

# Chapter 5: Functions

Each week, there are tasks that you repeat over and over: eat breakfast, brush your teeth, write your name, read books about Dart, and so on. Each of those tasks can be divided up into smaller tasks. Brushing your teeth, for example, includes putting toothpaste on the brush, brushing each tooth and rinsing your mouth out with water.

The same idea exists in computer programming. A **function** is one small task, or sometimes a collection of several smaller, related tasks that you can use in conjunction with other functions to accomplish a larger task.

In this chapter, you'll learn how to write functions in Dart, including both **named functions** and **anonymous functions**.

## Function basics

You can think of functions like machines; they take something you provide to them (the input), and produce something different (the output).



There are many examples of this in daily life. With an apple juicer, you put in apples and you get out apple juice. The input is apples; the output is juice. A dishwasher is another example. The input is dirty dishes, and the output is clean dishes. Blenders, coffee makers, microwaves and ovens are

all like real-world functions that accept an input and produce an output.

## Don't repeat yourself

Assume you have a small, useful piece of code that you've repeated in multiple places throughout your program:

```
// one place
if (fruit == 'banana') {
    peelBanana();
    eatBanana();
}

// another place
if (fruit == 'banana') {
    peelBanana();
    eatBanana();
}

// some other place
if (fruit == 'banana') {
    peelBanana();
    eatBanana();
}
```

Now, that code works rather well, but repeating that code in multiple spots presents at least two problems. The first problem is that you're duplicating effort by having this code in multiple places in your program. The second, and more troubling problem, is that if you need to change the logic in that bit of code later on, you'll have to track down all of those instances of the code and change them in the same way. That creates a high possibility that you'll make a

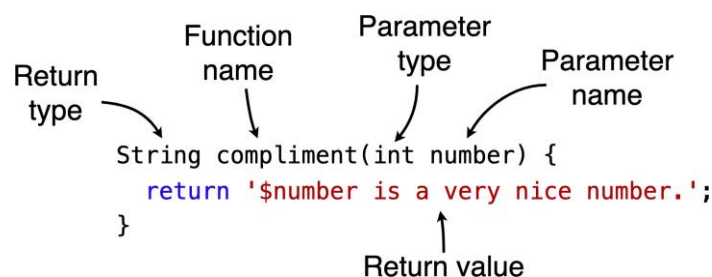
mistake somewhere, or even miss changing one of the instances because you didn't see it.

Over time, this problem has led to some sound advice for writing clean code: **don't repeat yourself**, abbreviated as **DRY**. This term was originally coined in the book *The Pragmatic Programmer* by Andrew Hunt and David Thomas. Writing DRY code will help you prevent many bugs from creeping into your programs.

Functions are one of the main solutions to the duplication problem in the example above. Instead of repeating blocks of code in multiple places, you can simply package that code into a function and call that function from wherever you need to.

## Anatomy of a Dart function

In Dart, a function consists of a return type, a name, a parameter list in parentheses and a body enclosed in braces.



Here is a short summary of the labeled parts of the function:

- **Return type**: This comes first; it tells you immediately what the type will be of the function output. This particular function will return a **String**, but your

functions can return any type you like. If the function won't return anything, that is, if it performs some work but doesn't produce an output value, you can use `void` as return type.

- **Function name** : You can name functions almost anything you like, but you should follow the **lowerCamelCase** naming convention. You'll learn a few more naming conventions a little later in this chapter.

- **Parameters** : Parameters are the input to the function; they go inside the parentheses after the function name. This example has only one parameter, but if you had more than one, you would separate them with commas. For each parameter, you write the type first, followed by the name. Just as with variable names, you should use **lowerCamelCase** for your parameter names.

- **Return value** : This is the function's output, and it should match the return type. In the example above, the function returns a `String` value by using the `return` keyword. If the return type is `void`, though, then you don't return anything.

The return type, function name and parameters are collectively known as the **function signature**. The code between the braces is known as the **function body**.

