```
final (name: n, price: p) = record;
print('This $n is $p yen.');
// => This cake is 300 yen.

// フィールド名がないのでマッチしない
final (String name, int price) = record;
```

フィールド名を変数名で推論させる記法もあります。

```
final record = (name: 'cake', price: 300);
// `:`に続けてフィールド名と同じ名前の変数を宣言
final (:name, :price) = record;
print('This $name is $price yen.');
// => This cake is 300 yen.
```

Object

その他のクラスをマッチさせることも可能です。クラスのゲッタから分解宣言できます。

```
class SomeClass {
  const SomeClass(this.x);
  final int x;
}

final someInstance = SomeClass(123);
final SomeClass(x: number) = someInstance;
print('x = $number');
// => x = 123
```

こちらもゲッタ名を変数名で推論させることができます。

```
class SomeClass {
  const SomeClass(this.x);
  final int x;
}

final someInstance = SomeClass(123);
final SomeClass(:x) = someInstance;
print('x = $x');
// => x = 123
```

このパターンはオブジェクト全体と一致する必要はありません。変数へのバインドを省略すれば、クラスの一致だけでマッチさせることも可能です。

```
final variable = // 省略
switch (variable) {
  case SomeClass():
```

```
    print('SomeClass');
  case String():
    print('String');
}
```

Tips for-in文での分解宣言

for-in文で分解宣言を利用する例をここで紹介します。Recordのリストをfor-in文で処理するときに、Recordのフィールドを変数にバインドしてみましょう。

```
final sweets = [
  (name: 'cake', price: 300),
  (name: 'dango', price: 250),
];

for (final (:name, :price) in sweets) {
  print('name = $name, price = $price');
}
// => name = cake, price = 300
// => name = dango, price = 250
```

このように、inの前に分解宣言を使って変数にバインドすることができます。

Mapの場合はというと、Iterableのサブクラスではないためfor-in文でループを回すことができません。ただし、Mapのentriesプロパティを使えば、キーとバリューのペアを変数にバインドしてループを回すことができます。

```
final map = {
  200 : 'OK',
  404 : 'Not Found',
  500 : 'Internal Server Error',
};
/* ◆ MapEntry
     キーとバリューを持ったMapの要素 */
// entriesプロパティでIterable<MapEntry>を取得できる
for (var MapEntry(key: key, value: value) in map.entries) {
  print('code: $key, $value');
}
// => code: 200, OK
// => code: 404, Not Found
// => code: 500, Internal Server Error
```

これは「Mapの分解宣言」というより、MapEntryをObjectとして分解宣言していることになります。少しややこしいので、Tipsとして紹介しました。

パターンを補助する構文

ここまで紹介したパターンと組み合わせ使うことで効果を発揮する記述方法を紹介します[注3]。

キャスト

分解宣言で変数に渡す際に、as演算子を使い値をキャストします。キャストに失敗すると実行時エラーとなります。

```
final List<Object> list = [0, 'one'];
final [number as int, str as String] = list;
```

nullチェック

値が非nullかどうかをチェックするパターンです。変数名の後ろに?を付与します。whenキーワードと組み合わせて非nullの場合にさらに条件を加えるような書き方ができます。

```
int? code = // 省略
switch (code) {
  case final i? when i >= 0:
    doSomething();
  default:
    print('code is null or negative');
}
```

nullアサーション

nullアサーションパターンは値がnullだった場合に実行時エラーとなります。変数名の後ろに!を付与します。

```
int? code = // 省略
switch (code) {
  case final i! when i >= 0:
    doSomething();
  default:
    print('code is negative');
}
```

注3　本書ではパターンを補助する構文として紹介していますが、Dartの公式ドキュメントではパターンの一種として扱われています。

ワイルドカード

_と記述するとワイルドカードパターンとなります。変数にバインドさせることなく、プレースホルダとして機能します。

```
final record = ('cake', 300);
final (name, _) = record;
print('name = $name');
// => name = cake
```

ワイルドカードパターンに型注釈を付与すると、クラスの一致だけでマッチさせることも可能です。

```
final variable = // 省略
switch (variable) {
  case SomeClass _:
    print('SomeClass');
  case String _:
    print('String');
}
```

2.7 例外処理

Dartの例外処理です。他の多くの言語と同様にthrowキーワードで例外をスローし、try-catch構文で例外を捕捉します。例外が捕捉されなければプログラムは中断します。ただし、Flutterはフレームワークが例外を捕捉する機構を持っているため、例外がスローされてもアプリは終了しません(意図的に終了させることはできます)。

```
void doSomething() {
  throw MyException();
}

try {
  doSomething();
} catch (e) {
  print(e);
}
```

