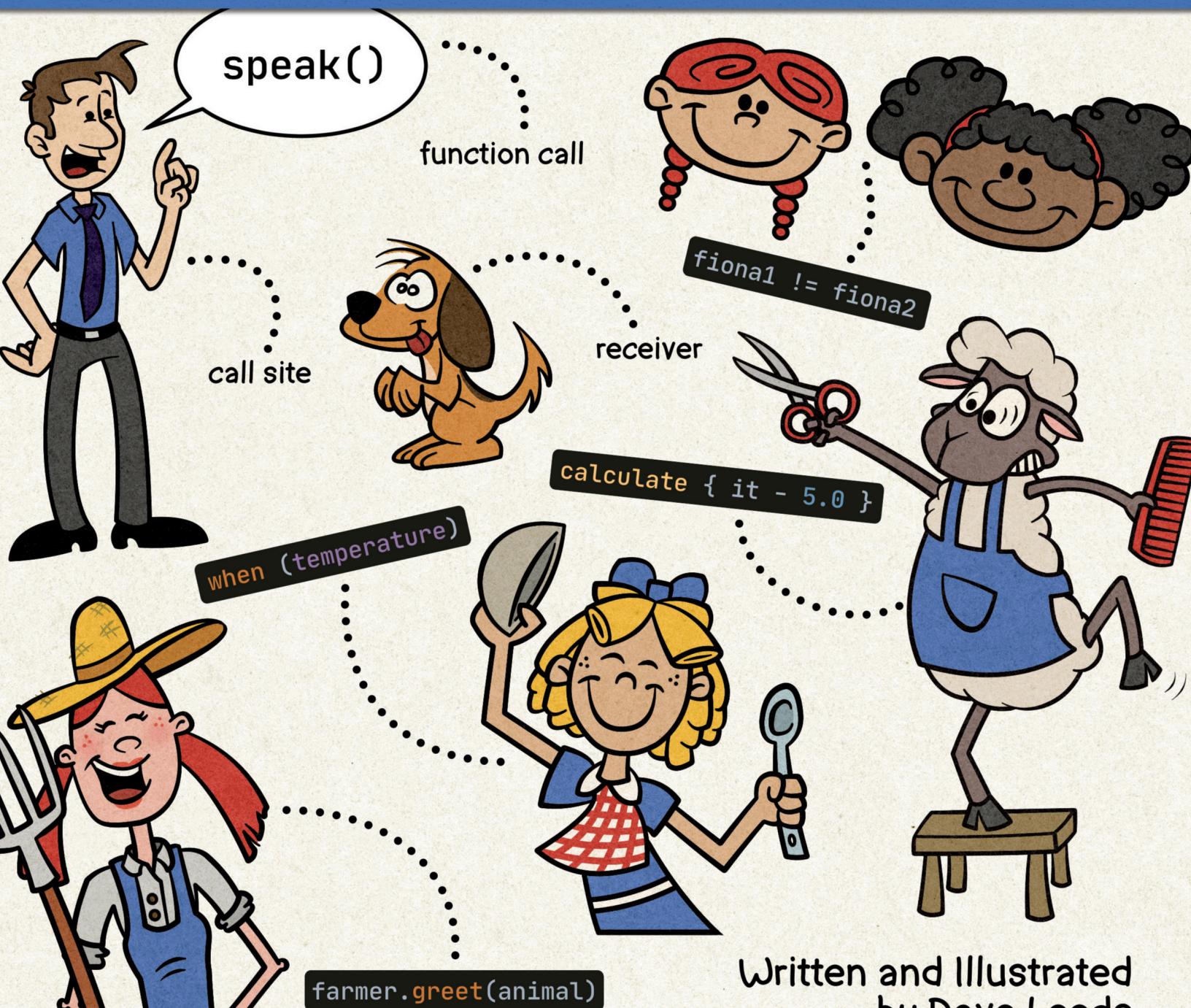


Kotlin:

An Illustrated Guide

The fun way to learn Kotlin programming, one concept at a time!



Written and illustrated
by Dave Leeds

Kotlin:

An Illustrated Guide

Written and Illustrated by

Dave Leeds

For more about Kotlin from Dave Leeds, visit <https://typealias.com>.

Kotlin: An Illustrated Guide

Copyright ©2020-2024 by Dave Leeds

The author has aimed for excellence when writing, illustrating, and preparing this book, but makes no expressed or implied warranty of any kind. No representation is made as to the accuracy, completeness, currentness, suitability, or validity of its information. The author does not assume and hereby disclaims any liability to any party for any errors, omissions, losses, injuries, or damages in connection with or arising out of the use of the information or code herein. All content of this book is provided on an as-is basis.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the author is aware of a trademark claim, the designations have been printed in initial caps or all caps.

Android™, Android robot logo, YouTube® and YouTube logo are trademarks of Google LLC.

GitHub® is a registered trademark of GitHub, Inc.

JetBrains® and IntelliJ IDEA® are registered trademarks of JetBrains s.r.o.

Oracle, Java®, JavaScript®, JVM® and JDK® are registered trademarks of Oracle and/or its affiliates.

UML® is a registered trademark of Object Management Group, Inc. in the United States and/or other countries.

The author is not affiliated with JetBrains®, the sponsors and developers of the Kotlin programming language.

