

# **A GUIDE TO MOBILE APP DEVELOPMENT**

**BY PRECIOUS DAMISA E.**

# CONTENTS

<b>What is your story?</b>	<b>1</b>
<b>Introduction</b>	<b>2</b>
This book, what is it about?	
How this book is organized	
 <b>Part 1: Learning the Dart Programming Language</b>	
 <b>1     Setting up Your Computer</b>	
Installing the IntelliJ IDEA Integrated Development Environment	
Setting Up IntelliJ IDEA to Run Dart Programs	
Getting The Dart Software Development kit	
 <b>2     The Building Blocks</b>	
Variables, values and types	
int	
double	
String	
List	
Map	
 <b>3     keywords, Comments and Errors</b>	
Dart's Keywords	
Types of Comments and Their Use	
Errors	
Types of Errors	
 <b>4     Functions</b>	
What Is A Function?	
Types of Functions	
Returning Values from a Function	
The Use of Parameters	
Lexical Scope	

## **5 Operators**

Arithmetic Operators

Relational Operators

Logical Operators

## **6 Control Flow Statements**

### **Selection Statements**

If, else and else if

Switch and case

Conditional expressions

### **Repetition Statements**

For loop

While loop

Do while loop

### **Jump Statements**

Break and continue

## **7 Object Oriented Programming**

What is a class?

What is an object?

Constructors

Extending a class

## **8 Advanced Topics**

Generics

Enum

Exceptions

## **Part 2: Building Mobile Applications with Dart and Flutter (Video Series)**

## WHAT IS YOUR STORY?

- ❖ Are you a student looking to acquire a skill that would make you more powerful and enterprising than your mates or one who needs some extra reading/training in order to do well in a beginning computer course?
- ❖ Are you a worker that is looking to learning how the computers in your workplace operate, i.e. how they've been programmed to do the amazing computation, analysis and problem solving that they do?
- ❖ Are you one that uses a computer for tasks such as word processing, designs etc. and wants to do something more interesting with your machine?
- ❖ Are you a job finder that is interested in delving into the world of computer programming and application development, which is one of the more prestigious and highly sought after career today?

Well, if you want to write computer programs and build applications, then this book is for you. This books takes a gentler approach at explaining computer programming terminologies, as it relates to building applications, so that its content can be easily grasped by the reader, who can be from any field, level, age or gender. Yes! Gender, because it is common in our world today, to find more males aspiring towards technological ventures such as programming and application development, than females. That should never be the case though. Programming is for all, so this book does its own part in making that perceivable and easily achievable.

# INTRODUCTION

## THIS BOOK, WHAT IS IT ABOUT?

This book shares solid information on how to use the programming language (Dart) to build mobile applications. Mobile applications that can work on an Android and IOS phone. I guess that sounds like killing two birds with one stone. How amazing is that? If you're wondering what a **programming language** is, then let me take this time to quickly explain to you what it is, as it is the requirement for building mobile applications.

Let me start by using an analogy. You and I communicate with one another using a known language (English, Yoruba, Hausa, Igbo, etc.) we express our ideas to one another, we give instructions to one another and expect the person receiving the instructions to carry out those instructions and then provide a result. That is exactly what a programming language is like. Although, a programming language is not used for a **human to human** communication, rather it is used for **human to computer communication**. You use a programming language to communicate your ideas to a computer, you tell the computer what to do, when to do it and how to do it. Some ignorant persons may argue that computers are smarter than humans, but it's actually the other way round, computers are very dull, they are useless without the instructions that tell them what to do.

We're going to start from scratch and learn how to properly tell the very dull computer how to do some simple things. From there, we would progress to more complex and interesting stuffs, such as writing instructions for the computer, to build our desired applications and execute relevant tasks.

I mentioned that we would learn how to build applications that work on an Android and IOS phone, which is very interesting because there are several ways to build mobile applications. Some ways don't allow you to build applications that would work on both platforms (Android & IOS), but in this book, we shall learn how to build multiplatform mobile applications.

This ability to build applications that work on both operating systems is made possible by a technology called **Flutter**. Flutter is a framework or tool, which was built by the renowned tech company giant, Google. And they made it free for everyone, to learn and use it in building applications as pleased. Also you should know that the programming language **Dart**, was also developed by the same company Google.

Just so you don't get it confused, let me reiterate. Dart is the name of the programming language which is used in tight collaboration with Flutter to build mobile applications. Dart on its own is a programming language, while Flutter is a framework for Dart. By framework, I mean that Flutter is a set of prebuilt components that make are

used in a mobile application. The tools which flutter provides, help us in developing an application, packaging it, and makes it possible that we can install it on our phones. The prebuilt components that Flutter ships with, makes it possible to easily get started with developing a mobile app. Take for example, when you want to develop an application, you're definitely going to have some screens in the app that display a button, which a user can tap on, or some text that convey some message to the user. Just so we don't have to design all these basic and needed components from scratch ourselves, Flutter had to make them available. Some of the UI (User Interface) components that Flutter contains are buttons and those for displaying text, images, amongst many others. As the programmer or app developer, you can easily use these already provided components in building your application and also based on your knowledge of the Dart programming language, which is what was used in designing the components in the first place, you can use your Dart knowledge to easily build new components or customize the prebuilt components for your application.

Don't worry much if all these explanations are being a little difficult to grasp. It is all what this book aims to guide you on. As you progress in your study of the book, it would all become clearer.

## **HOW THIS BOOK IS ORGANIZED**

This book is organized into two parts mainly: part 1 and part 2. The parts are further broken down into a series of chapters.

In part 1, we shall cover a lot on the Dart programming language, we shall look at the key areas that are required to get you started when it comes to building mobile applications with Flutter.

Part 2 of this book focuses on using the Flutter framework to build mobile applications. Being able to efficiently build apps with Flutter requires one to have a good knowledge of the Dart programming language. That is why I decided it was best we began, by first trying to understand the rudiments of the Dart language, before finally delving into Flutter. You can see it as us trying to grab our tools, which would be needed for work later.

Throughout the book, important keywords and definitions are bolded. This was done, so as easily draw the attention of the user to such keywords and definitions, which he/she should take proper note of.

If you encounter any word that you aren't familiar with, please first consult the glossary section of the book, which contains definitions of most of the technical words that are used in the book.

## CHAPTER 1

# SETTING UP YOUR COMPUTER

In this chapter, we shall go through the steps required in setting up your computer to run Dart programs. For convenience and also, to reduce the amount of downloads you would have to do, I put together the required applications that would be needed in setting up your computer to run Dart programs, these are bundled in the portable flash drive that is made available with this book. So all you have to do, is follow the installation steps for getting it all to work. Installation steps are specified in the video series.

## CHAPTER 2

# THE BUILDING BLOCKS

In this chapter, we shall look at what **variables, values and types are**, and how they are used in the Dart programming language. These are the building blocks of any programming language, not just Dart, so a good grasp of these concepts is required.

### Variables, Values and Types

A variable is like a container that something can be put into. While a value, is what is put into a variable. Just as a container, e.g. a bucket can hold different kinds of things, that's how a variable can hold different kinds of values. So, where do types come in? A type for a variable is used to specify exactly what kind of value can be put into a variable.

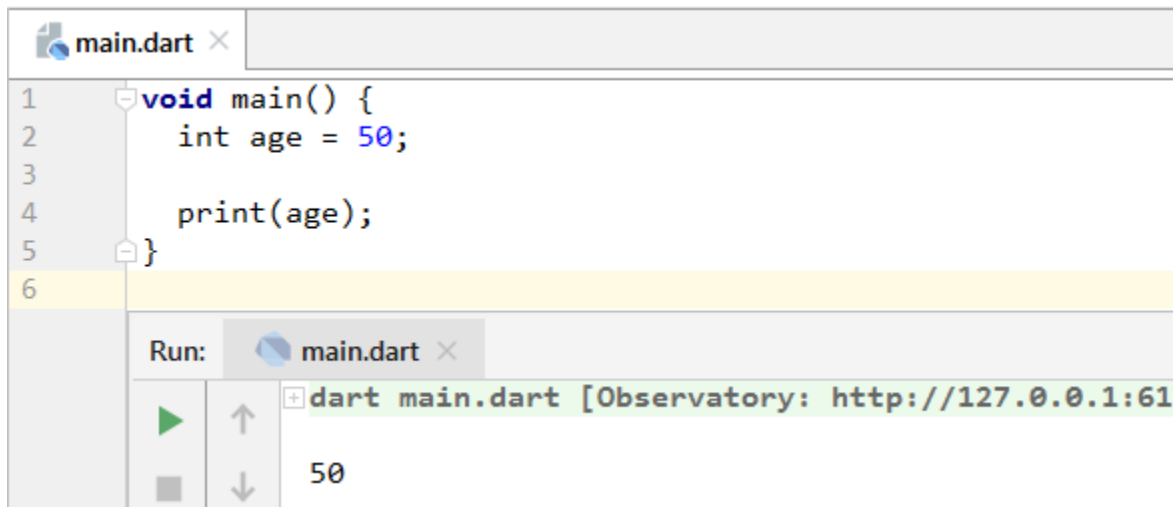
Dart is a **statically or strongly typed language**. What that means is that, Dart defines some types which can be used to specify the kind of value that a variable can hold, when the variable is defined. Some of these types include; **int, double, String, bool, List, Map**, etc. Don't worry if you don't know what these strange names mean, we will soon uncover what they all mean and how they can be used in writing programs.

I bet your hands are a little itchy right about now, you want to begin write some codes. Well, same with mine, enough talking already. Let's look at some example programs on how to define a variable. Although, an advice before that. The example programs in this book are screenshots of code written in the **IntelliJ IDEA code editor**. One major advantage of that is, as the reader, you get to do a lot of hands-on practice by typing out the code from the screenshots into your own editor, which would help quicken your learning, because typing out code and reflecting on it during the process is the best ways to learn programming. I would therefore advise that you take out time to do as much practice as possible. With that said, let's proceed to see our first few lines of Dart code.

### **int**

The int type short for integer, is used to define variables that can only store whole numbers, i.e. numbers that do not contain a decimal point.



The screenshot shows an IDE window with a file named 'main.dart'. The code is as follows:

```
1 void main() {  
2     int age = 50;  
3  
4     print(age);  
5 }  
6
```

Below the code editor, there is a 'Run' button and a console output area. The console shows the command 'dart main.dart' and the output '50'. The URL 'http://127.0.0.1:61' is also visible in the console area.

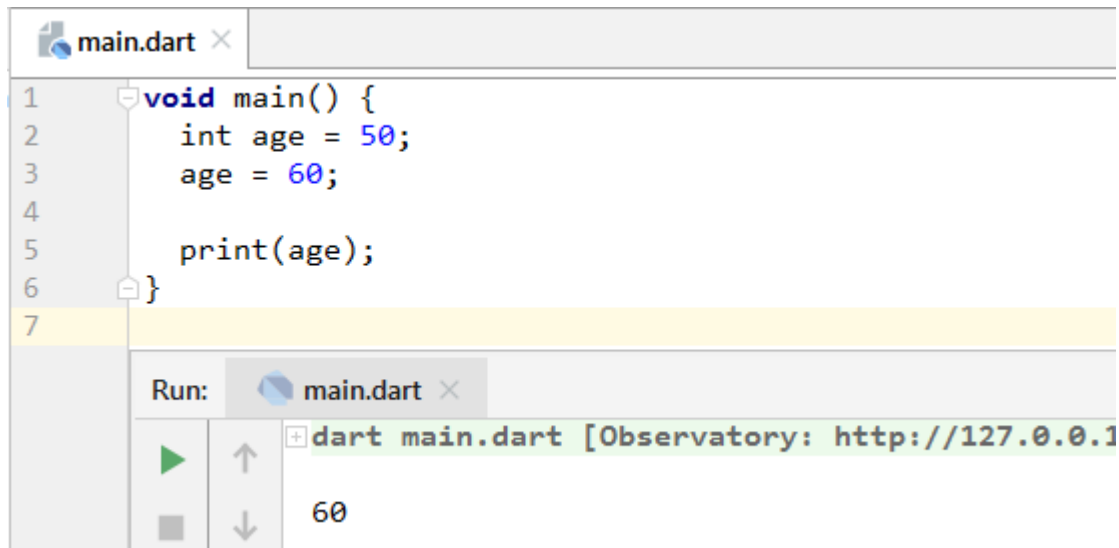
Screenshot 2.1

The program above contains a variable definition. A variable definition consists of two parts. One is the **declaration** part, while the other is the **initialization** part. In our program, which spans from line 1 to line 5. It is on line 2 that we have the variable definition. The variable has the name, *age* and it has been assigned the value 50. On that line, the part before the equality sign (=) is the **declaration** part. It consists of the variable type and the name of the variable, while the part after the equality sign is the **initialization** part, it consists of the value to be assigned to the variable. Observe that after the value 50, there's a semicolon, the semicolon signifies an end of statement. That is how you end a code statement in the Dart language, in this case, the variable definition statement. It's similar to how you end a written statement in a normal human language, say English for example, where you would use a full stop (.).

Let's further dissect each of those parts (the declaration and the initialization part). The word **int** as explained earlier, is a special type in Dart that allows you declare a variable that can only hold a number (an integer). Integers are numbers that don't have a decimal point, e.g. -3, -2, -1, 0, 1, 2, 3, etc. After the type **int**, next is the word *age*, which is the variable name. Variable names are user invented words, they're words that you and I come up with when writing programs. When you define a variable, you have to give the variable a name, so that you can use that name whenever you want to access the value that the variable refers to. Just after the variable name, we have the equality symbol (=). This is referred to as the **assignment operator**, not to be confused with the equality symbol that is used in mathematics, which is used to represent the fact that, what is at the left of the symbol, is exactly the same as what is at the right. After the assignment operator, is the number 50, which is the actual value that is passed to the variable, and can be used in the program. Here, we're simply printing out the value, which is why in the console view, you find the value 50. The printing is done with the help of the print function that you find on line 4. A function performs some operation.

The print function performs the operation of outputting a variable's value to the console view. More on functions in chapter 5.

**It is possible to reassign a new value to a variable after the first assignment, an example of how that is done is shown below.**



The screenshot shows an IDE window titled 'main.dart'. The code is as follows:

```
1 void main() {  
2   int age = 50;  
3   age = 60;  
4  
5   print(age);  
6 }  
7
```

Below the code editor is a 'Run' button and a console output area. The console shows the command 'dart main.dart [Observatory: http://127.0.0.1]' and the output '60'.

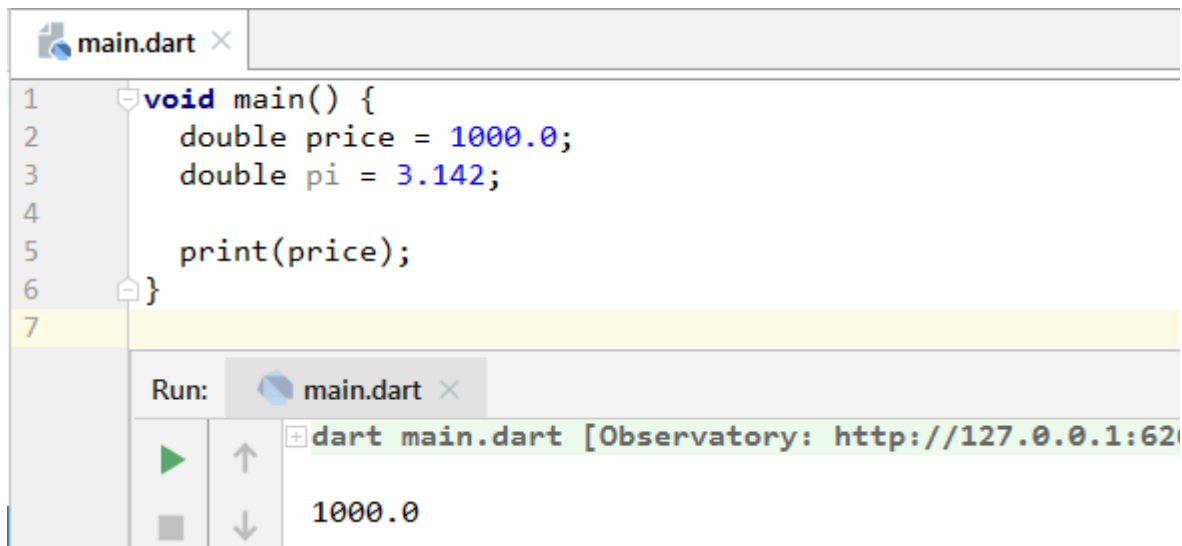
Screenshot 2.2

You can see that when the *age* variable was printed out, we got the recent value that was assigned to it, i.e. 60.

**Note that an attempt to assign a value that is not an integer to a variable of type *int*, results in an error. This is the same with other types. Once a variable has been declared to be of a particular type, the variable can only contain values that are of that type.**

## double

The type *double*, just like *int*, is also used for declaring a variable that can hold a number, although numbers that contain a decimal point. An example of this is shown below.



The screenshot shows an IDE window with a file named `main.dart`. The code is as follows:

```
1 void main() {  
2     double price = 1000.0;  
3     double pi = 3.142;  
4  
5     print(price);  
6 }  
7
```

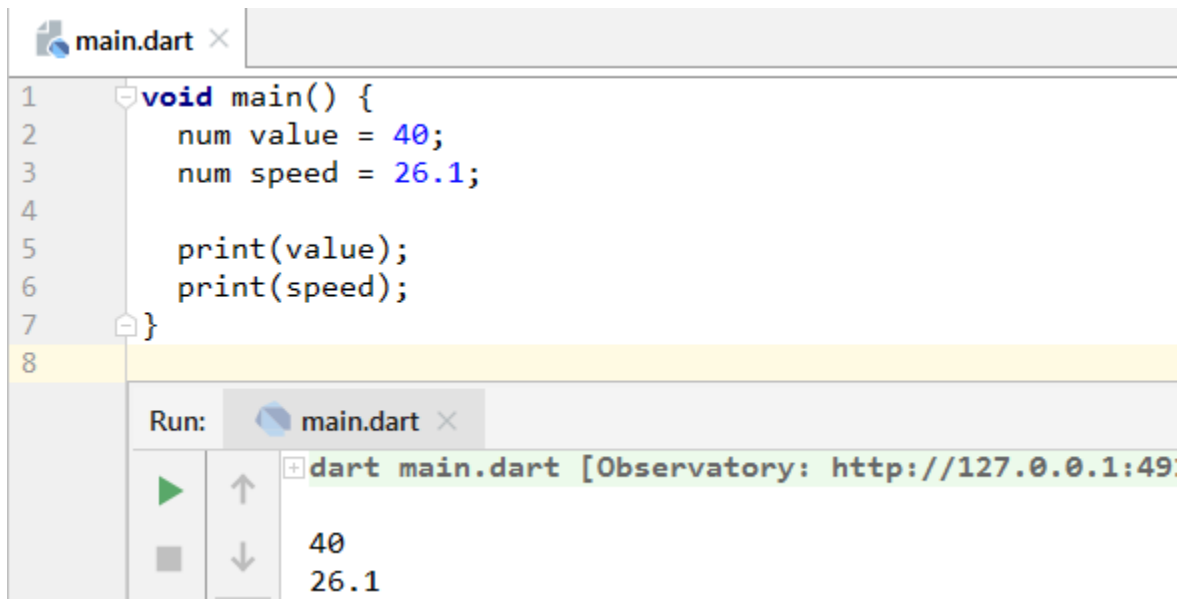
Below the code editor, there is a 'Run:' section. It shows a green play button, a button with an upward arrow, and a button with a downward arrow. To the right of these buttons, the text `dart main.dart [Observatory: http://127.0.0.1:621...]` is displayed. Below this, the output `1000.0` is shown.

Screenshot 2.3

Declaring a variable to be of type **double**, uses the same **syntax** as declaring a variable to be of type **int**, or of any other type. The word **syntax** refers to how the various parts of a program statement are structured. Just like every other programming language, Dart has its own language syntax.

As a Dart programmer, you need to have a good knowledge of the language syntax, so as to write proper Dart code, i.e. code that doesn't contain errors.

When it comes to defining number variables, we're not just limited to using the **int** and **double** types. There is a keyword (though not a type), that can be used to define number variables as well. It is called **num**. When a variable is defined using the **num** keyword, the variable is able to contain either integer values or decimal values. An example of this is shown below.



The screenshot shows an IDE window with a file named `main.dart`. The code defines a `main` function that declares two numeric variables, `value` and `speed`, and prints their values. Below the code editor, a 'Run' button is visible, and the execution output shows the values `40` and `26.1`.

```
1 void main() {  
2     num value = 40;  
3     num speed = 26.1;  
4  
5     print(value);  
6     print(speed);  
7 }  
8
```

Run: `main.dart` x

`dart main.dart [Observatory: http://127.0.0.1:49152]`

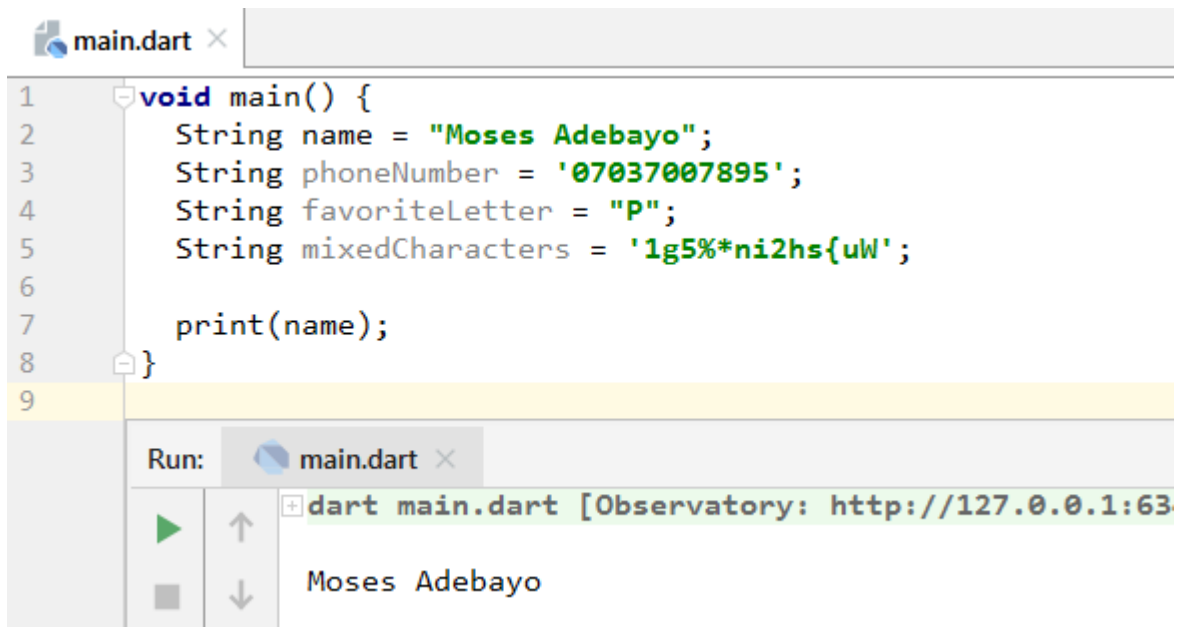
40  
26.1

Screenshot 2.4

## String

The **String** type is used to declare a variable that can hold a sequence of characters. A good example is the name of a person (e.g. Moses), or any bunch of characters, even a single character (e.g. A) can be treated as a String.

Observe that the type **String**, starts with an uppercase letter, unlike `int` and `double`. So, remember to always spell it that way.



The screenshot shows an IDE window with a file named `main.dart`. The code defines a `main` function that declares four String variables: `name`, `phoneNumber`, `favoriteLetter`, and `mixedCharacters`. It then prints the value of `name`. Below the code editor, a 'Run' button is visible, and the execution output shows the string `Moses Adebayo`.

```
1 void main() {  
2     String name = "Moses Adebayo";  
3     String phoneNumber = '07037007895';  
4     String favoriteLetter = "P";  
5     String mixedCharacters = '1g5%*ni2hs{uW';  
6  
7     print(name);  
8 }  
9
```

Run: `main.dart` x

`dart main.dart [Observatory: http://127.0.0.1:6354]`

Moses Adebayo

Screenshot 2.5

Any arbitrary sequence of characters can be used to form a String value. Actually, any character or symbol you type in from the keyboard can be used as the value for a String, so long as you tell Dart to treat that character, number, symbol, or set of characters as a String, by wrapping it in double or single quotation marks, as shown in the example code above.

Before proceeding, there are some concepts I want to draw your attention to, when it comes to choosing the name for a variable.

First, observe how the variables in the previous program are spelt (phoneNumber, favoriteLetter, and mixedCharacters), they are a combination of two words.

The thing is, there are some rules guiding how the name for a variable should be composed and how they should be spelt. Let's explore these rules.

1. Variable names in Dart can only begin with a letter or an underscore (\_). Using any other symbol or character, would result in an invalid name.
2. When you choose a name for a variable, ensure that the name is related to the value you intend to store in it. This will help you and other programmers who go through your code to easily understand what value a variable contains and what it is being used for.
3. A variable name cannot be the same as a Dart keyword. We will get to know what keywords are when we talk about them in chapter 3.
4. This 4<sup>th</sup> rule is more or less a recommendation. It explains why the variable names that are made up of more than one word in the program above were spelt the way they're.

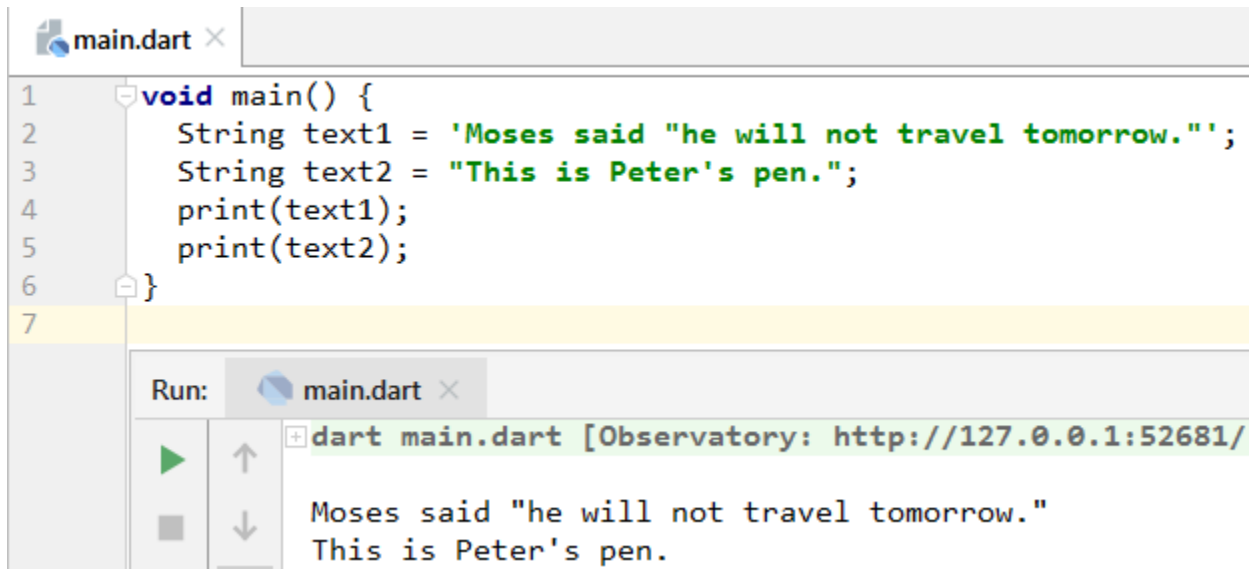
When you have a variable name that is made up of more than one word, you could either go with the style of making the first letter of each word, except the first word, an uppercase. This is referred to as **lowerCamelCasing**. Or you could go with the style of using an underscore to separate each word (e.g. phone\_number, favorite\_letter and mixed\_characters etc.). Adhering to this recommendation, brings about more code readability and ensures consistency.

We've seen how a String value can be defined using either single or double quotation marks. However, there are times that you're required to use either single or double quotation marks.

Assuming you have a text such as *Moses said "he will not come to class today"* and you want to make it a String value. This text contains double quotation marks within it,

therefore it won't be possible to make it a String value by wrapping it with double quotation marks. Doing so would result in an error. To make it work, it should be wrapped with single quotation marks.

Also, if you have some text that contains a single quotation mark, then in order to make that text a valid String value, it must be wrapped with double quotes.



The screenshot shows an IDE window titled 'main.dart'. The code is as follows:

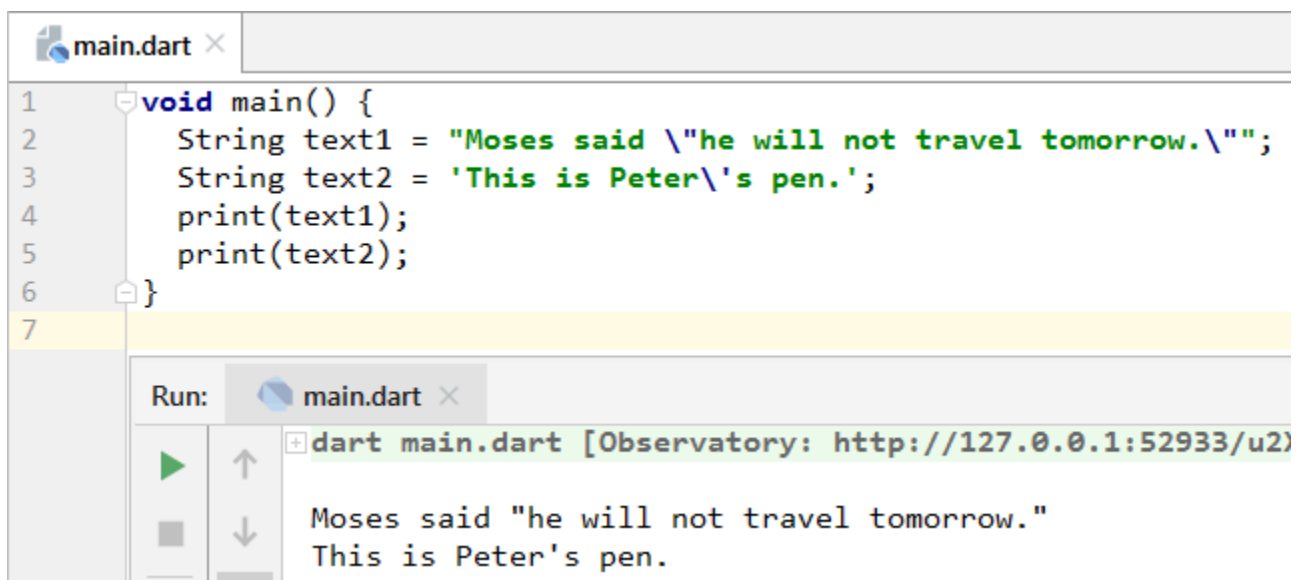
```
1 void main() {  
2   String text1 = 'Moses said "he will not travel tomorrow."';  
3   String text2 = "This is Peter's pen.";  
4   print(text1);  
5   print(text2);  
6 }  
7
```

Below the code editor, the 'Run' button is clicked, and the output is displayed in a console window. The output shows the two strings being printed:

```
dart main.dart [Observatory: http://127.0.0.1:52681/  
Moses said "he will not travel tomorrow."  
This is Peter's pen.
```

Screenshot 2.6

Another way such String values that contain single or double quotes can be defined is with the use of the escape character (\). This \ is one of many escape characters that can be used to alter the normal behavior of a String value.



The screenshot shows an IDE window titled 'main.dart'. The code is as follows:

```
1 void main() {  
2   String text1 = "Moses said \"he will not travel tomorrow.\"";  
3   String text2 = 'This is Peter\'s pen.';  
4   print(text1);  
5   print(text2);  
6 }  
7
```

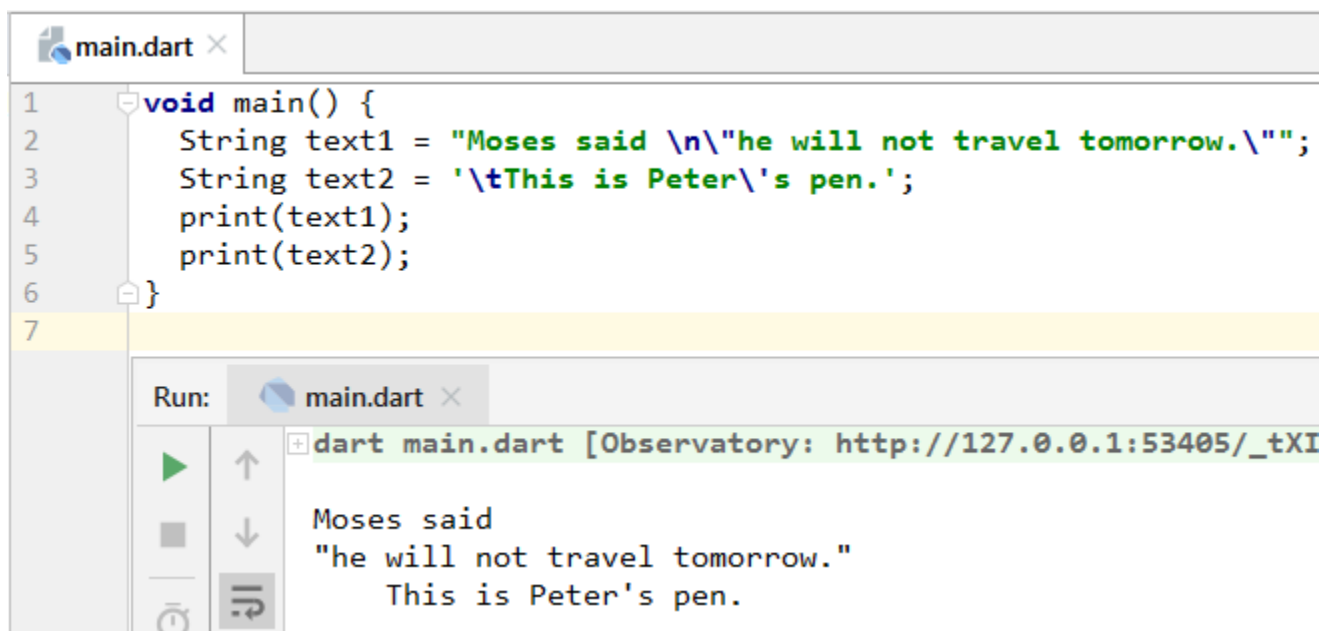
Below the code editor, the 'Run' button is clicked, and the output is displayed in a console window. The output shows the two strings being printed:

```
dart main.dart [Observatory: http://127.0.0.1:52933/u2  
Moses said "he will not travel tomorrow."  
This is Peter's pen.
```

## Screenshot 2.6

The `\` used in a String value, tells Dart to treat whatever character that comes after it just as it is, in this case, the double and single quotation marks. With that, it becomes possible to wrap the String value on line 2 with double quotes, while having double quotes inside the String value. Same with the String value on line 3.

Other common escape characters include the newline (`\n`) and the tab (`\t`). When Dart is printing a String value and it encounters the newline (`\n`) character, it jumps to the next line and continues printing it on that line. While the tab (`\t`) character tells Dart to include spaces in a String, similar to that which is made with the tab key on the keyboard.



The screenshot shows an IDE window titled 'main.dart'. The code is as follows:

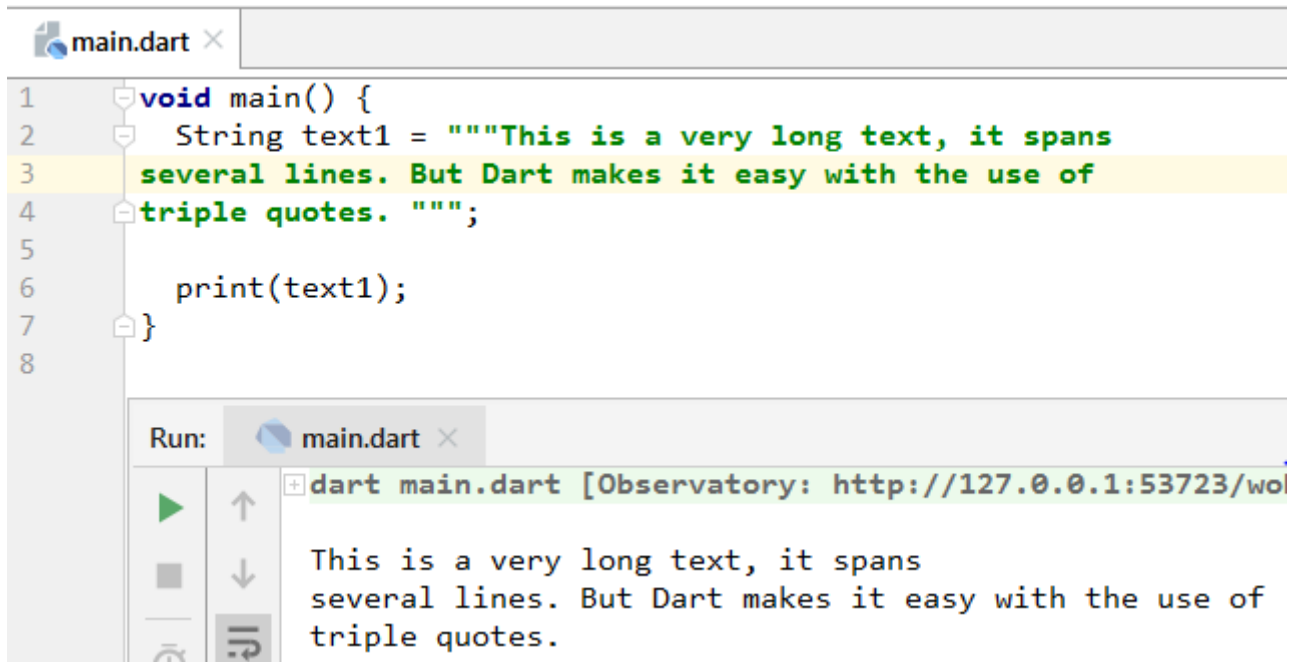
```
1 void main() {  
2   String text1 = "Moses said \n\"he will not travel tomorrow.\"";  
3   String text2 = '\tThis is Peter\'s pen.';  
4   print(text1);  
5   print(text2);  
6 }  
7
```

Below the code editor is a 'Run:' panel. It shows the command 'dart main.dart' and the Observatory URL 'http://127.0.0.1:53405/\_tXI'. The output of the program is displayed as follows:

```
Moses said  
"he will not travel tomorrow."  
    This is Peter's pen.
```

## Screenshot 2.7

It is possible to define a String value that spans several lines. To do that, simply wrap the String value with triple quotes (single or double) as shown below.



The screenshot shows an IDE window with a file named `main.dart`. The code is as follows:

```
1 void main() {  
2   String text1 = """"This is a very long text, it spans  
3   several lines. But Dart makes it easy with the use of  
4   triple quotes. """";  
5  
6   print(text1);  
7 }  
8
```

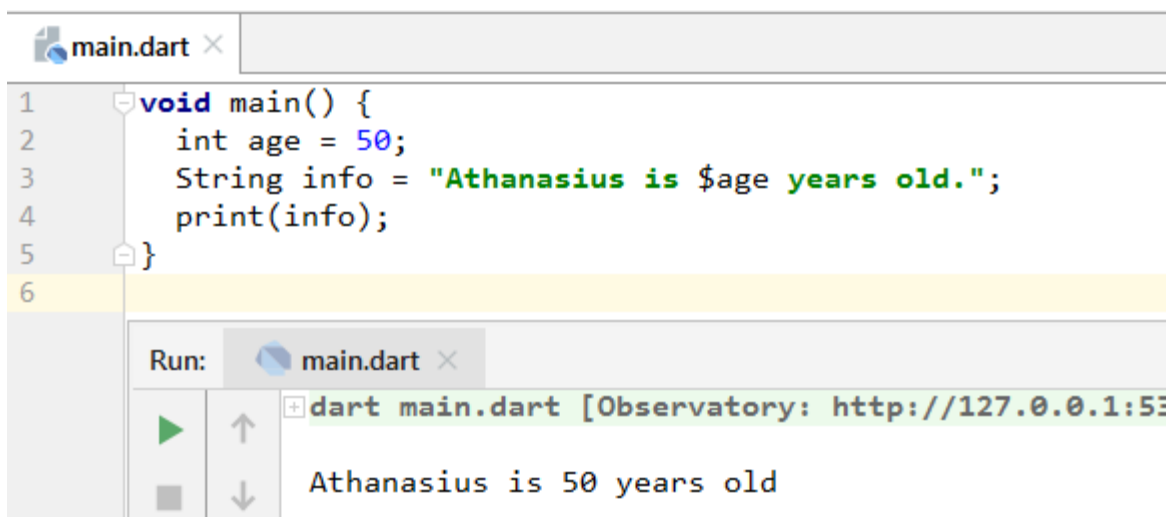
Below the code editor, the 'Run' panel shows the execution of `main.dart`. The output is:

```
dart main.dart [Observatory: http://127.0.0.1:53723/wol  
This is a very long text, it spans  
several lines. But Dart makes it easy with the use of  
triple quotes.
```

Screenshot 2.8

## String Interpolation

It is possible to include a variable inside of a String value. A process known as **string interpolation**.



The screenshot shows an IDE window with a file named `main.dart`. The code is as follows:

```
1 void main() {  
2   int age = 50;  
3   String info = "Athanasius is $age years old.";  
4   print(info);  
5 }  
6
```

Below the code editor, the 'Run' panel shows the execution of `main.dart`. The output is:

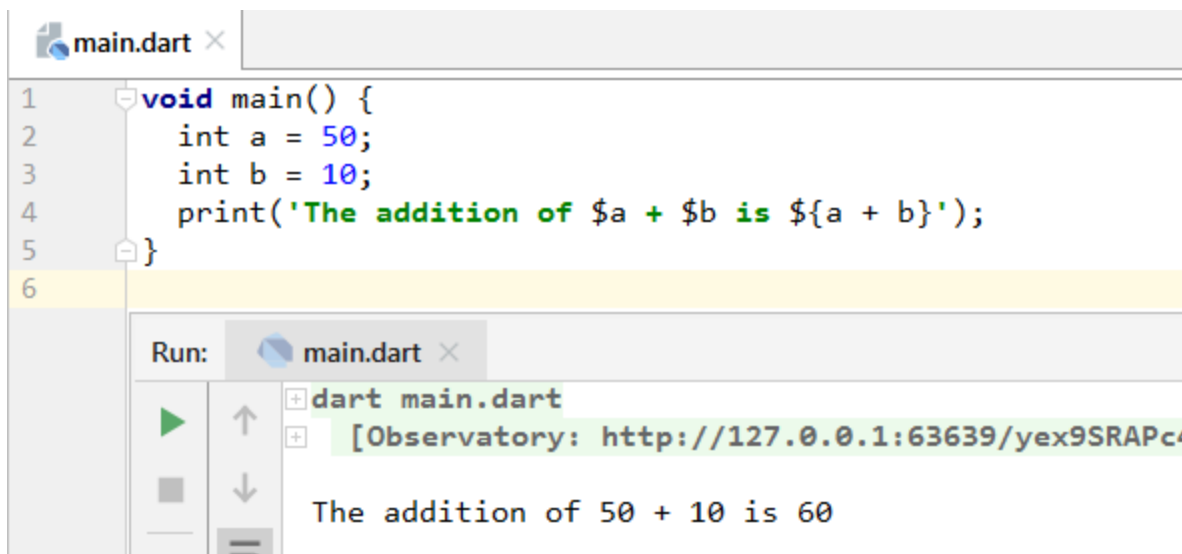
```
dart main.dart [Observatory: http://127.0.0.1:53  
Athanasius is 50 years old
```

Screenshot 2.9

Using the dollar sign (\$) to include a variable in a String value is referred to as string interpolation. What the dollar sign does, is to convert the value of the variable into a String value, so that it can be safely included as part of the String value.



When you need to include an expression in a String value, then curly brackets have to be used in wrapping the expression as shown below.



The screenshot shows an IDE window titled 'main.dart'. The code is as follows:

```
1 void main() {  
2     int a = 50;  
3     int b = 10;  
4     print('The addition of $a + $b is ${a + b}');  
5 }  
6
```

Below the code editor is a 'Run' panel. It shows the command 'dart main.dart' and the output:

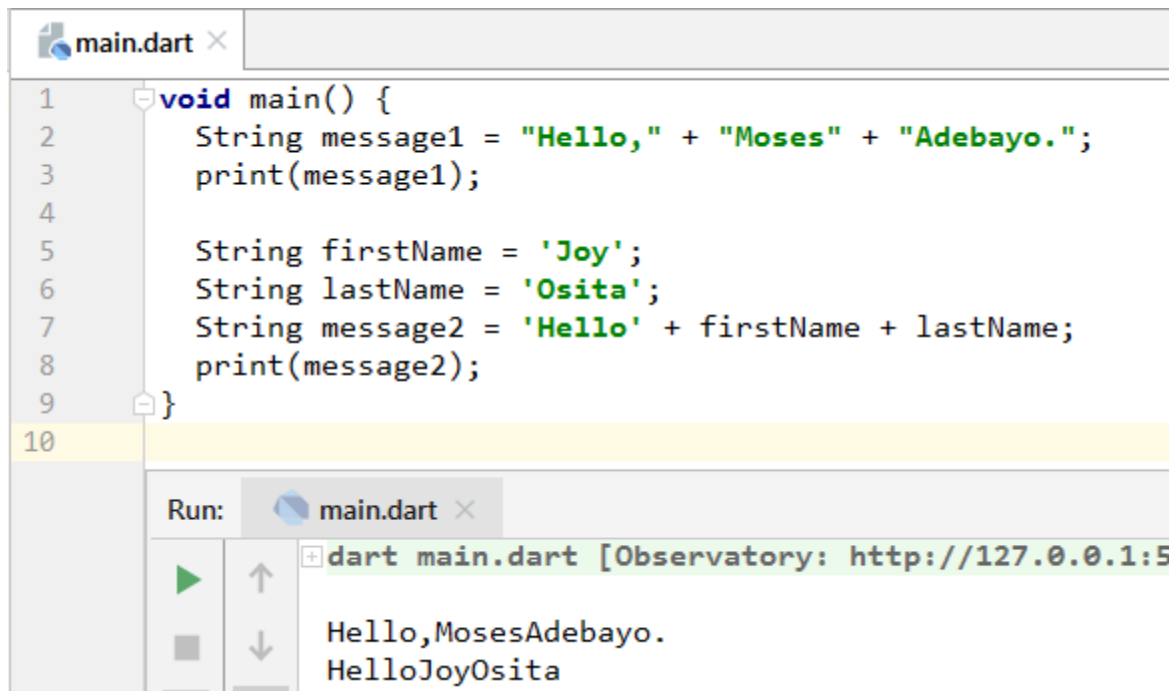
```
[Observatory: http://127.0.0.1:63639/yex9SRAPc]  
  
The addition of 50 + 10 is 60
```

Screenshot 2.10

Here, we've performed a simple addition operation, but more complex expressions, like function calls, calling methods on objects, etc. can be performed in a String value, using this same approach. You shall learn all about those in the later chapters.

## String Concatenation

It is possible to join two or more String values together to form a single String value. This is a process known as **string concatenation**.



```
1 void main() {  
2   String message1 = "Hello," + "Moses" + "Adebayo.";  
3   print(message1);  
4  
5   String firstName = 'Joy';  
6   String lastName = 'Osita';  
7   String message2 = 'Hello' + firstName + lastName;  
8   print(message2);  
9 }  
10
```

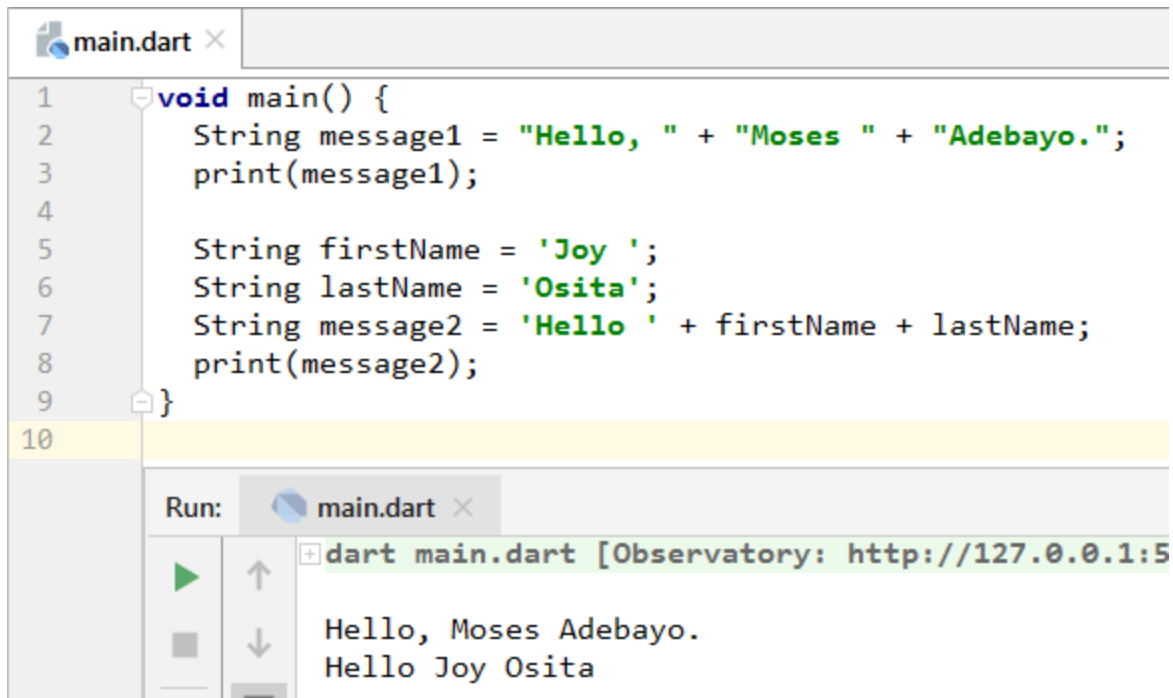
Run: main.dart x

+ dart main.dart [Observatory: <http://127.0.0.1:5>]

Hello,MosesAdebayo.  
HelloJoyOsita

Screenshot 2.11

The plus symbol (+) joins two or more String values together. Although, for the program above, there's an issue with the way the String values are printed out. There is no space between them. That is because in the code, the String values do not contain any space around them, so Dart simply joins them the way they're and prints them out like that. To remedy this, we can add spaces in the String values as we would want them to be when they're printed out. There are so many ways of achieving this. Here are two possible ways.



```
1 void main() {
2   String message1 = "Hello, " + "Moses " + "Adebayo.";
3   print(message1);
4
5   String firstName = 'Joy ';
6   String lastName = 'Osita';
7   String message2 = 'Hello ' + firstName + lastName;
8   print(message2);
9 }
10
```

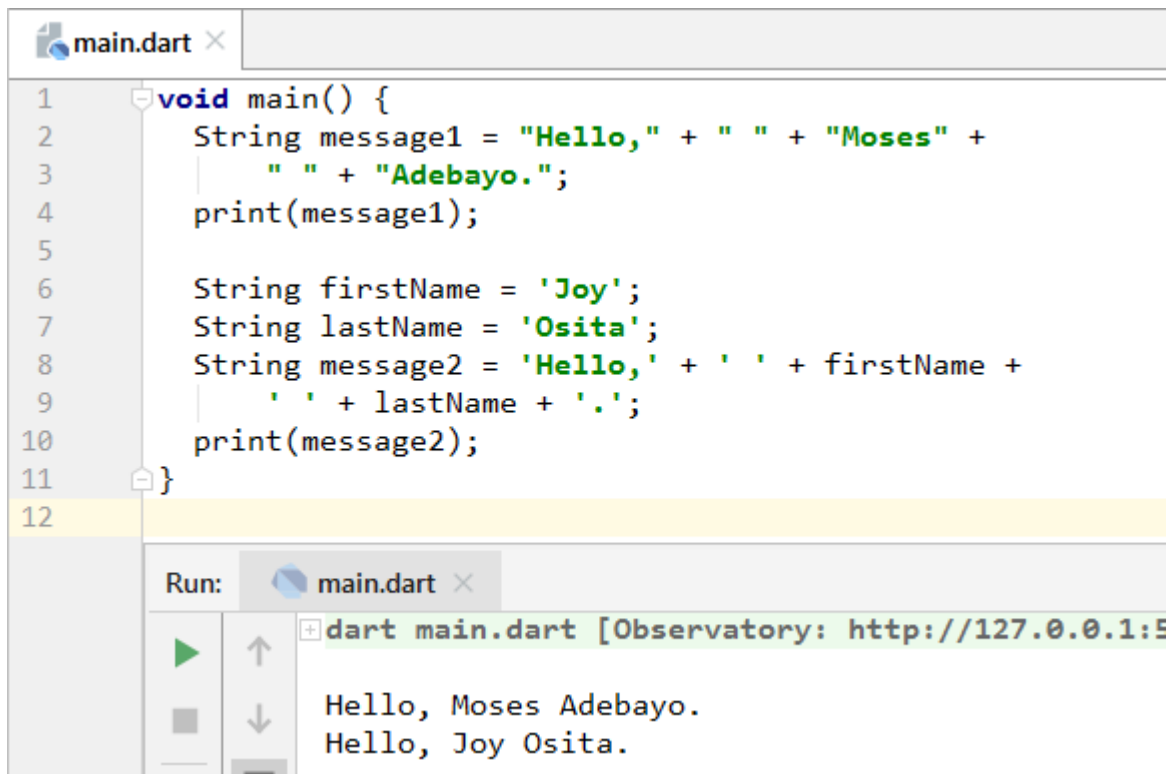
Run: main.dart x

dart main.dart [Observatory: <http://127.0.0.1:5>]

Hello, Moses Adebayo.  
Hello Joy Osita

Screenshot 2.12

In the program above, for the first String values that would be joined together and put in the *message1* variable, we included a space in the String value ("Hello "), also in the String value ("Moses "). When the String values are joined together, it becomes *Hello, Moses Adebayo*. The same thing was done for the *message2* variable.



The screenshot shows an IDE window titled 'main.dart'. The code defines a `main` function that prints two messages. The first message is constructed using double quotes and spaces: `"Hello," + " " + "Moses" + " " + "Adebayo."`. The second message uses single quotes for names: `'Hello,' + ' ' + firstName + ' ' + lastName + '.'`. Below the code editor, a 'Run' button is visible, and the output console shows the execution results: `dart main.dart [Observatory: http://127.0.0.1:5...]` followed by the printed messages: `Hello, Moses Adebayo.` and `Hello, Joy Osita.`

```
1 void main() {  
2   String message1 = "Hello," + " " + "Moses" +  
3     " " + "Adebayo.";  
4   print(message1);  
5  
6   String firstName = 'Joy';  
7   String lastName = 'Osita';  
8   String message2 = 'Hello,' + ' ' + firstName +  
9     ' ' + lastName + '.';  
10  print(message2);  
11 }  
12
```

Run: main.dart x

dart main.dart [Observatory: http://127.0.0.1:5...]

Hello, Moses Adebayo.  
Hello, Joy Osita.

Screenshot 2.13

Here, we created a String value that contains a single space character using either single or double quotes ( ' , " ") and joined it to the other String values.

## bool

The bool type is a very interesting type.

Analogy time: Imagine mama Dayo asking Dayo a question, and she expects him to give her a yes (true) or no (false) answer.

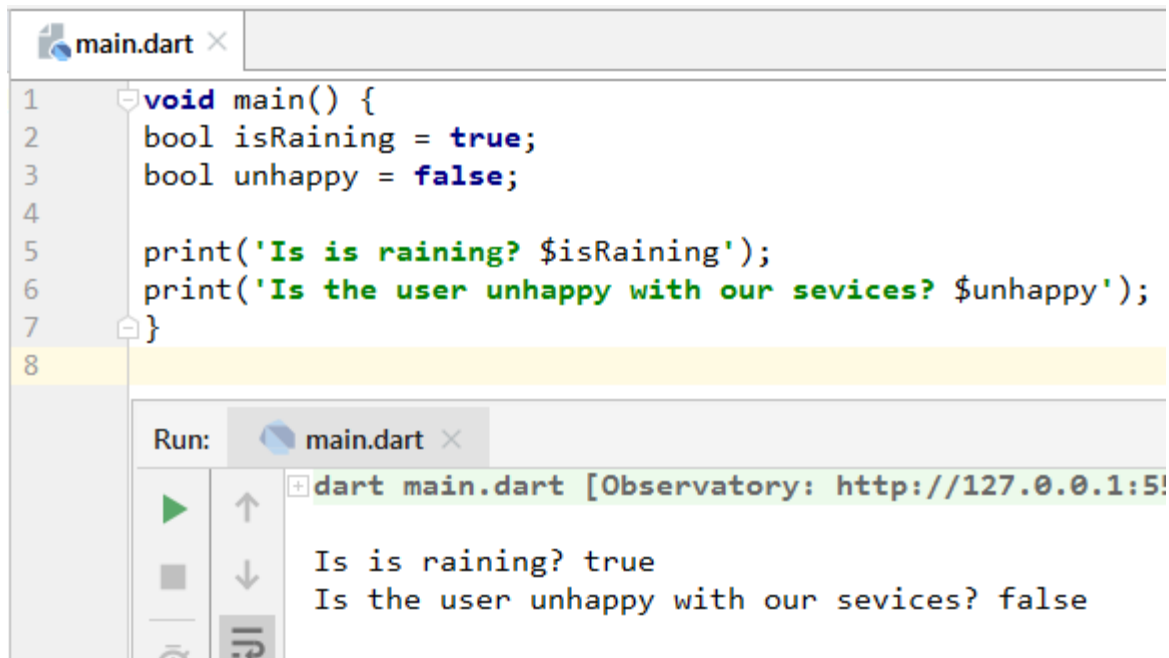
**Mama Dayo:** Dayo, did you take meat from the cooking pot?

**Dayo:** Mummy, I did not enter the kitchen today at all.

**Mama Dayo:** Dayo, I'm going to ask you for the last time, did you take the meat, yes or no?

With that last question, Dayo is going to have to provide his mum with a yes (true) or no (false) answer, nothing else, no stories.

That's exactly what you get when you declare a variable to be of type bool. A bool variable can only hold the value of **true** or **false**, nothing else. Please note that **true** or **false** are not String values. They're not wrapped in single or double quotes.



The screenshot shows an IDE window titled 'main.dart'. The code is as follows:

```
1 void main() {  
2   bool isRaining = true;  
3   bool unhappy = false;  
4  
5   print('Is is raining? $isRaining');  
6   print('Is the user unhappy with our sevices? $unhappy');  
7 }  
8
```

Below the code editor is a 'Run' panel. It shows the command 'dart main.dart [Observatory: http://127.0.0.1:5182]' and the output of the program:

```
Is is raining? true  
Is the user unhappy with our sevices? false
```

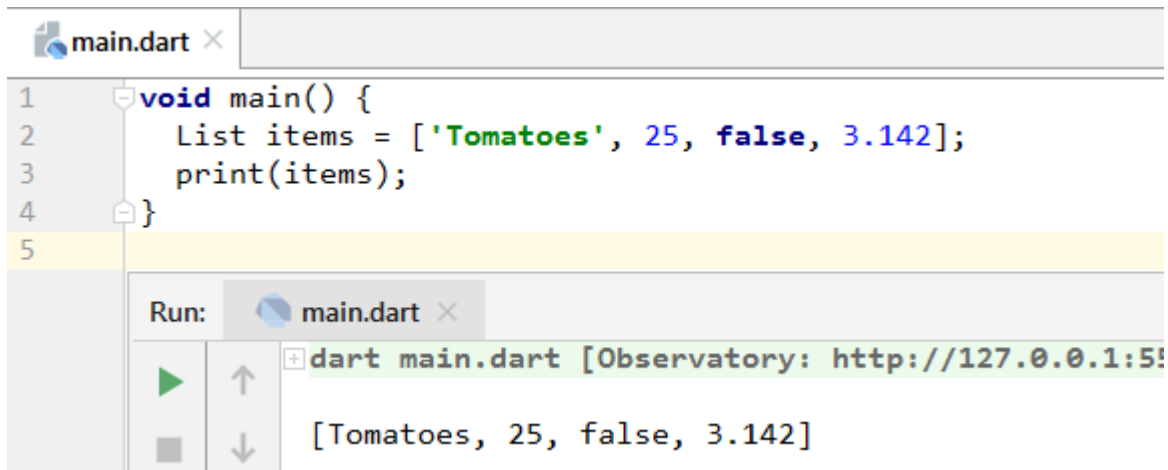
Screenshot 2.14

bool values (true or false) are most times used in writing code that gets executed conditionally. You could write a program that checks the value stored in a bool variable, if it is true, then the program would behave in a certain way, if it is false, then it would behave in a different way. You will learn all of that when we get to chapter 6 (control flow statements).

## List

In the real-world, you often find yourself putting together a list of some items. Take Mama Nkechi for example, whenever she wants to go to the market, she would get a piece of paper and write down the names of the items she wants to buy, and she would end up with a list of items. When little Ibrahim first learnt how to count the numbers, he would make several long lists of numbers, numbering 1 – 100 and so on.

A list in Dart is no different, it is made up of similar or dissimilar items. A list could contain values of different types (int, double, String, etc.), or values of the same type.

