```
enum Shape {
  circle, triangle, square,
}
```

フィールドやメソッド、constantコンストラクタを持った高機能なEnumも宣言できます。通常のクラスに似た構文ですが、いくつかの条件があります。

- **1つ以上のインスタンスすべてが冒頭で宣言されていなくてはならない**
- **インスタンス変数はfinalでなければならない（mixinで追加されるものも同様）**
- **コンストラクタはconstantコンストラクタまたはfactoryコンストラクタが宣言可能**
- **他のクラスを継承することはできない**
- **index、hashCode、==演算子をオーバーライドすることはできない**
- **valuesという名前のメンバを宣言することができない**

```
// フィールドやfactoryコンストラクタを持ったEnum
enum Shape {
  circle(corner: 0),
  triangle(corner: 3),
  square(corner: 4);

  final int corner;

  const Shape({
    required this.corner,
  });

  factory Shape.ellipse() {
    return circle;
  }
}

// factoryコンストラクタからインスタンスを取得
final ellipse = Shape.ellipse();
// フィールドにアクセス
print(ellipse.corner);
// => 0
```

Enumの利用

Enumの型名に続きドット(.)のあとに列挙子名でアクセスすることができます。

```
final myShape = Shape.circle;
assert(myShape == Shape.circle);
```

各列挙子には宣言された順にindexが振られ、ゲッタから取得できます。また列挙子の名前をString型で取得できるnameプロパティも生成されます。

```
final myShape = Shape.circle;
print(myShape.index);
// => 0
print(myShape.name);
// => circle
```

Enumの型にはすべての列挙子をリストで得られるvaluesプロパティも生成されます。

```
Shape.values.forEach((shape) {
  print(shape.name);
});
// => circle
// => triangle
// => square
```

クラス修飾子

クラス修飾子はクラスやミックスインに付与し、インスタンス化や継承に制限を与えます。その効果はさまざまありますが、本書では以下のように分類してみました。

- **タイプ1**
  インスタンス化、extendsキーワードによる継承、implementsキーワードによる実装、これらに制限を与える
- **タイプ2**
  タイプ1以外の効果を持つ修飾子（タイプ1の効果を併せ持つ場合もある）

以下はクラス修飾子の一覧です。

- **abstract**
- **base**
- **final**
- **interface**
- **sealed**

・mixin

abstract

abstract修飾子はタイプ1です。

インスタンス化	extendsキーワードによる継承	implementsキーワードによる実装
×	○	○

abstract修飾子を使って宣言されたクラスは本体のない関数を宣言できます。またクラスをインスタンス化できなくなります。

```
abstract class Animal {
  String greet(); // 本体のないabstract関数
}

class Dog extends Animal {
  @override
  String greet() => 'bowwow';
}

// インスタンス化はできない
// final animal = Animal();

Animal dog = Dog();
print(dog.greet());
// => bowwow
```

base

base修飾子はタイプ1です(表は自身が宣言されたライブラリ以外での制限を示しています)。

インスタンス化	extendsキーワードによる継承	implementsキーワードによる実装
○	○	×

base修飾子を使って宣言されたクラスは自身が宣言されたライブラリ以外ではimplementsキーワードを使った実装を禁止します。

ライブラリ1

```
base class Animal {
  String greet() {
    return 'hello';
  }
}
```

ライブラリ2

```
// クラスの継承はOK。Dogクラスにもbase修飾子を付与しなければならない理由は後述。
base class Dog extends Animal {
}

// クラスの実装はNG、コンパイルエラー
// base class Cat implements Animal {
//   @override
//   String greet() => 'mew';
// }

final animal = Animal(); // インスタンス化はOK

final dog = Dog();
print(dog.greet());
// => hello
```

implementsキーワードを使ったクラスの実装が自身のライブラリ内に限定されるため、プライベートメソッドも含めて実装を強制することになります。base修飾子を使う目的はプライベートメソッドまで含めて全体の整合性を保つことにあります。そのため、base修飾子を使って宣言されたクラスはライブラリ外でもbase修飾子か、同じように実装を制限するクラス修飾子を付与しなければなりません。上の例ではDogクラスにもbase修飾子を付与しています。

