Chapter 6: Classes

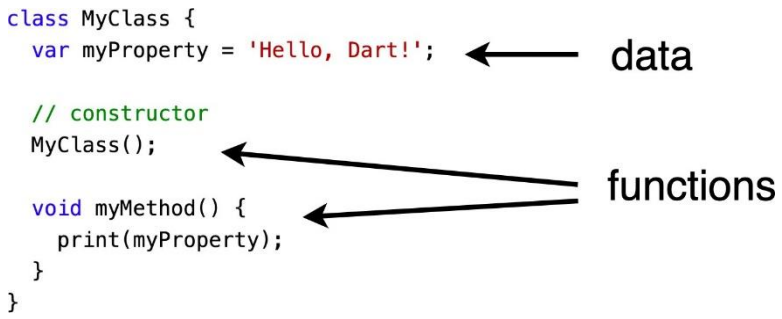
So far in this book, you've used built-in types such as `int`, `String` and `bool`. You've also seen one way to make custom types using `enum`. In this chapter, you'll learn a more flexible way to create your own types by using classes.

Dart classes

Classes are like architectural blueprints that tell the system how to make an object, where an object is the actual data that's stored in the computer's memory. If a class is the blueprint, then you could say the object is like the house that the blueprint represents. For example, the `String` class describes its data as a collection of UTF-16 code units, but a `String` object is something concrete like `"Hello, Dart!"`. All values in Dart are objects that are built from a class. This includes the values of basic types like `int`, `double` and `bool`

That's different from other languages like Java, where basic types are primitive. For example, if you have `x = 10` in Java, the value of `x` is `10` itself. However, Dart doesn't have primitive types. Even for a simple `int`, the value is an object that wraps the integer. You'll learn more on this concept later.

Classes are a core component of object-oriented programming. They're used to combine data *and* functions inside a single structure.



The functions exist to transform the data. Functions inside of a class are known as **methods,** while **constructors** are special methods you use to create objects from the class.

It's time to get your hands dirty. Working with classes is far more instructive than reading about them!

Defining a class

To get started creating your own types, you'll make a simple `User` class that has `id` and `name` properties. This is just the kind of class that you're highly likely to create in the future for an app that requires users to log in.

Write the following simple class at the top level of your Dart file. Your class should be outside of the main function, either above or below it.

```
class User {
int id = 0;
String name = '';
}
```

This creates a class named User. It has two properties; id is an int with a default value of 0, and name is a String with the default value of an empty string.

> **Note** : Depending on the situation, null may be a better default than 0 or '' However, since nullable types and null safety won't be fully covered until Chapter 7, this chapter will simply use reasonable defaults for properties.

Creating an object from a class

As mentioned above, the value you create from a class is called an object. Another name for an object is **instance**, so creating an object is sometimes called **instantiating a class.**

Since coding the class in your Dart file as you did above simply creates the blueprint, a User object doesn't exist yet. You can create one by calling the class name as you would call a function. Add the following line inside the main function:

```
final user = User();
```

This creates an instance of your `User` class and stores that instance, or object, in `user`. Notice the empty parentheses after `User`. It looks like you're calling a function without any parameters. In fact, you are calling a type of function called a **constructor method**. You'll learn a lot more about them later in the chapter. Right now, simply understand that using your class in this way creates an instance of your class.

The optional keyword new

Before version 2.0 of Dart came out, you had to use the `new` keyword to create an object from a class. At that time, creating a new instance of a class would have looked like this:

```
final user = new User();
```

In fact, this still works, but the `new` keyword is completely optional now, so it's better to just leave it off. Why clutter your code with unnecessary words, right?

You'll still come across `new` from time to time in the documentation or in legacy code, but at least now you won't be confused by it. You can delete it if you come across it.

Assigning values to properties

Now that you have an instance of `User` stored in `user`, you can assign new values to this object's properties by using

DOT notation. To access the name property, type user *dot* name, and then give it a value:

```
user.name = "Ray";
```

Now, set the ID in a similar way:

```
user.id = 42;
```

Your code should look like the following:

```
void main() {
final user = User();
user.name = "Ray"; user.id
= 42;
}

class User {
int id = 0;
String name = '';
}
```

You'll notice that you have both a function and a class together. Dart allows you to put multiple classes, top-level functions and even top-level variables all together in the same file. Their order in the file isn't important. `User` is located below `main` here, but if you put it above `main`, that's fine as well.

You've defined a `User` data type with a class, created an object from it and assigned values to its parameters. Run the code now, though, and you won't see anything special

happen. The next section will show you how to display data from an object.

Printing an object

You can print any object in Dart. However, if you try to print `user` now, you won't get quite what you hoped for. Add the following line at the bottom of the `main` function and run the code:

```
print(user);
```

Here's what you get:

```
Instance of 'User'
```

Hmm, you were likely expecting something about Ray and the ID. What gives?