The screenshot shows an IDE window with a file named 'main.dart'. The code is as follows:

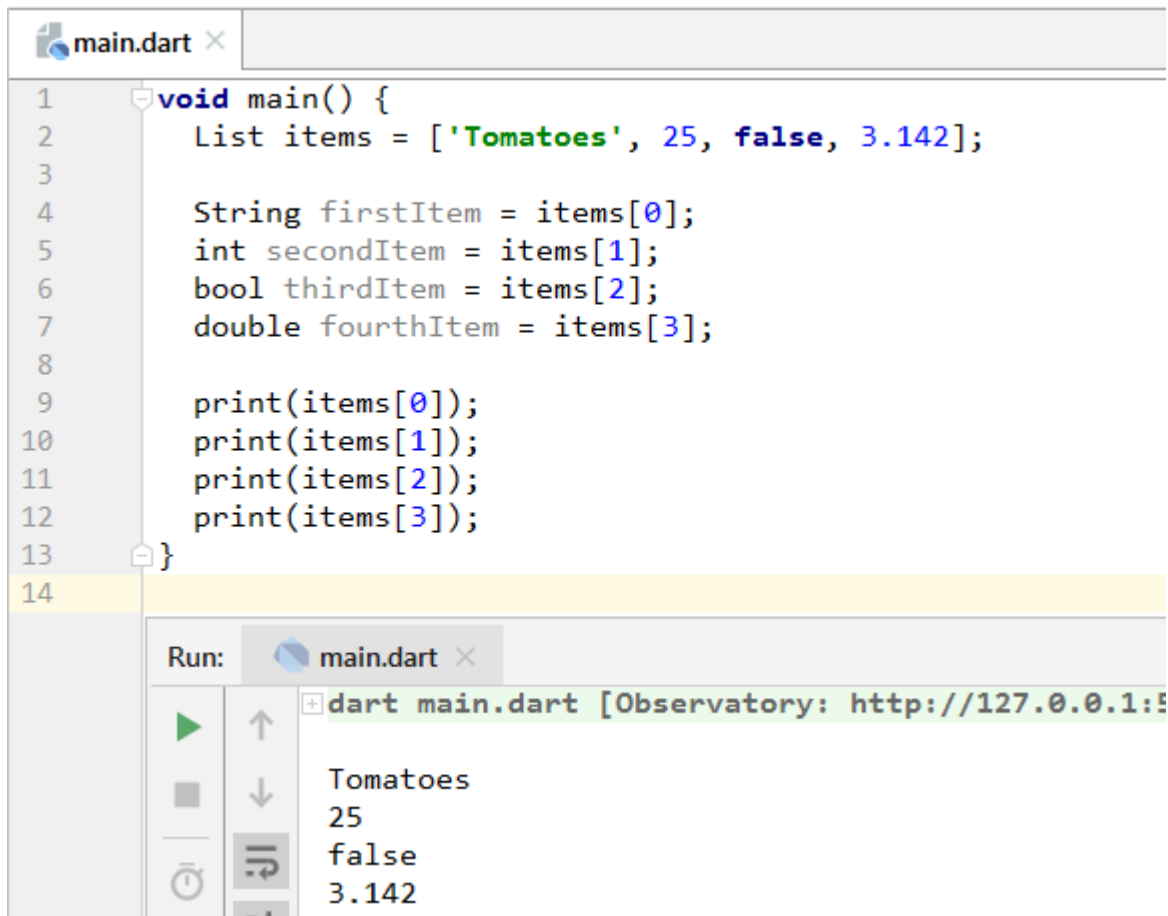
```
1 void main() {  
2   List items = ['Tomatoes', 25, false, 3.142];  
3   print(items);  
4 }  
5
```

Below the code editor, there is a 'Run' button and a console output area. The console shows the command 'dart main.dart [Observatory: http://127.0.0.1:59121]' and the output '[Tomatoes, 25, false, 3.142]'. The output is displayed in a green monospace font.

Screenshot 2.15

The code above shows how a list can be created in Dart. You can see the list is made up of items (values) of different types. Observe how all the items are wrapped with square brackets and each item in the list is separated with a comma (,). That's how Dart knows the number of items you have in a list. The square brackets for defining a list are referred to as the **list literal**. Later, in this book, you shall learn how to use the **List constructor** to create a list, it's more of an advanced way of doing it, and so we shall defer it till chapter 8, when you must have learnt about **Object Oriented Programming (OOP)**. One thing we can also do, is to create a list that contains items (values) of a particular type, e.g. a list that contains only values of type String, or int, or double etc. We shall explore how to create such lists, when we get to the topic of **generics** in chapter 8.

One very important thing to note about a list in Dart and other programming languages is that, lists use zero-based index. What does that mean? Using the analogy I cited earlier, when Mama Nkechi makes a list of the items she hopes to buy in the market, she gives the first item the number one (1), the next item she gives the number two (2), and so on. This is the normal way we number items in real-life. However, it is not the same in Dart, Dart doesn't assign the number 1 to the first item in a list, instead it assigns the number zero (0). That's what zero-based index means. The index of a list item refers to the position of the item in the list. So, if we want to retrieve any item from the list, we would use its index. Take the list in the below program for example, assuming we want to individually access the items in the list, to access the first item in the list, we would access it using its index, which is zero (0), the next item has the index 1, the next item has the index 2 and so on.



The screenshot shows an IDE window with a file named `main.dart`. The code defines a `main` function that creates a list `items` containing `'Tomatoes'`, `25`, `false`, and `3.142`. It then retrieves each element from the list and prints it. Below the code editor, a 'Run' panel shows the execution output: `Tomatoes`, `25`, `false`, and `3.142`. The output is displayed in a console window titled `dart main.dart [Observatory: http://127.0.0.1:5...]`.

```
1 void main() {  
2   List items = ['Tomatoes', 25, false, 3.142];  
3  
4   String firstItem = items[0];  
5   int secondItem = items[1];  
6   bool thirdItem = items[2];  
7   double fourthItem = items[3];  
8  
9   print(items[0]);  
10  print(items[1]);  
11  print(items[2]);  
12  print(items[3]);  
13 }  
14
```

Run: `main.dart` ×

`dart main.dart [Observatory: http://127.0.0.1:5...]`

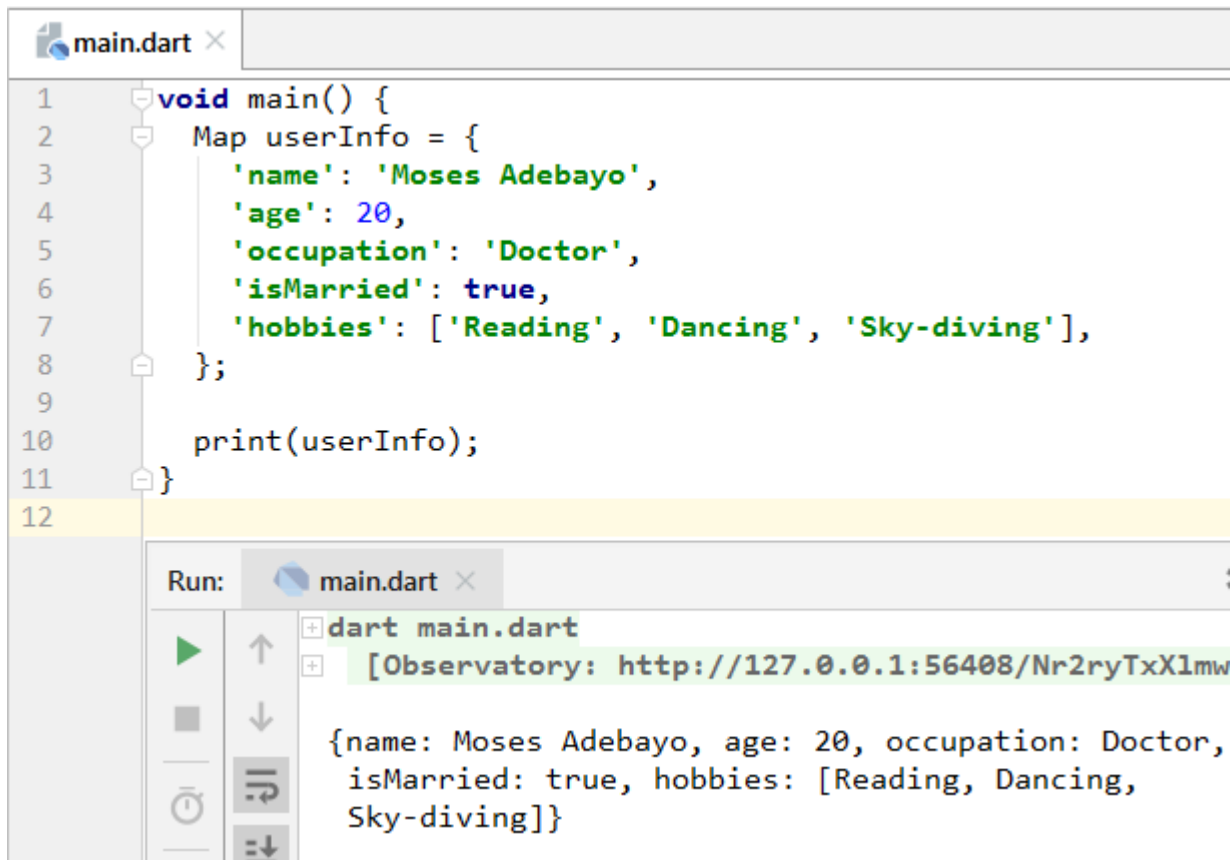
Tomatoes  
25  
false  
3.142

Screenshot 2.16

Observe how on lines 4 through 7, individual values in the list were retrieved and put into variables of their respective type. While on lines 9 through 12, the values were retrieved and printed.

## Map

The Map type isn't as straightforward as the previous types we've looked at, so please pay close attention to its explanation. When you declare a variable to be of type Map, the value you assign to it, is a map that contains **keys** and **values**, where each key corresponds to a value in the map.



The screenshot shows an IDE window titled 'main.dart'. The code defines a `main()` function that creates a `Map` named `userInfo` with the following data: `'name': 'Moses Adebayo', 'age': 20, 'occupation': 'Doctor', 'isMarried': true, 'hobbies': ['Reading', 'Dancing', 'Sky-diving']`. The map is then printed using `print(userInfo)`. Below the code editor, a 'Run' panel shows the execution of `dart main.dart`, displaying the output: `{name: Moses Adebayo, age: 20, occupation: Doctor, isMarried: true, hobbies: [Reading, Dancing, Sky-diving]}`. The output is shown in a console window with a URL: `[Observatory: http://127.0.0.1:56408/Nr2ryTxXlmw]`.

```
1 void main() {  
2   Map userInfo = {  
3     'name': 'Moses Adebayo',  
4     'age': 20,  
5     'occupation': 'Doctor',  
6     'isMarried': true,  
7     'hobbies': ['Reading', 'Dancing', 'Sky-diving'],  
8   };  
9  
10  print(userInfo);  
11 }  
12
```

Run: main.dart x

dart main.dart  
[Observatory: http://127.0.0.1:56408/Nr2ryTxXlmw]

{name: Moses Adebayo, age: 20, occupation: Doctor,  
isMarried: true, hobbies: [Reading, Dancing,  
Sky-diving]}

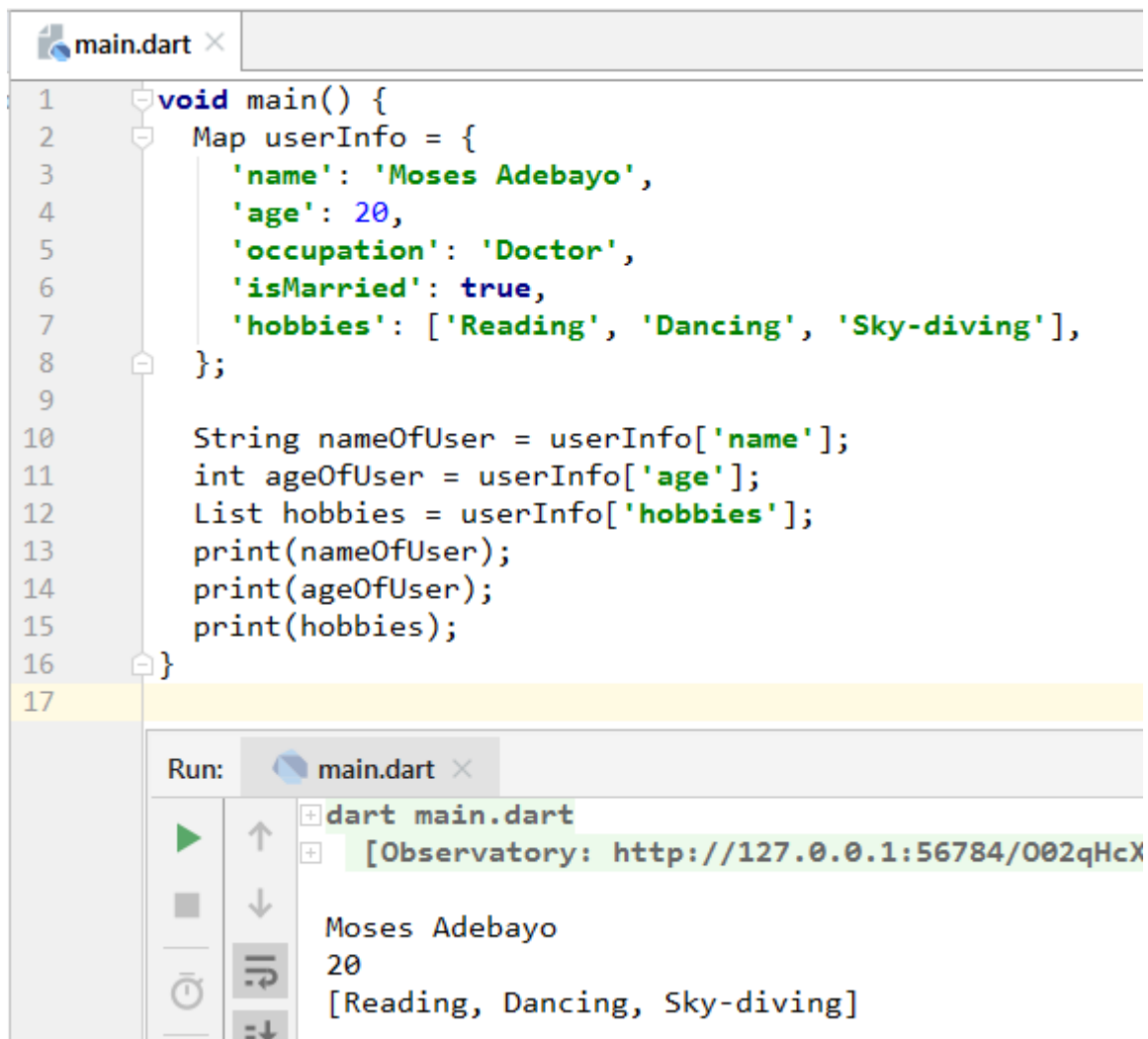
Screenshot 2.17

Observe the way a map is created. When we created a list, we wrapped the list items in square brackets (`[]`), but for a map, we use curly brackets (`{}`). The curly brackets for defining a map are referred to as the **map literal**. Later, in this book, you shall learn how to use the `Map` constructor to create a map, it's more of an advanced way of doing it, and so we shall defer it till chapter 8, when you must have learnt about **Object Oriented Programming (OOP)**.

Remember I said a map usually contains keys and values, I'm sure you can guess which are the keys and values in the map we created above. The keys in the map above are; *name*, *age*, *occupation*, *isMarried*, and *hobbies*, and they are all String values. While the values which the keys correspond to are; *Moses Adebayo*, *20*, *Doctor*, *true* and the list of String values. Also, observe how each key-value pair is separated with a comma. **Keys in a map are commonly string values and they must be unique**, but two or more values could be the same.

Other than printing out the full map, as we've done above, it is possible to access the individual values of the map, using their keys.





The screenshot shows an IDE window titled 'main.dart'. The code defines a `main()` function that creates a `Map` named `userInfo` with the following data: `'name': 'Moses Adebayo', 'age': 20, 'occupation': 'Doctor', 'isMarried': true, 'hobbies': ['Reading', 'Dancing', 'Sky-diving']`. It then retrieves the values for 'name', 'age', and 'hobbies' and prints them. The output console at the bottom shows the execution results: `Moses Adebayo`, `20`, and `[Reading, Dancing, Sky-diving]`.

```
1 void main() {  
2   Map userInfo = {  
3     'name': 'Moses Adebayo',  
4     'age': 20,  
5     'occupation': 'Doctor',  
6     'isMarried': true,  
7     'hobbies': ['Reading', 'Dancing', 'Sky-diving'],  
8   };  
9  
10  String nameOfUser = userInfo['name'];  
11  int ageOfUser = userInfo['age'];  
12  List hobbies = userInfo['hobbies'];  
13  print(nameOfUser);  
14  print(ageOfUser);  
15  print(hobbies);  
16 }  
17
```

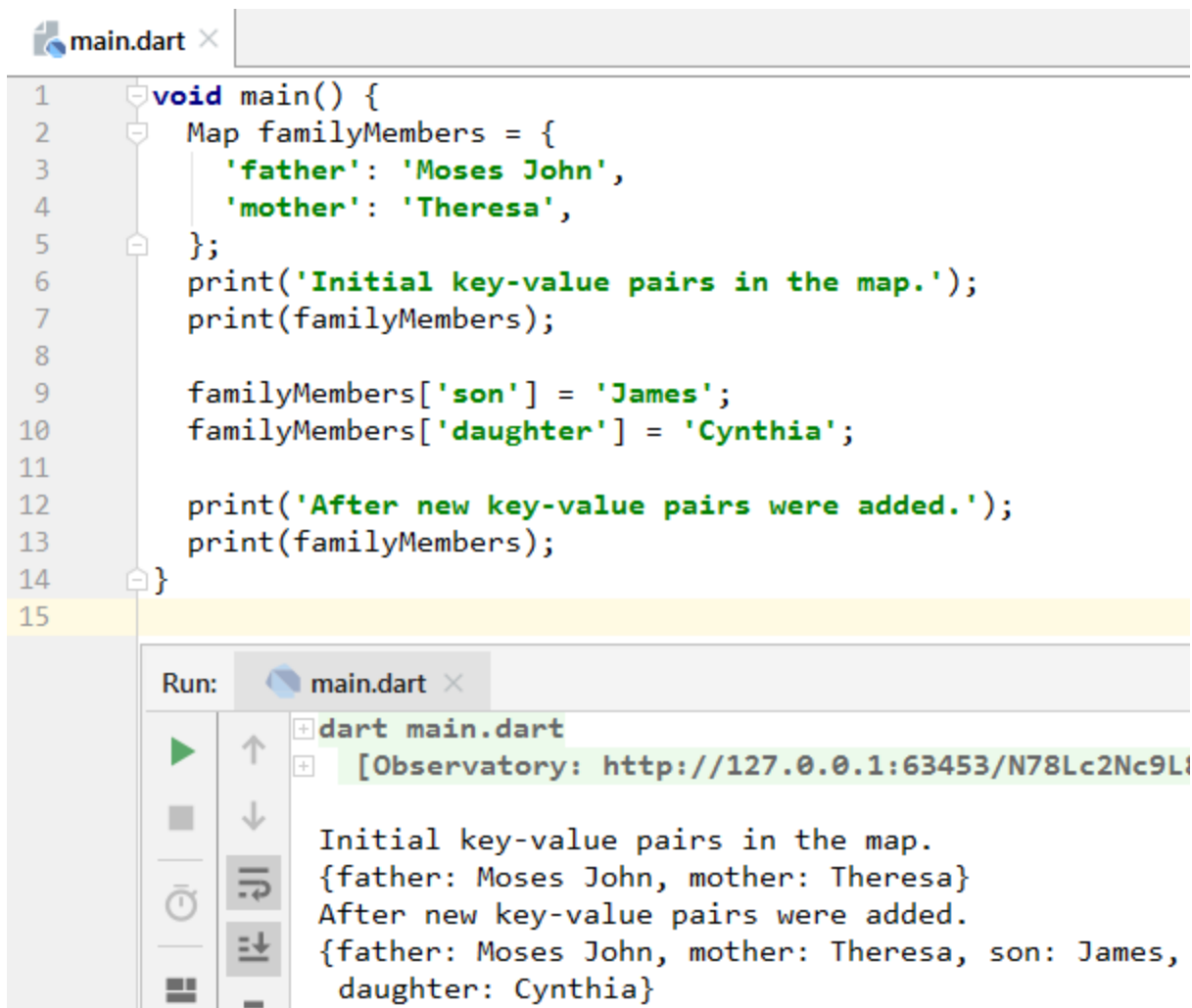
Run: main.dart

dart main.dart  
[Observatory: http://127.0.0.1:56784/002qHcX]

Moses Adebayo  
20  
[Reading, Dancing, Sky-diving]

Screenshot 2.18

Other than retrieving a value in a map through its key, we can also add a new key-value pair to the map, as shown in the following program, on lines 9 and 10.



The screenshot shows an IDE window with a file named `main.dart`. The code defines a `main` function that creates a `Map` named `familyMembers` with initial entries for 'father' and 'mother'. It then prints the map, adds entries for 'son' and 'daughter', and prints the map again. Below the code editor, a 'Run' panel shows the execution output for `dart main.dart`. The output includes an Observatory URL and the printed map contents at two different points in the program's execution.

```
1 void main() {
2   Map familyMembers = {
3     'father': 'Moses John',
4     'mother': 'Theresa',
5   };
6   print('Initial key-value pairs in the map.');
```

```
7   print(familyMembers);
8
9   familyMembers['son'] = 'James';
10  familyMembers['daughter'] = 'Cynthia';
11
12  print('After new key-value pairs were added.');
```

```
13  print(familyMembers);
14 }
15
```

Run: `main.dart`

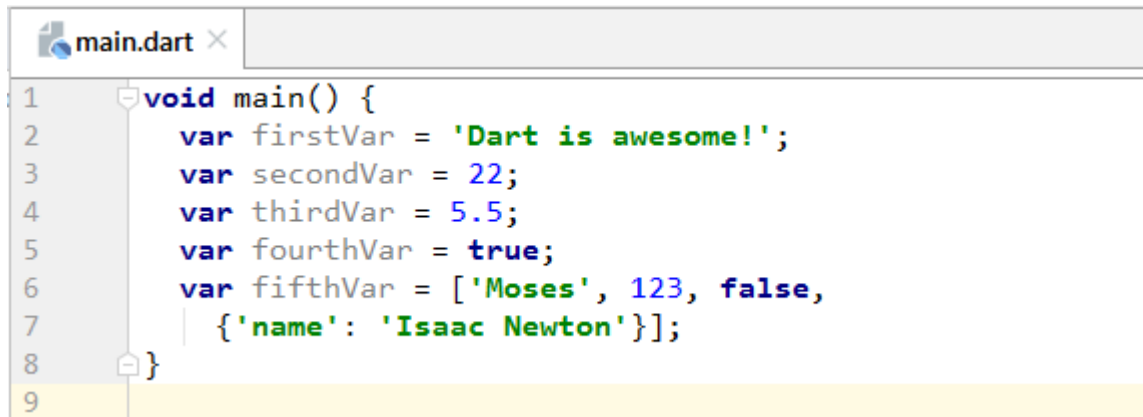
`dart main.dart`  
[Observatory: <http://127.0.0.1:63453/N78Lc2Nc9L8>]

```
Initial key-value pairs in the map.
{father: Moses John, mother: Theresa}
After new key-value pairs were added.
{father: Moses John, mother: Theresa, son: James,
 daughter: Cynthia}
```

Screenshot 2.19

As I mentioned in the beginning of this chapter, Dart is a strongly typed language, which means it allows us specify the exact type of a variable when it is defined. As a result of that, we are able to write solid code that is free of such errors as assigning a value of a different type to a variable, e.g. assigning a `String` value to an `int` variable. Dart can help us avoid such mistakes.

So far, we've seen how to declare variables using a specific type. Which would therefore mean that the variable can only be assigned a value of that type. It is possible to declare a variable that can be assigned any kind of value. To do so, we use the `var` keyword.

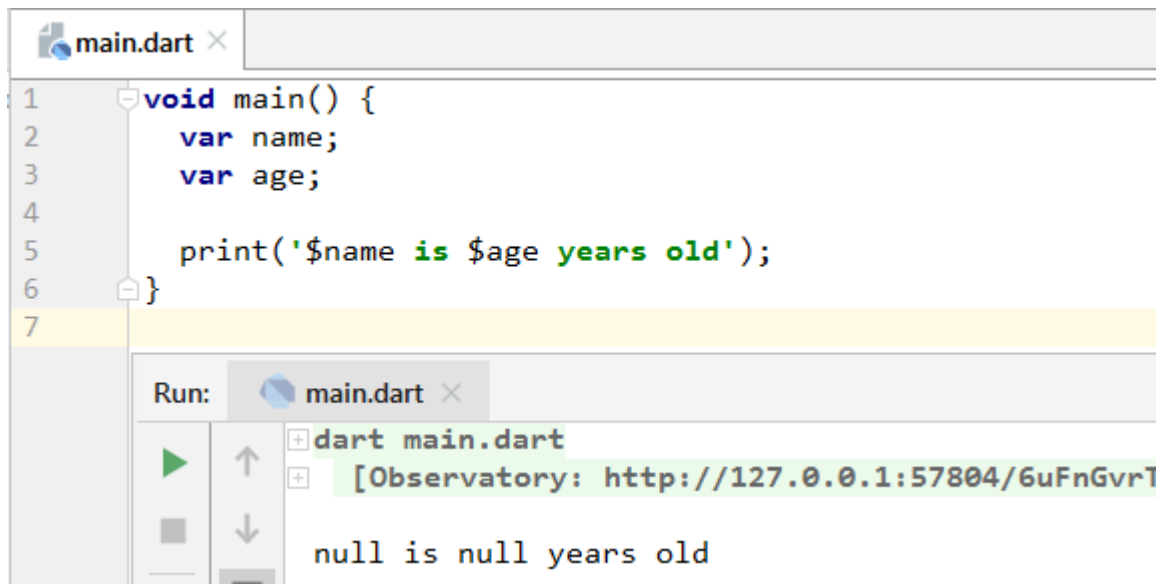


```
1 void main() {  
2   var firstVar = 'Dart is awesome!';  
3   var secondVar = 22;  
4   var thirdVar = 5.5;  
5   var fourthVar = true;  
6   var fifthVar = ['Moses', 123, false,  
7     {'name': 'Isaac Newton'}];  
8 }  
9
```

Screenshot 2.20

In screenshot 2.20, you can see how the **var** keyword is used in declaring variables that can hold any value. Note that once a value of a specific type has been assigned to a variable declared using the **var** keyword, the variable won't be able to hold any other kind of value, except values of the type that was initially assigned to it. Take the *firstVar* variable in the program above, since a String value was assigned to it, assigning a value of a different type later on won't be possible.

We've seen much on variables and the different values that can be put into them. However, one question that could come to mind is; what does a variable contain when it has not been assigned a value? Is it just empty or does it contain an initial value of some sort. In real life, a container doesn't really hold anything if nothing has been put inside of it. You can easily look into the container to know if it contains something or not. However, when a variable is defined in the Dart programming language, and hasn't been assigned a value, it holds an initial value of **null**. **null** simply means "empty" or "nothing". When you print out a variable and it shows **null**, it means that the variable has not been giving any value.

The screenshot shows an IDE window with a file named 'main.dart'. The code is as follows:

```
1 void main() {  
2     var name;  
3     var age;  
4  
5     print('$name is $age years old');  
6 }  
7
```

Line 7 is highlighted in yellow. Below the code editor is a 'Run' panel. It shows a play button, a refresh button, and a stop button. To the right of these buttons, it says 'dart main.dart' and '[Observatory: http://127.0.0.1:57804/6uFnGvr1]'. The output of the program is displayed as 'null is null years old'.

Screenshot 2.21

You can see that when the *name* and *age* variables were printed out, without assigning any value to them, they both retained their default value *null*.

## Summary

In this chapter, you learned about variables, types and values. As I mentioned earlier, these are the building blocks of any programming language. However, there are some other types in the Dart language that we didn't look at, because they're rarely used. As you progress in your Dart and Flutter journey, you may encounter these types and get to learn about them.

In order to get you started on building applications quickly, I felt it best to only introduce the more useful and important concepts.

Below are a number of questions or exercises that you should try do. They're meant to test your knowledge on some of the concepts that have been introduced to you, and to help you practice all that you've learnt in this chapter. Hence, some of them are basically about the definitions of terms, while some require you to write programs. Answers to the questions are in the questions and answers section of this book. Although I recommend that you attempt each question, before comparing your solution to the one that has been provided.

## Exercises

1. What is a variable?
2. What is a value?

3. What is a type?
4. Declare variables of the following types and assign values to them:
  - int
  - double
  - String
  - List
  - Map
5. When a variable is declared using the num keyword, what kind of value can be put in it? Write a program to support your answer.
6. When a variable is declared using the var keyword, what kind of values can be put in it? Write a program to support your answer.
7. After a variable that is declared using the var keyword is assigned a value of a particular type, can the variable be reassigned another value of a different type? Write a program to support your answer.
8. What is the difference between string concatenation and string interpolation?

## CHAPTER 3

# KEYWORDS, COMMENTS AND ERRORS

In this chapter, we shall look at the keywords in the Dart programming language, also we shall learn what comments are and how to use them. In the previous chapters, I sometimes mentioned some of the things you should not do, so as to avoid errors. This chapter covers the types of errors and how they can be managed.

### Keywords

Dart, as a programming language has some keywords which are used when writing Dart programs. When I say keywords, I mean the special words that are built into the Dart language. These words cannot be used as variable names, but are only used for their intended purpose. Below is an exhaustive list of all the keywords in the Dart language, as of version 2.4.

abstract	dynamic	implements	show
as	else	Import	static
asset	enum	In	super
async	export	interface	switch
await	extends	Is	sync
break	external	library	this
case	factory	mixin	throw
catch	false	new	true
class	final	null	try
const	finally	on	typedef
continue	for	operator	var
covariant	function	part	void
default	get	rethrow	while

deferred	hide	return	with
do	if	set	yield

As we progress in the course of the book, you shall see how these keywords can be used when writing programs. Bear in mind that you are not required to commit them all to memory. You only need remember the ones you use often.

## Comments

To explain what comments are and their importance, let me begin by first telling you a short story. There was this brilliant programmer, her name was Tracy. Tracy was writing a wonderful AI (Artificial Intelligence) program. The program was thousands of code lines long, it was going so well, and those who she told about it were really excited and were eager to see her finish. Just while it remained a little more work for Tracy to be done with her amazing AI program, she got called for some urgent business that needed her attention. So she had to stop working on her project for a while. This new endeavor took more time than she had planned. So, after about two years, she was done and was now ready to return back and finish up with her AI program.

Unfortunately, when Tracy went through the program, she couldn't really understand the structure, or the logic she had built the program with, everything was now so vague. It was as though it wasn't her program. So with that, Tracy couldn't continue with the AI program, she had to abandon it and begin work on other projects. Who knows, her AI program would have changed the world.

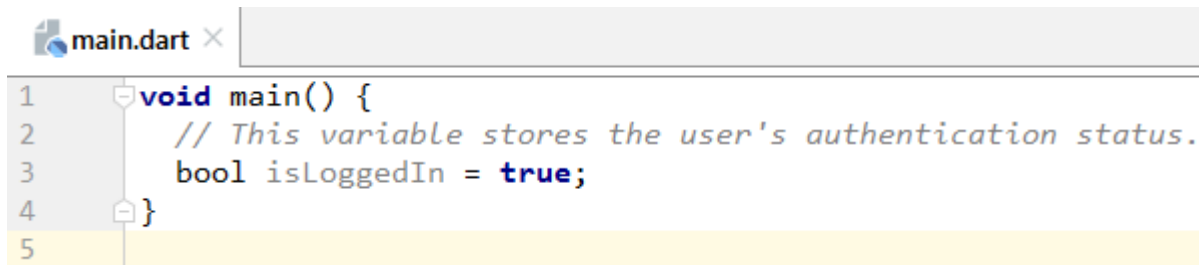
One thing that Tracy's program was lacking was good documentation, Tracy didn't add comments to her code, which would have helped her in recalling what each part of the program was doing. Now, this brings us to our topic "Comments". What are comments?

Comments are simply text descriptions that you add to your code, to explain what the code does. They're invisible to the Dart compiler - the tool that turns Dart code into machine code, to be executed by the computer. The compiler doesn't include comments when compiling programs. So comments in your source code are only meant for you the programmer and for other persons who may go through your code. When you declare a variable for example, you could add a comment, to tell what that variable is used for. There are 3 types of comments in Dart, they include:

1. Single line comment
2. Multi-line comment
3. Documentation comment

## Single Line Comment

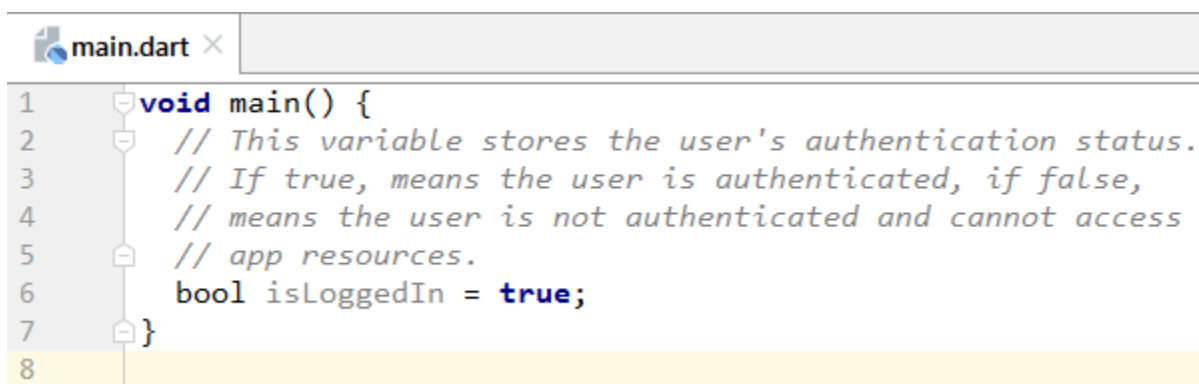
Single line comments are comments that take up only one line in your code. To write a single line comment, begin the line with double forward slashes, then followed by the text.



```
1 void main() {
2     // This variable stores the user's authentication status.
3     bool isLoggedIn = true;
4 }
5
```

Screenshot 3.1

The comment in the program above is referred to as a single line comment because of the double forward slashes that are placed before the text. It is possible to have more than one single line comment for a piece of code. To do that, simply precede the other lines with the same double forward slashes.



```
1 void main() {
2     // This variable stores the user's authentication status.
3     // If true, means the user is authenticated, if false,
4     // means the user is not authenticated and cannot access
5     // app resources.
6     bool isLoggedIn = true;
7 }
8
```

Screenshot 3.2

## Multiline Comment

A multiline comment is created by wrapping the text with `(/* */)`, is a way to create comments that span several lines, it is used when there's more to be said about some code that would otherwise not fit into a single line.



```

1 void main() {
2     /*
3     This variable stores the user's authentication status.
4     If true, means the user is authenticated, if false,
5     means the user is not authenticated and cannot access
6     app resources.
7     */
8     bool isLoggedIn = true;
9 }
10

```

Screenshot 3.4

## Documentation comment

Documentation comment is a special kind of comment. Dart treats documentation comments differently than other comments. When you add documentation comments to your code, within the comment, you can include the names of variables and other program features like functions and parameters. Documentation comments can be written using triple forward slashes and then the text.

```

1 void main() {
2     /// [isLoggedIn] stores the user's authentication status.
3     /// If true, means the user is authenticated, if false,
4     /// means the user is not authenticated and cannot access
5     /// app resources
6     bool isLoggedIn = true;
7 }
8

```

Screenshot 3.5

Observe how emphasis is placed on the *isLoggedIn* variable by wrapping it in square brackets.

## Errors

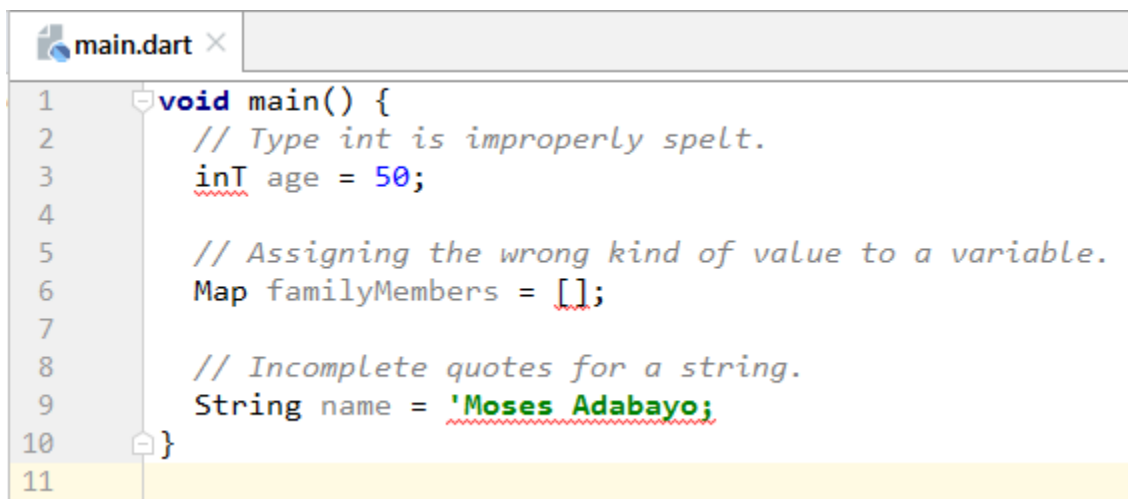
Errors refer to mistakes that are made when writing programs. There are three (3) major kinds of errors.

1. Syntax (compilation error)

2. Logical error
3. Runtime (execution error)

## Syntax Error (Compilation Error)

We defined syntax earlier as what constitutes a program statement. To have a valid program statement, one must adhere to the syntax rules for creating that statement. Going against such a rule would result in a syntax error. Examples of syntax errors include misspelling the names of Dart's keywords or type names like `int`, `double`, `String`, `Map`, etc. They're also called compilation errors, because they prevent your code from compiling at all.



Screenshot 3.6

The editor always draws your attention to syntax errors by underlining the particular piece of code with a red squiggly line. When you hover over the red line with the mouse pointer, Dart tells you what is probably causing the error, which could be really helpful in resolving the error. As an exercise, try resolve the syntax errors in the program above.

## Logical Error

Logical errors are the kind of errors that arise from improperly formed program logic. A logical error can be made when doing comparisons of values, using any of the comparison operators. We would see examples of some logical errors when we've learnt about how values can be compared using the comparison operators in chapter 4. But before that, let me paint a scenario for you.

Imagine you were to build a mobile application that would be used to collect the details of those attending a party. This party is open to only those whose age is greater

than 18. So in the application, you could write some code that checks the age of each attendee, when it's entered into the application. So such a code would check if the attendees' age is greater than 18, not equal to 18 or less than 18. A possible error would be writing the code, to check if the age of an attendee is greater than or equal to 18, meaning that all those who are currently 18 years of age and older can attend the party. Which is not what the application should do. This is one simple example of a logical error.

Logical errors are the most difficult to catch and fix, because they do not prevent your code from executing, instead your code executes, but doesn't do what you want it to do.

### **Runtime Error (Execution Error)**

Runtime errors are referred to as execution errors, because they are errors that occur at runtime – while the program is executing. A good example of what could cause a runtime error is making use of a variable that has not been assigned a value. Although you haven't yet seen how to perform arithmetic operations in Dart, the program below makes use of the addition operator. It attempts to add the values of two integer variables together, while one was assigned a value, the other wasn't.

The screenshot shows an IDE window with a file named `main.dart`. The code is as follows:

```
1 void main() {  
2     int a = 10;  
3     int b;  
4     print(a + b);  
5 }  
6
```

Below the code editor is a 'Run' console window. It shows the output of the program, which is an unhandled exception:

```
Run: main.dart  
+ dart main.dart  
+ [Observatory: http://127.0.0.1:50260/3V-bIh1Jbd:  
Unhandled exception:  
NoSuchMethodError: The method  
  '_addFromInteger' was called on null.  
Receiver: null  
Tried calling: _addFromInteger(10)  
#0      Object.noSuchMethod  
         (dart:core-patch/object_patch.dart:50:5)  
#1      int.+ (dart:core-patch/integers  
           .dart:9:38)  
#2      main (file:///C:/Users/Precious  
           %20Gabriel/IdeaProjects/Flutter-Book/lib/main  
           .dart:4:11)  
#3      _startIsolate.<anonymous closure>  
         (dart:isolate-patch/isolate_patch.dart:301:19)  
#4      _RawReceivePortImpl._handleMessage  
         (dart:isolate-patch/isolate_patch.dart:172:12)
```

Screenshot 3.7

In the program above, the variable `a`, was assigned the value 10, while the variable `b`, wasn't assigned any value. The program compiles successfully, but fails to execute, because Dart cannot do the addition operation, due to the fact that the variable `b` contains no number. This results in a runtime error, causing the program to crash and stop executing. Dart however tries to help us resolve the error with the message which is displayed in red color in the console view. The message tries to describe the error, and most times, provides a link to the part of the code where the error occurred, and even the statement line. In this case, it was line 4, position 11.

## Summary

In this chapter, you learnt about the usefulness of comments, the keywords that are used in the Dart language, what errors are and how to remedy them when they

accidentally manifest in your program. As I mentioned earlier, it is a very good practice to include comments in your code, in a very small program, it may not be so important, but as your program or application code gets larger, it becomes necessary to add comments to the part of your code where the meaning is not so obvious.

As your program gets larger, it becomes almost impossible to avoid errors, especially logical and runtime errors. Dart always tries to help you by showing the cause of the error in the console view, often times it tells you of the line number and it even provides a link to the exact part of your code where the error originated from.

Endeavour to read the error message diligently, so as to gain a good insight on what might be causing the error, and how to fix it.

## CHAPTER 4

# OPERATORS

Dart has three major set of operators which can be used for arithmetic, relational and logical operations. Some of these operators share same symbols with those used in the real world, like the arithmetic operators, etc. while others don't. In this chapter, we shall explore how these operators are used in the Dart programming language.

### Types of Operators

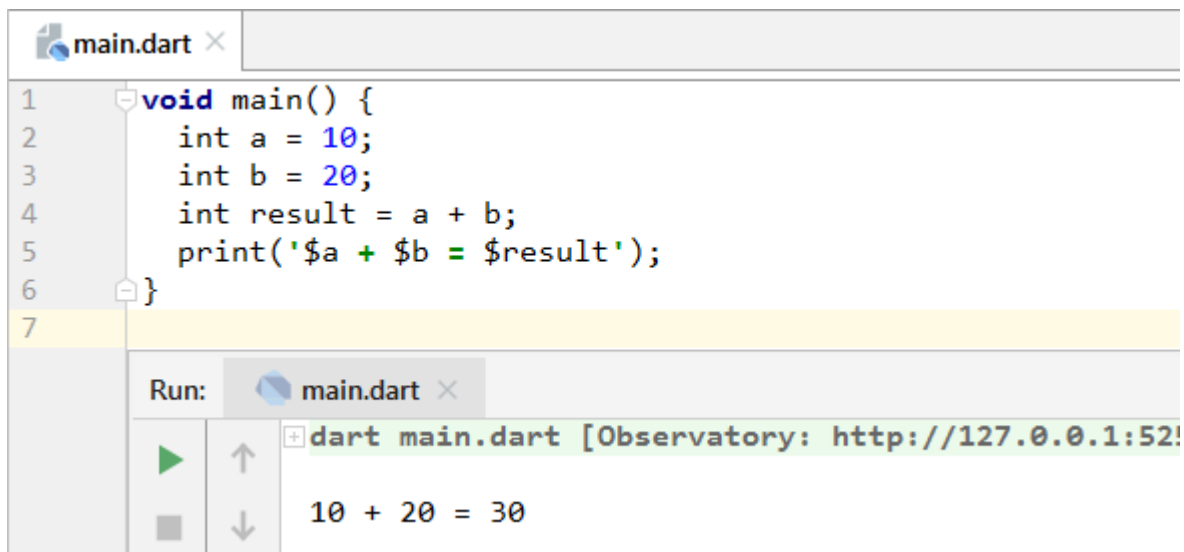
1. Arithmetic Operators
2. Relational Operators
3. Logical Operators

### Arithmetic Operators

Arithmetic operators include the addition (+), subtraction (-), multiplication (\*), division (/), integer division (~/) and modulus (%). These are the more commonly used arithmetic operators in Dart.

### Addition Operator (+)

The addition operator can be used in performing addition operations, just as is done in arithmetic. It's a binary operator, because it requires two operands. It adds the operand at its right to the operand at its left.



The screenshot shows an IDE window titled 'main.dart'. The code is as follows:

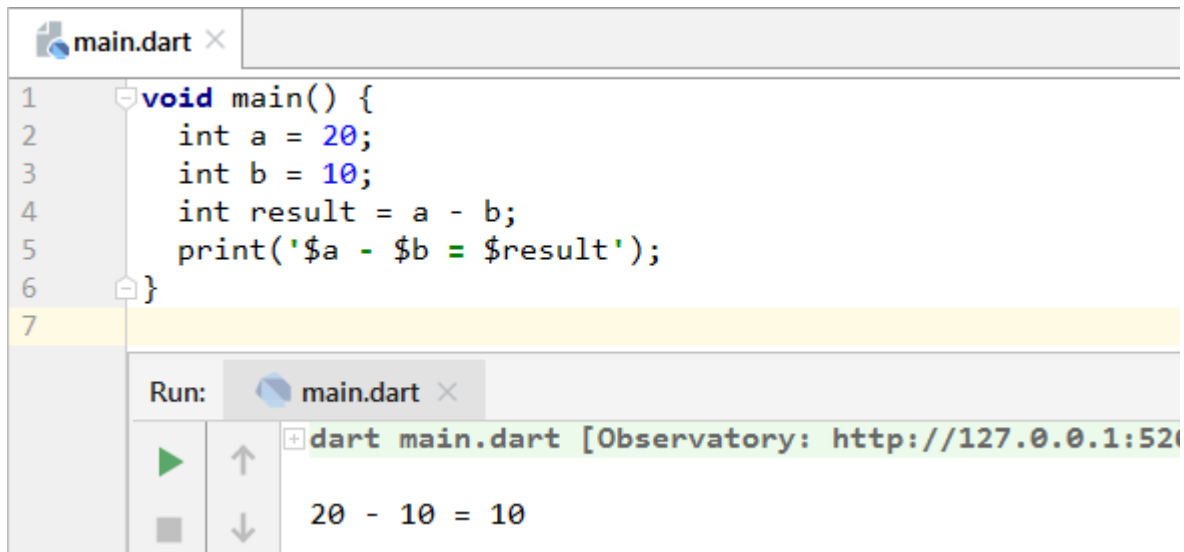
```
1 void main() {  
2     int a = 10;  
3     int b = 20;  
4     int result = a + b;  
5     print('$a + $b = $result');  
6 }  
7
```

Below the code editor, there is a 'Run' button and a console output area. The console shows the command 'dart main.dart [Observatory: http://127.0.0.1:52...]' and the output '10 + 20 = 30'.

Screenshot 4.1

## Subtraction operator (-)

The subtraction operator can be used in obtaining the difference between two numbers. It's also a binary operator, because it requires two operands. It subtracts the operand at its right from the operand at its left.



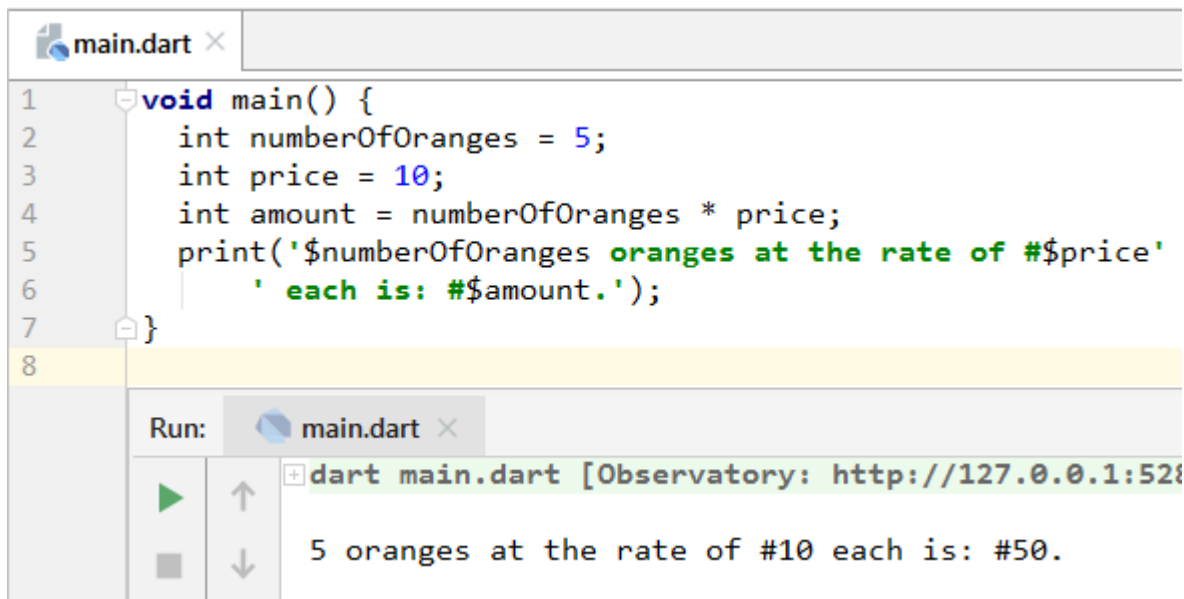
```
main.dart x
1 void main() {
2   int a = 20;
3   int b = 10;
4   int result = a - b;
5   print('$a - $b = $result');
6 }
7

Run: main.dart x
+ dart main.dart [Observatory: http://127.0.0.1:5200]
20 - 10 = 10
```

Screenshot 4.3

## Multiplication Operator (\*)

The multiplication operator in Dart is represented using asterisk (\*). It multiplies two numbers and yields their product.



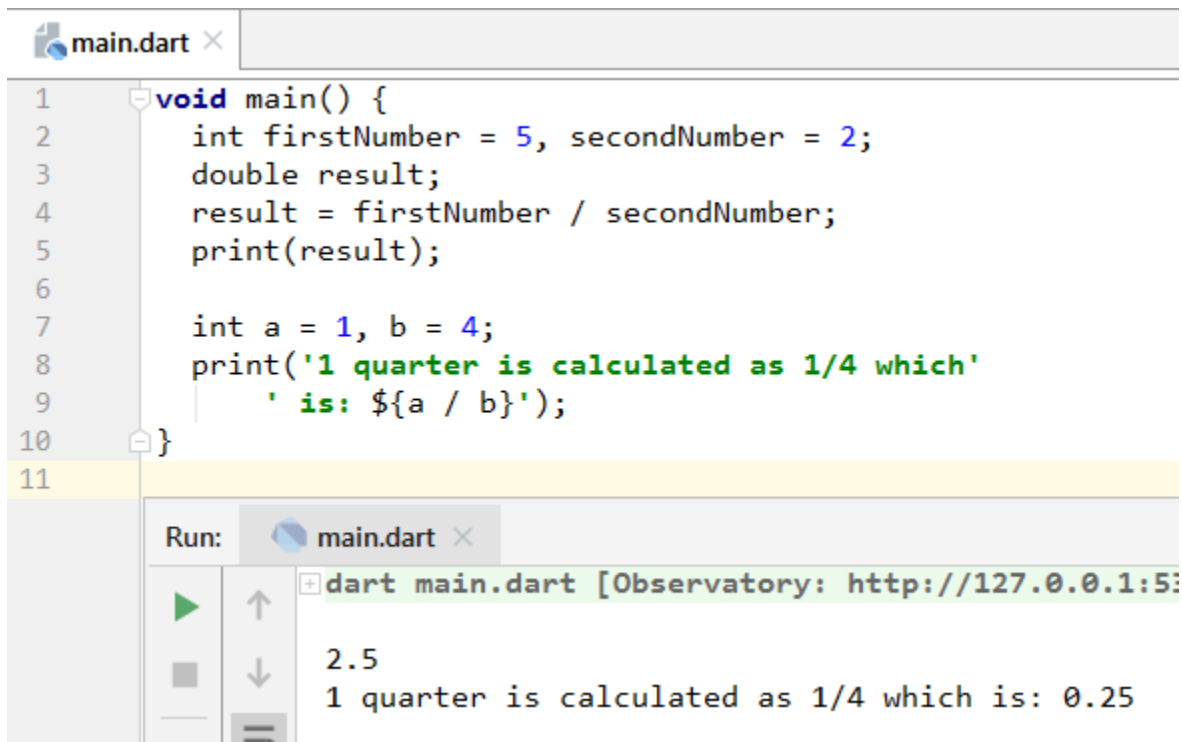
```
main.dart x
1 void main() {
2   int numberOfOranges = 5;
3   int price = 10;
4   int amount = numberOfOranges * price;
5   print('$numberOfOranges oranges at the rate of #${price}
6         ' each is: #${amount}.');
7 }
8

Run: main.dart x
+ dart main.dart [Observatory: http://127.0.0.1:5200]
5 oranges at the rate of #10 each is: #50.
```

Screenshot 4.4

## Division Operator (/)

The division operator in Dart is represented using forward slash (/). It divides two numbers and yields their quotient. Please pay close attention to the program shown below, it introduces a new way variables can be declared, it also uses string interpolation, which you learnt about earlier.



```
main.dart x
1 void main() {
2   int firstNumber = 5, secondNumber = 2;
3   double result;
4   result = firstNumber / secondNumber;
5   print(result);
6
7   int a = 1, b = 4;
8   print('1 quarter is calculated as 1/4 which'
9         ' is: ${a / b}');
10 }
11

Run: main.dart x
+ dart main.dart [Observatory: http://127.0.0.1:53
2.5
1 quarter is calculated as 1/4 which is: 0.25
```

Screenshot 4.5

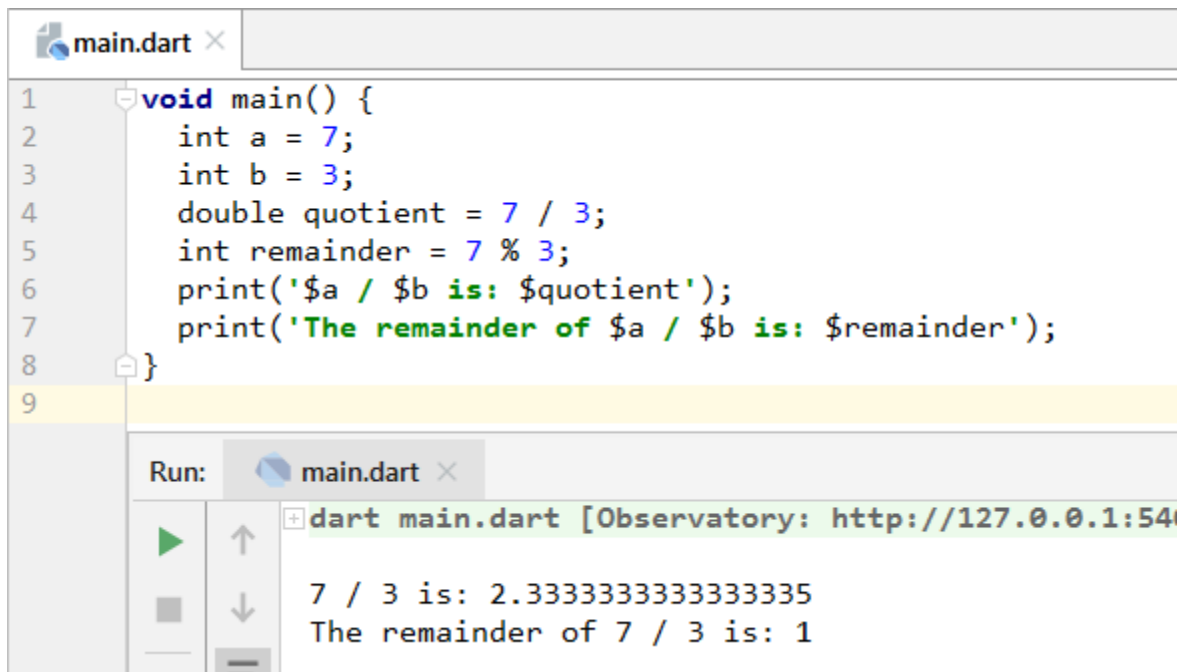
Observe how the *firstNumber* and *secondNumber* variables were declared on line 2. They were both declared on the same line to be of type *int*. This style of declaration is useful when you want to declare multiple variables that are of the same type.

One thing to note about division operations in Dart, is that a division operation always results in a decimal value. Which is why the *result* variable which stores the result from dividing *firstNumber* with *secondNumber* was declared to be of type *double*.

## Modulus Operator (%)

The modulus operator, also called the remainder operator, because it divides two numbers but doesn't return their quotient, instead it returns the remainder of the division, which is always an integer value.





The screenshot shows an IDE window titled 'main.dart'. The code is as follows:

```
1 void main() {  
2     int a = 7;  
3     int b = 3;  
4     double quotient = 7 / 3;  
5     int remainder = 7 % 3;  
6     print('$a / $b is: $quotient');  
7     print('The remainder of $a / $b is: $remainder');  
8 }  
9
```

Below the code editor is a 'Run' panel. It shows the command 'dart main.dart' and the output:

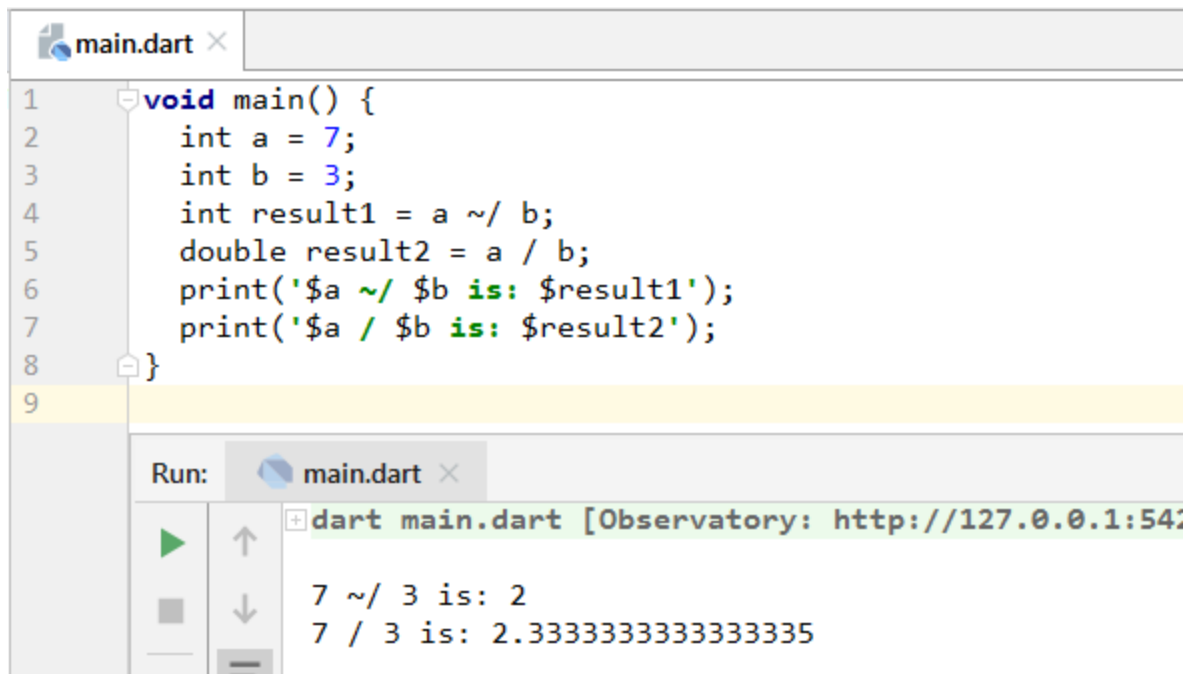
```
dart main.dart [Observatory: http://127.0.0.1:54...]  
  
7 / 3 is: 2.3333333333333335  
The remainder of 7 / 3 is: 1
```

Screenshot 4.6

The remainder from a division is always a whole number, which is why *remainder* had to be declared as an *int* variable, since it would hold an integer value.

## Integer Division (~/)

This is an operator that is specific to the Dart language, you may not find it as one of the usual arithmetic operators. What it does is to divide two numbers and return only the integer part of the result. It truncates i.e. removes the fractional part.



The screenshot shows an IDE window titled 'main.dart'. The code is as follows:

```
1 void main() {  
2     int a = 7;  
3     int b = 3;  
4     int result1 = a ~/ b;  
5     double result2 = a / b;  
6     print('$a ~/ $b is: $result1');  
7     print('$a / $b is: $result2');  
8 }  
9
```

Below the code editor is a 'Run' panel. It shows the command 'dart main.dart' and the Observatory URL 'http://127.0.0.1:5426'. The output of the program is displayed below the command:

```
7 ~/ 3 is: 2  
7 / 3 is: 2.3333333333333335
```

Screenshot 4.7

Observe the difference between the integer division operator and the normal division operator. The integer division returns a quotient with the decimal part (if any) removed, while the normal division operator retains the decimal part.

