・constantコンストラクタ
・名前付きコンストラクタ
・factoryコンストラクタ

なお、コンストラクタを記述しなければ引数なしのデフォルトコンストラクタが提供されます。

constantコンストラクタ

クラスインスタンスをコンパイル時定数として扱うためにはconstantコンストラクタが必要です。コンストラクタに`const`キーワードを付与します。インスタンス変数はすべて再代入不可な`final`である必要があります。

```
class Point {
  const Point(this.x, this.y);

  final int x;
  final int y;
}

const point = Point(1, 2);
```

constantコンストラクタは常にコンパイル時定数を生成するとは限りません。constantコンストラクタの前に`const`キーワードを付与する、または`const`変数に代入した場合に、常に同じインスタンスが使われます。無駄なインスタンス生成を避けることができるため、Flutterのパフォーマンス向上に役立ちます。

```
class Point {
  const Point(this.x, this.y);

  final int x;
  final int y;
}

final point1 = const Point(1, 2); // constantコンストラクタの前に`const`キーワードを付与
const point2 = Point(1, 2); // `const`変数に代入
final point3 = Point(1, 2);
print('${point1 == point2}'); // point1とpoint2は同じインスタンス
// => true
print('${point1 == point3}');
// => false
```

名前付きコンストラクタ

コンストラクタに識別子を追加して、名前付きコンストラクタを宣言することができます。クラスに複数のコンストラクタを宣言する場合、特別な意味を持ったインスタンスを生成する場合などに有効です。

通常のコンストラクタはクラス名で宣言しますが、名前付きコンストラクタは**クラス名 . 識別子**の形で宣言します。

```
class Point {
  const Point(this.x, this.y);
  const Point.zero() : x = 0, y = 0; // 名前付きコンストラクタ
  final int x;
  final int y;
}
```

また、コンストラクタから自クラスの別のコンストラクタを呼び出すことも可能です。

```
class Point {
  const Point(this.x, this.y);
  const Point.zero() : this(0, 0); // 名前のないコンストラクタを呼び出し
  final int x;
  final int y;
}
```

factoryコンストラクタ

必ずしも新しいインスタンスを生成しない場合(キャッシュの利用)や、初期化リストに記述できないロジックがある場合はfactoryコンストラクタを利用します。コンストラクタに**factory**キーワードを付与し、コンストラクタ本体でインスタンスを返すreturn文を記述する必要があります。

```
class UserData {
  static final Map<int, UserData> _cache = {};

  factory UserData.fromCache(int userId) {
    // キャッシュを探す
    final cache = _cache[userId];
    if (cache != null) {
      // キャッシュがあったので返す
      return cache;
    }

    // キャッシュがなかったので新しいインスタンスを生成して返す
    final newInstance = UserData();
```

```
    _cache[userId] = newInstance;
    return newInstance;
  }

  // 省略
}
```

クラス継承

Dartの公式ドキュメントではクラス継承のことを「拡張」(*Extend a class*)と呼んでいますが、本書では「継承」と呼ぶこととします。のちほど紹介します「拡張メソッド」(*Extension methods*)と区別するためです。

サブクラスの宣言はextendsキーワードに続けてスーパークラスの名前を記述します。

```
class Animal {
  String greet() => 'hello';
}

class Dog extends Animal {

}

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

スーパークラスを参照するにはsuperキーワードを用います。

```
class Animal {
  String greet() => 'hello';
}

class Dog extends Animal {
  String sayHello() => super.greet();
}

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

スーパークラスのメソッドをオーバーライドする際は、@overrideアノテーションを付与することが推奨されています。

```
class Animal {
  String greet() => 'hello';
}

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

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

メソッドのオーバーライドにはいくつかの条件があります。

- **戻り値の型がスーパークラスのメソッドの戻り値の型と同じ、またはそのサブタイプである**
- **引数の型がスーパークラスのメソッドの引数の型と同じ、またはそのスーパークラスである**
- **位置パラメータの数が同じである**
- **ジェネリックメソッドを非ジェネリックメソッドでオーバーライドできない、また非ジェネリックメソッドをジェネリックメソッドでオーバーライドできない**

また、戻り値の型がnull許容型のメソッドを非null許容型のメソッドでオーバーライドすることもできます。

```
class Animal {
  String? greet() => null; // 戻り値はnull許容型
}

class Dog extends Animal {
  @override
  String greet() => 'bowwow'; // 戻り値を非null許容型でオーバーライド
}
```

スーパークラスのコンストラクタ

サブクラスのコンストラクタでは、スーパークラスの引数のないコンストラクタが自動的に呼び出されます。スーパークラスに引数なしコンストラクタがない場合は、明示的にスーパークラスのコンストラクタを呼び出す必要があります。