All classes in Dart (well, except for `Null`, but that's a topic for Chapter 7) are derived from `Object`, which has a `toString` method. In this case, your object doesn't tell Dart how to write its internal data when you call `toString` on it, so Dart gives you this generic, default output instead. However, you can override the `Object` class's version of `toString` by writing your own implementation of `toString`, and thus customize how your own object will print out.

Add the following method to the `User` class:

```
@override
String toString() {
return "User(id: $id, name: $name)";
}
```

Words that start with @ are called **annotations**. Including them is optional and doesn't change how the code executes. However, annotations *do* give the compiler more information so that it can help you out at compile time. Here, the `@override` annotation is telling both you and the compiler that `toString` is a method in `Object` that you want to override with your own customized version, so if you accidentally wrote the `toString` method signature incorrectly, the compiler would warn you about it because of the `@override` annotation.

Since methods have access to the class properties, you simply use that data to output a more meaningful message when someone prints your object. Run the code now, and you'll see the following result:

```
User(id: 42, name: Ray)
```

That's far more useful!

Note: Your `User` class only has a single method right now, but in classes with many methods, most programmers put the `toString` method at or near the bottom of the class instead of burying it in the middle somewhere. As you continue to add to `User`, keep `toString` at the bottom. Following conventions like this makes navigating and reading the code easier.

Adding methods

Now that you've learned to override methods, you're going to move on and add your own methods to the `User` class. But before you do, there's a little background information that you should know.

Understanding object serialization

Being able to organize related data into a class is super useful, especially when you want to pass that data around as a unit within your app. One disadvantage, though, shows up when you're saving the object or sending it over the network. Files, databases and networks only know how to handle simple data types, such as numbers and strings. They don't know how to handle anything more complex, like your `User` data type.

Serialization is the process of converting a complex data object into a string. Once the object has been serialized, it's easy to save that data or transfer it across the network because everything from your app to the network and beyond knows how to deal with strings. Later, when you

want to read that data back in, you can do so by way of **deserialization**, which is simply the process of converting a string back into an object of your data type.

You didn't realize it, but you actually serialized your `User` object in the `toString` method above. The code you wrote was good enough to get the job done, but you didn't really follow any standardized format. You simply wrote it out in a way that looked nice to the human eye. If you gave that string to someone else, though, they might have some difficulty understanding how to deserialize it, that is, convert it back into a `User` object.
It turns out that serialization and deserialization are such common tasks that people have devised a number of standardized formats for serializing data. One of the most common is called **JSON: JavaScript Object Notation**. Despite the name, it's used far and wide outside the world of JavaScript.

JSON isn't difficult to learn, but this chapter will only show you enough JSON to serialize your `User` object into a more portable format. Check the *Where to go from here* section at the end of the chapter to find out where you can learn more about this format.

Adding a JSON serialization method

You're going to add another method to your class now that will convert a `User` object to JSON format. It'll be similar to what you did in `toString`.

Add the following method to the `User` class, putting it above the `toString` method:

```
String toJson() {
return '{"id":$id,"name":"$name"}';
}
```

Here are a few things to note:

- Since this is your own custom method and you're not overriding a method that belongs to another class, you don't add the `@override` annotation.
- In Dart naming conventions, acronyms are treated as words. Thus, `toJson` is a better name than `toJSON`.
- There's nothing magic about serialization in this case. You simply used string interpolation to insert the property values in the correct locations in the JSON formatted string.
- In JSON, objects are surrounded by curly braces, properties are separated by commas, property names are separated from property values by colons, and strings are surrounded by double-quotes. If a string needs to include a double-quote inside itself, you escape it with a backslash like so: `\"`
- JSON is very similar to a Dart data type called `Map`. In fact, Dart even has built-in functions in the `dart:convert` library to serialize and deserialize JSON maps. And that's actually what most people use to

serialize objects. However, you haven't read Chapter 8 about maps yet, so this example is going to be low-tech. You'll see a little preview of Map, though, in the fromJson example later in this chapter.

To test out your new function, add the following line to the bottom of the main method:

```
print(user.toJson());
```

This code calls the custom toJson method on your user object using dot notation. The dot goes between the object name and method name just like you saw earlier for accessing a property name.

Run the code and you'll see the following:

```
{"id":42,"name":"Ray"}
```

It's very similar to what toString gave you, but this time it's in standard JSON format, so a computer on the other side of the world could easily convert that back into a Dart object.

Cascade notation

When you created your User object above, you set its parameters like so:

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

However, Dart offers a cascade operator (`..`) that allows you to chain together multiple assignments on the same object without having to repeat the object name. The following code is equivalent:

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

Note that the semicolon only appears on the last line.

Cascade notation isn't strictly necessary, but it does makes your code just a little tidier when you have to assign a long list of properties or repeatedly call a method that modifies your object.