## More Complex Operations

Arithmetic operations do not always involve only one operator and two operands. Sometimes, they are usually more complex, containing multiple operands and operators and even parenthesis (which are used to remove ambiguity). Let's look at a program that deals with a little more complex arithmetic expression.

```
main.dart x
1 void main() {
2   int a = 2, b = 3, c = 10, d = 5;
3   double result = a + b * c / d;
4
5   print('$a + $b * $c / $d = $result');
6 }
7

Run: main.dart x
+ dart main.dart [Observatory: http://127.0.0.1:5441]
2 + 3 * 10 / 5 = 8.0
```

Screenshot 4.8

When Dart encounters such a complex expression as the one in the program above, it tries to resolve it, based on operator precedence. In which case the multiplication and division operators have a higher precedence than the addition and subtraction operators. So starting from the right, it evaluates the expression and does multiplication or division first, depends on which it encounters first. When all the multiplication and division operations are attended to, before addition and subtraction. This rule is required to remove ambiguity in arithmetic expressions. However, brackets can be used to increase the precedence of any of the operators, as shown below.

```
main.dart x
1 void main() {
2   int a = 2, b = 3, c = 10, d = 5;
3   double result = (a + b) * c / d;
4
5   print('($a + $b) * $c / $d = $result');
6 }
7

Run: main.dart x
+ dart main.dart [Observatory: http://127.0.0.1:5441]
(2 + 3) * 10 / 5 = 10.0
```

Screenshot 4.9

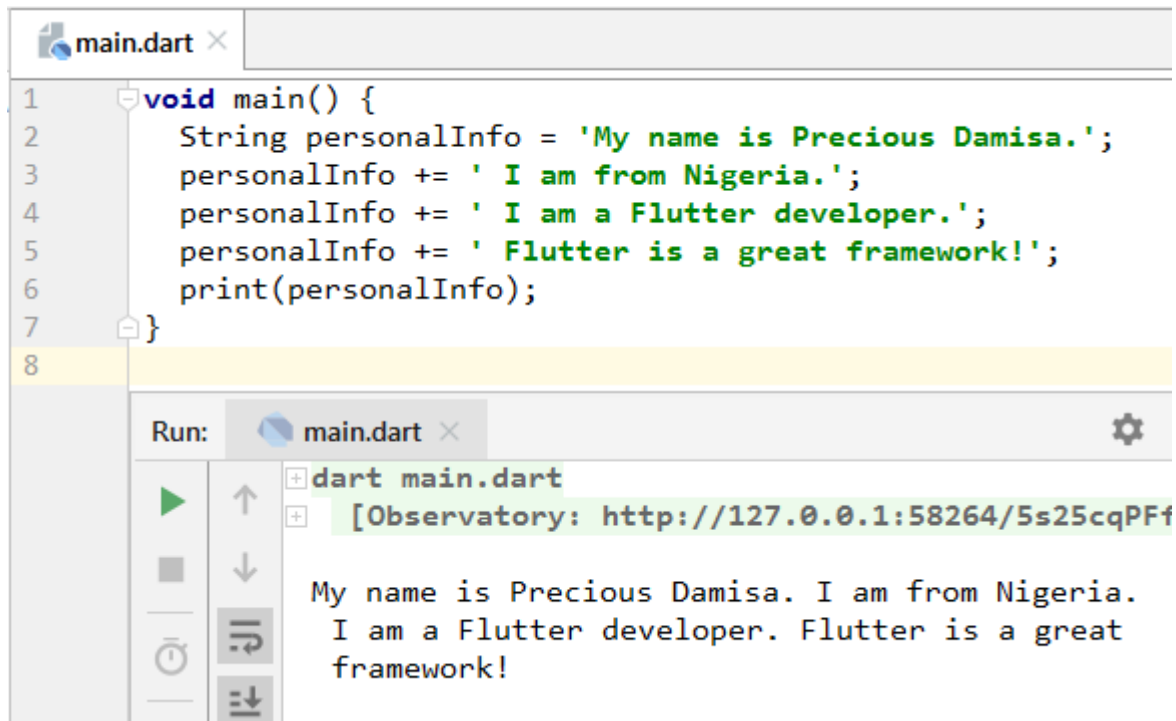
The result of the previous expression was 8.0, but when the addition operation is performed first, through the use of brackets, we get a different result of 10.0.

## Arithmetic Assignment Operators

The arithmetic operators we just looked at, can be combined with the assignment operator (=) to perform an arithmetic operation and an assignment at the same time. Here's a list of the more commonly used arithmetic assignment operators. Assuming the variable **a** holds the value 2.

Operator	Operation	Equivalent Operation	Result
<b>+=</b> (Addition Assignment)	a += 2	a = a + 2	a becomes 4
<b>-=</b> (Subtraction Assignment)	a -= 2	a = a - 2	a becomes 0
<b>*=</b> (Multiplication Assignment)	a *= 2	a = a * 2	a becomes 4
<b>/=</b> (Division Assignment)	a /= 2	a = a / 2	a becomes 1.0
<b>%=</b> (Modulus Assignment)	a %= 2	a = a % 2	a becomes 0
<b>~/=</b> (Integer Division Assignment)	a ~/= 2	a = a ~/ 2	a becomes 1

Of more interest is the **addition assignment** operator. Remember that the **plus (+)** operator is also used for achieving **string concatenation**. The addition assignment operator can as well be used in manipulating strings. It can be used to build a long string incrementally. Take a look at the example below.



The screenshot shows an IDE window with a file named `main.dart`. The code defines a `main` function that initializes a `String` variable `personalInfo` and concatenates several strings to it using the `+=` operator. The code is as follows:

```
1 void main() {  
2   String personalInfo = 'My name is Precious Damisa.';  
3   personalInfo += ' I am from Nigeria.';  
4   personalInfo += ' I am a Flutter developer.';  
5   personalInfo += ' Flutter is a great framework!';  
6   print(personalInfo);  
7 }  
8
```

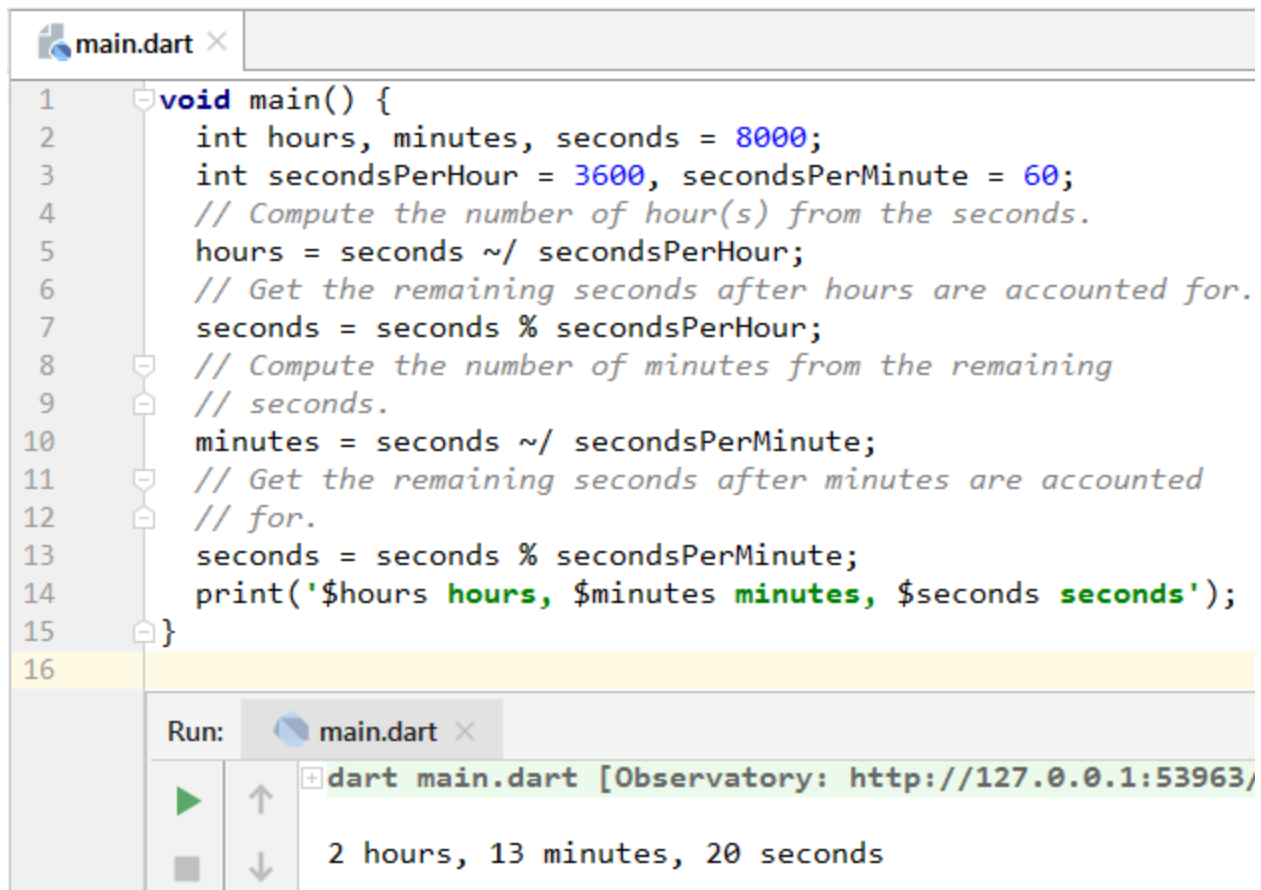
Below the code editor, the 'Run' button is visible. The output console shows the following:

```
Run: main.dart  
+ dart main.dart  
+ [Observatory: http://127.0.0.1:58264/5s25cqPFf  
My name is Precious Damisa. I am from Nigeria.  
I am a Flutter developer. Flutter is a great  
framework!
```

Screenshot 4.10

When the addition assignment operator is applied to a `String` value, it performs string concatenation and not addition. The new `String` is added to the old `String` and a new `String` value is created from it and assigned to the `String` variable, in this case *personalInfo*.

Before taking a look at the **Relational Operators**, let's write a fun program that does time conversion. With your knowledge of arithmetic operations, the program should be easy to comprehend.



```
1 void main() {
2   int hours, minutes, seconds = 8000;
3   int secondsPerHour = 3600, secondsPerMinute = 60;
4   // Compute the number of hour(s) from the seconds.
5   hours = seconds ~/ secondsPerHour;
6   // Get the remaining seconds after hours are accounted for.
7   seconds = seconds % secondsPerHour;
8   // Compute the number of minutes from the remaining
9   // seconds.
10  minutes = seconds ~/ secondsPerMinute;
11  // Get the remaining seconds after minutes are accounted
12  // for.
13  seconds = seconds % secondsPerMinute;
14  print('$hours hours, $minutes minutes, $seconds seconds');
15 }
16
```

Run: main.dart x

dart main.dart [Observatory: <http://127.0.0.1:53963/>]

2 hours, 13 minutes, 20 seconds

Screenshot 4.11

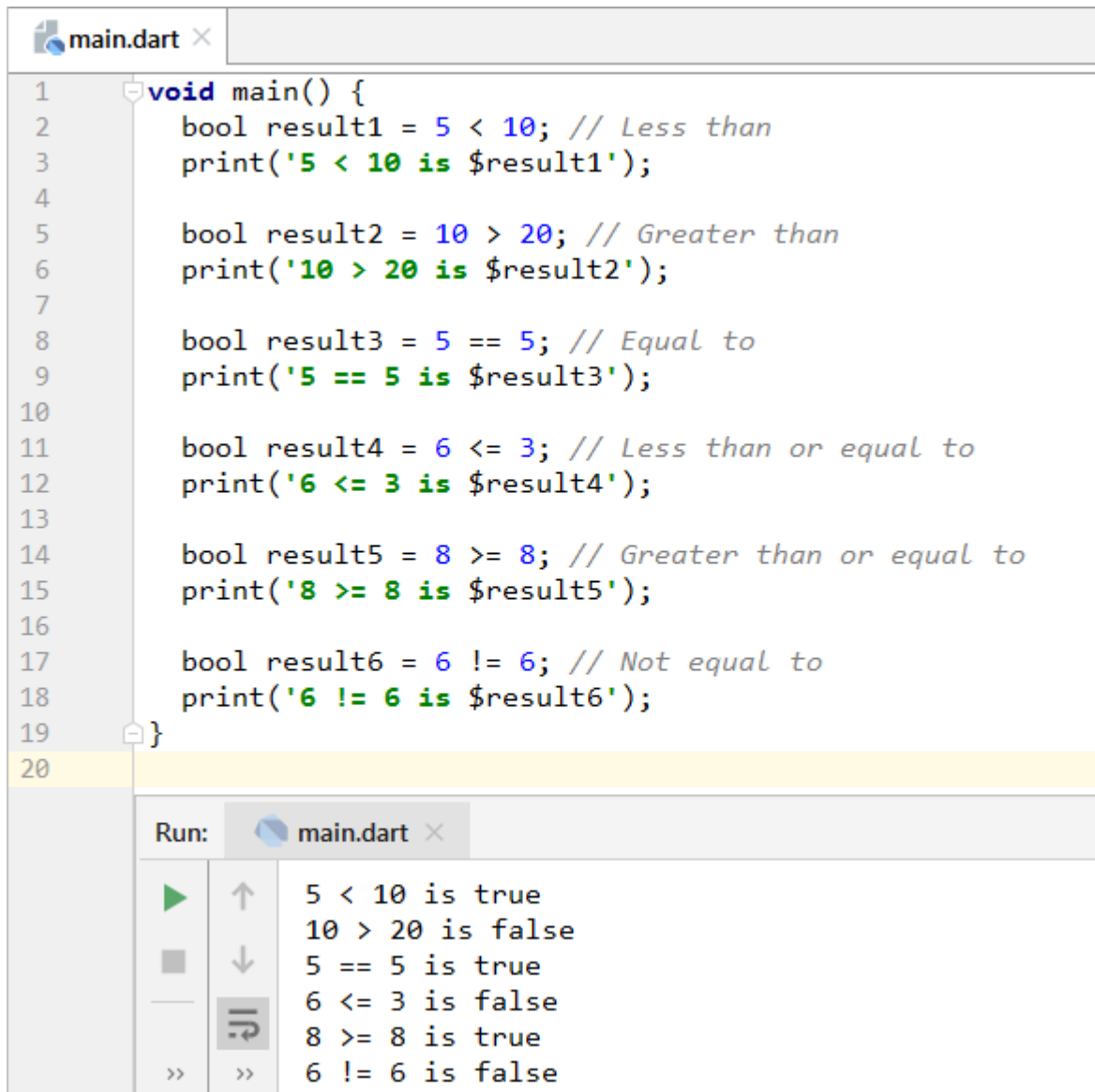
8000 seconds equal 2 hours, 13 minutes, and 20 seconds. Feel free to test the program with other seconds' values.

## Relational Operators

Relational operators are used for comparison. A value is either greater than, less than, equal to, or not equal to other another value, etc. Dart has some special operators which it uses to perform these comparisons and they're collectively referred to as the relational operators. The relational operators are represented using the following symbols:

1. Less than (<)
2. Greater than (>)
3. Equal to (==)
4. Less than or equal to (<=)
5. Greater than or equal to (>=)
6. Not equal to (!=)

The **equal to** operator is represented using two equality signs, not to be confused with the assignment operator (=) which is the single equality sign. One thing to note about expressions involving relational operators is that they result in the Boolean values of **true** or **false**. Let's write a program that shows how these operators can be used.



```
1 void main() {
2   bool result1 = 5 < 10; // Less than
3   print('5 < 10 is $result1');
4
5   bool result2 = 10 > 20; // Greater than
6   print('10 > 20 is $result2');
7
8   bool result3 = 5 == 5; // Equal to
9   print('5 == 5 is $result3');
10
11  bool result4 = 6 <= 3; // Less than or equal to
12  print('6 <= 3 is $result4');
13
14  bool result5 = 8 >= 8; // Greater than or equal to
15  print('8 >= 8 is $result5');
16
17  bool result6 = 6 != 6; // Not equal to
18  print('6 != 6 is $result6');
19 }
20
```

Run: main.dart

```
5 < 10 is true
10 > 20 is false
5 == 5 is true
6 <= 3 is false
8 >= 8 is true
6 != 6 is false
```

Screenshot 4.12

In the above program, we see how the result of a relational expression ends up being either true or false. Here, we're simply printing out the result, which doesn't really show their usefulness. Relational operators are better used in constructing tests or conditions in **control flow statements**, which we shall look at in chapter 6.

## Logical Operators

The logical operators are used in conjunction with the relational operators to develop more complex expressions, they also result in either true/false. They include:

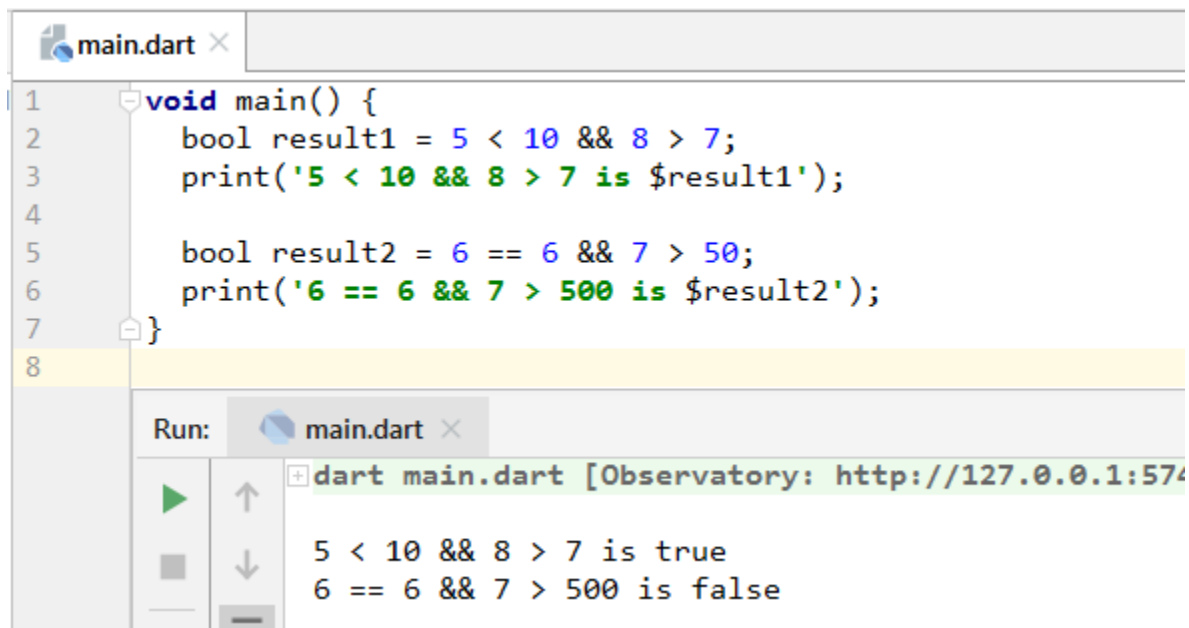
1. AND (&&)
2. OR (||)
3. NOT (!)

To better understand how these operators work, let's view them on a table.

### AND (&&) Operator

Expression	Operator	Expression	Result
True	&&	true	true
True	&&	false	false
False	&&	true	false
False	&&	false	false

For an expression involving the AND (&&) operator to be true, the expressions at the right and at the left must be true. For any other case, it evaluates to false. Let's write a program that uses the && operator.



```
main.dart x
1 void main() {
2   bool result1 = 5 < 10 && 8 > 7;
3   print('5 < 10 && 8 > 7 is $result1');
4
5   bool result2 = 6 == 6 && 7 > 50;
6   print('6 == 6 && 7 > 500 is $result2');
7 }
8

Run: main.dart x
+ dart main.dart [Observatory: http://127.0.0.1:574
5 < 10 && 8 > 7 is true
6 == 6 && 7 > 500 is false
```

Screenshot 4.13

Endeavour to try out the other cases for the && operator.

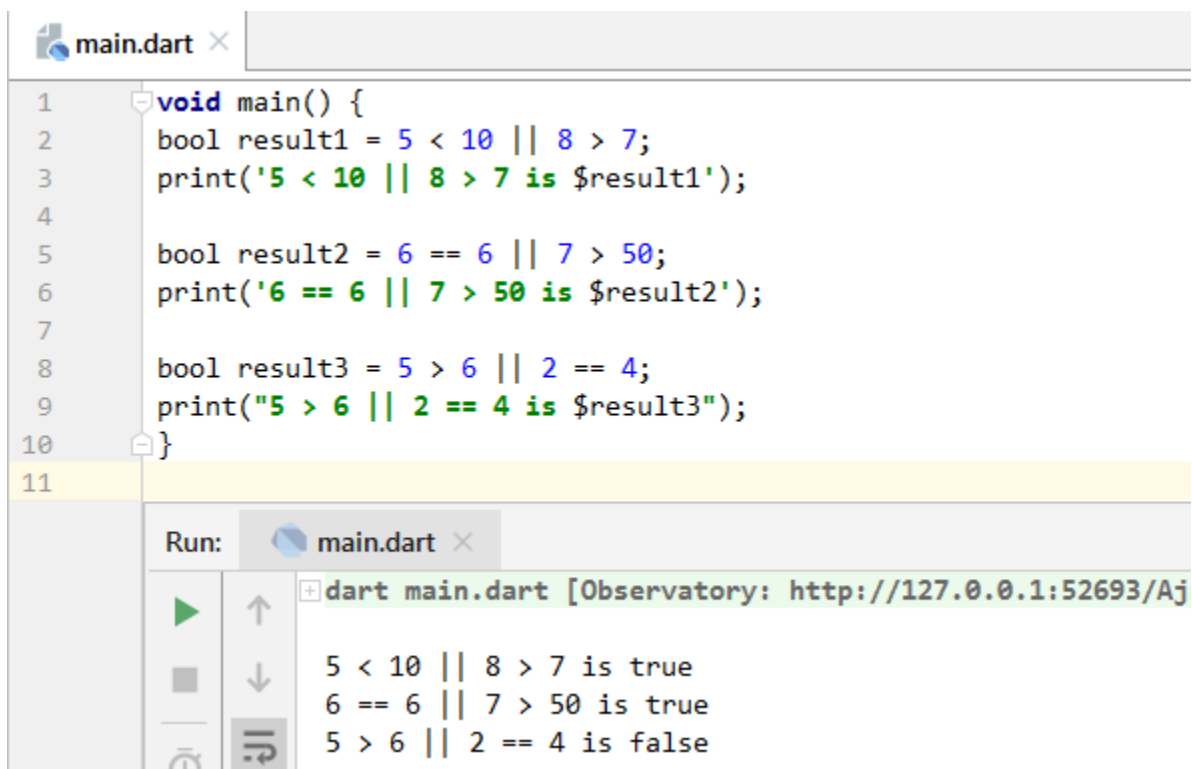


## OR (||) Operator

The OR operator is represented using the pipe (|) sign on the keyboard. The one on the same key with the backward slash (\).

Expression	Operator	Expression	Result
true		true	true
true		false	true
false		true	true
false		false	false

An expression involving the OR (||) operator is only false, when the expressions at the right and the left are both false. In any other case, it evaluates to true.



The screenshot shows an IDE window titled 'main.dart' with the following code:

```
1 void main() {  
2   bool result1 = 5 < 10 || 8 > 7;  
3   print('5 < 10 || 8 > 7 is $result1');  
4  
5   bool result2 = 6 == 6 || 7 > 50;  
6   print('6 == 6 || 7 > 50 is $result2');  
7  
8   bool result3 = 5 > 6 || 2 == 4;  
9   print("5 > 6 || 2 == 4 is $result3");  
10 }  
11
```

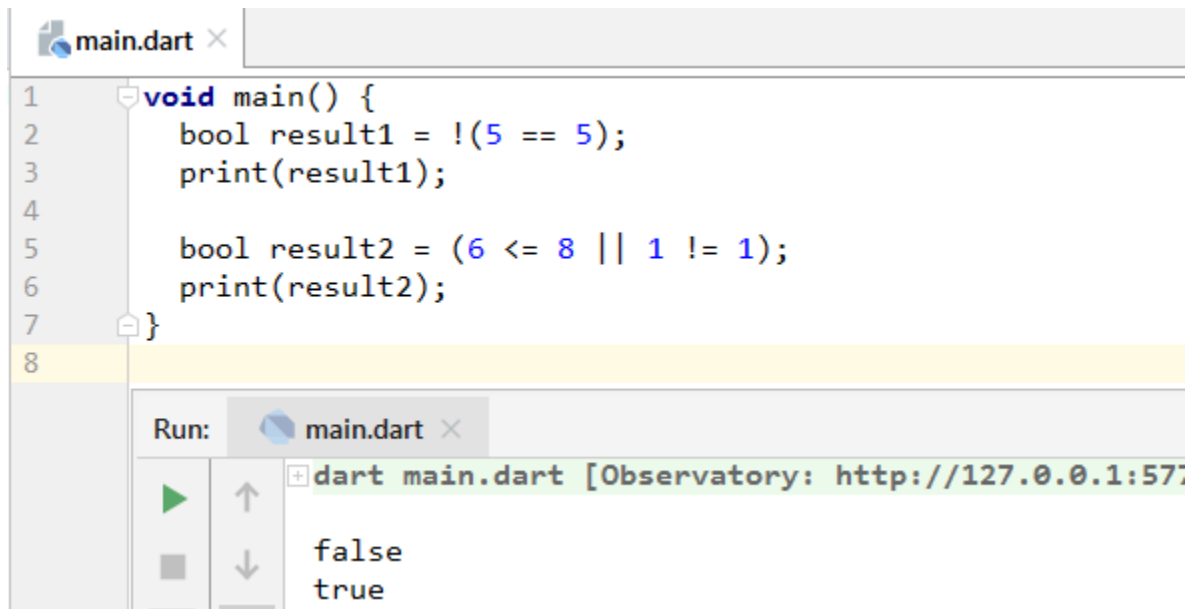
Below the code editor, the 'Run' button is clicked, and the output is displayed in a console window titled 'dart main.dart [Observatory: http://127.0.0.1:52693/Aj]':

```
5 < 10 || 8 > 7 is true  
6 == 6 || 7 > 50 is true  
5 > 6 || 2 == 4 is false
```

Screenshot 4.14

## NOT (!) Operator

The NOT (!) operator works in a rather odd way, it negates the result of a relational or logical expression, just as shown in the example below.



```
1 void main() {  
2   bool result1 = !(5 == 5);  
3   print(result1);  
4  
5   bool result2 = (6 <= 8 || 1 != 1);  
6   print(result2);  
7 }  
8
```

Run: main.dart x

dart main.dart [Observatory: http://127.0.0.1:5775]

false  
true

Screenshot 4.15

On a normal day, testing if 5 is equal to 5 ( $5 == 5$ ) would result to *true*, same with how testing  $(6 \leq 8 \parallel 1 \neq 1)$  would result to *false*, but with the magic of the NOT (!) operator, we're able to change the results of such expressions.

## Summary

In this chapter, you learnt about the different operators used in the Dart language. It would be impossible to write sophisticated programs or programs that do amazing things without the use of operators. They provide a way of adding logic and complexity to programs. Endeavour to learn and master the symbols for representing each. Practice by forming simple expressions, then proceed to more complex expressions.

## Exercises

1. Which symbol best describes the greater than or equal-to operator?
  - a.  $\rightarrow$
  - b. `greaterThanOrEqualTo`
  - c.  $\Rightarrow$
  - d.  $\geq$
2. Which symbol best describes the not equal-to operator?
  - a.  $\sim =$
  - b.  $\wedge =$
  - c.  $\# =$

d. !=

1. Which symbol best describes the less than or equal-to operator?

a. <-

b. lessThanOrEqualTo

c. =<

d. <=

2. Evaluate each of the following logical expressions

a.  $5 == 6 \parallel 6 == 5$

b.  $7 < 8 \parallel 4 != 4$

c.  $0 <= 1 \parallel 10 > 20$

d.  $7 > 1 \&\& 8 > 2$

e.  $!(9 > 9 \&\& 10 != 10)$

# CHAPTER 5

## FUNCTIONS

In this chapter, we shall explore all about functions, what they are, the types of functions, and how they can be used.

### What is a Function?

A function is a group or set of program statements that can be executed as a whole, to achieve a particular result. The major importance of functions is that they help us write reusable code and they provide structure for our programs. Let's take a look at a simple analogy. When you want to make a cup of tea, you're required to go through some certain steps:

**Step 1:**

Using a kettle, put water on the fire.

**Step 2:**

Add some quantity of Milo and Milk into a tea cup.

**Step 3:**

When the water is warm, pour some of it into the tea cup.

**Step 4:**

Stir for a little while.

With that, you have a cup of tea ready for your enjoyment. If for example you want to take tea every morning, throughout the week, you would have to go through these steps daily.

In programming, these different steps map to program statements. You could write a function that contains all the statements required to do a particular thing, and whenever you need to do that thing, all you have to do is call the function using its name.

### Types of Functions

There are basically two types of functions in Dart.

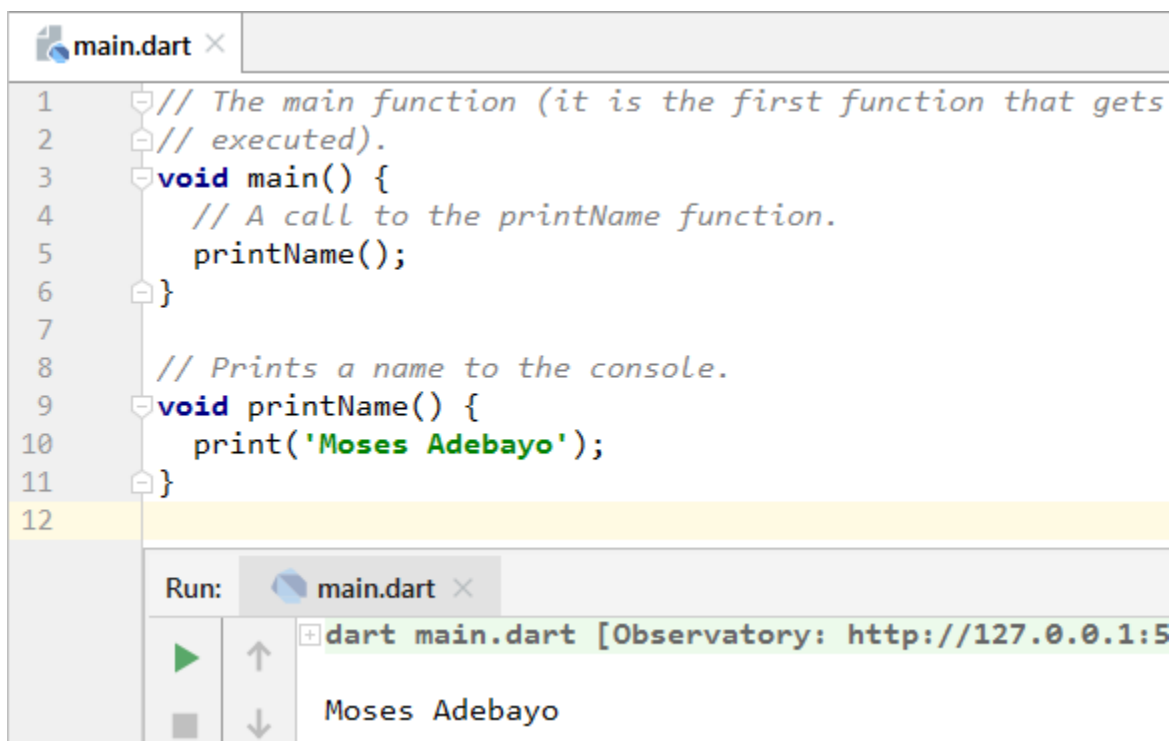
1. Named functions and
2. Anonymous functions (Nameless functions)

### Named Functions

Named functions are functions that have a name and can be called with that name. Using the analogy I cited earlier, if you write a function that makes a cup of tea, you

could call the function *makeTea*. Then whenever you want to make a cup of tea, all you have to do is call *makeTea*. Observe how *makeTea* is spelt, it is an identifier (a word I came up with), so it follows the same principles as naming identifiers, i.e. using lower camel case. With all that explained, let's see our first example on how a named function is created. Although, just before that, you should know that right from the beginning, we've been making use of a named function, i.e. the *main* function, which has been present in all the programs we've written thus far. The *main* function should always be present in every program you write, because it is the starting point for all programs, it is the first function that gets executed when a program runs, hence it always has to be present.

Think of it this way, imagine we defined so many different functions in a program, which of those functions should get executed first? The only way to execute all the functions, is to give priority to one, as being the first to execute, so that from within the first function, different calls can be made to the other functions so that they can be execute.



```
1 // The main function (it is the first function that gets
2 // executed).
3 void main() {
4     // A call to the printName function.
5     printName();
6 }
7
8 // Prints a name to the console.
9 void printName() {
10     print('Moses Adebayo');
11 }
12
```

Run: main.dart x

dart main.dart [Observatory: http://127.0.0.1:5

Moses Adebayo

Screenshot 5.1

In the program above, there are two functions. One is the *main* function, while the other is the *printName* function. As you can see, the two functions have a similar structure, so let's expose their different parts. We will use the *printName* function for our explanation. Starting from the left, we have the keyword **void**, this is the return type of the function (more on this later). Next, we have the name of the function, in this case, the function is

called *printName*. Next is the opening and closing parenthesis, followed by an opening brace, then we have a print statement. Finally, a closing brace. The opening and closing braces are what marks the body of the function. That's why the statement which should execute is placed inside of them. When the *printName* function is called, it is the statement that is in its body that would get executed. Here's the general syntax for defining a named function:

```
Return type  functionName() {  
    Statement 1;  
    Statement 2;  
    ...  
    Statement n;  
}
```

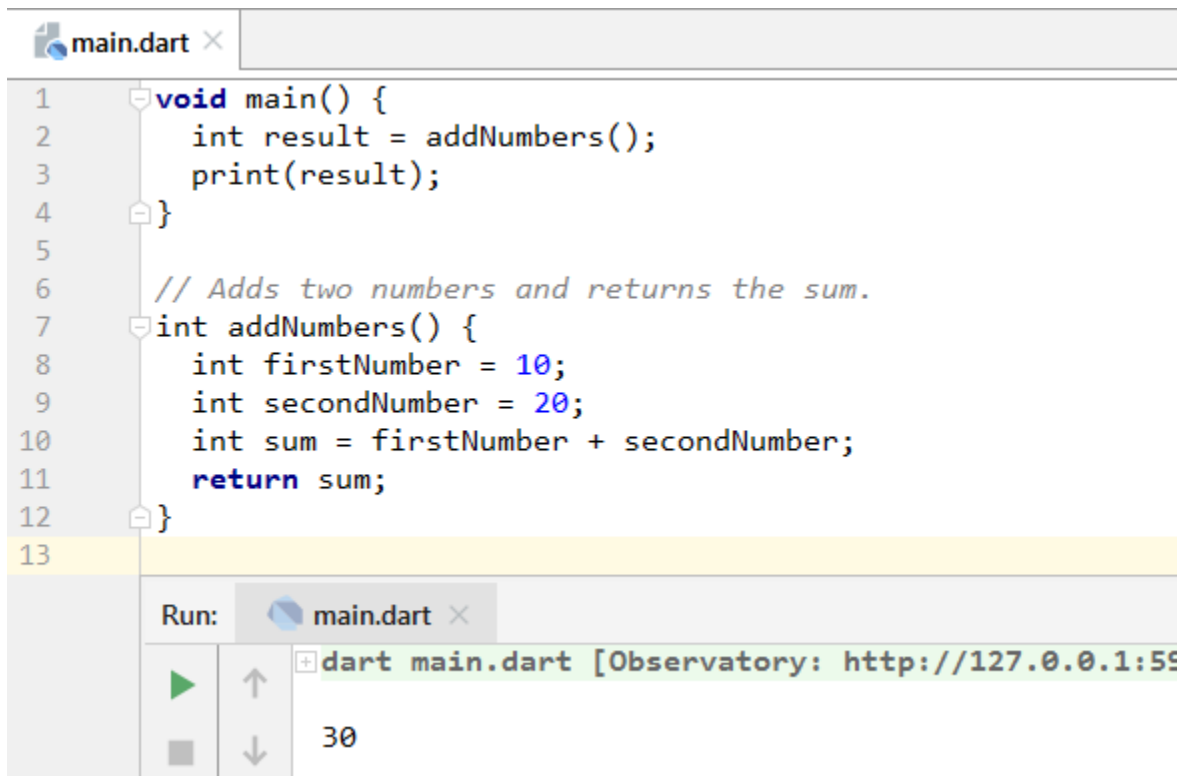
After a function is defined, in order for the function to run (execute), the function has to be called. That is where the *main* function comes in. From within the *main* function, we call the *printName* function. This causes the *printName* function to execute the statement it has in its body. That is why Moses Adebayo is shown in the console view

A good question one could ask is: If for the *printName* function to get executed, it has to be called from within the *main* function, what then calls the *main* function? The *main* function is called by Dart, which is why the *main* function is referred to as the starting point of program execution. It is the first function that executes when your program starts up.

## Returning Values from a Function

When we discussed the *printName* function, I mentioned that **void** is its return type. When functions are declared, they are usually given a return type, the return type signifies what kind of value the function would return back to the caller, when it is called.

In the case of *printName*, it returns no value, because it simply prints out some text to the console. If a function doesn't return any value, it is given a return type of **void**, just as we have it for the *main* and *printName* functions. However, when a function has to return a value of a particular type, it needs to have a return type. A good example is a function that adds two numbers together, after adding the numbers, it returns the sum of the numbers to the caller. In such a case, the function should have a return type of *int* or *double*, depending if it was integer or decimal values that were summed and would be returned. Let's look at some examples on how to define functions that return values to their caller.



The screenshot shows an IDE window with a file named `main.dart`. The code defines a `main` function and an `addNumbers` function. The `main` function calls `addNumbers` and prints its result. The `addNumbers` function takes no arguments and returns the sum of two integers, `firstNumber` (10) and `secondNumber` (20). The execution output at the bottom shows the program running successfully and printing the value 30.

```
1 void main() {  
2     int result = addNumbers();  
3     print(result);  
4 }  
5  
6 // Adds two numbers and returns the sum.  
7 int addNumbers() {  
8     int firstNumber = 10;  
9     int secondNumber = 20;  
10    int sum = firstNumber + secondNumber;  
11    return sum;  
12 }  
13
```

Run: `main.dart` ×

`dart main.dart [Observatory: http://127.0.0.1:59`

30

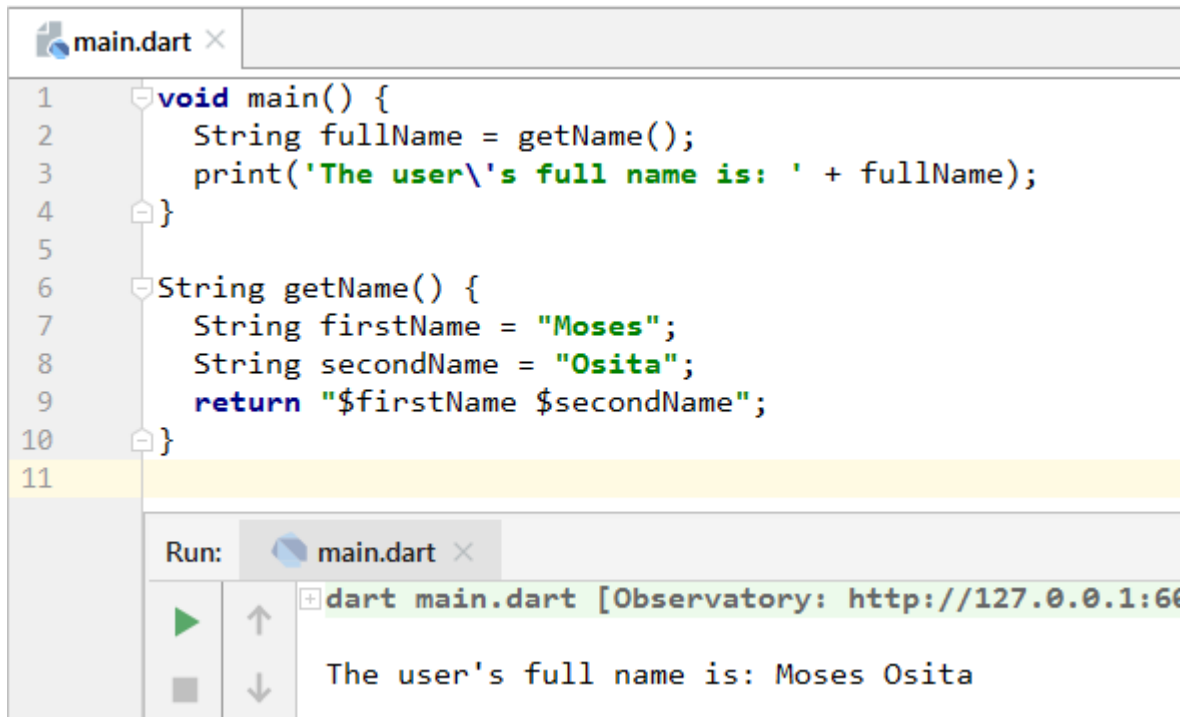
Screenshot 5.2

The `addNumbers` function in the program above has a return type of `int`. On line 10, you can see the plus (+) operator is used to add the values of `firstNumber` and `secondNumber`. So, 20 is added to 10, and the result is assigned to the `sum` variable. Observe that the `sum` variable had to be declared using type `int`. This is because `firstNumber` and `secondNumber` both hold integer values. So, adding them together would in turn yield an integer value. So the variable `sum` which holds the resulting value, has to be of type `int`.

Finally, on line 11, we get to return the result of the operation. **return** is a keyword. It is used when a value needs to be returned from a function. It has to be the last statement in the function, because it also causes the function to stop running, and to transfer execution to the calling function. **You shouldn't place any code statement(s) after the return statement in a function, for the code would be unreachable and won't be executed.**

Since the `addNumbers` function is called from within the `main` function, the value it returns gets received in the `main` function, i.e. on line 2, and it is assigned to the `result` variable, which is then printed to the console. Again, observe that the `result` variable had to be declared using type `int`. This is because the value that `addNumbers` returns, is of type `int`, as specified by its return type.

We aren't limited to defining functions that have a return type of `int`. Functions can have any return type (`String`, `bool`, `List`, `Map`, etc.). So, let's define a function that returns a `String` value.



The screenshot shows an IDE window titled 'main.dart'. The code defines a `main` function and a `getName` function. The `main` function calls `getName` and prints the result. The `getName` function returns a string formed by concatenating 'Moses' and 'Osita' with a space. Below the code, a 'Run' button is visible, and the output console shows the command `dart main.dart` and the output `The user's full name is: Moses Osita`.

```
1 void main() {  
2   String fullName = getName();  
3   print('The user\'s full name is: ' + fullName);  
4 }  
5  
6 String getName() {  
7   String firstName = "Moses";  
8   String secondName = "Osita";  
9   return "$firstName $secondName";  
10 }  
11
```

Run: main.dart x

dart main.dart [Observatory: http://127.0.0.1:6060]

The user's full name is: Moses Osita

Screenshot 5.3

Defining a function that returns a *String*, is just as similar to defining a function that returns a value of another type. One part of the code I want to draw your attention to, is how the return statement was used on line 9. Observe how both `String` variables were included in the returned `String` value, using string interpolation. Another way it could have been done, would have been to concatenate `firstName` with `secondName`, then put the resulting `String` value in another variable, which would then be returned.

## Parameters

Sometimes, when a function is defined, it requires some extra data that it needs for its internal operations. That is what parameters basically are; data that a function uses when executing. The question now is, where are parameters placed in a function? Let's extend the general structure for defining a function, with parameters included in it.



```
Return type  functionName (Type param 1, Type param 2, ... Type param n) {  
    Statement 1;  
    Statement 2;  
    ...  
    Statement n;  
}
```

Parameters are basically variables, they're defined in the parenthesis just after the name of the function. Parameters make functions more powerful and flexible. With parameters, functions are able to receive any valid data, which they use for their internal operations.

## Types of Parameters

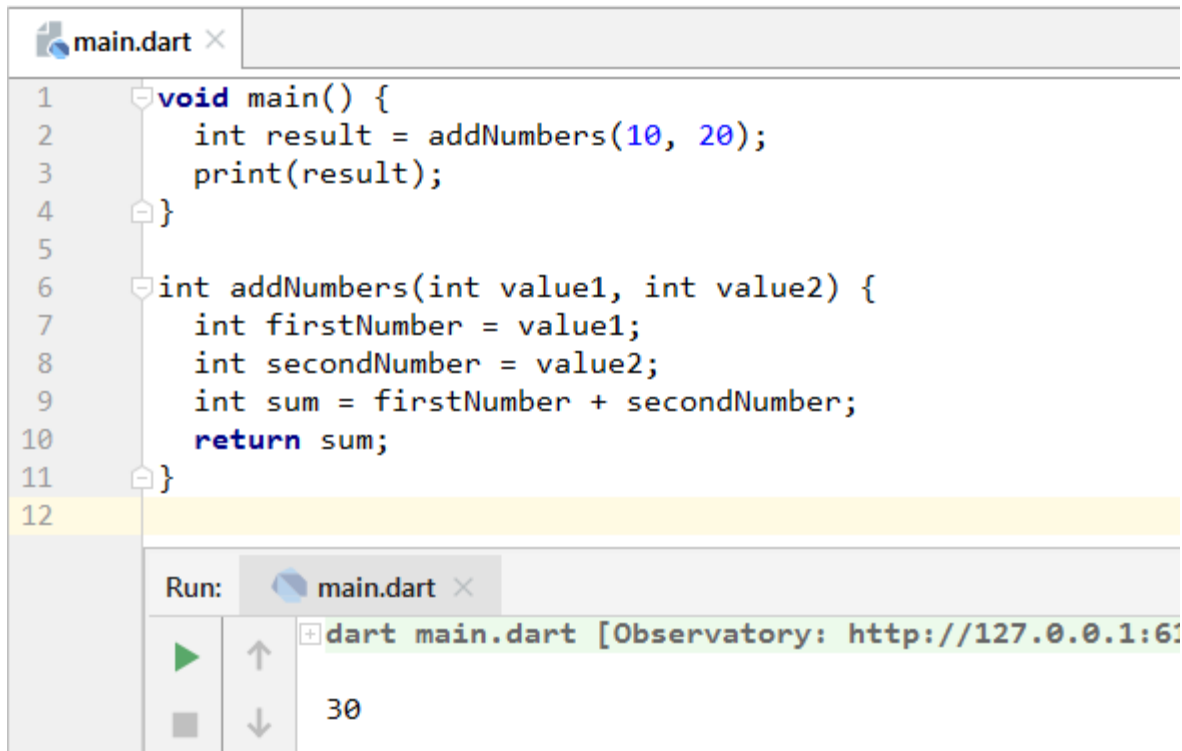
There are two types of parameters in Dart and they are:

1. Required Parameters
2. Optional Parameters

## Required Parameters

Just as the name implies, they are parameters (variables) that must be given values whenever a function is called. It would be an error to call a function that defines required parameters, without passing correct values (arguments) to it. Let's modify the *addNumbers* function we wrote earlier, to include parameters.

**Terminology Note:** Arguments are values passed to a function when it is called.



The screenshot shows an IDE window titled 'main.dart'. The code is as follows:

```
1 void main() {  
2     int result = addNumbers(10, 20);  
3     print(result);  
4 }  
5  
6 int addNumbers(int value1, int value2) {  
7     int firstNumber = value1;  
8     int secondNumber = value2;  
9     int sum = firstNumber + secondNumber;  
10    return sum;  
11 }  
12
```

Below the code editor, there is a 'Run:' button and a console window. The console shows the output of the program:

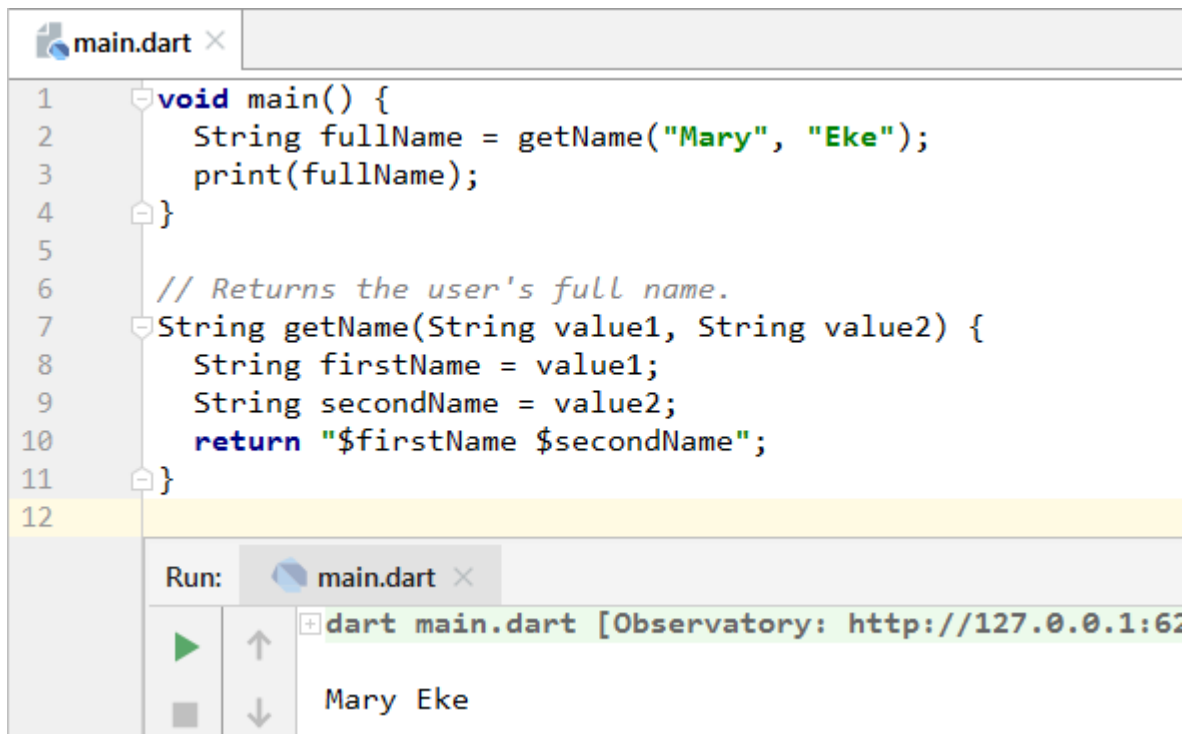
```
dart main.dart [Observatory: http://127.0.0.1:6154]  
30
```

Screenshot 5.4

The *addNumbers* function in the code sample above, performs the same operation as the previous one, but it is more flexible in how it can be used. Observe that after the function's name, in the parenthesis, two variables are defined, although they aren't assigned any values. Those two variables (*value1* and *value2*) are the parameters of the *addNumbers* function. When the *addNumbers* function is called, like it is done from within the *main* function, it has to be passed two integer values (arguments). In this case, we passed 10 and 20. The values passed to it are eventually assigned to the *value1* and *value2* variables. That is why within the function, we are able to assign *value1* and *value2* to *firstNumber* and *secondNumber* respectively.

If you call the *addNumbers* function without passing the required arguments, you'll get an error that "the function defines two parameters that need to be passed values".

Let's see another example of a function that defines String parameters, which is just a modification of the example we looked at before.



The screenshot shows an IDE window titled 'main.dart'. The code defines a `main` function that calls `getName` with arguments 'Mary' and 'Eke'. The `getName` function concatenates these arguments with a space. Below the code editor, a 'Run' button is visible. To its right, a console window shows the output 'Mary Eke'.

```
1 void main() {  
2     String fullName = getName("Mary", "Eke");  
3     print(fullName);  
4 }  
5  
6 // Returns the user's full name.  
7 String getName(String value1, String value2) {  
8     String firstName = value1;  
9     String secondName = value2;  
10    return "$firstName $secondName";  
11 }  
12
```

Run: main.dart ×

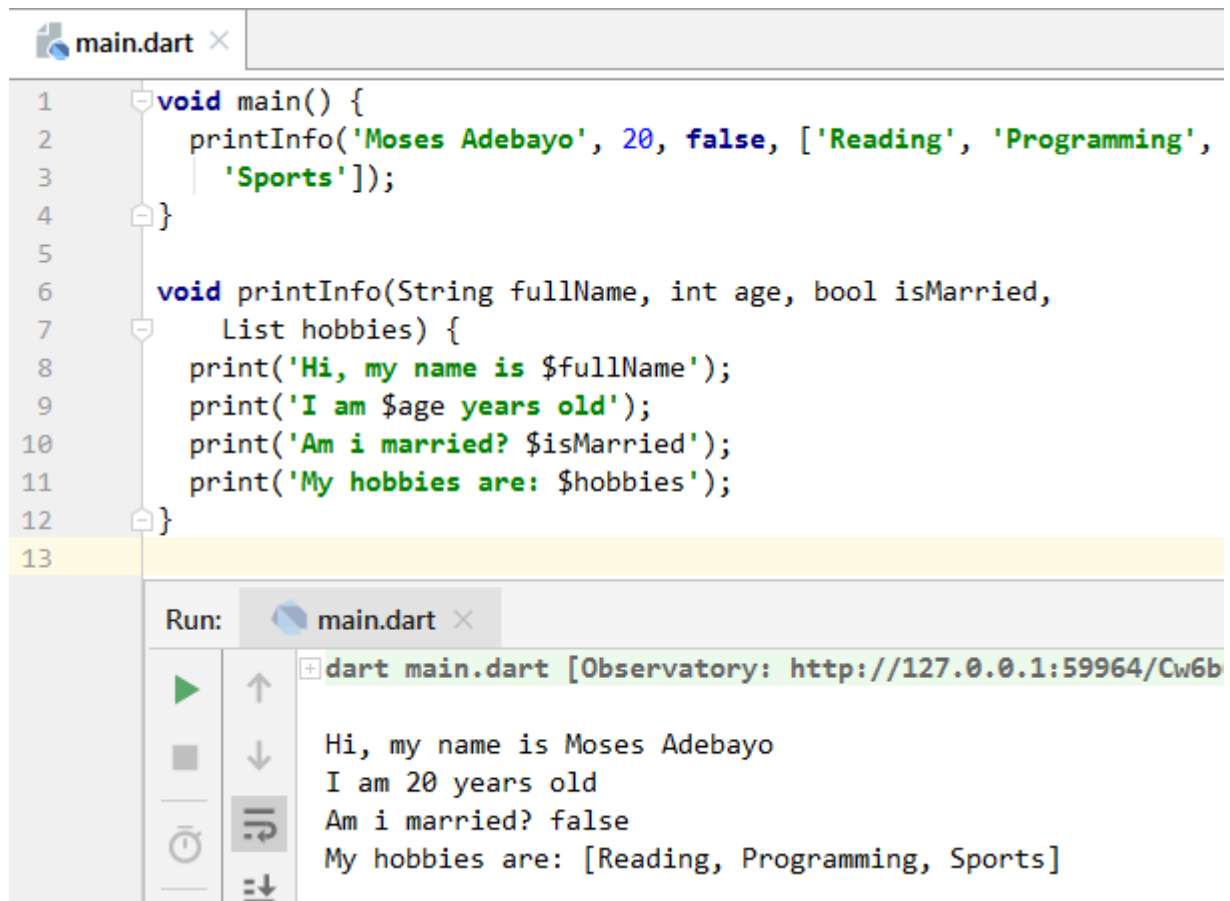
dart main.dart [Observatory: http://127.0.0.1:62...

Mary Eke

Screenshot 5.5

One other thing to note about required parameters is that, when a function that defines required parameters is called, the arguments passed to the function must be passed in the exact order which the parameters are defined. Take the `getName` function in the program above, if you flip the order of the arguments from `getName('Mary', 'Eke')` to `getName('Eke', 'Mary')`, then 'Eke' would be assigned to the `value1` parameter, while 'Mary' would be assigned to the `value2` parameter. That's how it works. So always make sure that for required parameters you pass the arguments in the exact order that the parameters were defined, having the type of each parameter in mind. If not, you might get unexpected results. As an exercise, try change the order of the names we passed to the `getName` function in the program above. That would cause the order of the names to change when they're printed to the console.

The program below shows how parameters of different types can be defined for a function.



The screenshot shows an IDE window with a file named `main.dart`. The code defines a `main` function that calls `printInfo` with arguments for a person's name, age, marital status, and hobbies. The `printInfo` function is a helper that prints these details. Below the code editor, a 'Run' panel shows the execution of `main.dart`, displaying the output of the `print` statements.

```
1 void main() {  
2   printInfo('Moses Adebayo', 20, false, ['Reading', 'Programming',  
3     'Sports']);  
4 }  
5  
6 void printInfo(String fullName, int age, bool isMarried,  
7   List hobbies) {  
8   print('Hi, my name is $fullName');  
9   print('I am $age years old');  
10  print('Am i married? $isMarried');  
11  print('My hobbies are: $hobbies');  
12 }  
13
```

Run: `main.dart` x

`dart main.dart [Observatory: http://127.0.0.1:59964/Cw6b`

```
Hi, my name is Moses Adebayo  
I am 20 years old  
Am i married? false  
My hobbies are: [Reading, Programming, Sports]
```

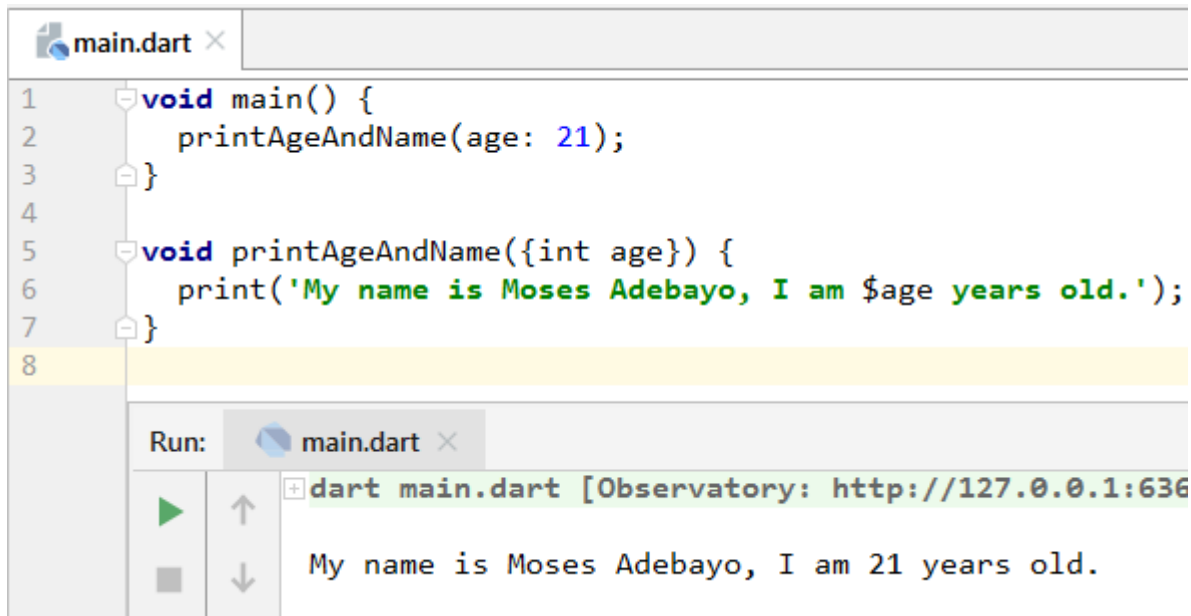
Screenshot 5.6

## Optional Parameters

Optional parameters are true to their name. When a function uses optional parameters, it means that the function can be called with or without arguments. Optional parameters are broken down into two:

1. Optional Named Parameters and
2. Optional Positional Parameters

To better understand the difference between the two, let's see examples on how they are used, starting with optional named parameters.



```
main.dart x
1 void main() {
2   printAgeAndName(age: 21);
3 }
4
5 void printAgeAndName({int age}) {
6   print('My name is Moses Adebayo, I am $age years old.');
```

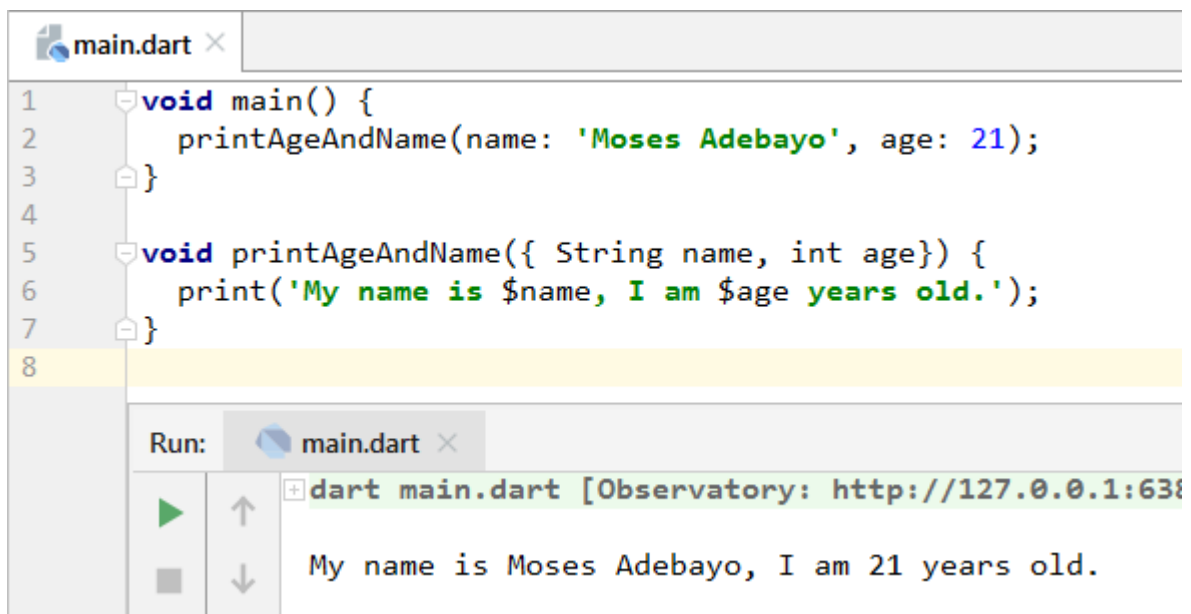
Run: main.dart x

dart main.dart [Observatory: http://127.0.0.1:6363]

My name is Moses Adebayo, I am 21 years old.

