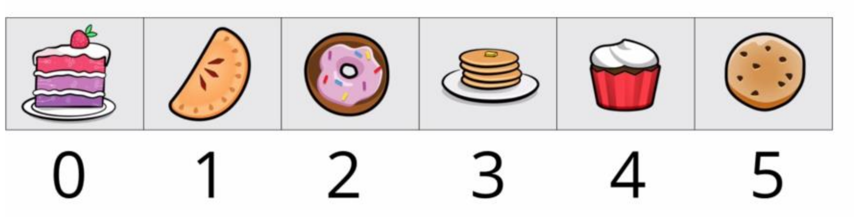
Chapter 8: Collections

In almost every application you make, you'll be dealing with collections of data. Data can be organized in multiple ways, each with a different purpose. Dart provides multiple solutions to fit your collection's needs, and in this chapter you'll learn about three of the main ones: lists, sets and maps.

Lists

Whenever you have a very large collection of objects of a single type that have an ordering associated with them, you'll likely want to use a list as the data structure for ordering the objects. Lists in Dart are similar to arrays in other languages.

The image below represents a list with six **elements**. Lists are zero-based, so the first element is at **index** `0`. The **value** of the first element is `cake`, the value of the second element is `pie`, and so on until the last element at index `5`, which is `cookie`.



The order of a list matters. Pie is after cake, but before donut. If you loop through the list multiple times, you can be sure the elements will stay in the same location and order.

Basic list operations

You'll start by learning how to create lists and modify elements.

Creating a list

You can create a list by specifying the initial elements of the list within square brackets. This is called a **list literal**.

```
var desserts = ['cookies', 'cupcakes', 'donuts',
'pie'];
```

Since all of the elements in this list are strings, Dart infers this to be a list of `String` types.

You can reassign desserts (but why would one ever want to reassign desserts?) with an empty list like so:

```
desserts = [];
```

Dart still knows that desserts is a list of strings. However, if you were to initialize a new empty list like this:

```
var snacks = [];
```

Dart wouldn't have enough information to know what kind of objects the list should hold. In this case, Dart simply infers it to be a list of `dynamic`. This causes you to lose type safety, which you don't want. If you're starting with an empty list, you should specify the type like so:

```
List<String> snacks = [];
```

There are a couple of details to note here:

- `List` is the data type, or class name, as you learned in Chapter 6.
- The angle brackets `< >` here are the notation for **generic types** in Dart. A generic list means you can have a list of anything; you just put the type you want inside the angle brackets. In this case, you have a list of strings, but you could replace `String` with any other type. For example, `List<int>` would make a list of integers, `List<bool>` would make a list of Booleans, and `List<Grievance>` would make a list of grievances — but you'd have to define that type yourself since Dart doesn't come with any by default.

A slightly nicer syntax for creating an empty list is to use `var` or `final` and move the generic type to the right:

```
var snacks = <String>[];
```

Dart still has all the information it needs to know this is an empty list of type `String`.

Printing lists

As you can do with any collection, you can print the contents of a list by using the `print` statement. Since `desserts` is currently empty, give it a list with elements again so that you have something interesting to show when you print it out.

```
desserts = ['cookies', 'cupcakes', 'donuts', 'pi
e'];
print(desserts);
```

Run that and you'll see the following:

```
[cookies, cupcakes, donuts, pie]
```

Accessing elements

To access the elements of a list, you reference its index via **subscript notation**, where the index number goes within square brackets after the list name.

```
final secondElement = desserts[1];
print(secondElement);
```

Don't forget that lists are zero-based, so index 1 fetches the second element. Run that code and you'll see `cupcakes` as expected.

If you know the value but don't know the index, you can use the `indexOf` method to look it up:

```
final  index  =  desserts.indexOf('pie');
final value = desserts[index];
```

Since `'pie'` is the fourth item in the zero-based list, `index` is 3 and `value` is `pie`.

Assigning values to list elements

Just as you access elements, you also assign values to specific elements using subscript notation:

```
desserts[1] = 'cake';
```

This changes the value at index 1 from cupcakes to cake.

Adding elements to a list

Lists are growable by default in Dart, so you can use the add method to add an element.

```
desserts.add('brownies');
print(desserts);
```

Run that and you'll see:

```
[cookies, cake, donuts, pie, brownies]
```

Now desserts has five elements and the last one is brownies.

Removing elements from a list

You can remove elements using the remove method. So if you'd gotten a little hungry and eaten the cake, you'd write:

```
desserts.remove('cake');
print(desserts);
```

This leaves a list with four elements:

```
[cookies, donuts, pie, brownies]
```

No worries, there's still plenty for a midnight snack tonight!

