Chapter 7: Nullability

You know that game where you try to find the item that doesn't belong in a list? Here's one for you:

```
horse, camel, pig, cow, sheep, goat
```

Which one doesn't belong?

It's the third one, of course! The other animals are raised by nomadic peoples, but a pig is a farmer's animal — it doesn't do so well trekking across the steppe. About now you're probably muttering to yourself why your answer was just as good — like, a sheep is the only animal with wool, or something similar. If you got an answer that works, good job. Here's another one:

```
196, 144, 169, 182, 121
```

Did you get it? The answer is one hundred and eighty-two. All the other numbers are squares of integers.

One more:

```
3, null, 1, 7, 4, 5
```

And the answer is . . . `null`! All of the other items in the list are integers, but `null` isn't an integer.

What? Was that too easy?

Null overview

As out of place as `null` looks in that list of integers, many computer languages actually include it. In the past Dart did, too, but as of version 2.12, Dart decided to take `null` out of the list and only put it back if you allow Dart to do so. This feature is called **sound null safety**, but to find out what was so dangerous about `null` in the first place, you'll have to keep reading.

What null means

Null means "no value" or "absence of a value". It's quite useful to have such a concept. Imagine not having null at all. Say you ask a user for their postal code so that you can save it as an integer in your program:

```
int postalCode = 12345;
```

Everything will go fine until you get a user who doesn't have a postal code. Your program requires some value, though, so what do you give it? Maybe `0` or `-1`?

```
int postalCode = -1;
```

Choosing a number like `-1`, though, is somewhat arbitrary. You have to define it yourself to mean "no value" and then tell other people that's what it means.

```
// Hey everybody, -1 means that the user
// doesn't have a postal code. Don't forget!
int postalCode = -1;
```

On the other hand, if you can have a dedicated value called `null`, which everyone already understands to mean "no value", then you don't need to add comments explaining what it means.

```
int postalCode = null;
```

It's obvious here that there's no postal code. In versions of Dart prior to 2.12 that line of code worked just fine. However, now it's no longer allowed. You get the following error:

```
A value of type 'Null' can't be assigned to a va
riable of type 'int'.
```

What's wrong? Null is a useful concept to have! Why not allow it, Dart?

The problem with null

As useful as `null` is for indicating the absence of a value, developers do have a problem with it. The problem is that they tend to forget that it exists. And when developers forget about `null`, they don't handle it in their code. Those nulls are like little ticking time bombs ready to explode the code. To see that in action, how about taking a trip back in time. Open **pubspec.yaml** and set the minimum Dart SDK version to 2.10:

```
environment:

sdk: ">=2.10.0 <3.0.0"
```

Save the file and run `dart pub get` if needed.

Dart 2.10 was the version before sound null safety was introduced in Dart. This will allow you to see first hand what a rogue `null` can do.
Now open **starter.dart** or whatever file you have your `main` function in, and replace the contents of the file with the following code:

```
void main() {
print(isPositive(3));        // true
print(isPositive(-1)); // false

}

bool isPositive(int anInteger) {
return !anInteger.isNegative;

}
```

Run that code and you'll get a result of `true` and `false` as expected. The `isPositive` method works fine as long as you give it integers. But what if you give it `null`?

Add the following line to the bottom of the `main` function:

```
print(isPositive(null));
```

Run that and your program will crash with the following error:

```
NoSuchMethodError: The getter 'isNegative' was c
alled on null.
```

You learned above that null means "no value", which is true, semantically. However, the Dart keyword `null` actually is a value in the sense that it's an object. That is, the object `null` is the sole instance of the `Null` class. Because the `Null` class doesn't have a method called `isNegative`, you get a `NoSuchMethodError` when you try to call `null.isNegative`. Now go back to **pubspec.yaml** and change the minimum Dart SDK version to 2.12, the first version that supported sound null safety:

```
environment:

sdk: ">=2.12.0 <3.0.0"
```

Save the file and again run `dart pub get` if needed.

Now all of a sudden you have an error where you tried to call your function with `null`:

With the advent of Dart's sound null safety, you *can't* assign a `null` value to an `int` even if you wanted to. Eliminating the possibility of being surprised by `null` prevents a whole host of errors.

Delete that line with the null error. Problem solved.

But wait? Isn't null useful? What about a missing postal code?

Yes, null is useful and Dart has a solution.

> **Note**: As you saw above, Dart only applies sound null safety checks to your project if you set the minimum version to 2.12. While this isn't required, the benefits for preventing null errors make it highly recommended.

