Built-in types

The Dart language has special support for the following:

- Numbers (`int`, `double`)
- Strings (`String`)
- Booleans (`bool`)
- Records (`(value1, value2)`)
- Functions (`Function`)
- Lists (`List`, also known as *arrays*)
- Sets (`Set`)
- Maps (`Map`)
- Runes (`Runes`; often replaced by the `characters` API)
- Symbols (`Symbol`)
- The value `null` (`Null`)

This support includes the ability to create objects using literals. For example, `'this is a string'` is a string literal, and `true` is a boolean literal.

Because every variable in Dart refers to an object—an instance of a *class*—you can usually use *constructors* to initialize variables. Some of the built-in types have their own constructors. For example, you can use the `Map()` constructor to create a map.

Some other types also have special roles in the Dart language:

- `Object`: The superclass of all Dart classes except `Null`.
- `Enum`: The superclass of all enums.
- `Future` and `Stream`: Used in asynchrony support.
- `Iterable`: Used in for-in loops and in synchronous generator functions.
- `Never`: Indicates that an expression can never successfully finish evaluating. Most often used for functions that always throw an exception.
- `dynamic`: Indicates that you want to disable static checking. Usually you should use `Object` or `Object?` instead.
- `void`: Indicates that a value is never used. Often used as a return type.

The `Object`, `Object?`, `Null`, and `Never` classes have special roles in the class hierarchy. Learn about these roles in Understanding null safety.

Numbers

Dart numbers come in two flavors:

`int`
Integer values no larger than 64 bits, depending on the platform. On native platforms, values can be from -2^{63} to $2^{63} - 1$. On the web, integer values are represented as JavaScript numbers (64-bit floating-point values with no fractional part) and can be from -2^{53} to $2^{53} - 1$.

`double`
64-bit (double-precision) floating-point numbers, as specified by the IEEE 754 standard.

Both `int` and `double` are subtypes of `num`. The num type includes basic operators such as +, -, /, and *, and is also where you'll find `abs()`, `ceil()`, and `floor()`, among other methods. (Bitwise operators, such as >>, are defined in the `int` class.) If num and its subtypes don't have what you're looking for, the `dart:math` library might.

Integers are numbers without a decimal point. Here are some examples of defining integer literals:

```dart
var x = 1;
var hex = 0xDEADBEEF;
```

If a number includes a decimal, it is a double. Here are some examples of defining double literals:

```dart
var y = 1.1;
var exponents = 1.42e5;
```

You can also declare a variable as a num. If you do this, the variable can have both integer and double values.

```dart
num x = 1; // x can have both int and double values
x += 2.5;
```

Integer literals are automatically converted to doubles when necessary:

```dart
double z = 1; // Equivalent to double z = 1.0.
```

Here's how you turn a string into a number, or vice versa:

```dart
// String -> int
var one = int.parse('1');
assert(one == 1);

// String -> double
var onePointOne = double.parse('1.1');
assert(onePointOne == 1.1);

// int -> String
String oneAsString = 1.toString();
assert(oneAsString == '1');

// double -> String
String piAsString = 3.14159.toStringAsFixed(2);
assert(piAsString == '3.14');
```

The `int` type specifies the traditional bitwise shift (`<<`, `>>`, `>>>`), complement (`~`), AND (`&`), OR (`|`), and XOR (`^`) operators, which are useful for manipulating and masking flags in bit fields. For example:

```dart
assert((3 << 1) == 6); // 0011 << 1 == 0110
assert((3 | 4) == 7); // 0011 | 0100 == 0111
assert((3 & 4) == 0); // 0011 & 0100 == 0000
```

For more examples, see the bitwise and shift operator section.

Number literals are compile-time constants. Many arithmetic expressions are also compile-time constants, as long as their operands are compile-time constants that evaluate to numbers.

```dart
const msPerSecond = 1000;
const secondsUntilRetry = 5;
const msUntilRetry = secondsUntilRetry * msPerSecond;
```

For more information, see Numbers in Dart.

You can use one or more underscores (`_`) as digit separators to make long number literals more readable. Multiple digit separators allow for higher level grouping.

```dart
var n1 = 1_000_000;
var n2 = 0.000_000_000_01;
var n3 = 0x00_14_22_01_23_45;  // MAC address
var n4 = 555_123_4567;  // US Phone number
var n5 = 100__000_000__000_000;  // one hundred million million!
```

版本提示

Using digit separators requires a language version of at least 3.6.0.