ライブラリ1

```
base class Animal {
  void _sleep() {
    print('sleep');
  }

  String greet() {
    return 'hello';
  }
}

// 同一ライブラリ内であればクラスの実装OK
base class Cat implements Animal {

  // 同一ライブラリ内なのでプライベートメソッドもオーバーライドが強制される
  @override
  void _sleep() {
    // 省略
  }
```

```
  @override
  String greet() {
    return 'mew';
  }
}
```

interface

interface修飾子はタイプ1です（表は自身が宣言されたライブラリ以外での制限を示しています）。

インスタンス化	extendsキーワードによる継承	implementsキーワードによる実装
○	×	○

interface修飾子を使って宣言されたクラスは自身が宣言されたライブラリ以外ではextendsキーワードを使ったクラスの継承を禁止します。

ライブラリ1

```
interface class Animal {
  String greet() {
    return 'hello';
  }
}
```

ライブラリ2

```
// クラスの継承はNG
// class Dog extends Animal {
// }

// クラスの実装はOK
class Cat implements Animal {
  @override
  String greet() => 'mew';
}

final animal = Animal(); // インスタンス化はOK

final cat = Cat();
print(cat.greet());
// => mew
```

implementsキーワードを使い、すべてのメソッドを実装する必要があります。常に同じライブラリで実装された既知の実装が呼び出されることが保証

できます。

abstractとinterfaceの組み合わせ

abstractとinterfaceの2つの修飾子を組み合わせると実装を持たない純粋なインタフェースを定義することが可能になります。interface修飾子の効果として、外部のライブラリではimplementsキーワードを使ったクラス実装が強制され、abstract修飾子の効果として実装を持たない関数を宣言できます。

final

final修飾子はタイプ1です(表は自身が宣言されたライブラリ以外での制限を示しています)。

インスタンス化	extendsキーワードによる継承	implementsキーワードによる実装
○	×	×

final修飾子を使って宣言されたクラスは、自身が宣言されたライブラリ以外ではすべてのサブタイプ化を禁止します。extendsキーワードを使ったクラスの継承、implementsキーワードを使ったクラスの実装の両方が禁止されます。

ライブラリ1

```
final class Animal {
  String greet() {
    return 'hello';
  }
}
```

ライブラリ2

```
// クラスの継承はNG
// base class Dog extends Animal {
// }

// クラスの実装もNG
// base class Cat implements Animal {
//  @override
//  String greet() => 'mew';
// }


final animal = Animal(); // インスタンス化はOK
```

mixin

mixin修飾子はタイプ2です。

mixin修飾子を使って宣言されたクラスはミックスインのように扱うことが可能でありながら、クラスなのでインスタンス化することができます。ただし、ミックスインと同様にextendsは使えずコンストラクタも宣言できません。

```
mixin class Horse { // `mixin class`で宣言
}

mixin Bird {
}

class Pegasus with Bird, Horse { // `with`キーワードでHorseをmixin

}

final horse = Horse(); // Horseはインスタンス化可能
```

sealed

sealed修飾子はタイプ2です。sealed修飾子を使うとサブタイプをEnumのように扱うことができます。sealed修飾子を使って宣言されたクラスは、自身が宣言されたライブラリ以外ではすべてのサブタイプ化を禁止します。この点はfinalと共通していますが、さらにクラス自身が暗黙的にabstract classとして扱われます。

ライブラリ1

```
sealed class Shape {
  abstract int corner;
}

// Shape shape = Shape(); インスタンス化はNG

class Rectangle extends Shape {
  @override
  int corner = 4;
}

class Triangle extends Shape {
  @override
  int corner = 3;
}
```

```
class Circle extends Shape {
  @override
  int corner = 0;
}
```

switch文ですべてのサブタイプが網羅されていなければ、コンパイラが警告を出します。

ライブラリ2

```
// サブクラス化はNG
// class Rectangle extends Shape {
//   @override
//   int corner = 4;
// }

final Shape shape = getShepe();

switch (shape) {
  case Rectangle():
    print('Rectangle');
  case Triangle():
    print('Triangle');
  case Circle():
    print('Circle');
}
```

2.13 非同期処理

Dartの非同期処理です。Future型とStream型、スレッドのようなしくみのアイソレートについて解説します。

Future型

Dartには非同期処理の結果を取り扱うFuture型があります。

```
import 'dart:io';

void main() {
```