Operators

Dart supports the operators shown in the following table. The table shows Dart's operator associativity and operator precedence from highest to lowest, which are an **approximation** of Dart's operator relationships. You can implement many of these operators as class members.

Description	Operator	Associativity
unary postfix	`expr++ expr-- () [] ?[] . ?. !`	None
unary prefix	`-expr !expr ~expr ++expr --expr await expr`	None
multiplicative	`* / % ~/`	Left
additive	`+ -`	Left
shift	`<< >> >>>`	Left
bitwise AND	`&`	Left
bitwise XOR	`^`	Left
bitwise OR	`|`	Left
relational and type test	`>= > <= < as is is!`	None
equality	`== !=`	None
logical AND	`&&`	Left
logical OR	`||`	Left
if-null	`??`	Left
conditional	`expr1 ? expr2 : expr3`	Right
cascade	`.. ?..`	Left
assignment	`= *= /= += -= &= ^= etc.`	Right
spread (See note)	`... ...?`	None

> ⚠ 请注意
>
> The previous table should only be used as a helpful guide. The notion of operator precedence and associativity is an approximation of the truth found in the language grammar. You can find the authoritative behavior of Dart's operator relationships in the grammar defined in the Dart language specification.

When you use operators, you create expressions. Here are some examples of operator expressions:

```dart
a++
a + b
a = b
a == b
c ? a : b
a is T
```

Operator precedence example

In the operator table, each operator has higher precedence than the operators in the rows that follow it. For example, the multiplicative operator `%` has higher precedence than (and thus executes before) the equality operator `==`, which has higher precedence than the logical AND operator `&&`. That precedence means that the following two lines of code execute the same way:

```dart
// Parentheses improve readability.
if ((n % i == 0) && (d % i == 0)) ...

// Harder to read, but equivalent.
if (n % i == 0 && d % i == 0) ...
```

> ⚠ 请注意
>
> For operators that take two operands, the leftmost operand determines which method is used. For example, if you have a `Vector` object and a `Point` object, then `aVector + aPoint` uses `Vector` addition (`+`).

Arithmetic operators

Dart supports the usual arithmetic operators, as shown in the following table.

Operator	Meaning
`+`	Add
`-`	Subtract
`-expr`	Unary minus, also known as negation (reverse the sign of the expression)
`*`	Multiply
`/`	Divide
`~/`	Divide, returning an integer result
`%`	Get the remainder of an integer division (modulo)

Example:

```dart
assert(2 + 3 == 5);
assert(2 - 3 == -1);
assert(2 * 3 == 6);
assert(5 / 2 == 2.5); // Result is a double
assert(5 ~/ 2 == 2); // Result is an int
assert(5 % 2 == 1); // Remainder

assert('5/2 = ${5 ~/ 2} r ${5 % 2}' == '5/2 = 2 r 1');
```

Dart also supports both prefix and postfix increment and decrement operators.

Operator	Meaning
`++var`	`var = var + 1` (expression value is `var + 1`)
`var++`	`var = var + 1` (expression value is `var`)
`--var`	`var = var - 1` (expression value is `var - 1`)
`var--`	`var = var - 1` (expression value is `var`)

Example:

```dart
int a;
int b;

a = 0;
b = ++a; // Increment a before b gets its value.
assert(a == b); // 1 == 1

a = 0;
b = a++; // Increment a after b gets its value.
assert(a != b); // 1 != 0

a = 0;
b = --a; // Decrement a before b gets its value.
assert(a == b); // -1 == -1

a = 0;
b = a--; // Decrement a after b gets its value.
assert(a != b); // -1 != 0
```

Equality and relational operators

The following table lists the meanings of equality and relational operators.

Operator	Meaning
`==`	Equal; see discussion below
`!=`	Not equal
`>`	Greater than
`<`	Less than

Operator	Meaning
`>=`	Greater than or equal to
`<=`	Less than or equal to

To test whether two objects x and y represent the same thing, use the `==` operator. (In the rare case where you need to know whether two objects are the exact same object, use the identical() function instead.) Here's how the `==` operator works:

1. If *x* or *y* is null, return true if both are null, and false if only one is null.
2. Return the result of invoking the `==` method on *x* with the argument *y*. (That's right, operators such as `==` are methods that are invoked on their first operand. For details, see Operators.)

