Chapter 4: Control Flow

In computer programming terms, this concept is known as **control flow**, because you can control the flow of decisions the code makes at multiple points. In this chapter, you'll learn how to make decisions and repeat tasks in your programs.

Making comparisons

You've already encountered a few different Dart types, such as `int`, `double` and `String`. Each of those types is a data structure which is designed to hold a particular type of data. The `int` type is for whole numbers while the `double` type is for decimal numbers. `String`, by comparison, is useful for storing textual information.

Boolean values

Dart has a data type just for this. It's called `bool`, which is short for **Boolean**. A Boolean value can have one of two states. While in general you could refer to the states as yes and no, on and off, or 1 and 0, most programming languages, Dart included, call them **true** and **false**

To start your exploration of Booleans in Dart, create some Boolean variables like so:

```
const bool yes = true;
const bool no = false;
```

Because of Dart's type inference, you can leave off the type annotation:

```
const yes = true;
const no = false;
```

In the code above, you use the keywords `true` and `false` to set the state of each Boolean constant.

Boolean operators

Booleans are commonly used to compare values. For example, you may have two values and you want to know if they're equal. Either they *are* equal, which would be `true`, or they *aren't* equal, which would be `false`.

Next you'll see how to make that comparison in Dart.

Testing equality

You can test for equality using the **equality operator**, which is denoted by `==`, that is, two equals signs.

Write the following line:

```
const doesOneEqualTwo = (1 == 2);
```

Dart infers that `doesOneEqualTwo` is a `bool`. Clearly, `1` does not equal `2`, and therefore `doesOneEqualTwo` will be `false`. Confirm that result by printing the value:

```
print(doesOneEqualTwo);
```

Sometimes you need parentheses to tell Dart what should happen first. However, the parentheses in that last example were there only for readability, that is, to show you that the two objects being compared were `1` and `2`. You could have also written it like so:

```
const doesOneEqualTwo = 1 == 2;
```

Note: You may use the equality operator to compare `int` to `double`, since they both belong to the `num` type.

Testing inequality

You can also find out if two values are *not* equal using the `!=` operator:

```
const doesOneNotEqualTwo = (1 != 2);
```

This time, the result of the comparison is `true` because `1` does not equal `2`, so `doesOneNotEqualTwo` will be `true`.

The prefix `!` operator, also called the **not-operator** **bang operator**, toggles `true` to `false` and `false` to `true`. Another way to write the above is:

```
const alsoTrue = !(1 == 2);
```

Because 1 does not equal 2, (1 == 2) is false, and then ! flips it to true.

Testing greater and less than

There are two other operators to help you compare two values and determine if a value is greater than (>) or less than (<) another value. You know these from mathematics:

```
const isOneGreaterThanTwo = (1 > 2);
const isOneLessThanTwo = (1 < 2);
```

It's not rocket science to work out that `isOneGreaterThanTwo` will equal `false` and that `isOneLessThanTwo` will equal `true`.

The <= operator lets you test if a value is *less than or equal to* another value. It's a combination of < and ==, and will therefore return `true` if the first value is less than, or equal to, the second value.

```
print(1 <= 2); // true
print(2 <= 2); // true
```

Similarly, the >= operator lets you test if a value is *greater than or equal to* another value.

```
print(2 >= 1); // true
print(2 >= 2); // true
```

Boolean logic

Each of the examples above tests just one condition. When George Boole invented the Boolean, he had much more planned for it than these humble beginnings. He invented Boolean logic, which lets you combine multiple conditions to form a result.

AND operator

In Dart, the operator for Boolean AND is written `&&`, used like so:

```
const isSunny = true;
const isFinished = true;
const willGoCycling = isSunny && isFinished;
```

Print `willGoCycling` and you'll see that it's `true`. If either `isSunny` or `isFinished` were `false`, then `willGoCycling`

would also be false.

OR operator

In Dart, the operator for Boolean OR is written ||, used like so:

```
const willTravelToAustralia = true;
const canFindPhoto = false;
const canDrawPlatypus = willTravelToAustralia ||
canFindPhoto;
```

Print canDrawPlatypus to see that its value is true. If both values on the right were false, then canDrawPlatypus would be false. If both were true, then canDrawPlatypus would still be true.

Operator precedence

```
const andTrue = 1 < 2 && 4 > 3;
const andFalse = 1 < 2 && 3 > 4;
const orTrue = 1 < 2 || 3 > 4;
const orFalse = 1 == 2 || 3 == 4;
```

Each of these tests two separate conditions, combining them with either AND or OR.