Mini-exercises

1. Create a class called `Password` and give it a string property called `value`.

2. Override the `toString` method of `Password` so that it prints `value`.

3. Add a method to `Password` called `isValid` that returns `true` only if the length of `value` is greater than `8`.

Constructors

Constructors are methods that create, or *construct*, instances of a class. That is to say, constructors build new objects. Constructors have the same name as the class, and the

implicit return type of the constructor method is also the same type as the class itself.

Default constructor

As it stands, your `User` class doesn't have an explicit constructor. In cases like this, Dart provides a default constructor that takes no parameters and just returns an instance of the class. For example, defining a class like this:

```
class Address { var
value = '';

}
```

Is equivalent to writing it like this:

```
class Address {
Address();
var value = '';

}
```

Including the default `Address()` constructor is optional.

Sometimes you don't want the default constructor, though. You'd like to initialize the data in an object at the same time that you create the object. The next section shows how to do just that.

Custom constructors

If you want to pass parameters to the constructor to modify how your class builds an object, you can. It's similar to how you wrote functions with parameters in Chapter 5.

Like the default constructor above, the constructor name should be the same as the class name. This type of constructor is called a **generative constructor** because it directly generates an object of the same type.

Long-form constructor

In Dart the convention is to put the constructor before the property variables. Add the following generative constructor method at the top of the class body:

```
User(int id, String name) {
this.id = id;
this.name = name;
}
```

This is known as a **long-form constructor**. You'll understand why it's considered "long-form" when you see the short-form constructor later.

This is what the class looks like (without the `toJson` and `toString` methods):

```
class User {
User(int id, String name) {
this.id = id;
this.name  =  name;
}
int id = 0;
String name = '';
// ...
}
```

this is a new keyword. What does it do?
The keyword this in the constructor body allows you to disambiguate which variable you're talking about. It means *this* object. So this.name refers the object property called name, while name (without this) refers to the constructor parameter. Using the same name for the constructor parameters as the class properties is called **shadowing**. So the constructor above takes the id and name parameters and uses this to initialize the properties of the object.

Delete everything from inside the main function body. Then add the following code in its place to create a User object by passing in some arguments:

```
final user = User(42, 'Ray');
print(user);
```

Once the object has been created, you can access its properties and other methods just as you did before. However, you can't use the default constructor User()

anymore, since `id` and `name` are required positional parameters.

Short-form constructor

Dart also has a **short-form constructor** where you don't provide a function body, but you instead list the properties you want to initialize, prefixed with the `this` keyword. Arguments you send to the short form constructor are used to initialize the corresponding object properties.

Here is the long form constructor that you currently have:

```
User(int id, String name) {
this.id = id;
this.name  =  name;
}
```

Now replace that with the following short-form constructor:

```
User(this.id, this.name);
```

Dart infers the constructor parameter types of `int` and `String` from the properties themselves that are declared in the class body.

The class should now look like this:

```
class User {
User(this.id, this.name);
int id = 0;
String name = '';
// ...
}
```

Run the code again. You'll see the short-form constructor works just like the longer form you replaced, but it's just a little tidier now.

> **Note**: You could remove the default property values of 0 and '' at this point since `id` and name are guaranteed to be initialized by the constructor parameters. However, there's an intermediate step in the next section where they'll still be useful. Keeping the default values a little longer will allow this chapter to postpone dealing with null safety until it can be covered more fully in Chapter 7.

Named constructors

Dart also has a second type of generative constructor called a **named constructor**, which you create by adding an identifier on to the class name. It takes the following pattern:

```
ClassName.identifierName()
```

From here on, this chapter will refer to a constructor without the identifier, that is, one which only uses the class name, as an **unnamed constructor**.

Why would you want a named constructor instead of the nice, tidy default one? Well, sometimes you have some common cases that you want to provide a convenience constructor for. Or maybe you have some special edge cases for constructing certain classes that need a slightly different approach.

Say, for example, that you want to have an anonymous user with a preset ID and name. You can do that by creating a named constructor. Add the following named constructor below the short-form constructor:

```
User.anonymous() { id
= 0;
name    =    "anonymous";
}
```

The identifier, or named part, of the constructor is `.anonymous`. Named constructors may have parameters, but in this case, there are none. And since there aren't any parameter names to get confused with, you don't need to use `this.id` or `this.name`. Rather, you just use the property variables `id` and `name` directly.

Call the named constructor in `main` like so:

```
final anonymousUser = User.anonymous();
print(anonymousUser);
```

Run that and you'll see the expected output:

```
User(id: 0, name: anonymous)
```

> **Note** : Without default values for id and name, Dart would have complained that these variables weren't being initialized, even though User.anonymous does in fact initialize them in the constructor body. You could solve the problem by using the late keyword, but that's a topic for Chapter 7 — hence the default values here.