Mutable and immutable lists

In the examples above, you were able to reassign list literals to desserts like so:

```
var desserts = ['cookies', 'cupcakes', 'donuts',
'pie'];
desserts = [];
desserts = ['cookies', 'cupcakes', 'donuts', 'pi
e'];
```

The reason you could do that is because you defined desserts using the var keyword. This has nothing to do with the list itself being immutable or not. It only means that you can swap out *different* lists in desserts.

Now try the following using final:

```
final desserts = ['cookies', 'cupcakes', 'donuts'
, 'pie'];
desserts = [];                     // not allowed
desserts = ['cake', 'ice cream']; // not allowed
desserts = someOtherList;          // not allowed
```

Unlike var, using final means that you're not allowed to use the assignment operator to give desserts a new list.

However, look at this:

```
final desserts = ['cookies', 'cupcakes', 'donuts'
, 'pie'];
desserts.remove('cookies');  // OK
desserts.remove('cupcakes');// OK
desserts.add('ice cream');    // OK
```

Obviously, the `final` keyword isn't keeping you from changing the contents of the list elements. What's happening?

Perhaps a little story will help.

The House on Wenderlich Way

You live in a house at 321 Lonely Lane. All you have at home are a few brownies, which you absentmindedly munch on as you scour the internet in hopes of finding work. Finally, you get a job as a senior Flutter developer, so you buy a new house at 122 Wenderlich Way. Best of all, your neighbor Ray brings over some cookies, cupcakes, donuts and pie as a house warming gift! The brownies are still at your old place, but in your excitement about the move you've forgotten all about them.

Using `var` is like giving you permission to move houses. The first house had brownies. The second house had cookies, cupcakes, donuts and pie. Different houses, different desserts.

Using `final`, on the other hand, is like saying, "Here's your house, but this is the last place you can ever live." However, just because you live at a fixed location doesn't mean that you can't change what's inside the house. You might live permanently at 122 Wenderlich Way, but it's fine to eat all the cookies and cupcakes in the house and then go to the store and bring home some ice cream. Well, it's fine in that it's permissible, but maybe you should pace yourself a little on the sweets.

So too with a `final` list. Even though the memory address is constant, the values at that address are mutable. Mutable data is nice and all, but when you open your cupboard expecting to find donuts but instead discover the neighbor kids traded them for slugs, it's not so pleasant. It's the same with lists — sometimes you just don't want to be surprised.

So how do you get an immutable list? Have you already guessed the answer? Good job if you have!

Creating deeply immutable lists

The solution to creating an immutable list is to mark the variable name with the `const` keyword. This forces the list to be **deeply immutable**.

That is, every element of the list must also be a compile-time constant.

```
const desserts = ['cookies', 'cupcakes', 'donut
s', 'pie'];
desserts.add('brownie'); // not allowed
desserts.remove('pie'); // not allowed
desserts[0] = 'fudge';  // not allowed
```

Since const precedes desserts in the example above, you're not allowed to add to, remove from, or update the list.

If you aren't able to use const for the variable itself, you can still make the value deeply immutable by adding the optional const keyword before the value.

```
final desserts = const ['cookies', 'cupcakes', 'd
onuts', 'pie'];
```

Finally, if you want an immutable list but you won't know the element values until runtime, then you can create one with the `List.unmodifiable` named constructor:

```
final modifiableList = [DateTime.now(), DateTime.n
ow()];
final unmodifiableList = List.unmodifiable(modifiabl
eList);
```

`DateTime.now()` returns the date and time when it's called. You're obviously not going to know that until runtime, so this prevents the list from taking const. Passing that list into `List.unmodifiable`, however, makes the new list immutable.

Note : Unfortunately, inadvertently trying to modify an unmodifiable list will cause a runtime error — not a compile-time error. So while mutable data can be unsafe, so too can unmodifiable lists. A good practice is to write tests to ensure your code works as intended.

That's enough about mutability for now. Next you'll see some properties you can access on lists.

List properties

Collections such as `List` have a number of properties. To demonstrate them, use the following list of drinks.

```
const drinks = ['water', 'milk', 'juice', 'soda'
];
```

Accessing first and last elements

You can access the first and last element in a list:

```
drinks.first   // water
drinks.last    // soda
```

Checking if a list contains any elements

You can also check whether a list is empty or not empty.

```
drinks.isEmpty    // false
drinks.isNotEmpty // true
```

This is equivalent to the following:

```
drinks.length == 0 // false
drinks.length > 0   // true
```

However, it's more readable to use `isEmpty` and `isNotEmpty`.