Nullable vs. non-nullable types

Starting with version 2.12, Dart separated its types into nullable and non-nullable. Nullable types end with a question mark (?) while non-nullable types do not.

Non-nullable types

Dart types are **non-nullable by default**. That means they're *guaranteed* to never contain the value `null`, which is the essence of the meaning of **sound** in the phrase "sound null safety". These types are easy to recognize because, unlike nullable types, they don't have a question mark at the end.

Here are some example values that non-nullable types could contain:

`int`: 3, 1, 7, 4, 5

`double`: 3.14159265, 0.001, 100.5

bool: true, false
String: "a", "hello", "Would you like fries with that?"
User: ray, vicki, anonymous

These are all acceptable ways to set the values:

```
int myInt = 1;
double myDouble = 3.14159265; bool
myBool = true;
String myString = "Hello, Dart!";
```

As you saw earlier, trying to set a non-nullable type to null is a compile-time error:

```
int postalCode = null; // error
```

Since only types that end in a question mark can potentially have a null value, every time you see a type without a question mark, you can be absolutely positive that it won't ever be null.

Note: Well, a type without a question mark *could* be null if you use the late keyword, but technically this is opting out of sound null safety. It's also completely under your control. You'll learn about the late keyword at the end of this chapter.

Nullable types

A **nullable type** can contain the `null` value in addition to its own data type. You can easily tell the type is nullable because it ends with a question mark (?), which is like saying, "Maybe you've got the data you want or maybe you've got `null`. That's the question." Here are some example values that nullable types could contain:

`int?: 3, null, 1, 7, 4, 5`

`double?: 3.14159265, 0.001, 100.5, null`

`bool?: true, false, null`

`String?: "a", "hello", "Would you like fries with that?", null`

`User?: ray, vicki, anonymous, null`

That means you can set any of them to `null`:

```
int? myInt = null; double?
myDouble = null; bool?
myBool = null; String?
myString = null; User?
myUser = null;
```

The question mark at the end of `String?` isn't an operator acting on the `String` type. Rather, `String?` is a whole new type separate from `String`. `String?` means that the variable can either contain a `String` or it can be `null`.

Every non-nullable type in Dart has a corresponding nullable type: `int` and `int?`, `bool` and `bool?`, `User` and `User?`, `Object` and `Object?`. By choosing the type, you get to choose when you want to allow null values and when you don't.

> **Note**: The non-nullable type is a subtype of its nullable form. For example, `String` is a subtype of `String?` since `String?` can be a `String`.

For any nullable variable in Dart, if you don't initialize it with a value, it'll be given the default value of `null`.

Create three variables of different nullable types:

```
int? age;
double? height;

String?  message;
```

Then print them:

```
print(age); print(height);
print(message);
```

You'll see `null` for each value.

Mini-exercises

1. Create a `String?` variable called `profession`, but don't give it a value. Then you'll have `profession` `null`. :]

2. Give `profession` a value of "basketball player".

3. Write the following line and then hover your cursor over the variable name. What type does Dart infer `iLove` to be?

```
const iLove = "Dart";
```

Handling nullable types

The big problem with the old nullable types in the past was how easy it was to forget to add code to handle `null` values. That's no longer true. Dart now makes it impossible to forget because you really can't do much at all with a nullable value until you've dealt with the possibility of `null`.

Try out this example:

```
String? name;
print(name.length);
```

Dart doesn't let you run that code, so there isn't even an opportunity to get a runtime `NoSuchMethodError` like before. Instead, Dart gives you a compile-time error:

```
The property 'length' can't be unconditionally a
ccessed because the receiver can be 'null'.
```

Compile-time errors are your friends because they're easy to fix. In the next few sections you'll see how to use the many tools Dart has to deal with null values.

Type promotion

The **Dart analyzer**, which is the tool that tells you what the compile-time errors and warning are, is smart enough to tell in a wide range of situations if a nullable variable is guaranteed to contain a non-null value or not.

Take the last example, but this time assign `name` a string literal on the line after declaring it:

```
String? name;
name = "ay";

print(name.length);
```

Even though the type is still nullable, Dart can see that `name` can't possibly be `null` because you assigned it a non-null value right before you used it. There's no need for you to explicitly "unwrap" `name` to get at its `String` value. Dart does this for you automatically. This is known as **type promotion**. Dart promotes the nullable and largely unusable `String?` type to a non-nullable `String` with no extra work from you! Your code stays clean and beautiful. Take some time right now to send the Dart team a thank you letter.

