Patterns

Version note

Patterns require a language version of at least 3.0.

Patterns are a syntactic category in the Dart language, like statements and expressions. A pattern represents the shape of a set of values that it may match against actual values.

This page describes:

- What patterns do.
- Where patterns are allowed in Dart code.
- What the common use cases for patterns are.

To learn about the different kinds of patterns, visit the pattern types page.

What patterns do

In general, a pattern may **match** a value, **destructure** a value, or both, depending on the context and shape of the pattern.

First, *pattern matching* allows you to check whether a given value:

- Has a certain shape.
- Is a certain constant.
- Is equal to something else.
- Has a certain type.

Then, *pattern destructuring* provides you with a convenient declarative syntax to break that value into its constituent parts. The same pattern can also let you bind variables to some or all of those parts in the process.

Matching

A pattern always tests against a value to determine if the value has the form you expect. In other words, you are checking if the value *matches* the pattern.

What constitutes a match depends on what kind of pattern you are using. For example, a constant pattern matches if the value is equal to the pattern's constant:

```dart
switch (number) {
  // Constant pattern matches if 1 == number.
  case 1:
    print('one');
}
```

Many patterns make use of subpatterns, sometimes called *outer* and *inner* patterns, respectively. Patterns match recursively on their subpatterns. For example, the individual fields of any collection-type pattern could be variable patterns or constant patterns:

```dart
const a = 'a';
const b = 'b';
switch (obj) {
  // List pattern [a, b] matches obj first if obj is a list with two fields,
  // then if its fields match the constant subpatterns 'a' and 'b'.
  case [a, b]:
    print('$a, $b');
}
```

To ignore parts of a matched value, you can use a wildcard pattern as a placeholder. In the case of list patterns, you can use a rest element.

Destructuring

When an object and pattern match, the pattern can then access the object's data and extract it in parts. In other words, the pattern *destructures* the object:

```dart
var numList = [1, 2, 3];
// List pattern [a, b, c] destructures the three elements from numList...
var [a, b, c] = numList;
// ...and assigns them to new variables.
print(a + b + c);
```

You can nest any kind of pattern inside a destructuring pattern. For example, this case pattern matches and destructures a two-element list whose first element is `'a'` or `'b'`:

```dart
switch (list) {
  case ['a' || 'b', var c]:
    print(c);
}
```

Places patterns can appear

You can use patterns in several places in the Dart language:

- Local variable declarations and assignments
- for and for-in loops
- if-case and switch-case
- Control flow in collection literals

This section describes common use cases for matching and destructuring with patterns.

Variable declaration

You can use a *pattern variable declaration* anywhere Dart allows local variable declaration. The pattern matches against the value on the right of the declaration. Once matched, it destructures the value and binds it to new local variables:

```dart
// Declares new variables a, b, and c.
var (a, [b, c]) = ('str', [1, 2]);
```

A pattern variable declaration must start with either `var` or `final`, followed by a pattern.

Variable assignment

A *variable assignment pattern* falls on the left side of an assignment. First, it destructures the matched object. Then it assigns the values to *existing* variables, instead of binding new ones.

Use a variable assignment pattern to swap the values of two variables without declaring a third temporary one:

```dart
var (a, b) = ('left', 'right');
(b, a) = (a, b); // Swap.
print('$a $b'); // Prints "right left".
```

Switch statements and expressions

Every case clause contains a pattern. This applies to switch statements and expressions, as well as if-case statements. You can use any kind of pattern in a case.

Case patterns are refutable. They allow control flow to either:

- Match and destructure the object being switched on.
- Continue execution if the object doesn't match.

The values that a pattern destructures in a case become local variables. Their scope is only within the body of that case.

```dart
switch (obj) {
  // Matches if 1 == obj.
  case 1:
    print('one');

  // Matches if the value of obj is between the
  // constant values of 'first' and 'last'.
  case >= first && <= last:
    print('in range');

  // Matches if obj is a record with two fields,
  // then assigns the fields to 'a' and 'b'.
  case (var a, var b):
    print('a = $a, b = $b');

  default:
}
```

Logical-or patterns are useful for having multiple cases share a body in switch expressions or statements:

```dart
var isPrimary = switch (color) {
  Color.red || Color.yellow || Color.blue => true,
  _ => false
};
```

Switch statements can have multiple cases share a body without using logical-or patterns, but they are still uniquely useful for allowing multiple cases to share a guard:

```dart
switch (shape) {
  case Square(size: var s) || Circle(size: var s) when s > 0:
    print('Non-empty symmetric shape');
}
```

Guard clauses evaluate an arbitrary condition as part of a case, without exiting the switch if the condition is false (like using an `if` statement in the case body would cause).

```dart
switch (pair) {
  case (int a, int b):
    if (a > b) print('First element greater');
  // If false, prints nothing and exits the switch.
  case (int a, int b) when a > b:
    // If false, prints nothing but proceeds to next case.
    print('First element greater');
  case (int a, int b):
    print('First element not greater');
}
```

For and for-in loops

You can use patterns in for and for-in loops to iterate-over and destructure values in a collection.

