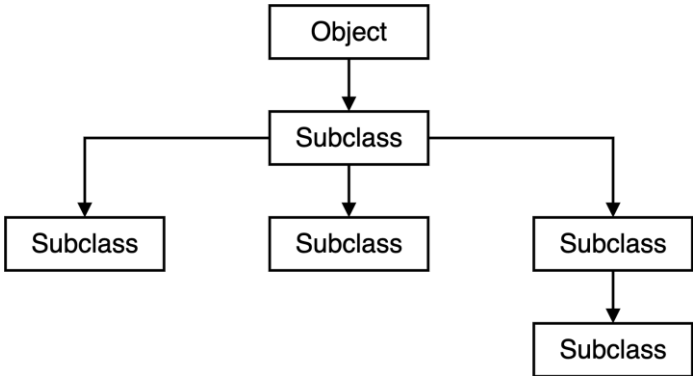
Chapter 9: Advanced Classes

Chapter 6 covered a lot of the foundational elements of classes, but now it's time to extend that knowledge. Object-oriented programming has captured developers' imaginations for decades. Just as the name `class` is inspired by biological notation, the true beauty of object-oriented programming is how you're able to elegantly build connective tissue between your classes. In this chapter you'll learn to use tools such as inheritance, interfaces, mixins and extension methods to move beyond simple coding and enter a world of software design.

Extending classes

In many situations, you'll need to create a hierarchy of classes that share some base functionality. You can create your own hierarchies by **extending classes**. This is also called **inheritance**, because the classes form a tree in which **child classes** inherit from **parent classes**. The parent and child classes are also called **super classes** and **subclasses** respectively. The `Object` class is the top superclass for all non-nullable types in Dart. All other classes (except `Null`) are subclasses of `Object`.



Note: Although there is no named top type in Dart, since all non-nullable Dart types derive from the `Object` type and `Object` itself is a subtype of the nullable `Object?` type, `Object?` can be considered in practice to be the root of the type system.

Creating your first subclass

To see how inheritance works, you'll create your own hierarchy of classes. In a little while, you'll make a `Student` class which needs grades, so first make a `Grade` enum:

```
enum Grade { A, B, C, D, F }
```

Creating similar classes

Next create two classes named `Person` and `Student`.

Here's `Person`:

```
class Person {
  Person(this.givenName, this.surname);

  String givenName;
  String surname;
  String get fullName => "$givenName $surname";

  @override
  String toString() => fullName;
}
```

And this is `Student`:

```
class Student {
  Student(this.givenName, this.surname);

  String givenName;
  String surname;
  var grades = <Grade>[];
  String get fullName => "$givenName $surname";

  @override
  String toString() => fullName;
}
```

Naturally, the `Person` and `Student` classes are very similar, since students are in fact persons. The only difference at the moment is that a `Student` will have a list of grades.

Subclassing to remove code duplication

You can remove the duplication between `Student` and `Person` by making `Student` **`extends`** `Person`. You do so by

adding `extends Person` after the class name, and removing everything but the `Student` constructor and the `grades` list.

Replace the `Student` class with the following code:

```
class Student extends Person {
  Student(String givenName, String surname)
    : super(givenName, surname);

  var grades = <Grade>[];
}
```

There are a few points to pay attention to:

- The constructor parameter names don't refer to `this` anymore. Whenever you see the keyword `this`, you should remember that `this` refers to the current object, which in this case would be an instance of the `Student` class. Since `Student` no longer contains the field names `givenName` and `surname`, using `this.givenName` or `this.surname` would have nothing to reference.

- In contrast to `this`, the `super` keyword is used to refer one level up the hierarchy. Similar to the forwarding constructor that you learned about in Chapter 6, using `super(givenName, surname)` passes the constructor parameters on to another constructor. However, since you're using `super` instead of `this`, you're forwarding the parameters to the parent class's constructor, that is, to the constructor of `Person`.

Calling super last in an initializer list

As a quick side note, if you use an initializer list, the call to `super` always goes last, that is, after `assert` statements and initializers. You can see the order in the following example:

```
class SomeChild extends SomeParent {

  SomeChild(double height)
      : assert(height != 0),   // assert
        _height = height,      // initializer
        super();               // super

  final double _height;
}
```

In this example, calling `super()` is actually unnecessary, because Dart always calls the default constructor for the super class if there are no arguments to pass. The reason that you or Dart always need to make the `super` call is to ensure that all of the field values have finished initializing.

Using the classes

OK, back to the primary example. Create `Person` and `Student` objects like so:

```
final jon = Person('Jon', 'Snow');
final jane = Student('Jane', 'Snow');
print(jon.fullName);
print(jane.fullName);
```

Run that and observe that both have full names:

```
Jon Snow
Jane Snow
```

The `fullName` for `Student` is coming from the `Person` class.

If you have a grade, you can only add that grade to the `Student` and not to the `Person`, because only the `Student` has grades. Add the following two lines to `main`:

```
final historyGrade = Grade.B;
jane.grades.add(historyGrade);
```

The student `jane` now has one grade in the `grades` list.

