挿入することもできます。変数名の前に$を置きます。式の結果を挿入する場合は${}で式を囲います。

```
final name = 'dart';

final str1 = 'Hello, $name!';
print(str1);
// => Hello, dart!

final str2 = 'Hello, ${name.toUpperCase()}!';
print(str2);
// => Hello, DART!
```

隣接する文字列リテラルは自動的に連結されます。+演算子で連結を明示することもできます。

```
final message1 = 'Hello, ' 'Dart!';
print(message1);
// => Hello, Dart!

final message2 = 'Hello, ' // 改行してもOK
  'Dart!';
print(message2);
// => Hello, Dart!

final message3 = 'Hello, ' +
  'Dart!';
print(message3);
// => Hello, Dart!
```

複数行の文字列を定義するには三重のダブルクオート、または三重のシングルクオートが便利です。

```
final message1 = "<div>\n  <p>Hello, Dart!</p>\n</div>";

final message2 = """
<div>
  <p>Hello, Dart!</p>
</div>
""";

final message3 = '''
<div>
```

```
  <p>Hello, Dart!</p>
</div>
''';
```

文字列リテラルの前にrを置くことで、改行文字などの特殊文字の解釈が無効にできます。

```
final message1 = 'Hello,\nDart!';
print(message1);
// => Hello,
// => Dart!

final message2 = r'Hello,\nDart!';
print(message2);
// => Hello,\nDart!
```

bool —— 論理型

論理型としてboolクラスが提供されます。

bool型のリテラルとしてtrueとfalseがあります。

```
final flag1 = true;
final flag2 = false;
```

List —— 配列

配列に相当する順序付きコレクションには、DartではListクラスが用意されています。リテラル表現は以下です。各要素をカンマ(,)で区切り、大括弧([ ])で囲います。

```
final list1 = [0, 1, 2, 3];
final list2 = [0, 1, 2, 3,]; // 末尾にカンマを付与してもOK
```

Listの要素の型は推論され、型の異なる要素を追加しようとするとコンパイル時にエラーとなります。

```
final intList = [0, 1, 2, 3];
intList.add(4); // OK
intList.add('abc'); // => Error: The argument type 'String' can't be assigned to
the parameter type 'int'.
```

Listの要素の型を明示するには以下のように型注釈を記述します。

```
final list = <int>[0, 1, 2, 3];
```

なお、Listには可変長と固定長の2種類が存在します。リテラルで作られるのは可変長Listになります。Listの名前付きコンストラクタunmodifiableを使うと、そのListは固定長となります(名前付きコンストラクタは「2.12 クラス」で解説します)。固定長Listの要素数を変更しようとすると実行時エラーとなります。

```
final baseList = [0, 1, 2, 3,];
final fixedLengthList = List.unmodifiable(baseList); // baseListを元に固定長の新しいインスタンスを生成
fixedLengthList.add(4); // 実行時エラー
```

Set —— 集合

順序が保持されない、要素が重複しないコレクションとしてSetクラスが用意されています。リテラル表現は以下です。各要素をカンマ(,)で区切り、中括弧({ })で囲います。

```
final map1 = { 'Apple', 'Orange', 'Grape' };
final map2 = { 'Apple', 'Orange', 'Grape', }; // 末尾にカンマを付与してもOK
```

Setの要素の型は推論され、型の異なる要素を追加しようとするとコンパイル時にエラーとなります。

```
final fruits = { 'Apple', 'Orange', 'Grape' };
fruits.add('Cherry'); // OK
fruits.add(123); // => Error: The argument type 'int' can't be assigned to the parameter type 'String'.
```

Setの要素の型を明示するには以下のように型注釈を記述します。

```
final fruits = <String>{ 'Apple', 'Orange', 'Grape' };
```

Map —— 連想配列

連想配列や辞書に相当するkey-valueペアとしてMapクラスが用意されています。他の多くの言語と同様にキーは重複しません。キーとバリューの型に制限はありません。リテラル表現は以下です。キーとバリューはコロン(:)、