Table of Contents

Who is this book for?	vii
What's the current state of this book?	vii
Variables, Expressions, and Types	
Introduction to Variables	12
Expressions and Statements	16
Types: Different Kinds of Values	18
Other Types	20
Summary	20
Functions	
Introduction to Functions	24
Removing Duplication with Functions.....	25
Function Basics.....	26
Functions with More Than One Parameter	29
Named Arguments	31
Default Arguments	31
Expression Bodies and Block Bodies	34
Functions without a Result	36
Summary	37
Conditionals: When and If	
Goldilocks and the Three Branches	40
Introduction to when in Kotlin.....	41
if Expressions	47
Summary	50
Introduction to Classes and Objects	
Putting Variables and Functions Together	53
Defining a Class.....	53
Objects	55

Everything is an Object	64
-------------------------------	----

Summary	66
---------------	----

Enum Classes

Creating an Enum Class	70
------------------------------	----

Using an Enum Class	70
---------------------------	----

Using Enum Classes with when Expressions	71
--	----

Adding Properties and Functions to Enum Classes	72
---	----

Built-In Properties	74
---------------------------	----

Summary	76
---------------	----

Nulls and Null Safety

Introduction to Nulls	79
-----------------------------	----

Compile Time and Runtime	82
--------------------------------	----

How Nullable and Non-Nullable Types are Related	84
---	----

Using Conditionals to Check for null	87
--	----

Using the Elvis Operator to Provide a Default Value	89
---	----

Using the Not-Null Assertion Operator to Insist that a Value is Present	91
---	----

Using the Safe-Call Operator to Invoke Functions and Properties	93
---	----

Summary	97
---------------	----

Lambdas and Function References

Bert's Snips & Clips	100
----------------------------	-----

Introduction to Function Types	102
--------------------------------------	-----

Introduction to Lambdas	108
-------------------------------	-----

The Implicit it parameter	110
---------------------------------	-----

Lambdas and Higher-Order Functions	111
--	-----

Lambdas with Multiple Statements	113
--	-----

Closures	114
----------------	-----

Summary	116
---------------	-----

Collections: Lists and Sets

Introduction to Lists	119
-----------------------------	-----

Adding and Removing an Element	122
--------------------------------------	-----

Loops and Iterations	127
----------------------------	-----

Collection Operations	129
-----------------------------	-----

Introduction to Sets	136
----------------------------	-----

Summary	138
---------------	-----

Collections: Maps

The Right Tool for the Job	141
Associating Data	141
Map Fundamentals.....	143
Creating a Map from a List	152
Summary	160

Receivers and Extensions

Standalone Functions and Object Functions.....	163
They're Not So Different After All	165
Introduction to Receivers	166
Introduction to Extension Functions	169
Summary	173

Scopes and Scope Functions

Introduction to Scopes.....	176
Scopes and Visibility.....	178
Introduction to Scope Functions.....	187
Choosing the Most Appropriate Scope Function.....	193
Shadowing Names.....	194
Scope Functions and Null Checks	197
Summary	198

Introduction to Interfaces

Sue Starts a Farm	202
Introducing Interfaces.....	207
Subtypes and Supertypes	209
Casting	212
Multiple Interfaces	214
Interface Inheritance.....	216
Default Implementations	218

Introduction to Class Delegation

Delegation in Restaurants and Code	223
Manual Delegation	223
Easy Delegation, the Kotlin Way	228
Multiple Delegates.....	229
Overriding a Delegated Call	231

Managing Conflicts.....	232
Delegation for General and Specific Types	234
Summary	236
Abstract and Open Classes	
Modeling a Car	240
Introduction to Abstract Classes.....	242
Extending Abstract Classes.....	243
Inheritance.....	245
Overriding Members.....	247
Introduction to Open Classes	251
Getter and Setter Visibility Modifiers	251
Combining Interfaces and Abstract/Open Classes	253
Comparing Interfaces, Abstract Classes, and Open Classes	254
Subclasses and Substitution	254
Class Hierarchies.....	255
The Any Type.....	255
Summary	256
Data Classes and Destructuring	
Overriding equals()	259
Overriding hashCode().....	263
Overriding toString().....	265
Introduction to Data Classes	267
Copying Data Classes.....	268
Destructuring	270
Limitations of Data Classes	276
Summary	278
Sealed Types	
Adding Another Type.....	283
Introduction to Sealed Types	285
Sealed Classes.....	287
Why Is the sealed Modifier Required At All?.....	288
Restrictions of a Sealed Type's Subtype	290
Sealed Types vs Enum Classes	291
Summary	294

Handling Runtime Exceptions

Problems at Runtime.....	297
Catching Exceptions.....	304
Throwing Exceptions	309
Exception Types	310
Handling Multiple Exception Types Differently.....	313
Evaluating a Try Expression	315
Try-Catch-Finally.....	316
A Functional Approach to Exception Handling.....	318
Summary	319

Generics

Mugs and Beverages.....	323
Declared Types, Actual Types, and Assignment Compatibility	326
Introduction to Generic Types.....	328
Type Parameter Constraints	330
Generics in Practice	332
Generics in the Standard Library	335
Trade-Offs of Generics.....	336
Summary	339

Generic Variance (DRAFT)

Covariance	342
Contravariance	348
What Makes a Subtype a Subtype?.....	353
Variance Modifiers	354
Variance on Multiple Type Parameters	358
Type Projections	359
Variance in the Standard Library	365
Summary	365

Preface

Thank you for purchasing this book! You've worked hard for your money, so I want this book to be everything that you hope it is.

Who is this book for?

If you're interested in learning Kotlin development, this book is for you. Everyone has a unique experience - some readers are new to software development entirely, some are new to Kotlin development, and some are existing Kotlin developers who want a deeper understanding of certain language concepts. My approach in this book is to explain Kotlin programming concepts with simple, tangible, and illustrated examples, so that regardless of your experience level, things will make sense. I do my best to define terms clearly before using them, so that as you read the book, you'll build upon a solid foundation of existing knowledge rather than an uncertain, tenuous foundation of ambiguity.

Even if you already know some of the concepts presented in this book, my goal is to give those concepts such a fun and vivid presentation that you'll *enjoy* reading about them again, which I hope will give you a definite clarity about them, shining light on the foggy corners of concepts that you might have wondered about.