Overriding parent methods

Suppose you want the student's full name to print out differently than the default way it's printed in `Person`. You can do so by **overriding** the `fullName` getter. Add the following two lines to the bottom of the `Student` class:

```
@override
String get fullName => '$surname, $givenName';
```

You've seen the `@override` annotation before in this book with the `toString` method. While using `@override` is technically optional in Dart, it does help in that the compiler will give you an error if you think you're overriding something that doesn't actually exist in the parent class.

Run the code now and you'll see the student's full name printed differently than the parent's.

```
Jon Snow
Snow, Jane
```

Calling super from an overridden method

As another aside, sometimes you override methods of the parent class because you want to *add* functionality, rather than replace it, as you did above. In that case, you usually make a call to super either at the beginning or end of the overridden method.

Have a look at the following example:

```
class SomeParent {
  void doSomeWork() {
    print('parent working');
  }
}

class SomeChild extends SomeParent {
  @override
  void doSomeWork() {
    super.doSomeWork();
    print('child doing some other work');
  }
}
```

Since doSomeWork in the child class makes a call to super.doSomeWork, both the parent and the child methods run. So if you were to call the child method like so:

```
final child = SomeChild();
child.doSomeWork();
```

You would see the following result:

```
parent working
child doing some other work
```

The parent method's work was done first, since you had the super call at the *beginning* of the overridden method in the child. If you wanted to do the child method's work first, though, you would put the super call at the *end* of the method, like so:

```
@override
void doSomeWork() {
  print('child doing some other work');
  super.doSomeWork();
}
```

Multi-level hierarchy

Back to the primary example again. Add more than one level to your class hierarchy by defining a class that extends from `Student`.

```
class SchoolBandMember extends Student {
  SchoolBandMember(String givenName, String surn
ame)
    : super(givenName, surname);
  static const minimumPracticeTime = 2;
}
```

`SchoolBandMember` is a `Student` that has a `minimumPracticeTime`. The `SchoolBandMember` constructor calls the `Student` constructor with the `super` keyword. The

Student constructor will, in turn, call the Person constructor.

Sibling classes

Create a sibling class to SchoolBandMember named StudentAthlete that also derives from Student.

```
class StudentAthlete extends Student {
  StudentAthlete(String givenName, String surnam
e)
      : super(givenName, surname);
  bool get isEligible =>
    grades.every((grade) => grade != Grade.F);
}
```

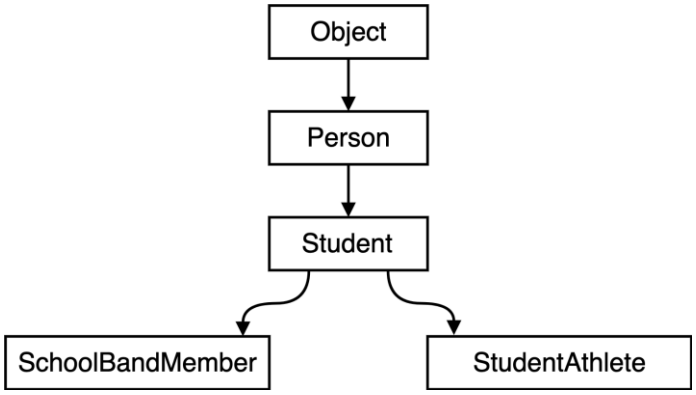
In order to remain eligible for athletics, a student athlete has an isEligible getter that makes sure the athlete has not failed any classes. The higher order method every on the grades list only returns true if every element of the list passes the given condition, which, in this case, means that none of the grades are F.

So now you can create band members and athletes.

```
final jessie = SchoolBandMember("Jessie", "Jones"
);
final marty = StudentAthlete("Marty", "McFly");
```

Visualizing the hierarchy

Here's what your class hierarchy looks like now:



You see that `SchoolBandMember` and `StudentAthlete` are both students, and all students are also persons.

Type inference in a mixed list

Since Jane, Jessie and Marty are all students, you can put them into a list.

```
final students = [jane, jessie, marty];
```

Recall that `jane` is a `Student`, `jessie` is a `SchoolBandMember` and `marty` is a `StudentAthlete`. Since they are all different types, what type is the list?

Hover your cursor over `students` to find out.

```
List<Student> students
final students = [jane, jessie, marty];
```

You can see that Dart has inferred the type of the list to be `List<Student>`. Dart used the most specific common ancestor as the type for the list. It couldn't use

SchoolBandMember or StudentAthlete, since that doesn't hold true for all elements of the list.

Checking an object's type at runtime

You can use the `is` and `is!` keywords to check whether a given object is or is not within the direct hierarchy of a class. Write the following code:

```
print(jessie is Object);
print(jessie is Person);
print(jessie is Student);
print(jessie is SchoolBandMember);
print(jessie is! StudentAthlete);
```

Knowing that `jessie` is a `SchoolBandMember`, first guess what Dart will show, and then run the code to see if you were right.

Ready? All five will print `true`, since `jessie` is `SchoolBandMember`, which is a subclass of `Student`, which is a subclass of `Person`, which is a subclass of `Object`. The only type that `jessie` is not, is `StudentAthlete` — which you confirmed by using the `is!` keyword.

> **Note** : The exclamation mark at the end of `is!` has nothing to do with the null assignment operator from null safety. It just means *not*.

