構文はバリュー全体をブレース（{ }）で囲い、プレースホルダ名に続けて、plural というキーワードを記述します。

たとえば、日本語の arb ファイルを以下のように作成したとします。

./lib/l10n/app_ja.arb

```
{
  // 省略
  "numOfSearchResult": "{count, plural, =0{検索結果はありません} other{検索結果は{count}件です}}",
  "@numOfSearchResult": {
    "description": "検索結果",
    "placeholders": {
      "count": {
        "type": "int"
      }
    }
  }
}
```

実行結果は次のようになります。

```
Text(l10n.numOfSearchResult(0)),
// => 検索結果はありません
Text(l10n.numOfSearchResult(1)),
// => 検索結果は1件です
Text(l10n.numOfSearchResult(2)),
// => 検索結果は2件です
```

この英語訳を作ります。検索結果を「result」と訳し、さらに「result」と「results」を使い分けます。arb ファイルを以下のように記述することで実現できます。

```
  "numOfSearchResult": "{count, plural, =0{There is no result} =1{1 result found} other{{count} results found}}"
```

英語での実行結果は次のようになります。

```
Text(l10n.numOfSearchResult(0)),
// => There is no result
Text(l10n.numOfSearchResult(1)),
// => 1 result found
Text(l10n.numOfSearchResult(2)),
// => 2 results found
```

場合分けは =0 や =1 のほかに、few や many といったキーワードも存在しますが、日本語や英語では使われません。具体的な数値がどのケースに該当するかは言語ごとに異なるので注意してください。その挙動は ICU（*International*

Components for Unicode）というライブラリに準拠しており、ICUはCLDR（*Common Locale Data Repository*）が提供するデータを活用しています。詳細な挙動を知りたい方はUnicode CLDR[注2]のドキュメントを参照してください。

複数の言語への対応

対応する言語ごとにarbファイルを追加することでアプリを複数の言語に翻訳できます。arbファイルがどの言語に対応しているかはファイル名で決定します。アンダーバーと拡張子の間の文字列が、そのarbファイルが対応する言語として扱われます。たとえば、`app_ja.arb`ではアンダーバーと拡張子の間が`ja`なので日本語のarbファイルとして扱われます。日本語と英語に対応する場合は次のようなファイル構成となります。

```
~/project_root
└── lib
    ├── l10n
    │   ├── app_en.arb
    │   └── app_ja.arb
    └── ...（省略）
```

また、それ以外にも`@@locale`キーに言語を指定する方法もあります。

```
./lib/l10n/japanese.arb
{
  "@@locale": "ja",

  "helloWorld": "こんにちは世界！",
  "@helloWorld": {
    "description": "お決まりの挨拶"
  }
}
```

この場合は自由にファイルを決めて問題ありません。

```
~/project_root
└── lib
    ├── l10n
    │   ├── english.arb
    │   └── japanese.arb
    └── ...（省略）
```

注2　https://cldr.unicode.org/index/cldr-spec/plural-rules

4.3 プロジェクトにアセットを追加する

Flutterではアプリに同梱（どうこん）する画像やテキストファイルなどをアセットと呼びます。本書では主に画像の取り扱いについて紹介します。後半ではアセットを扱う際に便利なflutter_genというパッケージについても紹介します。アセットを扱う際はこのパッケージを使うことをお勧めします。

アプリに画像を追加する

まずはアセットを配置するディレクトリを作成します。ディレクトリ名は任意ですが**assets**などが一般的です。**assets**ディレクトリを作成したら、その中に画像を配置します。今回はPNG形式で円の画像を用意し、**circle.png**という名前で**assets**ディレクトリに配置します（**図4.9**）。

```
~/project_root
└── assets
    └── circle.png
```

なお、Flutterが対応している画像フォーマットはJPEG、WebP、GIF、PNG、BMP、WBMPです。

図4.9 **circle.png**

続いて、**pubspec.yaml**にアセットのパスを記述します。**flutter**セクションのサブセクションに**assets**を追加し、その下にアセットのパスを記述します。