What's the current state of this book?

This book is in active development. Currently, you'll find the same main content as the online version of the book, plus you've got access to draft chapters as they're written.

- Chapters 1-18 are complete. You're welcome to [provide feedback](#) about them.
- Chapter 19 is available in draft form for you to preview.
- I'm expecting to have a total of about 20 chapters.

My main focus at the moment is to *complete the remaining chapters*. After that, I'll plan to return to earlier chapters in order to improve the quality of the images (improving resolution for PDF quality).

Laying out this content in book format - that is, with a definite page size and hard page breaks - is quite different from laying out content on a web page. On the one hand, it's easier because multiple screen sizes and resolutions don't need to be accounted for. On the other hand, hard page breaks add a challenge of splitting the content at inconvenient points, such as in the middle of an illustration or code listing. In this version of the book, I've laid out the content to keep entire code listings on one page as much as possible, but you'll find some extra white space as a result. Once the main content has been finalized, I intend to optimize the layout to make everything flow well and to minimize blank space.

Acknowledgments

Like all software developers, I have a tendency to underestimate how long a project can take. I expected to finish this book within a year or two while still working a full-time job. Here I am, about four years in, still working on it, and I've had lots of help and support along the way.

First, I'm very thankful to my wife Elizabeth for all the evening and weekend hours that she gave up so that I could write this book. She's been a massive support and encouragement to me at every step along the way. She's always cheering me on!

Thanks to Beverly Jamison, who was my mentor for the first decade of my software development career. Another decade has passed since working together, but I learned so much from her, and I wouldn't be the developer I am today without her guidance and leadership.

Thanks to these friends I've worked with, who have been especially encouraging as I've been writing about Kotlin - Jared Andrews, Jon Beebe, Jeremy Butler, Matthew Ensor, Aaron Fleckenstein, James Lorenzen, Dustin Majtan, Joe McSorley, Av Pinzur, Jacob Rakidzich, and everyone else.

Thanks so much to the following reviewers:

- James Lorenzen
- David Blanc
- Louis CAD
- Matt McKenna
- Mohit Sarveiya
- Tobenna Ezike
- Esraa Ibrahim
- Charles Muchene
- Raheel Naz
- Jacob Rakidzich
- Doug Smith
- Jayant Varma
- gbagd24 (...never did get your real name...!)

I'm very grateful to both [Kotlin Weekly](#) and [Android Weekly](#) for all the times that they've included my articles, chapters, and videos in their weekly newsletters.

Thanks to the Kotlin developer advocates at JetBrains - especially Hadi Hariri, who years ago suggested that I consider creating a Leanpub edition of this book!

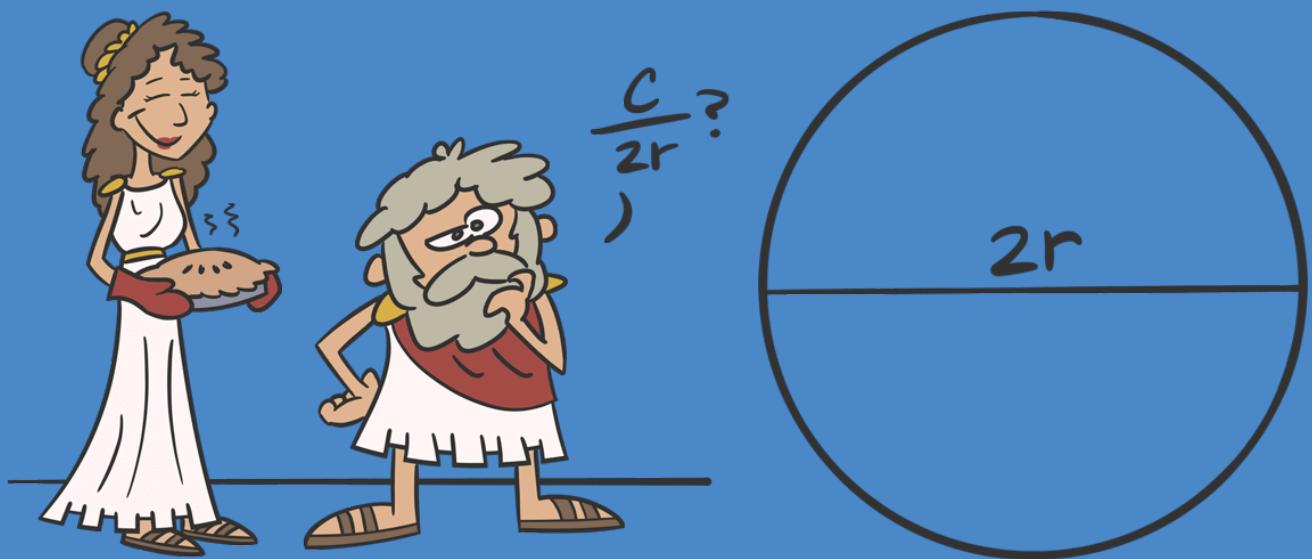
Thanks to the entire Kotlin community for your kind words of encouragement over the years. You've got no idea how much that's meant to me!

And finally, thanks to my cute little schnauzer, Ninja, who often keeps me company while I'm writing. Yep, I just acknowledged my dog!

Kotlin: An Illustrated Guide

Chapter 1

Variables, Expressions, and Types



So you want to be a Kotlin developer?

You've come to the right place!

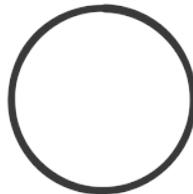
This book will take you through the fundamentals of Kotlin, gently introducing you to the core concepts of the language, in order to help you become a proficient Kotlin developer. Even if you're a seasoned pro, it's important to know the fundamentals in order to establish a solid foundation of understanding, so that you can be as effective as possible.