Here's an example of using each of the equality and relational operators:

```dart
assert(2 == 2);
assert(2 != 3);
assert(3 > 2);
assert(2 < 3);
assert(3 >= 3);
assert(2 <= 3);
```

Type test operators

The `as`, `is`, and `is!` operators are handy for checking types at runtime.

Operator	Meaning
`as`	Typecast (also used to specify library prefixes)
`is`	True if the object has the specified type
`is!`	True if the object doesn't have the specified type

The result of `obj is T` is true if `obj` implements the interface specified by `T`. For example, `obj is Object?` is always true.

Use the `as` operator to cast an object to a particular type if and only if you are sure that the object is of that type. Example:

```dart
(employee as Person).firstName = 'Bob';
```

If you aren't sure that the object is of type `T`, then use `is T` to check the type before using the object.

```dart
if (employee is Person) {
  // Type check
  employee.firstName = 'Bob';
}
```

> **提示**
>
> The code isn't equivalent. If `employee` is null or not a `Person`, the first example throws an exception; the second does nothing.

Assignment operators

As you've already seen, you can assign values using the `=` operator. To assign only if the assigned-to variable is null, use the `??=` operator.

```dart
// Assign value to a
a = value;
// Assign value to b if b is null; otherwise, b stays the same
b ??= value;
```

Compound assignment operators such as `+=` combine an operation with an assignment.

`=`	`*=`	`%=`	`>>>=`	`^=`
`+=`	`/=`	`<<=`	`&=`	`|=`
`-=`	`~/=`	`>>=`		

Here's how compound assignment operators work:

For an operator *op*:	`a op= b`	`a = a op b`
Example:	`a += b`	`a = a + b`

The following example uses assignment and compound assignment operators:

```dart
var a = 2; // Assign using =
a *= 3; // Assign and multiply: a = a * 3
assert(a == 6);
```

Logical operators

You can invert or combine boolean expressions using the logical operators.

Operator	Meaning
`!expr`	inverts the following expression (changes false to true, and vice versa)
`||`	logical OR
`&&`	logical AND

Here's an example of using the logical operators:

```dart
if (!done && (col == 0 || col == 3)) {
  // ...Do something...
}
```

Bitwise and shift operators

You can manipulate the individual bits of numbers in Dart. Usually, you'd use these bitwise and shift operators with integers.

Operator	Meaning
`&`	AND
`|`	OR
`^`	XOR

Operator	Meaning
`~expr`	Unary bitwise complement (0s become 1s; 1s become 0s)
`<<`	Shift left
`>>`	Shift right
`>>>`	Unsigned shift right

> ⓘ 提示
>
> The behavior of bitwise operations with large or negative operands might differ between platforms. To learn more, check out Bitwise operations platform differences.

Here's an example of using bitwise and shift operators:

```dart
final value = 0x22;
final bitmask = 0x0f;

assert((value & bitmask) == 0x02); // AND
assert((value & ~bitmask) == 0x20); // AND NOT
assert((value | bitmask) == 0x2f); // OR
assert((value ^ bitmask) == 0x2d); // XOR

assert((value << 4) == 0x220); // Shift left
assert((value >> 4) == 0x02); // Shift right

// Shift right example that results in different behavior on web
// because the operand value changes when masked to 32 bits:
assert((-value >> 4) == -0x03);

assert((value >>> 4) == 0x02); // Unsigned shift right
assert((-value >>> 4) > 0); // Unsigned shift right
```

> ↑ 版本提示
>
> The `>>>` operator (known as *triple-shift* or *unsigned shift*) requires a language version of at least 2.14.

Conditional expressions

Dart has two operators that let you concisely evaluate expressions that might otherwise require if-else statements:

`condition ? expr1 : expr2`
If *condition* is true, evaluates *expr1* (and returns its value); otherwise, evaluates and returns the value of *expr2*.

`expr1 ?? expr2`
If *expr1* is non-null, returns its value; otherwise, evaluates and returns the value of *expr2*.