Forwarding constructors

In the named constructor example above, you set the class properties directly in the constructor body. However, this doesn't follow the DRY principle you learned earlier. You're repeating yourself by having two different locations where you can set the properties. It's not a huge deal, but imagine that you have five different constructors instead of two. It would be easy to forget to update all five if you had to make a change, and if the constructor logic were complicated, it would be easy to make a mistake.
One way to solve this issue is by calling the main constructor from the named constructor. This is called **forwarding** *or* **redirecting**. To do that, you use the keyword this again.

Delete the anonymous named constructor that you created above and replace it with the following:

```
User.anonymous() : this(0, "anonymous");
```

This time there's no constructor body, but instead, you follow the name with a colon and then forward the properties on to the unnamed constructor. The forwarding syntax replaces `User` with `this`.

Also, now that you've moved property initialization from the constructor body to the parameter list, Dart is finally convinced that `id` and `name` are guaranteed to be initialized. Replace these two lines:

```
int id = 0;
String name = '';
```

with the following:

```
int id; String
name;
```

No complaints from Dart.

You call the named constructor exactly like you did before:

```
final anonymousUser = User.anonymous();
```

The results are the same as well.

Optional and named parameters

Everything you learned about function parameters in Chapter 5 also applies to constructor method parameters. That means you can make parameters optional using square brackets:

```
MyClass([this.myProperty]);
```

Or you can make them optional and named using curly braces:

```
MyClass({this.myProperty});
```

Or named and required using curly braces and the `required` keyword:

```
MyClass({required this.myProperty});
```

Adding named parameters for User

Earlier when you instantiated a `User` object, you did this:

```
final user = User(42, "Ray");
```

For someone not familiar with your `User` class, they might think 42 is Ray's age, or his password, or how many pet cats he has. Using named parameters here would help a lot with readability.

Refactor the unnamed constructor in `User` by adding braces around the parameters. Since that makes the parameters

optional, you could use the `required` keyword as you saw in Chapter 5, but this time simply give them default values:

```
User({this.id = 0, this.name = "anonymous"});
```

The compiler will complain at this, because that change also requires refactoring the `anonymous` named constructor to use the parameter names in the redirect.

However, since the parameter defaults are now just what the `anonymous` constructor was doing before, you can delete the `anonymous` constructor and replace it with the following:

```
User.anonymous() : this();
```

Using the named constructor will now forward to the unnamed constructor with no arguments.

In `main`, the way to create an anonymous user is still the same, but now the way to create Ray's user object is like so:

```
final user = User(id: 42, name: "Ray");
```

That's a lot more readable. They're not cats; it's clearly just an ID.

In case you've gotten lost, here's what things look like now:

```
class User {
// unnamed constructor
User({this.id = 0, this.name = "anonymous"});
// named constructor User.anonymous()
: this();

int id; String
name;
// ...
}
```

Initializer lists

You might have discovered a small problem that exists with your class as it's now written. Take a look at the following way that an unscrupulous person could use this class:

```
final vicki = User(id: 24, name: "Vicki");
vicki.name = "Nefarious Hacker";
print(vicki);
// User(id: 24, name: Nefarious Hacker)
```

If those statements were spread throughout the code base instead of being in one place as they are here, someone printing the `vicki` user object and expecting a real name would get a surprise. "Nefarious Hacker" is definitely not what you'd expect. Once you've created the `User` object, you don't want anyone to mess with it.

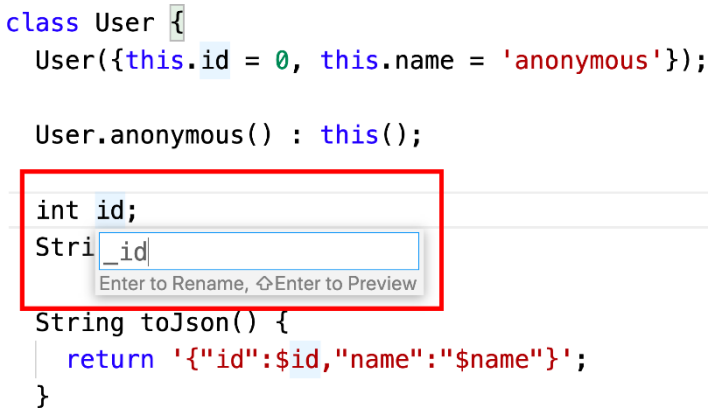
But forget nefarious hackers; *you're* the one who's most likely to change a property and then forget you did it.

There are a couple of ways to solve this problem. You'll see one solution now and a fuller solution later.

Private variables

Dart allows you to make variables private by adding an underscore (_) in front of their name.

Change the `id` property to `_id` and `name` to `_name`. Since these variables are used in several locations throughout your code, let VS Code help you out. Put your cursor on the variable name and press F2. Edit the variable name and press enter to change all of the references at once.



Oh..., that actually renamed a few more things than you intended since it also renamed what was in the `main` function. But that's OK. Just delete everything inside the body of `main` for now.