It's also possible to use Boolean logic to combine more than two comparisons. For example, you can form a complex comparison like so:

```
3 > 4 && 1 < 2 || 1 < 4
```

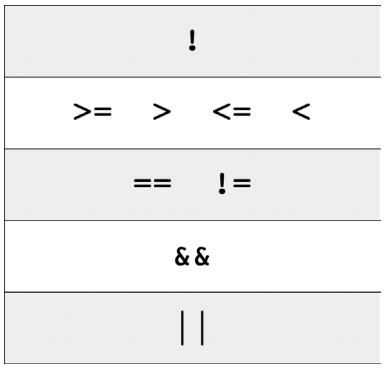
But now it gets a little confusing. You have three conditions with two different logical operators. With the comparisons

simplified, you have the following form:

```
false && true || true
```

Depending on the order you perform the AND and OR operations, you get different results. If you evaluate AND first, the whole expression is `true`, while if you evaluate OR first, the whole expression is `false`.

This is where **operator precedence** comes in. The following list shows the order that Dart uses to evaluate expressions containing comparison and logical operators:



Operators higher in the list are executed before operators lower in the list. You can see that && has a higher precedence than ||. So back to case from before:

```
false && true || true
```

First Dart will evaluate `false && true`, which is `false`. Then Dart will take that `false` to evaluate `false || true`, which is `true`. Thus the whole expression evaluates to `true`.

Overriding precedence with parentheses

If you want to override the default operator precedence, you can put parentheses around the parts Dart should evaluate first.

Compare the following two expressions:

```
3 > 4 && (1 < 2 || 1 < 4)  // false
(3 > 4 && 1 < 2) || 1 < 4  // true
```

String equality

In Dart, you can compare strings using the standard equality operator, ==, in exactly the same way as you compare numbers. For example:

```
const guess = "dog";
const dogEqualsCat = guess == "cat";
```

Here, `dogEqualsCat` is a Boolean, which in this case is `false` because the string `"dog"` does not equal the string `"cat"`.

Mini-exercises

1. Create a constant called `myAge` and set it to your age. Then, create a constant named `isTeenager` that uses Boolean logic to determine if the age denotes someone in the age range of 13 to 19.

2. Create another constant named maryAge and set it to 30. Then, create a constant named bothTeenagers that uses Boolean logic to determine if both you and Mary are teenagers.

3. Create a String constant named reader and set it to your name. Create another String constant named ray and set it to "Ray Wenderlich". Create a Boolean constant named rayIsReader that uses string equality to determine if reader and ray are equal.

Now that you understand Boolean logic, you're going to use that knowledge to make decisions in your code.

The if statement

The first and most common way of controlling the flow of a program is through the use of an **if statement**. It lets the program to do something only if a certain condition is true. For example, consider the following:

```
if (2 > 1) {
  print("Yes, 2 is greater than 1.");
}
```

This is a simple if statement. The **condition**, which is always a **Boolean expression**, is the part within the parentheses that follows the if statement. If the condition is true, then the if statement will execute the code between the braces. If the condition is false, then the if statement *won't* execute the code between the braces.

Obviously, the condition (2 > 1) is true, so when you run that you'll see:

```
Yes, 2 is greater than 1.
```

The else clause

You can extend an if statement to provide code to run in the event that the condition turns out to be false. This is known as the **else clause**.

Here's an example:

```
const animal = "Fox";
if (animal == "Cat" || animal == "Dog") {
  print("Animal is a house pet.");
} else {
  print("Animal is not a house pet.");
}
```

If animal equals either "Cat" or "Dog", then the statement will execute the first block of code. If animal does *not* equal either "Cat" or "Dog", then the statement will run the block inside the else clause of the if statement.

Run that code and you'll see the following in the debug console:

```
Animal is not a house pet.
```

Else-if chains

You can go even further with if statements. Sometimes you want to check one condition, and then check another condition if the first condition isn't true. This is where else-if comes into play, nesting another if statement in the else clause of a previous if statement.

You can use it like so:

```
const trafficLight = "yellow";
var command = "";
if (trafficLight == "red") {
  command = "Stop";
} else if (trafficLight == "yellow") {
  command = "Slow down";
} else if (trafficLight == "green") {
  command = "Go";
} else {
  command = "INVALID COLOR!";
}
print(command);
```

In this example, the first if statement will check if **trafficLight** is equal to **"red"**. Since it's not, the next **if** statement will check if **trafficLight** is equal to **"yellow"**. It is equal to **"yellow"**, so no check will be made for the case of **"green"**.

Run the code and it will print the following:

```
Slow  down
```

These nested if statements test multiple conditions, one by one, until a true condition is found. Only the code associated

with the first true condition encountered will be executed, regardless of whether there are subsequent `else-if` conditions that evaluate to `true`. In other words, the order of your conditions matters!

You can add an `else` clause at the end to handle the case where none of the conditions are `true`. This `else` clause is optional if you don't need it. In this example, you *do* need the `else` clause to ensure that `command` has a value by the time you print it out.