When you need to assign a value based on a boolean expression, consider using the conditional operator `?` and `:`.

```dart
var visibility = isPublic ? 'public' : 'private';
```

If the boolean expression tests for null, consider using the if-null operator `??` (also known as the null-coalescing operator).

```dart
String playerName(String? name) => name ?? 'Guest';
```

The previous example could have been written at least two other ways, but not as succinctly:

```dart
// Slightly longer version uses ?: operator.
String playerName(String? name) => name != null ? name : 'Guest';

// Very long version uses if-else statement.
String playerName(String? name) {
  if (name != null) {
    return name;
  } else {
    return 'Guest';
  }
}
```

Cascade notation

Cascades (`..`, `?..`) allow you to make a sequence of operations on the same object. In addition to accessing instance members, you can also call instance methods on that same object. This often saves you the step of creating a temporary variable and allows you to write more fluid code.

Consider the following code:

```dart
var paint = Paint()
  ..color = Colors.black
  ..strokeCap = StrokeCap.round
  ..strokeWidth = 5.0;
```

The constructor, `Paint()`, returns a `Paint` object. The code that follows the cascade notation operates on this object, ignoring any values that might be returned.

The previous example is equivalent to this code:

```dart
var paint = Paint();
paint.color = Colors.black;
paint.strokeCap = StrokeCap.round;
paint.strokeWidth = 5.0;
```

If the object that the cascade operates on can be null, then use a *null-shorting* cascade (`?..`) for the first operation. Starting with `?..` guarantees that none of the cascade operations are attempted on that null object.

```dart
querySelector('#confirm') // Get an object.
  ?..text = 'Confirm' // Use its members.
  ..classes.add('important')
  ..onClick.listen((e) => window.alert('Confirmed!'))
  ..scrollIntoView();
```

> ↥ 版本提示
>
> The `?..` syntax requires a language version of at least 2.12.

The previous code is equivalent to the following:

```dart
var button = querySelector('#confirm');
button?.text = 'Confirm';
button?.classes.add('important');
button?.onClick.listen((e) => window.alert('Confirmed!'));
button?.scrollIntoView();
```

You can also nest cascades. For example:

```dart
final addressBook = (AddressBookBuilder()
      ..name = 'jenny'
      ..email = 'jenny@example.com'
      ..phone = (PhoneNumberBuilder()
            ..number = '415-555-0100'
            ..label = 'home')
          .build())
    .build();
```

Be careful to construct your cascade on a function that returns an actual object. For example, the following code fails:

```dart
var sb = StringBuffer();
sb.write('foo')
  ..write('bar'); // Error: method 'write' isn't defined for 'void'.
```

The `sb.write()` call returns void, and you can't construct a cascade on `void`.

> ⓘ 提示
>
> Strictly speaking, the "double dot" notation for cascades isn't an operator. It's just part of the Dart syntax.

Spread operators

Spread operators evaluate an expression that yields a collection, unpacks the resulting values, and inserts them into another collection.

The spread operator isn't actually an operator expression. The `...`/`...?` syntax is part of the collection literal itself. So, you can learn more about spread operators on the Collections page.

Because it isn't an operator, the syntax doesn't have any "operator precedence". Effectively, it has the lowest "precedence" — any kind of expression is valid as the spread target, such as:

```dart
[...a + b]
```

Other operators

You've seen most of the remaining operators in other examples:

Operator	Name	Meaning
`()`	Function application	Represents a function call
`[]`	Subscript access	Represents a call to the overridable `[]` operator; example: `fooList[1]` passes the int `1` to `fooList` to access the element at index `1`

Operator	Name	Meaning
`?[]`	Conditional subscript access	Like `[]`, but the leftmost operand can be null; example: `fooList?[1]` passes the int `1` to `fooList` to access the element at index `1` unless `fooList` is null (in which case the expression evaluates to null)
`.`	Member access	Refers to a property of an expression; example: `foo.bar` selects property `bar` from expression `foo`
`?.`	Conditional member access	Like `.`, but the leftmost operand can be null; example: `foo?.bar` selects property `bar` from expression `foo` unless `foo` is null (in which case the value of `foo?.bar` is null)
`!`	Non-null assertion operator	Casts an expression to its underlying non-nullable type, throwing a runtime exception if the cast fails; example: `foo!.bar` asserts `foo` is non-null and selects the property `bar`, unless `foo` is null in which case a runtime exception is thrown

For more information about the `.`, `?.`, and `..` operators, see Classes.