There is still one problem left with the unnamed constructor in `User`:

```
User({this._id = 0, this._name = 'anonymous'});
```

The compiler gives you an error:

```
Named parameters can't start with an
```

Fix that by deleting the unnamed constructor and replacing it with the following:

```
User({int id = 0, String name = "anonymous"})
: _id = id,
_name = name;
```

Do you see the colon that precedes _id? The comma-separated list that comes after it is called the **initializer list**. One use for this list is exactly what you've done here. Externally, the parameters have one name, while internally, you're using private variable names.

The initializer list is always executed before the body of the constructor, if the body exists. You don't need a body for this constructor, but if you wanted to add one, it would look like this:

```
User({int id = 0, String name = "anonymous"})
: _id = id,
_name = name { print("User name
is $_name");
}
```

The constructor would initialize _id and _name before it ran the print statement inside the braces.

Note: The member variables of a class are generally called **fields**. However, when the fields are public, that is, when they're visible to the outside world, you can also call them **properties**. The Getters and Setters sections below will show you how to use public properties and private fields at the same time.

Why aren't the private properties private?

It turns out that your nefarious hacker can still access the "private" fields of `User`. Add the following two lines to `main` to see this in action:

```
final vicki = User(id: 24, name: "Vicki");
vicki._name = "Nefarious Hacker";
```

What's that all about? Well, using an underscore before a variable or method name makes it **library private**, not **class private**. For your purposes in this chapter, a library is simply a file. Since the `main` function and the `User` class are in the same file, nothing in `User` is hidden from `main`. To see private variables in action, you'll need to make another file so that you aren't using your class in the same file in which it's defined.

Create a new file called **user.dart** in the same folder as the file you've been working with. In VS Code, you can click on the current file in the Explorer panel and then press the **New File button.**

Now move the entire `User` class over to **user.dart**.

Back in the previous file with the `main` function (if you're using the starter project, it would be named **starter.dart**), add the library import to the top of the class like so:

```
import 'user.dart';
```

Now you'll notice that in the `main` function you no longer have access to _name:

```
vicki._name = 'Nefarious Hacker';
```

This produces an error:

```
The setter '_name' isn't defined for the type 'User'.
```

Great! Now it's no longer possible to change the properties after the object has been created. Delete or comment out that entire line:

```
// vicki._name = 'Nefarious Hacker';
```

Before leaving this section on initializer lists, there's one more thing to talk about: errors.

Checking for errors

Initializer lists are a great place to check for errors in the constructor parameters, which you can do by adding **assert** statements. Think of asserts like sanity checks that make sure you aren't doing anything silly, by checking that a condition is in fact true.

An **assert** statement takes a condition, and if the condition is **false**, terminates the app. This only happens during debugging, though. The compiler completely ignores **assert** statements in release builds.

Replace the unnamed constructor with the following updated version:

```
User({int id = 0, String name = "anonymous"})
: assert(id >= 0), assert(name.isNotEmpty),
_id = id,
_name = name;
```

Note the two **assert** statements in the initializer list; it's customary to put these asserts at the top of the list. The first assert checks that **id** is greater than or equal to zero, and the second one checks that **name** actually has a value. Of course you don't want a negative ID or an empty name. If either of those conditions ever occurs, then terminating is your best bet, as it gives you a loud and clear message that you need to handle that situation in code before a **User** is ever created.

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

```
final jb = User(id: -1, name: "JB Lorenzo");
```

Run the project and you'll get an error:

Note: If you're running Dart from the command line rather than from VS Code, you won't get assert errors unless you enable them with the `--enable-asserts` flag. Assuming your `main` function is in a file called starter.dart, you should run your program in this way: `dart --enable-asserts starter.dart`

```
Failed assertion: line 3 pos 16: "id >= 0": is n
ot true.
```

Since `id` must be greater than or equal to zero, either comment out the line or change `id` to a positive number.

```
// final jb = User(id: 100, name: "JB Lorenzo");
```

Constant constructors

You've already learned how to keep people from modifying the properties of a class by making them private. Another thing you can do is to make the properties **immutable**, that is, unchangeable. By using immutable properties, you don't even have to make them private.

Making properties immutable

There are two ways to mark a variable immutable in Dart: `final` and `const`. However, since the compiler won't know

what the properties are until runtime, your only choice here is to use `final`.

In the `User` class in **user.dart**, add the `final` keyword before both property declarations. Your code should look like this:

```
final String _name; fi
nal int _id;
```

Adding `final` means that `_name` and `_id` can only be given a value once, that is, when the constructor is called. After the object has been created, those properties will be immutable. You should keep the `String` and `int` type annotations, because removing them would cause the compiler to fall back to `dynamic`.

Making classes immutable

If the objects of a particular class can never change, because all fields of the class are `final`, you can add `const` to the constructor to ensure that all instances of the class will be constants at compile-time.

Since both the fields of your `User` class are now `final`, this class is a good candidate for a compile-time constant.

