Branches

This page shows how you can control the flow of your Dart code using branches:

- `if` statements and elements
- `if-case` statements and elements
- `switch` statements and expressions

You can also manipulate control flow in Dart using:

- Loops, like `for` and `while`
- Exceptions, like `try`, `catch`, and `throw`

If

Dart supports `if` statements with optional `else` clauses. The condition in parentheses after `if` must be an expression that evaluates to a boolean:

```dart
if (isRaining()) {
  you.bringRainCoat();
} else if (isSnowing()) {
  you.wearJacket();
} else {
  car.putTopDown();
}
```

To learn how to use `if` in an expression context, check out Conditional expressions.

If-case

Dart `if` statements support `case` clauses followed by a pattern:

```dart
if (pair case [int x, int y]) return Point(x, y);
```

If the pattern matches the value, then the branch executes with any variables the pattern defines in scope.

In the previous example, the list pattern `[int x, int y]` matches the value `pair`, so the branch `return Point(x, y)` executes with the variables that the pattern defined, `x` and `y`.

Otherwise, control flow progresses to the `else` branch to execute, if there is one:

```dart
if (pair case [int x, int y]) {
  print('Was coordinate array $x,$y');
} else {
  throw FormatException('Invalid coordinates.');
}
```

The if-case statement provides a way to match and destructure against a *single* pattern. To test a value against *multiple* patterns, use switch.

> 版本提示
>
> Case clauses in if statements require a language version of at least 3.0.

Switch statements

A `switch` statement evaluates a value expression against a series of cases. Each `case` clause is a pattern for the value to match against. You can use any kind of pattern for a case.

When the value matches a case's pattern, the case body executes. Non-empty `case` clauses jump to the end of the switch after completion. They do not require a `break` statement. Other valid ways to end a non-empty `case` clause are a `continue`, `throw`, or `return` statement.

Use a `default` or wildcard `_` clause to execute code when no `case` clause matches:

```dart
var command = 'OPEN';
switch (command) {
  case 'CLOSED':
    executeClosed();
  case 'PENDING':
    executePending();
  case 'APPROVED':
    executeApproved();
  case 'DENIED':
    executeDenied();
  case 'OPEN':
    executeOpen();
  default:
    executeUnknown();
}
```

Empty cases fall through to the next case, allowing cases to share a body. For an empty case that does not fall through, use `break` for its body. For non-sequential fall-through, you can use a `continue` statement and a label:

```dart
switch (command) {
  case 'OPEN':
    executeOpen();
    continue newCase; // Continues executing at the newCase label.

  case 'DENIED': // Empty case falls through.
  case 'CLOSED':
    executeClosed(); // Runs for both DENIED and CLOSED,

  newCase:
  case 'PENDING':
    executeNowClosed(); // Runs for both OPEN and PENDING.
}
```

You can use logical-or patterns to allow cases to share a body or a guard. To learn more about patterns and case clauses, check out the patterns documentation on Switch statements and expressions.

Switch expressions

A `switch` *expression* produces a value based on the expression body of whichever case matches. You can use a switch expression wherever Dart allows expressions, *except* at the start of an expression statement. For example:

```dart
var x = switch (y) { ... };

print(switch (x) { ... });

return switch (x) { ... };
```

If you want to use a switch at the start of an expression statement, use a switch statement.

Switch expressions allow you to rewrite a switch *statement* like this:

```dart
// Where slash, star, comma, semicolon, etc., are constant variables...
switch (charCode) {
  case slash || star || plus || minus: // Logical-or pattern
    token = operator(charCode);
  case comma || semicolon: // Logical-or pattern
    token = punctuation(charCode);
  case >= digit0 && <= digit9: // Relational and logical-and patterns
    token = number();
  default:
    throw FormatException('Invalid');
}
```

Into an *expression*, like this:

```dart
token = switch (charCode) {
  slash || star || plus || minus => operator(charCode),
  comma || semicolon => punctuation(charCode),
  >= digit0 && <= digit9 => number(),
  _ => throw FormatException('Invalid')
};
```

The syntax of a `switch` expression differs from `switch` statement syntax:

- Cases *do not* start with the `case` keyword.
- A case body is a single expression instead of a series of statements.
- Each case must have a body; there is no implicit fallthrough for empty cases.
- Case patterns are separated from their bodies using `=>` instead of `:`.
- Cases are separated by `,` (and an optional trailing `,` is allowed).
- Default cases can *only* use `_`, instead of allowing both `default` and `_`.

> **版本提示**
>
> Switch expressions require a language version of at least 3.0.

Exhaustiveness checking

Exhaustiveness checking is a feature that reports a compile-time error if it's possible for a value to enter a switch but not match any of the cases.

```dart
// Non-exhaustive switch on bool?, missing case to match null possibility:
switch (nullableBool) {
  case true:
    print('yes');
  case false:
    print('no');
}
```

A default case (`default` or `_`) covers all possible values that can flow through a switch. This makes a switch on any type exhaustive.

Enums and sealed types are particularly useful for switches because, even without a default case, their possible values are known and fully enumerable. Use the `sealed` modifier on a class to enable exhaustiveness checking when switching over subtypes of that class:

```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
    };
```

If anyone were to add a new subclass of `Shape`, this `switch` expression would be incomplete. Exhaustiveness checking would inform you of the missing subtype. This allows you to use Dart in a somewhat functional algebraic datatype style.

Guard clause

To set an optional guard clause after a `case` clause, use the keyword `when`. A guard clause can follow `if case`, and both `switch` statements and expressions.

```dart
// Switch statement:
switch (something) {
  case somePattern when some || boolean || expression:
    //             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ Guard clause.
    body;
}

// Switch expression:
var value = switch (something) {
  somePattern when some || boolean || expression => body,
  //             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ Guard clause.
}

// If-case statement:
if (something case somePattern when some || boolean || expression) {
  //                            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ Guard clause.
  body;
}
```

Guards evaluate an arbitrary boolean expression *after* matching. This allows you to add further constraints on whether a case body should execute. When the guard clause evaluates to false, execution proceeds to the next case rather than exiting the entire switch.