Looping over the elements of a list

For this section you can return to your list of desserts:

```
const desserts = ['cookies', 'cupcakes', 'donut
s', 'pie'];
```

In Chapter 4, you saw how to iterate over lists, so this is a review of the `for-in` loop.

```
for (var dessert in desserts) {
  print(dessert);
}
```

Each time through the loop, `dessert` is assigned an element from `desserts`.

You also saw how to use `forEach` with an anonymous function.

```
desserts.forEach((dessert) => print(dessert));
```

And since the input of `print` is the same as the output of the `forEach` function, Dart allows you to rephrase that like so:

```
desserts.forEach(print);
```

This is known as a **tear-off** because you *tear off* the unnecessary syntax.

Run any of the loops above and you'll get the same result.

```
cookies
cupcakes
donuts
pie
```

Code as UI

The Flutter framework chose Dart because of its unique characteristics. However, Flutter has also influenced the development of Dart. One area you can see this is with the addition of the spread operator, **collection `if`** and **collection `for`**.

They make it easier for Flutter developers to compose user interface layouts completely in code, without the need for a separate markup language.

Flutter UI code is composed of classes called **widgets**. Three common Flutter widgets are rows, columns and stacks, which all store their children as `List` collections. Being able to manipulate lists using the spread operator, collection `if` and collection `for` makes it easier to build the UI with code.

The examples below use strings, but in a Flutter app you would see the same things with lists of `Text`, `Icon`, `ElevatedButton` and other `Widget` elements.

Spread operator

Suppose you have two lists to start with.

```
const pastries = ['cookies', 'croissants'];
const candy = ['Junior Mints', 'Twizzlers', 'M&M
s'];
```

You can use the **spread operator** (`...`) to expand those lists into another list.

```
const desserts = ['donuts', ...pastries, ...cand
y];
print(desserts);
```

By using the ... operator, the second element of `desserts` is not `pastries`, but instead, the elements of `pastries` are themselves the elements of `desserts`. The same goes for `candy`. Run that code and you'll see the following:

```
[donuts, cookies, croissants, Junior Mints, Twiz
zlers, M&Ms]
```

There's also a **null spread operator** (`...?`), which will omit a list if the list itself is `null`.

```
List<String>? coffees;
final hotDrinks = ['milk tea', ...?coffees];
```

Here `coffees` has not been initialized and therefore is `null`. By using the ...? operator, you avoid an error that would come by trying to add a null list. The list `hotDrinks` will only include `milk tea`.

Collection if

When creating a list, you can use a **collection `if`** to determine whether an element is included based on some condition.

So if you had a peanut allergy, you'd want to avoid adding certain candy with peanut butter to a list of candy.

```
const peanutAllergy = true;

const candy = [
  'Junior Mints',
  'Twizzlers',
  if (!peanutAllergy) 'Reeses',
];
print(candy);
```

Run that and you'll see that the `false` condition for the collection `if` prevented `Reeses` from being included in the list:

```
[Junior Mints, Twizzlers]
```

It's also interesting to note that collection `if` doesn't prevent a list from being a compile-time constant, as demonstrated by the presence of the `const` keyword.

Collection for

There's also a **collection `for`**. So if you have a list, you can use a collection `for` to iterate over the list and generate another list.

```
const deserts = ['gobi', 'sahara', 'arctic'];
var bigDeserts = [
  'ARABIAN',
  for (var desert in deserts) desert.toUpperCase
(),
];
print(bigDeserts);
```

Here you've created a new list where the final three elements are the uppercase version of the elements from the input list. The syntax is very much like a `for-in` loop but without the braces.

Run the code to see:

```
[ARABIAN, GOBI, SAHARA, ARCTIC]
```

Before moving on to learn about the `Set` collection type, test your knowledge so far with some mini-exercises.

Mini-exercises

1. Create an empty list of type `String`. Name it `months`. Use the `add` method to add the names of the twelve months.

2. Make an immutable list with the same elements as in Mini-exercise 1.

3. Use collection `for` to create a new list with the month names in all uppercase.

Sets

Sets are used to create a collection of unique elements. Sets in Dart are similar to their mathematical counterparts. Duplicates are not allowed in a set, in contrast to lists, which do allow duplicates.

You can also think of sets as a bag of elements with no particular ordering, unlike lists, which do maintain a specific order. Because order doesn't matter in a set, sets can be faster than lists, especially when dealing with large datasets.