Screenshot 5.6

The *printAgeAndName* function defines an optional named parameter. Observe the braces that are used in wrapping the *age* parameter. It is the braces that make it optional and named. It is referred to as named, because, when the function is called, the name of the parameter must be specified. Just as it is done on line 2 of the program. The name of the parameter, followed by a colon, then the value that is to be passed to the parameter. If there is more than one parameter, then they're separated by a comma, as shown in the example below.



```
main.dart x
1 void main() {
2   printAgeAndName(name: 'Moses Adebayo', age: 21);
3 }
4
5 void printAgeAndName({ String name, int age}) {
6   print('My name is $name, I am $age years old.');
```

Run: main.dart x

dart main.dart [Observatory: http://127.0.0.1:6363]

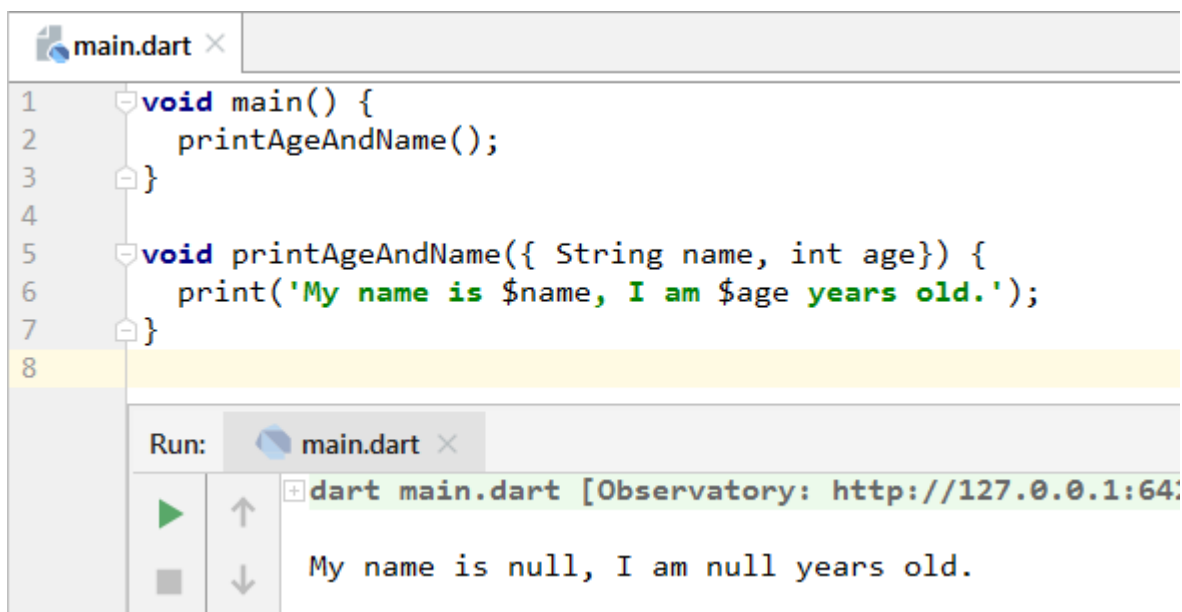
My name is Moses Adebayo, I am 21 years old.

Screenshot 5.7

Here, we've defined two optional named parameters (*name* and *age*). When the function is called, the names of the parameters have to be specified and the appropriate values passed to them. Values for named parameters can be passed in any order, unlike for required parameters that we looked at earlier. Try calling the *printAgeAndName* function and pass the arguments in any order, using the name of the parameters, you would get the same result. This is possible, because Dart is able to match each value that is provided when the function is called, to the parameter, using its name.

We've seen how optional named parameters can be defined, which only boils down to wrapping the parameter(s) with curly brackets and using the name(s) to pass values when the function is called.

Remember that they are also referred to as **optional**. So, the question is, Can we call a function that defines optional named parameters without actually passing any or all of the arguments to it? The answer is yes. Let's test that out.



The screenshot shows an IDE window titled 'main.dart'. The code is as follows:

```
1 void main() {  
2   printAgeAndName();  
3 }  
4  
5 void printAgeAndName({ String name, int age }) {  
6   print('My name is $name, I am $age years old.');7 }  
8
```

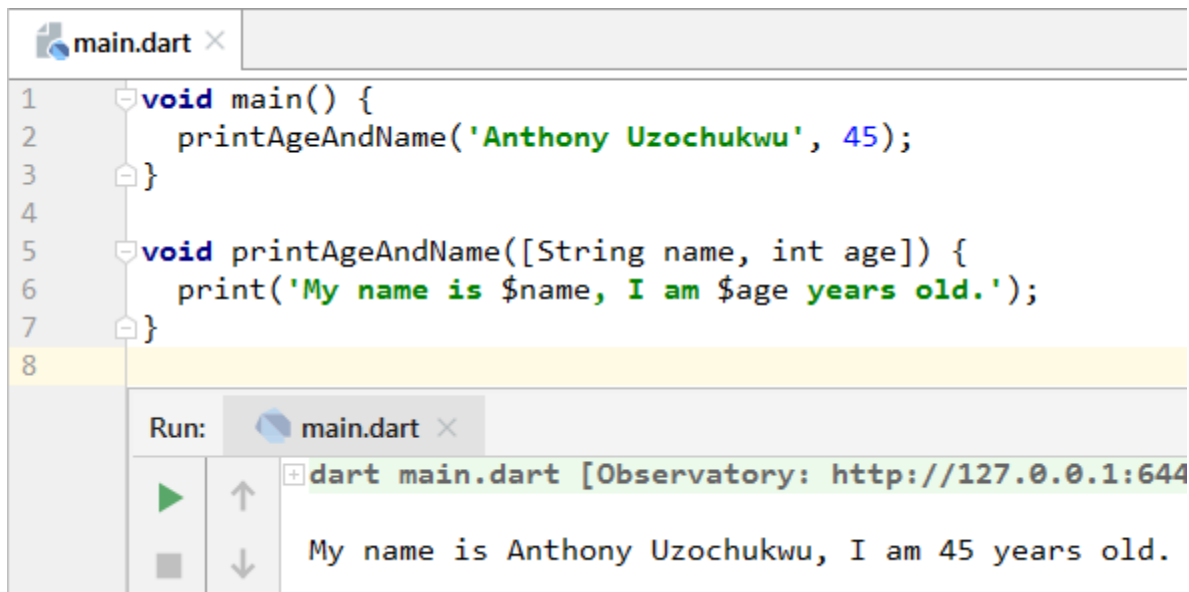
Below the code editor, there is a 'Run:' button and a console output area. The console shows the following output:

```
dart main.dart [Observatory: http://127.0.0.1:6464]  
My name is null, I am null years old.
```

Screenshot 5.8

Here, the *printAgeAndName* function is called without any arguments. This is only possible because *name* and *age* are optional named parameters in the *printAgeAndName* function. What is printed to the console is a confirmation that no argument(s) were passed when the function was called. *My name is null, I am null years old*. Remember that when a variable is not assigned any value, it retains the default value *null*.

Next, let's look at an example on how optional positional parameters are defined.



The screenshot shows an IDE window with a file named `main.dart`. The code defines a `main` function that calls `printAgeAndName` with the arguments `'Anthony Uzochukwu'` and `45`. The `printAgeAndName` function is defined with two parameters, `String name` and `int age`, and prints a string using these parameters. The code is as follows:

```
1 void main() {  
2   printAgeAndName('Anthony Uzochukwu', 45);  
3 }  
4  
5 void printAgeAndName([String name, int age]) {  
6   print('My name is $name, I am $age years old.');7 }  
8
```

Below the code editor, there is a 'Run' button and a console window. The console shows the output of the program: `My name is Anthony Uzochukwu, I am 45 years old.`

Screenshot 5.9

Observe the syntax for defining optional positional parameters. In contrast to optional named parameters, it uses square brackets in wrapping the parameters. Also, observe how the arguments are passed when the function is called, the names of the parameters aren't used.

For optional positional parameters, it is also possible to call the function without passing any arguments. If you were to do such in this program, you would get the same result of *null* for the *name* and *age* parameters.

We've learnt about parameter definition and argument passing for functions. We've looked at the different ways these parameters can be defined and how arguments can be passed. You may be wondering why there are so many options for defining parameters and passing arguments. Isn't one method enough? Well, it is true that your programs won't require all the different methods at a time. There are times when a particular method is more appropriate or useful than the other. You're going to have to always pick the one that best fits your program. Take for example, you have a function that defines up to 5 or 6 parameters. You may want to use optional named parameters in that case, because it would make it easy for you to be able to provide the arguments without having them mixed up. With the IntelliJ IDEA Editor running on a windows machine, when you call a function that defines optional named parameters, you can place the cursor in between the parenthesis of the function call, and press **ctrl + space**. Doing so will bring up a list of the parameters that are defined in the function.

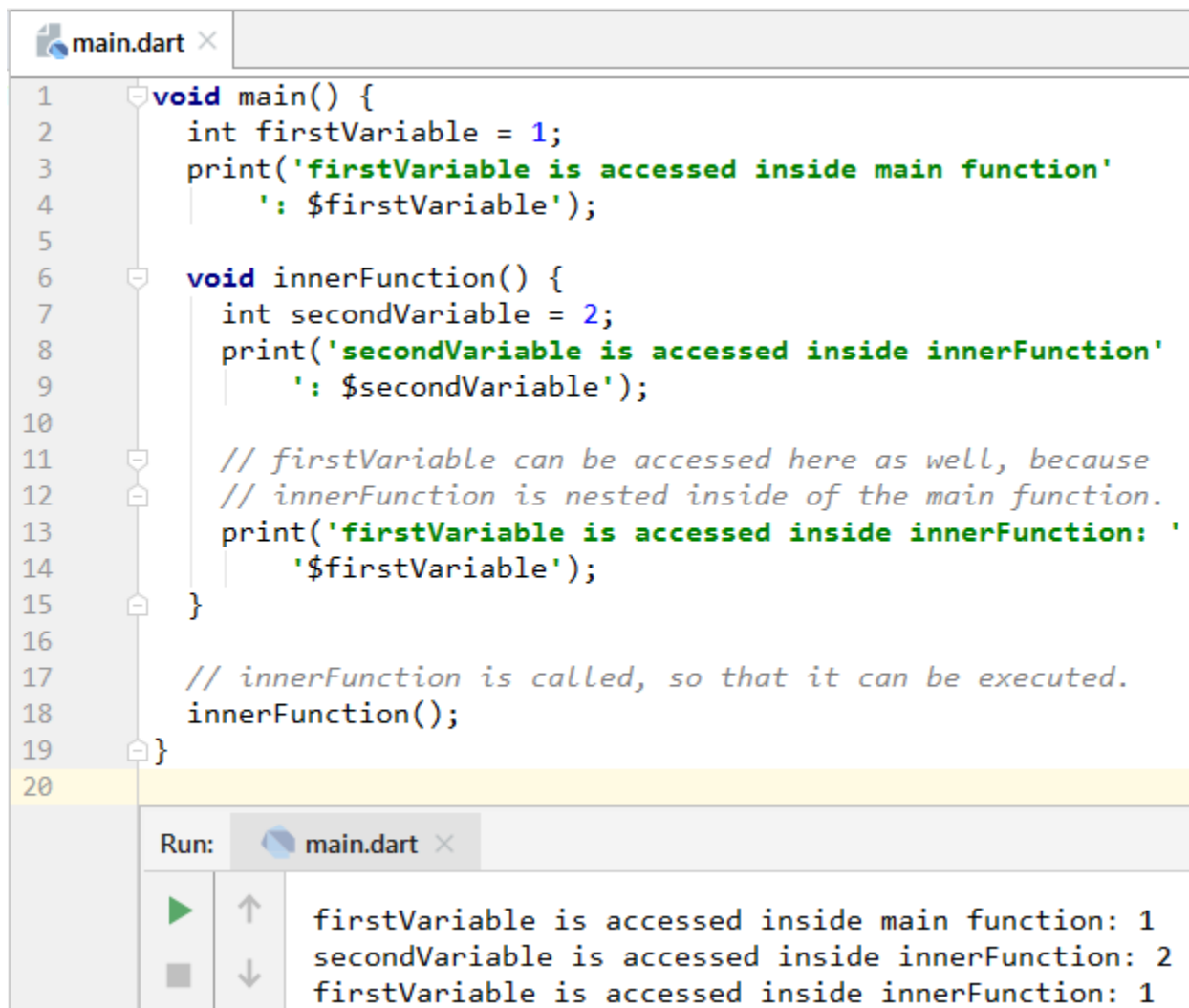
## Lexical Scope

Lexical scope or scope for short, refers to visibility. Dart is a lexically scoped language. What that means is that, you can basically know the beginning and end of a program

block (a function for example) by their opening and closing braces. A function's body begins with an opening brace and ends with a closing brace. Everything within the body of a function can be said to be within the scope of the function.

Imagine there are two boxes, a bigger box and a smaller box. The bigger box encloses the smaller box. In the bigger box, there are some items, also the smaller box contains some items. However, there is a strict rule that no item in the smaller box can be taken out of it and used in the bigger box, but items in the bigger box can be taken and used in the smaller box. All the things in the smaller box are within the scope of it, but those things aren't visible in the larger box. While the things in the bigger box are within its scope as well but are visible to the smaller box, because the smaller box is kind of like in the scope of the larger box as well.

That is what lexical scope is like in Dart, it is all about **visibility, visibility and visibility**. What is visible to what, where can what be accessed and where can't it be accessed. Let's write a program to see lexical scope in play.



```
1 void main() {
2   int firstVariable = 1;
3   print('firstVariable is accessed inside main function
4     ': $firstVariable');
5
6   void innerFunction() {
7     int secondVariable = 2;
8     print('secondVariable is accessed inside innerFunction'
9       ': $secondVariable');
10
11     // firstVariable can be accessed here as well, because
12     // innerFunction is nested inside of the main function.
13     print('firstVariable is accessed inside innerFunction: '
14       '$firstVariable');
15   }
16
17   // innerFunction is called, so that it can be executed.
18   innerFunction();
19 }
20
```

Run: main.dart

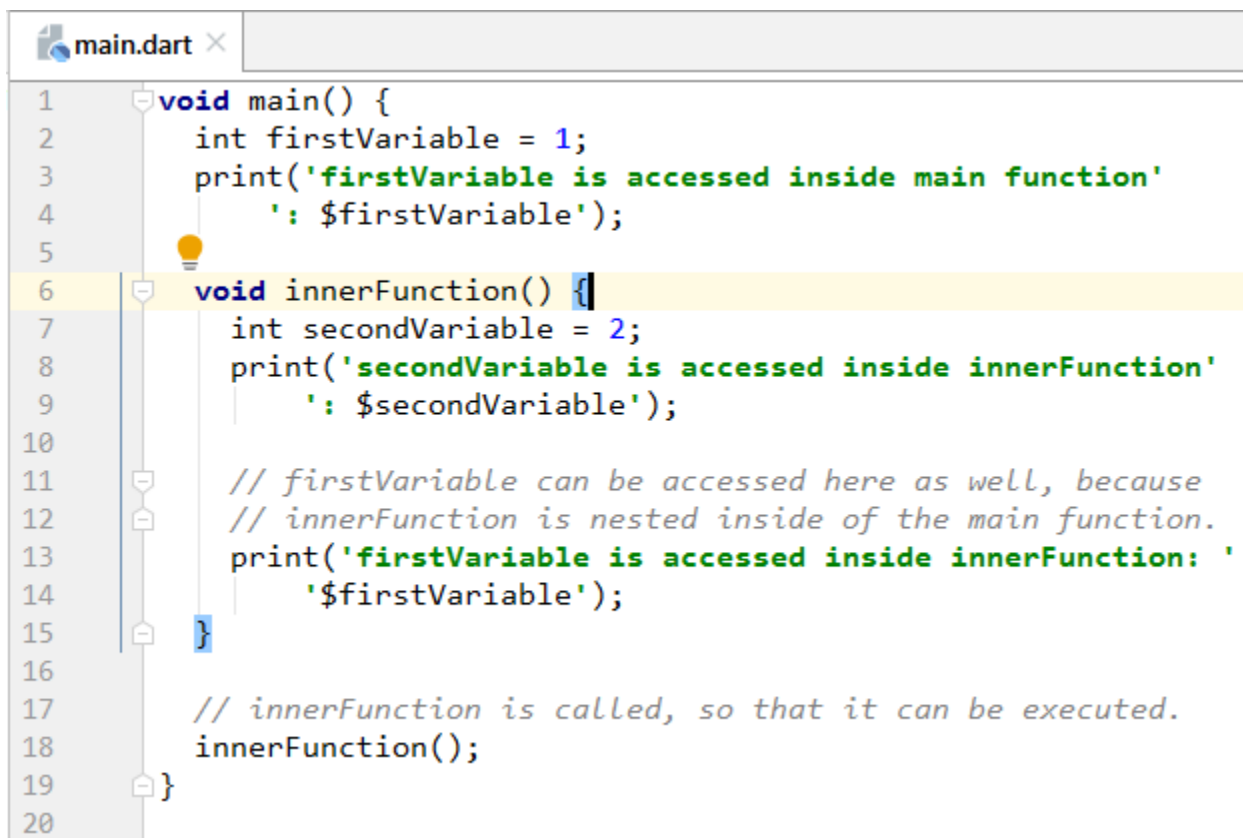
```
firstVariable is accessed inside main function: 1
secondVariable is accessed inside innerFunction: 2
firstVariable is accessed inside innerFunction: 1
```



Screenshot 5.10

In the above program, you can see how the different variables are being accessed. Only the variable(s) within a function's scope can be accessed from within it. On your own, try to print out the *secondVariable* or assign a new value to it from within the *main* function. You would get an error that the variable is not defined, because the *secondVariable* is only visible in the *innerFunction*. Observe that *innerFunction* was called on line 18 after its definition inside *main*. This is because, we cannot call a function before it is defined. To place a call to *innerFunction* before its definition would result in an error, due to the fact that Dart executes the program from top to bottom.

If you're ever in doubt about the scope of a function, suppose you want to know where it begins and ends, you can simply place the cursor at the opening brace and it would highlight the closing brace for you, as shown in the screenshot below. Let's use the cursor to find out where *innerFunction* begins and ends.

A screenshot of an IDE window titled 'main.dart'. The code is as follows:

```
1 void main() {
2   int firstVariable = 1;
3   print('firstVariable is accessed inside main function'
4         ': $firstVariable');
5
6   void innerFunction() {
7     int secondVariable = 2;
8     print('secondVariable is accessed inside innerFunction'
9           ': $secondVariable');
10
11     // firstVariable can be accessed here as well, because
12     // innerFunction is nested inside of the main function.
13     print('firstVariable is accessed inside innerFunction: '
14           '$firstVariable');
15   }
16
17   // innerFunction is called, so that it can be executed.
18   innerFunction();
19 }
20
```

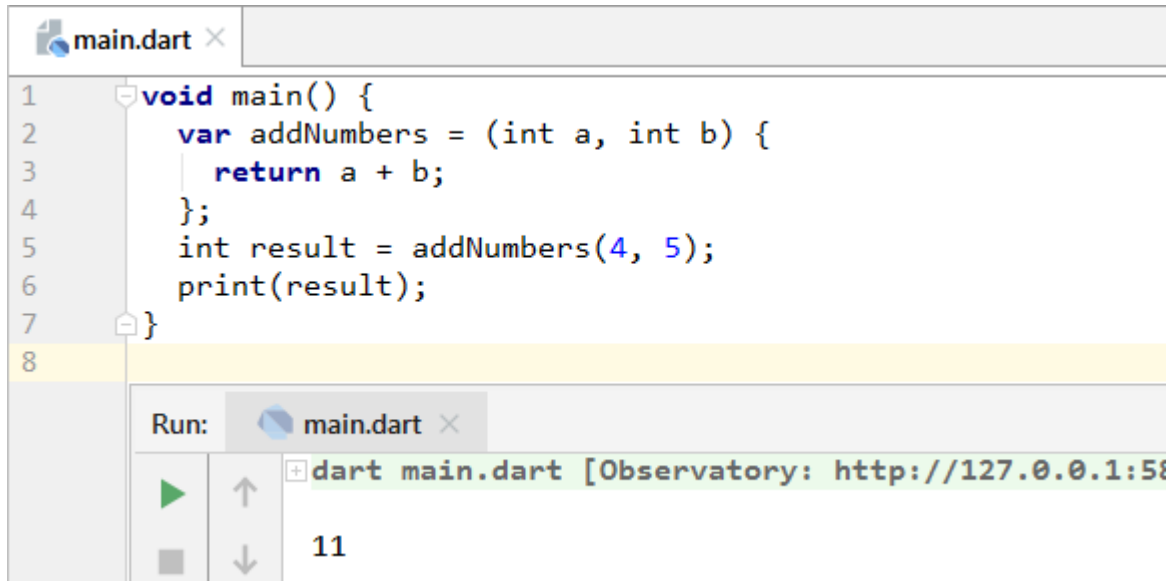
The line containing the definition of `innerFunction` (lines 6-15) is highlighted in yellow. A blue cursor is positioned at the opening curly brace of `innerFunction` on line 6. The closing curly brace on line 15 is also highlighted in yellow, indicating the function's scope.

Screenshot 5.11

When the cursor is placed at the opening brace of the function, the closing brace is highlighted along with it. This is just one of the ways the IntelliJ IDEA code editor helps us to easily find our way in such a program that contains nested blocks of code.

## Anonymous Functions (Nameless Functions)

An anonymous function is a function without a name. One of the ways an anonymous function can be used is passing it as a value to a variable. An example is shown below.



The screenshot shows an IDE window titled 'main.dart'. The code is as follows:

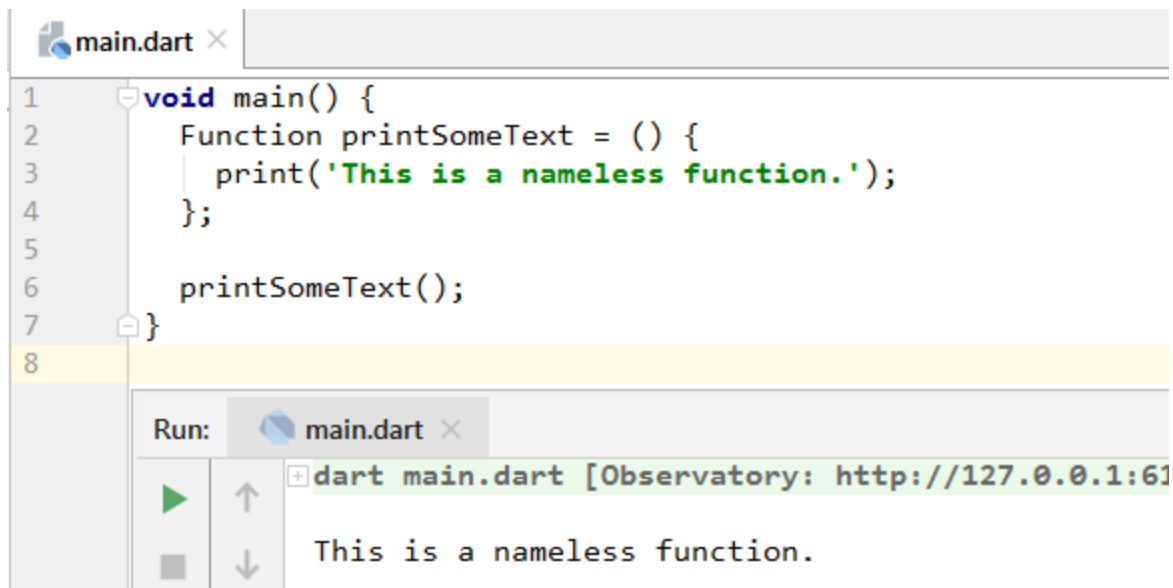
```
1 void main() {  
2     var addNumbers = (int a, int b) {  
3         return a + b;  
4     };  
5     int result = addNumbers(4, 5);  
6     print(result);  
7 }  
8
```

Below the code editor, there is a 'Run' button and a console output area. The console shows the command 'dart main.dart' and the output '11'. The output area also includes the text '[Observatory: http://127.0.0.1:5858]'.

Screenshot 5.12

Notice that the function after the equals sign has no name. It is an anonymous function. One easy way to understand how passing an anonymous function to a variable works, is to reason the variable name as becoming the name of the anonymous function. So that when the function that is saved in the variable is to be used (called), one simply need add a pair of parenthesis to the name of the variable, as was done on line 5.

If the anonymous function defines any parameters, then the right arguments have to be passed. We can also write anonymous functions that don't define any parameters. An example of this is shown below.

The screenshot shows an IDE window with a file named 'main.dart'. The code is as follows:

```
1 void main() {  
2     Function printSomeText = () {  
3         print('This is a nameless function.');4     };  
5  
6     printSomeText();  
7 }  
8
```

Line 8 is highlighted in yellow. Below the code editor is a 'Run' button and a console window. The console shows the command 'dart main.dart [Observatory: http://127.0.0.1:61...]' and the output 'This is a nameless function.'

Screenshot 5.13

One major thing to observe in the program above is the type of the *printSomeText* variable. It is declared using the special type **Function**. That is because the type of a function is **Function**. So, since an anonymous function would be assigned to the *printSomeText* variable, it can be declared using the **Function** type, which visibly, is more precise than using the **var** keyword, as was done in the previous example.

In a later chapter, we shall look at how an anonymous function can be passed as an argument to a function.

## Summary

In this chapter, you've learned much about writing functions in Dart. Functions are very essential in programming, as they allow us write reusable code, which can come in very handy. Parameters are used to make a function flexible. With different inputs, the function is able to produce different results based on the input. In a function that returns a value, remember to not place any statements after return is used.

## Exercises

1. Write a function that defines three int parameters (required), it adds their values together and then prints the sum when the function is called.
2. Write a function that defines two String parameters (required), it concatenates the values of the parameters and returns the resulting String value.
3. Rewrite the functions you wrote for questions 1 & 2, using optional named parameters

4. Rewrite the functions you wrote for questions 1 & 2, using optional positional parameters.

## CHAPTER 6

# CONTROL FLOW STATEMENTS

So far, we've only seen how to write code that gets executed sequentially i.e. from top to bottom. In this chapter, we shall learn how to write code that doesn't necessarily get executed from top to bottom, in a sequential order. Such code usually entails multiple code paths in which the decision on which code should be executed is made based on some condition. A decision is made on whether to deviate from the current path of execution and move to a different path, or to skip an entire block of code, or to repeat the execution of a piece of code for a certain number of times.

Control flow statements can be grouped into three major categories:

3. Selection Statements
4. Iteration/Repetition Statements
5. Jump Statements

### Selection Statements

Selection statements are all about making decisions. With selection statements, we're able to write code that tells Dart to make a decision between two or more code paths, which to execute, based on some condition or test. The selection statements include:

3. If, else and else if statements
4. Switch statement

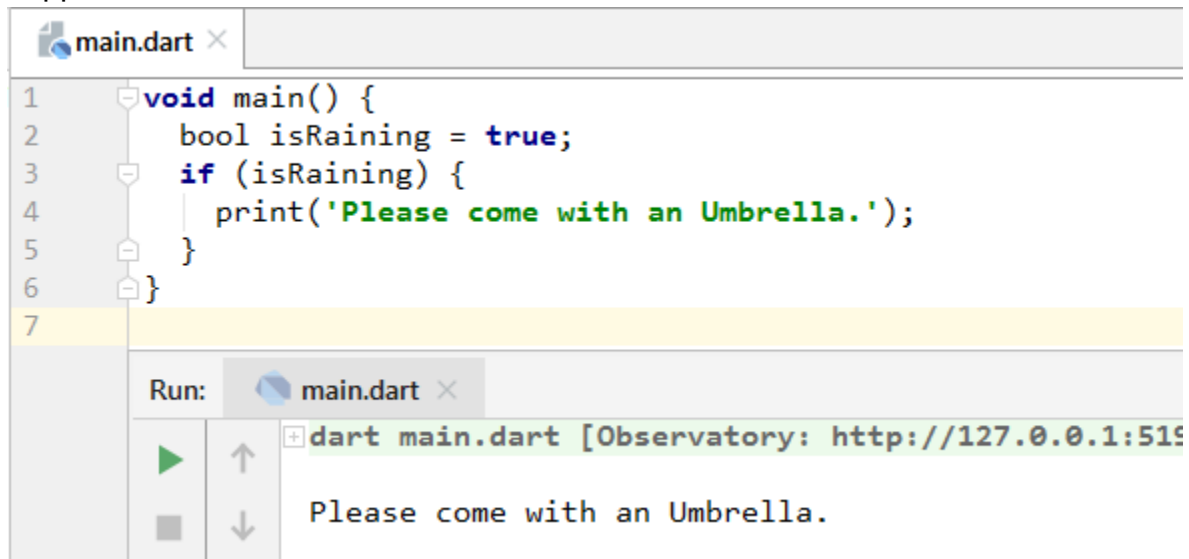
### If and else statement

The if and else statement has the following general structure;

```
if (condition) {  
    Statement 1;  
    Statement 2;  
    .....  
    Statement n;  
}
```

The if statement begins with the Dart keyword, **if**. Which is then followed by a pair of parenthesis. Inside the parenthesis goes the **condition**. The condition could be either a **bool** value of true/false or an expression that evaluates to a bool value of true/false. If the condition is true, then the code block immediately following the condition would be

branched into and the statement(s) executed, while if the condition is false, then nothing happens.



```
main.dart x
1 void main() {
2   bool isRaining = true;
3   if (isRaining) {
4     print('Please come with an Umbrella.');
```

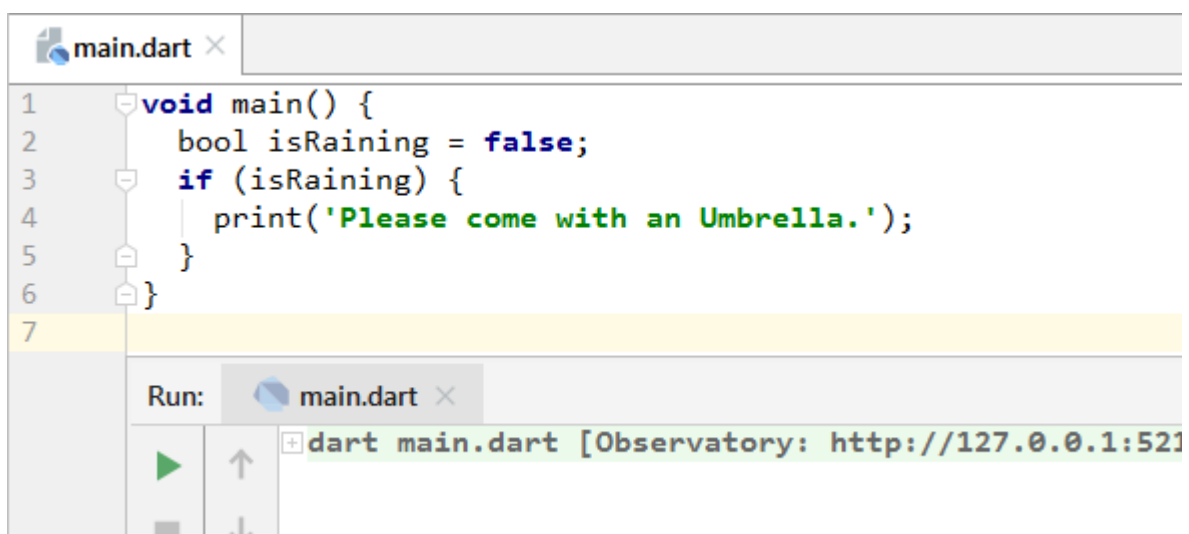
Run: main.dart x

dart main.dart [Observatory: http://127.0.0.1:5195]

Please come with an Umbrella.

Screenshot 6.1

This is a simple weather program. The program's structure makes it very self-explanatory. The *isRaining* variable is a bool variable, meaning it can hold either true or false, i.e. if it is raining or not. The *if* statement simply acts based on the value of *isRaining*. If *isRaining* is true, then Dart would have to execute the block of code that comes immediately after the condition, which simply prints out *Please come with an Umbrella.* as shown in the console. After the printing is done, the program would stop executing. How about the case of when *isRaining* is false. What would happen? Well, I am sure you guessed it, nothing would get printed.



```
main.dart x
1 void main() {
2   bool isRaining = false;
3   if (isRaining) {
4     print('Please come with an Umbrella.');
```

Run: main.dart x

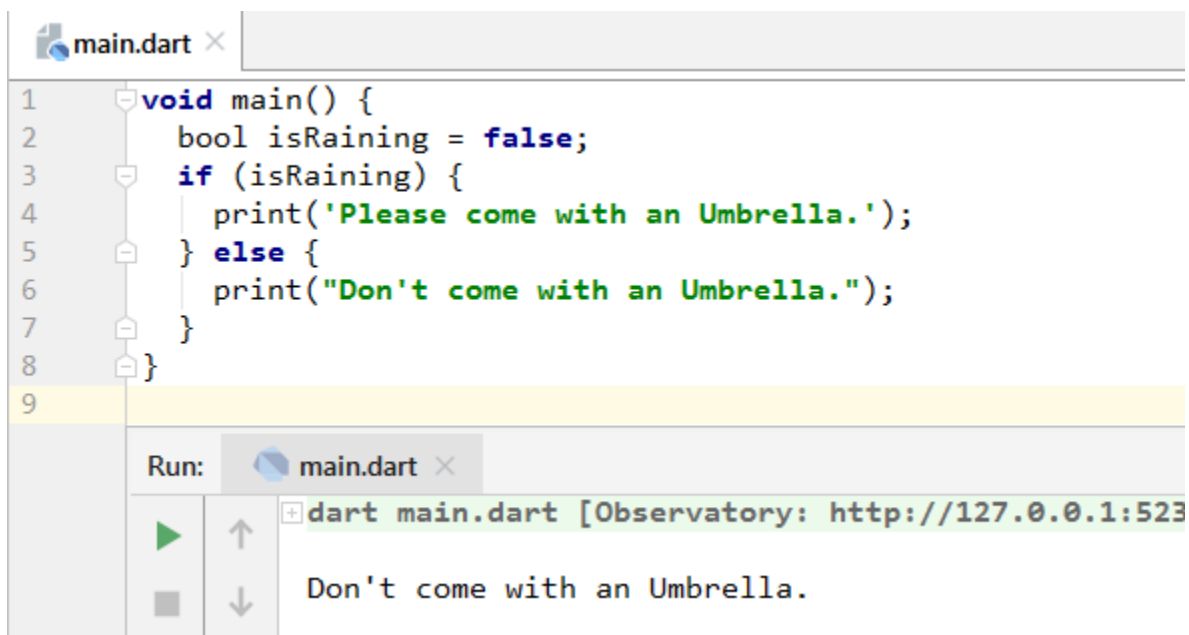
dart main.dart [Observatory: http://127.0.0.1:5210]

Screenshot 6.2

As you can see, when *isRaining* is false, nothing gets printed, because when Dart checks the value of *isRaining*, it finds that it is false, so it doesn't executed what is in the if block.

In order to tell Dart what should be done when the condition in the if statement is false, we introduce the **else** statement, which provides an alternative block of code that should be executed when the if condition is false. Adding the else statement to the general structure of the if statement above, gives:

```
if (condition) {  
    Statement 1;  
    Statement 2;  
    .....  
    Statement n;  
} else {  
    Statement 1;  
    Statement 2;  
    .....  
    Statement n;  
}
```



The screenshot shows an IDE window titled 'main.dart'. The code is as follows:

```
1 void main() {  
2     bool isRaining = false;  
3     if (isRaining) {  
4         print('Please come with an Umbrella.');5     } else {  
6         print("Don't come with an Umbrella.");  
7     }  
8 }  
9
```

Below the code editor, there is a 'Run' button and a console output area. The console shows the output of the program:

```
dart main.dart [Observatory: http://127.0.0.1:523  
Don't come with an Umbrella.
```

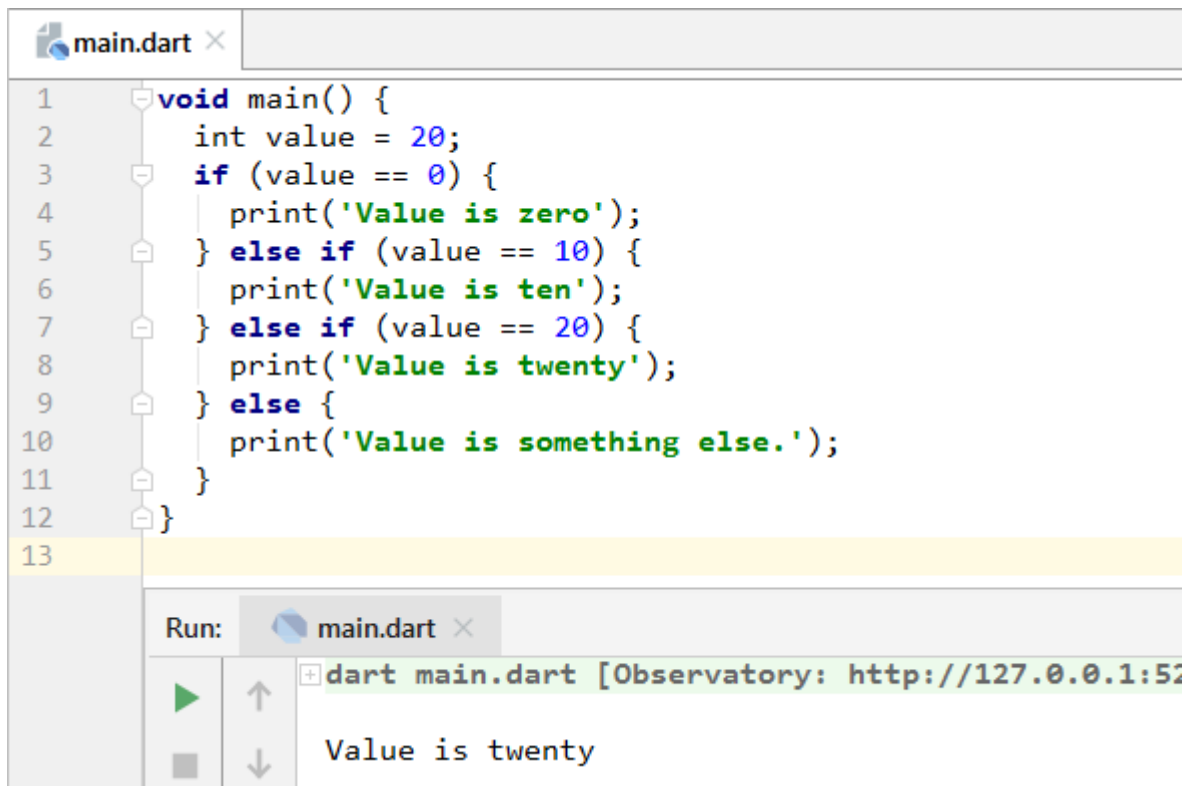
Screenshot 6.3

Unlike the previous code, something now happens when the if condition is false.

There are also times when just having two code paths is not enough, you may want to define multiple code paths, where only the one that matches a condition is executed. We can extend the if statement to accommodate as many code paths as required through the use of the **else if** statement. Adding the else if statement to the general structure of the if statement gives:

```
if (condition) {  
    Statement 1;  
    Statement 2;  
    .....  
    Statement n;  
}  
else if (condition) {  
    Statement 1;  
    Statement 2;  
    ....  
    Statement n;  
} else {  
    Statement 1;  
    Statement 2;  
    .....  
    Statement n;  
}
```





```
1 void main() {
2   int value = 20;
3   if (value == 0) {
4     print('Value is zero');
5   } else if (value == 10) {
6     print('Value is ten');
7   } else if (value == 20) {
8     print('Value is twenty');
9   } else {
10    print('Value is something else.');
```

Run: main.dart x

dart main.dart [Observatory: http://127.0.0.1:52]

Value is twenty

Screenshot 6.4

The example uses the **else if** to include multiple conditions for the **if** statement. Although, no matter how many conditions are present, only the block of code for the one that is true would get executed. When all the conditions are false, then the code in the **else** block would get executed. The next example shows how you can define conditions with more complex boolean expressions.



The screenshot shows an IDE window titled 'main.dart'. The code is as follows:

```
1 void main() {  
2   int age = 18;  
3   if (age >= 18 && age <= 35) {  
4     print('You are invited to the party.');5   } else if (age >= 1 && age < 18) {  
6     print("You're too young.");  
7   } else if (age > 35 && age <= 120) {  
8     print("You're too old.");  
9   }  
10 }  
11
```

Below the code editor, there is a 'Run:' section with a play button and a dropdown menu showing 'main.dart'. To the right of the play button, the command 'dart main.dart [Observatory: http://127.0.0.1:53...]' is visible. The output of the program is displayed below the command: 'You are invited to the party.'

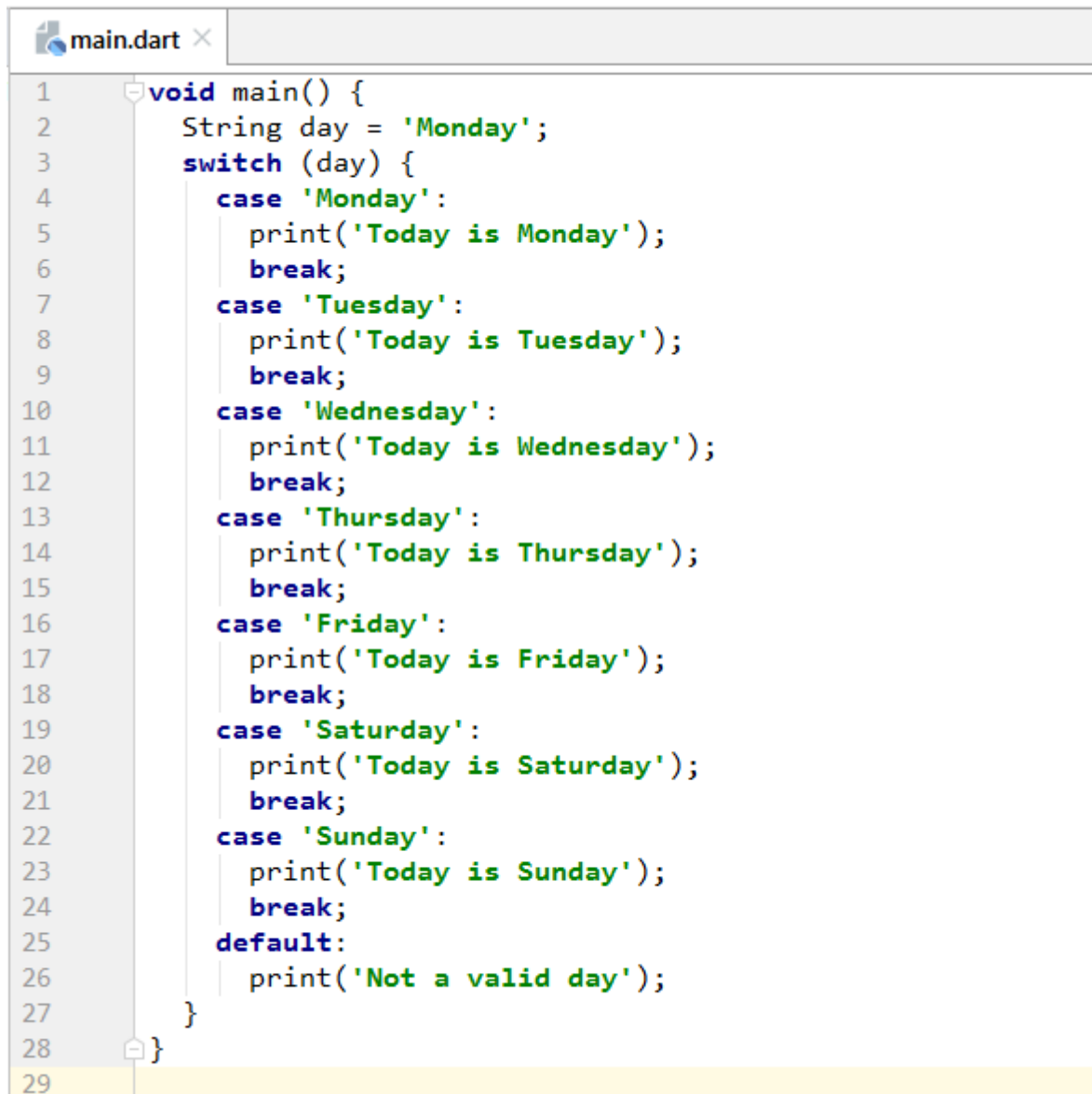
Screenshot 6.3

The condition of the **if** statement or **else if** statement can be any valid boolean expression. Notice that there's no **else** block. The **else** block, just like the **else if** block is also optional.

## Switch Statement

With the **else if**, it is possible to include multiple conditions for the **if** statement, however when there are too many **else if** statements, the program may become verbose and inelegant. The solution to this is the **switch** statement, which provides a more elegant way of performing multiple code checks or tests.

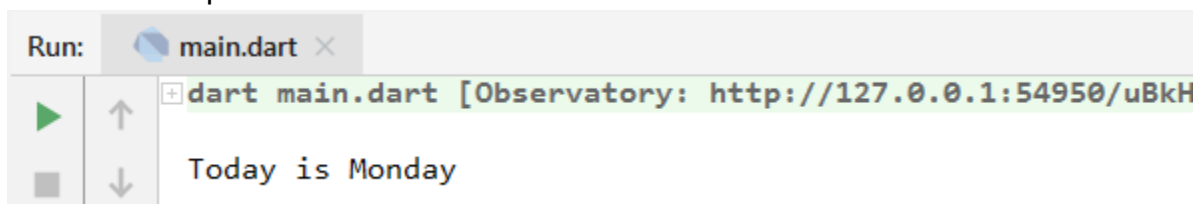
The **switch** statement entails testing the value of a variable, to see if it matches a particular case.



```
1 void main() {
2   String day = 'Monday';
3   switch (day) {
4     case 'Monday':
5       print('Today is Monday');
6       break;
7     case 'Tuesday':
8       print('Today is Tuesday');
9       break;
10    case 'Wednesday':
11      print('Today is Wednesday');
12      break;
13    case 'Thursday':
14      print('Today is Thursday');
15      break;
16    case 'Friday':
17      print('Today is Friday');
18      break;
19    case 'Saturday':
20      print('Today is Saturday');
21      break;
22    case 'Sunday':
23      print('Today is Sunday');
24      break;
25    default:
26      print('Not a valid day');
27  }
28 }
29
```

Screenshot 6.4

Here's the output:

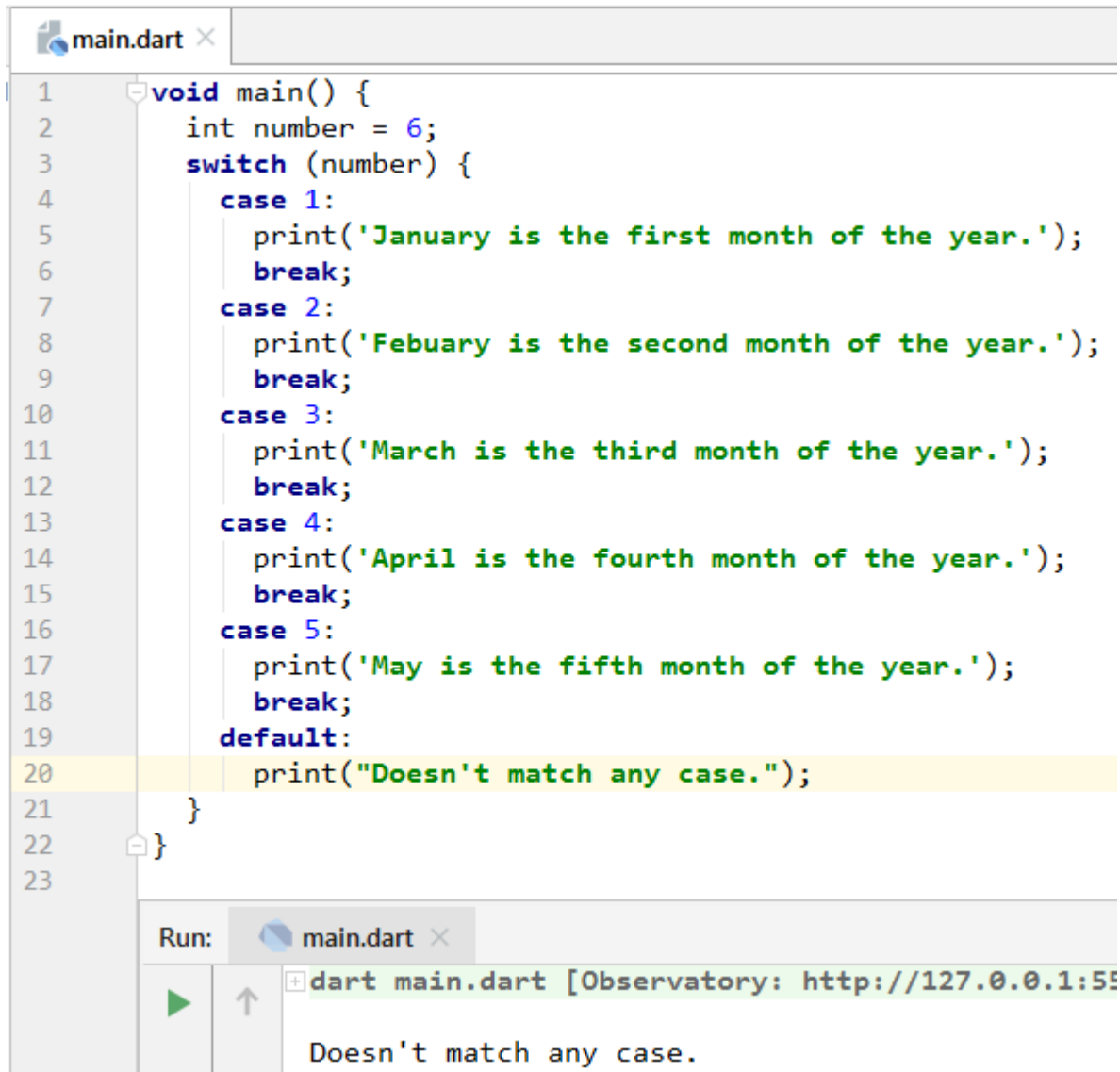


```
Run: main.dart x
+ dart main.dart [Observatory: http://127.0.0.1:54950/uBkH]
Today is Monday
```

Screenshot 6.5

In the program, the string variable *day* is assigned the string value “Monday”, then the variable is matched against several cases inside the switch statement, if it matches any of the cases, then the statement(s) tied to that case would be executed.

The **break** statement is used to specify the end of each case statement. The **default** statement is only executed when the value doesn't match any case. Take this program for example.



```
1 void main() {
2   int number = 6;
3   switch (number) {
4     case 1:
5       print('January is the first month of the year.');
```

```
6       break;
7     case 2:
8       print('Febuary is the second month of the year.');
```

```
9       break;
10    case 3:
11      print('March is the third month of the year.');
```

```
12      break;
13    case 4:
14      print('April is the fourth month of the year.');
```

```
15      break;
16    case 5:
17      print('May is the fifth month of the year.');
```

```
18      break;
19    default:
20      print("Doesn't match any case.");
21  }
22 }
23
```

The screenshot shows a Dart IDE with a file named `main.dart`. The code defines a `main` function that initializes `number` to 6 and uses a `switch` statement to print the month for values 1 through 5. Since `number` is 6, the `default` case is executed, printing "Doesn't match any case.". Below the code editor, a "Run" button is visible, and the output console shows the message "Doesn't match any case.".

Screenshot 6.6

This switch statement in this program, as opposed to the previous one, matches the value of an **int** variable against the defined case values. The allowed types for a switch statement are String, int and enumerated types (more on this later).

Notice that the default statement is executed, this is due to the fact that the value being switched on doesn't match any of the defined cases. Also, observe that no **break** statement is provided after the **default** statement, this is because the **default** statement

usually marks the end of the switch statement, meaning there can be no other **case** statement(s) coming after it.

Bear in mind that case statements are not limited to just **print** statements, they could be any kind of statement, or valid expressions, including function calls, etc.

## Conditional Expression

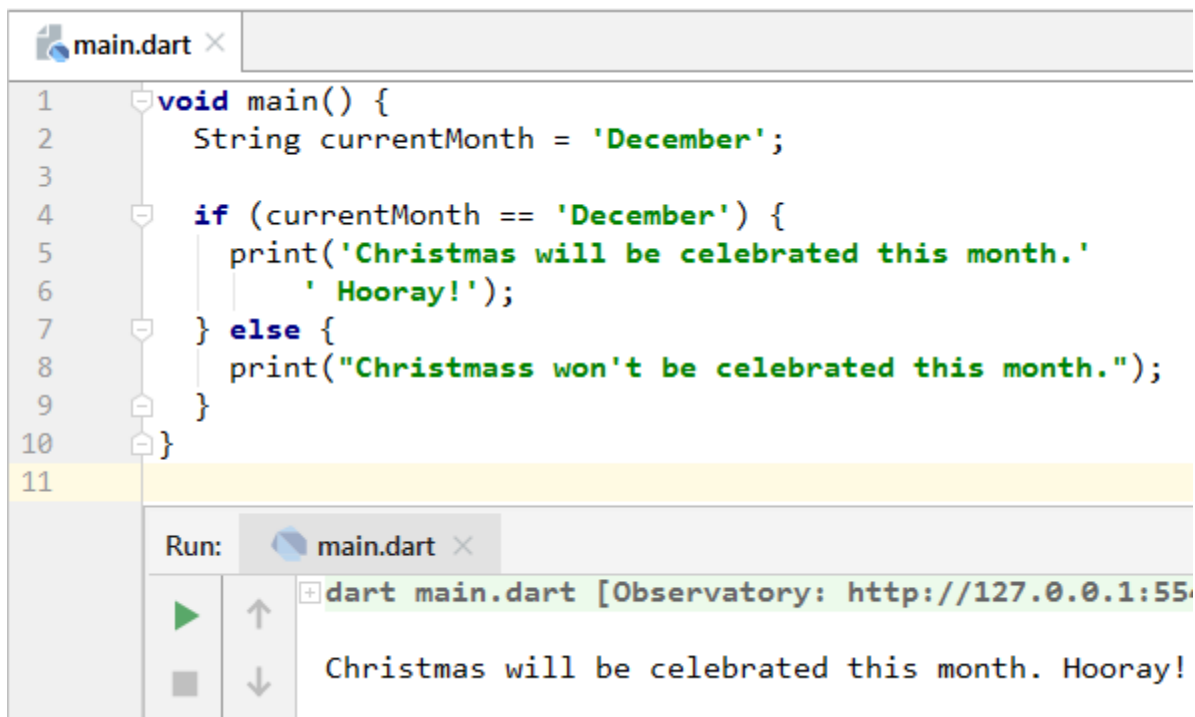
The most common conditional expression is the one that uses the **ternary operator**. It has the following syntax:

Condition ? expression one : expression two;

The **condition** is usually an expression that evaluates to a bool value of true/false.

**Expression one** represents the code statement that would get executed if the condition is true, while **expression two** is the code that would get executed if the condition is false.

This conditional expression can be used for situations that involve only two code paths, compared to the other selection statements that we've looked at, which can accommodate as many code paths as needed. Let's see an example program on how this conditional expression can be used. In order to do this, first we would write a program that uses an **if** statement for selecting between two code paths. Then we would write an equivalent program that uses this conditional expression to solve a similar problem.



The screenshot shows an IDE window titled 'main.dart'. The code is as follows:

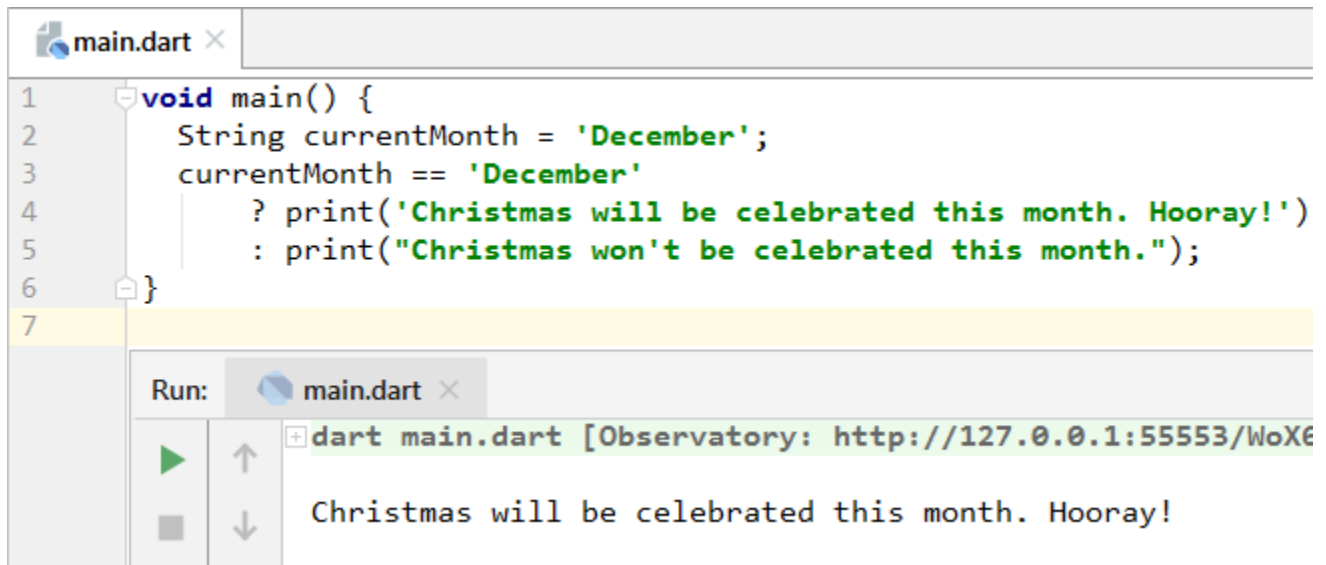
```
1 void main() {  
2   String currentMonth = 'December';  
3  
4   if (currentMonth == 'December') {  
5     print('Christmas will be celebrated this month.'  
6       ' Hooray!');  
7   } else {  
8     print("Christmass won't be celebrated this month.");  
9   }  
10 }  
11
```

Below the code editor, there is a 'Run:' button and a console window. The console output shows:

```
dart main.dart [Observatory: http://127.0.0.1:5555]  
Christmas will be celebrated this month. Hooray!
```

Screenshot 6.7

An equivalent of the above program written using the conditional expression.



The screenshot shows an IDE window titled 'main.dart'. The code is as follows:

```
1 void main() {  
2   String currentMonth = 'December';  
3   currentMonth == 'December'  
4     ? print('Christmas will be celebrated this month. Hooray!')  
5     : print("Christmas won't be celebrated this month.");  
6 }  
7
```

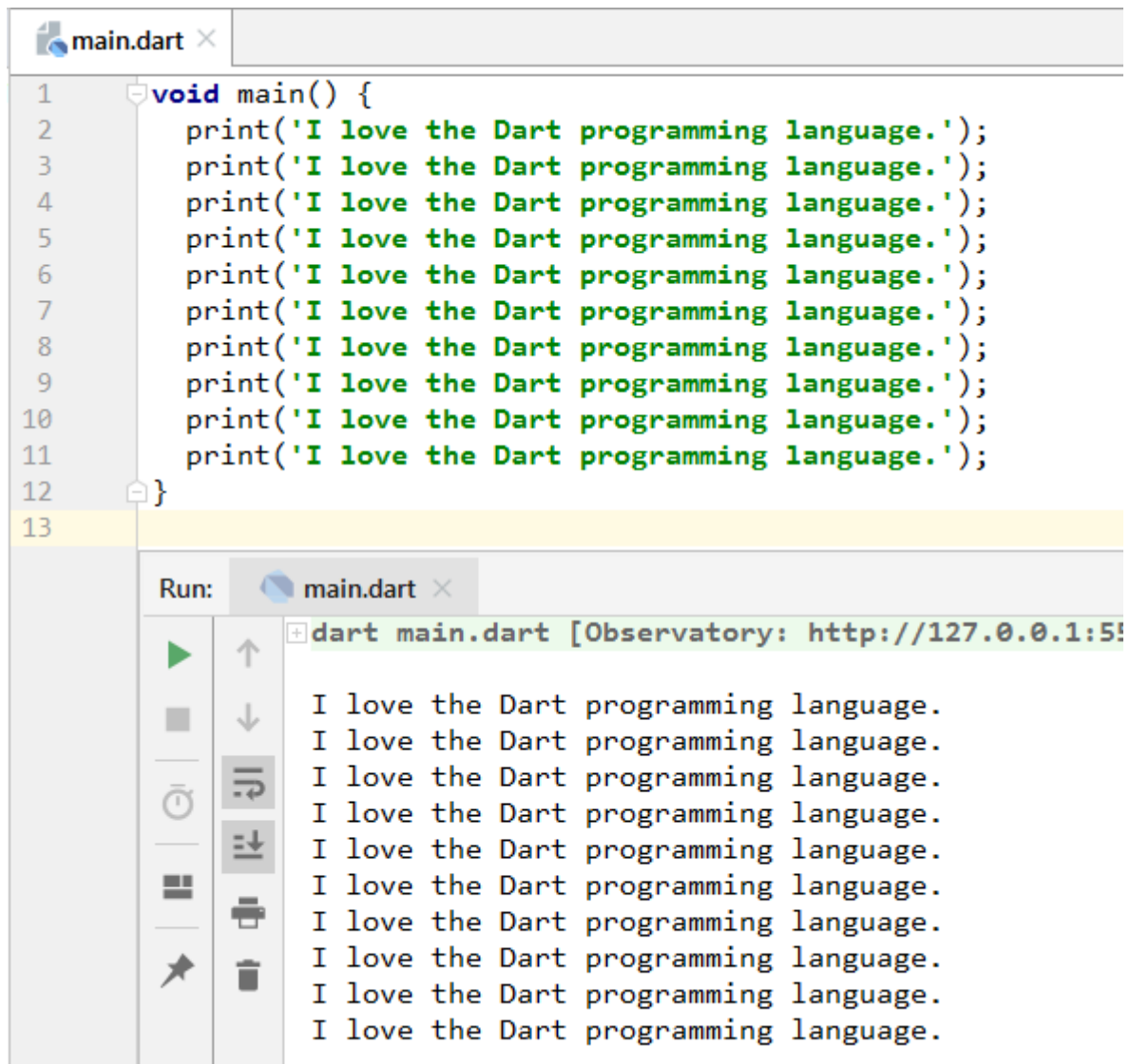
Below the code editor is a 'Run' panel. It shows the command 'dart main.dart' being executed. The output in the console is 'Christmas will be celebrated this month. Hooray!'.