This is what the function above looks like in the context of a program:

```

void main() {
  const input = 12;
  final output = compliment(input);
  print(output);
}

String compliment(int number) {
  return '$number is a very nice number.';
}

```

What have we here? Not one function, but two? Yes, `main` is also a function, and one you've seen many times already. It's the function that every Dart program starts with. Since `main` doesn't return a value, the return type of `main` must be `void`. Although `main` can take parameters, there aren't any in this case, so there's only a pair of empty parentheses that follow the function name.

Notice that the `compliment` function is *outside* of `main`. Dart supports **top-level functions**, which are functions that aren't inside a class or another function. Conversely, you may nest one function inside another. And when a function is inside a class, it's called a **method**, which you'll learn more about in Chapter 6.

You call a function by writing its name, and providing the **argument** which is the value you provide inside the parentheses as the parameter to the function. In this case, you're calling the `compliment` function and passing in an argument of 12. Run the code now and you'll see the following result:

12 is a very nice number.

Indeed, twelve *is* a nice number. It's the largest one-syllable number in English.

**Note:** It's easy to get the words *parameter* and *argument* mixed up. A **parameter** is the name and type that you define as an input for your function. An **argument** on the other hand, is the actual value that you pass in. A parameter is abstract, while an argument is concrete.

## More about parameters

Parameters are incredibly flexible in Dart, so they deserve their own section.

## Using multiple parameters

In a Dart function, you can use any number of parameters. If you have more than one parameter for your function, simply separate them with commas. Here's a function with two parameters:

```
void helloPersonAndPet(String person, String pet) {  
  print('Hello, $person, and your furry friend, $pet!');  
}
```

Parameters like the ones above are called **positional parameters** because you have to supply the arguments in

the same order that you defined the parameters when you wrote the function. If you call the function with the parameters in the wrong order, you'll get something obviously wrong:

```
helloPersonAndPet('Fluffy', 'Chris');  
// Hello_ Fluffy_ and your furry friend_ Chris!
```

## Making parameters optional

The function above was very nice, but it was a little rigid. For example, try the following:

```
helloPersonAndPet();
```

If you don't have exactly the right number of parameters, the compiler will complain to you:

```
2 positional argument(s) expected, but 0 found.
```

You defined `helloPersonAndPet` to take two arguments, but in this case, you didn't pass in any. It would be nice if the code could detect this, and just say, "Hello, you two!" if no names are provided. Thankfully, it's possible to have optional parameters in a Dart function!

Imagine you want a function that takes a first name, a last name and a title, and returns a single string with the various pieces of the person's name strung together:

```
String fullName(String first, String last, String title) {  
    return "$title $first $last";  
}
```

The thing is, not everyone has a title, or wants to use their title, so your function needs to treat the title as optional. To indicate that a parameter is optional, you surround the parameter with square brackets and add a question mark after the type, like so:

```
String fullName(String first, String last, [String? title]) {  
    if (title != null) {  
        return "$title $first $last";  
    } else {  
        return "$first $last";  
    }  
}
```

Writing `[String? title]` makes `title` optional. If you don't pass in a value for `title`, then it will have the value of `null`, which means "no value". The updated code checks for `null` to decide how to format the return string.

Here are two examples to test it out:

```
print(fullName("Ray", "Wenderlich"));  
print(fullName("Albert", "Einstein", "Professor"));
```



Run that now and you'll see the following:

```
Ray Wenderlich  
Professor Albert Einstein
```

The function correctly handles the optional title.

**Note:** Technically speaking, the question mark in `String?` is not written *after* the type; it's an integral part of the type, that is, the nullable `String` type. More on this in Chapter 7.

## Providing default values

In the example above, you saw that the default value for an optional parameter was `null`. This isn't always the best value for a default, though. That's why Dart also gives you the power to change the default value of any parameter in your function by using the assignment operator.

Take a look at this example:

```
bool withinTolerance(int value, [int min = 0, int max = 10]) {  
  return min <= value && value <= max;  
}
```

There are three parameters here, two of which are optional: `min` and `max`. If you don't specify a value for them, then `min` will be 0 and `max` will be 10.

Here are some specific examples to illustrate that:

```
withinTolerance(5)    // true  
withinTolerance(15)  // false
```

Since 5 is between 0 and 10, this evaluates to true; but since 15 is greater than the default max of 10, it evaluates to **false**.