Variable scope

`if` statements introduce a new concept called **scope**. Scope is the extent to which a variable can be seen throughout your code. Dart uses curly braces as the boundary markers in determining a variable's scope. If you define a variable inside a pair of curly braces, then you're not allowed to use that variable outside of those braces.

To see how this works, replace the `main` function with the following code:

```
const global = "Hello, world";

void main() {
  const local = "Hello, main";

  if (2 > 1) {
    const insideIf = "Hello, anybody?";

    print(global);
    print(local);
    print(insideIf);
  }

  print(global);
  print(local);
  print(insideIf); // Not allowed!
}
```

Note the following points:

- There are three variables: `global`, `local` and `insideIf`.
- There are two sets of nested curly braces, one for the body of `main` and one for the body of the `if` statement.
- The variable named `global` is defined outside of the main function and outside of any curly braces. That makes it a **top-level variable**, which means it has a global scope. That is, it's visible everywhere in the file. You can see `print(global)` references it both in the `if` statement body and in the `main` function body.
- The variable named `local` is defined inside the body of the `main` function. This makes it a **local variable** and it

has local scope. It's visible inside the `main` function, including inside the `if` statement, but `local` is not visible outside of the `main` function.

- The variable named `insideIf` is defined inside the body of the `if` statement. That means `insideIf` is only visible within the scope defined by the `if` statement's curly braces.

Since the final `print` statement is trying to reference `insideIf` outside of its scope, Dart gives you the following error:

```
Undefined name 'insideIf'.
```

Delete that final `print` statement to get rid of the error.

As a general rule, you should make your variables have the smallest scope that they can get by with. Another way to say that is, define your variables as close to where you use them as possible. Doing so makes their purpose more clear, and it also prevents you from using or changing them in places where you shouldn't.

The ternary conditional operator

You've worked with operators that have two operands. For example, in `(myAge > 16)`, the two operands are `myAge` and `16`. But there's also an operator that takes three operands: the **ternary conditional operator**. It's strangely related to `if` statements — you'll see why this is in just a bit.

Let's take an example of telling a student whether their exam score is passing or not. Write an if-else statement to achieve this:

```
const score = 83;

String  message;
if (score >= 60) {
  message = "You passed";
} else {
  message  =  "You  failed";
}
```

That's pretty clear, but it's a lot of code. Wouldn't it be nice if you could shrink this to just a couple of lines? Well, you can, thanks to the ternary conditional operator!

The ternary conditional operator takes a condition and returns one of two values, depending on whether the condition is true or false. The syntax is as follows:

```
(condition) ? valueIfTrue : valueIfFalse;
```

Use the ternary conditional operator to rewrite your long code block above, like so:

```
const score = 83;
const message = (score >= 60) ? "You passed" :
"You failed";
```

In this example, the condition to evaluate is `score >= 60`. If the condition is `true`, the result assigned to `message` will be

"You passed"; if the condition is false, the result will instead be "You failed". Since 83 is greater than 60, the student receives good news.

The ternary conditional operator makes basic if-else statements much more compact, which in turn can make your code more readable.

However, for situations where using this operator makes your code *less* readable, then stick with the full **if–else** statement. Readability is always more important than fancy programming tricks that give the same result.

Mini-exercises

1. Create a constant named myAge and initialize it with your age. Write an **if** statement to print out "Teenager" if your age is between 13 and 19, and "Not a teenager" if your age is not between 13 and 19.

2. Use a ternary conditional operator to replace the **else-if** statement that you used above. Set the result to a variable named answer.

Switch statements

An alternate way to handle control flow, especially for multiple conditions, is with a switch statement. The switch statement takes the following form:

```
switch (variable) {
  case value1:
    // code
    break;
  case value2:
    // code
    break;

    ---

  default:
    //  code
}
```

There are a few different keywords, so here are what they mean:

- `switch`: Based on the value of the variable in parentheses, which can be an `int`, `String` or compile-time constant, `switch` will redirect the program control to one of the `case` values that follow.

- `case`: Each `case` keyword takes a value and compares that value using `==` to the variable after the `switch` keyword. You add as many `case` statements as there are values to check. When there's a match Dart will run the code that follows the colon.

- `break`: The `break` keyword tells Dart to exit the switch statement because the code in the `case` block is finished.

- `default`: If none of the `case` values match the `switch` variable, then the code after `default` will be executed.

The following sections will provide more detailed examples of `switch` statements.

Replacing else-if chains

Using `if` statements are convenient when you have one or two conditions, but the syntax can be a little verbose when you have a lot of conditions. Check out the following example:

```
const number = 3;
if (number == 0) {
  print("zero");
} else if (number == 1) {
  print("one");
} else if (number == 2) {
  print("two");
} else if (number == 3) {
  print("three");
} else if (number == 4) {
  print("four");
} else {
  print("something else");
}
```

Run that code and you'll see that it gets the job done — it prints `three` as expected. The wordiness of the `else-if` lines make the code kind of hard to read, though.

