要素はカンマ(,)で区切り、全体を中括弧({ })で囲います。

```
final map1 = {
  200: 'OK',
  403: 'access forbidden',
  404: 'not found'
};
final map2 = {
  200: 'OK',
  403: 'access forbidden',
  404: 'not found', // 末尾にカンマを付与してもOK
};
```

ListやSetと同様にキーやバリューの型は推論され、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 statusCodes = {
  200: 'OK',
  403: 'access forbidden',
  404: 'not found'
};

statusCodes[204] = 'No Content'; // OK
statusCodes['204'] = 'No Content'; // => Error: A value of type 'String' can't be
assigned to a variable of type 'int'.
```

Map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 statusCodes = <int, String>{
  200: 'OK',
  403: 'access forbidden',
  404: 'not found'
};
```

SetとMapはリテラルが似ていますが、以下はMapとして推論されます。

```
final setOrMap = {};
print(setOrMap is Map); // is演算子で型を確認
// => true
```

Record ―― タプル

Recordは複数の値を集約した不変の匿名型を表現します。他の多くの言語にあるタプル型によく似ています。

Recordの初期化はカンマ(,)区切りでフィールドを記述し括弧(( ))で囲います。

```
final record1 = (300, 'cake');
```

Recordの型注釈はカンマ(,)区切りでフィールドの型注釈を記述し括弧(( ))で囲います。

```
final (int, String) record2 = record1;
```

フィールドに名前を付与することもできます。名前を付けたフィールドを「名前付きフィールド」、名前を付けないフィールドを「位置フィールド」と呼びます。型注釈では名前付きフィールドを中括弧({ })で囲います。

```
final record1 = (price: 300, name: 'cake');
// 型注釈では名前付きフィールドを中括弧で囲う
final ({int price, String name}) record2 = (price: 300, name: 'cake');
```

名前付きフィールドの記述順は等値性に影響を与えません。

```
final record1 = (price: 300, name: 'cake');
final record2 = (name: 'cake', price: 300);
print(record1 == record2);
// => true
```

型注釈の中では位置フィールドに名前を付与することができます。その名前はフィールドの等値性に影響を与えません。

```
// 左辺、Recordの型注釈でフィールドに名前を付与している
// 中括弧で囲っていないので名前付きフィールドではない
final (int price, String name) record1 = (300, 'cake');
final (int x, String y) record2 = (300, 'cake');
print(record1 == record2);
// => true
```

名前付きフィールドと位置フィールドを混在させることが可能です。その場合、型注釈では位置フィールドが常に先頭に配置されます。

```
// 99のみが位置フィールド
final record1 = (price: 300, name: 'cake', 99);
// 型注釈では位置フィールドが先頭
final (int count, {String name, int price}) record2 = record1;
```

名前付きフィールドは、同名のゲッタから読み取りが可能です。位置フィールドは$に続けて引数の順序のゲッタが作られます。なお、Recordは不変なためセッタはありません(ゲッタ、セッタについては「2.12 クラス」で解説します)。

```
final record = (price: 300, name: 'cake', 99);

print(record.price);
// => 300
print(record.name);
// => cake
print(record.$1);
// => 99
```

Objectクラス —— すべてのクラスのスーパークラス

Objectクラスは Dartのすべてのクラスのスーパークラスです（スーパークラスや継承については「2.12 クラス」で解説します）。代表的な用途は、型の異なる要素を持ったコレクションを扱う場合です。この例では、変数listは List<Object>型に推論されます。

```
final list = [
  0,
  'abc',
  true,
];
```

近い表現に dynamic という型があるので紹介します。

```
final List<dynamic> list = [
  0,
  'abc',
  true,
];
```

dynamicは特殊な型で、コンパイル時に型のチェックが行われません。存在しないメソッドを呼び出すようなコードであってもコンパイルエラーになりませんし、nullかどうかの判断もされません（nullについては「2.9 null安全」で解説します）。よって、実行時エラーのリスクが高まります。明確な理由がない限り、dynamicの利用は避けObjectまたはObject?（この?の文法は「2.9 null安全」で解説します）を利用すべきです。

2.3 ジェネリクス

他の多くの言語と同じように、Dartにもジェネリクスの機能があります。型をパラメータ化し、特定の型に依存しない汎用的な実装を行うことができます。すでに紹介したListやMapは、要素の型をパラメータとして受け取るジェネリック型です。

```
final List<int> intList = [0, 1, 2]; // intのリスト
final stringList = <String>['a', 'b', 'c']; // Stringのリスト
```