Screenshot 6.8

Notice that the semicolon which terminates a statement is added after the second print statement, which signifies that using the ternary operator creates a single statement. One advantage of using this conditional expression is that it is more concise. Also, just as I pointed out for the **switch** statement, this conditional expression can be used to execute other statements other than **print** statements, like function calls, etc.

## Repetition Statements

Assuming you wanted to execute a program statement more than once, e.g. printing some text to the console for  $n$  number of times. To make that happen, one would have to write several print statements, which could get really tiring and seem unprofessional.



The screenshot shows an IDE window titled 'main.dart'. The code is as follows:

```
1 void main() {  
2   print('I love the Dart programming language.');
```

```
3   print('I love the Dart programming language.');
```

```
4   print('I love the Dart programming language.');
```

```
5   print('I love the Dart programming language.');
```

```
6   print('I love the Dart programming language.');
```

```
7   print('I love the Dart programming language.');
```

```
8   print('I love the Dart programming language.');
```

```
9   print('I love the Dart programming language.');
```

```
10  print('I love the Dart programming language.');
```

```
11  print('I love the Dart programming language.');
```

```
12 }
```

Below the code editor, the 'Run' tab is active, showing the command 'dart main.dart [Observatory: http://127.0.0.1:5984]'. The output of the program is displayed in the console:

```
I love the Dart programming language.  
I love the Dart programming language.  
I love the Dart programming language.  
I love the Dart programming language.  
I love the Dart programming language.  
I love the Dart programming language.  
I love the Dart programming language.  
I love the Dart programming language.  
I love the Dart programming language.  
I love the Dart programming language.
```

Screenshot 6.9

It is to avoid writing such code as this, that the repetition structures were made. Repetition statements are true to their name, they're used to repeatedly execute a statement or a particular block of code. The more commonly used ones are:

1. for loop
2. while loop
3. do while loop

## for loop

The for loop has the following syntax:

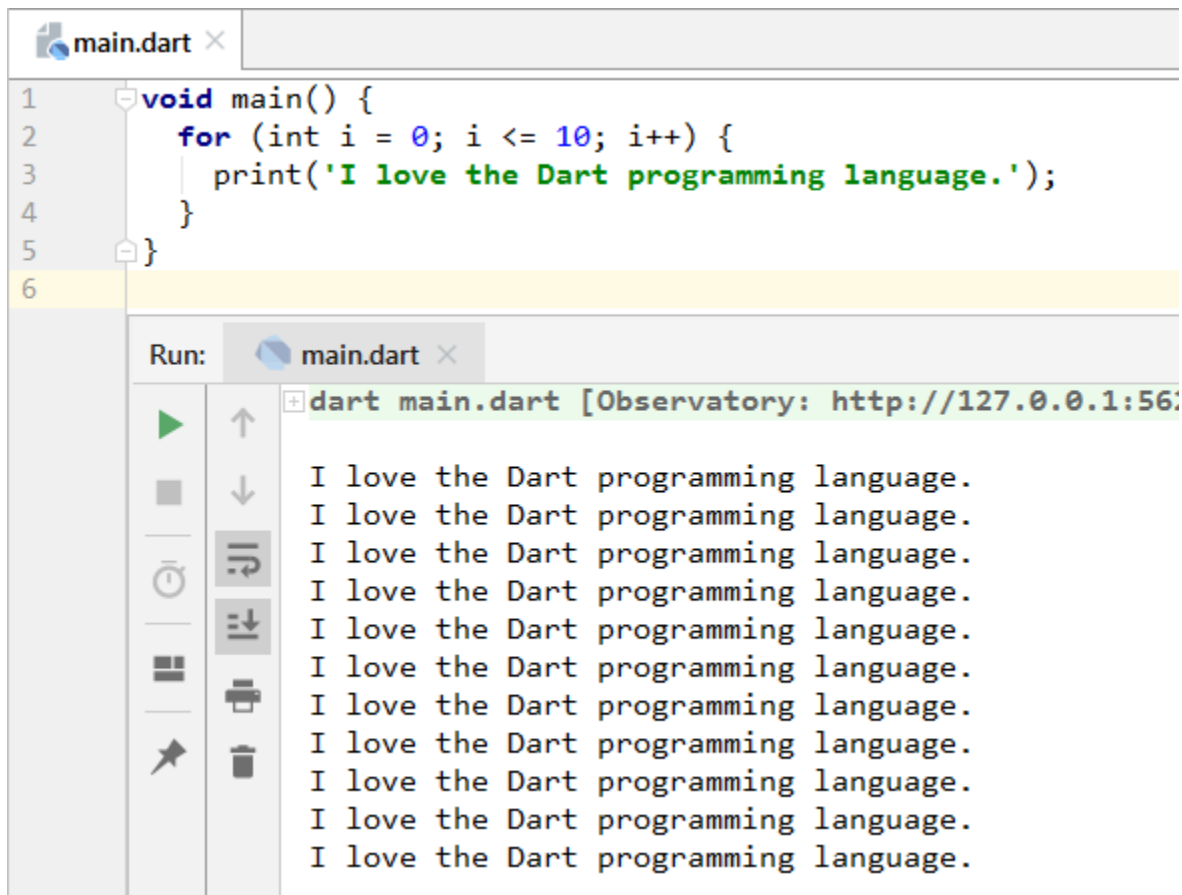
```
for (initialization; condition; increment/decrement) {  
    statement 1;  
    statement 2;  
    ...  
    Statement n;  
}
```

The **initialization** part is where the loop counter is initialized, the loop counter is a variable helps to keep track of how many times the loop has run, i.e. how many times the statement(s) has been executed.

The **condition** part is where the loop condition is specified, the loop will continue to run, for as long as the condition remains true. The loop condition is usually any valid bool expression that evaluates to true or false.

Lastly is the **increment/decrement** part. This part is where the loop counter gets increased or decreased, so as to keep track of how many times the loop has run and eventually cause the loop to end, when the condition becomes false. With all of that explained, let's rewrite the program above, in a better and more professional way, using the for loop.





The screenshot shows an IDE window with a file named `main.dart`. The code is as follows:

```
1 void main() {  
2   for (int i = 0; i <= 10; i++) {  
3     print('I love the Dart programming language.');4   }  
5 }  
6
```

Below the code editor is a 'Run' panel. It shows the command `dart main.dart` and the Observatory URL `http://127.0.0.1:56...`. The output of the program is displayed in the console, showing the message 'I love the Dart programming language.' repeated ten times.

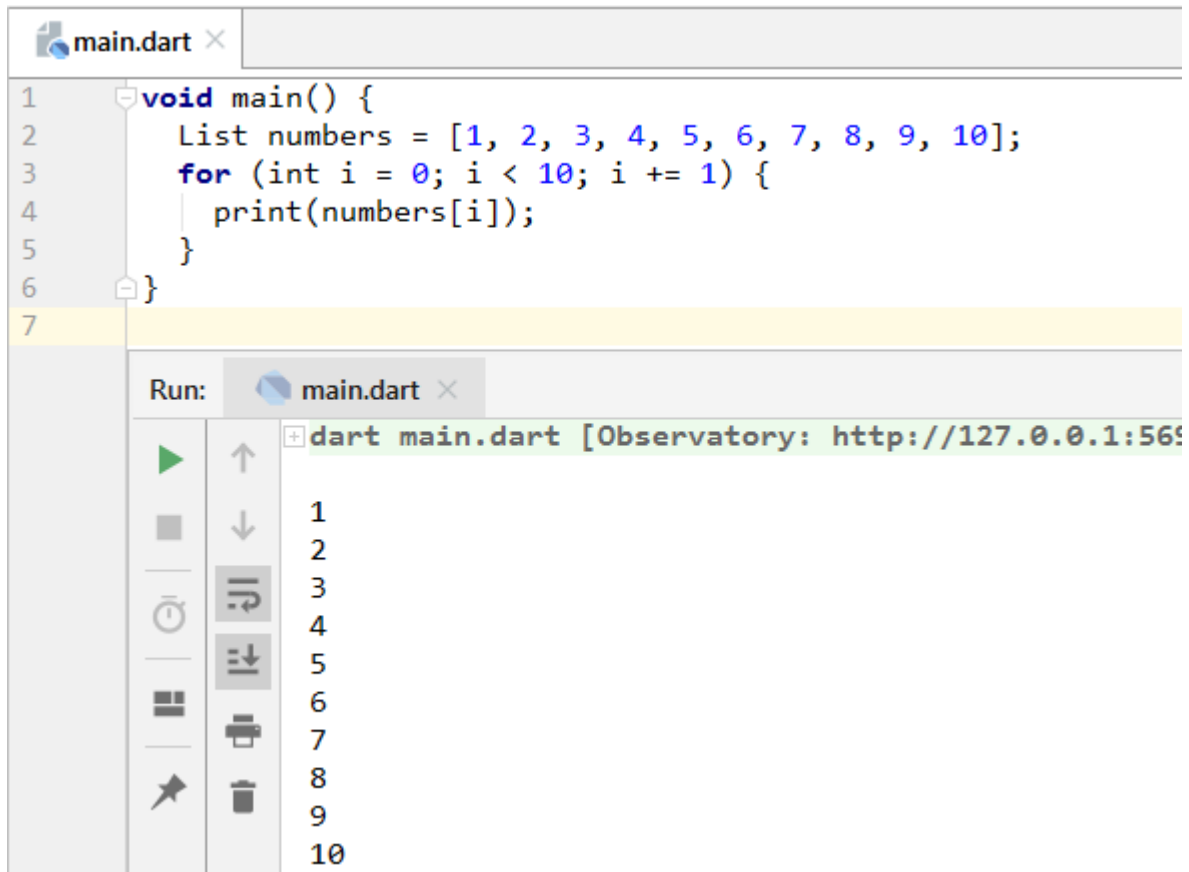
Screenshot 6.10

Just like magic, a single print statement is able to print text to the console ten (10) times. Well, there isn't any abracadabra going on here, it's all being done with the aid of the for loop. That's how powerful loops are and how much they can help us achieve. Let's dissect the program, so we get the full picture of how the program uses the for loop to perform the task.

As was shown in the for loop syntax above, the first part is the initialization part, in this case, the `int i = 1`. This is the same variable declaration and initialization that we've done thus far. We're simply telling Dart that the variable `i` should be used as the loop's counter (to keep track of how many times the loop has run). Next is the condition part. In our program, the condition is `i <= 10`. What that means is that, the loop should run for as long as the variable `i` is less than or equal to 10, meaning the loop should stop running when `i` becomes greater than 10. When that happens, the expression would evaluate to false, and the loop would cease to run. Lastly, is the increment/decrement part. Although, in our program, we're doing an increment of the value stored in `i`, we'll see how to do a decrement soon. Whenever the loop runs, the value stored in `i` is increased by one (1). This is done with the `i++`. `i++` translates to `i = i + 1`. So the `i++` in the code above could be swapped with `i = i + 1`, and it would produce

the same output. One is only a shorter form of the other. The `i = i + 1` can also be expressed in another form, `i += 1`, just as you learnt in chapter 4, where we looked at operators.

For loops are commonly used when dealing with a list of items. Our next example showcases that.



The screenshot shows an IDE window with a file named `main.dart`. The code is as follows:

```
1 void main() {  
2   List numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];  
3   for (int i = 0; i < 10; i += 1) {  
4     print(numbers[i]);  
5   }  
6 }  
7
```

Below the code editor, the 'Run' panel is active, showing the execution of `main.dart`. The output in the console is:

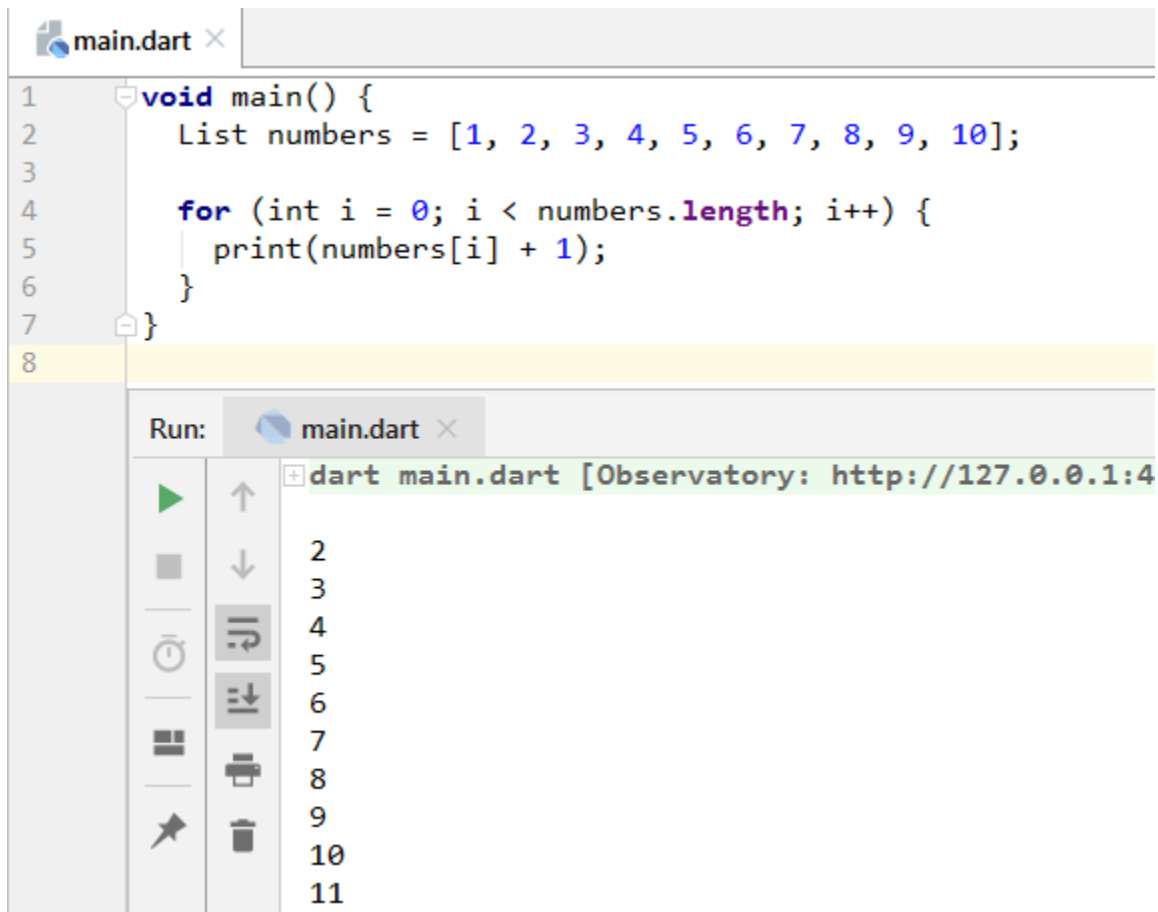
```
dart main.dart [Observatory: http://127.0.0.1:569]  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

Screenshot 6.11

Remember what I told you about lists using zero-based index in chapter 2. That idea comes to play here. The program above uses the loop counter as the index to individually retrieve items from the list and then prints them to the console. Based on the fact that the loop counter is increased by 1, each time the loop runs, it keeps increasing, until it becomes greater than 10, at which point the condition of the loop becomes false, causing the loop to terminate.

The index of the items in the *numbers* list begins at zero (0) and ends at nine (9), which is why the condition is `i < 10`. An alternative is to check for `i <= 9`, that would also work fine for the number of items in the *numbers* list. You should know that an attempt to access an item, using an index that is not within the range of the current list's indices would result in an error, so always ensure that you only access items in a list using valid indices.

An obvious limitation to the way we structure the loop condition in the program above is that we're hardcoding the number of items that are present in the list. While this may not be a problem in a list containing such few items, it could be a serious problem when we're faced with a list of which we do not know its exact length. It is for this reason that Dart provides the **length** property for a list. The **length** property always contains the number of items in a list. It is accessed with the **.length** syntax. Simply attach **.length** to a list, to determine the number of items in the list.



The screenshot shows an IDE window with a file named `main.dart`. The code defines a `main` function that creates a list of numbers from 1 to 10 and iterates over it using a `for` loop. The loop condition uses `numbers.length` to determine the number of items. Inside the loop, each number is incremented by 1 before being printed. Below the code editor, the 'Run' button is clicked, and the output console shows the results: 2, 3, 4, 5, 6, 7, 8, 9, 10, and 11. The console also displays the URL of the Dart Observatory.

```
1 void main() {  
2   List numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];  
3  
4   for (int i = 0; i < numbers.length; i++) {  
5     print(numbers[i] + 1);  
6   }  
7 }  
8
```

Run: main.dart x

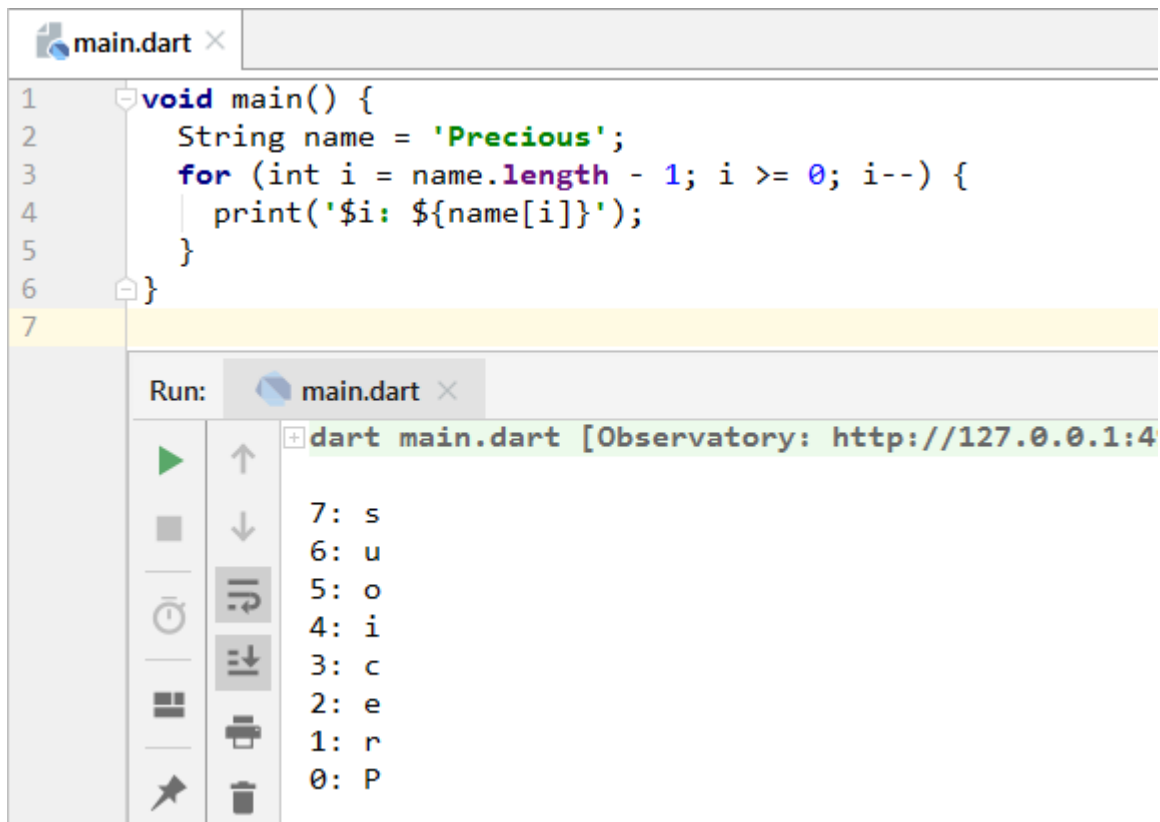
dart main.dart [Observatory: http://127.0.0.1:4

2  
3  
4  
5  
6  
7  
8  
9  
10  
11

Screenshot 6.12

The program above adds 2 to each number in the list, just before it gets printed out. It uses the **length** property of the list, to determine the number of items the list contains. This is a more convenient way of writing for loops that go through the items in a list. We shall adopt this method of writing for loops and use it hence forth.

A String as defined in chapter two, is an array of characters. An array is another name for a list in Dart. This means that we can write a for loop that loops through all the characters in a String.



```
main.dart x
1 void main() {
2   String name = 'Precious';
3   for (int i = name.length - 1; i >= 0; i--) {
4     print('$i: ${name[i]}');
5   }
6 }
7

Run: main.dart x
+ dart main.dart [Observatory: http://127.0.0.1:4
7: s
6: u
5: o
4: i
3: c
2: e
1: r
0: P
```

Screenshot 6.13

From the above program, it is clear that a string value is itself an array (list of characters). We could choose to just retrieve the first or any character in a String. For the string above, we can get any character in it using **name[index]**. One other thing to observe in the code, is how the loop counter is initialized to the number of characters in *name*, using the length property. Subtracting one (1) from it ensures that the value of *i*, isn't greater than the index of the last character in *name*. Also, notice that we're decreasing the value of *i*, as opposed to what was done in the previous loop programs, where we increased it. Due to this decrement, the string value 'Precious', is outputted in the reverse order, because the loop prints out the characters, starting from the last index.

## while loop

A while loop, just like a for loop is also used for iteration, but for a certain kind of iteration. while loops are best for iterating through items whose length is unknown. Assuming we had a directory that contains a lot of files, or a book whose text characters needed to be counted, then a while loop would be a good choice for such operations. However, let's look at a simple program on how to use the while loop.

```
1 void main() {
2   List ingredients = [
3     'Maggi',
4     'Tomato',
5     'Pepper',
6     'Onions',
7     'Curry',
8   ];
9
10  int counter = 0;
11  while (counter < ingredients.length) {
12    print(ingredients[counter]);
13
14    // Increment counter, so as to prevent the loop from
15    // running infinitely.
16    counter++;
17  }
18 }
19
```

Run: main.dart x

Maggi  
Tomato  
Pepper  
Onions  
Curry

Screenshot 6.14

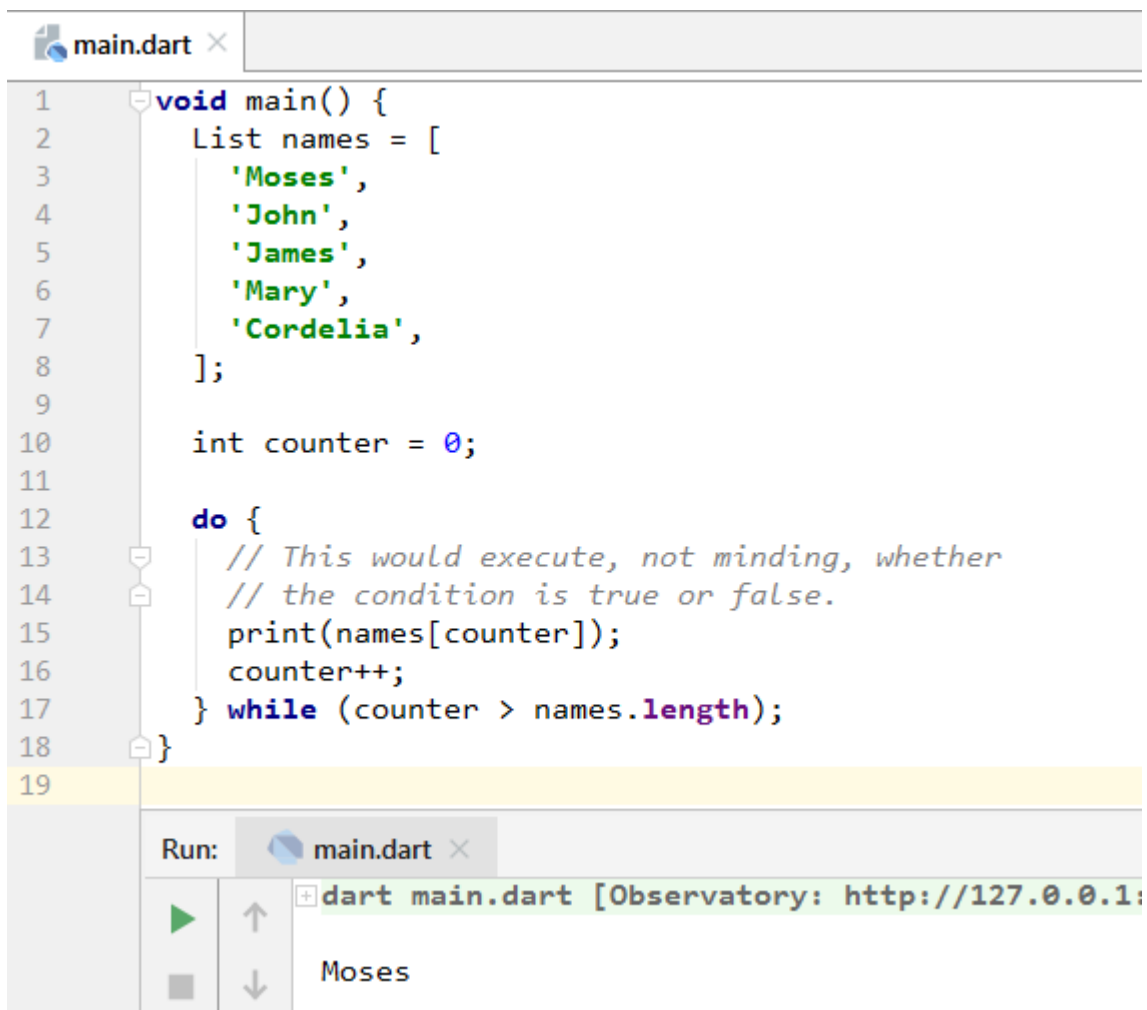
The while loop's structure is a bit different from that of the for loop. However, the comprising parts are the same. Observe that the counter variable for the loop is defined outside of its body, and it is incremented from within the loop's body.

In the program above, there's a comment stating that incrementing the loop counter must be done, so as to prevent the loop from running indefinitely. An indefinite loop is a loop that runs without end. To intentionally create an indefinite loop, try removing the **counter++** inside the loop's body. Doing that would cause the loop to run forever, printing just **maggi**, which is the first element in the list. This is because the loop counter was initialized to zero (0) and it never increases, so it only accesses the first element in the list, using the zero (0) index. In the IntelliJ IDEA code editor, you can stop a running program by clicking on the red stop icon, on the same area as the play icon which runs a program.

As a warning, endeavor to always to structure your loop condition, that it always becomes false at some point, so as to avoid creating an infinite loop.

## do while loop

The do while loop is similar to the while loop, except for the fact that the statement(s) to be executed are executed at least once, not minding if the loop condition is true or false. This is because the body of the loop is encountered before the condition. do while loops are useful, if you need to ensure the execution of some code, before the loop starts running, i.e. if the loop will eventually run or not. Let's look at a program that describes how the do while loop works.



```
1 void main() {
2   List names = [
3     'Moses',
4     'John',
5     'James',
6     'Mary',
7     'Cordelia',
8   ];
9
10  int counter = 0;
11
12  do {
13    // This would execute, not minding, whether
14    // the condition is true or false.
15    print(names[counter]);
16    counter++;
17  } while (counter > names.length);
18 }
19
```

Run: main.dart x

dart main.dart [Observatory: http://127.0.0.1:8080]

Moses

Screenshot 6.15

Even though the condition for the loop is false, the statement inside the body of the loop, is executed at least once. The loop doesn't perform any iterations.

As an exercise, refactor the condition of the loop, so it performs its iterations and stops when all the items in the list have been printed.

## **Jump Statements**

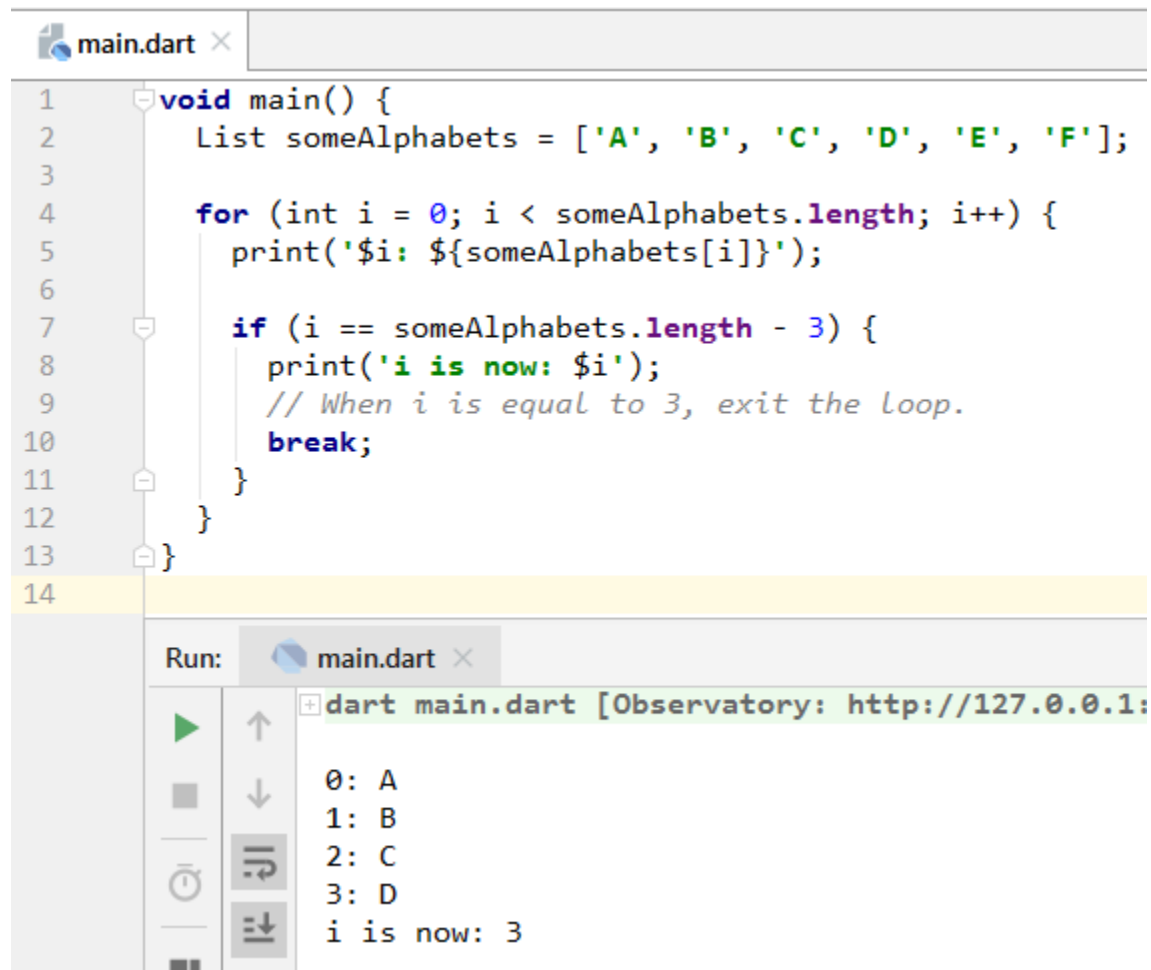
The jump statements are primarily used in a loop's body, they're used to conditionally alter the flow of the loop, or to halt its execution. They include:

1. `break`
2. `continue`

### **`break`**

You've already seen how to use the `break` statement when we looked at the `switch` statement in chapter 5. The effect of the `break` statement when used in a loop is the same, i.e. causing it to end abruptly.

Screenshot 6.16



The screenshot shows an IDE window titled 'main.dart'. The code defines a `main` function that iterates over a list of alphabets. It prints each letter and then, when the index reaches 3, it prints 'i is now: 3' and breaks the loop. Below the code, the 'Run' panel shows the execution output: '0: A', '1: B', '2: C', '3: D', and 'i is now: 3'.

```

1 void main() {
2   List someAlphabets = ['A', 'B', 'C', 'D', 'E', 'F'];
3
4   for (int i = 0; i < someAlphabets.length; i++) {
5     print('$i: ${someAlphabets[i]}');
6
7     if (i == someAlphabets.length - 3) {
8       print('i is now: $i');
9       // When i is equal to 3, exit the loop.
10      break;
11    }
12  }
13 }
14
Run: main.dart x
+ dart main.dart [Observatory: http://127.0.0.1:
0: A
1: B
2: C
3: D
i is now: 3

```

The loop runs until the loop counter (*i*) is equal to 3, at which point it stops running. With the **if** statement, we ensure that the **break** statement isn't encountered in the loop's other iterations, rather, at its 4<sup>th</sup> iteration, when the counter is equal to 3. As a result, only the last two items ('E' and 'F') are not printed.

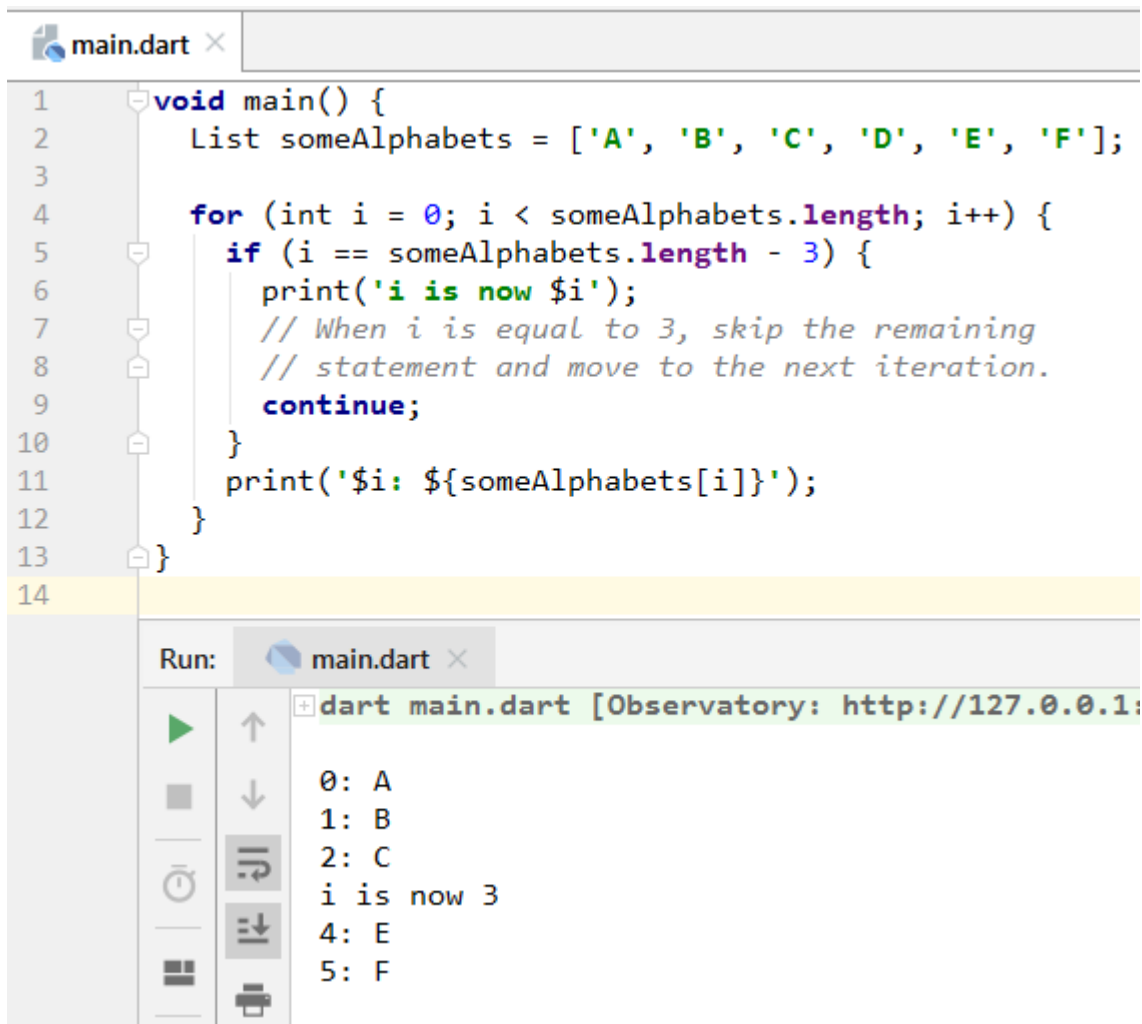
## continue

The **continue** statement is true to its name. When it's used, the loop skips its current iteration and continues to the next iteration. If some statements occur before and after the **continue** statement in a loop, only the statements that occur before the continue statement would be executed, while the ones that occur after the continue statement would be skipped.

Let's rewrite the previous example to use a continue statement.



Screenshot 6.17



The screenshot shows an IDE window with a file named `main.dart`. The code defines a `main` function that iterates over a list of alphabets. A `continue` statement is used to skip the iteration for index 3. The output console shows the execution results, including the skipped iteration.

```
1 void main() {  
2   List someAlphabets = ['A', 'B', 'C', 'D', 'E', 'F'];  
3  
4   for (int i = 0; i < someAlphabets.length; i++) {  
5     if (i == someAlphabets.length - 3) {  
6       print('i is now $i');  
7       // When i is equal to 3, skip the remaining  
8       // statement and move to the next iteration.  
9       continue;  
10    }  
11    print('$i: ${someAlphabets[i]}');  
12  }  
13 }  
14
```

Run: `main.dart`

`dart main.dart [Observatory: http://127.0.0.1:8181]`

```
0: A  
1: B  
2: C  
i is now 3  
4: E  
5: F
```

The loop experiences a normal execution, until its counter (*i*) becomes 3, at which point the condition of the `if` statement becomes true and the loop is forced to print out *i is now 4*, then it skips the remaining statement, which is printing out the item at that particular index, in this case 'D'.

## CHAPTER 7

# OBJECT ORIENTED PROGRAMMING

Just like most modern programming languages, Dart is an object oriented programming language (OOP).

In the world of object oriented programming, everything is seen as an object. A Car is an object, a House is an object, an animal is an object, and even a human is an object. With object oriented programming, we're able to model real-life objects and create an interaction with their properties and actions. When I say **properties**, I mean the different attributes that an object has, e.g. A Dog has a tail, eyes, head, etc. While **actions** refer to what an object can do, e.g. a Dog can bark, run, swim, etc.

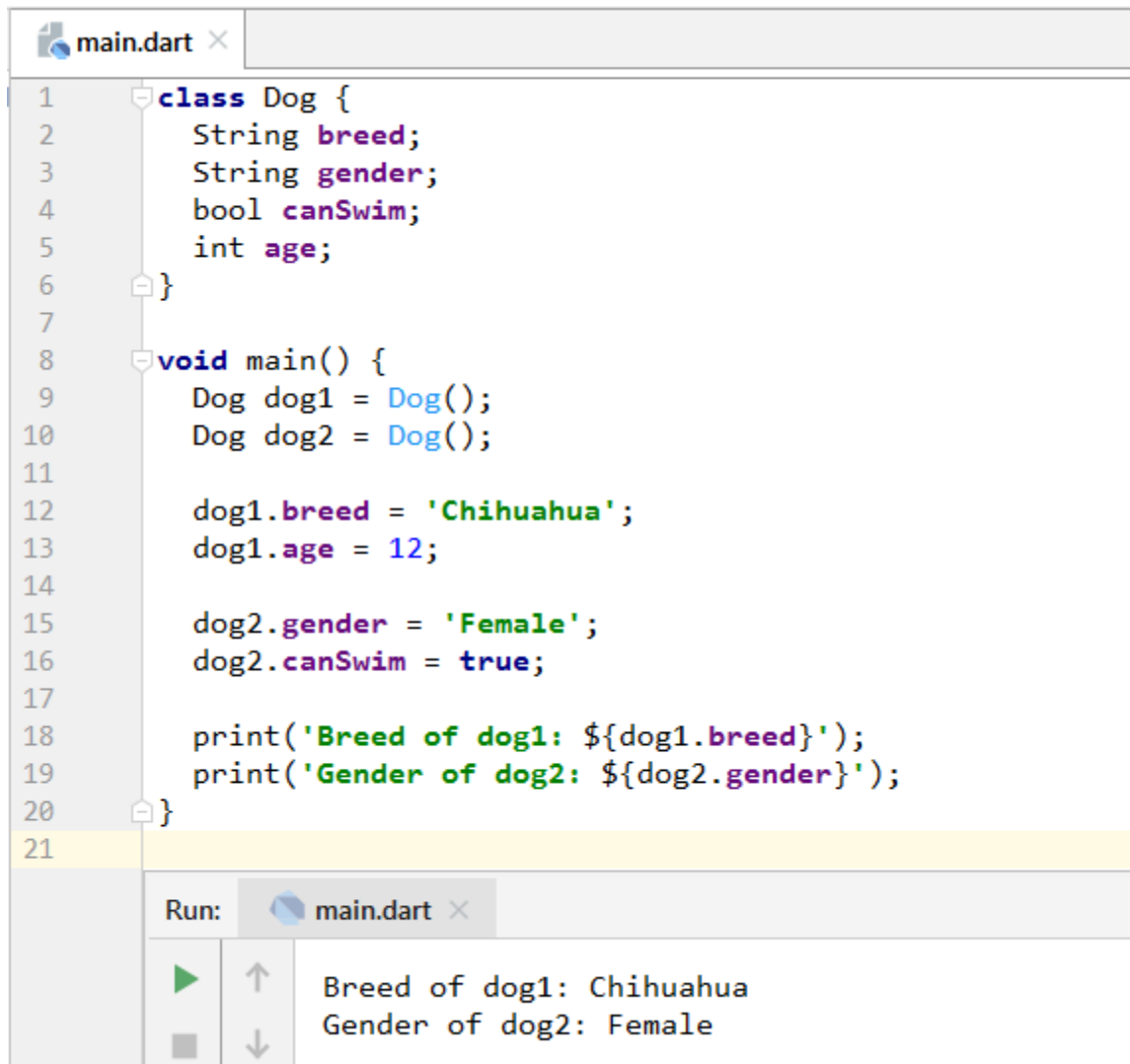
Object oriented programming revolves around the concepts of classes and objects. It is important to know and understand these concepts.

### What is a class?

A class can be seen as a blueprint from which objects can be created. A good analogy is a blueprint for a house, from which several houses can be made, all having the same properties. A class encapsulates the attributes and actions of an object and is seen as the type of the object. Defining a class is like creating your own custom type. Everything in a class is seen as the members of the class, which includes the properties and actions.

### What is an object?

An object is what is created from a class, or using a more technical term, an object is an **instance** of a class. For example, from a Dog class, several Dog objects can be created, all of which would have the same attributes and can perform the same actions. The example code below shows how to define a class and how to create an object from it.



```
1 class Dog {
2   String breed;
3   String gender;
4   bool canSwim;
5   int age;
6 }
7
8 void main() {
9   Dog dog1 = Dog();
10  Dog dog2 = Dog();
11
12  dog1.breed = 'Chihuahua';
13  dog1.age = 12;
14
15  dog2.gender = 'Female';
16  dog2.canSwim = true;
17
18  print('Breed of dog1: ${dog1.breed}');
19  print('Gender of dog2: ${dog2.gender}');
20 }
21
```

Run: main.dart ×

Breed of dog1: Chihuahua  
Gender of dog2: Female

Screenshot 7.1

As earlier explained, a class is an encapsulation of an object's attributes and actions, although the class above contains only attributes, we shall see how to include actions in a moment.

A class is defined using the **class** keyword, followed by the name of the class, then an opening and closing brace, which marks the body of the class. Inside the body of the class is where the attributes and actions are specified. The name of a class could be anything the programmer chooses. Class names also adhere to the same rules as those for choosing a variable name, but they must begin with a capital letter, just like **Dog**.

The attributes in a class are commonly referred to as **fields** or **instance variables**, because they're the variables that are tied to each instance (object) that is created from a class. **All objects created from a class, have a copy of the instance variable(s) defined in the class.** Remember that defining a class is like creating your

own custom type. So, in the program above, Dog is the type of all objects that are created from it.

In the main function, we have two instances (objects) of the Dog class, *dog1* and *dog2*. The method of creating objects from a class is similar to how we've defined normal variables in the past. First, specify the type of the object, in this case Dog, then the name of the object, in this case *dog1* or *dog2*, then the assignment operator (=), lastly is the value that is being assigned to the object. This is the general syntax for creating an object from a class.

Just below the creation of the Dog objects, on lines 14, 15, 17 and 18, we've accessed the properties of the *dog1* and *dog2* objects. To access the property of an object, append a dot (.) to the name of the object. The dot (.) is how we access any property of the object and is referred to as the **member access** operator. Here, we're assigning values to the properties (which are basically variables in defined in the Dog class). We assigned a String and an int value to the *breed* and *age* properties of the *dog1* object respectively, while we assigned a String and bool value to the *gender* and *canSwim* properties of the *dog2* object.

Let's see how to include actions for objects in a class.

```
main.dart x
1 class Dog {
2   // Attributes.
3   String breed;
4   String gender;
5   bool canSwim;
6   int age;
7
8   // Actions
9   void bark() {
10    print('This dog is barking');
11  }
12
13  void swim() {
14    print('This dog is swimming');
15  }
16 }
17
18 void main() {
19   // dog1 and dog2 are instances of the Dog class.
20   Dog dog1 = Dog();
21   Dog dog2 = Dog();
22
23   dog1.breed = 'Chihuahua';
24   dog1.age = 12;
25
26   dog2.gender = 'Female';
27   dog2.canSwim = true;
28
29   print('Breed of dog1: ${dog1.breed}');
30   print('Gender of dog2: ${dog2.gender}');
31
32   dog1.bark();
33   dog2.swim();
34 }
35
36 Run: main.dart x
37
38 dart main.dart [Observatory: http://127.0.0.1:
39
40 Breed of dog1: Chihuahua
41 Gender of dog2: Female
42 This dog is barking
43 This dog is swimming
```

## Screenshot 7.2

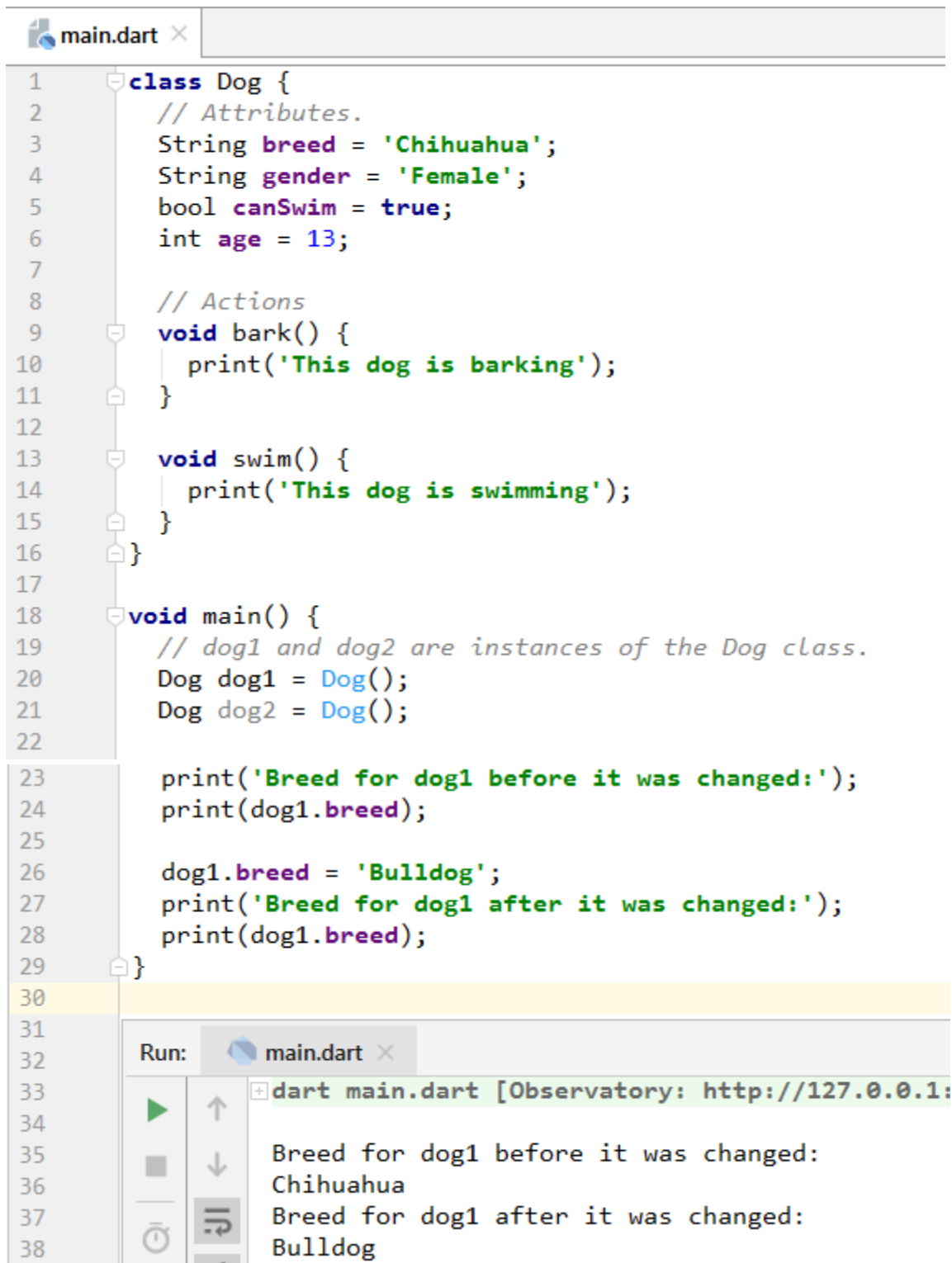
In the previous program, we saw how to define properties for objects in a class and how to access those properties. Here, we've included actions. Actions are basically functions, just as we have the *bark* and *swim* functions defined in the Dog class above. Although, when a function is defined in a class, it is called a **method**, while functions that exist outside of a class are called functions.

Remember that everything in a class is seen as a member of the class, with that in mind, we can refer to instance variables as member variables while actions (methods) can be referred to as member methods or instance methods. These are just names that are given to these different class entities, so as to properly distinguish between them.

In the program above, we've defined two instance methods, *bark* and *swim*. These two methods are accessed or called using the *dog1* and *dog2* objects in the main function i.e. on lines 32 and 33. Recall that each object has its own copy of the instance variables and methods in a class. So *dog1* and *dog2* can access their individual copy of these methods, i.e. *bark* and *swim*. These two methods do nothing, other than printing out some text, "This dog is barking" and "This dog is swimming". A method in a class, can be defined to do anything.

## Default Values For Instance Variables

The instance variables in a class can be initialized (i.e. given default values) before objects are created. When that is done, all the objects created from the class would automatically have their instance variables set to those values. Here's an example that shows how that can be done.



```
1 class Dog {
2   // Attributes.
3   String breed = 'Chihuahua';
4   String gender = 'Female';
5   bool canSwim = true;
6   int age = 13;
7
8   // Actions
9   void bark() {
10    print('This dog is barking');
11  }
12
13  void swim() {
14    print('This dog is swimming');
15  }
16 }
17
18 void main() {
19   // dog1 and dog2 are instances of the Dog class.
20   Dog dog1 = Dog();
21   Dog dog2 = Dog();
22
23   print('Breed for dog1 before it was changed:');
24   print(dog1.breed);
25
26   dog1.breed = 'Bulldog';
27   print('Breed for dog1 after it was changed:');
28   print(dog1.breed);
29 }
30
31 Run: main.dart x
32
33 + dart main.dart [Observatory: http://127.0.0.1:
34
35 Breed for dog1 before it was changed:
36 Chihuahua
37 Breed for dog1 after it was changed:
38 Bulldog
```

Screenshot 7.3

In the Dog class of the program above, initial or default values have been assigned to the instance variables, which causes any object created from the class to have its

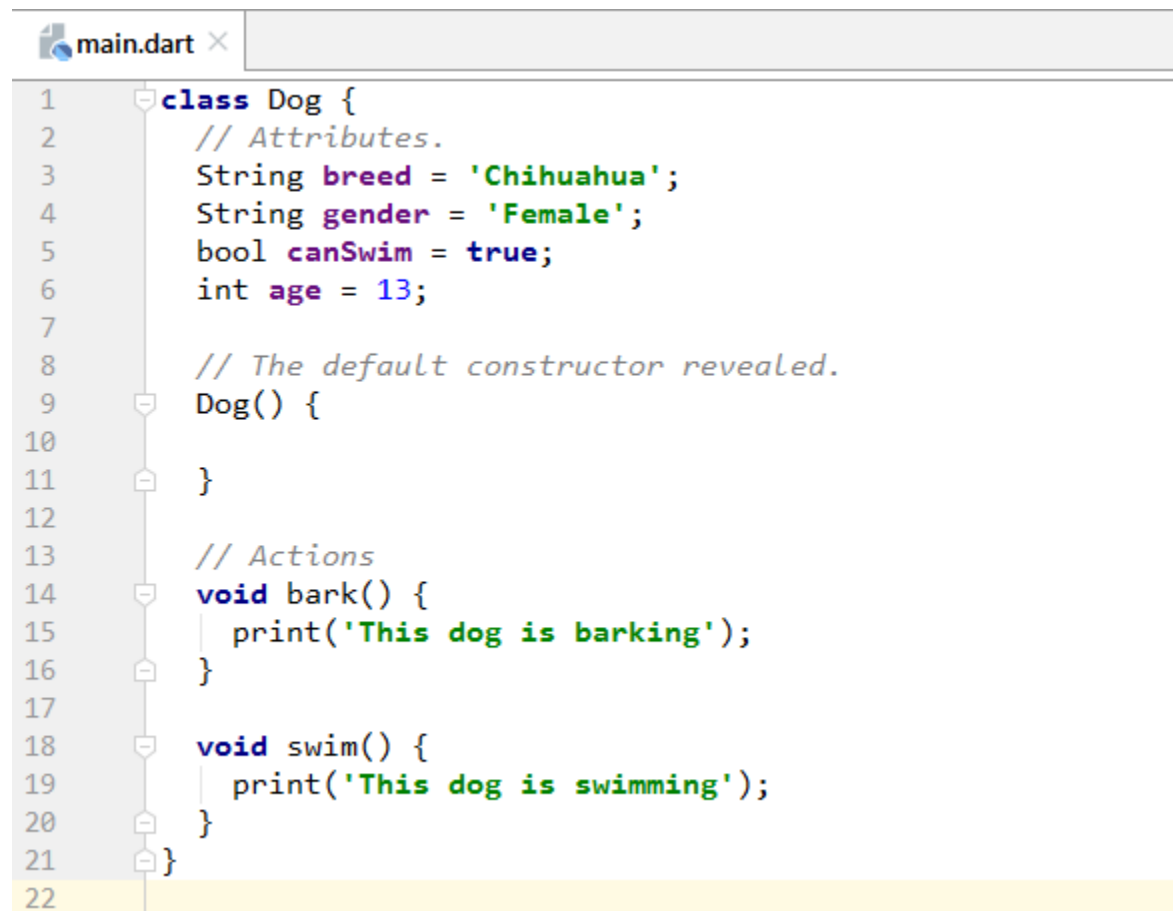
variables set to those values. We've put this to test by first printing out the breed property (variable) of *dog1*, then later changing it to another value. This would be the same for all the other instance variables and you're free to test it out, for the *dog1* and *dog2* objects.

## Constructor

We've seen how to access the instance variables of an object and then assign values to them outside of a class, also we've seen how to assign values (default values) to instance variable in a class. Next, let's look at how to assign values to instance variables when an object is being defined or created. To do this, we require a **constructor**.

A constructor is a special function that exists in a class and is used in creating or constructing objects from that class. Dart calls the constructor each time you tell it to create an object of any class. A constructor in a class usually takes the same name as the class.

So far we've only constructed objects using the **default constructor**, which Dart creates internally when we define a class. Using the Dog class for example, the default constructor has the following syntax:



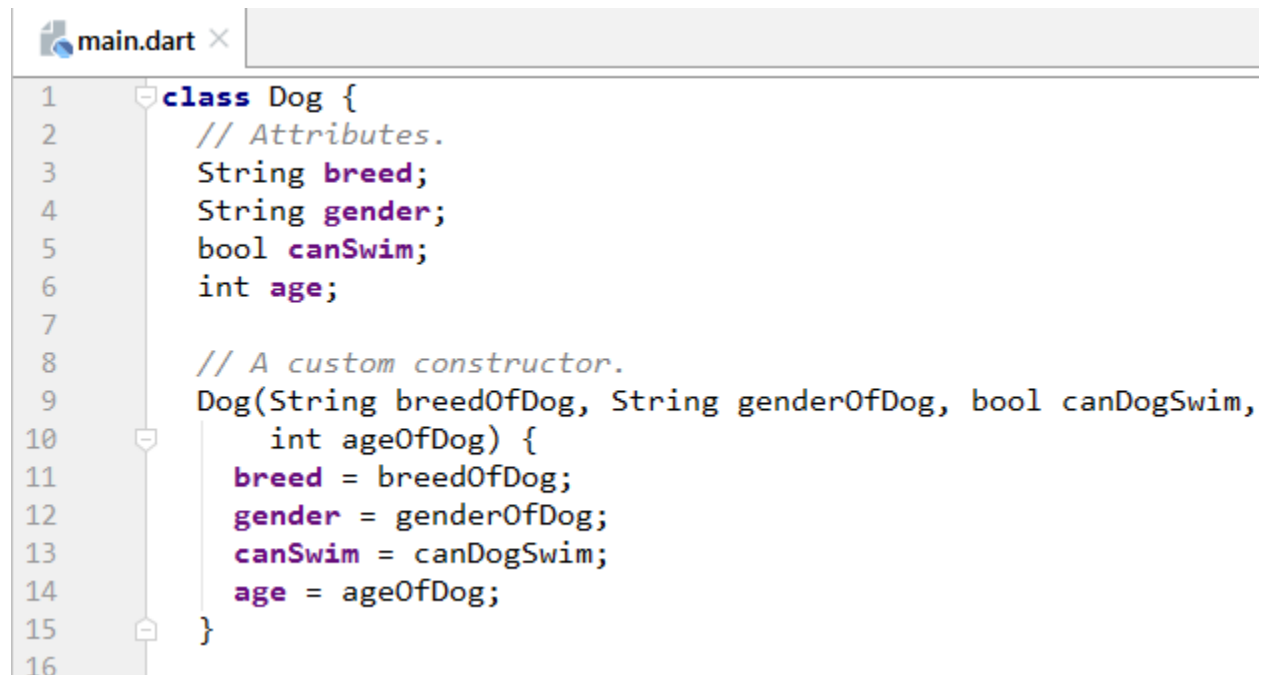
```
main.dart x
1  class Dog {
2      // Attributes.
3      String breed = 'Chihuahua';
4      String gender = 'Female';
5      bool canSwim = true;
6      int age = 13;
7
8      // The default constructor revealed.
9      Dog() {
10
11     }
12
13     // Actions
14     void bark() {
15         print('This dog is barking');
16     }
17
18     void swim() {
19         print('This dog is swimming');
20     }
21 }
22
```



## Screenshot 7.4

As earlier mentioned, a constructor is similar to a function, except that it takes the name of the class.

The major reason why you should define your own constructor in a class, is so that you're able to pass initial values to the instance variables defined in the class. That way, you're able to provide unique values for each object that is created using the constructor you defined. As you would see, passing values to a constructor, is similar to how values are passed to a function. Let's see an example on how you can define your own constructor.

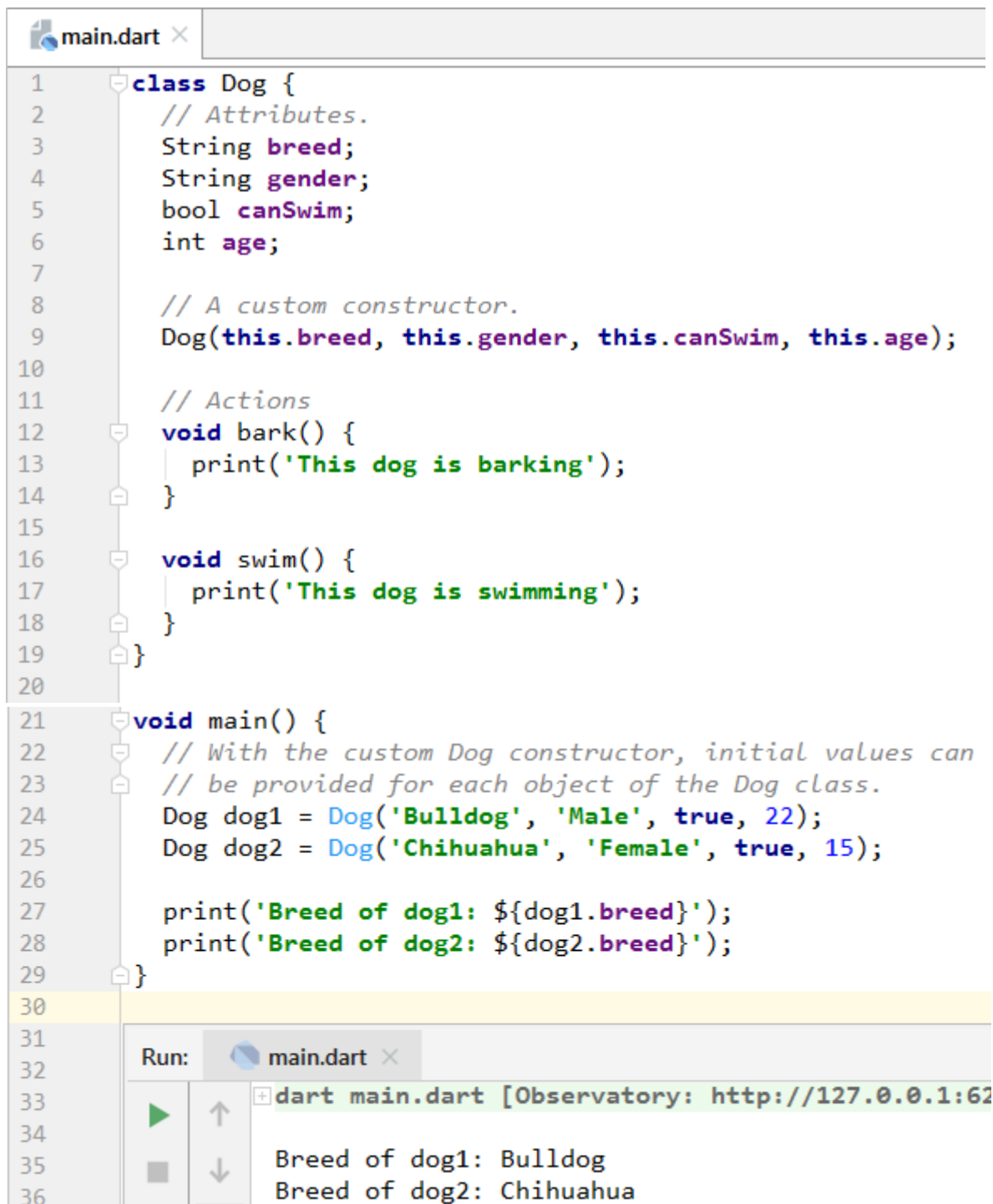