Having an object be able to take multiple forms is known as **polymorphism**. This is a key part of object-oriented

programming. You'll learn to make polymorphic classes in an even more sophisticated way with abstract classes in just a bit.

First, though, a word of caution.

Prefer composition over inheritance

Now that you know about inheritance, you may feel ready to conquer the world. You can model anything as a hierarchy. Experience, though, will teach you that deep hierarchies are not always the best choice.

You may have already noticed this fact in the code above. For example, when you're overriding a method, do you need to call `super`? And if you do, should you call `super` at the beginning of the method, or at the end? Often the only way to know is to check the source code of the parent class. Jumping back and forth between levels of the hierarchy can make coding difficult.

Another problem with hierarchies is that they're tightly bound together. Changes to a parent class can break a child class. For example, say that you wanted to "fix" the `Person` class by removing `givenName` and replacing it with `firstName` and `middleName`.

Doing this would also require you to update, or refactor, all of the code that uses the subclasses as well. Even if you didn't remove `givenName`, but simply added `middleName`, users of classes like `StudentBandMember` would be affected without realizing it.

Tight coupling isn't the only problem. What if Jessie, who is a school band member, also decides to become an athlete? Do you make another class called `SchoolBandMemberAndStudentAthlete`? What if she joins the student union, too? Obviously, things could get out of hand quickly.

This has led many people to say, **prefer composition over inheritance**. The phrase means that, when appropriate, you should *add* behavior to a class rather than share behavior with an ancestor. It's more of a focus on what an object *has*, rather than what an object *is*. For example, you could flatten the hierarchy for `Student` by giving the student a list of roles, like so:

```
class Student {
  List<Role>? roles;
}
```

When you create a student, you could pass in the roles as a constructor parameter. This would also let you add and remove roles later. Of course, since Dart doesn't come with the `Role` type, you'd have to define it yourself. You'd need to make `Role` abstract enough so that a role could be a band member, an athlete or a student union member. You'll learn about making abstract classes like this in the next section of the chapter.

All this talk of composition isn't to say that inheritance is *always* bad. It might make sense to still have `Student` extend `Person`. Inheritance can be good when a subclass needs *all* of the behavior of its parent. However, when you only need

some of that behavior, you should consider passing in the behavior as a parameter, or perhaps even using a mixin, which you'll learn about later in this chapter.

> **Note**: The whole Flutter framework is organized around the idea of composition. You build your UI as a tree of widgets, where each widget *does* one simple thing and *has* zero or more child widgets that also do one simple thing. This type of architecture generally makes it easier to understand the purpose of a class.

Mini-exercises

1. Create a class named `Fruit` with a `String` field named `color` and a method named `describeColor`, which uses `color` to print a message.

2. Create a subclass of `Fruit` named `Melon` and then create two `Melon` subclasses named `Watermelon` and `Cantaloupe`.

3. Override `describeColor` in the `Watermelon` class to vary the output.

Abstract classes

The classes and subclasses you created in the last section were **concrete classes**. It's not that they're made of cement; it just means that you can make actual objects out of them.

That's in contrast to **abstract classes**, from which you can't make objects.

"What's the use of a class you can't make an object out of?" asks the pragmatist. "What are the use of ideas?" answers the philosopher. You deal with abstract concepts all the time, and you don't think about them at all.

Have you ever seen an animal? "Uh, are you seriously asking me that?" you answer. The question isn't "have you ever seen a chicken or a platypus or a mouse." Have you ever seen a generic animal, devoid of all features that are relevant to only one kind of animal — just the essence of "animal" itself? What would that even look like? It can't have four legs because ducks are animals and they have two legs. It can't have hair because rattlesnakes are animals and they don't have hair. Worms are animals, too, right? So there go the eyes and bones.

No one has seen an "animal" in the abstract sense, but everybody has seen concrete instances of things that fit the abstract animal category. Humans are good at generalizing and categorizing the observations they make, and honestly, these abstract ideas are very useful. They allow you to make short statements like "I saw a lot of animals at the zoo" instead of "I saw a lion, an elephant, a lemur, a shark, ..."

The same thing applies in object oriented programming. After making lots of concrete classes, you begin to notice patterns and more generalized characteristics of the classes you're writing. So when you come to the point of just wanting to describe the general characteristics and behavior

of a class without specifying the exact way that class is implemented, you're ready to write abstract classes. Don't be put off by the word "abstract". It's no more difficult than the idea of an animal.

Creating your own abstract classes

Have a go at working this out in Dart now. Without venturing too far into the fringes of how taxonomists make their decisions, create the following `Animal` class:

```
abstract class Animal {
  bool isAlive = true;
  void eat();
  void move();

  @override
  String toString() {
    return "I'm a $runtimeType";
  }
}
```

Here are a few important points about that code:

- The way you define an abstract class in Dart is to put the `abstract` keyword before `class`.
- In addition to the class itself being abstract, `Animal` also has two abstract methods: `eat` and `move`. You know they're abstract because they don't have curly braces; they just end with a semicolon.

- These abstract methods describe behavior that a subclass must implement but don't tell *how* to implement. That's up to the subclass. Leaving implementation details up to the subclass is a good thing because there are such a variety of ways to eat and move throughout the animal kingdom, so it would be almost impossible for `Animal` to specify anything meaningful here.