Flow analysis

Type promotion works for more than just the trivial example above. Dart uses sophisticated **flow analysis** to check every possible route the code could take. As long as none of the routes come up with the possibility of `null`, it's promotion time!

Take the following slightly less trivial example:

```
bool isPositive(int? anInteger) {
if (anInteger == null) {
return false;
}
return !anInteger.isNegative;
}
```

In this case, you can see that by the time you get to the `anInteger.isNegative` line, `anInteger` can't possibly be `null` because you've already checked for that. Dart's flow analysis could also see that, so Dart promoted `anInteger` to its non-nullable form, that is, to `int` instead of `int?`.

Even if you had a much longer and nested `if-else` chain, Dart's flow analysis would still be able to determine whether to promote a nullable type or not.

Null-aware operators

In addition to flow analysis, Dart also gives you a whole set of tools called **null-aware operators** that can help you handle potentially null values. Here they are in brief:

If-null operator (??)

Null-aware assignment operator (??=) Null-

aware access operator (?.)

Null-aware method invocation operator (?.)

Null assertion operator (!)

Null-aware cascade operator (?..)

Null-aware index operator (?[]) Null-

aware spread operator (...?)

The following sections describe in more detail how these operators work.

If-null operator (??)

One very convenient way to handle null values is to use the double question mark (??), also known as the **if-null operator**. This operator says, "If the value on the left isn't null, then use it; otherwise, go with the value on the right."

Take a look at the following example:

```
String?  message;

final text = message ?? "Error";
```

Here are a couple points to note:

- Since message is null, ?? will set text equal to the right-hand value: "Error".
- Using ?? ensures that text can never be null, thus Dart infers the variable type of text to be String and not String?.

Print text to confirm that Dart assigned it the "rror" string rather than null.

Using the ?? operator in this example is equivalent to the following:

```
String text;
if (message == null) {
text = 'Error';
} else {
text = message;
}
```

That's six lines of code instead of one when you use the ?? operator. You know which one to choose.

Null-aware assignment operator (??=)

In the example above, you had two variables: message and text. However, another common situation is when you have a single variable that you want to update if its value is null.

For example, say you have an optional font size setting in your app:

```
double? fontSize;
```

When it's time to apply the font size to the text, your first choice is to go with the user selected size. If they haven't chosen one, then you'll fall back on a default size of 20.0. One way to achieve that is by using the if-null operator like so:

```
fontSize = fontSize ?? 20.0;
```

However, there's an even more compact way to do it. In the same way that the following two forms are equivalent,

```
x = x + 1; x
+= 1;
```

there's also a **null-aware assignment operator (??=)** to simplify if-null statements that have a single variable:

```
fontSize ??= 20.0;
```

If `fontSize` is `null` then it will be assigned `20.0`, but otherwise it retains its value. The `??=` operator combines the null check with the assignment.

Both `??` and `??=` are useful for initializing variables when you want to guarantee a non-null value.

Null-aware access operator (?.)

Earlier with `anInteger.isNegative`, you saw that trying to access the `isNegative` property when `anInteger` was `null` caused a `NoSuchMethodError`. There's also an operator for null safety when accessing object members. The **Null-aware access operator** returns `null` if the left-hand side is `null`. Otherwise, it returns the property on the right-hand side.

Look at the following example:

```
int? age;
print(age?.isNegative);
```

Since `age` is `null`, the `?.` operator prevents that code from crashing. Instead, it just returns `null` for the whole expression inside the `print` statement. Run that and you'll see the following:

```
null
```

Internally, a property is just a getter method on an object, so the `?.` operator works the same way to call methods as it does to access properties.
Therefore, another name for `?.` is the **null-aware method invocation operator**. As you can see, invoking the `toDouble()` method works the same way as accessing the `isNegative` property:

```
print(age?.toDouble());
```

Run that and it'll again print "null" without an error.

The `?.` operator is useful if you want to only perform an action when the value is non-null. This allows you to gracefully proceed without crashing the app.

Null assertion operator (`!`)

Sometimes Dart isn't sure whether a nullable variable is `null` or not, but *you* know it's not. Dart is smart and all, but machines don't rule the world yet.
So if you're absolutely sure that a variable isn't `null`, you can turn it into a non-nullable type by using the **Null assertion**

operator (**!**), or sometimes more generally referred to as the **bang operator**.

```
String nonNullableString = myNullableString!;
```

Note the ! at the end of myNullableString.