```
main.dart x
1  class Dog {
2      // Attributes.
3      String breed;
4      String gender;
5      bool canSwim;
6      int age;
7
8      // A custom constructor.
9      Dog(String breedOfDog, String genderOfDog, bool canDogSwim,
10         int ageOfDog) {
11         breed = breedOfDog;
12         gender = genderOfDog;
13         canSwim = canDogSwim;
14         age = ageOfDog;
15     }
16 }
```

```
17 // Actions
18 void bark() {
19     print('This dog is barking');
20 }
21
22 void swim() {
23     print('This dog is swimming');
24 }
25 }
26
27 void main() {
28     // With the custom Dog constructor, initial values can
29     // be provided for each object of the Dog class.
30     Dog dog1 = Dog('Bulldog', 'Male', true, 22);
31     Dog dog2 = Dog('Chihuahua', 'Female', true, 15);
32
33     print('Breed of dog1: ${dog1.breed}');
34     print('Breed of dog2: ${dog2.breed}');
35 }
36
37 Run: main.dart x
38
39 + dart main.dart [Observatory: http://127.0.0.1:62
40
41 Breed of dog1: Bulldog
42 Breed of dog2: Chihuahua
```

Screenshot 7.5

The Dog class now has a constructor that allows the provision of values for instance variables when objects are being created. Constructors are called just like you would functions. When you call a constructor that defines parameters like the constructor in the Dog class above, you're required to provide arguments. Although, this is just one of the ways a constructor can be defined. Let me show you another way for defining a constructor, this new way is commonly used by most people, because it is concise and more convenient.



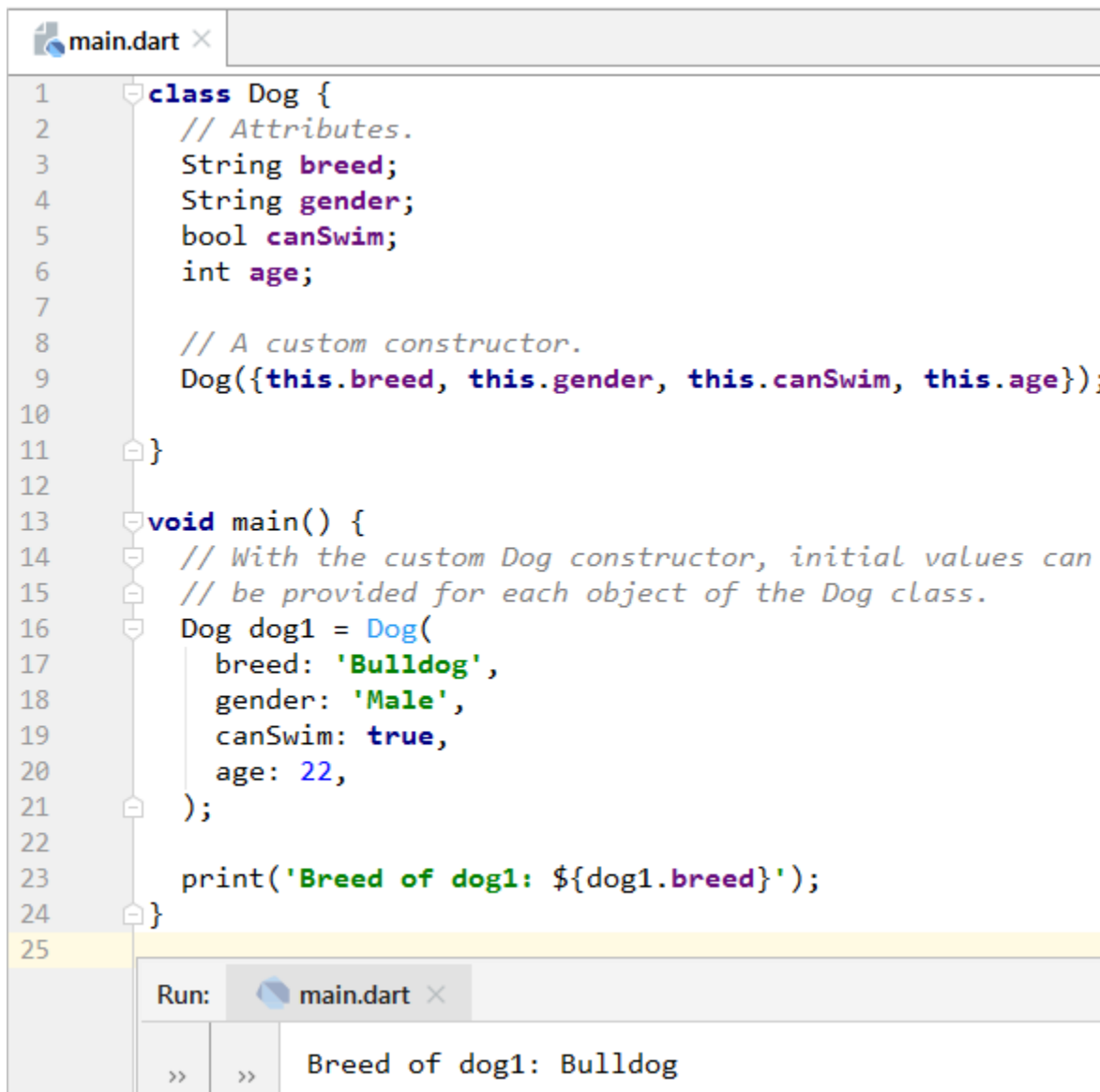
```
1 class Dog {
2     // Attributes.
3     String breed;
4     String gender;
5     bool canSwim;
6     int age;
7
8     // A custom constructor.
9     Dog(this.breed, this.gender, this.canSwim, this.age);
10
11    // Actions
12    void bark() {
13        print('This dog is barking');
14    }
15
16    void swim() {
17        print('This dog is swimming');
18    }
19 }
20
21 void main() {
22     // With the custom Dog constructor, initial values can
23     // be provided for each object of the Dog class.
24     Dog dog1 = Dog('Bulldog', 'Male', true, 22);
25     Dog dog2 = Dog('Chihuahua', 'Female', true, 15);
26
27     print('Breed of dog1: ${dog1.breed}');
28     print('Breed of dog2: ${dog2.breed}');
29 }
30
31 Run: main.dart x
32
33 dart main.dart [Observatory: http://127.0.0.1:62
34
35 Breed of dog1: Bulldog
36 Breed of dog2: Chihuahua
```

Screenshot 7.6

Running the above code still gives us the same result as the previous one did, except for the fact that we've defined a more concise constructor. Attaching the **this** keyword to the name of the parameters defined for the constructor function, automatically causes the values that are passed to the constructor, when it is called, to be assigned to the

instance variables defined in the class. This is a more convenient way of defining a constructor and it's the style we shall adopt henceforth.

Guess what? Constructors can also use optional named parameters and optional positional parameters, which I'm sure you recall how to define from when we first discussed functions and parameters in chapter 5. All that is required is to wrap the parameters in curly brackets ({} ) to make them optional named parameters, or wrap them in square brackets ([]) to make them optional positional parameters. An example of how optional named parameters can be defined for a constructor is shown below.



```
1 class Dog {  
2     // Attributes.  
3     String breed;  
4     String gender;  
5     bool canSwim;  
6     int age;  
7  
8     // A custom constructor.  
9     Dog({this.breed, this.gender, this.canSwim, this.age});  
10  
11 }  
12  
13 void main() {  
14     // With the custom Dog constructor, initial values can  
15     // be provided for each object of the Dog class.  
16     Dog dog1 = Dog(  
17         breed: 'Bulldog',  
18         gender: 'Male',  
19         canSwim: true,  
20         age: 22,  
21     );  
22  
23     print('Breed of dog1: ${dog1.breed}');  
24 }  
25
```

Run: main.dart ×

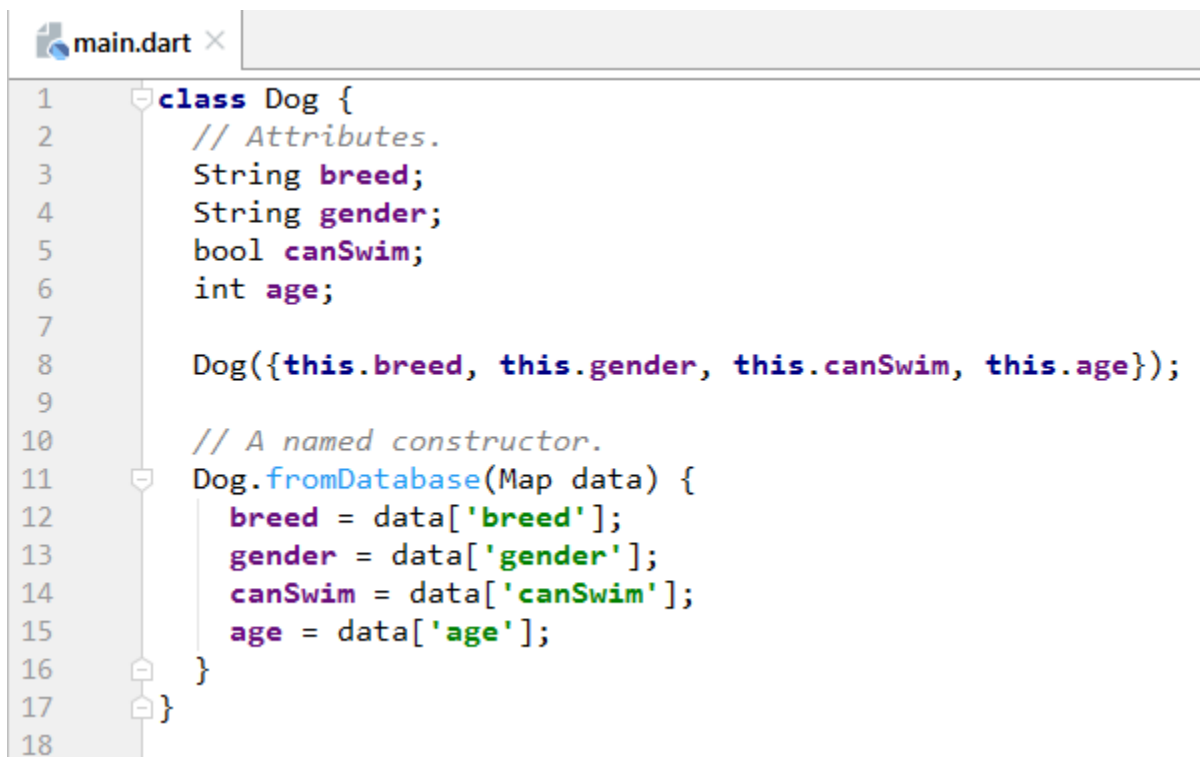
>> >> Breed of dog1: Bulldog

Screenshot 7.6

The need for using named parameters in a constructor is the same reason as that which was explained for normal functions. When the parameters to initialize are plenty, then

using named parameters is a better approach, as it makes it easy to recall the names of the parameters that need to be initialized. As an exercise, try refactor the constructor in the above program, to use optional positional parameters. Remember to fix the call to the constructor function, to reflect the change.

The constructors we've defined above all took the name of their class, nothing else. However there are times that you may need to define a constructor that has another name attached to it, along with the name of its class, this name would mean the source of the data, which is used in creating that particular object. Let's create such a constructor for the Dog class defined above.



```
main.dart x
1  class Dog {
2    // Attributes.
3    String breed;
4    String gender;
5    bool canSwim;
6    int age;
7
8    Dog({this.breed, this.gender, this.canSwim, this.age});
9
10   // A named constructor.
11   Dog.fromDatabase(Map data) {
12     breed = data['breed'];
13     gender = data['gender'];
14     canSwim = data['canSwim'];
15     age = data['age'];
16   }
17 }
18
```

```
19 void main() {
20   Dog dog1 = Dog.fromDatabase({
21     'breed': 'Chihuahua',
22     'gender': 'Male',
23     'canSwim': true,
24     'age': 20,
25   });
26
27   print('Breed of dog1 is: ${dog1.breed}');
28 }
29
```

Run: main.dart ×

Breed of dog1 is: Chihuahua

Screenshot 7.7

The constructor with the name *Dog.fromDatabase* is an example of a special kind of constructor that gets its data from a special source. Such a source could be a database, the internet, etc. You could have several such constructors in your programs. They only help clarify the source of the data that is used in constructing a particular object. Here, the constructor expects a map, when it is called. Internally, it retrieves the values from the map and uses them to initialize the instance variables in the class, so as to create an object.

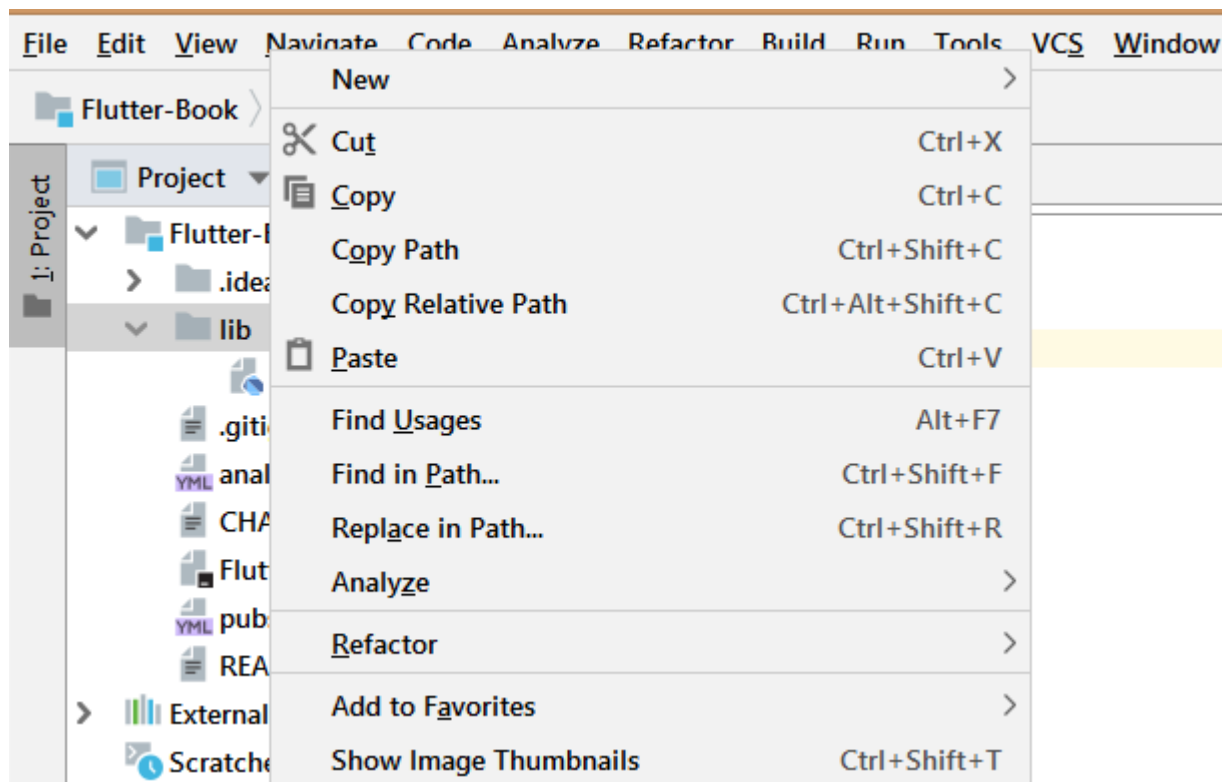
## Extending a class

Just as children inherit traits from their parents in the real world, so can classes inherit properties and behavior from one another. This concept is referred to as **inheritance**, and it is one of the major features of object oriented programming.

When class A inherits from class B, class A is said to be the child class (subclass) while class B is said to be the parent class (superclass). We will look at an example on this in a moment.

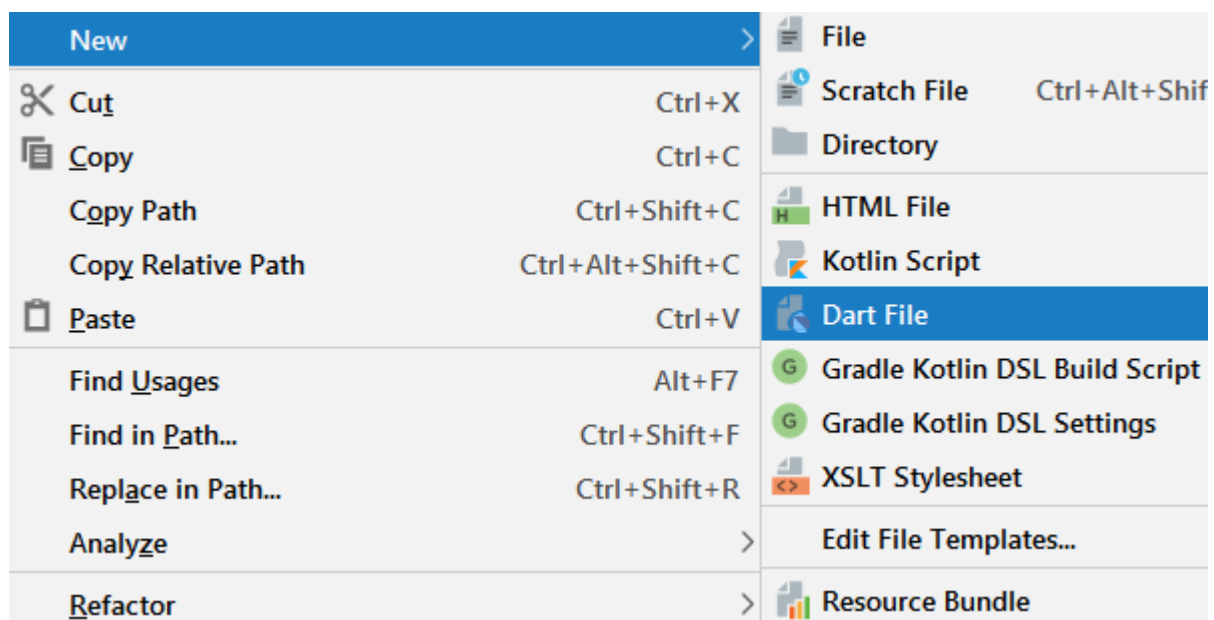
So far, we've been writing code in a single file, the **main.dart** file. Usually, that is not a problem when your code spans only a few lines. However, as your code gets plenty, it becomes necessary to spread your code across several files, so as to keep it clean and easy to manage. The program we shall look at next will consist of three (3) files. One file shall contain the definition of our usual *main* function, while the other files will contain class definitions.

To create a new file, right-click on the **lib** folder in your project's structure and you would see a menu pop up as shown in the screenshot below.



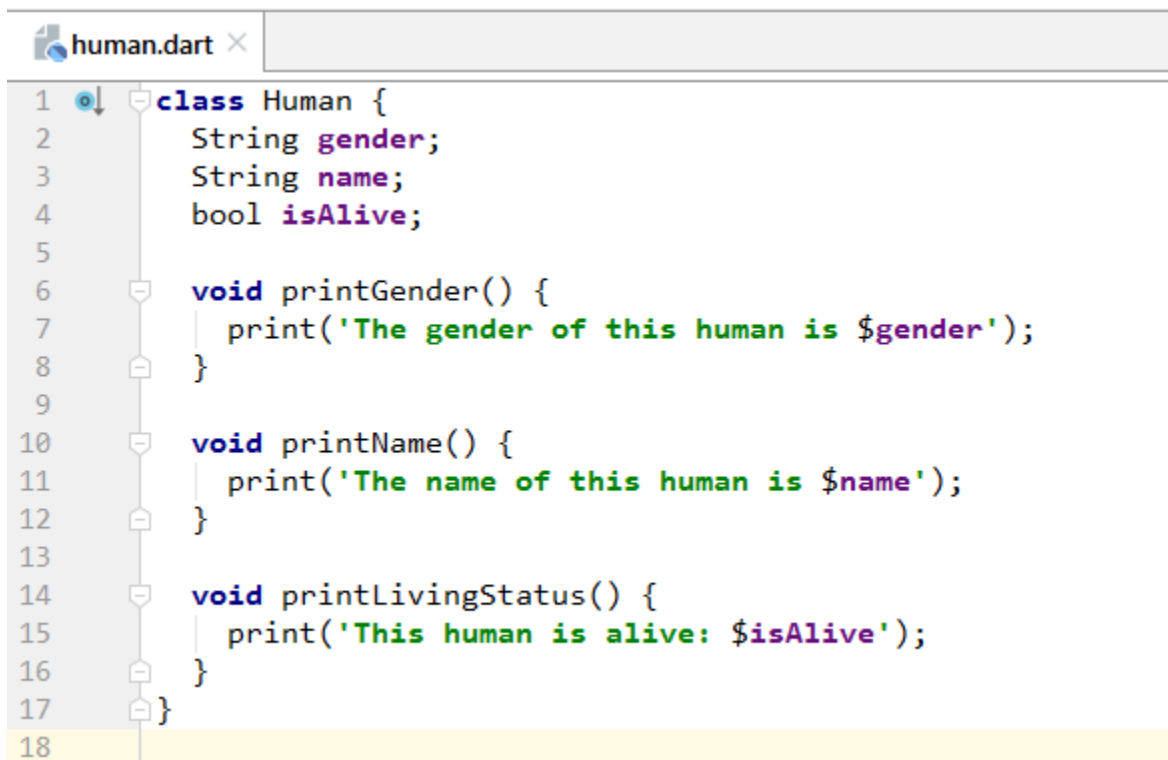
Screenshot 7.8

In the menu that comes up afterwards, click on **New**, then you would get another menu, from which you can click on **Dart File**, to create a new Dart file.



Screenshot 7.9

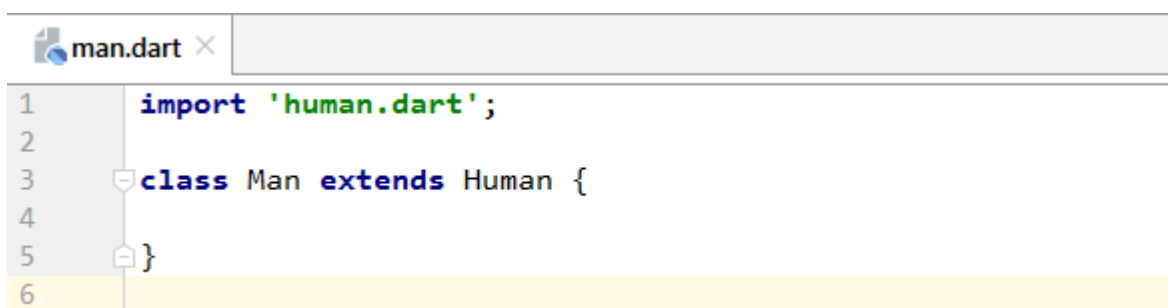
After clicking on **Dart File**, a pop up which contains an input field in which you can specify the name of the file would show. Enter the name of the file and click on OK. This would create a new Dart file with the name you entered. Create two files using the steps above and name them **human.dart** and **man.dart**. The human.dart file should contain the following class definition.



```
1 class Human {
2     String gender;
3     String name;
4     bool isAlive;
5
6     void printGender() {
7         print('The gender of this human is $gender');
8     }
9
10    void printName() {
11        print('The name of this human is $name');
12    }
13
14    void printLivingStatus() {
15        print('This human is alive: $isAlive');
16    }
17 }
18
```

Screenshot 7.10

The man.dart file should contain the following class definition.



```
1 import 'human.dart';
2
3 class Man extends Human {
4
5 }
6
```

Screenshot 7.11

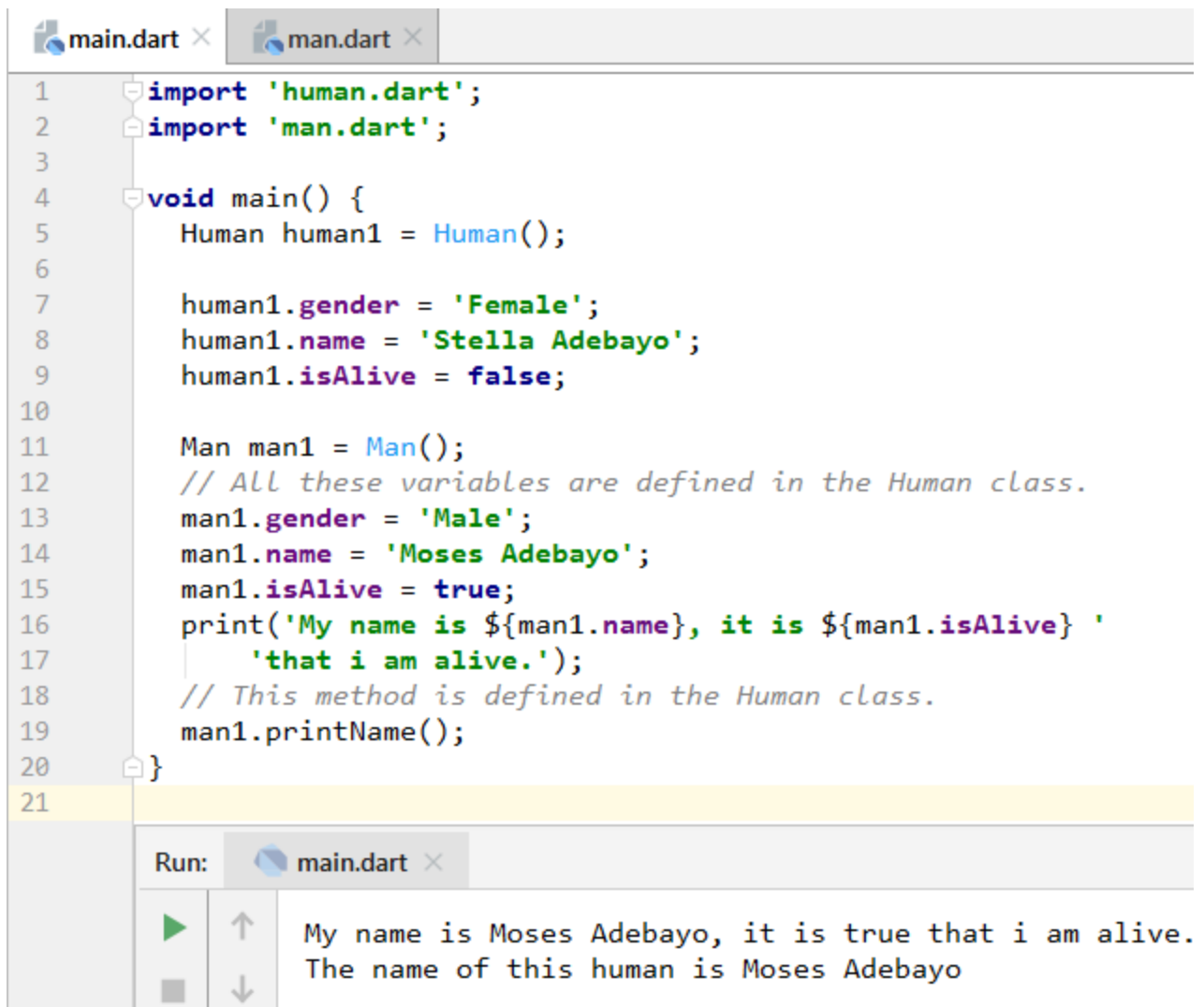
The *Human* class in the human.dart file acts as the superclass (parent) for the *Man* class which is the subclass (child). This parent-child relationship was established using



the **extends** keyword. When a class (e.g. class A) extends another class (e.g. class B), then class A (child class) automatically inherits all the instance variables and instance methods that are defined in class B (parent class).

It is the same with the *Human* and *Man* classes we've define above. The *Man* class being a subclass of the *Human* class, inherits all the members (variables and methods) of the *Human* class. What that means is that, if we create an object of the *Man* class, it would have a copy of all the instance variables and instance methods that are defined in the *Human* class, just as though they were defined in the *Man* class. That is how inheritance works. Observe that at the top of the **man.dart** file, we have an **import** statement that imports the **animal.dart** file. The import statement, helps us bring in all the code that is in the **animal.dart** file, and makes it available in the **man.dart** file. By so doing, the *Human* class becomes visible in the **man.dart** file, which makes it possible for the *Man* class to extend it.

In the program below, we've created an object of the *Human* and *Man* classes. Again, notice that in order to create objects of these two classes, we had to import the files which the classes are defined in, into the **main.dart** file, which we did on the lines 1 and 2.



The screenshot shows an IDE with two tabs: `main.dart` and `man.dart`. The `main.dart` tab is active, displaying the following Dart code:

```
1 import 'human.dart';
2 import 'man.dart';
3
4 void main() {
5     Human human1 = Human();
6
7     human1.gender = 'Female';
8     human1.name = 'Stella Adebayo';
9     human1.isAlive = false;
10
11     Man man1 = Man();
12     // All these variables are defined in the Human class.
13     man1.gender = 'Male';
14     man1.name = 'Moses Adebayo';
15     man1.isAlive = true;
16     print('My name is ${man1.name}, it is ${man1.isAlive} '
17         'that i am alive. ');
18     // This method is defined in the Human class.
19     man1.printName();
20 }
21
```

Below the code editor, there is a 'Run:' button and a console window. The console shows the output of the program:

```
My name is Moses Adebayo, it is true that i am alive.
The name of this human is Moses Adebayo
```

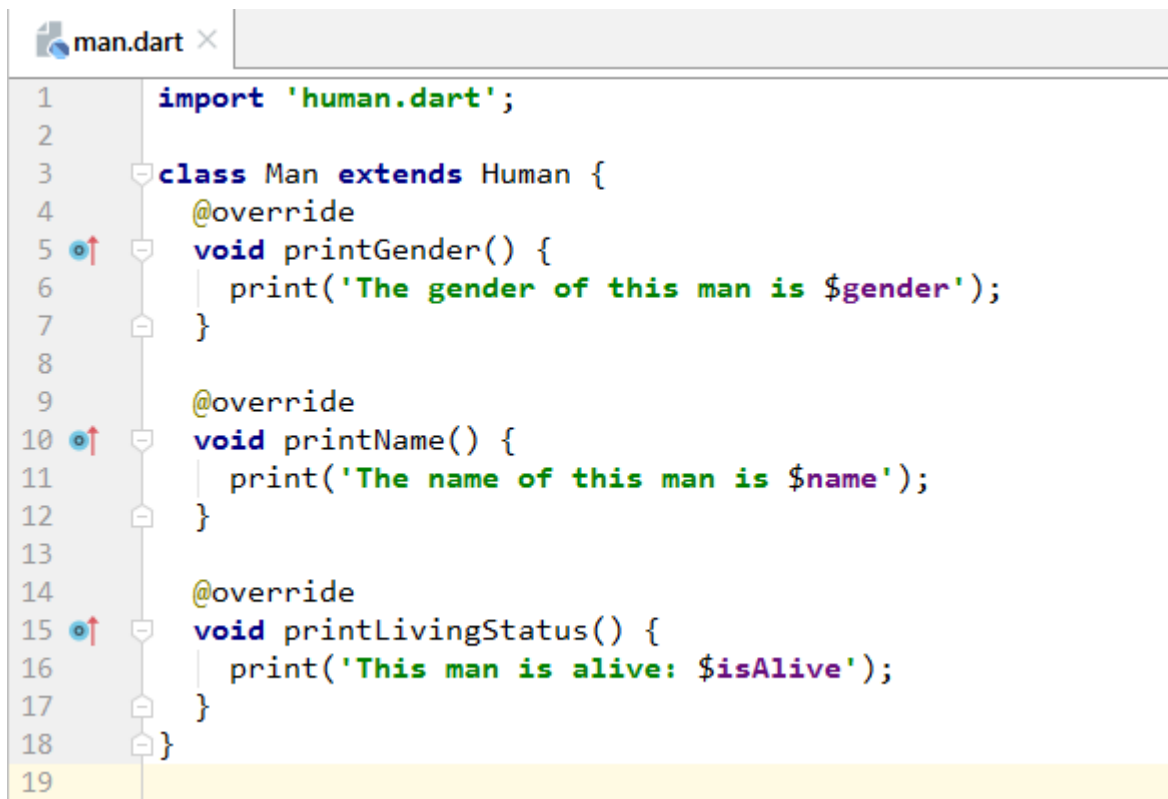
Screenshot 7.12

Just as I explained earlier, every object of the *Man* class has a copy of the variables and methods defined in the *Human* class. You can see how the *man1* object is able to access the *gender*, *name* and *isAlive* variables that were defined in the *Human* class.

**Note that a class can only extend one class at a time.**

## Overriding methods

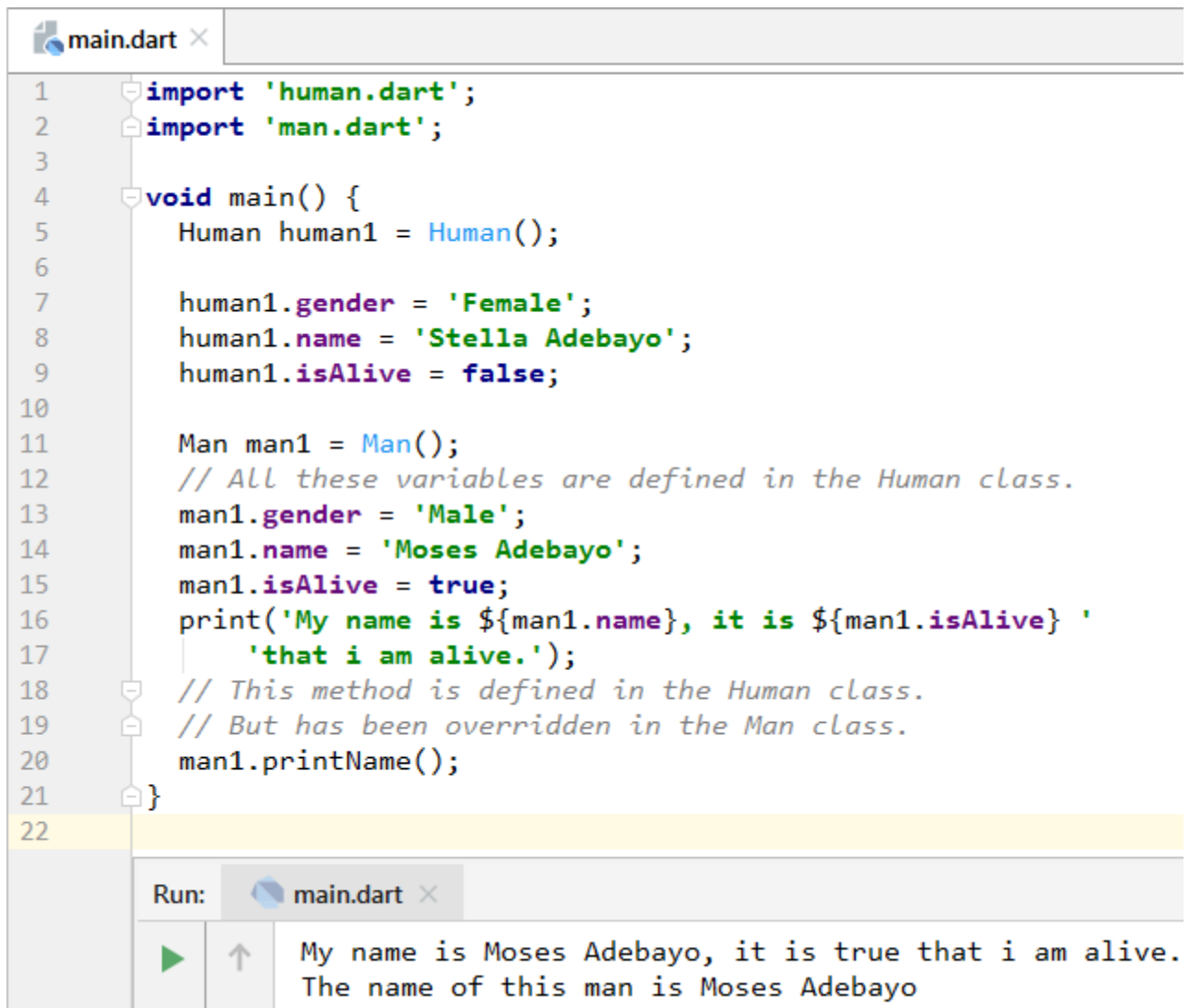
A subclass can override the method(s) it inherits from its superclass. Overriding a method simply means providing a custom implementation for the method. Using the *printGender*, *printName* and *printLivingStatus* methods defined in the *Human* class for example, the *Man* class can override their current implementation and provide a new implementation. Let's see how that can be done.



```
1  import 'human.dart';
2
3  class Man extends Human {
4      @override
5      void printGender() {
6          print('The gender of this man is $gender');
7      }
8
9      @override
10     void printName() {
11         print('The name of this man is $name');
12     }
13
14     @override
15     void printLivingStatus() {
16         print('This man is alive: $isAlive');
17     }
18 }
19
```

Screenshot 7.13

The *Man* class has overridden the methods it inherits from the *Human* class. Observe the difference in the print statements in all the methods, compared to the ones in the methods that are defined in the *Human* class (in Screenshot 7.9). Also, observe that there is an **@override** annotation at the top of each method. The **@override** annotation is a way we tell Dart, that we want to override a method (provide a different implementation for the method, as opposed to the original implementation). Now, when we create an object of the *Man* class and call any of the methods, we get the implementation of the method in the *Man* class, not the one in the *Human* class. This is shown in the example below.



```
1 import 'human.dart';
2 import 'man.dart';
3
4 void main() {
5     Human human1 = Human();
6
7     human1.gender = 'Female';
8     human1.name = 'Stella Adebayo';
9     human1.isAlive = false;
10
11     Man man1 = Man();
12     // All these variables are defined in the Human class.
13     man1.gender = 'Male';
14     man1.name = 'Moses Adebayo';
15     man1.isAlive = true;
16     print('My name is ${man1.name}, it is ${man1.isAlive} '
17         'that i am alive. ');
18     // This method is defined in the Human class.
19     // But has been overridden in the Man class.
20     man1.printName();
21 }
22
```

Run: main.dart

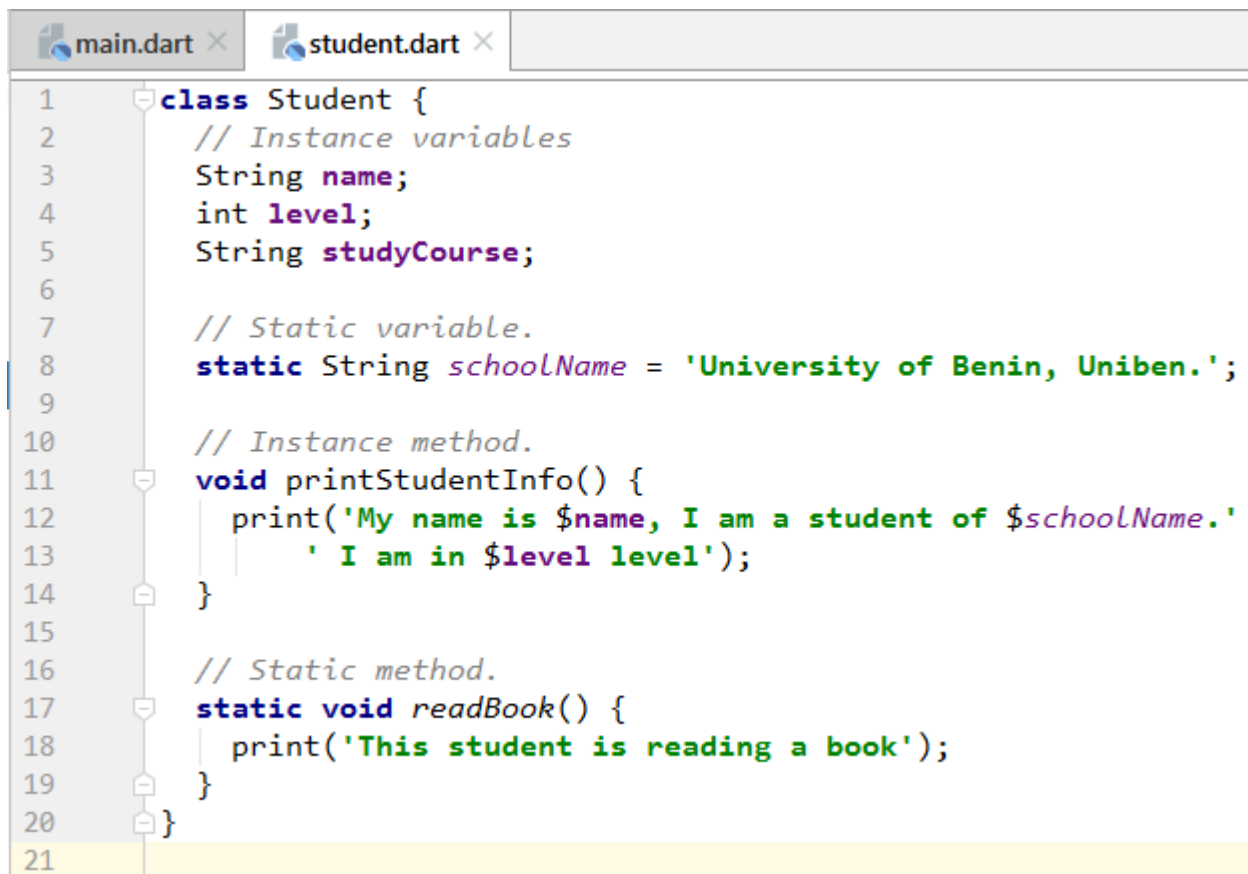
My name is Moses Adebayo, it is true that i am alive.  
The name of this man is Moses Adebayo

Screenshot 7.14

Instead of the text “The name of this human is Moses Adebayo”, we get “The name of this man is Moses Adebayo”, which is what is in the *printName* method that is defined in the *Man* class. Feel free to call the other methods.

## Static Members of a Class

Other than instance variables and methods, a class can also have **static variables** and **methods**. Static variables and methods only belong to the class which they’re defined in. Objects that are created from the class do not have a copy of them. If a static variable is to be accessed, it is done using the class name. Let’s create a class that contains a static variable and method. To do that, create a new file and call it **student.dart**.

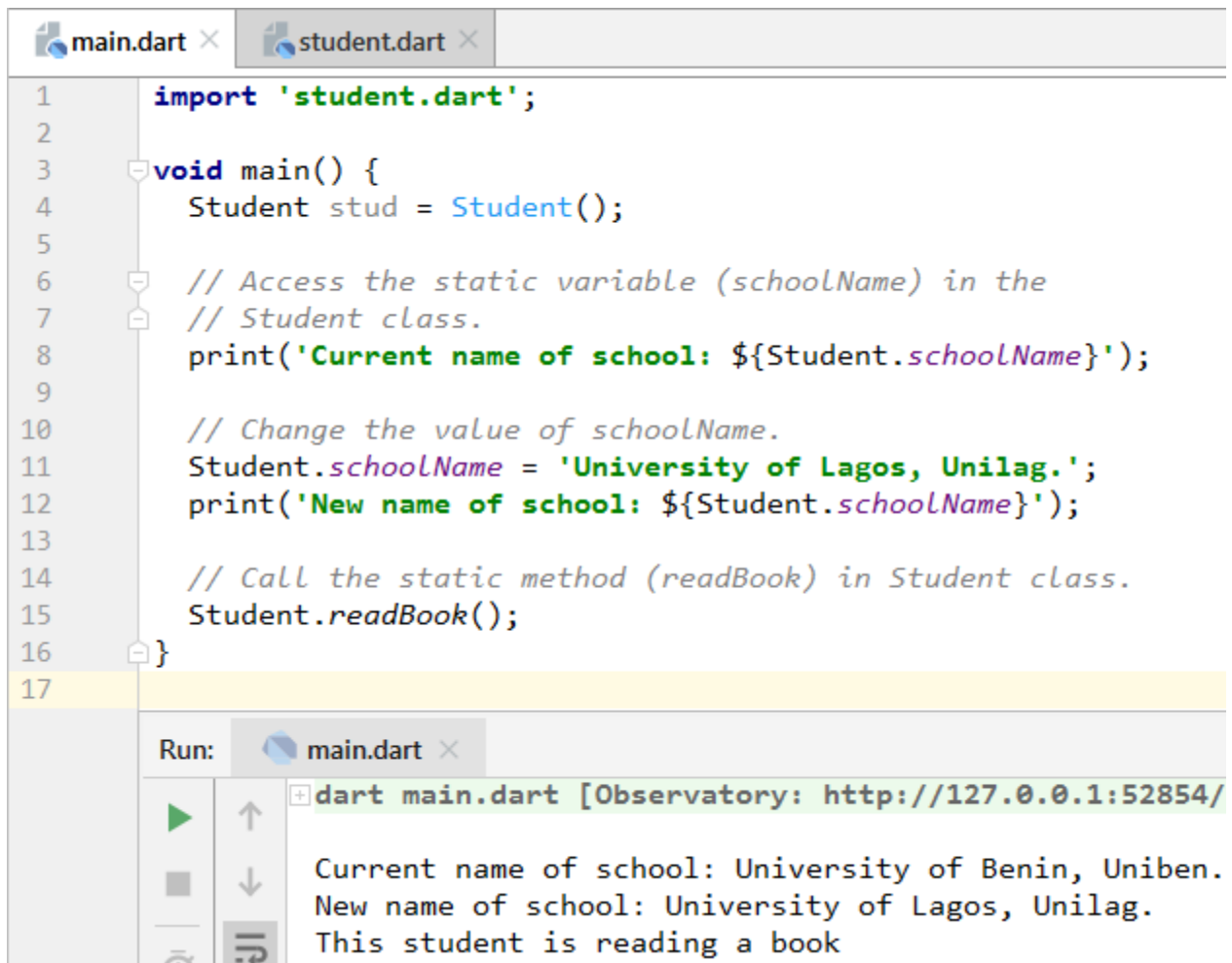


```
1 class Student {
2     // Instance variables
3     String name;
4     int level;
5     String studyCourse;
6
7     // Static variable.
8     static String schoolName = 'University of Benin, Uniben.';
9
10    // Instance method.
11    void printStudentInfo() {
12        print('My name is $name, I am a student of $schoolName.'
13            ' I am in $level level');
14    }
15
16    // Static method.
17    static void readBook() {
18        print('This student is reading a book');
19    }
20 }
21
```

Screenshot 7.15

In the program above, the *Student* class has a static variable called *schoolName*. To create a static variable, simply add the **static** keyword before the type of the variable as was done for the *schoolName* variable above. This would tell Dart, that the variable belongs only to the class and not to any object or instance of the class. To define a static method, simply add the **static** keyword before the return type of the method, just as is done for the *readBook* method above.

The *schoolName* variable has been given an initial value, however it can be changed, just like for instance variables.



```
1 import 'student.dart';
2
3 void main() {
4   Student stud = Student();
5
6   // Access the static variable (schoolName) in the
7   // Student class.
8   print('Current name of school: ${Student.schoolName}');
9
10  // Change the value of schoolName.
11  Student.schoolName = 'University of Lagos, Unilag.';
12  print('New name of school: ${Student.schoolName}');
13
14  // Call the static method (readBook) in Student class.
15  Student.readBook();
16 }
17
```

Run: main.dart x

dart main.dart [Observatory: http://127.0.0.1:52854/

Current name of school: University of Benin, Uniben.  
New name of school: University of Lagos, Unilag.  
This student is reading a book

Screenshot 7.16

A static variable is accessed using the name of the class. Observe how on line 8, *schoolName* is accessed using the name of the class (*Student*) and the initial value that was assigned is printed. While on lines 11 and 12, a new value is assigned to *schoolName* and then printed. On line 15, the static method *readBook* is called and it prints the expected message to the console.

Accessing *schoolName* or *readBook* using an object of the *Student* class won't be possible, because they're static members of the *Student* class. Feel free to test that out.

You might be wondering what the usefulness of static variables and methods are in a class. Why not simply make everything an instance variable or method? Usually, it is good to make a particular variable or method static when that variable or method is common to all objects that would be created from that class. Take the *Student* class for example, the *schoolName* attribute is common to all *Student* objects. All *Student* objects have the same school name. Imagine the *Student* class was for the students of the University of Benin. All students of the University of Benin have the same school name,

so it would be best to make such a variable in a Student model class **static**, because it would never change for any of the student objects. It is same with the *readBook* method, all students read their books, so such a method in a Student model class is best made **static**.

## Revisiting the Earlier Types

In chapter 2, we looked at how to use the **int**, **double**, **String**, **bool**, **List**, and **Map** types to define variables. In that chapter, we didn't really go into much detail on how variables that are created with these types can be manipulated or the kinds of operations that can be performed on them. I decided to introduce all of that to you after you must have learnt about **Object Oriented Programming**.

It is important you know that everything in Dart is an object, even the number 2 is an object. The **int**, **double**, **String**, **List**, and **Map** types are actually predefined classes that provide some methods and properties for their objects.

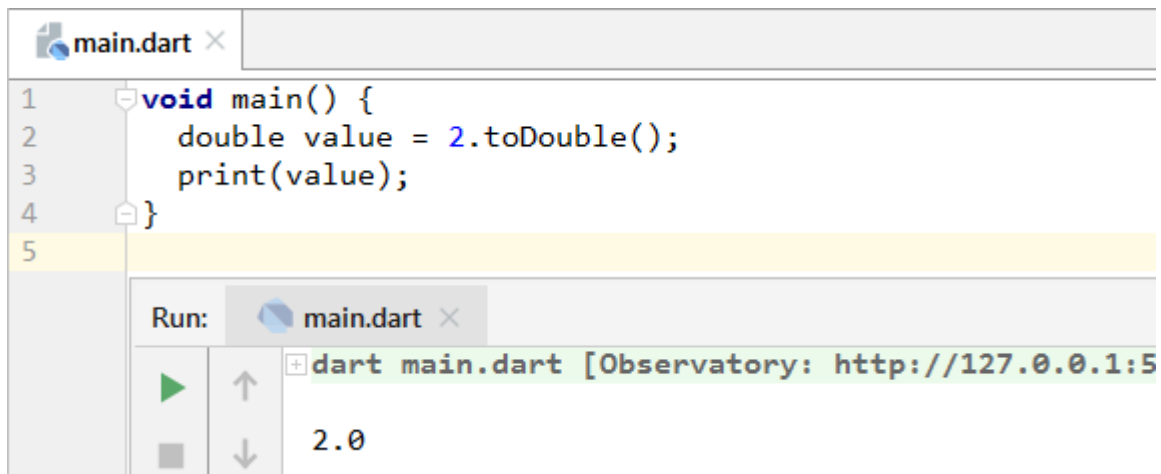
### Int

When you declare a variable to be of type **int**, various operations can be performed on that variable. We shall look at the following operations:

- 9. `toDouble`
- 10. `abs`
- 11. `isEven`
- 12. `toString`
- 13. `int.parse`

### toDouble

This int method turns an integer value into a decimal value.



```
main.dart x
1 void main() {
2   double value = 2.toDouble();
3   print(value);
4 }
5
```

Run: main.dart x

dart main.dart [Observatory: http://127.0.0.1:5

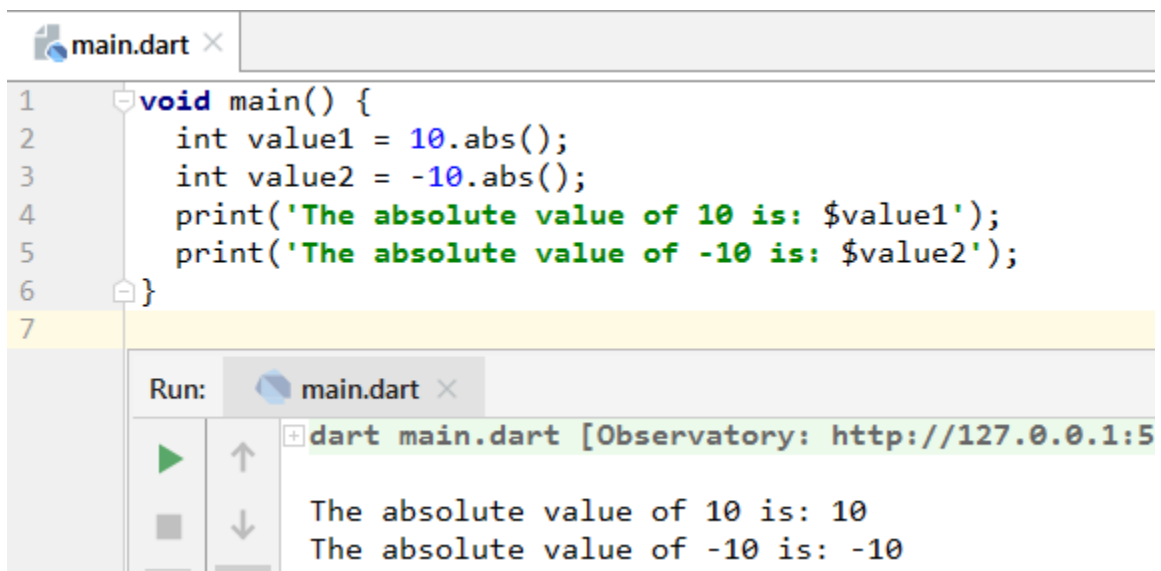
2.0

Screenshot 7.17

You can see from the screenshot above, in the program, the method *toDouble* was called on the integer value 2, to turn it into a double value. It becomes 2.0 which is shown in the console.

## abs

When the *abs* method is called on an integer value, it returns the absolute value of it. If it is called on a positive integer value, it simply returns the value, but if it is called on a negative integer value, it returns the absolute (positive) value of it.



```
main.dart x
1 void main() {
2   int value1 = 10.abs();
3   int value2 = -10.abs();
4   print('The absolute value of 10 is: $value1');
5   print('The absolute value of -10 is: $value2');
6 }
7
```

Run: main.dart x

dart main.dart [Observatory: http://127.0.0.1:5

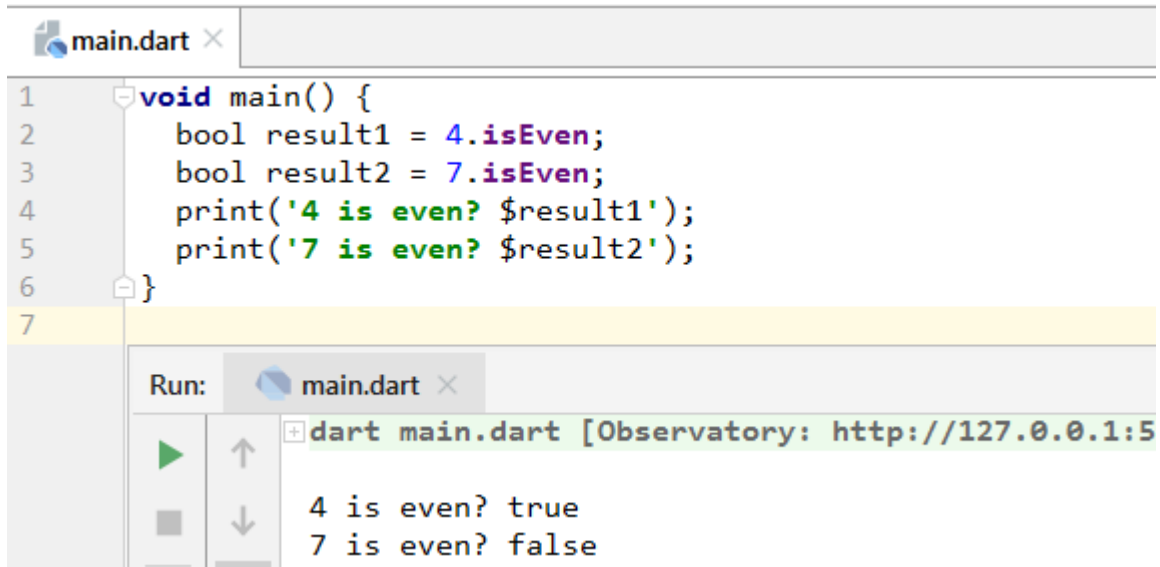
The absolute value of 10 is: 10  
The absolute value of -10 is: -10

Screenshot 7.18



## isEven

The `isEven` property returns **true**, if an integer number is even, else it returns **false**.



The screenshot shows an IDE window with a file named `main.dart`. The code defines a `main` function that calculates the evenness of 4 and 7 using the `isEven` property. Below the code editor, the 'Run' button is pressed, and the output console shows the results of the `print` statements.

```
1 void main() {  
2     bool result1 = 4.isEven;  
3     bool result2 = 7.isEven;  
4     print('4 is even? $result1');  
5     print('7 is even? $result2');  
6 }  
7
```

Run: `main.dart` ×

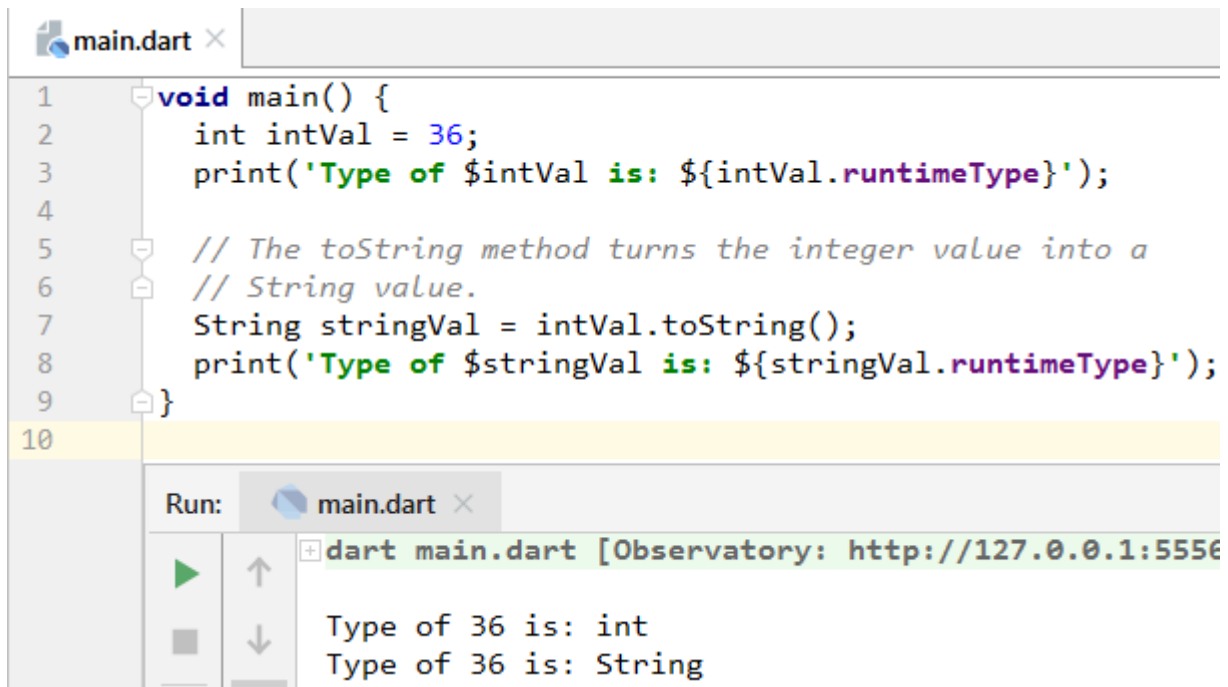
`dart main.dart [Observatory: http://127.0.0.1:5`

```
4 is even? true  
7 is even? false
```

Screenshot 7.19

## toString

The `toString` method is a very special method, it can be called on any object in Dart, not just on an `int` object. The `toString` method turns an object to its **String** equivalent. If the `toString` method is called on an integer value, it turns it into a `String` value.



