# Records

> ⅄ 版本提示
>
> Records require a language version of at least 3.0.

Records are an anonymous, immutable, aggregate type. Like other collection types, they let you bundle multiple objects into a single object. Unlike other collection types, records are fixed-sized, heterogeneous, and typed.

Records are real values; you can store them in variables, nest them, pass them to and from functions, and store them in data structures such as lists, maps, and sets.

## Record syntax

*Records expressions* are comma-delimited lists of named or positional fields, enclosed in parentheses:

```dart
var record = ('first', a: 2, b: true, 'last');
```

*Record type annotations* are comma-delimited lists of types enclosed in parentheses. You can use record type annotations to define return types and parameter types. For example, the following `(int, int)` statements are record type annotations:

```dart
(int, int) swap((int, int) record) {
  var (a, b) = record;
  return (b, a);
}
```

Fields in record expressions and type annotations mirror how parameters and arguments work in functions. Positional fields go directly inside the parentheses:

```dart
// Record type annotation in a variable declaration:
(String, int) record;

// Initialize it with a record expression:
record = ('A string', 123);
```

In a record type annotation, named fields go inside a curly brace-delimited section of type-and-name pairs, after all positional fields. In a record expression, the names go before each field value with a colon after:

```dart
// Record type annotation in a variable declaration:
({int a, bool b}) record;

// Initialize it with a record expression:
record = (a: 123, b: true);
```

The names of named fields in a record type are part of the record's type definition, or its *shape*. Two records with named fields with different names have different types:

```dart
({int a, int b}) recordAB = (a: 1, b: 2);
({int x, int y}) recordXY = (x: 3, y: 4);

// Compile error! These records don't have the same type.
// recordAB = recordXY;
```

In a record type annotation, you can also name the *positional* fields, but these names are purely for documentation and don't affect the record's type:

```dart
(int a, int b) recordAB = (1, 2);
(int x, int y) recordXY = (3, 4);

recordAB = recordXY; // OK.
```

This is similar to how positional parameters in a function declaration or function typedef can have names but those names don't affect the signature of the function.

For more information and examples, check out Record types and Record equality.

## Record fields

Record fields are accessible through built-in getters. Records are immutable, so fields do not have setters.

Named fields expose getters of the same name. Positional fields expose getters of the name `$<position>`, skipping named fields:

```dart
var record = ('first', a: 2, b: true, 'last');

print(record.$1); // Prints 'first'
print(record.a); // Prints 2
print(record.b); // Prints true
print(record.$2); // Prints 'last'
```

To streamline record field access even more, check out the page on Patterns.

## Record types

There is no type declaration for individual record types. Records are structurally typed based on the types of their fields. A record's *shape* (the set of its fields, the fields' types, and their names, if any) uniquely determines the type of a record.

Each field in a record has its own type. Field types can differ within the same record. The type system is aware of each field's type wherever it is accessed from the record:

```dart
(num, Object) pair = (42, 'a');

var first = pair.$1; // Static type `num`, runtime type `int`.
var second = pair.$2; // Static type `Object`, runtime type `String`.
```

Consider two unrelated libraries that create records with the same set of fields. The type system understands that those records are the same type even though the libraries are not coupled to each other.

## Record equality

Two records are equal if they have the same *shape* (set of fields), and their corresponding fields have the same values. Since named field *order* is not part of a record's shape, the order of named fields does not affect equality.

For example:

```dart
(int x, int y, int z) point = (1, 2, 3);
(int r, int g, int b) color = (1, 2, 3);

print(point == color); // Prints 'true'.
```

```dart
({int x, int y, int z}) point = (x: 1, y: 2, z: 3);
({int r, int g, int b}) color = (r: 1, g: 2, b: 3);

print(point == color); // Prints 'false'. Lint: Equals on unrelated types.
```

Records automatically define `hashCode` and `==` methods based on the structure of their fields.

## Multiple returns

Records allow functions to return multiple values bundled together. To retrieve record values from a return, destructure the values into local variables using pattern matching.

```dart
// Returns multiple values in a record:
(String name, int age) userInfo(Map<String, dynamic> json) {
  return (json['name'] as String, json['age'] as int);
}

final json = <String, dynamic>{
  'name': 'Dash',
  'age': 10,
  'color': 'blue',
};

// Destructures using a record pattern with positional fields:
var (name, age) = userInfo(json);

/* Equivalent to:
  var info = userInfo(json);
  var name = info.$1;
  var age  = info.$2;
*/
```

You can also destructure a record using its named fields, using the colon `:` syntax, which you can read more about on the Pattern types page:

```dart
({String name, int age}) userInfo(Map<String, dynamic> json)
// ...
// Destructures using a record pattern with named fields:
final (:name, :age) = userInfo(json);
```

You can return multiple values from a function without records, but other methods come with downsides. For example, creating a class is much more verbose, and using other collection types like `List` or `Map` loses type safety.

> ⓘ 提示

Records' multiple-return and heterogeneous-type characteristics enable parallelization of futures of different types, which you can read about in the `dart:async` documentation.