Note: In Chapter 4, you learned about the not-operator, which is also an exclamation mark. To differentiate the not-operator from the null assertion operator, you can also refer to the not-operator as the **prefix ! operator** because it goes before an expression. By the same reasoning, you can refer to the null assertion operator as the **postfix ! operator** since it goes after an expression.

Here's an example to see the assertion operator at work. In your project, add the following function that returns a nullable Boolean:

```
bool? isBeautiful(String? item) { if
(item == "flower") {
return true;
} else if (item == "garbage") {
return false;
}
return null;
}
```

Now in main, write this line:

```
bool flowerIsBeautiful = isBeautiful('flower');
```

You'll see this error:

```
A value of type 'bool?' can't be assigned to a v
ariable of type bool
```

The `isBeautiful` function returned a nullable type of `bool?`, but you're trying to assign it to `flowerIsBeautiful`, which has a non-nullable type of `bool`. The types are different, so you can't do that. However, you *know* that `'flower'` is beautiful, that is, the function won't return `null`. So you can use the null assertion operator to tell Dart that.

Add the postfix ! operator to the end of the function call:

```
bool flowerIsBeautiful = isBeautiful('flower')!;
```

Now there are no more errors.

Alternatively, since `bool` is a subtype of `bool?`, you could also cast `bool?` down using the `as` keyword that you learned about in Chapter 3.

```
bool flowerIsBeautiful = isBeautiful('flower') as
bool;
```

This is equivalent to using the assertion operator. The advantage of `!` is that it's shorter.

Beware, though. Using the assertion operator (or casting down to a non-nullable type) will crash your app with a runtime error if the value actually does turn out to be `null`, so don't use the assertion operator unless you can guarantee that the variable isn't `null`.

It's not really safe to trust the `isBeautiful` function. Who knows but that one day you'll hire someone who hates flowers and changes the internal workings of the function. That scenario isn't as far-fetched as you might think. Imagine that instead of working with a local function, you're returning data from a web server using a REST API. Someone on the server end changes a value and then your app breaks.

Here's an alternative to the assertion operator that won't ever crash the app:

```
bool flowerIsBeautiful = isBeautiful('flower') ??
true;
```

You're leaving the decision up to the function, but giving it a default value by using the ?? operator if the function doesn't know what it should be.

Think of the `!` assertion operator as a dangerous option and one to be used sparingly. By using the assertion operator, you're telling Dart that you want to opt-out of null safety, that you can handle it yourself. This is something akin to using `dynamic` to tell Dart that you want to opt-out of type safety.

Note: You'll see a common and valid use of the null assertion operator in the section below titled **No promotion for non-local variables.**

Null-aware cascade operator (?..)

In Chapter 6 you learned about the **cascade operator (..)**, which allows you to call multiple methods or set multiple properties on the same object.

Give a class like this:

```
class User {
String? name; int?
id;

}
```

If you know the object isn't nullable, you can use the cascade operator like so:

```
User user = User()
  ..name = "Ray"
 ..id = 42;
```

However, if your object *is* nullable, like in the following example:

```
User? user;
```

Then you can use the **null-aware cascade operator (?..)**:

```
user
?..name = "Ray"
..id = 42;
```

You only need to use `?..` for the first item in the chain. If `user` is `null`, then the chain will be short-circuited, that is, terminated, without calling the other items in the cascade chain.

This is similar for the null-aware access operator (`?.`) as well. Look at this example:

```
String? lengthString = user?.name?.length.toString();
```

Since `user` might be `null`, it needs the `?.` operator to access `name`. Since `name` also might be `null`, it needs the `?.` operator to access `length`. However, as long as `name` isn't `null`, `length` will never be `null`, so you only use the `.` dot operator to call `toString`. If either `user` or `name` is `null`, then the entire chain is immediately short-circuited and `lengthString` is assigned `null`.

Null-aware index operator (?[])

The **null-aware index operator** (**?[]**) is used for accessing the elements of a list when the list itself might be `null`. You've used lists already a couple of times in this book, but since you won't cover them in depth until Chapter 8, this section will just give a simple explanation of how the `?[]` operator is used.

This is an example of a nullable list:

```
List<int>? myList = [1, 2, 3];
```

What you have here is a list of integers. The list itself can be set to `null`, indicated by the question mark at the end of `List<int>?`. However, the members of the list can't be `null`, indicated by the lack of a question mark after `int`. That is, the type is `List<int>?` instead of `List<int?>?`.