```
main.dart x
1 void main() {
2   int intVal = 36;
3   print('Type of $intVal is: ${intVal.runtimeType}');
4
5   // The toString method turns the integer value into a
6   // String value.
7   String stringVal = intVal.toString();
8   print('Type of $stringVal is: ${stringVal.runtimeType}');
9 }
10
```

Run: main.dart x

dart main.dart [Observatory: http://127.0.0.1:5556]

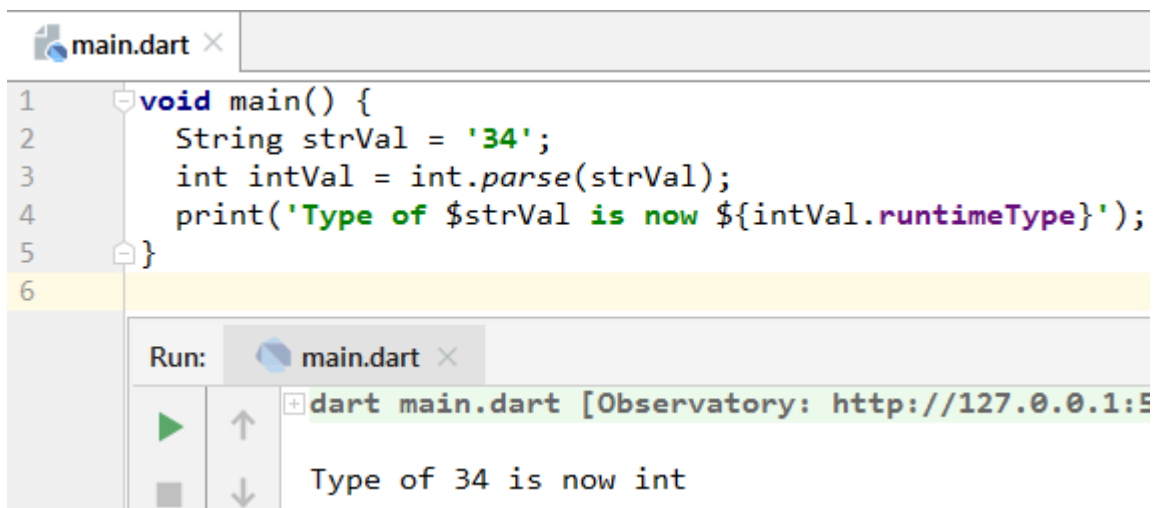
Type of 36 is: int  
Type of 36 is: String

Screenshot 7.20

Notice how the **runtimeType** property is used to obtain the type of an object when the program is executed.

## int.parse

The *int.parse* method can be used to turn a String value into an int value. Although a condition has to be satisfied, the String value must be an integer number. It isn't possible to use the *int.parse* method to convert a String value that isn't an integer number into an int value.



```
main.dart x
1 void main() {
2   String strVal = '34';
3   int intVal = int.parse(strVal);
4   print('Type of $strVal is now ${intVal.runtimeType}');
5 }
6
```

Run: main.dart x

dart main.dart [Observatory: http://127.0.0.1:5556]

Type of 34 is now int

Screenshot 7.21

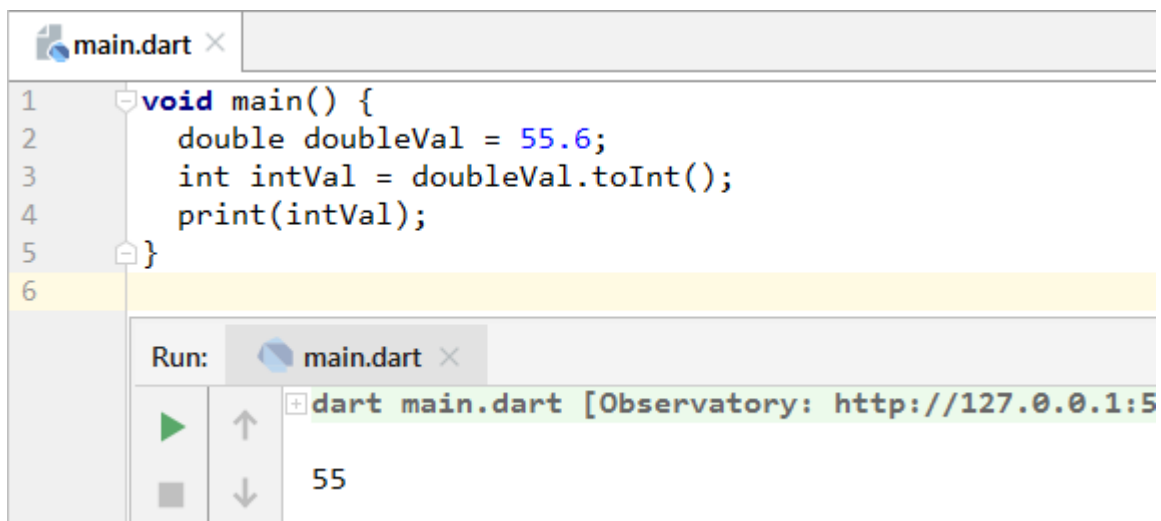
## double

Just like the **int** type, when you declare a variable to be of type **double**, various operations can be performed on the variable. We shall look at the following operations:

1. `toInt`
2. `toString`
3. `floor`
4. `ceil`
5. `truncate`

## toInt

The `toInt` method turns a decimal value into an int value.



The screenshot shows an IDE window titled 'main.dart' with the following code:

```
1 void main() {  
2   double doubleVal = 55.6;  
3   int intVal = doubleVal.toInt();  
4   print(intVal);  
5 }  
6
```

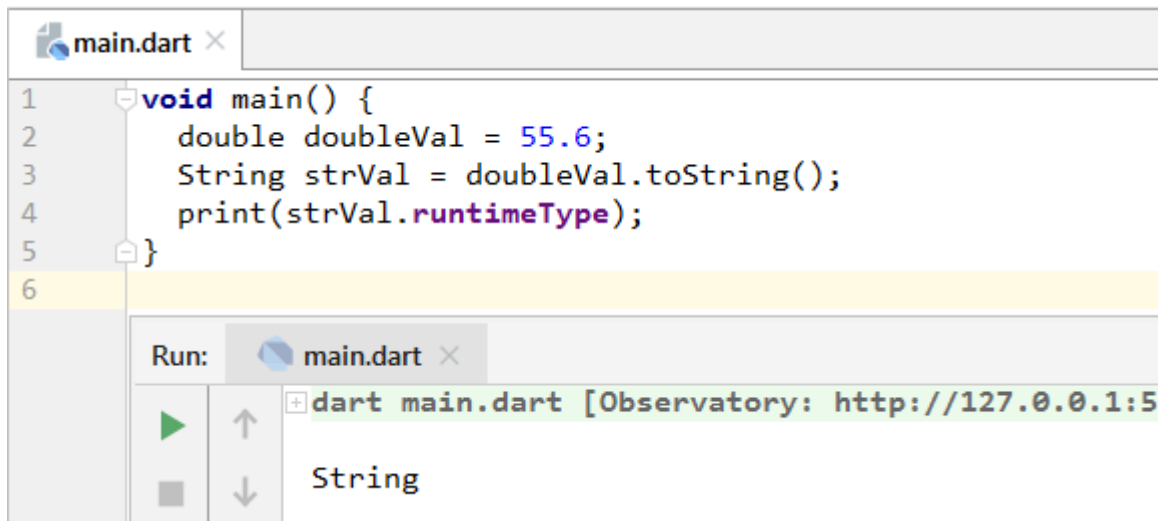
Below the code editor, there is a 'Run' button and a console output area. The console shows the output of the program:

```
dart main.dart [Observatory: http://127.0.0.1:5555]  
55
```

Screenshot 7.22

## toString

The `toString` method turns a double value into a String value (i.e. the String equivalent).



The screenshot shows an IDE window with a file named `main.dart`. The code is as follows:

```
1 void main() {  
2   double doubleVal = 55.6;  
3   String strVal = doubleVal.toString();  
4   print(strVal.runtimeType);  
5 }  
6
```

Below the code editor, the 'Run' button is active, and the output console shows the following:

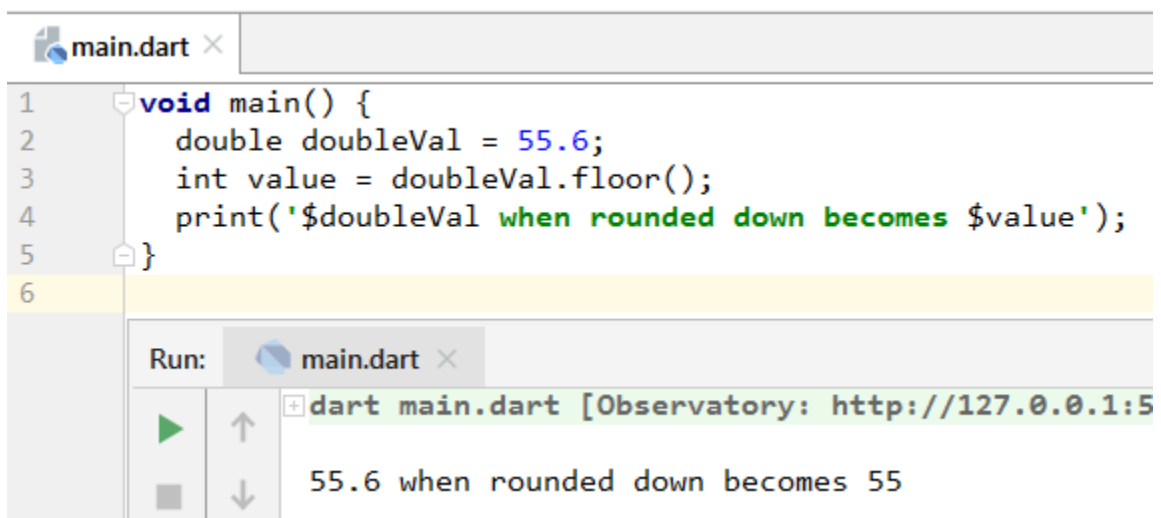
```
Run: main.dart  
+ dart main.dart [Observatory: http://127.0.0.1:5  
String
```

Screenshot 7.23

Here, the type of `strVal` is printed as proof that the value it contains is a String value.

## floor

The `floor` method rounds down a decimal value to its nearest whole number. The result is usually an integer.



The screenshot shows an IDE window with a file named `main.dart`. The code is as follows:

```
1 void main() {  
2   double doubleVal = 55.6;  
3   int value = doubleVal.floor();  
4   print('$doubleVal when rounded down becomes $value');  
5 }  
6
```

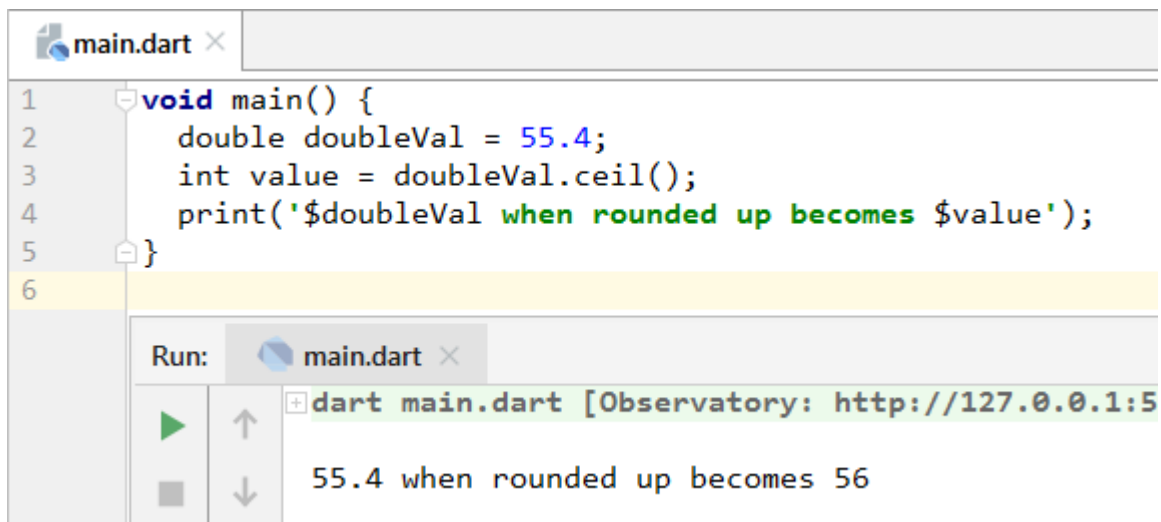
Below the code editor, the 'Run' button is active, and the output console shows the following:

```
Run: main.dart  
+ dart main.dart [Observatory: http://127.0.0.1:5  
55.6 when rounded down becomes 55
```

Screenshot 7.24

## ceil

The `ceil` method rounds up a decimal value to its nearest whole number. The result is usually an integer.



```
main.dart x
1 void main() {
2   double doubleVal = 55.4;
3   int value = doubleVal.ceil();
4   print('$doubleVal when rounded up becomes $value');
5 }
6
```

Run: main.dart x

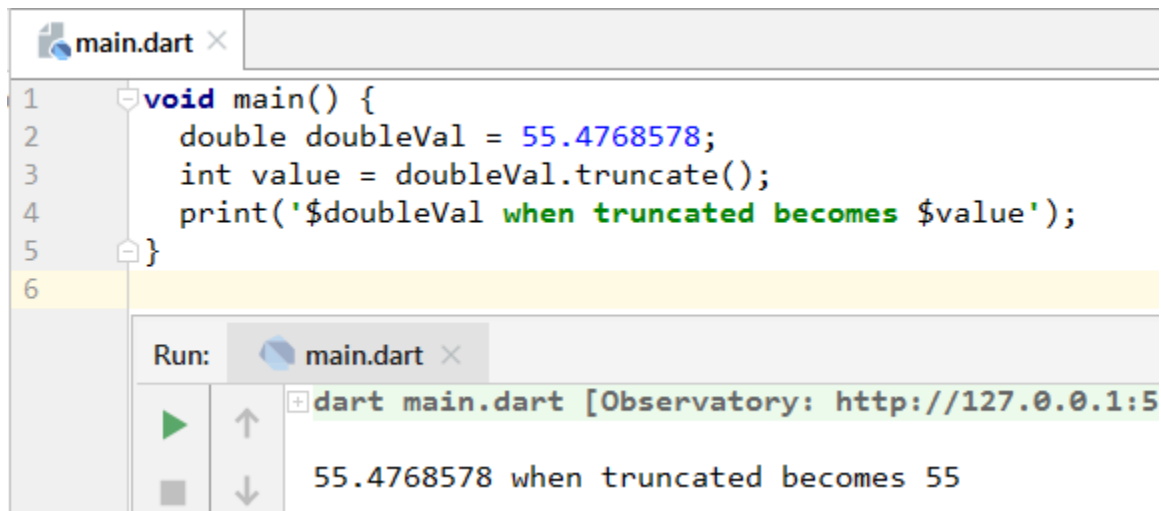
dart main.dart [Observatory: http://127.0.0.1:5

55.4 when rounded up becomes 56

Screenshot 7.25

## truncate

The *truncate* method removes the fractional part from a decimal number. It results in an integer value.



```
main.dart x
1 void main() {
2   double doubleVal = 55.4768578;
3   int value = doubleVal.truncate();
4   print('$doubleVal when truncated becomes $value');
5 }
6
```

Run: main.dart x

dart main.dart [Observatory: http://127.0.0.1:5

55.4768578 when truncated becomes 55

Screenshot 7.26

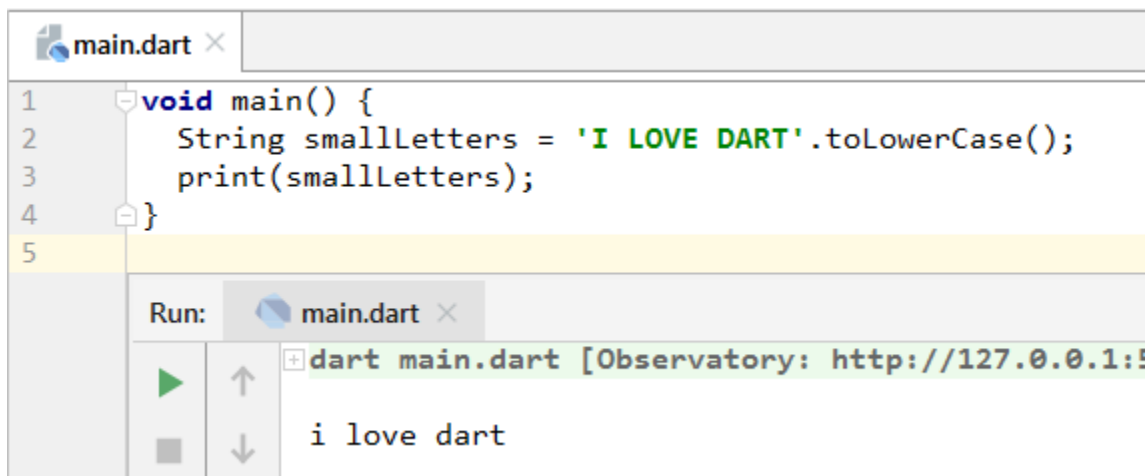
## String

The String class defines a lot of methods and properties that String objects can use, some of them include:

1. toUpperCase
2. toLowerCase
3. trim
4. split
5. isEmpty
6. length

## toLowerCase

The *toLowerCase* method transforms the uppercase letters of a string to their lowercase equivalent.



The screenshot shows an IDE window with a file named `main.dart`. The code is as follows:

```
1 void main() {  
2   String smallLetters = 'I LOVE DART'.toLowerCase();  
3   print(smallLetters);  
4 }  
5
```

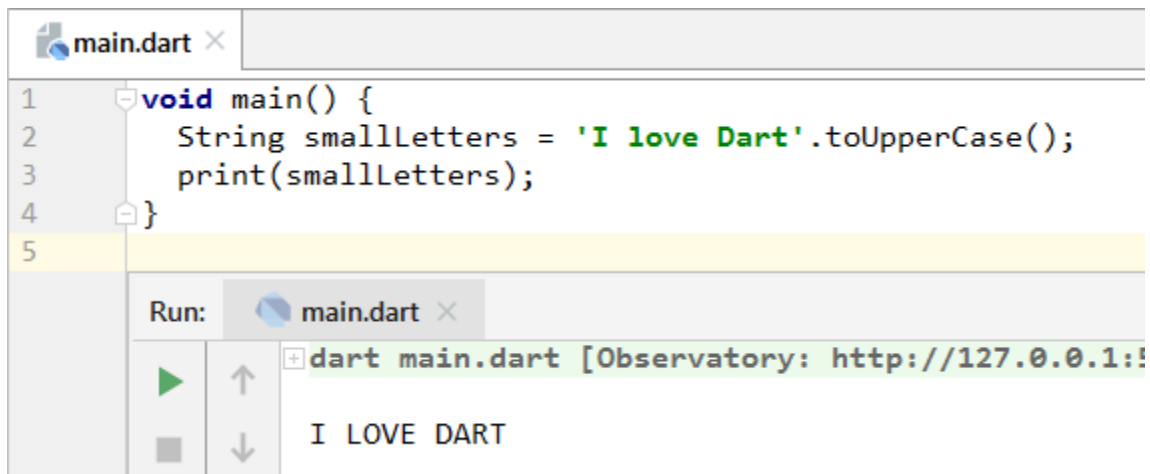
Below the code editor, the 'Run' tab is active, showing the execution of `main.dart`. The output console displays the result of the `print` statement:

```
dart main.dart [Observatory: http://127.0.0.1:8080]  
i love dart
```

Screenshot 7.27

## toUpperCase

Just as you probably guessed it, the *toUpperCase* method transforms the lowercase letters of a string to their uppercase equivalent.



The screenshot shows an IDE window with a file named `main.dart`. The code is as follows:

```
1 void main() {  
2   String smallLetters = 'I love Dart'.toUpperCase();  
3   print(smallLetters);  
4 }  
5
```

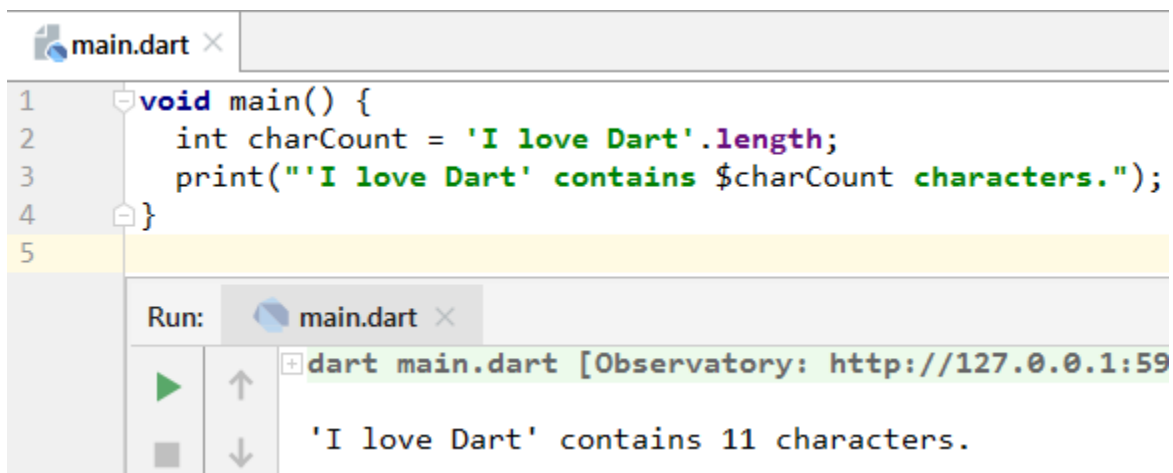
Below the code editor, the 'Run' tab is active, showing the execution of `main.dart`. The output is:

```
dart main.dart [Observatory: http://127.0.0.1:59123] I LOVE DART
```

Screenshot 7.28

## length

The *length* property returns the number of characters that are in a String value.



The screenshot shows an IDE window with a file named `main.dart`. The code is as follows:

```
1 void main() {  
2   int charCount = 'I love Dart'.length;  
3   print("'I love Dart' contains $charCount characters.");  
4 }  
5
```

Below the code editor, the 'Run' tab is active, showing the execution of `main.dart`. The output is:

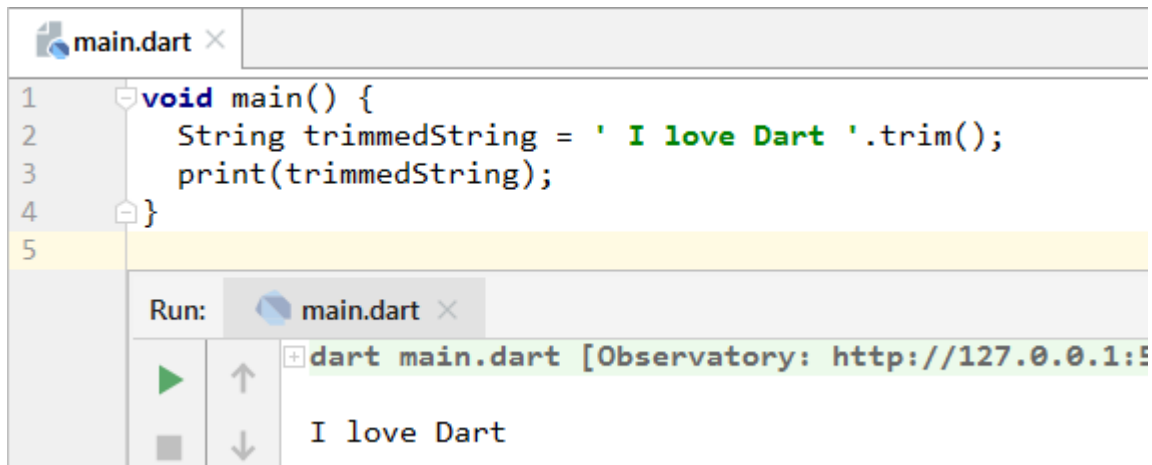
```
dart main.dart [Observatory: http://127.0.0.1:59123] 'I love Dart' contains 11 characters.
```

Screenshot 7.29

Know that the spaces in a String value also count as characters in it.

## trim

The *trim* method removes the spaces (if any) that are present at the left and right part of a String value.



The screenshot shows an IDE window with a file named `main.dart`. The code is as follows:

```
1 void main() {  
2   String trimmedString = ' I love Dart '.trim();  
3   print(trimmedString);  
4 }  
5
```

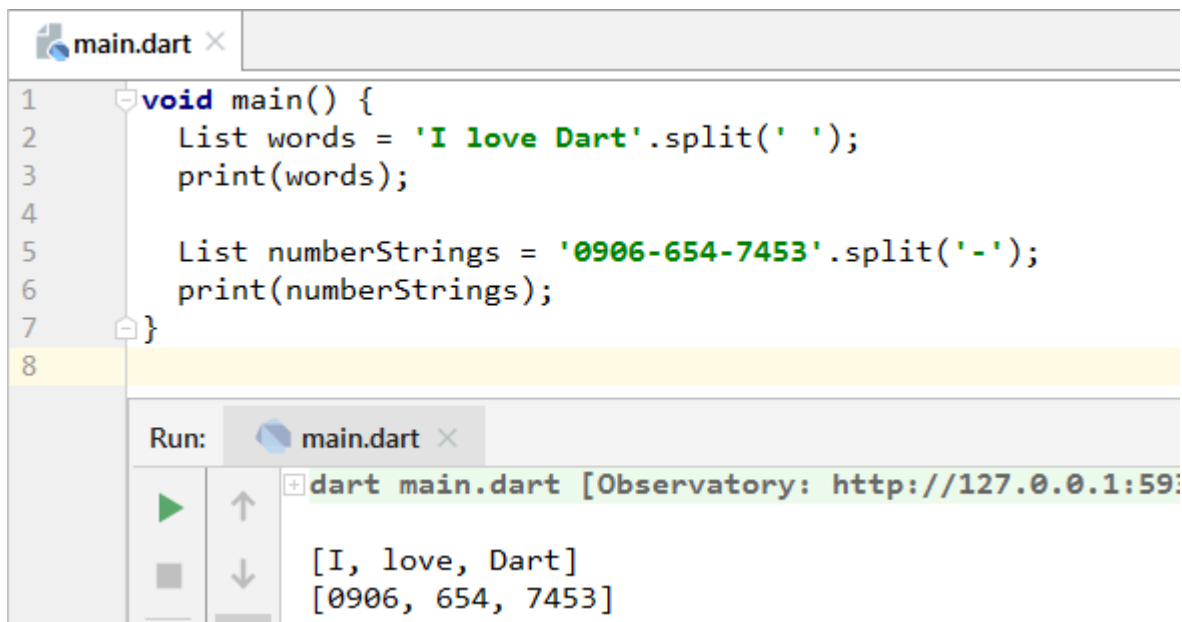
Below the code editor, the 'Run' tab is active, showing the output of the program:

```
Run: main.dart  
+ dart main.dart [Observatory: http://127.0.0.1:59121]  
I love Dart
```

Screenshot 7.30

## split

The *split* method is used to divide up a String value using a particular pattern or character.



The screenshot shows an IDE window with a file named `main.dart`. The code is as follows:

```
1 void main() {  
2   List words = 'I love Dart'.split(' ');  
3   print(words);  
4  
5   List numberStrings = '0906-654-7453'.split('-');  
6   print(numberStrings);  
7 }  
8
```

Below the code editor, the 'Run' tab is active, showing the output of the program:

```
Run: main.dart  
+ dart main.dart [Observatory: http://127.0.0.1:59121]  
[I, love, Dart]  
[0906, 654, 7453]
```

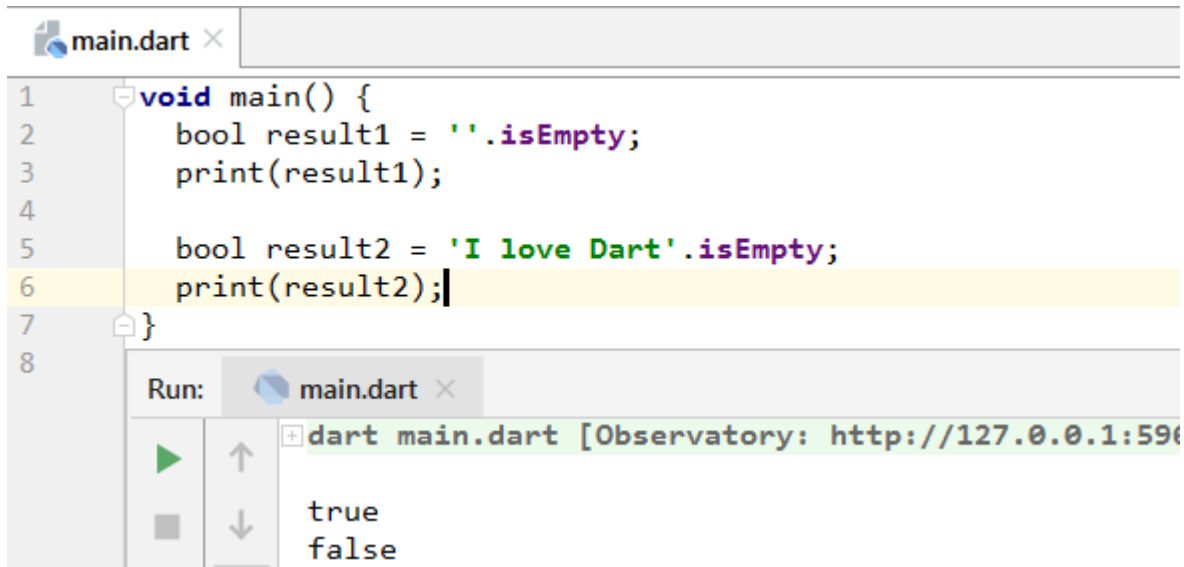
Screenshot 7.31

When the String value has been split, the result String values are put into a list.



## isEmpty

The *isEmpty* method returns **true** if a String value is empty (contains no characters), otherwise it returns **false**, when the String value contains characters.



The screenshot shows an IDE window with a file named `main.dart`. The code is as follows:

```
1 void main() {  
2   bool result1 = ''.isEmpty;  
3   print(result1);  
4  
5   bool result2 = 'I love Dart'.isEmpty;  
6   print(result2);  
7 }  
8
```

Below the code editor, there is a 'Run' button and a console output area. The console shows the output of the program:

```
dart main.dart [Observatory: http://127.0.0.1:596  
true  
false
```

Screenshot 7.32

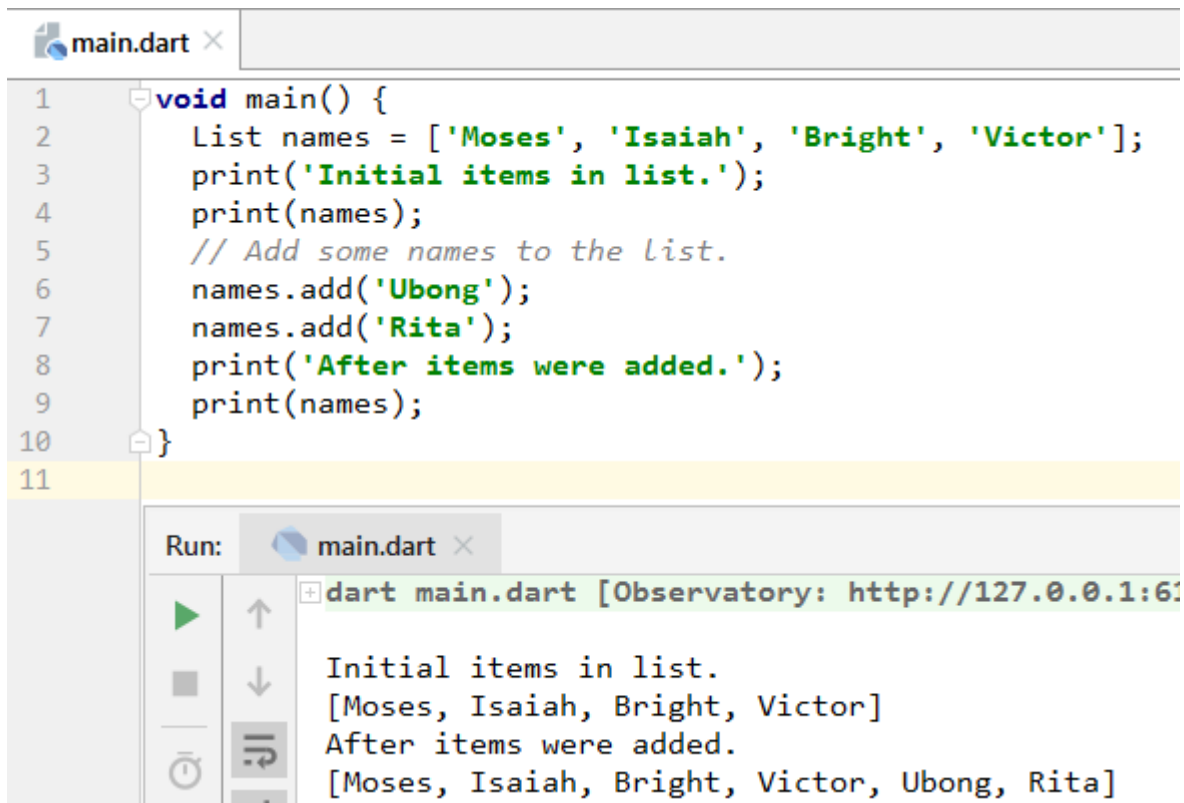
## List

The *List* type defines several methods or operations that can be carried out on a list.

1. add
2. remove
3. removeAt
4. indexOf
5. contains
6. isEmpty
7. map
8. forEach

### add

After a list is created with some initial items, more items can be added to it at a later point in time using the *add* method. The new item to be added is passed as an argument to the *add* method.



The screenshot shows an IDE window with a file named `main.dart`. The code defines a `main` function that creates a list of names, prints it, adds two more names, and prints it again. Below the code editor, the 'Run' panel shows the execution output for `dart main.dart`.

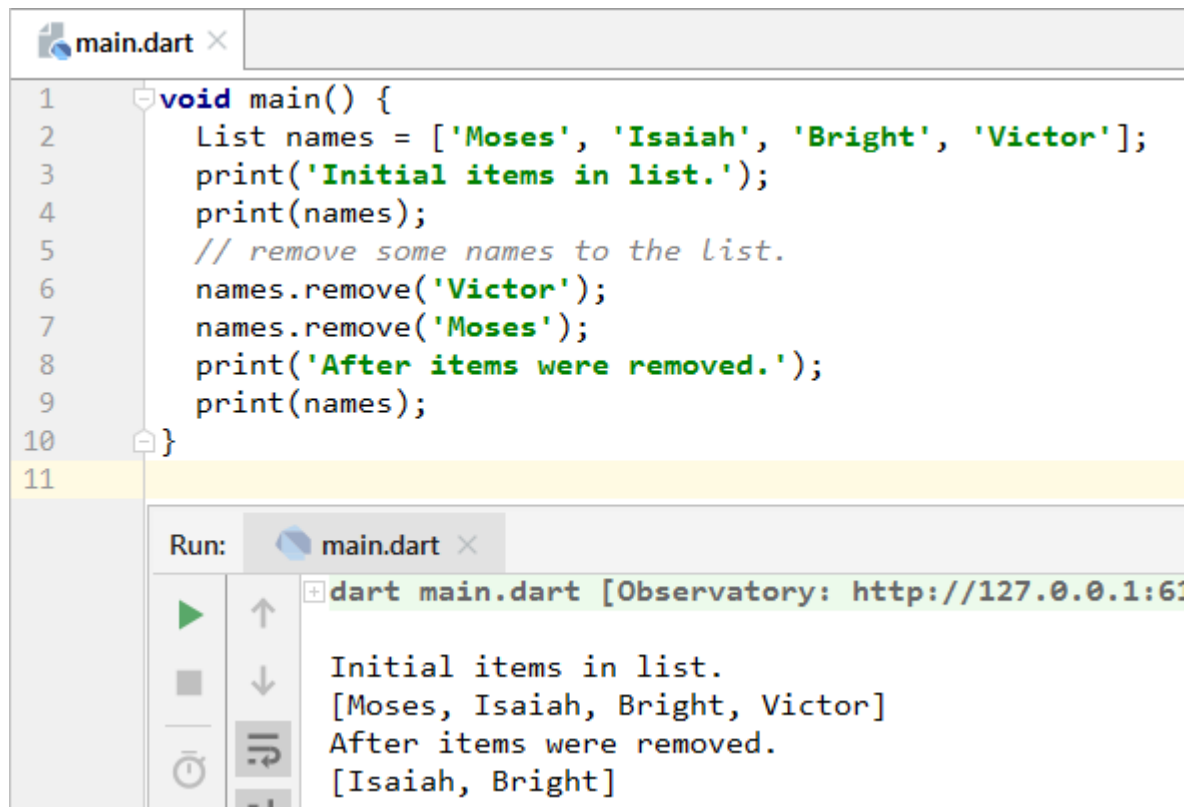
```
1 void main() {  
2   List names = ['Moses', 'Isaiah', 'Bright', 'Victor'];  
3   print('Initial items in list.');
```

```
Initial items in list.  
[Moses, Isaiah, Bright, Victor]  
After items were added.  
[Moses, Isaiah, Bright, Victor, Ubong, Rita]
```

Screenshot 7.33

## remove

An item in a list can be removed using the *remove* method. The item to be remove is passed to the *remove* method when it is called.



The screenshot shows an IDE window with a file named `main.dart`. The code defines a `main` function that creates a list of names, prints it, removes two elements, and prints it again. Below the code editor, the 'Run' tab is active, showing the execution output. The output matches the code's logic, displaying the initial list and the list after removing 'Victor' and 'Moses'.

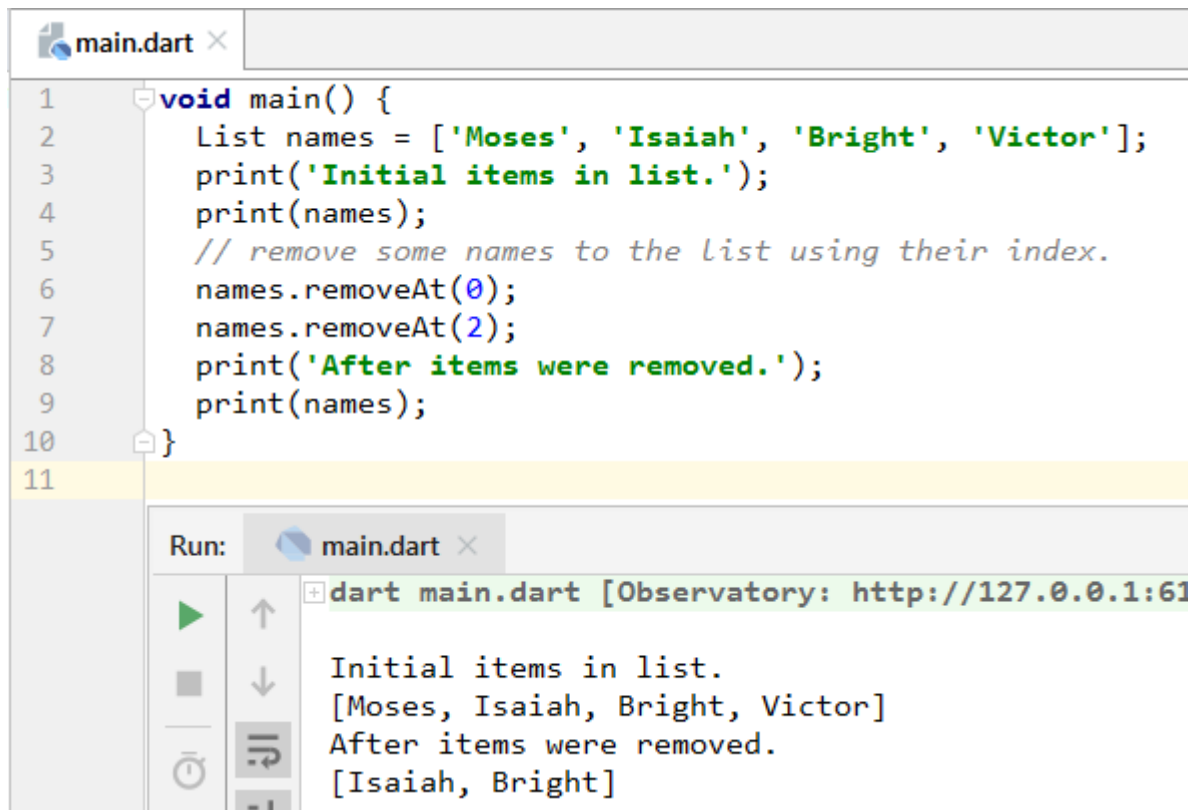
```
1 void main() {  
2   List names = ['Moses', 'Isaiah', 'Bright', 'Victor'];  
3   print('Initial items in list.');
```

```
Initial items in list.  
[Moses, Isaiah, Bright, Victor]  
After items were removed.  
[Isaiah, Bright]
```

Screenshot 7.34

## removeAt

The `removeAt` method is used to remove an item at a particular position in the list. The index or position of the item is passed to the `removeAt` method when it is called.



```
1 void main() {  
2   List names = ['Moses', 'Isaiah', 'Bright', 'Victor'];  
3   print('Initial items in list.');
```

4 print(names);  
5 // remove some names to the list using their index.  
6 names.removeAt(0);  
7 names.removeAt(2);  
8 print('After items were removed.');

9 print(names);  
10 }  
11

Run: main.dart x

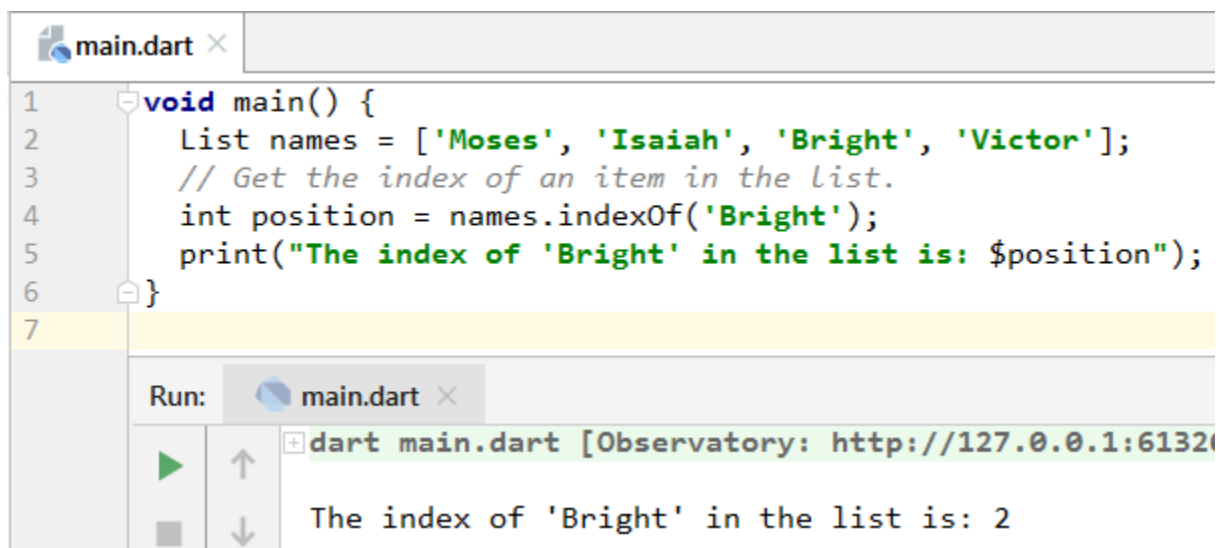
dart main.dart [Observatory: http://127.0.0.1:61320]

Initial items in list.  
[Moses, Isaiah, Bright, Victor]  
After items were removed.  
[Isaiah, Bright]

Screenshot 7.35

## indexOf

The *indexOf* method returns the index or position of an item in a list. The item is passed to the *indexOf* method when it is called.



```
1 void main() {  
2   List names = ['Moses', 'Isaiah', 'Bright', 'Victor'];  
3   // Get the index of an item in the list.  
4   int position = names.indexOf('Bright');  
5   print("The index of 'Bright' in the list is: $position");  
6 }  
7
```

Run: main.dart x

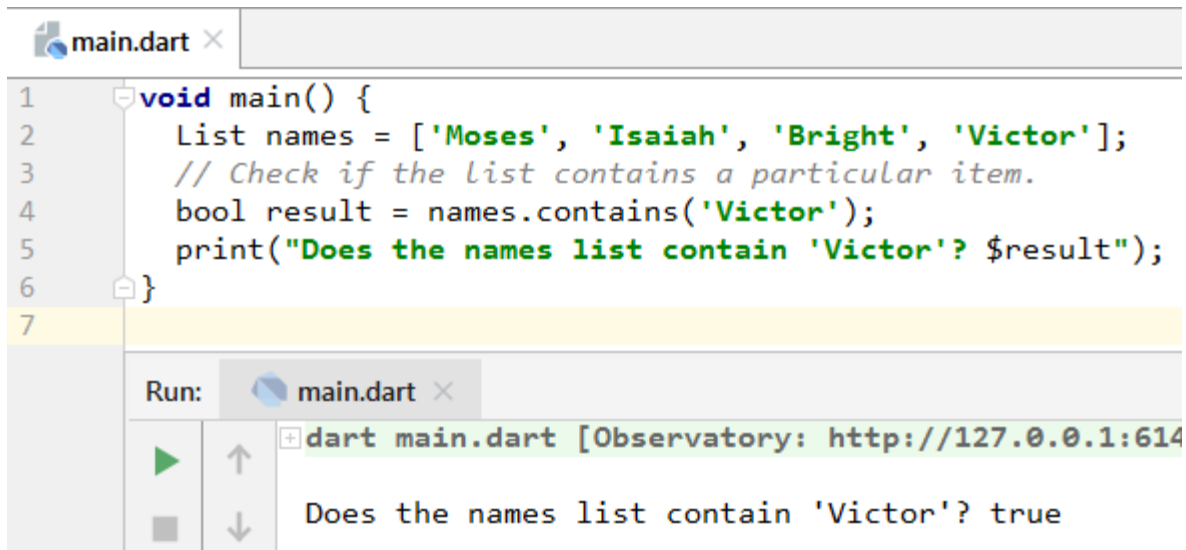
dart main.dart [Observatory: http://127.0.0.1:61320]

The index of 'Bright' in the list is: 2

Screenshot 7.36

## contains

The *contains* method returns **true** if a list contains a particular item, otherwise it returns **false**.



The screenshot shows an IDE window titled 'main.dart'. The code is as follows:

```
1 void main() {  
2   List names = ['Moses', 'Isaiah', 'Bright', 'Victor'];  
3   // Check if the list contains a particular item.  
4   bool result = names.contains('Victor');  
5   print("Does the names list contain 'Victor'? $result");  
6 }  
7
```

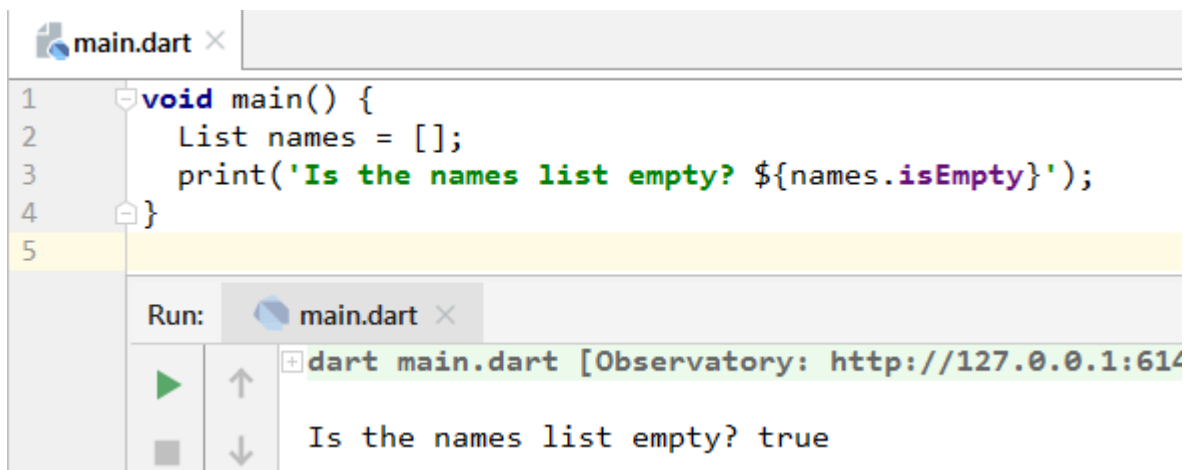
Below the code editor, the 'Run' button is clicked, and the output is displayed in a console window:

```
Run: main.dart  
+ dart main.dart [Observatory: http://127.0.0.1:614  
Does the names list contain 'Victor'? true
```

Screenshot 7.37

## isEmpty

The *isEmpty* property returns **false** if a list contains at least one (1) character, otherwise it returns **true**.



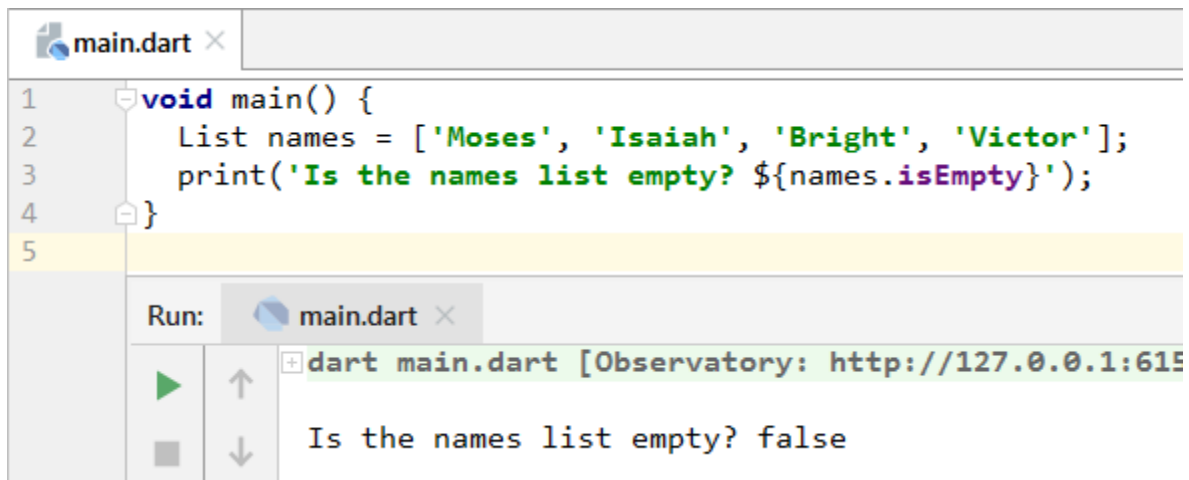
The screenshot shows an IDE window titled 'main.dart'. The code is as follows:

```
1 void main() {  
2   List names = [];  
3   print('Is the names list empty? ${names.isEmpty}');  
4 }  
5
```

Below the code editor, the 'Run' button is clicked, and the output is displayed in a console window:

```
Run: main.dart  
+ dart main.dart [Observatory: http://127.0.0.1:614  
Is the names list empty? true
```

Screenshot 7.38



```
main.dart x
1 void main() {
2   List names = ['Moses', 'Isaiah', 'Bright', 'Victor'];
3   print('Is the names list empty? ${names.isEmpty}');
4 }
5
```

Run: main.dart x

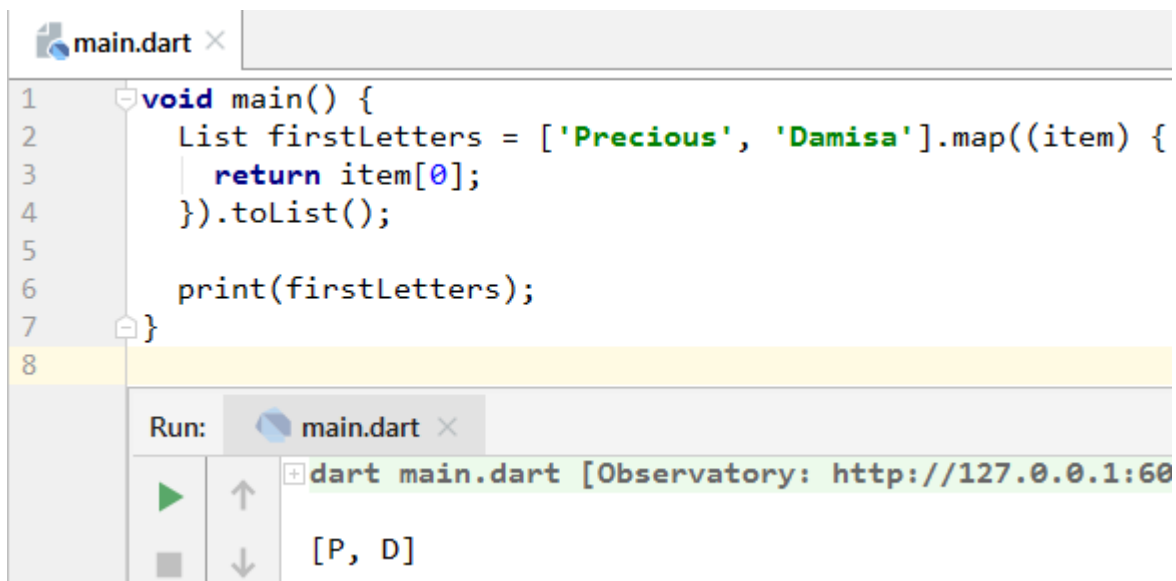
dart main.dart [Observatory: http://127.0.0.1:615

Is the names list empty? false

Screenshot 7.39

## map

The *map* method is used to transform the items in a list, by performing an operation on each item. The transformed items are put into a new list.



```
main.dart x
1 void main() {
2   List firstLetters = ['Precious', 'Damisa'].map((item) {
3     | return item[0];
4   }).toList();
5
6   print(firstLetters);
7 }
8
```

Run: main.dart x

dart main.dart [Observatory: http://127.0.0.1:60

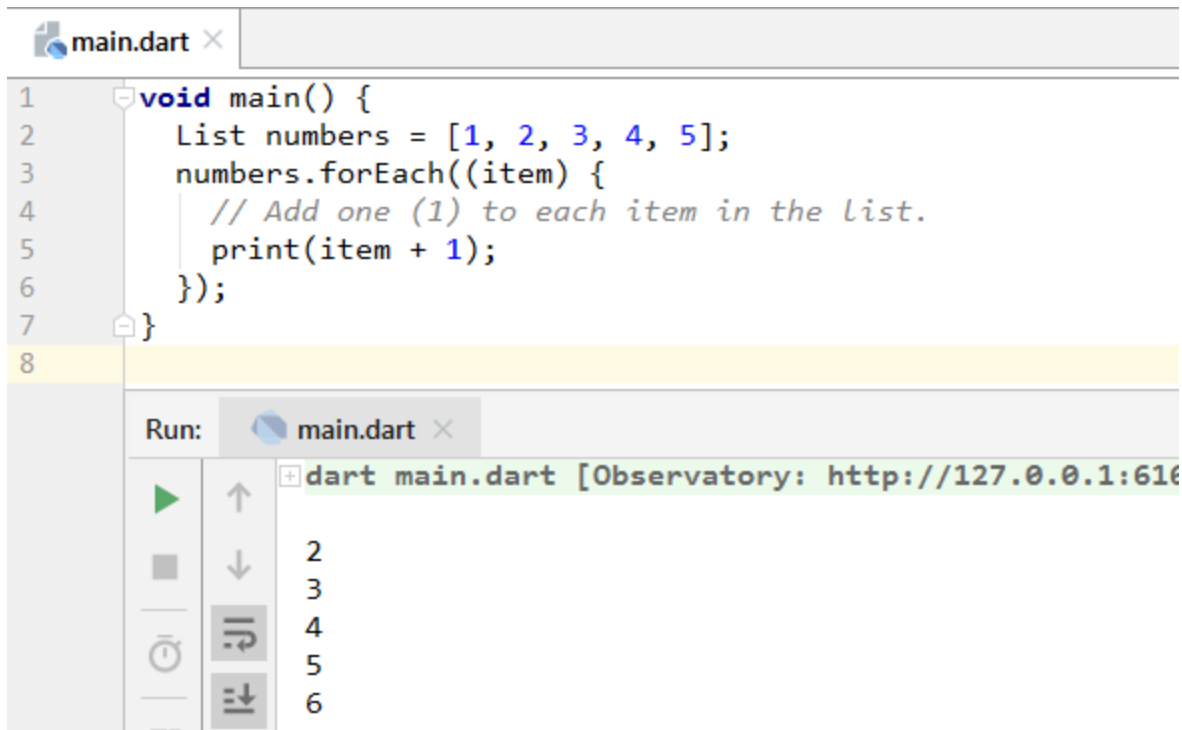
[P, D]

Screenshot 7.40

Notice that the *toList* method had to be called on what is returned by the *map* method. This is because, the *map* method returns an **Iterable**, which is a kind of list, but not the type of list we can deal with directly. So, we have to call the *toList* method on that **Iterable**, to turn it into a list that we can use directly.

## forEach

The *forEach* method can be used to apply an action to each item in a list. It loops through all the items in a list and performs a specified operation of each one.



The screenshot shows an IDE window titled 'main.dart' with the following code:

```
1 void main() {  
2   List numbers = [1, 2, 3, 4, 5];  
3   numbers.forEach((item) {  
4     // Add one (1) to each item in the list.  
5     print(item + 1);  
6   });  
7 }  
8
```

Below the code editor, the 'Run' tab is active, showing the command 'dart main.dart' and the Observatory URL 'http://127.0.0.1:616'. The output console displays the results of the `print` statements:

```
2  
3  
4  
5  
6
```

7.41

Screenshot

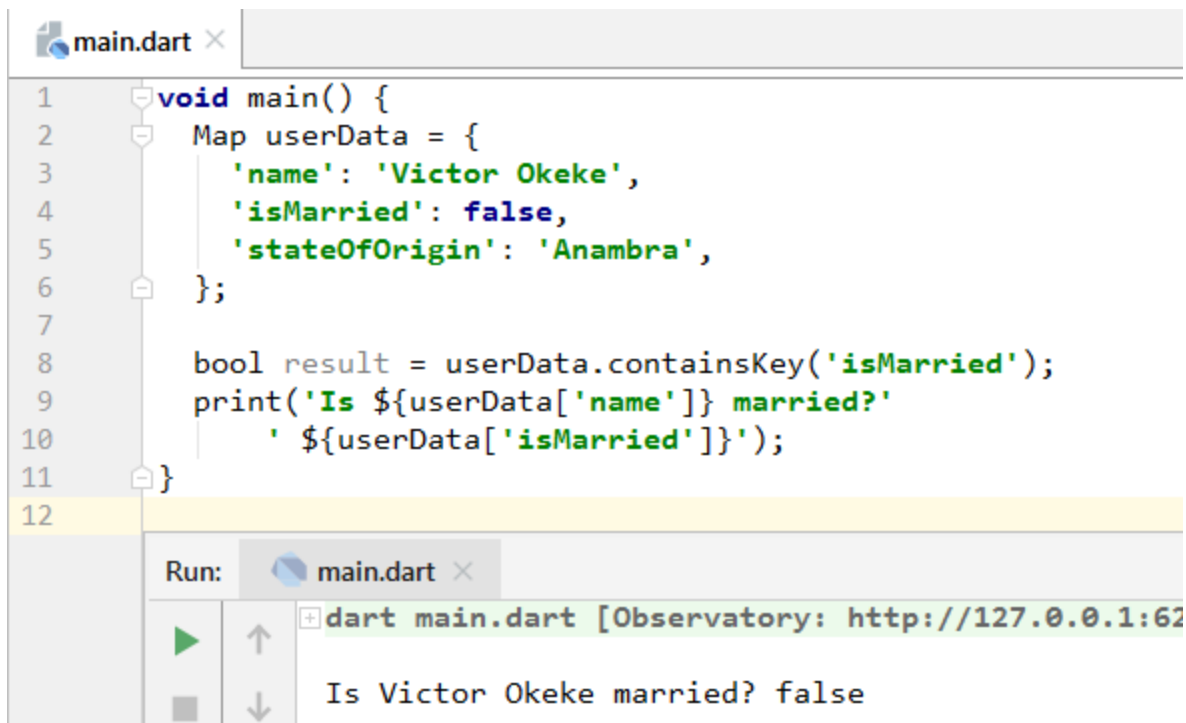
## Map

The *Map* type also defines some methods or operations that be performed on a *Map* object, some of which include:

1. `containsKey`
2. `containsValue`
3. `remove`

### containsKey

Just as you may have guessed it, the *containsKey* method checks if a map contains a key. If it does, it returns **true**, otherwise it returns **false**.



The screenshot shows an IDE window with a file named `main.dart`. The code defines a `main` function that creates a `Map` named `userData` with the following entries: `'name': 'Victor Okeke'`, `'isMarried': false`, and `'stateOfOrigin': 'Anambra'`. It then checks if the map contains the key `'isMarried'` using `containsKey` and prints the result. The output console shows the command `dart main.dart` and the result `Is Victor Okeke married? false`.

