Loops

Computer programming is just as full of repetitive actions as your life is. The way you can accomplish them are by using **loops**. Dart, like many programming languages, has `while` loops and `for` loops. You'll learn how to make them in the following sections.

While loops

A **while loop** repeats a block of code as long as a Boolean condition is true. You create a while loop like so:

```
while (condition) {
// loop code
}
```

The loop checks the condition on every iteration. If the condition is `true`, then the loop executes and moves on to another iteration. If the condition is `false`, then the loop stops. Just like `if` statements, `while` loops introduce a scope because of their curly braces.

The simplest `while` loop takes this form:

```
while (true) { }
```

This is a `while` loop that never ends, because the condition is always `true`. Of course, you would never write such a `while` loop, because your program would spin forever! This situation is known as an **infinite loop**, and while it might not cause your program to crash, it will very likely cause your

computer to freeze.

Here's a (somewhat) more useful example of a `while` loop:

```
var sum = 1;
while (sum < 10) {
sum += 4;
print(sum);
}
```

Run that to see the result. The loop executes as follows:

- Before 1st iteration: sum = 1, loop condition = true
- After 1st iteration: sum = 5, loop condition = true After 2nd iteration: sum = 9, loop condition = true After 3rd iteration: sum = 13, loop condition = false

After the third iteration, the sum variable is 13, and therefore the loop condition of `sum < 10` becomes `false`. At this point, the loop stops.

Do-while loops

A variant of the `while` loop is called the `do-while` loop. It differs from the `while` loop in that the condition is evaluated at the *end* of the loop rather than at the beginning. Thus, the body of a `do-while` loop is always executed at least once.

You construct a `do-while` loop like this:

```
do {
// loop code
} while (condition)
```

Whatever statements appear inside the braces will be executed. Finally, if the `while` condition after the closing brace is true, you jump back up to the beginning and repeat the loop.

Here's the example from the last section, but using a `do-while` loop:

```
sum = 1; do
{
sum += 4;
print(sum);
} while (sum < 10);
```

In this example, the outcome is the same as before.

Comparing while and do-while loops

It isn't always the case that `while` loops and `do-while` loops will give the same result. For example, here's a `while` loop where sum starts at 11:

```
sum = 11;
while (sum < 10) {
sum += 4;
}
```

Since the initial condition is false, the loop never executes. Run that code and you'll see that sum remains 11.

On the other hand, check out a similar do-while loop:

```
sum = 11;
do {
sum += 4;
} while (sum < 10);
print(sum);
```

Run this and you'll find the sum at the end to be 15. This is because the do-while loop executed the body of the loop

before checking the condition.

Breaking out of a loop

Sometimes you'll need to break out of a loop early. You can do this using the break statement, just as you did from inside the switch statement earlier. This immediately stops the execution of the loop and continues on to the code that follows the loop.

For example, consider the following while loop:

```
sum = 1;
while (true) {
sum += 4;
if (sum > 10) {
break;
}
```

Here, the loop condition is `true`, so the loop would normally iterate forever. However, the `break` means the `while` loop will exit once the sum is greater than 10.

You've now seen how to write the same loop in different ways. This demonstrates that in computer programming there are often many ways to achieve the same result. You should choose the method that's easiest to read and that conveys your intent in the best way possible. This is an approach you'll internalize with enough time and practice.

A random interlude

common need in programming is to be able to generate random numbers. And Dart provides this functionality in the `dart:math` library, which is pretty handy!

As an example, imagine an application that needs to simulate rolling a die. You may want to do something in your code until a six is rolled, and then stop. Now that you know about `while` loops, you can do that with the `Random` feature.

First import the `dart:math` library at the top of your Dart file:

```
import 'dart:math';
```

Then create the `while` loop like so:

```
final random = Random();
while (random.nextInt(6) + 1 != 6) {
print('Not a six!');
}
```

Random is a class to help with random numbers, and nextInt is a method that generates a random integer between 0 and one less than the maximum value you give it, in this case, 6. Since you want a number between 1 and 6, not 0 to 5, you must add 1 to the random number in the while loop condition.

> **Note** : You'll learn more about classes and methods in Chapter 6.

Run the loop and you'll get a variable number of outputs:

```
Not a six!
Not a six!

Finally, you got a six!
```

In this case it was only two loops before a lucky six was rolled. You probably had a different number of rolls, though.

For loops

In addition to while loops, Dart has another type of loop called a **for loop**. This is probably the most common loop you'll see, and you use it to run a block of code a set number of times. In this section you'll learn about C-style for loops, and in the next section, about for-in loops.

Here's a simple example of a **C-style for loop** in Dart:

```
for (var i = 0; i < 5; i++) {
print(i);
}
```

Run the code above and you'll see the following output:

```
0
1
2
3
```

The counter index `i` started at `0` and continued until it equaled `5`. At that point the `for` loop condition `i < 5` was no longer `true`, so the loop exited before running the `print` statement again.

The continue keyword

Sometimes you want to skip an iteration only for a certain condition. You can do that using the `continue` keyword. Have a look at the following example:

```
for (var i = 0; i < 5; i++) { if
(i == 2) {
continue;
}
print(i);
}
```

This example is similar to the last one, but this time, when `i` is `2`, the `continue` keyword will tell the `for` loop to immediately go on to the next iteration. The rest of the code in the block won't run on this iteration.