If you want to specify values other than the defaults, you can do that as well:

```
withinTolerance(9, 7, 11) // true
```

Since 9 is between 7 and 11, the function returns true.

Look at that function call again: `withinTolerance(9, 7, 11)`. Imagine that you're reading through your code for the first time in a month. What do those three numbers even mean? If you've got a good memory, you might recall that one of them is `value`, but which one? The first one? Or was it the second one? Or maybe it was the last one.

If that wasn't bad enough, the following function call also returns **true**:

```
withinTolerance(9, 7) // true
```

Since the function uses positional parameters, the provided arguments must follow the order you defined the parameters. That means `value` is 9, `min` is 7 and `max` has the default of 10. But who could ever remember that?

Of course you could just **Command+click** the function name on a Mac, or **Control+click** on a PC, to go to the definition and remind yourself of what the parameters meant. But the point is that this code is extremely hard to read. If only there were a better way!

Well, now that you mention it...

## Naming parameters

Dart allows you to use **named parameters** to make the meaning of the parameters more clear in function calls.

To create a named parameter, you surround it with curly braces instead of square brackets. Here's the same function as above, but using named parameters instead:

```
bool withinTolerance(int value, {int min = 0, int max = 10}) {  
  return min <= value && value <= max;  
}
```

Note the following:

`min` and `max` are surrounded by braces, which means you *must* use the parameter names when you provide their argument values to the function.

Like square brackets, curly braces make the parameters inside optional. Since `value` isn't inside the braces, though, it's still required.

To provide an argument, you use the parameter name, followed by a colon and then the argument value. Here is how you call the function now:

```
withinTolerance(9, min: 7, max: 11) // true
```

That's a lot clearer, isn't it? The names `min` and `max` make it obvious where the tolerance limits are now.

An additional benefit of named parameters is that you don't have to use them in the exact order in which they were defined. These are both equivalent ways to call the function:

```
withinTolerance(9, min: 7, max: 11) // true  
withinTolerance(9, max: 11, min: 7) // true
```

And since named parameters are optional, that means the following function calls are also valid:

```
withinTolerance(5)           // true  
withinTolerance(15)          // false  
withinTolerance(5, min: 7)   // false  
withinTolerance(15, max: 20) // true
```

In the first two lines, since `min` is 0 and `max` is 10 by default, values of 5 and 15 evaluate to `true` and `false` respectively. In the last two lines, the `min` and `max` defaults were changed, which also changed the outcomes of the evaluations.

## Making named parameters required

You might like to make `value` a named parameter as well. That way you could call the function like so:

```
withinTolerance(value: 9, min: 7, max: 11)
```

However, this brings up a problem. Named parameters are optional by default, but `value` can't be optional. If it were, someone might try to use your function like this:

```
withinTolerance()
```

Should that return `true` or `false`? It doesn't make sense to return anything if you don't give the function a value. This is just a bug waiting to happen.

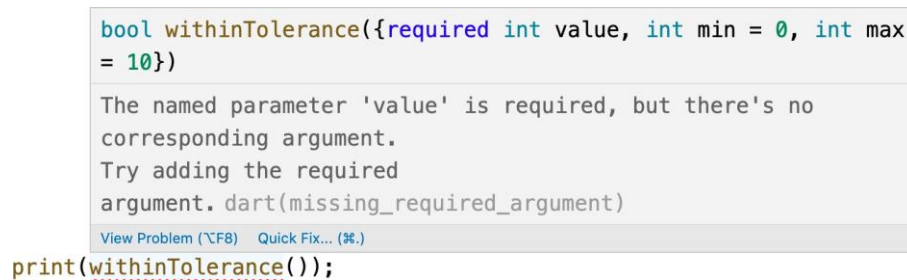
What you want is to make `value` required instead of optional, while still keeping it as a named parameter. You can achieve this by including `value` inside the curly braces and adding the `required` keyword in front:

```
bool withinTolerance({  
  required int value,  
  int min = 0,  
  int max = 10,  
}) {  
  return min <= value && value <= max;  
}
```

Since the function signature was getting a little long, adding a comma after the last parameter lets the IDE format it vertically. You still remember how to auto-format in VS

Code, right? That's **Shift+Option+F** on a Mac or **Shift+F** on a PC.