./pubspec.yaml
```
flutter:
  # 省略
  assets:
    - assets/circle.png
```

これでアセットの準備は完了です。このアセットを表示するシンプルなアプリを作成してみましょう。

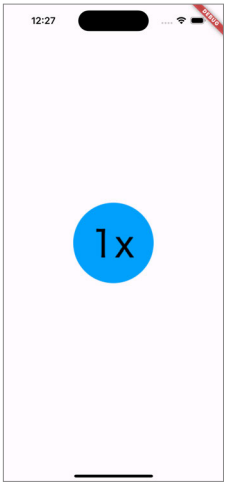
./lib/main.dart

```
import 'package:flutter/material.dart';

void main() {
  runApp(
    const MaterialApp(
      home: HomeScreen(),
    ),
  );
}

class HomeScreen extends StatelessWidget {
  const HomeScreen({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child: Image.asset('assets/circle.png'), ❶
      ),
    );
  }
}
```

画像を表示するウィジェットとしてImageウィジェットがあります。❶でアセットのパスを指定し、Imageウィジェットを作成しています。

アプリを実行すると、**図4.10**のように円が表示されます。

図4.10 **アセットの画像が表示される様子**



pubspec.yamlに記述するアセットのパスは、ディレクトリ単位で指定することもできます。以下のように、/で終わるパスを指定すると、そのディレクトリのすべてのファイルがアセットとして扱われます。ただし、通常はサブディレクトリを再帰的に探索しないので注意しましょう（後述の解像度バリエーションを除く）。

以下のようにアセットのパスを指定したとします。

./pubspec.yaml
```
flutter:
  # 省略
  assets:
    - assets/
```

次のようなディレクトリ構成であれば、circle.pngとsquare.pngはアセットしてアプリに組み込まれますが、icon.pngはアセットとして扱われません。

```
~/project_root
└── assets
    ├── circle.png
    ├── square.png
    └── icons
        └── icon.png
```

アプリ内でassets/icons/icon.pngを参照すると実行時エラーになります。

端末の解像度に応じて画像を切り替える

スマートフォンのディスプレイ解像度はさまざまです。解像度別にいくつか画像を用意し、実行時に切り替える手法があります。以下のようなディレクトリ構成でアセットを配置します。

```
~/project_root
└── assets
    ├── 2x
    │   └── circle.png
    ├── 3x
    │   └── circle.png
    └── circle.png
```

そして、前節で紹介したようにpubspec.yamlにアセットのパスをディレクトリ単位で指定します。

./pubspec.yaml
```
flutter:
  # 省略
```

```
assets:
  - assets/
```

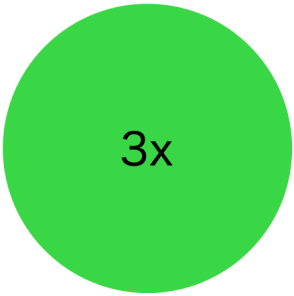
このように、数値と末尾に「x」で終わるディレクトリを作成すると、解像度別のバリエーションとして解釈されます。`assets/circle.png`が縦横72pxだとしたら、これを基準に2xには縦横144px、3xには縦横216pxの画像を配置します。

iPhoneを例に、これらの画像がどのように選択されるか説明します。iPhone15 Pro Maxのディスプレイ解像度は縦横2796 × 1290pxですが、論理解像度は縦横932 × 430ptです。論理解像度に対して、ディスプレイの物理解像度が3倍なのでiPhone15 Pro Maxで実行した場合、3xの画像が自動的に選択されます。