This example uses object destructuring in a for-in loop to destructure the `MapEntry` objects that a `<Map>.entries` call returns:

```dart
Map<String, int> hist = {
  'a': 23,
  'b': 100,
};

for (var MapEntry(key: key, value: count) in hist.entries) {
  print('$key occurred $count times');
}
```

The object pattern checks that `hist.entries` has the named type `MapEntry`, and then recurses into the named field subpatterns `key` and `value`. It calls the `key` getter and `value` getter on the `MapEntry` in each iteration, and binds the results to local variables `key` and `count`, respectively.

Binding the result of a getter call to a variable of the same name is a common use case, so object patterns can also infer the getter name from the variable subpattern. This allows you to simplify the variable pattern from something redundant like `key: key` to just `:key`:

```dart
for (var MapEntry(:key, value: count) in hist.entries) {
  print('$key occurred $count times');
}
```

Use cases for patterns

The previous section describes *how* patterns fit into other Dart code constructs. You saw some interesting use cases as examples, like swapping the values of two variables, or destructuring key-value pairs in a map. This section describes even more use cases, answering:

- *When and why* you might want to use patterns.
- What kinds of problems they solve.
- Which idioms they best suit.

Destructuring multiple returns

Records allow aggregating and returning multiple values from a single function call. Patterns add the ability to destructure a record's fields directly into local variables, inline with the function call.

Instead of individually declaring new local variables for each record field, like this:

```dart
var info = userInfo(json);
var name = info.$1;
var age = info.$2;
```

You can destructure the fields of a record that a function returns into local variables using a variable declaration or assigment pattern, and a record pattern as its subpattern:

```dart
var (name, age) = userInfo(json);
```

To destructure a record with named fields using a pattern:

```dart
final (:name, :age) =
    getData(); // For example, return (name: 'doug', age: 25);
```

Destructuring class instances

Object patterns match against named object types, allowing you to destructure their data using the getters the object's class already exposes.

To destructure an instance of a class, use the named type, followed by the properties to destructure enclosed in parentheses:

```dart
final Foo myFoo = Foo(one: 'one', two: 2);
var Foo(:one, :two) = myFoo;
print('one $one, two $two');
```

Algebraic data types

Object destructuring and switch cases are conducive to writing code in an algebraic data type style. Use this method when:

- You have a family of related types.
- You have an operation that needs specific behavior for each type.
- You want to group that behavior in one place instead of spreading it across all the different type definitions.

Instead of implementing the operation as an instance method for every type, keep the operation's variations in a single function that switches over the subtypes:

```dart
sealed class Shape {}

class Square implements Shape {
  final double length;
  Square(this.length);
}

class Circle implements Shape {
  final double radius;
  Circle(this.radius);
}

double calculateArea(Shape shape) => switch (shape) {
      Square(length: var l) => l * l,
      Circle(radius: var r) => math.pi * r * r
    };
```

Validating incoming JSON

Map and list patterns work well for destructuring key-value pairs in JSON data:

```dart
var json = {
  'user': ['Lily', 13]
};
var {'user': [name, age]} = json;
```

If you know that the JSON data has the structure you expect, the previous example is realistic. But data typically comes from an external source, like over the network. You need to validate it first to confirm its structure.

Without patterns, validation is verbose:

```dart
if (json is Map<String, Object?> &&
    json.length == 1 &&
    json.containsKey('user')) {
  var user = json['user'];
  if (user is List<Object> &&
      user.length == 2 &&
      user[0] is String &&
      user[1] is int) {
    var name = user[0] as String;
    var age = user[1] as int;
    print('User $name is $age years old.');
  }
}
```

A single case pattern can achieve the same validation. Single cases work best as if-case statements. Patterns provide a more declarative, and much less verbose method of validating JSON:

```dart
if (json case {'user': [String name, int age]}) {
  print('User $name is $age years old.');
}
```

This case pattern simultaneously validates that:

- `json` is a map, because it must first match the outer map pattern to proceed.
    - And, since it's a map, it also confirms `json` is not null.
- `json` contains a key `user`.
- The key `user` pairs with a list of two values.
- The types of the list values are `String` and `int`.
- The new local variables to hold the values are `name` and `age`.