```
class Animal {
  Animal(this.name);
  final String name;
```

```
}

class Dog extends Animal {
  Dog(String name) : super(name);
}
```

コンストラクタの後ろ、**super**キーワードに続けてスーパークラスのコンストラクタを呼び出します。上の例のように、コンストラクタ引数をそのままスーパークラスのコンストラクタに渡す記述は糖衣構文が用意されています。

```
class Animal {
  Animal(this.name);
  final String name;
}

class Dog extends Animal {
  Dog(super.name);
}
```

暗黙のインタフェース

Dartではすべてのクラスは暗黙的にインタフェースが定義されています。そのクラスのすべての関数とインスタンスメンバを持ったインタフェースです。

implementsキーワードに続けてインタフェースとして実装する型名を記述します。

```
class Animal {
  String greet() => 'hello';
}

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

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

すべてのインスタンスメンバ、メソッドをオーバーライドしなければならない点が、**extends**キーワードで継承するときとの違いです。

拡張メソッド

既存のクラスへメソッドやゲッタ、セッタを追加することができます。拡張メソッドは以下のような文法で宣言します。

```
extension <拡張名> on <拡張対象の型> {
  ...
}
```

List型に要素を入れ替えるswap関数を拡張する例を示します。

```
extension SwapList<T> on List<T> {
  // 引数のインデックスの要素を入れ替える拡張メソッド
  void swap(int index1, int index2) {
    final tmp = this[index1];
    this[index1] = this[index2];
    this[index2] = tmp;
  }
}

final list = [1, 2, 3];
list.swap(0, 2); // インデックス0と2の要素を入れ替える
print(list);
// => [3, 2, 1]
```

静的な拡張メソッドを宣言することはできません。ですが、拡張メソッドから呼び出し可能なヘルパ関数として利用することができます。

```
extension SwapList<T> on List<T> {

  // 静的メソッド（拡張メソッドから呼び出し可能）
  static bool noNeedToSwap(List<T> list) {
    return list.isEmpty;
  }

  void swap(int index1, int index2) {
    if (noNeedToSwap(this)) { // 拡張メソッド内で静的メソッドを利用するのはOK
      return;
    }

    final tmp = this[index1];
    this[index1] = this[index2];
    this[index2] = tmp;
  }
}

final list = [1, 2, 3];
```

```
// 拡張メソッド以外からは呼び出せない
List.noNeedToSwap(list); // => Error: The method 'noNeedToSwap' isn't defined for
the type 'List'.
```

拡張名のない拡張メソッドは同一ファイル内でのみ参照可能です。

```
extension on List<T> {
  void swap(int index1, int index2) {
    final tmp = this[index1];
    this[index1] = this[index2];
    this[index2] = tmp;
  }
}
```

mixin ―― クラスに機能を追加する

Dartは多重継承を許可していませんが、それに似た言語仕様としてmixin（ミックスイン）があります。**with**キーワードに続けてミックスイン名を記述します。

```
mixin Horse {
  void run() {
    print('run');
  }
}

mixin Bird {
  void fly() {
    print('fly');
  }
}

class Pegasus with Bird, Horse {
}

final pegasus = Pegasus();
pegasus.run(); // PegasusはHorseのメソッドを持つ
// => run
pegasus.fly(); // PegasusはBirdのメソッドも持つ
// => fly
```

ミックスイン（上の例では**Horse**と**Bird**）はクラスのようにメソッドやフィールドを宣言できます。クラスとの違いは、

・インスタンス化できないこと
・extendsキーワードを使って他のクラスから継承できないこと
・コンストラクタを宣言できないこと

です。

ミックスインを宣言する際に、使用するクラスを制限することも可能です。次の例ではミックスインHorseとBirdはonキーワードでクラスAnimalでしか使用できないよう制限をしています。この制限によりミックスインHorseとBird内でクラスAnimalのメソッドが利用できます。

```
class Animal {
  String greet() => 'hello';
}

// onキーワードで使用可能なクラスをAnimalに制限
mixin Horse on Animal {
  void run() {
    greet(); // Animalのメソッドを使用可能
    print('run');
  }
}

// onキーワードで使用可能なクラスをAnimalに制限
mixin Bird on Animal {
  void fly() {
    greet(); // Animalのメソッドを使用可能
    print('fly');
  }
}

class Pegasus extends Animal with Bird, Horse {
}
```

なお、mixin classで宣言する場合はonキーワードは使えません。

Enum

Dartの列挙型です。

Enumの宣言

列挙型はenumキーワードを使い宣言します。