- Note that just because a class is abstract doesn't mean that it can't have concrete methods or data. You can see that `Animal` has a concrete `isAlive` field, with a default value of `true`. `Animal` also has a concrete implementation of the `toString` method, which belongs to the `Object` superclass. The `runtimeType` property also comes from `Object` and gives the object type at runtime.

Can't instantiate abstract classes

As mentioned, you can't create an instance of an abstract class. See for yourself by writing the following line:

```
final animal = Animal();
```

You'll see the following error:

```
Abstract classes can't be instantiated.
Try creating an instance of a concrete subtype.
```

Isn't that good advice! That's what you're going to do next.

Concrete subclass

Create a concrete `Platypus` now. Stop thinking about cement. Just add the following empty class to your IDE below your `Animal` class:

```
class Platypus extends Animal {}
```

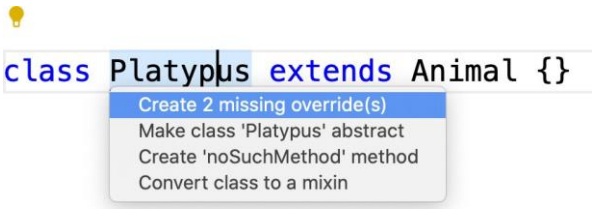
Immediately you'll notice the wavy red line:

```
class Platypus extends Animal {}
```

That's not because you spelled platypus wrong. It really does have a *y*. Rather, the error is because when you extend an abstract class, you must provide an implementation of any abstract methods, which in this case are `eat` and `move`.

Adding the missing methods

You could write the methods yourself, but VS Code gives you a shortcut. Put your cursor on `Platypus` and press `Command+.` on a Mac or `Control+.` on a PC. You'll see the following pop-up:



To quickly add the missing methods, choose **Create 2 missing override(s)**.

This will give you the following code:

```
class Platypus extends Animal {
  @override
  void eat() {
    // TODO: implement eat
  }

  @override
  void move() {
    // TODO: implement move
  }
}
```

Starting a comment with `TODO:` is a common way to mark parts of your code where you need to do more work. Later, you can search your entire project in VS Code for the remaining TODOs by pressing **Command+Shift+F** on a Mac or **Control+Shift+F** on a PC and writing "TODO" in the search box. You're going to complete these TODOs right now, though.

Filling in the TODOs

Since this is a concrete class, it needs to provide the actual implementation of the `eat` and `move` methods. In the `eat` method body, add the following line:

```
print('Munch munch');
```

A platypus may not have teeth, but it should still be able to munch.

In the move method add:

```
print('Glide glide');
```

As was true with subclassing normal classes, abstract class subclasses can also have their own unique methods. Add the following method to `Platypus`:

```
void layEggs() {
  print('Plop  plop');
}
```

Readers who are well-acquainted with how platypuses (Or is it platypi?) eat, swim and give birth can make additional word suggestions for the next edition of this book.

Testing the results

Test your code out now in `main`:

```
final platypus = Platypus();
print(platypus.isAlive);
platypus.eat();
platypus.move();
platypus.layEggs();
print(platypus);
```

Run the code to see the following:

```
true
Munch munch
Glide  glide
Plop plop
I'm a Platypus
```

Look at what this tells you:

- A concrete class has access to concrete data, like `isAlive`, from its abstract parent class.
- Dart recognized that the runtime type was `Platypus`, even though `runtimeType` comes from `Object`, and was accessed in the `toString` method of `Animal`.

Treating concrete classes as abstract

There is one more interesting thing to do before moving on. In the line where you declared `platypus`, hover your cursor over the variable name:

```
Platypus platypus
final platypus = Platypus();
```

Dart infers `platypus` to be of type `Platypus`. That's normal, but here's the interesting part. Replace that line with the following one, adding the `Animal` type annotation:

```
Animal platypus = Platypus();
```

Hover your cursor over platypus again:

```
Animal platypus
Animal platypus = Platypus();
```

Now Dart sees `platypus` as merely an `Animal` with no more special ability to lay eggs. Comment out the line calling the `layEggs` method:

```
// platypus.layEggs();
```

Run the code again paying special attention to the `print(platypus)` results:

```
I'm a Platypus
```

So at compile time, Dart treats `platypus` like an `Animal` even though at runtime Dart knows it's a `Platypus`. This is useful when you don't care about about the concrete implementation of an abstract class, but you only care that it's an `Animal` with `Animal` characteristics.

Now, you're probably thinking, "Making animal classes is very cute and all, but how does this help me save data on the awesome new social media app I'm making?" That's where interfaces come in.