Replace both constructors with the following:

```
const User({int id = 0, String name = "anonymou s"})
: assert(id >= 0),
_id = id,
_name = name;

const User.anonymous() : this();
```

Note the const keyword in front of both constructors. In the first constructor, there used to be an assert statement for name.isNotEmpty, but unfortunately, this is a runtime check and not allowed for a compile-time constant, so that had to be removed.

Now you can declare your User objects as compile-time constants like so:

```
const user = User(id: 42, name: "Ray");
const anonymousUser = User.anonymous();
```

Benefits of using const

In addition to being immutable, another benefit of const variables is that they're **canonical instances**, which means that no matter how many instances you create, as long as the properties used to create them are the same, Dart will only see a single instance. You could instantiate User.anonymous() a thousand times across your app without incurring the performance hit of having a thousand different objects.

> **Note**: Flutter uses this pattern frequently with its const widget classes in the user interface of your app. Since Flutter knows that the const widgets are immutable, it doesn't have to waste time recalculating and drawing the layout when it finds these widgets.

Make it your goal to use const objects and constructors as much as possible. It's a performance win!

Factory constructors

All of the constructors that you've seen up until now have been generative constructors. Dart also provides another type of constructor called a **factory constructor**.

A factory constructor provides more flexibility in how you create your objects. A generative constructor can only create a new instance of the class itself. However, factory constructors can return *existing* instances of the class, or even subclasses of it. (You'll learn about subclasses in Chapter 9.) This is useful when you want to hide the implementation details of a class from the code that uses it.

The factory constructor is basically a special method that starts with the `factory` keyword and returns an object of the class type. For example, you could add the following factory constructor to your `User` class:

```
factory User.ray() {
return User(id: 42, name: "Ray");
}
```

The factory method uses the generative constructor to create and return a new instance of `User`. You could also accomplish the same thing with a named constructor, though.

A more common example you'll see is using a factory constructor to make a `fromJson` method:

```
factory User.fromJson(Map<String, Object> json)
{
final userId = json['id'] as int;
final userName = json['name'] as String; return
User(id: userId, name: userName);
}
```

You would create a `User` object from the constructor like so:

```
final map = {'id': 10, 'name': 'Manda'}; fi
nal manda = User.fromJson(map);
```

As mentioned earlier, you'll learn how the `Map` collection works in Chapter 8. The thing to pay attention to now is that the factory constructor body allows you to perform some work before returning the new object, without exposing the inner wiring of that instantiation process to whoever is using the class. For example, you could create a `User.fromJson` constructor with a named constructor like so:

```
User.fromJson(Map<String, Object> json)
: id = json['id'] as int,
name = json['name'] as String;
```

However, besides adding some simple asserts to the initializer list, there isn't much else you can do to with `id` and `name`. With a factory constructor, though, you could do all kinds of validation, error checking and even modification of the arguments before creating the object. This is actually highly desireable in the case here because if `"id"` or `"String"` didn't exist in the map, then your app would crash because you aren't handling `null`.

> **Note**: Using a factory constructor over a named constructor can also help to prevent breaking changes for subclasses of your class. That topic is a little beyond the scope of this chapter,

You'll see a few more uses of the factory constructor in the section below on static members.

Constructor summary

Since there are so many ways that constructors can vary, here's a brief comparison.

Constructors can be:

- Forwarding or non-forwarding
- Named or unnamed Generative or
- factory

Constant or not constant Take the

following example:

```
const User(this.id, this.name);
```

This is a non-forwarding, unnamed, generative, const constructor.

Mini-exercises

Given the following class:

```
class Password {
String value = '';

}
```

1. Make `value` a `final` variable, but not private.

2. Add a `const` constructor as the only way to initialize a `Password` object.

Dart objects

Objects act as *references* to the instances of the class in memory. That means if you assign one object to another, the other object simply holds a reference to the same object in memory — not a new instance.

So if you have a class like this:

```
class MyClass {
var myProperty = 1;
}
```

And you instantiate it like so:

```
final myObject = MyClass();
final anotherObject = myObject;
```

Then `myObject` and `anotherObject` both reference the *same* place in memory. Changing `myProperty` in either object will affect the other, since they both reference the same instance:

```
print(myObject.myProperty);          // 1
anotherObject.myProperty = 2;
print(myObject.myProperty);          // 2
```

As you can see, changing the value of the property on `anotherObject` also changed it in `myObject`, as they are really just two names for the same object.

Note: If you want to make an *actual* copy of the class — not just a copy of its reference in memory but a whole *new* object with a deep copy of all the data it contains — then you'll need to implement that mechanism yourself by creating a method in your class that builds up a whole new object.

If that still doesn't make sense, you can look forward to sitting down by the fire and listening to the story of *The*

House on Wenderlich Way in Chapter 8, which will make this all clear, or at least help you see it from a different perspective.

For now, though, there are a few more improvements you can make to the `User` class.