ジェネリッククラス

型名のあとに括弧(< >)で型のパラメータ名を与えます。慣習的にTなど1文字で表現します。クラス内で型のパラメータ名を実際の型名のように扱うことができます。

```
// Tが型のパラメータ名
class Foo<T> {
  // フィールド `_value`の型をパラメータ名Tで宣言
  T _value;
  Foo(this._value);

  // 戻り値の型をパラメータ名Tで宣言
  T getValue() {
    return _value;
  }
}

final intFoo = Foo(3);
print(intFoo.getValue());
// => 3

final stringFoo = Foo('hoge');
print(stringFoo.getValue());
// => hoge
```

ジェネリック関数

ジェネリクスな関数は関数名のあとに型パラメータ名を記述します。型パラメータは戻り値、引数、またローカル変数で使用可能です。

```
// `T?`はT型またはnullを表す
T? firstOrNull<T>(List<T> list) {
  if (list.isEmpty) {
    return null;
  }
  return list[0];
}

final list1 = [1, 2, 3];
print(firstOrNull(list1));
// => 1

final list2 = <String>[];
print(firstOrNull(list2));
// => null
```

2.4 演算子

他の多くの言語と同じように扱えるものを中心に、Dartのオペレータの一部を紹介します。他の言語機能と併せて解説すべきものは別の項で紹介します。

算術演算子

和算や乗算などの四則演算を行う演算子は他の多くの言語と同じように扱うことができます。

```
print(2 + 3);
// => 5
print(2 - 3);
// => -1
print(2 * 3);
// => 6
print(5 / 2);
```

```
// => 2.5
print(5 % 2);
// => 1
```

インクリメント、デクリメントについても他の多くの言語と同じように扱うことができます。

```
int a;
int b;

a = 0;
b = ++a;
print("$a, $b");
// => 1, 1

a = 0;
b = a++;
print("$a, $b");
// => 1, 0

a = 0;
b = --a;
print("$a, $b");
// => -1, -1

a = 0;
b = a--;
print("$a, $b");
// => -1, 0
```

比較演算子

比較演算についても、他の多くの言語と同じように扱うことができます。

```
print(2 == 2);
// => true
print(2 != 1);
// => true
print(10 > 2);
// => true
print(2 < 10);
// => true
print(5 >= 5);
// => true
print(5 <= 5);
// => true
```

==オペレータはデフォルトの動作は参照の比較です。オーバーライドして同値性を指定することも可能です。また、両方がnullの場合はtrue、一方のみがnullの場合はfalseとなります。

三項演算子

Dartは以下の三項演算子が利用できます。

```
条件式 ? 式1 : 式2
```

条件式がtrueなら式1が評価され戻り値となり、falseなら式2が評価され戻り値となります。

```
int a = 128;
int b = 256;
final max = a > b ? a : b;
print(max);
// => 256
```

カスケード記法

カスケード記法は同じオブジェクトに対して、繰り返し操作を行うときに便利な記述方法です。オブジェクトのメソッドやプロパティへドット2つ(..)でアクセスすると、そのオブジェクトそのものが戻り値となります。

```
final sb = StringBuffer()
  ..write('Hello');
print(sb.toString());
// => Hello
```

上の例ではStringBufferのインスタンスを生成しwriteメソッドを呼び出しました。writeメソッドの戻り値はvoid型ですが、カスケード記法で呼び出しているため、変数sbにはStringBufferのインスタンスが代入されます。

以下のように、同じインスタンスに繰り返しアクセスする場合に便利です。

```
final sb = StringBuffer()
  ..write('Hello')
  ..write(', ')
  ..write('Dart!!');
print(sb.toString());
// => Hello, Dart!!
```

コレクションのオペレータ

List、Set、Mapのリテラルでのみ利用できるオペレータです。

Spread演算子

複数のコレクションを結合する際に便利なオペレータです。コレクションリテラル内で...を記述すると、そのコレクションの要素が展開されます。

```
final list1 = [0, 1, 2, 3];
// list1はその要素が展開され、list2の要素となる
final list2 = [-1, ...list1];
print(list2);
// => [-1, 0, 1, 2, 3]
```

制御構文演算子

コレクションのリテラル内でifやforが記述できます。要素を追加する条件を記述したり、他のコレクションを追加したりする際に前処理を行うことができます。

```
// flagがtrueのときのみ、3を追加
final list = [0, 1, 2, if (flag) 3];

final list1 = [1, 2, 3];
// list1の要素を2倍したものを追加
final list2 = [0, for (var i in list1) i * 2];
print(list2);
// => [0, 2, 4, 6]
```