Interfaces

Interfaces are similar to abstract classes in that they let you define the behavior you expect for all classes that implement the interface. They're a means of hiding the implementation

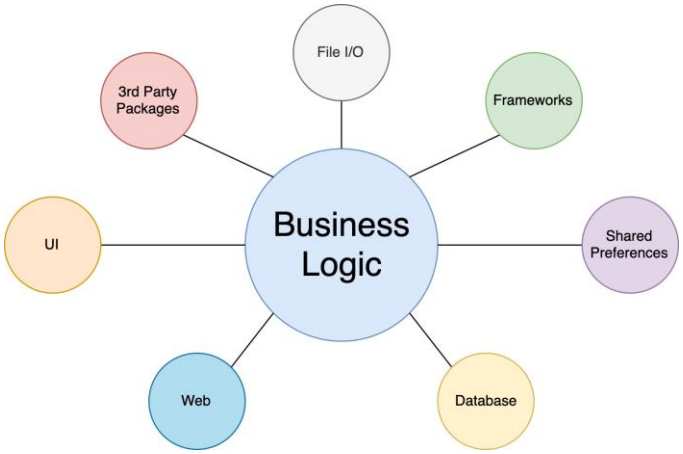
details of the concrete classes from the rest of your code. Why is that important? To answer that question it's helpful to understand a little about architecture. Not the Taj Mahal kind of architecture, software architecture.

Software architecture

When you're building an app, your goal should be to keep core business logic separate from infrastructure like the UI, database, network and third-party packages. Why? The core business logic doesn't change frequently, while the infrastructure often does. Mixing unstable code with stable would cause the stable code to become unstable.

> Note: **Business logic**, which is sometimes called **business rules** or **domain logic**, refers to the essence of what your app does. The business logic of a calculator app would be the mathematical calculations themselves. Those calculations don't depend on what your UI looks like or how you store the answers.

The following image shows an idealized app with the stable business logic in the middle and the more volatile infrastructure parts surrounding it:



The UI shouldn't communicate directly with the web. You also shouldn't scatter direct calls to the database across your app. Everything goes through the central business logic. In addition to that, the business logic shouldn't know any implementation details about the infrastructure.

This gives you a plug-in style architecture, where you can swap one database framework for another and the rest of the app won't even know anything changed. You could replace your mobile UI with a desktop UI, and the rest of the app wouldn't care. This is useful for building scalable, maintainable and testable apps.

Communication rules

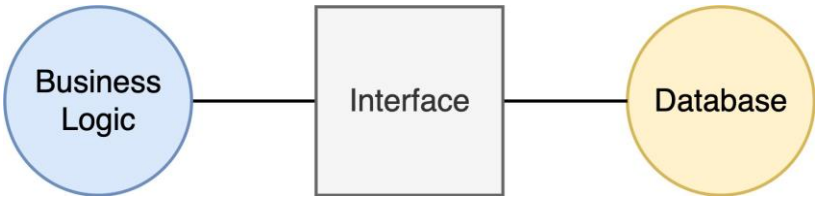
Here's where interfaces come in. An **interface** is a description of how communication will be managed between two parties. A phone number is a type of interface. If you want to call your friend, you have to dial your friend's phone number. Dialing a different number won't work. Another word for interface is **protocol**, as in Internet Protocol or Hypertext

Transfer Protocol. Those protocols are the rules for how communication happens among the users of the protocol.

When you create an interface in Dart, you define the rules for how one part of your codebase will communicate with another part. As long as both parts follow the interface rules, each part can change independently of the other. This makes your app much more manageable. In team settings, interfaces also allow different people to work on different parts of the codebase without worrying that they're going to mess up someone else's code.

Separating business logic from infrastructure

In the image below, you can see the interface is between the business logic and the code for accessing the database.



The business logic doesn't know anything about the database. It's just talking to the interface. That means you could even swap out the database for a completely different form of storage like cloud storage or file storage. The business logic doesn't care.

There's a famous adage related to this that goes, **code against interfaces, not implementations**. You define an interface, and then you code your app to use that interface only. While you must implement the interface with concrete

classes, the rest of your app shouldn't know anything about those concrete classes, only the interface.

Creating an interface

There's no `interface` keyword in Dart. Instead, you can use any class as an interface. Since only the field and method names are important, most interfaces are made from abstract classes that contain no logic.

Say you want to make a weather app, and your business logic needs to get the current temperature in some city. Since those are the requirements, your Dart interface class would look like this:

```
abstract class DataRepository {
  double? fetchTemperature(String city);
}
```

Note that **repository** is a common term to call an interface that hides the details of how data is stored and retrieved. Also, the reason the result of `fetchTemperature` is nullable is that someone might ask for the temperature in a city that doesn't exist.

Implementing the interface

The Dart class above was just a normal abstract class, like the one you made earlier. However, when creating a concrete class to implement the interface, you must use the `implements` keyword instead of the `extends` keyword.

Add the following concrete class:

```
class FakeWebServer implements DataRepository {
  @override
  double? fetchTemperature(String city) {
    return 42.0;
  }
}
```

Here are a couple of points to note:

- Besides the benefits mentioned previously, another great advantage of using an interface is that you can create mock implementations to temporarily replace real implementations. In the `FakeWebServer` class, you are simply returning a random number instead of going to all the work of contacting a real server. This allows you to have a "working" app until you can get around to writing the code to contact the web server. This is also useful when you're testing your code and you don't want to wait for a real connection to the server.

- Speaking of waiting for a web server, a real interface would return a type of `Future<double?>` instead of returning `double?` directly. However, you haven't read Chapter 10 about asynchronous programming yet, so this example omits the `Future` part.