Getters

Right now the `User` class fields are private:

```
class User {
// ...
final int _id;
final String _name;
// ...
```

That means there's no way to access the user ID and name outside of the class, which makes your object kind of useless. You can solve this problem by adding a **getter**, which is a special method that uses the `get` keyword before a property name and returns a value. This gives you, as the class author, some control over how people access and modify properties, instead of giving people raw and unfettered access.

In the `User` class, near the `_id` and `_name` properties, *add* the following lines:

```
int get id => _id;
String get name => _name;
```

The get keyword provides a public-facing property name; in this case, id and name. When you call the getter using its name, the get method returns a value. The two getter methods here are simply returning the _id and _name field values.

Now that the properties are exposed, you can use them like so:

```
const ray = User(id: 42, name: "Ray");
print(ray.id);      // 42
print(ray.name); // Ray
```

Calculated properties

You can even create getters that aren't backed by a dedicated field value, but instead are calculated when called. Here's an example:

```
bool get isBigId => _id > 1000;
```

There's no internal variable named isBigId or _isBigId. Rather, the return value of isBigId is calculated when necessary.

All of the getters above use arrow syntax, but you could also use brace syntax with a return statement if you wanted to use more than a single line of code to calculate the return value.

Setters

If you need mutable data in a class, there's a special set method to go along with get.

Rather than modifying any of the existing fields of User to have a setter, here's a simple new class called Email:

```
class Email {
var _address = '';
String get value => _address;
set value(String address) => _address = addres s;
}
```

The last line in the body is the setter, which starts with the set keyword. The name value is the same name that's used by the getter above it. The set method takes a parameter, which you can use to set some value. In this case, you're setting the email address.

You can now assign and retrieve the value of _address like this:

```
final email = Email();
email.value = 'ray@example.com'; fi
nal emailString = email.value;
```

The second line sets the internal _address field, and the third line gets it.

You can see how this could give you some extra control over what's assigned to your properties; for instance, you could

sanitize input, check for properly formatted email addresses, and more.

Refactoring

You don't always need to use getters and setters explicitly. In fact, if all you're doing is shadowing some internal field variable, then you're better off just using a public variable.

Refactoring the Email class

Refactoring the `Email` class from above to use a public variable would look like this:

```
class Email {
var value = '';
}
```

Dart implicitly generates the needed getters and setters for you. That's quite a bit more readable and it still works exactly the same:

```
final email = Email();
email.value = 'ray@example.com'; fi
nal  emailString  =  email.value;
```

That's the beauty of how Dart handles class properties. You can change the internal implementation, without the external world being any the wiser.

If you only want a getter but not a setter, then just make the property `final`, which will also require adding a constructor to initialize the property:

```
class Email {
Email(this.value); fi
nal value;

}
```

Since you added a constructor, that does affect the outside world when instantiating the object, but it's nothing different than what you've seen previously:

```
final email = Email("ray@example.com"); final
emailString = email.value;
```

Retrieving `value` is still the same as before.

Refactoring User

You can use these same principles to refactor the `User` class. Replace the entire class with the following code:

```
class User {
const User({this.id = 0, this.name = "anonymou s"})
: assert(id >= 0);

const User.anonymous() : this(); final

String name;

final int id;

String toJson() {
return "{"id":$id,"name":"$name"}";
}

@override
String toString() {
return "User(id: $id, name: $name)";
}
}
```

Here's what changed:

- Since the getters were only shadowing internal private fields, it was cleaner to remove the fields and just use a `final` property. If you ever need to use hidden fields again later, you can do so without affecting how the class is used from the outside.

- Since you removed the private fields, setting them in the initializer list is no longer necessary.

- Because the properties are now public variables, you can use `this.id` and `this.name` to initialize them in the short-form constructor style.

“Why all the runaround?” you ask. “I could have done all this in the beginning.”

True, true. But then you wouldn’t have learned so much! :]

Static members

There is just one more thing to cover for your well-rounded foundation in Dart classes. That’s the **static** keyword.

If you put **static** in front of a member variable or method, that causes the variable or method to belong to the *class* rather than the *instance*:

```
class SomeClass {
static int myProperty = 0; static
void myMethod() {
print("Hello, Dart!");
}
```

And you access them like so:

```
final value = SomeClass.myProperty;
SomeClass.myMethod();
```

In this case, you didn’t have to instantiate an object to access **myProperty** or to call **myMethod**. Instead, you were able to use the class name directly to get the value and call the method.

The following sections will cover a few common use cases for static members.

Static variables

Static variables are often used for constants and in the singleton pattern.

Note: Variables are given different names according to where they belong or where they're located. Since static variables belong to the class, they're called **class variables**. Non-static member variables are called **instance variables** because they only have a value after an object is instantiated. Variables within a method are called **local variables**, and top-level variables outside of a class are called **global variables**.