Your adventure starts here in Chapter 1, where we'll cover the basics of variables, expressions, and types.

Let's get to it!

Introduction to Variables

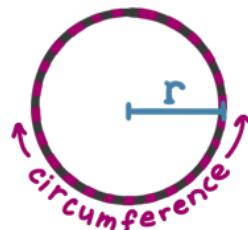
This is a circle.



Thousands of years ago, a Greek fellow named Archimedes roughly figured out how to measure the length of the outside of a circle, called the *circumference*. You probably remember the equation the good old days of junior high school...

$$\text{Circumference} = 2\pi r$$

"The circumference of a circle is equal to 2 times π times the radius of the circle".



In the equation above, the letter r isn't a number itself. It's a letter that *represents the radius* - a number that measures the distance between the center of the circle and the edge of the circle.

We call r a variable, because the radius can vary, depending on the size of the circle. In other words, the variable r isn't a number itself; it's just sort of a "bucket" that holds a number - any number.

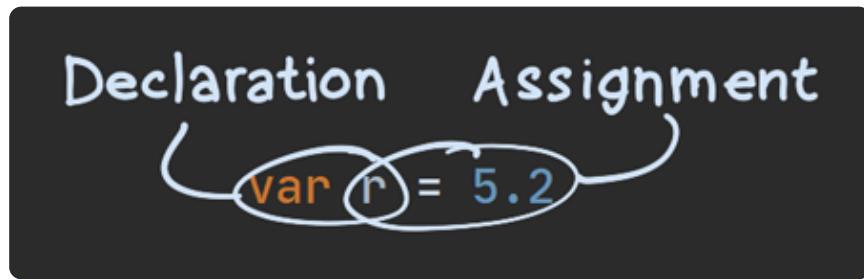


Variables aren't just for algebra and geometry. They're also used all the time in programming, where they serve the same purpose: holding values. In Kotlin, you can create a variable and put a number into it like this:

```
var r = 5.2
```

Listing 1.1 - Creating a variable in Kotlin.

This code is actually doing two things on one line.



1. **Declaration** - When we write `var r`, we're declaring a new variable called `r`. You can think of declaring a variable as creating a bucket.
2. **Assignment** - When we write `r = 5.2`, we're assigning the value of 5.2 to the variable `r`. In other words, we're putting the number 5.2 into the bucket.



Let's break down the three main parts of this line of code:

In this code...

- `var` is a **keyword** that tells Kotlin to create a new variable.
- `r` is the *name* of the variable. You might also hear this referred to as the variable's **identifier**.
- The number `5.2` is the **value** that is being assigned.

`var` is just one of many keywords that we'll see as we continue to learn Kotlin. The important thing to remember about keywords is that they can't be used as the name of your own things. For example, you can't use `var` as the name of the variable:

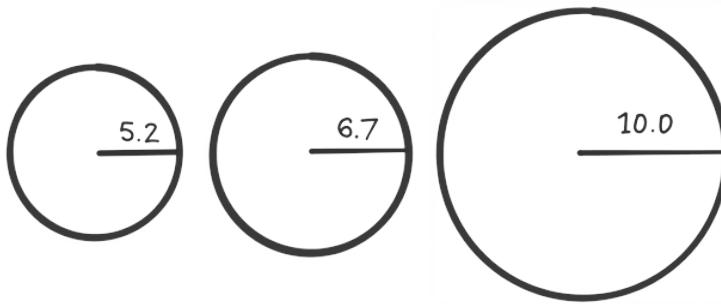
```
var var = 5.2
```

Listing 1.2 - Error: "Expecting property name or receiver type" (does not compile)

We'll come across other keywords in future chapters.

Reassigning Variables

As we noted above, in the equation $2\pi r$, the variable `r` can represent different numbers in different situations.



In Kotlin, if you decide that you want `r` to represent a different number, you can just reassign it, as shown in the second line here:

```
var r = 5.2
r = 6.7
```

Listing 1.3 - Assigning and then reassigning a value to a variable

When reassigning this variable, we didn't need to use the `var` keyword again, because the variable was already declared on that first line.

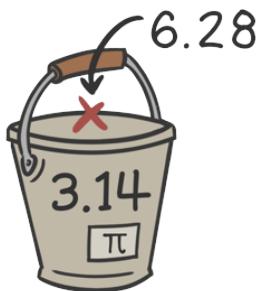
Variables that Cannot Be Reassigned

Let's look at that circumference equation again:

$$\text{Circumference} = 2\pi r$$

The Greek letter π (pronounced like the word "pie") is very different from the variable `r`. Whereas `r` can hold any number in it, π only ever holds one very specific number, which we'll abbreviate to 3.14.

In the same way, when programming in Kotlin, there are times when you want to make sure that a variable *never* changes.



In those cases, instead of declaring it with `var`, you declare it with `val`, like this:

```
val pi = 3.14
```

Listing 1.4 - Creating a read-only variable in Kotlin

Now, if you try to *reassign* π , you'll get an error:

```
val π = 3.14
π = 1.23
```

Listing 1.5 - Error: "Val cannot be reassigned" (does not compile).

Once you put something in this bucket, you can never replace it!

Declaring your variables with `val` is a great way to make sure that you don't accidentally change something that shouldn't change. In fact, it's a great idea to always *start* with `val`, and only switch to `var` when you absolutely need to.

For the Nerds

If you're already familiar with the concept of objects, it's important to note that `val` does not make objects immutable. In other words, you can't put something new in that bucket, but what's in that bucket could still change itself in some way. That's why Kotlin programmers tend to call variables declared with `val` "read-only" instead of "immutable".

We'll cover this distinction in more detail once we get to the chapter on classes and objects.