Using the interface

How do you use the interface on the business logic side? Remember that you can't instantiate an abstract class, so this won't work:

```
final repository = DataRepository();
```

You could potentially use the `FakeWebServer` implementation directly like so:

```
final DataRepository repository = FakeWebServer
();
final temperature = repository.fetchTemperature(
"Berlin");
```

But this defeats the whole point of trying to keep the implementation details separate from the business logic. When you get around to swapping out the `FakeWebServer` with another class, you'll have to go back and make updates at every place in your business logic that mentions it.

Adding a factory constructor

Do you remember factory constructors from Chapter 6? If you do, you'll recall that factory constructors can return subclasses. Add the following line to your interface class:

```
factory DataRepository() => FakeWebServer();
```

Your interface should look like this now:

```
abstract class DataRepository {
  factory DataRepository() => FakeWebServer();
  double? fetchTemperature(String city);
}
```

Since `FakeWebServer` is a subclass of `DataRepository`, the factory constructor is allowed to return it. The neat trick is that by using an unnamed constructor for the factory, you can make it look like it's possible to instantiate the class now.

Write the following in `main`:

```
final repository = DataRepository();
final temperature = repository.fetchTemperature(
"Manila");
```

Ah, now your code on this side has no idea that that repository is actually `FakeWebServer`. When it comes time to swap in the real implementation, you only need to update the subclass returned by the factory constructor in the `DataRepository` interface.

Note: In the code above, you used a factory to return the concrete implementation of the interface. There are other options, though. Do a search for **service locators**, of which the `get_it` package is a good example, and **dependency injection** to learn more about this topic.

Interfaces and the Dart SDK

If you browse the Dart source code, which you can do by Command or Control-clicking Dart class names like `int` or `List` or `String`, you'll see that Dart makes heavy use of interfaces to define its API. That allows the Dart team to change the implementation details without affecting

developers. The only time developers are affected is when the interface changes.

This concept is key to the flexibility that Dart has as a language. The Dart VM implements the interface one way and gives you the ability to hot-reload your Flutter apps. The `dart compile js` tool implements the interface using JavaScript and gives you the ability to run your code on the web. The `dart compile exe` tool implements the interface on Windows or Linux or Mac to let you run your code on those platforms.

The implementation details are different for every platform, but you don't have to worry about that, because your code will only talk to the interface, not to the platform. Are you starting to see how powerful interfaces can be?

Extending vs implementing

There are a couple of differences between `extends` and `implements`. Dart only allows you to extend a single superclass. This is known as **single inheritance**, in contrast with other languages that allow **multiple inheritance**.

So the following is not allowed in Dart:

```
class MySubclass extends OneClass, AnotherClass
{} // Not OK
```

However, you can implement more than one interface:

```
class MyClass implements OneClass, AnotherClass
{} // OK
```

You can also combine `extends` and `implements`:

```
class MySubclass extends OneClass implements Ano
therClass {}
```

But what's the difference between just extending or implementing? That is, how are these two lines different?

```
class SomeClass extends AnotherClass {}
class SomeClass implements AnotherClass {}
```

When you extend `AnotherClass`, `SomeClass` has access to any logic or variables in `AnotherClass`. However, if `SomeClass` *implements* `AnotherClass`, `SomeClass` must provide its own version of all methods and variables in `AnotherClass`.

Example of extending

Assume `AnotherClass` looks like the following:

```
class AnotherClass {
  int myField = 42;
  void myMethod() => print(myField);
}
```

You can extend it like this with no issue:

```
class SomeClass extends AnotherClass {}
```

Check that SomeClass objects have access to AnotherClass data and methods:

```
final someClass = SomeClass();
print(someClass.myField);          // 42
someClass.myMethod();              // 42
```

Run that and you'll see 42 printed twice.

Example of implementing

Using implements in the same way doesn't work:

```
class SomeClass implements AnotherClass {} // No
t OK
```

The implements keyword tells Dart that you only want the field types and method signatures. You'll provide the concrete implementation details for everything yourself. How you implement it is up to you, as demonstrated in the following example:

```
class SomeClass implements AnotherClass {
  @override
  int myField = 0;

  @override
  void  myMethod()  =>  print("Hello");
}
```

Test that code again as before:

```
final someClass = SomeClass();
print(someClass.myField);          // 0
someClass.myMethod();              // Hello
```

This time you see your custom implementation results in `0` and `Hello`.

Mini-exercises

1. Create an interface called `Bottle` and add a method to it called `open`.

2. Create a concrete class called `SodaBottle` that implements `Bottle` and prints "Fizz fizz" when `open` is called.

3. Add a factory constructor to `Bottle` that returns a `SodaBottle` instance.

4. Instantiate `SodaBottle` by using the `Bottle` factory constructor and call `open` on the object.

Mixins

Mixins are an interesting feature of Dart that you might not be familiar with, even if you know other programming languages. They're a way to reuse methods or variables among otherwise unrelated classes.

> **Note**: For you Swift developers, Dart mixins work like protocol extensions.

Before showing you what mixins look like, first you'll take a look at why you need them.