```
1 void main() {
2   Map userData = {
3     'name': 'Victor Okeke',
4     'isMarried': false,
5     'stateOfOrigin': 'Anambra',
6   };
7
8   bool result = userData.containsKey('isMarried');
9   print('Is ${userData['name']} married?'
10     ' ${userData['isMarried']}');
11 }
```

Run: `main.dart` ×

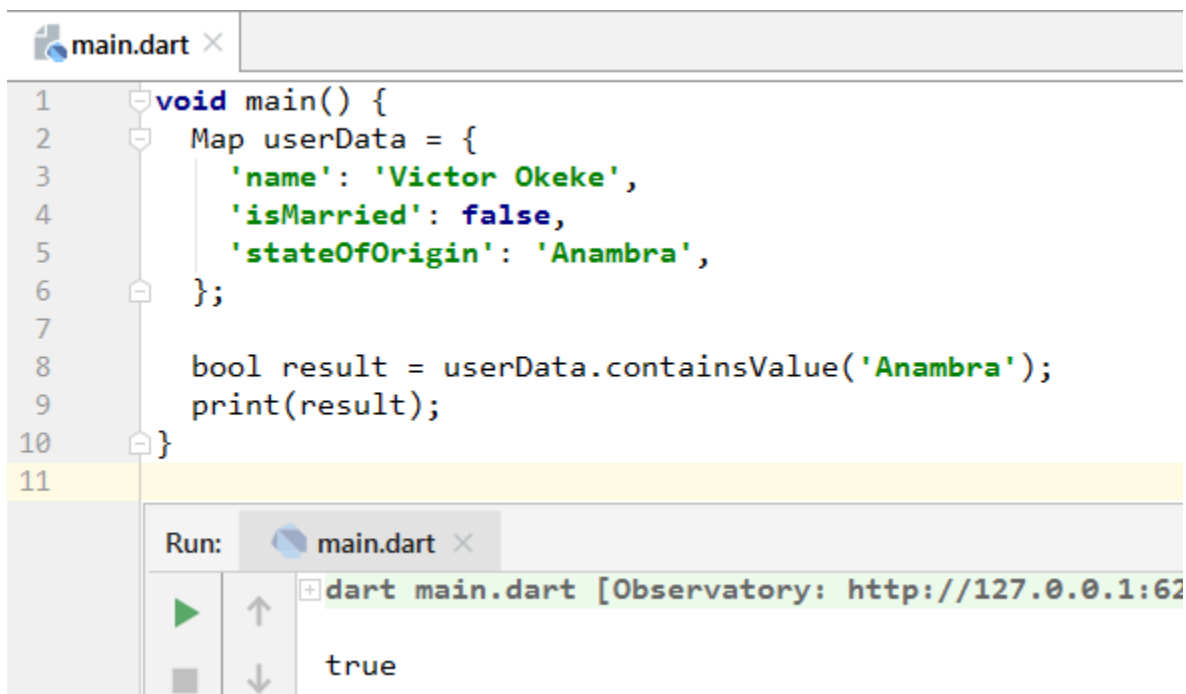
`dart main.dart` [Observatory: <http://127.0.0.1:62>]

Is Victor Okeke married? false

Screenshot 7.42

## containsValue

The `containsValue` method checks if a map contains a value. If it does, it returns **true**, otherwise it returns **false**.



The screenshot shows an IDE window with a file named `main.dart`. The code defines a `main` function that creates a `Map` named `userData` with the following entries: `'name': 'Victor Okeke'`, `'isMarried': false`, and `'stateOfOrigin': 'Anambra'`. It then checks if the map contains the value `'Anambra'` using `containsValue` and prints the result. The output console shows the command `dart main.dart` and the result `true`.

```
1 void main() {
2   Map userData = {
3     'name': 'Victor Okeke',
4     'isMarried': false,
5     'stateOfOrigin': 'Anambra',
6   };
7
8   bool result = userData.containsValue('Anambra');
9   print(result);
10 }
11
```

Run: `main.dart` ×

`dart main.dart` [Observatory: <http://127.0.0.1:62>]

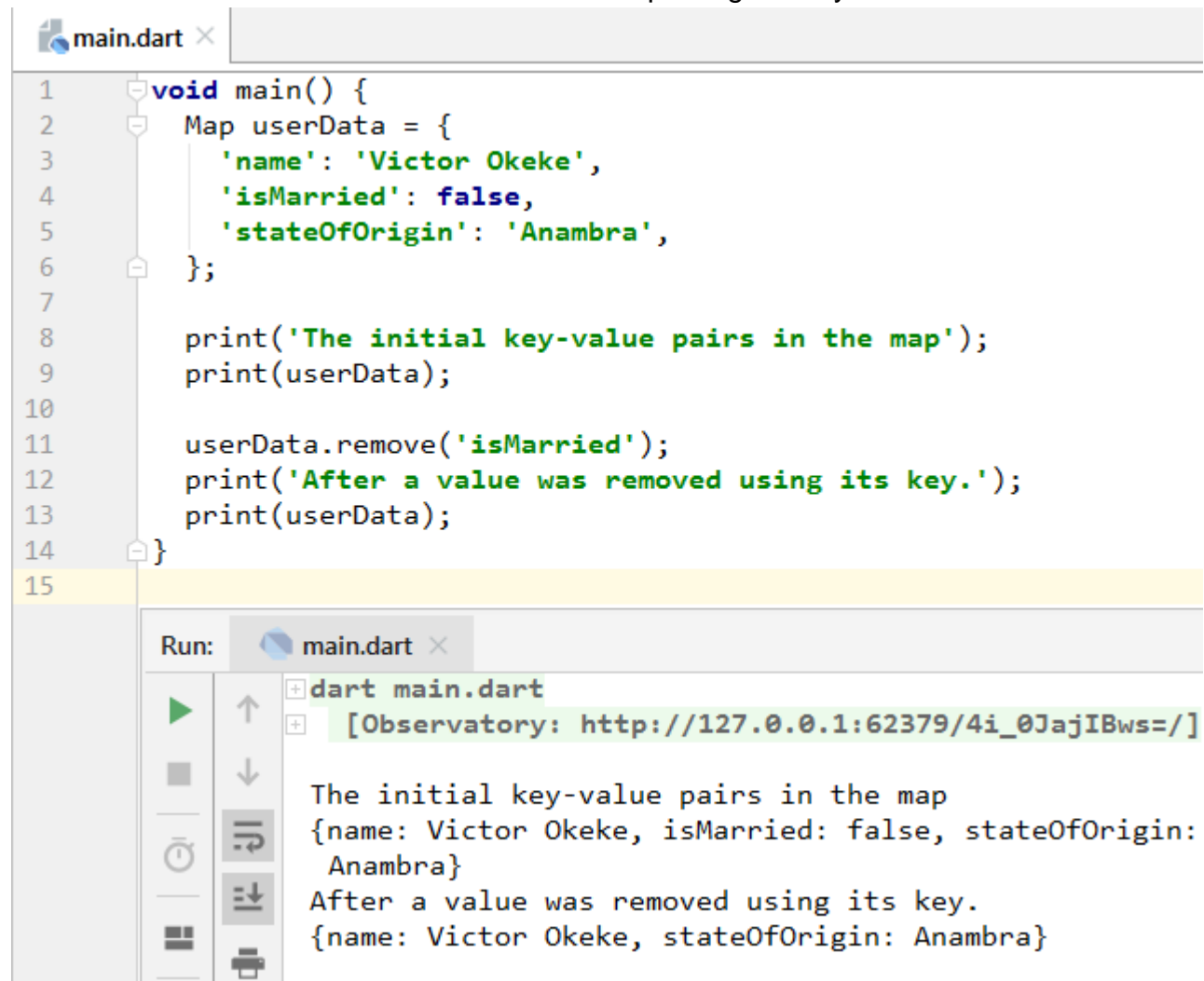
true



Screenshot 7.43

## remove

The *remove* method removes a value from a map using the key.



The screenshot shows an IDE window titled 'main.dart' with the following Dart code:

```
1 void main() {  
2   Map userData = {  
3     'name': 'Victor Okeke',  
4     'isMarried': false,  
5     'stateOfOrigin': 'Anambra',  
6   };  
7  
8   print('The initial key-value pairs in the map');  
9   print(userData);  
10  
11   userData.remove('isMarried');  
12   print('After a value was removed using its key.');
```

The code is executed, and the output is shown in the 'Run' console. The output is as follows:

```
dart main.dart  
[Observatory: http://127.0.0.1:62379/4i_0JajIBws=/]  
  
The initial key-value pairs in the map  
{name: Victor Okeke, isMarried: false, stateOfOrigin:  
  Anambra}  
After a value was removed using its key.  
{name: Victor Okeke, stateOfOrigin: Anambra}
```

Screenshot 7.44

We've explored some of the methods and operations that can be performed on the objects that are created using the inbuilt Dart types. Not all of them were covered, so I encourage you to take out time to try out each one and learn how it works.

One thing you may have observed is that some of the types have common methods and properties. Like the **length** property and the **toString** method for example. These methods and properties that are common to most of the inbuilt types and even the types (classes) you create, are all defined in the **Object** class. The **Object** class is the ancestor of every class, both the inbuilt classes and the classes created by you.

Every other class extends the Object class implicitly, thereby inheriting the methods and properties in the Object class. Although some classes have to override some of the methods they inherit from the object class, so as to provide a different implementation for the method.

## Chapter Eight

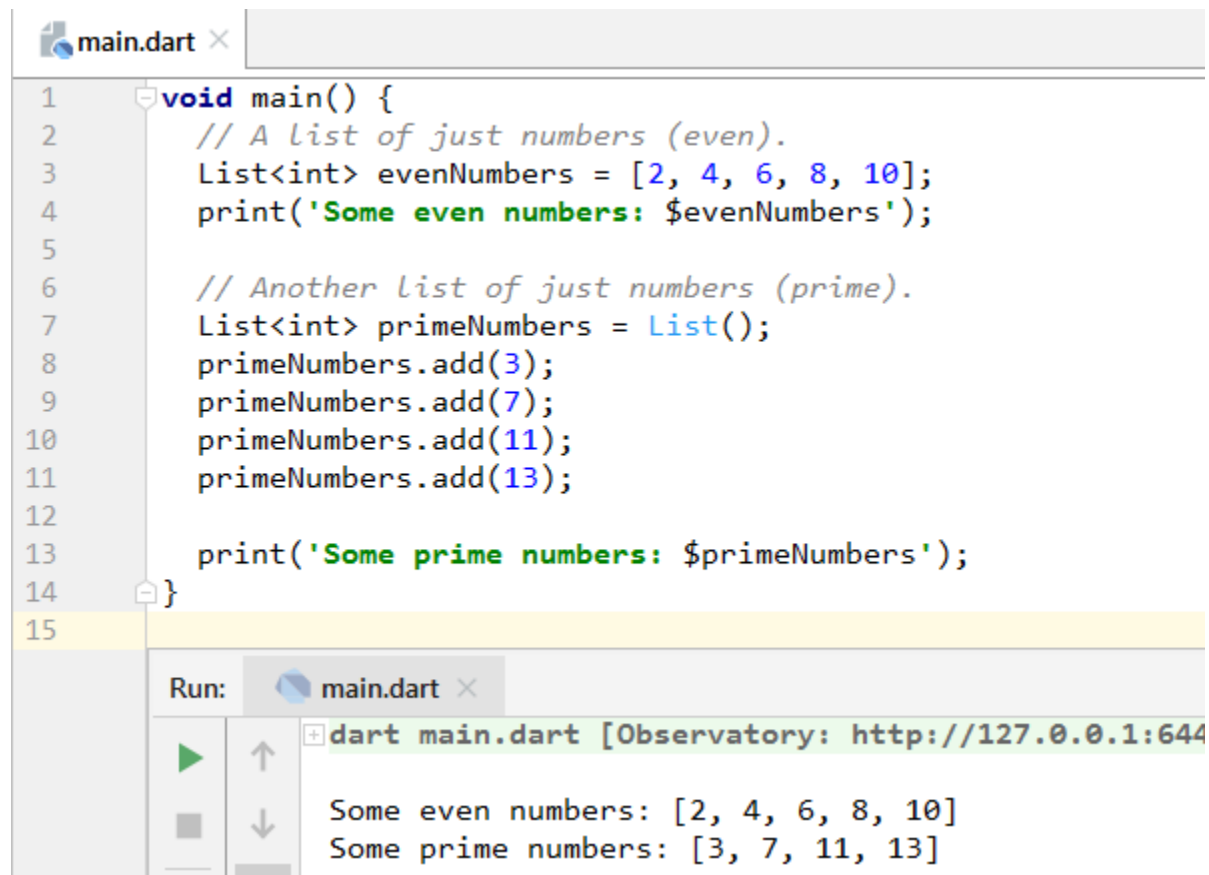
# Advanced Topics: Generics, enum, Exceptions and Asynchronicity

In this chapter, we shall look at some advanced topics of the Dart Programming language; Generics, Enum, Exceptions and Futures. A good understanding of these concepts is required to be able to write fluid code.

### Generics

In very simple terms, Generics are a way of writing code that is **type** safe. It is used to ensure type safety.

Remember when I first showed you how to define a list in chapter two (2). The lists we defined could contain items of any type (String, int, double, bool, etc.). With Generics, we can define a list that would contain only items of a specific type. Let's see an example on this.



```
1 void main() {
2   // A list of just numbers (even).
3   List<int> evenNumbers = [2, 4, 6, 8, 10];
4   print('Some even numbers: $evenNumbers');
5
6   // Another list of just numbers (prime).
7   List<int> primeNumbers = List();
8   primeNumbers.add(3);
9   primeNumbers.add(7);
10  primeNumbers.add(11);
11  primeNumbers.add(13);
12
13  print('Some prime numbers: $primeNumbers');
14 }
15
```

Run: main.dart x

dart main.dart [Observatory: http://127.0.0.1:644

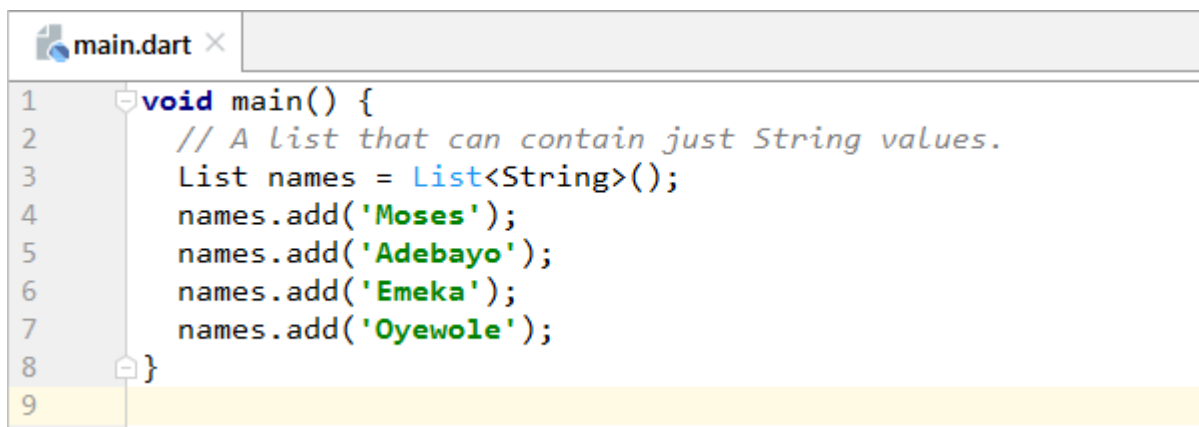
Some even numbers: [2, 4, 6, 8, 10]  
Some prime numbers: [3, 7, 11, 13]

## Screenshot 8.1

Observe how the *evenNumbers* list was defined. When a list is declared that way, it means that the list can only items of type **int**. So, it becomes impossible to add a different kind of item to the list, doing so would result in an error. **List<int>** is read as **List of type int**.

Also, observe how the *primeNumbers* list was defined on line 7. There, the list was initialized using the **list constructor**, which is another way of initializing a list. Although it doesn't provide a way of specifying initial items for the list, as opposed to when a list literal (**[]**), i.e. the square brackets is used in initializing a list.

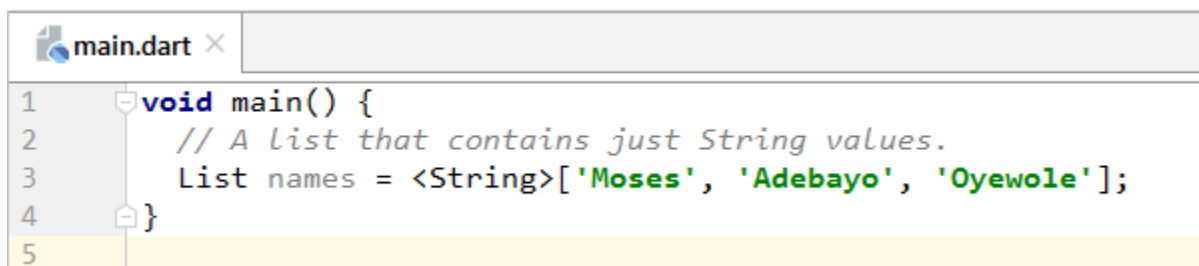
Other than creating lists that contain just integers, it is possible to create a list that contains items of any type. All that is required is to wrap the desired type with angle brackets. Here's a list that contains just String values.



```
main.dart x
1 void main() {
2   // A List that can contain just String values.
3   List names = List<String>();
4   names.add('Moses');
5   names.add('Adebayo');
6   names.add('Emeka');
7   names.add('Oyewole');
8 }
9
```

## Screenshot 8.2

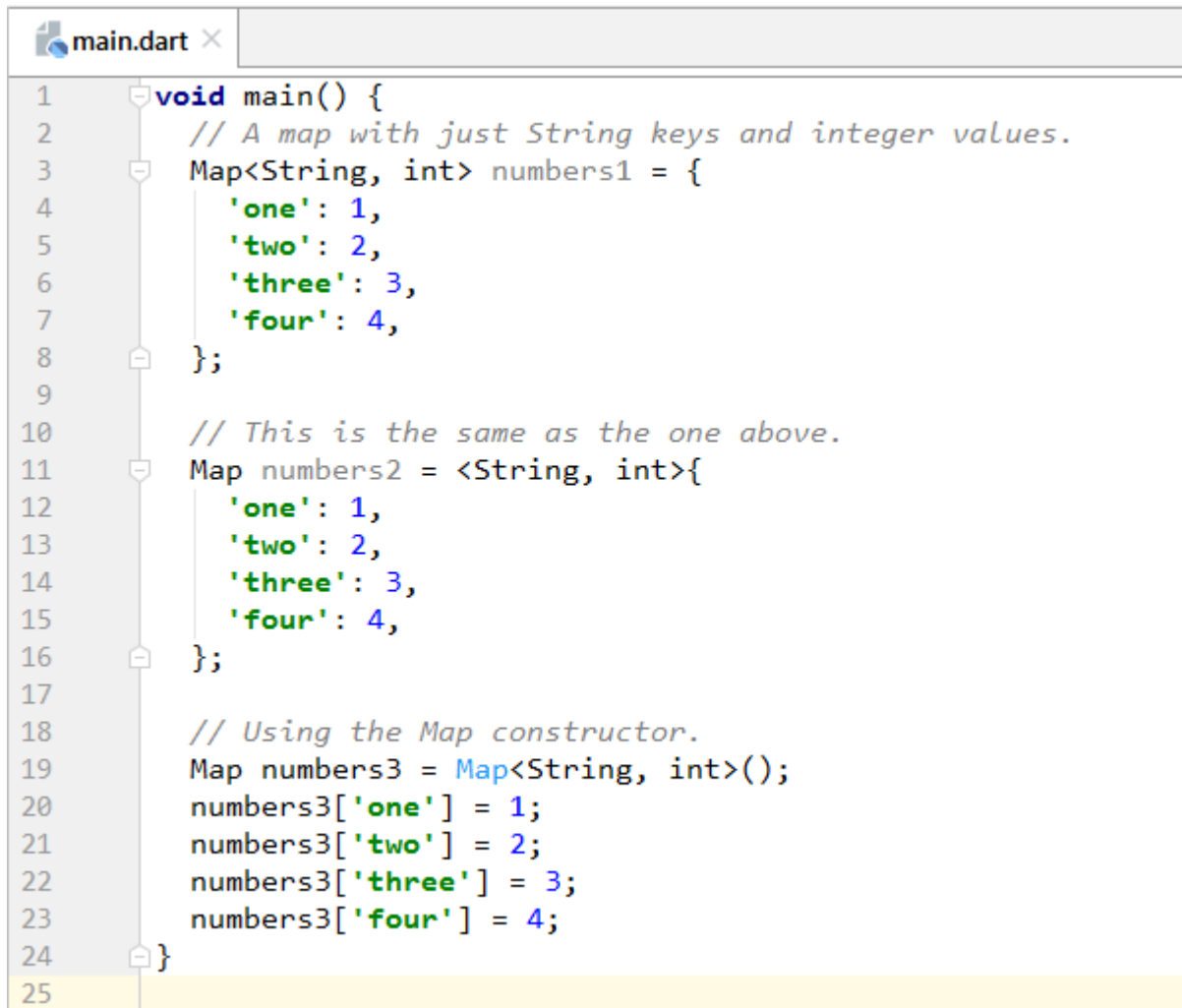
The declaration of the *names* list takes a different turn from how the previous lists were defined. Notice how the square brackets and the String type is attached to the List constructor call and not to the **List** type at the left. This is another valid syntax for using generics in defining a list. The same could be done when using the list literal (**[]**), i.e. the square brackets, the **<String>** would be added before the list literal, as shown below.



```
main.dart x
1 void main() {
2   // A List that contains just String values.
3   List names = <String>['Moses', 'Adebayo', 'Oyewole'];
4 }
5
```

## Screenshot 8.3

Generics can also be applied when defining a map. Although applying generics in the creation of maps is slightly different from that of a list. This is due to the fact that a map is made up of key-value pairs.

A screenshot of an IDE window titled 'main.dart'. The code defines a 'main' function with three map creation methods. Line 1: 'void main() {'; Line 2: '// A map with just String keys and integer values.'; Line 3: 'Map<String, int> numbers1 = {'; Line 4: ' 'one': 1,'; Line 5: ' 'two': 2,'; Line 6: ' 'three': 3,'; Line 7: ' 'four': 4,'; Line 8: '};'; Line 9: (blank); Line 10: '// This is the same as the one above.'; Line 11: 'Map numbers2 = <String, int>{'; Line 12: ' 'one': 1,'; Line 13: ' 'two': 2,'; Line 14: ' 'three': 3,'; Line 15: ' 'four': 4,'; Line 16: '};'; Line 17: (blank); Line 18: '// Using the Map constructor.'; Line 19: 'Map numbers3 = Map<String, int>()'; Line 20: 'numbers3['one'] = 1;'; Line 21: 'numbers3['two'] = 2;'; Line 22: 'numbers3['three'] = 3;'; Line 23: 'numbers3['four'] = 4;'; Line 24: '};'; Line 25: (blank).

```
1 void main() {
2   // A map with just String keys and integer values.
3   Map<String, int> numbers1 = {
4     'one': 1,
5     'two': 2,
6     'three': 3,
7     'four': 4,
8   };
9
10  // This is the same as the one above.
11  Map numbers2 = <String, int>{
12    'one': 1,
13    'two': 2,
14    'three': 3,
15    'four': 4,
16  };
17
18  // Using the Map constructor.
19  Map numbers3 = Map<String, int>();
20  numbers3['one'] = 1;
21  numbers3['two'] = 2;
22  numbers3['three'] = 3;
23  numbers3['four'] = 4;
24 }
25
```

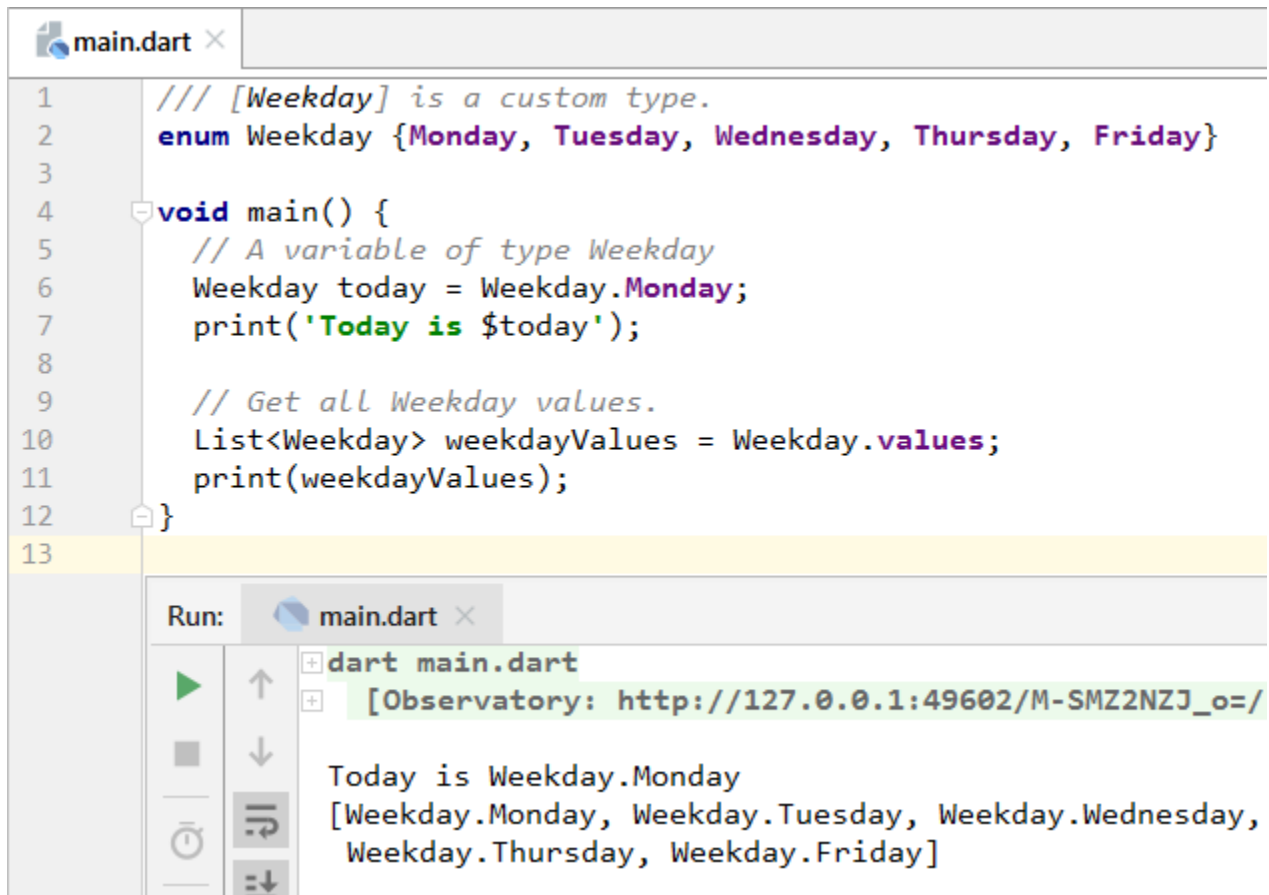
Screenshot 8.4

The maps in the above program contain just String keys and integer values. It is possible to define a map that contains keys of any type and values of any type. Although, it is common to find maps that have their keys as Strings.

## enum

enum provides a way for you to define your own custom data type. Remember the **bool** type, which is used to define variables that can contain only two kinds of values (true or false). Well, you could create your own custom type and provide a limited amount of

values that are allowed when it is used to define a variable. Let's see how that can be done.



The screenshot shows an IDE window titled 'main.dart'. The code defines an enum `Weekday` with values `Monday`, `Tuesday`, `Wednesday`, `Thursday`, and `Friday`. A `main` function is defined, which assigns `Weekday.Monday` to a variable `today` and prints it. It also retrieves all values of the `Weekday` enum using `Weekday.values` and prints them. Below the code editor, the 'Run' button is clicked, and the output console shows the execution results: 'Today is Weekday.Monday' and a list of all `Weekday` values.

```
1  /// [Weekday] is a custom type.
2  enum Weekday {Monday, Tuesday, Wednesday, Thursday, Friday}
3
4  void main() {
5      // A variable of type Weekday
6      Weekday today = Weekday.Monday;
7      print('Today is $today');
8
9      // Get all Weekday values.
10     List<Weekday> weekdayValues = Weekday.values;
11     print(weekdayValues);
12 }
13
```

Run: main.dart

dart main.dart  
[Observatory: [http://127.0.0.1:49602/M-SMZ2NZJ\\_o=/](http://127.0.0.1:49602/M-SMZ2NZJ_o=/)]

Today is Weekday.Monday  
[Weekday.Monday, Weekday.Tuesday, Weekday.Wednesday, Weekday.Thursday, Weekday.Friday]

Screenshot 8.4

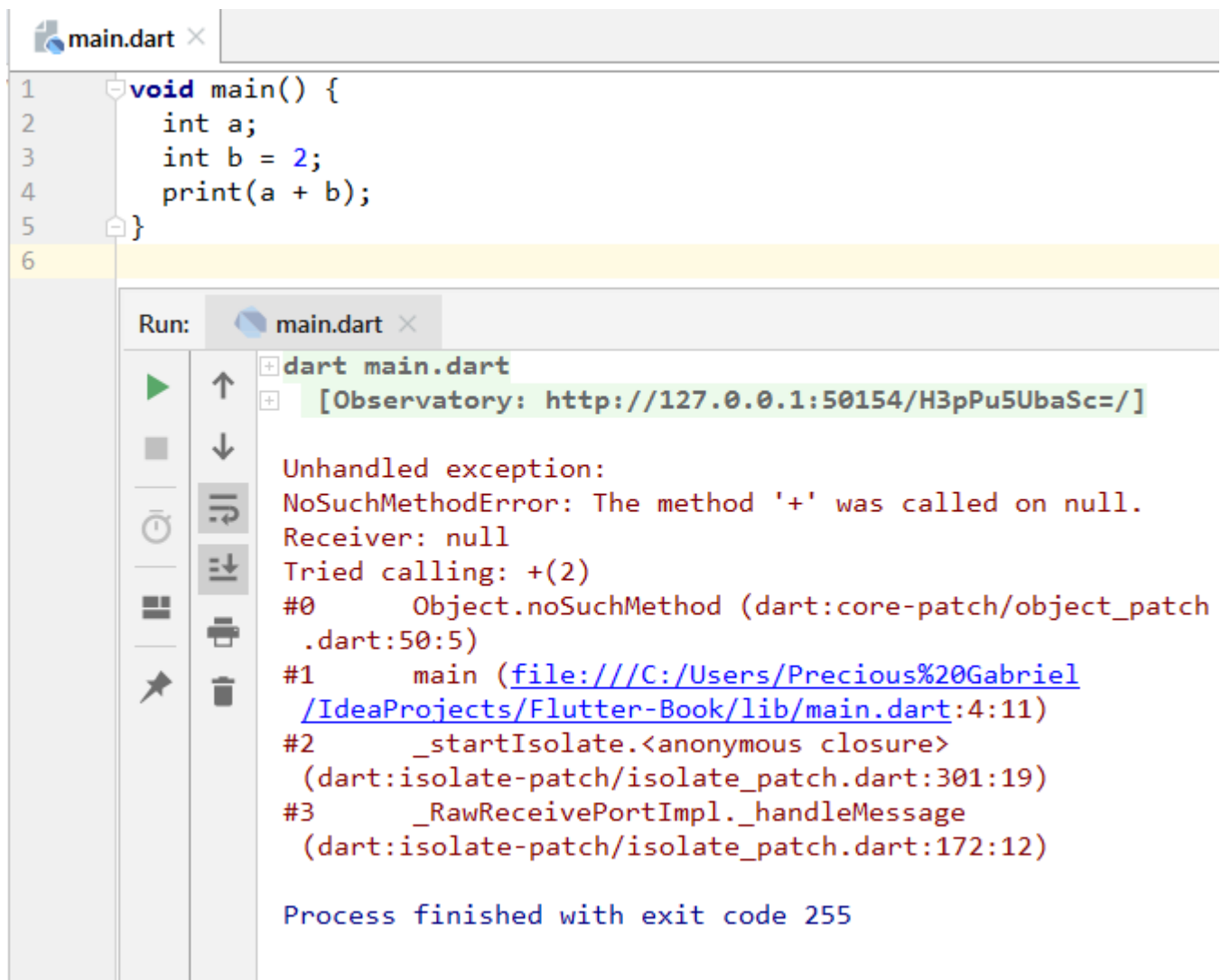
As shown above, the keyword **enum** is used when creating a custom type. In the example code, *Weekday* is the type, while *Monday*, *Tuesday*, *Wednesday*, *Thursday* and *Friday* are the only values that can be assigned to a variable of type *Weekday*. Observe how the value *Monday* is assigned to the *today* variable on line 6 in the program. Also, notice how all the values of the *Weekday* type are retrieved using the *Weekday.values* property.

The permitted values for a custom type created with enum, could be as many or as few as your program requires.

## Exceptions

Exceptions are basically errors, they're errors that could occur in a program at runtime. When such an error occurs, it could lead to an abrupt end of the program, if the error is not properly managed. A good example of an exception is using the addition operator (+) to add **null** to a number (e.g. 2).

One of the ways that Dart handles exceptions is the use of a try-catch block. Let's see an example on this.



```
main.dart x
1 void main() {
2   int a;
3   int b = 2;
4   print(a + b);
5 }
6

Run: main.dart x
+ dart main.dart
+ [Observatory: http://127.0.0.1:50154/H3pPu5UbaSc=/]

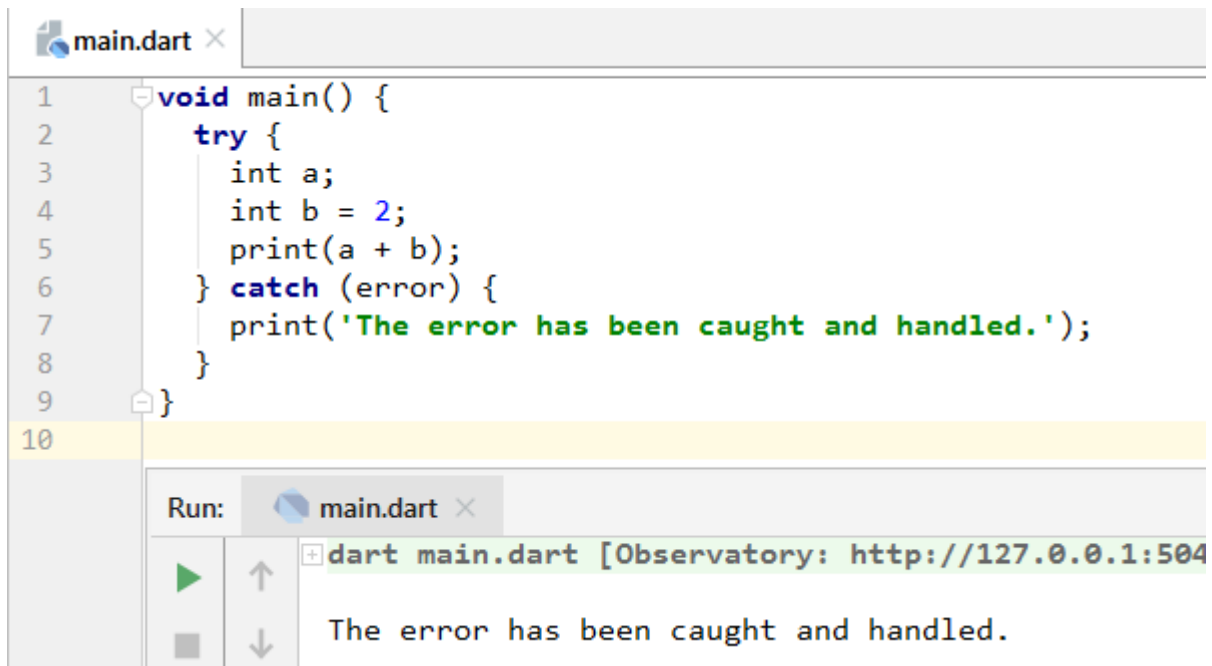
Unhandled exception:
NoSuchMethodError: The method '+' was called on null.
Receiver: null
Tried calling: +(2)
#0      Object.noSuchMethod (dart:core-patch/object_patch
.dart:50:5)
#1      main (file:///C:/Users/Precious%20Gabriel
/IdeaProjects/Flutter-Book/lib/main.dart:4:11)
#2      _startIsolate.<anonymous closure>
(dart:isolate-patch/isolate_patch.dart:301:19)
#3      _RawReceivePortImpl._handleMessage
(dart:isolate-patch/isolate_patch.dart:172:12)

Process finished with exit code 255
```

Screenshot 8.5

The program in the screenshot above contains an error that would manifest at execution time. Variable `b` is assigned the value `2`, while variable `a` isn't assigned any value, therefore it holds the default value `null`. On line 4, the program tries to add the values of these two variable together, but it is unable to, because it isn't possible to add `null` to an actual value. As a result, an exception is raised and the program crashes at that point.

A proper way to write code that may be subject to runtime errors is to wrap the code in a **try-catch** block. Doing so, makes it possible to properly handle any error that could occur while the program is executing. Let's rewrite the previous code using a **try-catch** block.



```
1 void main() {  
2   try {  
3     int a;  
4     int b = 2;  
5     print(a + b);  
6   } catch (error) {  
7     print('The error has been caught and handled.');8   }  
9 }  
10
```

Run: main.dart x

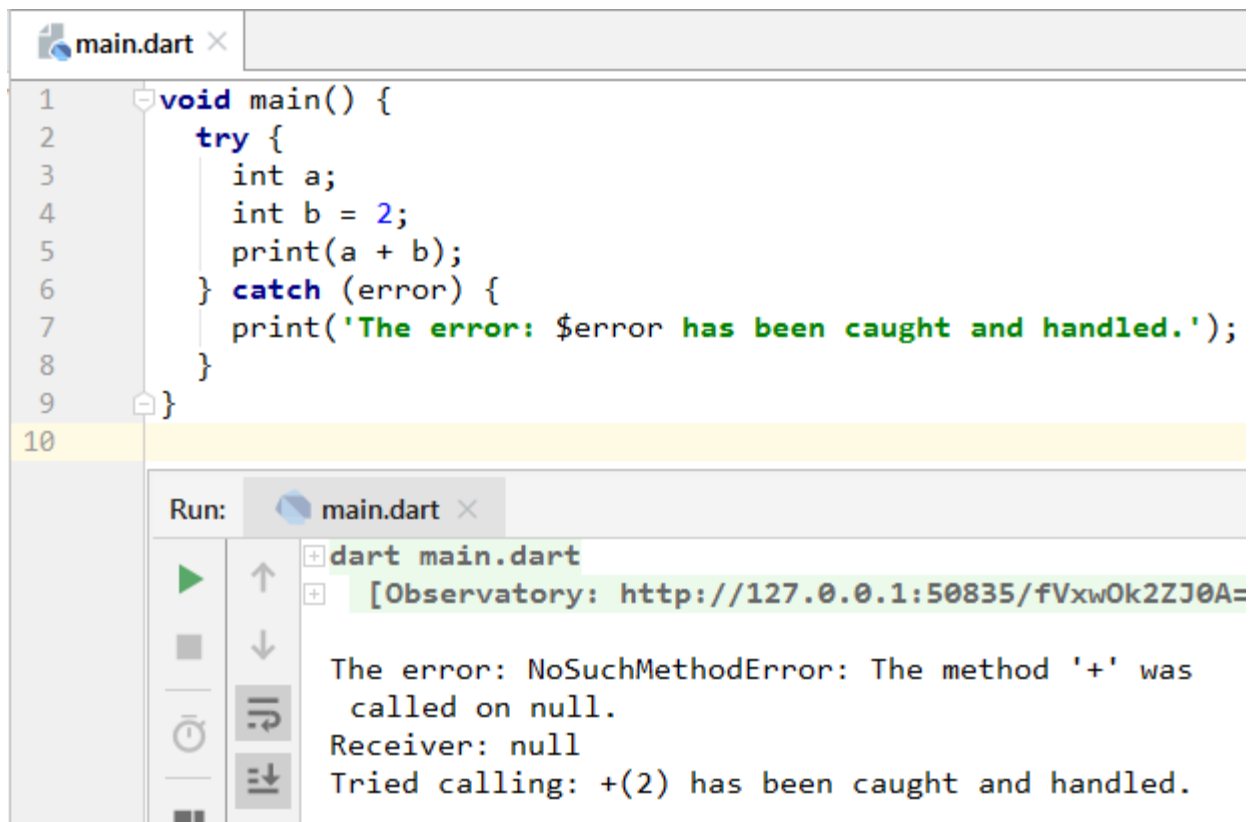
dart main.dart [Observatory: <http://127.0.0.1:504>]

The error has been caught and handled.

Screenshot 8.6

To use the **try-catch** block, wrap the code that an error may occur, inside the **try** block, then specify how you want to handle the possible error in the **catch** block. With the **try-catch** block, we're able to catch the error and handle it the way we want, thereby preventing the program from crashing. Here, I've simply printed a message to the console. You could define any appropriate handle for the error that occurs in your program. The **catch** block is called with the error that occurred from within the **try** block, so it is good to make use of the **error object** that gets passed when the error indeed happens, as shown below.





```
1 void main() {
2   try {
3     int a;
4     int b = 2;
5     print(a + b);
6   } catch (error) {
7     print('The error: $error has been caught and handled.');
```

Run: main.dart

dart main.dart  
[Observatory: <http://127.0.0.1:50835/fVxwOk2ZJ0A=>]

The error: NoSuchMethodError: The method '+' was called on null.  
Receiver: null  
Tried calling: +(2) has been caught and handled.

Screenshot 8.7

## Asynchronicity

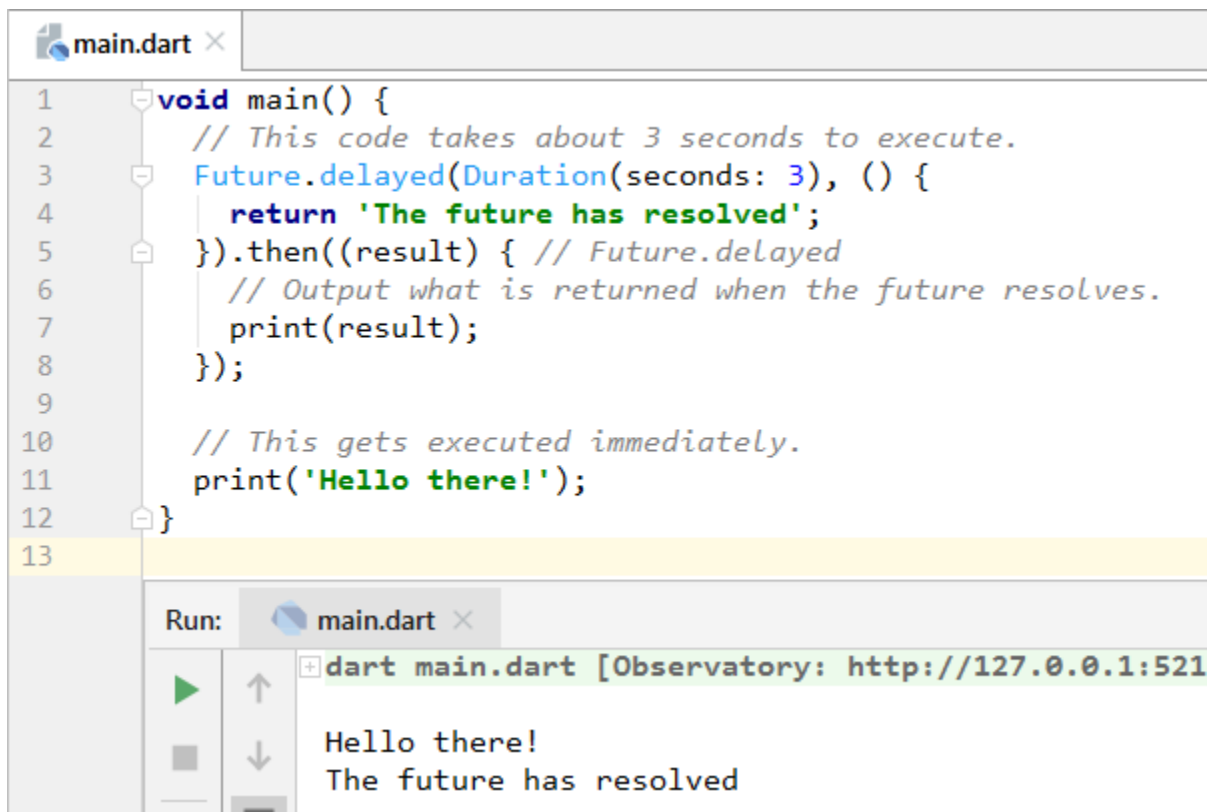
So far, we've only written code where each line of the code is executed immediately, before the next line, or each block gets executed before the next line or the next block. Such code is referred to as **synchronous code**. However, there are some programs or applications that contain code that takes time to execute. Such code is referred to as **asynchronous code**. A good example is an application that performs database access or network calls, or accesses some device features like the Camera, Barometer, Flash light, etc. The code to perform these operations wouldn't necessarily get executed immediately, it may take some time between when the request is made and a response gotten. Where the response could be the desired result, or an error.

Dart provides two ways of writing asynchronous code, which includes:

1. The Future class
2. `async/await` syntax.

## Future class

To demonstrate how to use the Future class in writing asynchronous code, I am going to write code that takes some time to execute.

The screenshot shows an IDE window with a file named 'main.dart'. The code is as follows:

```
1 void main() {  
2   // This code takes about 3 seconds to execute.  
3   Future.delayed(Duration(seconds: 3), () {  
4     return 'The future has resolved';  
5   }).then((result) { // Future.delayed  
6     // Output what is returned when the future resolves.  
7     print(result);  
8   });  
9  
10  // This gets executed immediately.  
11  print('Hello there!');  
12 }  
13
```

Below the code editor, there is a 'Run' button and a console window. The console shows the output of the program:

```
dart main.dart [Observatory: http://127.0.0.1:521  
Hello there!  
The future has resolved
```

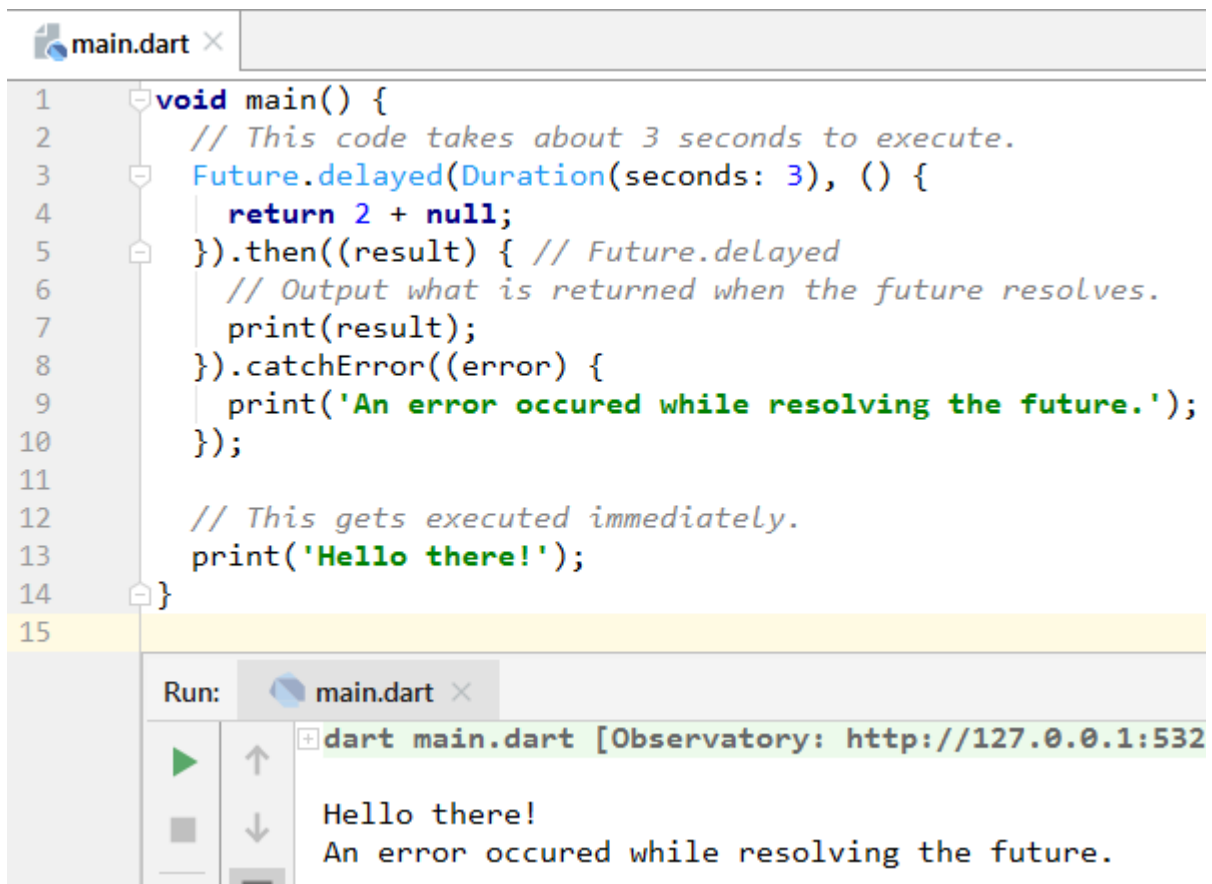
Screenshot 8.8

The program in the screenshot above uses the *Future.delayed* constructor, which is a named constructor of the *Future* class, to simulate a time taking process. The *Future.delayed* constructor takes two required arguments. First is the amount of time it would take before the future resolves (produce a result or an error). The time is specified using a **Duration** object, which accepts time in microseconds, milliseconds, seconds, minutes, hours, and days. Here, I chose just 3 seconds. Meaning, it would take about 3 seconds before the future resolves with a result or an error. The second argument is a function that would get called when the specified duration (3 seconds in this case) is reached. Here, I'm simply returning a *String* value, which would serve as the result of the future, when it eventually resolves.

The *then* function is used in retrieving the result of the future, when it has resolved. Notice that a callback function that has a parameter (in this case *result*) is passed to the *then* function as an argument. The callback function would be called by the *then* function, once the future resolves with a valid result.

In the console view, notice that the *print* function that outputs the string *Hello there!*, is printed executed before the result of the future is outputted by the callback function. The reason why this happens would be explained when we look at how to write asynchronous code using the **async/await** syntax.

A future may not always resolve with a desired result, sometimes an error may occur. To handle such an error, the Future class provides the `catchError` function which can be called on a Future object, to handle whichever error that arises while the future is being resolved.



The screenshot shows an IDE window titled 'main.dart'. The code is as follows:

```
1 void main() {  
2   // This code takes about 3 seconds to execute.  
3   Future.delayed(Duration(seconds: 3), () {  
4     return 2 + null;  
5   }).then((result) { // Future.delayed  
6     // Output what is returned when the future resolves.  
7     print(result);  
8   }).catchError((error) {  
9     print('An error occurred while resolving the future.');10  });  
11  
12  // This gets executed immediately.  
13  print('Hello there!');  
14 }  
15
```

Below the code editor is a 'Run:' panel. It shows the command 'dart main.dart [Observatory: http://127.0.0.1:532]' and the output:

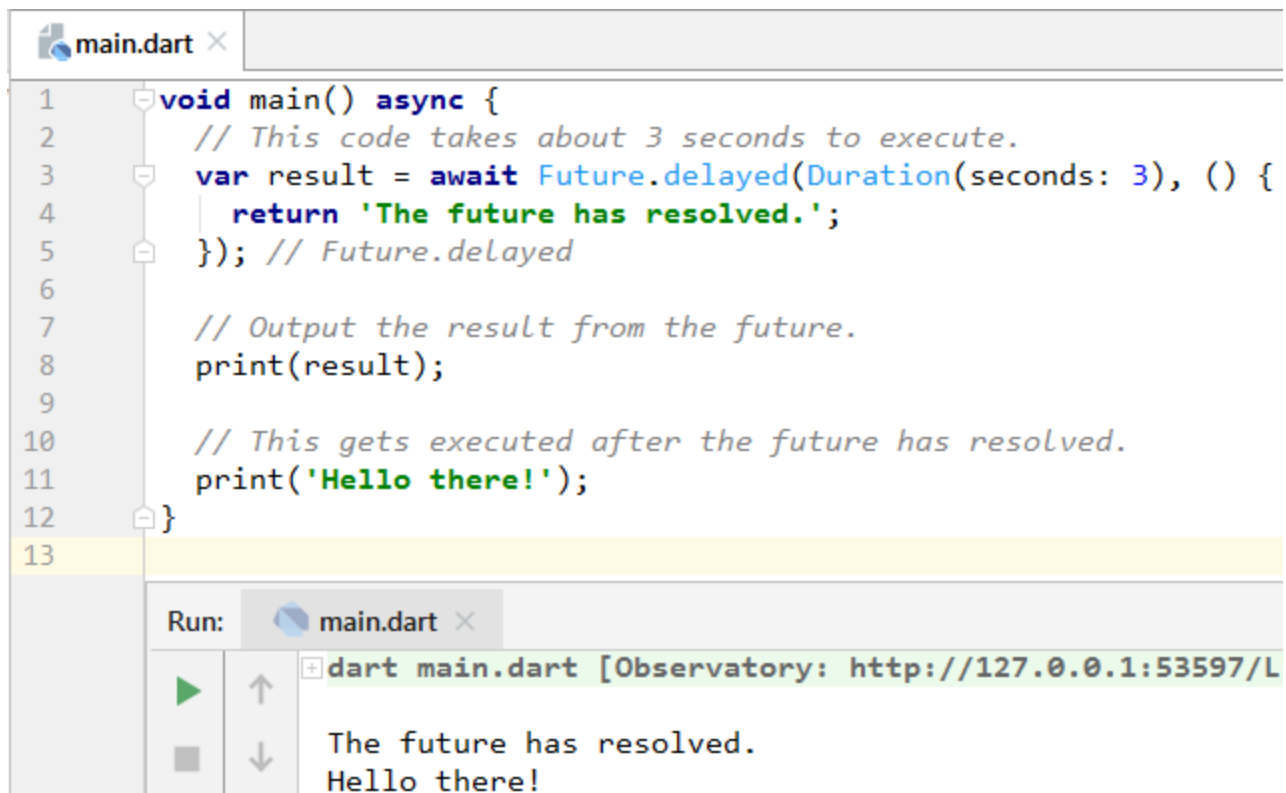
```
Hello there!  
An error occurred while resolving the future.
```

Screenshot 8.9

Remember that the operation of adding a number (e.g. 2) to **null** results in an error. When the error occurs, the future resolves and yields an error, which is caught by the `catchError` function and passed as an argument to a callback function. Here, I've handled the error by printing a simple message. Observe that the `then` function doesn't get called, because the future resolved with an error, not a result.

## async/await

Asynchronous code can also be written using the `async/await` syntax. The `async/await` syntax as you would see, provides a simpler and more readable way for writing asynchronous code.



The screenshot shows an IDE window titled 'main.dart'. The code is as follows:

```
1 void main() async {  
2   // This code takes about 3 seconds to execute.  
3   var result = await Future.delayed(Duration(seconds: 3), () {  
4     return 'The future has resolved.';  
5   }); // Future.delayed  
6  
7   // Output the result from the future.  
8   print(result);  
9  
10  // This gets executed after the future has resolved.  
11  print('Hello there!');  
12 }  
13
```

Below the code editor, there is a 'Run' button and a console window. The console shows the output of the program:

```
dart main.dart [Observatory: http://127.0.0.1:53597/L  
The future has resolved.  
Hello there!
```

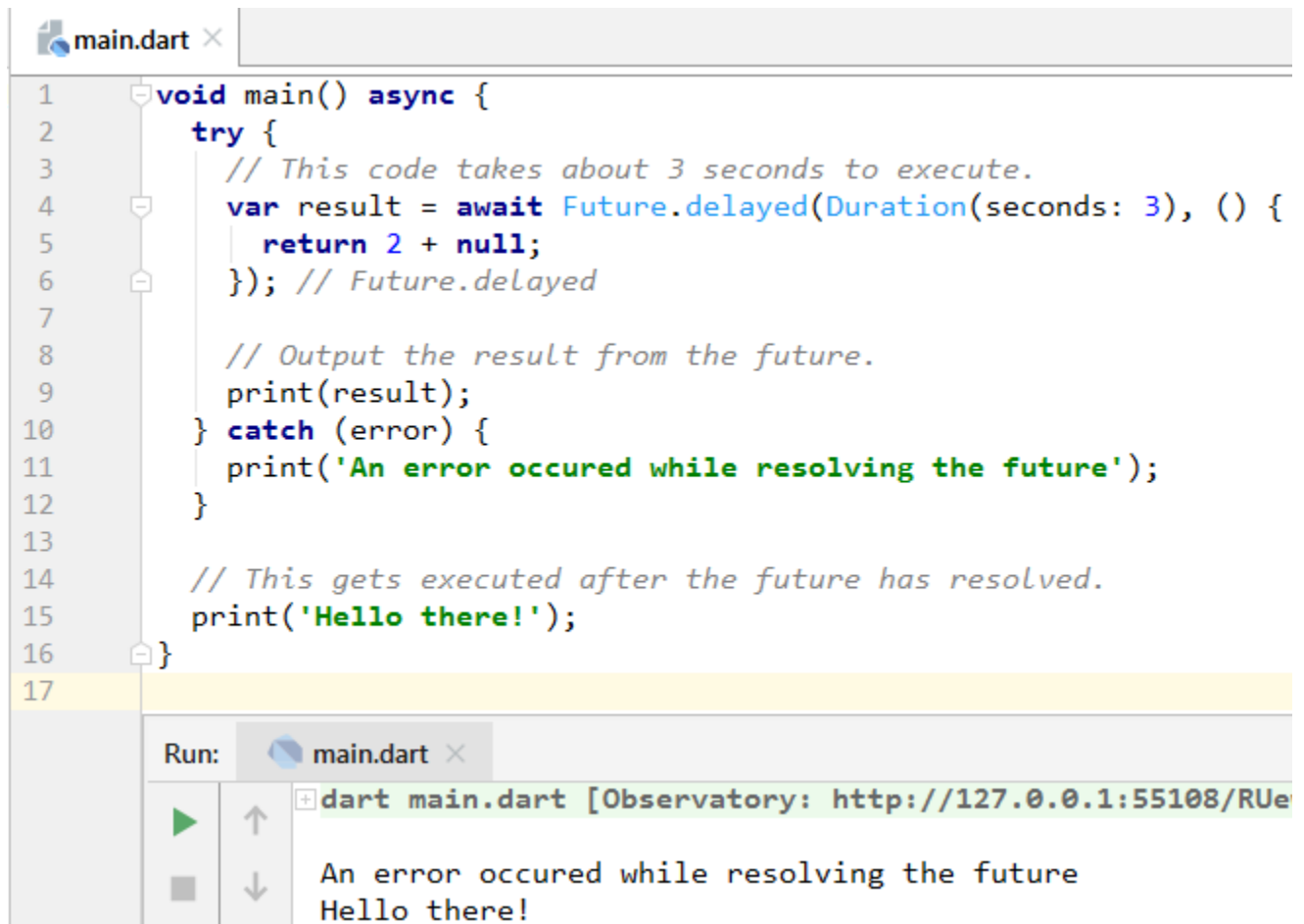
Screenshot 8.10

The asynchronous code in screenshot 8.8 has been rewritten using the `async/await` syntax. Notice that on line 1, the keyword **async** is placed before the body of the `main` function. It is required that any function that uses **await**, must have its body decorated with the **async** keyword. On line 3, the keyword **await** is used in preceding the call to the `Future.delayed` constructor. The effect of that is to pause the execution of the program, till the future resolves. All the code that comes after it won't be executed until the future resolves. What that means is that the print function on line 8 that prints the result of the future won't be executed, as well as the print function that prints *Hello there!*

That is one of the differences between using the `Future` class and using the `async/await` syntax.

When the `Future` Class is used (i.e. using the `then` function to get the result of the asynchronous operation), the code that comes after the `then` function is executed even though the future has not resolved. It is only the code that is inside the callback function that is passed to the `then` function that isn't executed, until the future resolves. While for `async/await`, the code that comes after **await**, doesn't get executed, until a result is gotten from the asynchronous operation.

When using `async/await`, in order to catch and handle errors that may occur from an asynchronous operation, the code that uses **await** is wrapped in a try-catch block, as shown below.



The screenshot shows an IDE window with a file named `main.dart`. The code is as follows:

```
1 void main() async {  
2   try {  
3     // This code takes about 3 seconds to execute.  
4     var result = await Future.delayed(Duration(seconds: 3), () {  
5       return 2 + null;  
6     }); // Future.delayed  
7  
8     // Output the result from the future.  
9     print(result);  
10  } catch (error) {  
11    print('An error occurred while resolving the future');  
12  }  
13  
14  // This gets executed after the future has resolved.  
15  print('Hello there!');  
16 }  
17
```

Below the code editor, there is a 'Run' button and a console window. The console shows the output of the program:

```
Run: main.dart  
dart main.dart [Observatory: http://127.0.0.1:55108/RUE  
An error occurred while resolving the future  
Hello there!
```

Screenshot 8.11

Now, any error that arises during the asynchronous operation, can be caught and handled.