Creating a set

You can create an empty set in Dart using the `Set` type annotation like so:

```
final Set<int> someSet = {};
```

The generic syntax with `int` in angle brackets tells Dart that only integers are allowed in the set. The following form is shorter but identical in result:

```
final someSet = <int>{};
```

The curly braces are the same symbols used for sets in mathematics, so that should help you remember them. Be sure to distinguish curly braces in this context from their use for defining scopes, though.

You can also use type inference with a **set literal** to let Dart determine the types of elements in the set.

```
final anotherSet = {1, 2, 3, 1};
print(anotherSet);
```

Since the set literal contains only integers, Dart is able to infer the type as `Set<int>`.

Additionally, you probably noticed that there are two 1s there. But because `anotherSet` is a set, it ends up with only one 1. Run that code to verify the contents of `anotherSet` has only one 1.

```
{1, 2, 3}
```

Operations on a set

In this section you'll see some general collection operations that also apply to sets.

Checking the contents

To see if a set contains an item, you use the `contains` method, which returns a `bool`.

Add the following two lines and run the code again:

```
print(anotherSet.contains(1)); // true
print(anotherSet.contains(99)); // false
```

Since anotherSet does contains 1, the method returns true, while checking for 99 returns false.

Adding single elements

Like growable lists, you can add and remove elements in a set. To add an element, use the add method.

```
final someSet = <int>{};
someSet.add(42);
someSet.add(2112);
someSet.add(42);
print(someSet);
```

Run that to see the following set:

```
{42, 2112}
```

You added 42 twice, but only one 42 shows up as expected.

Removing elements

You can also remove elements using the remove method.

```
someSet.remove(2112);
```

Print someSet to reveal only a single element is left:

```
{42}
```

Adding multiple elements

You can use `addAll` to add elements from a list into a set.

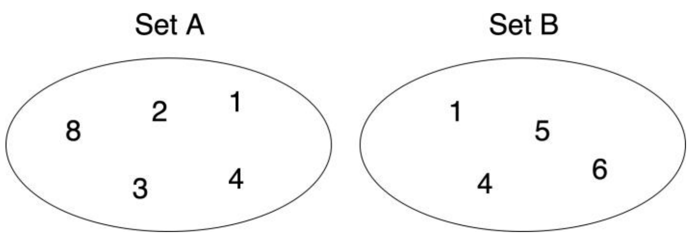
```
someSet.addAll([1, 2, 3, 4]);
```

Print `someSet` again to show the new contents:

```
{42, 1, 2, 3, 4}
```

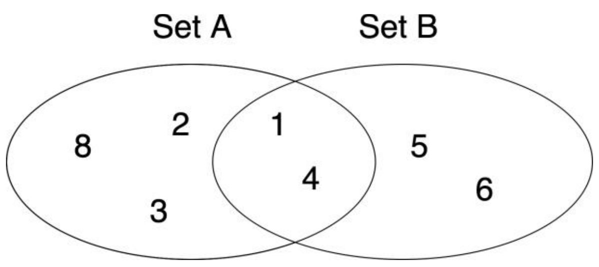
Intersections and Unions

You'll often have multiple sets of data and you'll want to know how they fit together.



You've probably seen your share of Venn diagrams; if not in school, then at least as memes on the internet. They're useful for showing the common elements between two sets.

As you can see, 1 and 4 are in both sets.



Intersections

Like Venn diagrams and mathematical sets, you can find the **intersection** of two sets in Dart; that is, the common elements that occur in both sets.

Given the following two sets:

```
final setA = {8, 2, 3, 1, 4};
final setB = {1, 6, 5, 4};
```

You can find the intersection like so:

```
final intersection = setA.intersection(setB);
```

Since both sets share the numbers 1 and 4, that's the answer you're expecting. Print `intersection` to see:

```
{1, 4}
```

No disappointments there.

Unions

Finding all the unique values by combining both sets gives you the **union**, and that's just as easy to find in Dart as the intersection.

```
final union = setA.union(setB);
```

Print `union` to see the results:

```
{8, 2, 3, 1, 4, 6, 5}
```

This union represents all the elements from both sets. Remember — sets have no requirement to be in order.

Other operations

Almost everything that you learned earlier about lists also applies to sets. Specifically, you can perform any of the following operations with sets:

- collection `if`
- collection `for`
- `for-in` loops
- `forEach` loops
- spread operators

Next you'll learn about another important collection type: maps.

Maps

Maps in Dart are the data structure used to hold key-value pairs. They're similar to HashMaps and Dictionaries in other languages.