This is what you'll see:

```
0
1
3
```

No `2` here!

For-in loops

There's another type of `for` loop that has simpler syntax; it's called a `for-in loop`. It doesn't have any sort of index or counter variable associated with it, but it makes iterating over a collection very convenient.

You haven't formally learned about collections yet in this book; you'll get to them in Chapter 8. Dart collections aren't that difficult to learn, though, especially if you're familiar with them from other languages. In fact, Chapter 3 already snuck some in. Remember? Strings are a collection of characters.

When you get the runes from a string, that gives you a collection of Unicode code points. You can use that knowledge now to loop over the code points in a string like so:

```
const myString = 'I ♥ Dart';

for (var codePoint in myString.runes) {
print(String.fromCharCode(codePoint));

}
```

Here's what's happening:

`myString.runes` is a collection of all the code points in `myString`.

The `in` keyword tells the `for-in` loop to iterate over the collection in order, and on each iteration, to assign the current code point to the `codePoint` variable. Since `runes` is a collection of integers, `codePoint` is inferred to be an `int`.

Inside the braces you use `String.fromCharCode` to convert the code point integer back into a string.

In terms of scope, the `codePoint` variable is only visible inside the scope of the `for-in` loop, which means it's not available outside of the loop.

Run the code and you'll see the following output:

```
I


♥D
a
r t
```

For-each loops

You can sometimes simplify for-in loops even more with the forEach method that is available to collections.

Even though you haven't learned about Dart collections in depth yet, here's another one for you:

```
const myNumbers = [1, 2, 3];
```

This is a comma-separated list of integers surrounded by square brackets.

Loop through each of the elements in that list by using

```
myNumbers.forEach((number)  =>  print(number));
```

forEach like so:

The part inside the forEach parentheses is a function, where => is **arrow syntax** that points to the statement that the function runs.

It has exactly the same meaning as the following, which uses { } braces instead of arrow syntax:

```
myNumbers.forEach((number) { print(number);
});
```

You might be wondering why number isn't declared anywhere. That's because Dart automatically gives number the type that's inside the collection; in this case, int.

Don't worry if this still looks strange to you. You'll learn all about functions in the next chapter. Consider this a sneak preview.

Run either of the `forEach` examples above and you'll see the same results:

```
1
2
3
```

Mini-exercises

1. Create a variable named `counter` and set it equal to `0`. Create a `while` loop with the condition `counter < 10`. The loop body should print out "counter is X" (where X is replaced with the value of `counter`) and then increment `counter` by 1.

2. Write a `for` loop starting at `1` and ending with `10` inclusive. Print the square of each number.

3. Write a `for-in` loop to iterate over the following collection of numbers. Print the square root of each number.

```
const numbers = [1, 2, 4, 7];
```

4. Repeat Mini-exercise 3 using a `forEach` loop.

Challenges

Before moving on, here are some challenges to test your knowledge of control flow. It's best if you try to solve them

yourself, but solutions are available in the challenge folder if you get stuck.

Challenge 1: Find the error

What's wrong with the following code?

```
const firstName = "Bob"; if
(firstName == "Bob") {
const lastName = "Smith";
} else if (firstName == "Ray") {
const lastName = "Wenderlich";
}
```

Challenge 2: Boolean challenge

In each of the following statements, what is the value of the Boolean expression?

```
true && true false
|| false
(true && 1 != 2) || (4 > 3 && 100 < 1)
((10 / 2) > 3) && ((10 % 2) == 0)
```

Challenge 3: Next power of two

Given a number, determine the next power of two above or equal to that number. Powers of two are the numbers in the sequence of 2^1, 2^2, 2^3, and so on. You may also recognize the series as 1, 2, 4, 8, 16, 32, 64...

Challenge 4: Fibonacci

Calculate the *n*th Fibonacci number. The Fibonacci sequence starts with 1, then 1 again, and then all subsequent numbers in the sequence are simply the previous two values in the sequence added together (1, 1, 2, 3, 5, 8...). You can get a refresher here:

Challenge 5: How many times?

In the following `for` loop, what will be the value of `sum`, and how many iterations will happen?

```
var sum = 0;
for (var i = 0; i <= 5; i++) {
sum += i;
}
```

Challenge 6: The final countdown

Print a countdown from 10 to 0.

Challenge 7: Print a sequence

Print the sequence 0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0.

Key points

You use the Boolean data type `bool` to represent `true` and `false`.

The comparison operators, all of which return a Boolean, are:

Name	Operator
Equal	==
Not equal	!=
Less than	<
Greater than	>
Less than or equal	<=
Greater than or equal	>=

You can use Boolean logic (`&&` and `||`) to combine comparison conditions.

You use `if` statements to make simple decisions based on a condition.

You use `else` and `else-if` within an `if` statement to extend the decision making beyond a single condition.

Variables and constants belong to a certain scope, beyond which you cannot use them. A scope inherits variables and constants from its parent.

You can use the ternary operator (`a ? b : c`) in place of simple `if` statements.

You can use `switch` statements to decide which code to run depending on the value of a variable or constant.

Enumerated types define a new type with a finite list of distinct values.

`while` loops perform a certain task repeatedly as long as a condition is true.

`do-while` loops always execute the loop at least once. The

`break` statement lets you break out of a loop.

`for` loops allow you to perform a loop a set number of times.

The `continue` statement ends the current iteration of a loop and begins the next iteration.