Naming Variables

It's been fun using the letter π in our code, but unless you live in Greece, you probably don't have it on your keyboard. From here on out, we'll make life easier for everyone by naming it `pi` instead. Also, instead of `r`, we'll name it `radius`, so that any other developers who come along later will know exactly what that variable represents - we don't want others to have to guess what the letter `r` stands for!

Sometimes you need more than one word for a variable's identifier. In Kotlin, it's customary to start the first word in lowercase, and then capitalize the remaining words, like this:

```
var radiusOfTheCircle = 5.2
```

Listing 1.6 - Variable naming convention in Kotlin: camel case.

Best Practice

It's a good idea to write your code in a way that's similar to other developers, so that when everyone looks at it, they don't have to think too hard about it.

The official Kotlin documentation provides a set of [coding conventions](#), and it's recommended that you follow those. In case you use IntelliJ or Android Studio, that document also tells you how you can configure a formatter that will help you more easily conform to some of those conventions.

Now that you've got down the basics for declaring and assigning variables, we can start assigning more than just *simple numbers* - we can start assigning more complex calculations, such as our circumference equation! Let's dive into *expressions*!

Expressions and Statements

Let's look at that equation again:

$$\text{Circumference} = 2\pi r$$

We've already created a variable for pi and a variable for radius, so now we just need Kotlin to do some math for us, and we can get the circumference of *any* circle, regardless of how big that circle is.

All we have to do is multiply together 2, pi, and radius. In Kotlin, as in most programming languages, multiplication isn't represented with an x, it's represented with an asterisk: *, so our code can look like this:

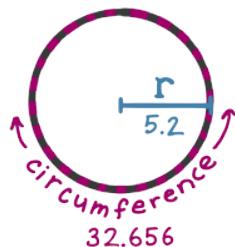
```
var radius = 5.2
val pi = 3.14

val circumference = 2 * pi * radius
```

Listing 1.7 - Assigning a complex expression to a variable.

So far we've only assigned simple values - such as 5.2 and 3.14. This is the first time we're assigning something more complex: $2 * \pi * \text{radius}$.

When Kotlin sees this, it simply calculates the result for you - it multiplies 2 times pi times radius, and then, of course, it takes the resulting value and assigns it to the variable named circumference. In this case (with a radius of 5.2), circumference will equal 32.656.



Since $2 * \pi * \text{radius}$ can be calculated into a value like this, we say that it can be **evaluated**. Code that can be *evaluated* is called an **expression**. Here are a few examples of expressions:

- $2 + 3$
- $2 * \pi * r$
- $\pi * r * r$

Variables by themselves are also expressions - they evaluate to whatever value they hold:

- radius
- pi

When you type out a number by hand (as opposed to typing a variable), it's called a number **literal**. Literals themselves are also expressions - they evaluate to themselves! Here are a few examples:

- 2
- 5.2
- 3.14

All of those examples will evaluate to some value. On the other hand, when you have a chunk of code that does not evaluate to a value, it's called a **statement**.

Here's an easy rule of thumb to know if you've got an expression or a statement:

Rule: If you can assign a chunk of code to a variable, it's an expression. Otherwise, it's a statement.

Let's apply this rule to the first expression from each of the three lists above ($2 + 3$, radius, and 2). The parts highlighted below are the expressions:

```
val a = 2 + 3
val b = radius
val c = 2
```

Listing 1.8 - You can tell each of these is an expression because each can be assigned to a variable.

Since each of those can be assigned to a variable, they're all expressions.

The only *statement* we've seen so far is the *assignment statement*, such as `val pi = 3.14`. Applying our rule of thumb to an assignment statement looks confusing, of course, because we're using an assignment on an assignment (yikes!):

```
val a = val pi = 3.14
```

Listing 1.9 - Error: "Expecting an expression" (does not compile)

If you try to do this, Kotlin gives you a helpful error message, "*Expecting an expression*". If you ever see this error message, it just means you tried to use a statement where Kotlin wanted an expression.

The distinction between *statements* and *expressions* is important as you're learning Kotlin, and we'll use those terms often in this series.

So far, whether we've used *literals* or *complex expressions*, we've still only ever assigned *numbers* to variables. But there are lots of different things that variables can hold! Let's explore some of these things next.

Types: Different Kinds of Values

In Kotlin, different variables can hold different *kinds* of values. The *kind* of value that a variable holds is called its **type**.

Let's take another look at the variables radius and pi:

```
var radius = 5.2
val pi = 3.14
```

Listing 1.10 - Our radius and pi variables from Listing 1.8.

In this case, both radius and pi represent *numbers that have a decimal point*, which is a type that Kotlin calls a Double. If you want to, you can tell Kotlin the type of the variable yourself, like this:

```
var radius: Double = 5.2
val pi: Double = 3.14
```

Listing 1.11 - Explicitly specifying the type of these two variables.

When we do this, we are *explicitly specifying* the type.

Very often, you do not have to specify the type of a variable yourself. In that case, Kotlin will do its best to **infer** the type based on whatever it is that you're assigning to the variable. That process is called **type inference**.

So, when we write this...

```
var radius = 5.2
```

Listing 1.12 -Allowing Kotlin to apply type inference. The radius variable still has a type of Double.

...then Kotlin can tell that 5.2 is a Double, so it *automatically* uses Double as the type of radius.

Quick Tip

It's usually nice to let Kotlin do type inference for you, but sometimes it'd still be nice to see what type it inferred.

To see what type Kotlin inferred, IntelliJ and Android Studio have a feature called [Quick Documentation](#). Just put the text caret on the variable name, and hit the keyboard shortcut. By default, that shortcut is F1 on Mac, and Ctrl+Q on Windows and Linux.