Strings

A Dart string (`String` object) holds a sequence of UTF-16 code units. You can use either single or double quotes to create a string:

```dart
var s1 = 'Single quotes work well for string literals.';
var s2 = "Double quotes work just as well.";
var s3 = 'It\'s easy to escape the string delimiter.';
var s4 = "It's even easier to use the other delimiter.";
```

You can put the value of an expression inside a string by using `${`*`expression`*`}`. If the expression is an identifier, you can skip the `{}`. To get the string corresponding to an object, Dart calls the object's `toString()` method.

```dart
var s = 'string interpolation';

assert('Dart has $s, which is very handy.' ==
    'Dart has string interpolation, '
        'which is very handy.');
assert('That deserves all caps. '
        '${s.toUpperCase()} is very handy!' ==
    'That deserves all caps. '
        'STRING INTERPOLATION is very handy!');
```

提示

The `==` operator tests whether two objects are equivalent. Two strings are equivalent if they contain the same sequence of code units.

You can concatenate strings using adjacent string literals or the `+` operator:

```dart
var s1 = 'String '
    'concatenation'
    " works even over line breaks.";
assert(s1 ==
    'String concatenation works even over '
        'line breaks.');

var s2 = 'The + operator ' + 'works, as well.';
assert(s2 == 'The + operator works, as well.');
```

To create a multi-line string, use a triple quote with either single or double quotation marks:

```dart
var s1 = '''
You can create
multi-line strings like this one.
''';

var s2 = """This is also a
multi-line string.""";
```

You can create a "raw" string by prefixing it with `r`:

```dart
var s = r'In a raw string, not even \n gets special treatment.';
```

See Runes and grapheme clusters for details on how to express Unicode characters in a string.

String literals are compile-time constants, as long as any interpolated expression is a compile-time constant that evaluates to null or a numeric, string, or boolean value.

```dart
// These work in a const string.
const aConstNum = 0;
const aConstBool = true;
const aConstString = 'a constant string';

// These do NOT work in a const string.
var aNum = 0;
var aBool = true;
var aString = 'a string';
const aConstList = [1, 2, 3];

const validConstString = '$aConstNum $aConstBool $aConstString';
// const invalidConstString = '$aNum $aBool $aString $aConstList';
```

For more information on using strings, check out Strings and regular expressions.

Booleans

To represent boolean values, Dart has a type named `bool`. Only two objects have type bool: the boolean literals `true` and `false`, which are both compile-time constants.

Dart's type safety means that you can't use code like `if (`*`nonbooleanValue`*`)` or `assert (`*`nonbooleanValue`*`)`. Instead, explicitly check for values, like this:

dart

```dart
// Check for an empty string.
var fullName = '';
assert(fullName.isEmpty);

// Check for zero.
var hitPoints = 0;
assert(hitPoints == 0);

// Check for null.
var unicorn = null;
assert(unicorn == null);

// Check for NaN.
var iMeantToDoThis = 0 / 0;
assert(iMeantToDoThis.isNaN);
```

Runes and grapheme clusters

In Dart, runes expose the Unicode code points of a string. You can use the characters package to view or manipulate user-perceived characters, also known as Unicode (extended) grapheme clusters.

Unicode defines a unique numeric value for each letter, digit, and symbol used in all of the world's writing systems. Because a Dart string is a sequence of UTF-16 code units, expressing Unicode code points within a string requires special syntax. The usual way to express a Unicode code point is `\uXXXX`, where XXXX is a 4-digit hexadecimal value. For example, the heart character (♥) is `\u2665`. To specify more or less than 4 hex digits, place the value in curly brackets. For example, the laughing emoji (😆) is `\u{1f606}`.

If you need to read or write individual Unicode characters, use the `characters` getter defined on String by the characters package. The returned `Characters` object is the string as a sequence of grapheme clusters. Here's an example of using the characters API:

dart

```dart
import 'package:characters/characters.dart';

void main() {
  var hi = 'Hi DK';
  print(hi);
  print('The end of the string: ${hi.substring(hi.length - 1)}');
  print('The last character: ${hi.characters.last}');
}
```

The output, depending on your environment, looks something like this:

```
$ dart run bin/main.dart
Hi DK
The end of the string: ???
The last character: DK
```

For details on using the characters package to manipulate strings, see the example and API reference for the characters package.

Symbols

A `Symbol` object represents an operator or identifier declared in a Dart program. You might never need to use symbols, but they're invaluable for APIs that refer to identifiers by name, because minification changes identifier names but not identifier symbols.

To get the symbol for an identifier, use a symbol literal, which is just `#` followed by the identifier:

```
#radix
#bar
```

Symbol literals are compile-time constants.