Pattern types

This page is a reference for the different kinds of patterns. For an overview of how patterns work, where you can use them in Dart, and common use cases, visit the main Patterns page.

Pattern precedence

Similar to operator precedence, pattern evaluation adheres to precedence rules. You can use parenthesized patterns to evaluate lower-precedence patterns first.

This document lists the pattern types in ascending order of precedence:

- Logical-or patterns are lower-precedence than logical-and, logical-and patterns are lower-precedence than relational patterns, and so on.
- Post-fix unary patterns (cast, null-check, and null-assert) share the same level of precedence.
- The remaining primary patterns share the highest precedence. Collection-type (record, list, and map) and Object patterns encompass other data, so are evaluated first as outer-patterns.

Logical-or

`subpattern1 || subpattern2`

A logical-or pattern separates subpatterns by `||` and matches if any of the branches match. Branches are evaluated left-to-right. Once a branch matches, the rest are not evaluated.

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

Subpatterns in a logical-or pattern can bind variables, but the branches must define the same set of variables, because only one branch will be evaluated when the pattern matches.

Logical-and

`subpattern1 && subpattern2`

A pair of patterns separated by `&&` matches only if both subpatterns match. If the left branch does not match, the right branch is not evaluated.

Subpatterns in a logical-and pattern can bind variables, but the variables in each subpattern must not overlap, because they will both be bound if the pattern matches:

```dart
switch ((1, 2)) {
  // Error, both subpatterns attempt to bind 'b'.
  case (var a, var b) && (var b, var c): // ...
}
```

Relational

`== expression`

`< expression`

Relational patterns compare the matched value to a given constant using any of the equality or relational operators: `==`, `!=`, `<`, `>`, `<=`, and `>=`.

The pattern matches when calling the appropriate operator on the matched value with the constant as an argument returns `true`.

Relational patterns are useful for matching on numeric ranges, especially when combined with the logical-and pattern:

```dart
String asciiCharType(int char) {
  const space = 32;
  const zero = 48;
  const nine = 57;

  return switch (char) {
    < space => 'control',
    == space => 'space',
    > space && < zero => 'punctuation',
    >= zero && <= nine => 'digit',
    _ => ''
  };
}
```

Cast

`foo as String`

A cast pattern lets you insert a type cast in the middle of destructuring, before passing the value to another subpattern:

```dart
(num, Object) record = (1, 's');
var (i as int, s as String) = record;
```

Cast patterns will throw if the value doesn't have the stated type. Like the null-assert pattern, this lets you forcibly assert the expected type of some destructured value.

Null-check

`subpattern?`

Null-check patterns match first if the value is not null, and then match the inner pattern against that same value. They let you bind a variable whose type is the non-nullable base type of the nullable value being matched.

To treat `null` values as match failures without throwing, use the null-check pattern.

```dart
String? maybeString = 'nullable with base type String';
switch (maybeString) {
  case var s?:
  // 's' has type non-nullable String here.
}
```

To match when the value *is* null, use the constant pattern `null`.

Null-assert

`subpattern!`

Null-assert patterns match first if the object is not null, then on the value. They permit non-null values to flow through, but throw if the matched value is null.

To ensure `null` values are not silently treated as match failures, use a null-assert pattern while matching:

```dart
List<String?> row = ['user', null];
switch (row) {
  case ['user', var name!]: // ...
  // 'name' is a non-nullable string here.
}
```

To eliminate `null` values from variable declaration patterns, use the null-assert pattern:

```dart
(int?, int?) position = (2, 3);

var (x!, y!) = position;
```

To match when the value *is* null, use the constant pattern `null`.

Constant

`123, null, 'string', math.pi, SomeClass.constant, const Thing(1, 2), const (1 + 2)`

Constant patterns match when the value is equal to the constant:

```dart
switch (number) {
  // Matches if 1 == number.
  case 1: // ...
}
```

You can use simple literals and references to named constants directly as constant patterns:

- Number literals (`123`, `45.56`)
- Boolean literals (`true`)
- String literals (`'string'`)
- Named constants (`someConstant`, `math.pi`, `double.infinity`)
- Constant constructors (`const Point(0, 0)`)
- Constant collection literals (`const []`, `const {1, 2}`)

More complex constant expressions must be parenthesized and prefixed with `const` (`const (1 + 2)`):

```dart
// List or map pattern:
case [a, b]: // ...

// List or map literal:
case const [a, b]: // ...
```

Variable

`var bar, String str, final int _`

Variable patterns bind new variables to values that have been matched or destructured. They usually occur as part of a destructuring pattern to capture a destructured value.

The variables are in scope in a region of code that is only reachable when the pattern has matched.

```dart
switch ((1, 2)) {
  // 'var a' and 'var b' are variable patterns that bind to 1 and 2, respectively.
  case (var a, var b): // ...
  // 'a' and 'b' are in scope in the case body.
}
```

A *typed* variable pattern only matches if the matched value has the declared type, and fails otherwise:

```dart
switch ((1, 2)) {
  // Does not match.
  case (int a, String b): // ...
}
```

You can use a wildcard pattern as a variable pattern.