In addition to Double, there are some other basic types that are good to know about! Let's take a look at some of those now.

Integers

So far we've only used numbers that have a decimal point in them, such as 5.2 and 3.14. But you might also use a number that does not have a decimal point, such as just 5 or 3.

These kinds of numbers are called integers, and in Kotlin the type for an integer is just called `Int` for short. Here's an example of creating an integer variable:

```
val numberOfDayDogs: Int = 3
```

Listing 1.13 - Declaring and assigning an integer variable. Here we explicitly specify the `Int` type, but that's not required.

Booleans

Sometimes you want a variable to hold a value that is either on or off, yes or no, true or false, and so on.

In those cases, you want a Boolean variable.

```
val earthIsRound: Boolean = true  
val earthIsFlat: Boolean = false
```

Listing 1.14 - Declaring and assigning Boolean variables. Again, we explicitly specify the `Boolean` type, but that's not required.

Why is it called a Boolean?

It's named after a British chap from the 1800s, George Boole, who created a branch of algebra that works with true and false values instead of numbers.

Awesome.

Strings

You can also store text into a variable. The fancy programmer word for this is string, because it's a bunch of characters - such as letters, numbers, and symbols - "strung" together:



In Kotlin, the type for this is called `String`, and you can create a `String` variable like this:

```
val text: String = "This is a string"
```

Listing 1.15 - Declaring and assigning a simple `String` variable, explicitly specifying the type.

Other Types

These are just some common types of variables. As we continue this series, we'll find out how we can create our own types - classes - which build upon these basic types that we just looked at.

Types and Reassignment

In Kotlin, the type of each variable is established when you're *writing* the code, and its type will *never* change (unless you rewrite the code). This is why we call it **static typing**. Once a variable has been declared with a particular type, no other type of value can go into it. For example, if we create a variable of type `Int`, we can't later *reassign* it with a `String`:

```
var number_of_dogs: Int = 5
number_of_dogs = "five"
```

Listing 1.16 - Error: "Type mismatch. Required: Int. Found: String." (does not compile)

In future chapters, we'll see how this static typing plays out in more ways than just reassigning variables. We'll also see how some types can have *subtypes*.

But we're getting ahead of ourselves - for now, let's wrap up this chapter!

Summary

In this chapter, we looked at:

- **Variables**

- Using `var` for [variables that can be reassigned](#).
- Using `val` for [variables that are read-only](#).

- **Expressions and Statements**

- [Expressions](#) can be evaluated to a value.
- [Statements](#) do not evaluate to a value.
- You can try assigning a chunk of code to a variable [to see if it's a statement or an expression](#).

- **Basic Types**

- Number types like [Double](#) and [Int](#).
- The [Boolean](#) type for true and false values.

- The [String](#) type for text values.

Now that we have a good grasp on variables, it's time to put them together in exciting new ways! Stay tuned for chapter 2, where we'll learn all about *functions* in Kotlin!

Kotlin: An Illustrated Guide

Chapter 2

Functions



Don't Duplicate Those Expressions!

In the last chapter, we wrote some Kotlin code to calculate the circumference of a particular circle.

In this chapter, we're going to write some *functions* that will make it easy to calculate the circumference of *any* circle!

Introduction to Functions

As we saw in the last chapter, calculating the circumference of a circle is easy:

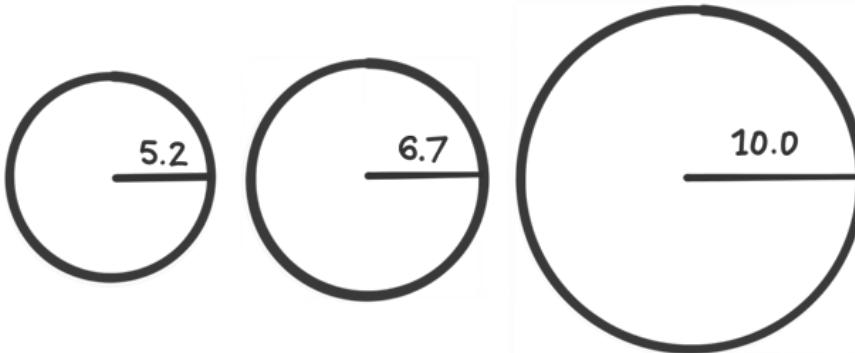
$$\text{Circumference} = 2\pi r$$

And here's some Kotlin code that we wrote to do that calculation:

```
val pi = 3.14
var radius = 5.2
val circumference = 2 * pi * radius
```

Listing 2.1 - Calculating the circumference of a circle that has a radius of 5.2

That code calculates the circumference of a circle that has a radius of 5.2. But of course, not all circles have a radius of 5.2! What happens if we also want to determine the circumference of a circle that has a radius of 6.7? Or 10.0?



Well, we could just write out the equation multiple times.

```
val pi = 3.14

var radius = 5.2
val circumferenceOfSmallCircle = 2 * pi * radius

radius = 6.7
val circumferenceOfMediumCircle = 2 * pi * radius

radius = 10.0
val circumferenceOfLargeCircle = 2 * pi * radius
```

Listing 2.2 - Calculating the circumference of a circle that has a radius of 5.2, and another with radius 6.7, and another with radius 10.0.