In the example above, `myList` isn't `null` because you assigned it the value `[1, 2, 3]`. Now set `myList` to `null`:

```
myList = null;
```

Try to get the value of one of the items in the list:

```
int? myItem = myList?[2];
```

Print `myItem` and you'll see `null`.

If you had tried to retrieve a value from a null list in the days before null safety, you would have crashed your app. However, the `?[]` operator gracefully passes a `null` value on to `myItem`.

In Chapter 8 you'll learn more about collections, including the **null-aware spread operator** (`...?`), which is used to expand one non-null collection inside another.

Initializing non-nullable fields

When you create an object from a class, Dart requires you to initialize any non-nullable member variables before you use them.

Say you have a `User` class like this:

```
class User {
String name;

}
```

Since `name` is `String` and not `String?`, you must initialize it somehow. If you recall what you learned in Chapter 6, there are a few different ways to do that.

Using initializers

One way to initialize a property is to use an **initializer** value:

```
class User {
String name = "anonymous";

}
```

In this example, the value is `"anonymous"`, so Dart knows that `name` will always get a non-null value when an object is created from this class.

Using initializing formals

Another way to initialize a property is to use an **initializing formal**, that is, by using `this` in front of the field name:

```
class User {
User(this.name);
String name;

}
```

Having `this.name` as a required parameter ensures that `name` will have a non-null value.

Using an initializer list

You can also use an **initializer list** to set a field variable:

```
class User {
User(String  name)

: _name = name; String
_name;

}
```

The private `_name` field is guaranteed to get a value when the constructor is called.

Using default parameter values

Optional parameters default to `null` if you don't set them, so for non-nullable types, that means you *must* provide a default value.

You can set a default value for ordered parameters like so:

```
class User {

User([this.name = "anonymous"]);
String name;

}
```

Or like this for named parameters:

```
class User {
User({this.name = "anonymous"}); String
name;
}
```

Now even when creating an object without any parameters, `name` will still at least have a default value.

Required named parameters

As you learned in Chapter 5, if you want to make a named parameter required, use the `required` keyword.

```
class User {
User({required this.name}); String
name;
}
```

Since `name` is required, there's no need to provide a default value.

Nullable instance variables

All of the methods above guaranteed that the class field will be initialized, and not only initialized, but initialized with a non-null value. Since the field is non-nullable, it's not even possible to make the following mistake:

```
final user = User(name: null);
```

Dart won't let you do that. You'll get the following compile-time error:

```
The argument type 'Null' can't be assigned to th e
parameter type 'String'
```

Of course, if you want the property to be nullable, then you can use a nullable type, and then there's no need to initialize the value.

```
class User {
User({this.name}); String?
name;

}
```

`String?` makes `name` nullable. Now it's your responsibility to handle any `null` values it may contain.

No promotion for non-local variables

One topic that people often get confused about is the lack of type promotion for nullable instance variables.

As you recall from earlier, Dart promotes nullable variables in a method to their non-nullable counterpart if Dart's flow analysis can guarantee the variable will never be null:

```
bool isLong(String? text) {
if (text == null) {

return false;

}

return text.length > 100;

}
```

In this example, the local variable text is guaranteed to be non-null if the line with text.length is ever reached, so Dart promotes text from String? to String.

However, take a look at this modified example:

```
class TextWidget {
String? text;

bool isLong() {
if (text == null) {
return false;
}
return text.length > 100; // error
}
}
```

The line with text.length now gives an error:

```
The property 'length' can't be unconditionally a
ccessed because the receiver can be 'null'.
```

Why is that? You just checked for null after all.

The reason is that the Dart compiler can't guarantee that other methods or subclasses won't change the value of a non-local variable before it's used.

Since Dart has gone the path of *sound* null safety, this guarantee is essential before type promotion can happen.

You do have options, however. One is to use the ! operator:

```
bool isLong() {
if (text == null) {
return false;
}
return text!.length > 100;
}
```

Even if the compiler don't know that text isn't null, *you* know it's not, so you apply that knowledge with text!.

Another option is to shadow the non-local variable with a local one:

```
class TextWidget {
String? text;

bool isLong() {
final text = this.text; // shadowing if
(text == null) {
return false;
}
return text.length > 100;
}
}
```

The local variable text shadows the instance variable this.text and the compiler is happy.

Note: The topic of type promotion for non-local variables is in active discussion at the time of writing this chapter. It may be that when you read this Dart will have an updated solution.