If you're not familiar with maps, though, you can think of them like a collection of variables that contain data. The**key** is the variable name and the **value** is the data that the variable holds. The way to find a particular value is to give the map the name of the key that is *mapped* to that value.

In the image below, the cake is mapped to 500 calories, and the donut is mapped to 150 calories. cake and donut are keys, while 500 and 150 are values.

{ : 500 calories,
: 150 calories, ...}
Map<Food, Calories>
Key type
Value type

The key and value in each pair are separated by colons, and consecutive key-value pairs are separated by commas.

Creating an empty map

Like List and Set, Map is a generic type, but Map takes two type parameters: one for the key and one for the value. You can create an empty map variable using Map and specifying the type for both the key and value:

```
final Map<String, int> emptyMap = {};
```

In this example, `String` is the type for the key, and `int` is the type for the value.

A slightly shorter way to do the same thing is move the generic types to the right-hand side:

```
final emptyMap = <String, int>{};
```

Notice that maps also use curly braces just as sets do. What do you think you'd get if you wrote this?

```
final emptySomething = {};
```

Is `emptySomething` a set, or is it a map?

Well, it turns out that map literals came before set literals in Dart's history, so Dart infers the empty braces to be a `Map` of `<dynamic, dynamic>`; that is, the types of the key and value are both `dynamic`. If you want a set, and not a map, then you need to be explicit:

```
final mySet = <String>{};
```

Like lists, maps have a `length`, which tells you the number of key-value pairs stored in the map.

```
final emptyMap = <String, int>{};
print(emptyMap.length);
```

Since it's an empty map, when you print that, you'll get `0` for the length.

Initializing a Map with values

You can create a non-empty map variable using braces, where Dart infers the key and value types. Dart knows it's a map because each element is a pair separated by a colon.

```
final inventory = {
  'cakes': 20,
  'pies':  14,
  'donuts':  37,
  'cookies':  141,
};
```

In this case, `inventory` is a map of `String` to `int`, from bakery item to quantity in stock.

The key doesn't have to be a string. For example, here's a map of `int` to `String`, from a digit to its English spelling:

```
final digitToWord = {
  1: 'one',
  2: 'two',
  3: 'three',
  4: 'four',
};
```

Print both of those:

```
print(inventory);
print(digitToWord);
```

You'll see the output in horizontal format rather than the vertical format you had above:

```
{cakes: 20, pies: 14, donuts: 37, cookies: 141}
{1: one, 2: two, 3: three, 4: four}
```

Unique keys

The keys of a map should be unique. A map like the following wouldn't work:

```
final treasureMap = {
  'garbage': 'in the dumpster',
  'glasses': 'on your head',
  'gold': 'in the cave',
  'gold': 'under your mattress',
};
```

There are two keys named `gold`. How are you going to know where to look? You're probably thinking, "Hey, it's gold. I'll just look both places." If you really wanted to set it up like that, then you could map `String` to `List`:

```
final treasureMap = {
  'garbage': ['in the dumpster'],
  'glasses': ['on your head'],
  'gold': ['in the cave', 'under your mattress'
],
};
```

Now every key contains a list of items, but the keys themselves are unique.

Values don't have that same restriction of being unique. This is fine:

```
final myHouse = {
  'bedroom':  'messy',
  'kitchen': 'messy',
  'living room': 'messy',
  'code': 'clean',
};
```

Operations on a map

Interacting with a map to access, add, remove and update elements is very similar to what you've already seen.

Accessing elements from a map

You access individual elements from a map by using a subscript notation similar to lists, except for maps you use the key rather than an index.

```
final numberOfCakes = inventory['cakes'];
```

If you recall from above, the key cakes is mapped to the integer 20, so print `numberOfCakes` to see 20.

A map will return `null` if the key doesn't exist. Because of this, accessing an element from a map always gives a nullable value. In the example above, Dart infers `numberOfCakes` to be of type `int?`. If you want to use `numberOfCakes`, then you need to treat it as you would any other nullable value.

In this case you can use the null-aware access operator to check if the number of cakes is even:

```
print(numberOfCakes?.isEven);
```

There were 20 so that's true.

Adding elements to a map

You can add new elements to a map simply by assigning to elements that are not yet in the map.

```
inventory['brownies'] = 3;
```

Print `inventory` to see brownies and its value at the end of the map:

```
{cakes: 20, pies: 14, donuts: 37, cookies: 141,
brownies: 3}
```

Updating an element

Remember that the keys of a map are unique, so if you assign a value to a key that already exists, you'll overwrite the existing value.

```
inventory['cakes'] = 1;
```

Print `inventory` to confirm that cakes was 20 but now is 1:

```
{cakes: 1, pies: 14, donuts: 37, cookies: 141, b
rownies: 3}
```

Removing elements from a map

You can use `remove` to remove elements from a map by key.

```
inventory.remove('cookies');
```

COOKIE! Om nom nom nom nom.

```
{cakes: 1, pies: 14, donuts: 37, brownies: 3}
```

No more cookies.