This certainly works, but wow - look at how we had to type the same thing over and over again!

```

val pi = 3.14
var radius = 5.2

val circumferenceOfSmallCircle = 2 * pi * radius ←
radius = 6.7
val circumferenceOfMediumCircle = 2 * pi * radius ← Same thing
radius = 10.0
val circumferenceOfLargeCircle = 2 * pi * radius ←

```

When we have the same code over and over again like this, we call it **duplication**. In most cases, *duplicated* code is bad because:

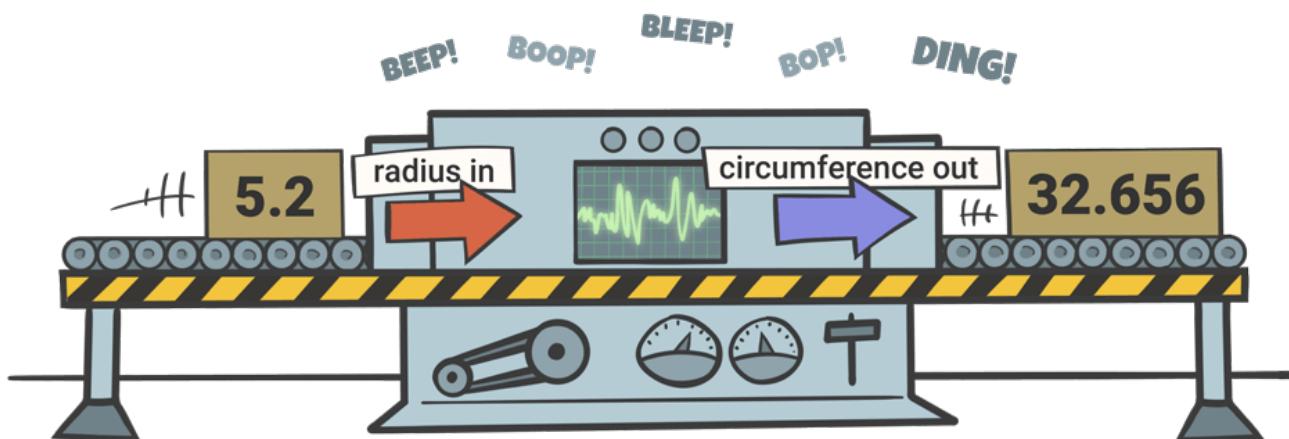
1. When you type the same thing so many times, it becomes more likely that you might type it wrong in one of those cases. For example, one time, you might accidentally type `3 * pi radius`.
2. If you want to change the equation, you have to find all of the places where you typed it, and make sure you update each one of them.
3. It can be more difficult to read when your eyes see the same thing written over and over again on the screen.

Let's change our code so that we only have to write `2 * pi * radius` one time, and then use that to calculate the circumference of any circle. In other words, let's remove the duplication!

Removing Duplication with Functions

Even though we wrote `2 * pi * radius` three times, the only value that actually changed each time was `radius`. In other words, `2` never changed, and `pi` never changed (it equaled `3.14` each time). But the value of `radius` was different each time: first `5.2`, then `6.7`, and then `10.0`.

Since the `radius` is the only thing that changes each time, it would be awesome if we could just convert a `radius` into a `circumference`. In other words, what if we could build a *machine* where we insert a `radius` on one side, and a `circumference` pops out of the other side?



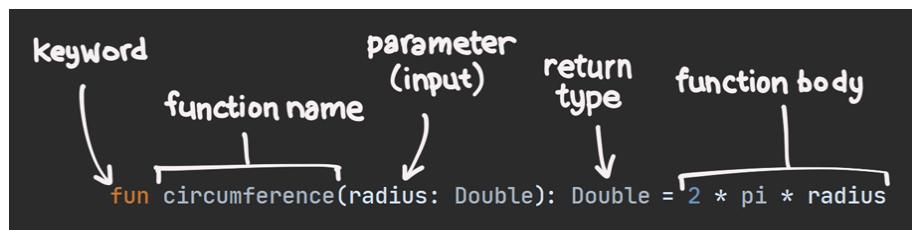
- Since we're *putting in* a radius, we might call that the **input**.
- And since a circumference comes *out* of the other side, we might call that the **output**.

Now, we won't be creating a *real* machine, but instead, we will create a **function**, which will do exactly what we want - we'll give it a *radius* and get a *circumference* back from it!

Function Basics

Creating a Function

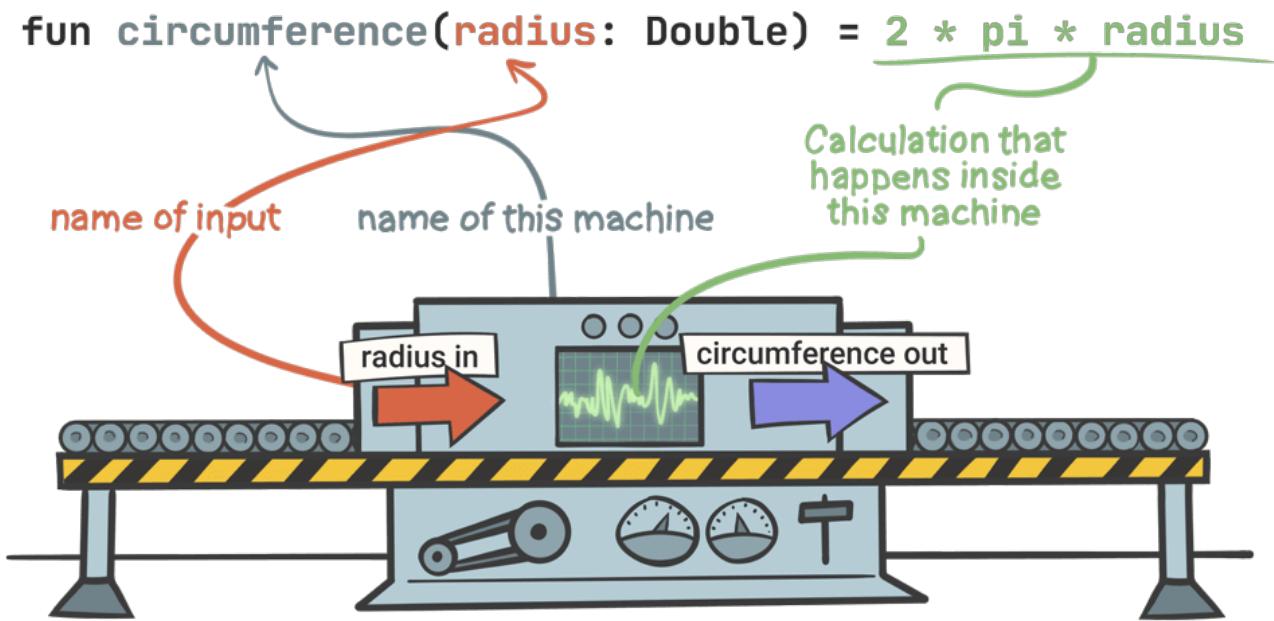
Here's how you can write a simple function in Kotlin:



This looks like a lot, but there are really just a few pieces, and they're all easy to understand.