With the `required` keyword in place, VS Code will warn you if you don't provide a value for `value` when you call the function:



```
bool withinTolerance({required int value, int min = 0, int max = 10})

The named parameter 'value' is required, but there's no
corresponding argument.
Try adding the required
argument. dart(missing_required_argument)
View Problem (⌘F8) Quick Fix... (⌘.)

print(withinTolerance());
```

Using named parameters makes your code more readable and is an important part of writing clean code when you have multiple inputs to a function. In the next section, you'll learn some more best practices for writing good functions.

## Writing good functions

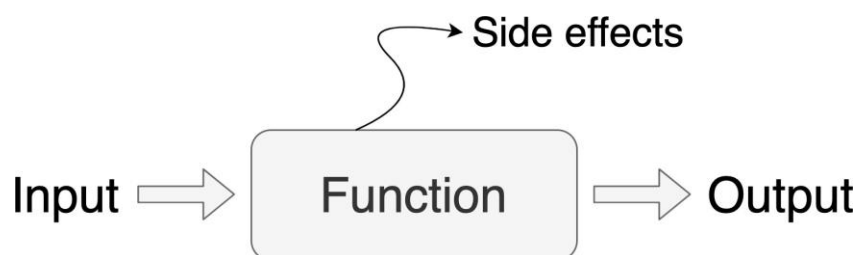
People have been writing code for decades. Along the way, they've designed some good practices to improve code quality and prevent errors. One of those practices is writing DRY code as you saw earlier. Here are a few more things to pay attention to as you learn about writing good functions.

## Avoiding side effects

When you take medicine to cure a medical problem, but that medicine makes you fat, that's known as a side effect. If you put some bread in a toaster to make toast, but the toaster

burns your house down, that's also a side effect. Not all side effects are bad, though. If you take a business trip to Paris, you also get to see the Eiffel Tower. *Magnifique!*

When you write a function, you know what the inputs are: the parameters. You also know what the output is: the return value. Anything beyond that, that is, anything that affects the world outside of the function, is a side effect.



Have a look at this function:

```
void hello() {  
    print("Hello!");  
}
```

Printing something to the console is a side effect, because it's affecting the world outside of the function. If you wanted to rewrite your function so that there were no side effects, you could write it like this:

```
String hello() {  
    return "Hello!";  
}
```

Now, there's nothing inside the function body that affects the outside world. You'll have to write the string to the



console somewhere outside of the function.

It's fine, and even necessary, for some functions to have side effects. But as a general rule, functions without side effects are easier to deal with and reason about. You can rely on them to do exactly what you expect because they always return the same output for any given input. These kinds of functions are also called **pure functions**.

Here is another function with side effects to further illustrate the point:

```
var myPreciousData = 5782;

String anInnocentLookingFunction(String name) {
  myPreciousData = -1;
  return "Hello, $name. Heh, heh, heh.";
}
```

Unless you took the time to study the code inside of `anInnocentLookingFunction`, you'd have no idea that calling this innocent-looking function would also change your precious data. That's because the function had an unknown side effect. This is also a good reminder about the dangers of using global variables like `myPreciousData`, as you never know who might change it.

Make it your ambition to maximize your use of pure functions, and minimize your use of functions with side effects.

## Doing only one thing

Proponents of “clean code” recommend keeping your functions small and logically coherent. *Small* here means only a handful of lines of code. **If a function is too big, or contains unrelated parts, consider breaking it into smaller functions.**

Write your functions so that each one has only a single job to do. If you find yourself adding comments to describe different sections of a complex function, that’s usually a good clue that you should break your function up into smaller functions. In clean coding, this is known as the **Single Responsibility Principle**. In addition to functions, this principle also applies to classes and libraries. But that’s a topic for another chapter.

## Choosing good names

You should always give your functions names that describe exactly what they do. If your code sounds like well-written prose, it’ll be faster to read and easier to understand.

This naming advice applies to almost every programming language. However, there are a few additional naming conventions that Dart programmers like to follow. These are recommendations, not requirements, but keep them in mind as you code:

Use noun phrases for pure functions; that is, ones without side effects. For example, use `averageTemperature` instead of `getAverageTemperature` and `studentNames` instead of `extractStudentNames`.