Map properties

Maps have properties just as lists do. For example, the following properties indicate (using different metrics) whether or not the map is empty:

```
inventory.isEmpty     // false
inventory.isNotEmpty // true
inventory.length      // 4
```

You can also access the keys and values separately using the `keys` and `values` properties.

```
print(inventory.keys);
print(inventory.values);
```

When you print that out, you'll see the following:

```
(cakes, pies, donuts, brownies)
(1, 14, 37, 3)
```

Checking for key or value existence

To check whether a key is in a map, you can use the `containsKey` method:

```
print(inventory.containsKey('pies'));
// true
```

You can do the same for values using `containsValue`.

```
print(inventory.containsValue(42));
// false
```

Looping over elements of a map

Unlike lists, you can't iterate over a map using a `for-in` loop.

```
for (var item in inventory) {
  print(inventory[item]);
}
```

This will produce the following error:

```
The type 'Map<String, int>' used in the 'for' lo
op must implement Iterable.
```

`Iterable` is a type that knows how to move sequentially, or *iterate*, over its elements. `List` and `Set` both implement `Iterable`, but `Map` does not.

Note: In Chapter 9 you'll learn what "implement" means.

There is a solution, though, for looping over a Map. The `keys` and `values` properties of a map are iterables, so you can loop over them. Here's an example of iterating over the keys:

```
for (var item in inventory.keys) {
  print(inventory[item]);
}
```

You can also use `forEach` to iterate over the elements of a map, which gives you both the keys and the values.

```
inventory.forEach((key, value) => print('$key ->
$value'));
```

And this `for` loop does the same thing:

```
for (final entry in inventory.entries) {
  print('${entry.key} -> ${entry.value}');
}
```

Running either loop gives the following result:

```
cakes -> 1
pies -> 14
donuts -> 37
brownies -> 3
```

Before going on, test your knowledge of maps with the following mini-exercises.

Mini-exercises

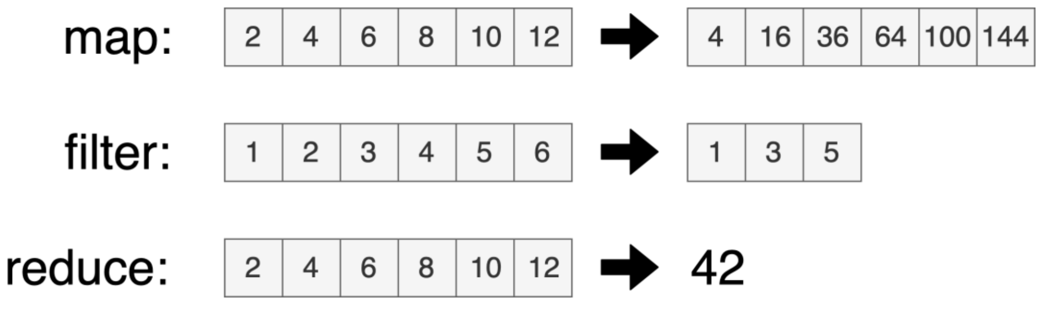
1. Create a map with the following keys: `name`, `profession`, `country` and `city`. For the values, add your own information.

2. You suddenly decide to move to Toronto, Canada. Programmatically update the values for `country` and `city`.

3. Iterate over the map and print all the values.

Higher order methods

There are a number of collection operations common to many programming languages, including transforming, filtering and consolidating the elements of the collection. These operations are known as **higher order methods,** because they take functions as parameters. This is a great opportunity to apply what you learned in Chapter 5 about anonymous functions.



In this section, you'll learn about the higher order methods named map, where, reduce, fold and sort. There are many more methods than this small sampling, though. Don't worry — you'll discover them in time.

Mapping over a collection

Mapping over a collection allows you to perform an action on each element of the collection as if you were running it through a loop. To do this, collections have a map method that takes an anonymous function as a parameter, and returns another collection based on what the function does to the elements.

Note : The map method in this section is different than the Map data type that you studied earlier in this chapter. List, Set and Map all have a map method.