わかりやすいように、2xの画像(**図4.11**)と3xの画像(**図4.12**)として意図的に異なるものを用意しました。

図4.11 **2x配下のcircle.png**

図4.12 **3x配下のcircle.png**



iPhone14 Pro Maxのシミュレータ、iPhone SE(第3世代)のシミュレータそれぞれでアプリを実行すると、**図4.13**と**図4.14**のようになります。

ただし、この方法は画像の準備に手間がかかったり、アプリのファイルサイズが大きくなるという問題があります。この問題はベクタ画像を使うことで解決できますので、flutter_svgというパッケージを利用して画像を扱うことをお勧めします。後述のflutter_genと組み合わて利用する方法を紹介します。

図4.13 iPhone15 Pro Maxのシミュレータで実行した様子

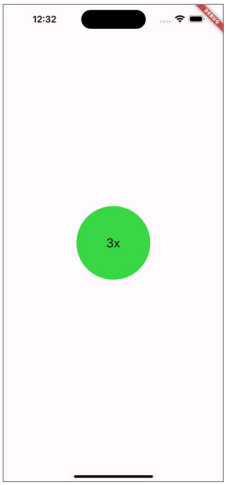
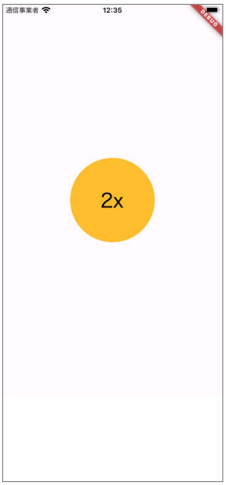


図4.14 iPhone SE（第3世代）のシミュレータで実行した様子



flutter_gen —— 型安全にアセットを扱うパッケージ

本節の最初に画像アセットを表示するアプリの例を紹介しましたが、アセットのパスを文字列で指定していました。パスを誤って入力すると、実行時エラーとなってしまいます。これを防ぐために、アセットを扱う際はflutter_genというパッケージを利用することをお勧めします。flutter_genは、アセットにアクセスするコードを自動生成してくれるパッケージです。

なお、前節でメッセージをローカライズした際に、生成されたコードが`.dart_tool/flutter_gen/gen_l10n`ディレクトリに出力されていましたが、ここで紹介するflutter_genパッケージとは関係ありません。

flutter_genを導入する

flutter_genを利用してみましょう。パッケージを導入するためにプロジェクトのディレクトリで、ターミナルから以下のコマンドを実行してください。

```
# build_runnerパッケージとflutter_gen_runnerパッケージを導入
$ flutter pub add --dev build_runner flutter_gen_runner
```

ソースコード生成ツールのbuild_runnerと、flutter_genのコードジェネレータであるflutter_gen_runnerを導入しました。

パッケージを追加したら、コードを生成するコマンドを実行します。

```
$ flutter packages pub run build_runner build
```

さっそくflutter_genが生成したコードを利用するコードを書いてみましょう。

./lib/main.dart

```
import 'package:flutter/material.dart';
import 'gen/assets.gen.dart'; ——❶

void main() {
  runApp(
    const MaterialApp(
      home: HomeScreen(),
    ),
  );
}

class HomeScreen extends StatelessWidget {
  const HomeScreen({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: Center(
        child:
            // Image.asset('assets/circle.png'),
            Assets.circle.image(), ——❷
      ),
    );
  }
}
```

まずflutter_genが生成したコードをインポートしています(❶)。アセットへのアクセスはflutter_genの実行によってAssetsクラスに定義されています。また、画像のアセットに関してはImageウィジェットを返すメソッドも生成され便利です(❷)。

SVG画像の利用

先にも述べましたが、さまざまなスマートフォンの解像度に合わせて画像を複数用意するのは手間がかかります。それを解決する方法として、SVG形式のファイルを利用する方法があります。SVGファイルはベクタ画像の一種で、拡大／縮小しても画質が劣化しません。FlutterはSVG画像をサポートしていませんが、flutter_svgというパッケージがSVG画像を描画するウィジェットを提供しています。