Use verb phrases for functions with side effects. For example, `updateDatabase` or `printHello`.

Also use verb phrases if you want to emphasize that the function does a lot of work. For example, `calculateFibonacci` or `parseJson`.

Don't repeat parameter names in the function name. For example, use `cube(int number)` instead of `cubeNumber(int number)`, or `printStudent(String name)` instead of `printStudentName(String name)`.

## Optional types

Earlier you saw this function:

```
String compliment(int number) {  
  return "$number is a very nice number.";  
}
```

The return type is `String`, and the parameter type is `int`. Dart is an optionally-typed language, so it's possible to omit the types from your function declaration. In that case, the function would look like this:

```
compliment(number) {  
  return "$number is a very nice number.";  
}
```

Dart can infer that the return type here is `String`, but it has to fall back on `dynamic` for the unknown parameter type. The following function is the equivalent of what Dart sees:

```
String compliment(dynamic number) {  
  return "$number is a very nice number.";  
}
```

While it's permissible to omit return and parameter types, this book recommends that you include them at the very least for situations where Dart can't infer the type. As you learned in Chapter 3, there's a much greater advantage to writing Dart in a statically-typed way.

## Mini-exercises

1. Write a function named `youAreWonderful`, with a `String` parameter called `name`. It should return a string using `name`, and say something like "You're wonderful, Bob."
2. Add another `int` parameter to that function called `numberOfPeople` so that the function returns something like "You're wonderful, Bob. 10 people think so."
3. Make both inputs named parameters. Make `name` required and set `numberOfPeople` to have a default of 30.

## Anonymous functions

All the functions you've seen previously in this chapter, such as `main`, `hello`, and `withinTolerance` are **named** functions, which means, well, they have a name.

Function  
name



```
String compliment(int number) {  
    return '$number is a very nice number.';  
}
```

But not every function needs a name. If you remove the return type and the function name, then what you have left is an anonymous function:

```
String compliment(int number) {  
    return '$number is a very nice number.';  
}
```

The return type will be inferred from the return value of the function body, `String` in this case.

So, why all the stealth by being anonymous, you ask? Are functions concerned about their online privacy, too? Well, that's not quite it. Sometimes you only need functions in one specific spot in your code, for one specific reason, and there's no reason to give that function a name. You'll see some examples of this soon.

## First-class citizens

In Dart, functions are **first-class citizens**. That means you can treat them like any other type, assigning functions as values to variables and even passing functions around as parameters or returning them from other functions.

## Assigning functions to variables

When assigning a value to a variable, functions behave just like other types:

```
int number = 4;
String greeting = "hello";
bool isHungry = true;
Function multiply = (int a, int b) {
    return a * b;
};
```

The type of `multiply` is `Function`, the same way that `number` is `int`, `greeting` is `String` and `isHungry` is `bool`. On the right hand side of each assignment, you have literal values: `4` is an integer literal, `"hello"` is a string literal, `true` is a Boolean literal, and the anonymous function you see above is a function literal.

One reason that you need anonymous functions is that you can't assign a named function to a variable:

```
Function myFunction = int multiply(int a, int b)
{
    return a * b;
};
```

Trying to do that produces the following error:

```
Function expressions can't be named.
```

## Passing functions to functions

Just as you can write a function to take `int` or `String` as a parameter, you can also have `Function` as a parameter:

```
void namedFunction(Function anonymousFunction) {  
    // function body  
}
```

Here, `namedFunction` takes an anonymous function as a parameter.

## Returning functions from functions

Just as you can pass in functions as input parameters, you can also return them as output:

```
Function namedFunction() { return  
    () {  
        print('hello');  
    };  
}
```

The return value is an anonymous function of type `Function`.

Functions that return functions, or that accept them as parameters, are called **higher order functions**.

## Using anonymous functions

Now that you know where you can use anonymous functions,

try a hand at doing it yourself. Take the `multiply` function again:

```
final multiply = (int a, int b) {  
    return a * b;  
};
```

To call the function that the variable `multiply` refers to, simply use the variable name followed by the arguments in parentheses:

```
print(multiply(2, 3));
```

This will print 6 just as if you were calling the named function `multiply`.