Problems with extending and implementing

Think back to the `Animal` examples again. Say you've got a bunch of birds, so you're carefully planning an abstract class to represent them. Here's what you come up with:

```
abstract class Bird {
  void fly();
  void layEggs();
}
```

"It's looking good!" you think. "I'm getting the hang of this." So you try it out on `Robin`:

```
class Robin extends Bird {
  @override
  void fly() {
    print('Swoosh swoosh');
  }

  @override
  void layEggs() {
    print('Plop plop');
  }
}
```

“Perfect!” You smile contentedly at your handiwork.

Then you hear a sound behind you.

“Munch, munch. Glide, glide. Plop, plop. I’m a platypus.”

Oh. Right. The platypus.

Your `layEggs` code for `Robin` is exactly the same as it is for `Platypus`. That means you’re duplicating code, which violates the DRY principle. If there are any future changes to `layEggs`, you’ll have to remember to change both instances. Consider your options:

1. Platypus can’t extend `Bird` or `Robin`, because platypi can’t fly.

2. Birds probably shouldn’t extend `Platypus`, because who knows when you’re going to add the `stingWithVenomSpur` method?

3. You could create an `EggLayer` class and have `Bird` and `Platypus` both extend that. But then what about flying? Make a `Flyer` class, too? Dart only allows you to extend one class, so that won’t work.

4. You could have birds implement `EggLayer` and `Flyer` while Platypus implements only `EggLayer`. But then you’re back to code duplication, since implementing requires you to supply the implementation code for every class.

The solution? Mixins!

Mixing in code

To make a **mixin**, you take whatever concrete code you want to share with different classes, and package it in its own special mixin class.

Write the following two mixins:

```
mixin EggLayer {
  void layEggs() {
    print("Plop plop");
  }
}

mixin Flyer {
  void fly() {
    print("Swoosh  swoosh");
  }
}
```

The `mixin` keyword indicates that these classes can *only* be used as mixins. You can use any class as a mixin, though, so if you wanted to use `EggLayer` as a normal class, then just replace the `mixin` keyword with `class` or `abstract class`. In fact, the `mixin` keyword is a fairly new addition to Dart, so you may still see legacy code that just uses regular classes as mixins even though those classes aren't needed as standalone classes.

Now refactor `Robin` as follows, using the `with` keyword to identify the mixins:

```
class Robin extends Bird with EggLayer, Flyer {}
```

There are two mixins, so you separate them with a comma. Since those two mixins contain all the code that `Bird` needs, the class body is now empty.

Refactor `Platypus` as well:

```
class Platypus extends Animal with EggLayer {
  @override
  void eat() {
    print("Munch munch");
  }

  @override
  void move() {
    print("Glide glide");
  }
}
```

The `layEggs` logic has moved to the mixin. Now both `Robin` and `Platypus` share the code that the `EggLayer` mixin contains. Just to make sure it works, run the following code:

```
final platypus = Platypus();
final robin = Robin();
platypus.layEggs();
robin.layEggs();
```

Four plops, and all is well.

Mini-exercises

1. Create a class called `Calculator` with a method called `sum` that prints the sum of any two integers you give it.

2. Extract the logic in `sum` to a mixin called `Adder`.

3. Use the mixin in `Calculator`.

Extension methods

Up to this point in the chapter, you've been writing your own classes and methods. Often, though, you use other people's classes when you're programming. Those classes may be part of a core Dart library, or they may be from packages that you got off Pub. In either case, you don't have the ability to modify them at will.

However, Dart has a feature called **extension methods** that allow you to add functionality to existing classes. Even though they're called extension *methods*, you can also add other members like getters, setters or even operators.

Extension syntax

To make an extension, you use the following syntax:

```
extension on SomeClass {
  // your custom code
}
```

This should be located at the top-level in a file; that is, not inside another class or function. Replace `SomeClass` with whatever class you want to add extra functionality to.

You may give the extension itself a name if you like. In that case the syntax is as follows:

```
extension YourExtensionName on ClassName {
  // your custom code
}
```

You can use whatever name you like in place of `YourExtensionName`. The name is only used to show or hide the extension when importing it in another library.

Have a look at a few of the following examples to see how extension methods work.

String extension example

Did you ever make secret codes when you were a kid, like `a=1`, `b=2`, `c=3`, and so on? For this example, you're going to make an extension that will convert a string into a secret coded message. Then you'll add another extension method to decode it.

In this secret code, each letter will be bumped up to the next one. So **a** will be **b**, **b** will be **c**, and so on. To accomplish that you'll increase the Unicode value of each code point in the input string by 1. If the original message were "abc", the encoded message should be "bcd".

Solving in the normal way

First, solve the problem as you would with a normal function.

```
String encode(String input) {
  final output = StringBuffer();
  for (final codePoint in input.runes) {
    output.writeCharCode(codePoint + 1);
  }
  return output.toString();
}
```

This function uses a `StringBuffer` for efficient string manipulation. A normal `String` is immutable, but a `StringBuffer` is mutable. That means your function doesn't have to create a new string every time you append a character. You loop through each Unicode code point and increment it by 1 before writing it to output. Finally, you convert the `StringBuffer` back to a regular `String` and return it.

Test it out:

```
final original = "abc";
final secret = encode(original);
print(secret);
```

The result is bcd. It works!