Constants

You can define class constants by combining `static` and `const`. For example:

```
static const myConstant = "42";
```

Doing so in the `User` class would improve readability for the default user ID and name. Add the following two lines to the class body below the properties:

```
static const _anonymousId = 0;
static const _anonymousName = "anonymous";
```

Replace the unnamed constructor with the following:

```
const User({
this.id = _anonymousId, this.name
= _anonymousName,
}) : assert(id >= 0);
```

While anonymous was already pretty readable, _anonymousId is much more meaningful than 0.

Singleton pattern

A second use of static variables is to create a **singleton** class. Singletons are a common design pattern where there is only ever one instance of an object. While some people debate their benefits, they do make certain tasks more convenient.

It's easy to create a singleton in Dart. You wouldn't want User to be a singleton, since you'll likely have lots of distinct users, requiring lots of distinct instances of User. However, you might want to create a singleton class as a database helper so that you can ensure that you don't open multiple connections to the database.

Here is what a basic singleton class would look like:

```
class MySingleton {
MySingleton._();
static final MySingleton instance = MySingleto
n._();
}
```

The MySingleton._() part is a private named constructor. Some people like to call it _internal to emphasize that it can't be called from the outside. The underscore makes it

impossible to instantiate the class normally. However, the static property, which is only initialized once, provides a reference to the instantiated object.

Note: Static fields and top-level variables, that is, global variables outside of a class, are **lazily initialized**. That means Dart doesn't actually calculate and assign their values until you use them the first time.

You would access the singleton like so:

```
final mySingleton = MySingleton.instance;
```

Since factory constructors don't need to return new instances of an object, you can also implement the singleton pattern with a factory constructor:

```
class MySingleton {
MySingleton._();
static final MySingleton _instance = MySingleto
n._();
factory MySingleton() => _instance;
}
```

The advantage here is that you can hide the fact that it's a singleton from whoever uses it:

```
final mySingleton = MySingleton();
```

From the outside, this looks exactly like a normal object. This gives you the freedom to change it back into a generative constructor later without affecting the code in other parts of your project.

The last two sections have been about static variables. Next, you'll take a look at static methods.

Static methods

There are a few interesting things you can do with static methods.

Utility methods

One use for a static method is to create a utility or helper method that's associated with the class, but not associated with any particular instance.

In other languages, some developers like to group related static utility methods in classes to keep them organized. However, in Dart it's usually better to just put these utility methods in their own file as top-level functions. You can then import that file as a library wherever you need the utility methods contained within.

Creating new objects

You can also use static methods to create new instances of a class based on some input passed in. For example, you could use a static method to achieve precisely the same result as you did earlier with the `fromJson` factory constructor. Here's the static method version:

```
static User fromJson(Map<String, Object> json) { fi
nal userId = json['id'] as int;
final userName = json['name'] as String; return
User(id: userId, name: userName);
}
```

From the outside as well, you use it as you did with the factory version:

```
final map = {'id': 10, 'name': 'Manda'}; fi
nal manda = User.fromJson(map);
```

All this goes to show that there are often multiple ways of accomplishing the same thing.

Comparing static methods with factory constructors

Factory constructors in many ways are just like static methods, but there are a few differences:

A factory constructor can only return an instance of the class type or subtype, while a static method can return anything. For example, a static method can be asynchronous and return a `Future`, which you'll learn about in Chapter 10, but a factory constructor can't do this.

A factory constructor can be unnamed so that, from the caller's perspective, it looks exactly like calling a generative constructor. The singleton example above is an example of this. A static method, on the other hand, must have a name.

A factory constructor can be `const` if it's a forwarding constructor, but a static method can't.

Challenges

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

Challenge 1: Bert and Ernie

Create a `Student` class with `final` `firstName` and `lastName` `String` properties and a variable `grade` as an `int` property. Add a constructor to the class that initializes all the properties. Add a method to the class that nicely formats a `Student` for printing. Use the class to create students `bert` and `ernie` with grades of 95 and 85, respectively.

Challenge 2: Spheres

Create a `Sphere` class with a `const` constructor that takes a positive length `radius` as a named parameter. Add getters for the the volume and surface area but none for the radius. Don't use the `dart:math` package but store your own version of `pi` as a `static` constant. Use your class to find the volume and surface area of a sphere with a radius of 12.

Key points

- Classes package data and functions inside a single structure.
- Variables in a class are called fields, and public fields or getter methods are called properties.
- Functions in a class are called methods.
- You can customize how an object is printed by overriding the `toString` method.
- You create an object from a class by calling a constructor method.
- Generative constructors can be unnamed or named.
- Unnamed generative constructors have the same name as the class, while named generative constructors have an additional identifier after the class name.
- You can forward from one constructor to another by using the keyword `this`.
- Initializer lists allow you to check constructor parameters with `assert` and initialize field variables.
- Adding `const` to a constructor allows you to create immutable, canonical instances of the class.
- Factory constructors allow you to hide the implementation details of how you provide the class instance.

- Classes have getters and setters which you can customize without affecting how the object is used.

- Adding the `static` keyword to a property or method makes it belong to the class rather than the instance.