## Returning a function

Have a look at a different example:

```
Function applyMultiplier(num multiplier) {  
    return (num value) {  
        return value * multiplier;  
    };  
}
```

This one looks a little crazy at first. There are two `return` statements! To make sense of a function like this, look at it this way: `applyMultiplier` is a named function that returns an anonymous function. It's like a machine that makes a machine. That second `return` statement belongs to the anonymous function and won't get called when `applyMultiplier` is called.



Now write the following line:

```
final triple = applyMultiplier(3);
```

**triple** is a constant variable of type **Function**; that is, the anonymous function that **applyMultiplier** returned. You haven't run that anonymous function yet. You're simply storing it in a variable named **triple**.

Run it now:

```
print(triple(6));  
print(triple(14.0));
```

Passing arguments to the variable runs the function. Because the parameter type was **num**, it can accept both **int** and **double** inputs. This is the result:

```
18  
42.0
```

Going back to the machine-that-makes-a-machine analogy, passing 3 into **applyMultiplier** was like setting a dial on the first machine. You set it to “make tripling machines.” So what you got out was a machine that triples everything you give it. If you had set the dial on the first machine to 4, then you would have gotten quadrupling machines, if 2, then doubling machines.

## Anonymous functions in **forEach** loops

Chapter 4 introduced you to `forEach` loops, which iterate over a collection. Although you may not have realized it, that was an example of using an anonymous function. So if you have a list of numbers, like so:

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

You can call `forEach` on the list and pass in an anonymous function that triples each number in the list and prints out that value.

```
numbers.forEach((number) {  
  final tripled = number * 3;  
  print(tripled);  
});
```

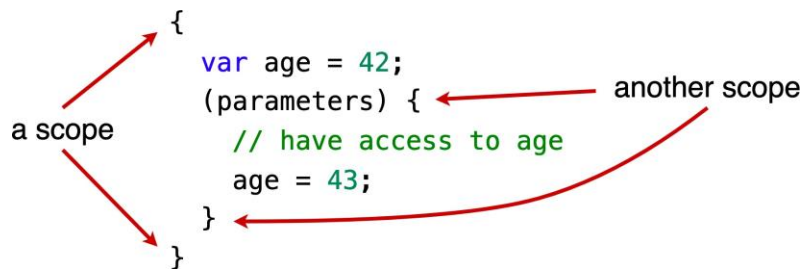
The parameter type of `number` is inferred from the list element types; in this case, `int`. Run the code and you'll see the following result:

```
3  
6  
9
```

Earlier in this chapter you learned how functions are a way to package reusable code that you can call in multiple places. The example here shows one of the main benefits of anonymous functions, which is packaging up logic that you *don't* need in multiple places, so you don't need to make it a named function. You simply need to pass the logic around either as input to, or as output from, another function.

## Closures and scope

Anonymous functions in Dart act as what are known as **.closures**. The term closure means that the code “closes around” the surrounding scope, and therefore has access to variables and functions defined within that scope.



A scope in Dart is defined by a pair of curly braces. All the code within these braces is a scope. You can even have nested scopes within other scopes. Examples of scopes are function bodies and the bodies of loops.

The return value of the `applyMultiplier` function from before is an example of a closure.

```
Function applyMultiplier(num multiplier) {
  return (num value) {
    return value * multiplier;
  };
}
```

The anonymous function it returns closes over the `multiplier` value that's passed in as a parameter to `applyMultiplier`.

As another example, if you have a variable `counter` and then define an anonymous function below it, that anonymous function acts like a closure and has access to `counter`, and so can change it.

```
var counter = 0;
final incrementCounter = () {
  counter += 1;
};
```

The anonymous function that defines `incrementCounter` can access `counter`, even though `counter` is not a parameter to the anonymous function, nor is it defined in the function body.

Call `incrementCounter` five times and print `counter`:

```
incrementCounter();
incrementCounter();
incrementCounter();
incrementCounter();
incrementCounter();
print(counter); // 5
```

You'll see that `counter` now has a value of 5.