The late keyword

Sometimes you want to use a non-nullable type, but you can't initialize it in any of the ways you learned above.

Here's an example:

```
class User {
User(this.name);

final String name;

final int _secretNumber = _calculateSecret();

int _calculateSecret() { return
name.length + 42;
}
```

You have this non-nullable field named `_secretNumber`. You want to initialize it based on the return value from a complex algorithm in the `_calculateSecret` instance method. You have a problem, though, because Dart doesn't let you access instance methods during initialization.

```
The instance member '_calculateSecret' can't be
accessed in an initializer.
```

To solve this problem, you can use the `late` keyword. Add `late` to the start of the line initializing `_secretNumber`:

```
late final int _secretNumber = _calculateSecret
();
```

Dart accepts it now, and there are no more errors.

Using `late` means that Dart doesn't initialize the variable right away. It only initializes it when you access it the first time. This is also known as **lazy initialization**. It's like procrastination for variables.

It's also common to use `late` to initialize a field variable in the constructor body. Here's an alternate version of the example above:

```
class User {
User(this.name) {
_secretNumber = _calculateSecret();
}
late final int _secretNumber;
// ...
}
```

Initializing a final variable in the constructor body wouldn't have been allowed if it weren't marked as `late`.

Dangers of being late

The example above was for initializing a final variable, but you can also use `late` with non-final variables. You have to be careful with this, though. Take a look at the following example:

```
class User {
late String name;
}
```

Dart doesn't complain at you, because using `late` means that you're promising Dart that you'll initialize the field before it's ever used. This moves checking from compile-time to runtime.

Now add the following code to `main` and run it:

```
final user = User();
print(user.name);
```

You broke your word and never initialized `name` before you used it. Dart is disappointed with you, and complains accordingly:

```
LateInitializationError: Field 'name' has not be
en initialized.
```

For this reason, it's somewhat dangerous to use `late` when you're not initializing it either in the constructor body or in the same line that you declare it.

Like with the null assertion operator (`!`), using `late` sacrifices the assurances of sound null safety and puts the responsibility of handling `null` into your hands. If you mess up, that's on you.

Benefits of being lazy

Who knew that it pays to be lazy sometimes? Dart knows this, though, and uses it to great advantage.

There are times when it might take some heavy calculations to initialize a variable. If you never end up using the variable, then all that initialization work was a waste. Since *lazy* initialization is never done until you actually use the variable, though, this kind of work will never be wasted.

Top-level and static variables have always been lazy in Dart. As you learned above, the `late` keyword makes other variables lazy, too. That means even if your variable is nullable, you can still use `late` to get the benefit of making it lazy.

Here's what that would look like:

```
class SomeClass {
late String? value = doHeavyCalculation(); String?
doHeavyCalculation() {
// do heavy calculation
}
}
```

The method `doHeavyCalculation` is only run after you access `value` the first time. And if you never access it, you never do the work.

Well, that wraps up this chapter. Sound null safety has made Dart an even stronger language than it already was. Aren't you glad you chose to learn Dart?

Challenges

Before moving on, here are some challenges to test your knowledge of nullability. It's best if you try to solve them yourself, but solutions are available with the supplementary materials for this book if you get stuck.

Challenge 1: Random nothings

Write a function that randomly returns `42` or `null`. Assign the return value of the function to a variable named `result` that will never be `null`. Give `result` a default of `0` if the function returns `null`.

Challenge 2: Naming customs

People around the world have different customs for giving names to children. It would be difficult to create a data class to accurately represent them all, but try it like this:

Create a class called `Name` with `givenName` and `surname` properties.

Some people write their surname last and some write it first. Add a Boolean property called `surnameIsFirst` to keep track of this.

Not everyone in the world has a surname.

Add a `toString` method that prints the full name.

Key points

Null means "no value."

A common cause of errors for programming languages in general comes from not properly handling null.

Dart 2.12 introduced sound null safety to the language.

Sound null safety distinguishes nullable and non- nullable types.

A non-nullable type is guaranteed to never be null.

Null aware operators help developers to gracefully handle null.

```
??      if-null operator
??=     null-aware assignment operator
?.      null-aware access operator
?.      null-aware method invocation operator
!       null assertion operator
?..     null-aware cascade operator
?[]     null-aware index operator
...?    null-aware spread operator
```

The late keyword allows you to delay initializing a field in a class.

Using late also makes initialization lazy, so a variable's value won't be calculated until you access the variable for the first time.