Identifier

`foo, _`

Identifier patterns may behave like a constant pattern or like a variable pattern, depending on the context where they appear:

- Declaration context: declares a new variable with identifier name: `var (a, b) = (1, 2);`
- Assignment context: assigns to existing variable with identifier name: `(a, b) = (3, 4);`
- Matching context: treated as a named constant pattern (unless its name is _):

```dart
const c = 1;
switch (2) {
  case c:
    print('match $c');
  default:
    print('no match'); // Prints "no match".
}
```

- Wildcard identifier in any context: matches any value and discards it: `case [_, var y, _]: print('The middle element is $y');`

Parenthesized

`(subpattern)`

Like parenthesized expressions, parentheses in a pattern let you control pattern precedence and insert a lower-precedence pattern where a higher precedence one is expected.

For example, imagine the boolean constants `x`, `y`, and `z` equal `true`, `true`, and `false`, respectively. Though the following example resembles boolean expression evaluation, the example matches patterns.

```dart
// ...
x || y => 'matches true',
x || y && z => 'matches true',
x || (y && z) => 'matches true',
// `x || y && z` is the same thing as `x || (y && z)`.
(x || y) && z => 'matches nothing',
// ...
```

Dart starts matching the pattern from left to right.

1. The first pattern matches `true` as `x` matches `true`.

2. The second pattern matches `true` as `x` matches `true`.

3. The third pattern matches `true` as `x` matches `true`.

4. The fourth pattern `(x || y) && z` has no match.

   - The `x` matches `true`, so Dart doesn't try to match `y`.
   - Though `(x || y)` matches `true`, `z` doesn't match `true`
   - Therefore, pattern `(x || y) && z` doesn't match `true`.
   - The subpattern `(x || y)` doesn't match `false`, so Dart doesn't try to match `z`.
   - Therefore, pattern `(x || y) && z` doesn't match `false`.
   - As a conclusion, `(x || y) && z` has no match.

List

`[subpattern1, subpattern2]`

A list pattern matches values that implement [`List`](), and then recursively matches its subpatterns against the list's elements to destructure them by position:

```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');
}
```

List patterns require that the number of elements in the pattern match the entire list. You can, however, use a [rest element]() as a place holder to account for any number of elements in a list.

Rest element

List patterns can contain *one* rest element (`...`) which allows matching lists of arbitrary lengths.

```dart
var [a, b, ..., c, d] = [1, 2, 3, 4, 5, 6, 7];
// Prints "1 2 6 7".
print('$a $b $c $d');
```

A rest element can also have a subpattern that collects elements that don't match the other subpatterns in the list, into a new list:

```dart
var [a, b, ...rest, c, d] = [1, 2, 3, 4, 5, 6, 7];
// Prints "1 2 [3, 4, 5] 6 7".
print('$a $b $rest $c $d');
```

Map

`{"key": subpattern1, someConst: subpattern2}`

Map patterns match values that implement [Map](), and then recursively match its subpatterns against the map's keys to destructure them.

Map patterns don't require the pattern to match the entire map. A map pattern ignores any keys that the map contains that aren't matched by the pattern.

Record

`(subpattern1, subpattern2)`

`(x: subpattern1, y: subpattern2)`

Record patterns match a record object and destructure its fields. If the value isn't a record with the same shape as the pattern, the match fails. Otherwise, the field subpatterns are matched against the corresponding fields in the record.

Record patterns require that the pattern match the entire record. To destructure a record with *named* fields using a pattern, include the field names in the pattern:

```dart
var (myString: foo, myNumber: bar) = (myString: 'string', myNumber: 1);
```

The getter name can be omitted and inferred from the variable pattern or identifier pattern in the field subpattern. These pairs of patterns are each equivalent:

```dart
// Record pattern with variable subpatterns:
var (untyped: untyped, typed: int typed) = record;
var (:untyped, :int typed) = record;

switch (record) {
  case (untyped: var untyped, typed: int typed): // ...
  case (:var untyped, :int typed): // ...
}

// Record pattern with null-check and null-assert subpatterns:
switch (record) {
  case (checked: var checked?, asserted: var asserted!): // ...
  case (:var checked?, :var asserted!): // ...
}

// Record pattern with cast subpattern:
var (untyped: untyped as int, typed: typed as String) = record;
var (:untyped as int, :typed as String) = record;
```

Object

`SomeClass(x: subpattern1, y: subpattern2)`

Object patterns check the matched value against a given named type to destructure data using getters on the object's properties. They are refuted if the value doesn't have the same type.

```dart
switch (shape) {
  // Matches if shape is of type Rect, and then against the properties of Rect.
  case Rect(width: var w, height: var h): // ...
}
```

The getter name can be omitted and inferred from the variable pattern or identifier pattern in the field subpattern:

```dart
// Binds new variables x and y to the values of Point's x and y properties.
var Point(:x, :y) = Point(1, 2);
```

Object patterns don't require the pattern to match the entire object. If an object has extra fields that the pattern doesn't destructure, it can still match.

Wildcard

`_`

A pattern named _ is a wildcard, either a variable pattern or identifier pattern, that doesn't bind or assign to any variable.

It's useful as a placeholder in places where you need a subpattern in order to destructure later positional values:

```dart
var list = [1, 2, 3];
var [_, two, _] = list;
```

A wildcard name with a type annotation is useful when you want to test a value's type but not bind the value to a name:

```dart
switch (record) {
  case (int _, String _):
    print('First field is int and second is String.');
}
```