Write the following code:

```
const numbers = [1, 2, 3, 4];
final squares = numbers.map((number) => number *
 number);
```

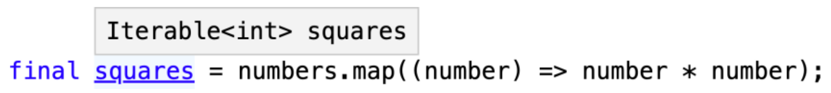
The map method is very similar to forEach in that it loops through every element of the list. It takes each element and passes it in as an argument to the anonymous function. In the example above, since numbers is a list of int values, number is inferred to be of type int. The first time through the loop, number is 1, the second time through, number is 2, and so on through 4.

Inside the anonymous function body, you're allowed to do whatever you want. In the case above, you square each input value. Next is where map and forEach differ. Instead of just performing some action, map takes each resulting value and inserts it as an element in a new collection, in this case one named squares.

Print squares to see the result:

```
(1, 4, 9, 16)
```

The numbers make sense. Those are all squares of the input list. Look at the parentheses, though. Set and Map use curly braces and List uses square brackets. Hover your cursor over squares to see the type.



```
Iterable<int> squares
final squares = numbers.map((number) => number * number);
```

It's actually an Iterable of int, rather than a List of int. That's why when you print squares it has parentheses rather than square brackets. If you really want a List instead of an Iterable, you can call the toList method on the result.

```
print(squares.toList());
```

Run that and now you'll have square brackets:

```
[1, 4, 9, 16]
```

It's a common mistake to forget that map produces an `Iterable` rather than a `List`, but now you know what to do. There's a similar method called `toSet` if you need a set instead of a list.

Filtering a collection

You can filter an iterable collection like `List` and `Set` down to another shorter collection by using the `where` method.

Add the following line below the code you already have:

```
final evens = squares.where((square) => square.is
Even);
```

Like map, the `where` method takes an anonymous function. The function's input is also each element of the list, but unlike map, the value the function returns must be a Boolean. If the function returns `true` for a particular element, then that element is added to the resulting collection, but if `false`, then the element is excluded. Using `isEven` makes the condition `true` for even numbers, so you've filtered down squares to just the even values.

Print evens and you'll get:

```
(4, 16)
```

These are indeed even squares. As you can see by the parentheses, `where` also returns an `Iterable`.

You can use where with List and Set but not with Map, that is, unless you access the keys or values properties of Map.

Consolidating a collection

Some higher order methods take all the elements of an iterable collection and consolidate them into a single value using the function you provide. You'll learn two ways to do this.

Using reduce

One way to combine all of the elements of a list into a single value is to use the reduce method. You can combine the elements in any way you like, but the example below shows how to find their sum.

Given the following list of amounts, find the total value by passing in an anonymous function that adds each element to the sum of the previous ones:

```
const amounts = [199, 299, 299, 199, 499];
final total = amounts.reduce((sum, element) => su
m + element);
```

The first function parameter always contains the result of the previous function call, while the second parameter contains the current element in the collection. In this example, on each iteration sum stores the current total while element is the current integer in the list.

Print total to see the final result of 1495.

Using fold

If you try to call reduce on an empty list, you'll get an error. For that reason, using fold may be more reliable when a collection has a possibility of containing zero elements. The fold method works like reduce, but it takes an extra parameter that provides the function with a starting value.

Here is the same result as above, but this time using fold:

```
const amounts = [199, 299, 299, 199, 499];
final total = amounts.fold(
  0,
  (int sum, element) => sum + element,
);
```

Notice that there are two arguments that you gave the fold method. The first argument 0 is the starting value. The second argument takes that 0, feeds it to sum, and keeps adding to it based on the value of each element in the list.

Sorting a list

While where, reduce and fold all work equally well on lists or sets, you can only call sort on a list. That's because sets are by definition unordered, so it wouldn't make sense to sort them.

Calling sort on a list sorts the elements based on their data type.

```
final desserts = ['cookies', 'pie', 'donuts', 'br
ownies'];
desserts.sort();
```

Print desserts and you'll see the following:

```
[brownies, cookies, donuts, pie]
```

Since desserts holds strings, calling sort on the list arranges them in alphabetical order. The sorting is done in place, which means sort mutates the input list itself. This also means if you tried to sort a const list, you'd get an error.

Reversing a list

You can use reversed to produce a list in reverse order.

```
var dessertsReversed = desserts.reversed;
```

This produces the following result:

```
(pie, donuts, cookies, brownies)
```

Mind you, using reversed doesn't re-sort the list in reverse order. It just returns an Iterable that starts at the last element of the list and works forward. This will become clear if you look at an unsorted list:

```
final desserts = ['cookies', 'pie', 'donuts', 'br
ownies'];
final dessertsReversed = desserts.reversed;
print(desserts);
print(dessertsReversed);
```

Run that to see the results:

```
[cookies, pie, donuts, brownies]
(brownies, donuts, pie, cookies)
```

Neither collection is sorted, but the second one is in reverse order of the first.