例外の型 —— ErrorとException

Dartには**Error**型と**Exception**型があり、それぞれ**throw**キーワードで例外としてスローすることができます。

Error型はプログラムの失敗によりスローされるものとされています。間違った関数の使い方や、無効な引数が渡された場合など、プログラム上の問題に使用されます。呼び出し元で捕捉する必要のないものです。

一方、**Exception**型は捕捉されることを目的にしたクラスで、エラーに関する情報を持たせるべきとされています。

Dartは以上2つのタイプのほかに任意のオブジェクトを例外としてスローすることも可能ですが、製品レベルのコードでは推奨されていません。

例外の捕捉

捕捉する例外の型を指定する場合は**on**キーワードを使用します。

```
try {
  doSomething();
} on MyException {
  print('catch MyException');
}
```

例外オブジェクトを受け取りたい場合は**catch**キーワードを使用します。**catch**の第一引数は例外オブジェクト、第二引数はスタックトレースです。第二引数は省略可能です。

```
try {
  doSomething();
} catch(e, st) {
  print('catch $e');
  print('stackTrace $st');
}
```

捕捉する型を指定しつつ、例外オブジェクトを受け取る場合は**on**と**catch**を併用します。

```
try {
  doSomething();
} on MyException catch(e) {
  print('catch $e');
}
```

例外の再スロー

例外処理の中で、呼び出し元へ例外を再スローする場合はrethrowキーワードを使用します。

```
try {
  doSomething();
} on MyException catch(e) {
  print('catch $e');
  rethrow; // 呼び出し元へ例外を再スロー
}
```

finally句

例外がスローされるかどうかにかかわらず、最後に実行したい処理をfinally句に記述できます。

```
try {
  doSomething();
} on MyException catch(e) {
  print('catch $e');
} finally {
  doClean();
}
```

例外に一致するcatch句がない場合はfinally句が実行されたあとに例外が呼び出し元に伝播(でんぱ)します。

アサーション

プログラムの開発中に思わぬバグが潜んでいないかチェックする機能です。assertの第一引数へbool型の条件を渡します。条件がfalseの場合にプログラムの実行を中断します。

```
final variable = nonNullObject();
assert(variable != null); // オブジェクトがnullでないことをチェック
assert(variable != null, 'variables should not be null'); // メッセージを付与することもできる
```

Flutterではdebugビルドのときにだけassert文が処理されます。その特徴を利用し、debugビルドのときだけ実行したい処理を以下のように記述する

ことも可能です。

```
assert(() {
  print('debug mode');
  return true;
}());
```

Flutterの例外処理

Flutterアプリはフレームワークが例外を捕捉する機構を持っており、例外がスローされてもプログラムが終了するとは限りません。

フレームワークは2つの例外ハンドラを提供しています。

Flutterのフレームワーク自身がトリガするコールバック（レンダリング処理やウィジェットのbuildメソッドなど）で発生した例外はFlutterError.onErrorにルーティングされます。デフォルトではログをコンソールに出力する動作ですが、コールバックを上書きして独自に処理することも可能です。

```
void main() {
  FlutterError.onError = (details) {
    // do something
  };
  runApp(const MyApp());
}
```

それ以外のFlutter内で発生した例外（ボタンのタップイベントハンドラなど）はPlatformDispatcherでハンドリングします。

```
void main() {
  PlatformDispatcher.instance.onError = (error, stack) {
    print(error);
    return true; // 例外を処理した場合はtrueを返す
  };
  runApp(const MyApp());
}
```

2.8 コメント

Dartのコメント、およびドキュメントコメントについて解説します。

単一行のコメントは//、複数行のコメントは/*で開始し*/で終了します。

```
// 引数を2倍にする
int doubleValue(int value) {
  return value * 2;
}

/*
  以下のように書くこともできる
  int doubleValue(int value) => value * 2;
*/
```

ドキュメントコメントもサポートしています。///または/**で開始するコメントはドキュメントコメントとして扱われます(///を採用することが推奨されています)。クラスや関数、引数名などを[ ]で囲うとその定義へジャンプできるようになります。

```
///
/// 引数の値を2倍にして返す
///
/// この関数は、引数の値を2倍にして返す関数です。
/// 引数を半分にする仮数を返す関数として[half]関数があります。
int doubleValue(int value) {
  return value * 2;
}

/**
 * 引数の値を半分にして返す
 *
 * この関数は、引数の値を半分にして返す関数です。
 * 引数を2倍にする関数として[doubleValue]関数があります。
 */
double half(double value) {
  return value / 2;
}
```