You can use a closure as a function return value, and, as in this example, count the number of times a given function has been called.

```
Function countingFunction() { var  
  counter = 0;  
  final incrementCounter = () {  
    counter += 1;  
    return counter;  
  };  
  return incrementCounter;  
}
```

Each function returned by `countingFunction` will have its own version of `counter`. So if you were to generate two functions with `countingFunction`, like so:

```
final counter1 = countingFunction();  
final counter2 = countingFunction();
```

...then each call to those functions will increment its own `counter` independently:

```
print(counter1()); // 1  
print(counter2()); // 1  
print(counter1()); // 2  
print(counter1()); // 3  
print(counter2()); // 2
```

## Mini-exercises

1. Change the `youAreWonderful` function in the first mini-exercise of this chapter into an anonymous function. Assign it to a variable called `wonderful`.
2. Using `forEach`, print a message telling the people in the following list that they're wonderful.

```
const people = ['Chris', 'Tiffani', 'Pablo'];
```

## Arrow functions

Dart has a special syntax for functions whose body is only one line. Consider the following named function `add` that adds two numbers together:

```
int add(int a, int b) {  
  return a + b;  
}
```

Since the body is only one line, you can convert it to the following form:

```
int add(int a, int b) => a + b;
```

You replaced the function's braces and body with an arrow (`=>`) and left off the `return` keyword. The return value is whatever the value of the expression is. Writing a function in this way is known as **arrow syntax** or **arrow notation**.

You can also use arrow syntax with anonymous functions. You're simply left with the parameter list, the arrow, and a single expression:

```
(parameters) => expression;
```

In the following examples, you're going to **refactor**, or rewrite, some of the anonymous functions you saw earlier in

the chapter.

## Refactoring example 1

The body of the anonymous function you assigned to `multiply` has one line:

```
final multiply = (int a, int b) {  
    return a * b;  
};
```

You can convert it to use arrow syntax as follows:

```
final multiply = (int a, int b) => a * b;
```

You can call it just as you did before, with the same result:

```
print(multiply(2, 3)); // 6
```

## Refactoring example 2

You can also use arrow syntax for the anonymous function returned by `applyMultiplier`:

```
Function applyMultiplier(num multiplier) {  
    return (num value) {  
        return value * multiplier;  
    };  
}
```

With arrow syntax, the function becomes:

```
Function applyMultiplier(num multiplier) {  
  return (num value) => value * multiplier;  
}
```

The result of the function is the same as before.

## Refactoring example 3

You can't use arrow syntax on the `forEach` example, though:

```
numbers.forEach((number) {  
  final tripled = number * 3;  
  print(tripled);  
});
```

That's because there's more than one line in the function body. However, if you rewrote it to fit on line, that would work:

```
numbers.forEach((number) => print(number * 3));
```

## Mini-exercise

Change the `forEach` loop in the previous “You're wonderful” mini-exercise to use arrow syntax.

## Challenges

Before moving on, here are some challenges to test your knowledge of functions. It's best if you try to solve them yourself, but solutions are available in the **challenge** folder for this chapter if you get stuck.



## Challenge 1: Prime time

Write a function that checks if a number is prime.

## Challenge 2: Can you repeat that?

Write a function named `repeatTask` with the following definition:

```
int repeatTask(int times, int input, Function task)
```

It repeats a given task on `input` for `times` number of times.

Pass an anonymous function to `repeatTask` to square the input of 2 four times. Confirm that you get the result 65536, since 2 squared is 4, 4 squared is 16, 16 squared is 256, and 256 squared is 65536.

## Challenge 3: Darts and arrows

Update Challenge 2 to use arrow syntax.

## Key points

Functions package related blocks of code into reusable units.

A function signature includes the return type, name and parameters. The function body is the code between the braces.

Parameters can be positional or named, and required or optional.

Side effects are anything, besides the return value, that change the world outside of the function body.

To write clean code, use functions that are short and only do one thing.

Anonymous functions don't have a function name, and the return type is inferred.

Dart functions are first-class citizens and thus can be assigned to variables and passed around as values.

Anonymous functions act as closures, capturing any variables or functions within its scope.

Arrow syntax is a shorthand way to write one-line functions.