Rewrite the code above using a `switch` statement:

```
const number = 3;
switch (number) {
  case 0:
    print('zero');
    break;
  case 1:
    print('one');
    break;
  case 2:
    print('two');
    break;
  case 3:
    print('three');
    break;
  case 4:
    print('four');
    break;
  default:
    print('something else');
}
```

Execute this code and you'll get the same result of `three` again. However, the code looks cleaner than the `else-if` chain because you didn't need to include the explicit condition check for every `case`.

> **Note**: In Dart, switch statements don't support ranges like `number > 5`. Only `==` equality checking is allowed. If your conditions involve ranges, then you should use `if` statements.

Switching on strings

A switch statement also works with strings. Try the following example:

```
const weather = 'cloudy';
switch (weather) {
  case 'sunny':
    print('Put on sunscreen.');
    break;
  case 'snowy':
    print('Get your skis.');
    break;
  case 'cloudy':
  case 'rainy':
    print('Bring an umbrella.');
    break;
  default:
    print("I'm not familiar with that weather."
);
}
```

Run the code above and the following will be printed in the console:

```
Bring an umbrella.
```

In this example, the 'cloudy' case was completely empty, with no break statement. Therefore, the code "falls through" to the 'rainy' case. This means that if the value is equal to either 'cloudy' or 'rainy', then the switch statement will execute the same case.

Enumerated types

Enumerated types, also known as **enums**, play especially well with `switch` statements. You can use them to define your own type with a finite number of options.

Consider the previous example with the `switch` statement about weather. You're expecting `weather` to contain a string with a recognized weather word. But it's conceivable that you might get something like this from one of your users:

```
const weather = 'I like turtles.';
```

You'd be like, "What? What are you even talking about?"

That's what the `default` case was there for — to catch all the weird stuff that gets through. Wouldn't it be nice to make weird stuff impossible, though? That's where enums come in.

Create the `enum` as follows, placing it outside of the `main` function:

```
enum Weather {
  sunny,
  snowy,
  cloudy,
  rainy,
}
```

This enum defines four different kinds of weather. Yes, yes, you can probably think of more kinds than that; feel free to add them yourself. But please don't make `iLikeTurtles` an option. Separate each of the values with a comma.

Formatting tip: If you like the enum options listed in a vertical column as they are above, make sure the final item in the list has a comma after it. On the other hand, if you like them laid out horizontally, remove the comma after the last item. Once you've done that, pressing **Shift+Option+F** on a Mac or **Shift+Alt+F** on a PC in VS Code will auto-format it to your preferred style:

```
enum Weather { sunny, snowy, cloudy, rainy }
```

This formatting trick works with many kinds of lists in Dart.

Naming enums

When creating an `enum` in Dart, it's customary to write the `enum` name with an initial capital letter, as `Weather` was written in the example above. The values of an enum should use `lowerCamelCase` unless you have a special reason to do otherwise.

Switching on enums

Now that you have the `enum` defined, you can use a `switch` statement to handle all the possibilities, like so:

```
const weatherToday = Weather.cloudy;
switch (weatherToday) {
  case Weather.sunny:
    print("Put on sunscreen.");
    break;
  case Weather.snowy:
    print("Get your skis.");
    break;
  case Weather.cloudy:
  case Weather.rainy:
    print("Bring an umbrella.");
    break;
}
```

As before, this will print the following message:

```
Bring an umbrella.
```

Notice that there was no `default` case this time since you handled every single possibility. In fact, Dart will warn you if you leave one of the enum items out. That'll save you some time chasing bugs.

Enum values and indexes

Before leaving the topic of enums, there's one more thing to note. If you try to print an enum, you'll get its value:

```
print(weatherToday);
// Weather.cloudy
```

Unlike some languages, a Dart enum isn't an integer. However, you can get the index, or ordinal placement, of a

value in the enum like so:

```
final index = weatherToday.index;
```

Since `cloudy` is the third value in the enum, the zero-based `index` is 2.

Avoiding the overuse of switch statements

Switch statements, or long `else-if` chains, can be a convenient way to handle a long list of conditions. If you're a beginning programmer, go ahead and use them; they're easy to use and understand.

However, if you're an intermediate programmer and still find yourself using `switch` statements a lot, there's a good chance you could replace some of them with more advanced programming techniques that will make your code easier to maintain. If you're interested, do a web search for **refactoring switch statements with polymorphism** and read a few articles about it.

Mini-exercises

1. Make an `enum` called `AudioState` and give it values to represent `playing`, `paused` and `stopped` states.

2. Create a constant called `audioState` and give it an `AudioState` value. Write a `switch` statement that prints a message based on the value.