This also brings up a couple of important points about naming conventions in Dart:

- You should use a commanding verb for a method that produces a side effect. The **sort()** method mutates itself, which is a side effect. Also notice the parentheses on the **sort()** method; they clearly say that this is a method, not a property, and as such, may be doing some potentially expensive work.

- In comparison, reversed is a getter property, which you recognize because it doesn't have any parentheses. This indicates that the work is lighter, usually because getters only return a value. Additionally, reversed is an adjective, not a commanding verb. That's because there are no side effects as it doesn't mutate the collection.

You won't *always* see these conventions followed, but they're general guidelines that are helpful to consider in your own naming.

> **Note**: One interesting characteristic of iterables is that they're lazy. That means they don't do any work until you ask them to. Since reversed returns an iterable, it doesn't actually reverse the elements of the collection until you try to access those elements, such as by printing the collection or converting it to a list using the toList method. Understanding this can help you put off work that doesn't need to be done yet.

Performing a custom sort

For the sort method, you can pass in a function as an argument to perform custom sorting. Say you want to sort strings by length and not alphabetically; you could give sort an anonymous function like so:

```
desserts.sort((d1, d2) => d1.length.compareTo(d
2.length));
```

The names d1 and d2 aren't going to win any good naming prizes, but they fit on the page of a book better than dessertOne and dessertTwo do. The compareTo method returns -1 if the first length is shorter, 1 if it's longer, and 0 if both lengths are the same. This is all sort needs to do the custom sort.

So now desserts is sorted by the size of each string element.

```
[pie, donuts, cookies, brownies]
```

Combining higher order methods

You can chain together the higher order methods that you learned above. For example, if you wanted to take only the desserts that have a name length greater than 5 and then convert those names to uppercase, you would do it like so:

```
const desserts = ['cake', 'pie', 'donuts', 'brow
nies'];
final bigTallDesserts = desserts
    .where((dessert) => dessert.length > 5)
    .map((dessert) => dessert.toUpperCase());
```

Wrapping that expression onto multiple lines makes it easier to read. First you filtered the list with where and then mapped the resulting iterable to get the final result.

Printing bigTallDesserts reveals:

```
(DONUTS, BROWNIES)
```

Mini-exercises

Given the following exam scores:

```
final scores = [89, 77, 46, 93, 82, 67, 32, 88];
```

1. Use sort to find the highest and lowest grades.

2. Use `where` to find all the B grades, that is, all the scores between 80 and 90.

When to use lists, sets or maps

Congratulations on making it through another chapter! You’ve made a ton of progress. This chapter will leave you with some advice about when to use which type of collection. Each type has its strengths.

- Choose **lists** if order matters. Try to insert at the end of lists wherever possible to keep things running smoothly. And be aware that searching can be slow with big collections.

- Choose **sets** if you are only concerned with whether something is in the collection or not. This is faster than searching a list.

- Choose **maps** if you frequently need to search for a value by a key. Searching by key is also fast.

Challenges

Before moving on, here are some challenges to test your knowledge of collections. 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: A unique request

Write a function that takes a paragraph of text and returns a collection of unique `String` characters that the text contains.

Challenge 2: Counting on you

Repeat Challenge 1, but this time have the function return a collection that contains the frequency, or count, of every unique character.

Challenge 3: Mapping users

Create a class called `User` with properties for `id` and `name`. Make a `List` with three users, specifying any appropriate names and IDs you like. Then write a function that converts your user list to a list of maps whose keys are `id` and `name`.

Key points

- Lists store an ordered collection of elements.
- Sets store an unordered collection of unique elements.
- Maps store a collection of key-value pairs.
- The elements of a collection are mutable by default.
- The spread operator (`...`) allows you to expand one collection inside another collection.
- Collection `if` and `for` can be used to dynamically create the content of a list or set.

- You can iterate over any collection, but for a map you need to iterate over the keys or values if you use a `for-in` loop.

- Higher order methods take a function as a parameter and act on the elements of a collection.

- The `map` method, not to be confused with the `Map` type, performs an operation on each element of a collection and returns the results as an `Iterable`.

- The `where` method filters an iterable collection based on a condition.

- The `reduce` and `fold` methods consolidate a collection down to a single value.

- The `sort` method sorts a list in place according to its data type.