Converting to an extension

The next step is to convert the encode function above to an extension so that you can use it like so:

```
final secret = "abc".encoded;
```

Since this extension won't mutate the original string itself, remember the naming convention of using an adjective rather than a commanding verb. That's the reason for choosing `encoded`, rather than `encode`, for the extension name.

Add the following code somewhere outside the `main` method:

```
extension on String {
  String get encoded {
    final output = StringBuffer();
    for (final codePoint in runes) {
      output.writeCharCode(codePoint + 1);
    }
    return output.toString();
  }
}
```

Look at what's changed here:

- The keywords `extension on` are what make this an extension. You can add whatever you want inside the body. It's as if `String` were your own class now.
- Rather than making a normal method, you can use a getter method. This makes it so that you can call the extension using `encoded`, without the parentheses, rather than `encoded()`.
- Since you're inside `String` already, there's no need to pass `input` as an argument. If you need a reference to the string object, you can use the `this` keyword. Thus, instead of `input.runes`, you could write `this.runes`.

However, this is unnecessary and you can directly access runes. Remember that runes is a member of String and you're inside String.

Check that the extension works:

```
final secret = "abc".encoded;
print(secret);
```

You should see bcd as the output. Nice! It still works.

Adding a decode extension

Add the decoded method inside the body of the String extension as well:

```
String get decoded {
  final output = StringBuffer();
  for (final codePoint in runes) {
    output.writeCharCode(codePoint - 1);
  }
  return output.toString();
}
```

If you compare this to the encoded method, though, there's a lot of code duplication. Whenever you see code duplication, you should think about how to make it DRY.

Refactoring to remove code duplication

Refactor your String extension by replacing the entire extension with the following:

```
extension on String {
  String get encoded {
    return _code(1);
  }
  String get decoded {
    return _code(-1);
  }
  String _code(int step) {
    final output = StringBuffer();
    for (final codePoint in runes) {
      output.writeCharCode(codePoint + step);
    }
    return output.toString();
  }
}
```

Now the private _code method factors out all of the common parts of encoded and decoded. That's better.

Testing the results

To make sure that everything works, test both methods like so:

```
final original = 'I like extensions!';
final secret = original.encoded;
final revealed = secret.decoded;
print(secret);
print(revealed);
```

This will display the following encoded and decoded messages:

```
J!mjlf!fyufotjpot"
I like extensions!
```

Great! Now you can amuse your friends by giving them encoded messages. They're actually a lot of fun to solve.

int extension example

Here's an example for an extension on int.

```
extension on int {
  int get cubed {
    return this * this * this;
  }
}
```

Notice the use of this to get a reference to the int object, which will be 5 in the example below.

You use the extension like so:

```
print(5.cubed);
```

The answer is 125.

Enum extension example

Dart enums, which you learned about in Chapter 4, are pretty basic in themselves. However, with the power of extensions, you can do much more with them.

First define an enum like so:

```
enum ProgrammingLanguage { dart, swift, javaScri
pt }
```

Normally you wouldn't be able to perform any internal logic on those enum values, but you can by adding the following extension on `ProgrammingLanguage`:

```
extension on ProgrammingLanguage {
  bool get isStronglyTyped {
    switch (this) {
      case ProgrammingLanguage.dart:
      case  ProgrammingLanguage.swift:
        return true;
      case ProgrammingLanguage.javaScript:
        return false;
      default:
        throw Exception("Unknown Programming Lan
guage $this");
    }
  }
}
```

Now you can check at runtime whether a particular language is strongly typed or not:

```
final language = ProgrammingLanguage.dart;
print(language.isStronglyTyped);
```

Run that and you'll see `true` printed to the console.

As you can see, you can do a lot with extensions. Although they can be very powerful, extensions by definition add non-standard behavior, and this can make it harder for other

developers to understand your code. Use extensions when they make sense, but try not to overuse them.

Challenges

Before moving on, here are some challenges to test your knowledge of advanced classes. 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: Heavy monotremes

Dart has a class named `Comparable`, which is used by the the `sort` method of `List` to sort its elements. Add a `weight` field to the `Platypus` class you made in this lesson. Then make `Platypus` implement `Comparable` so that when you have a list of `Platypus` objects, calling `sort` on the list will sort them by weight.

Challenge 2: Fake notes

Design an interface to sit between the business logic of your note-taking app and a SQL database. After that, implement a fake database class that will return mock data.

Challenge 3: Time to code

Dart has a `Duration` class for expressing lengths of time. Make an extension on `int` so that you can express a duration like so:

```
final timeRemaining = 3.minutes;
```

Key points

- A subclass has access to the data and methods of its parent class.
- You can create a subclass of another class by using the `extends` keyword.
- A subclass can override its parent's methods or properties to provide custom behavior.
- Dart only allows single inheritance on its classes.
- Abstract classes define class members and may or may not contain concrete logic.
- Abstract classes can't be instantiated.
- One rule of clean architecture is to separate business logic from infrastructure logic like the UI, storage, third- party packages and the network.
- Interfaces define a protocol for code communication.
- Use the `implements` keyword to create an interface.
- Mixins allow you to share code between classes.
- Extension methods allow you to give additional functionality to classes that are not your own.