1. First, `fun` is a **keyword** (just like `val` and `var` are keywords). It tells Kotlin that you are writing a **function**.
2. Next, `circumference` is the **name** of our function. We can name the function just about anything we want, but here, I chose `circumference`.
3. Then, `(radius: Double)` says that this function has an *input* called `radius`, which has a **type** called `Double`. We call `radius` a **parameter** of this function.
4. Then, the `: Double` after the closing parenthesis indicates that the output of the function will be of type `Double`. This is called the **return type** of the function.
5. And finally, `2 * pi * radius` is called the **body** of the function. Whatever this expression evaluates to will be the *output* of the function. The *value* of that output is referred to as the **result** of the function. It's the thing that comes out of the machine. Note that whatever this expression evaluates to *must* be the same type that's specified by the return type. In this example, `2 * pi * radius` must evaluate to a `Double` or we'll get an error.

In the image below, we can compare our function with the machine we imagined above.



You might recall from the last chapter that you often don't have to specify the type of a variable, and instead let Kotlin use its [type inference](#). You can also use type inference when writing functions like this. Just leave off the return type, so that it looks like this:

```
fun circumference(radius: Double) = 2 * pi * radius
```

Listing 2.3 - Using Kotlin's type inference on a function by leaving off the return type.

Kotlin programmers will often use type inference for simple functions like this one.

Quick Tip

If you're using IntelliJ or Android Studio, there's a setting that will let you see what the return type is, even when you're using type inference.

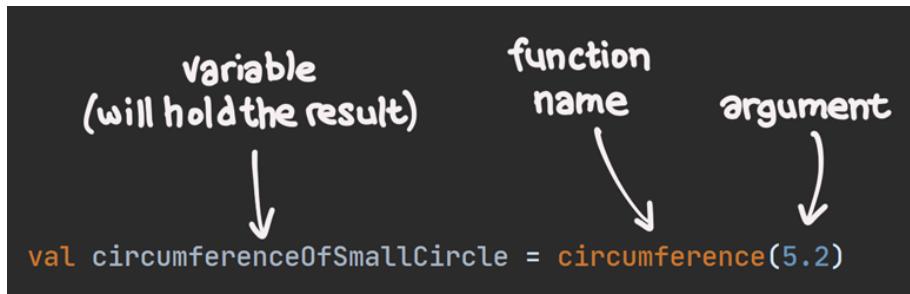
To turn it on, open *Settings*, and search for "Inlay Hints", and enable them for types. If you don't want to always see the hints, you can map a keyboard shortcut to *Toggle Inlay Hints* under the *Keymap* section of the settings.

And now that we've created our function, it's time to use it!

Calling a Function

When you use the function - that is, when you put something into the machine - it's referred to as **calling** or **invoking** the function. The place where it is called is referred to as the calling code or the call site.

Here's how you call a function in Kotlin:



1. First, `circumferenceOfSmallCircle` is a variable that will hold the *result* of the function call (that is, whatever comes out of the machine).
2. Next, `circumference` is the *name* of the function that we're calling.
3. The value `5.2` is the **argument** of this function - it's the thing that we're putting into the machine. When we call a function with an argument, we sometimes say that we are **passing** the argument to the function.

Note

From this point forward, I'll usually include the parentheses after the name of a function. For example, I'll refer to it as `circumference()` rather than just `circumference`. This is done to help make it clear that I'm referring to a function.

What happens when you call a function? Let's say we wrote a function and called it, like this:

```
fun circumference(radius: Double) = 2 * pi * radius  
val circumferenceOfSmallCircle = circumference(5.2)
```

Listing 2.4 - A function definition and a function call.

When we call that function, it's kind of like we're just *plopping* the body of the function - `2 * pi * radius` - right where we see the function call - `circumference(5.2)`.

So you can imagine it like this:

```
fun circumference(radius: Double) = 2 * pi * radius  
val circumferenceOfSmallCircle = 2 * pi * radius
```

Listing 2.5 - When you call a function, it's almost like you're substituting the function call with the body of the function.

Then, since we passed `5.2` as the radius, we could imagine substituting the value `5.2` where we had plopped in `radius`:

```
fun circumference(radius: Double) = 2 * pi * radius

val circumferenceOfSmallCircle = 2 * pi * 5.2
```

Listing 2.6 - When calling a function, imagine that you're substituting the parameter name with the argument value that was passed in.

In summary, when we call `circumference(5.2)`, it's as if we had written `2 * pi * 5.2` at the same spot.

Now that we've got a function that can calculate the circumference from a radius, we can call that function as many times as we need!

```
val pi = 3.14
fun circumference(radius: Double) = 2 * pi * radius

val circumferenceOfSmallCircle = circumference(5.2)
val circumferenceOfMediumCircle = circumference(6.7)
val circumferenceOfLargeCircle = circumference(10.0)
```

Listing 2.7 - Calling the `circumference()` function three times in a row.

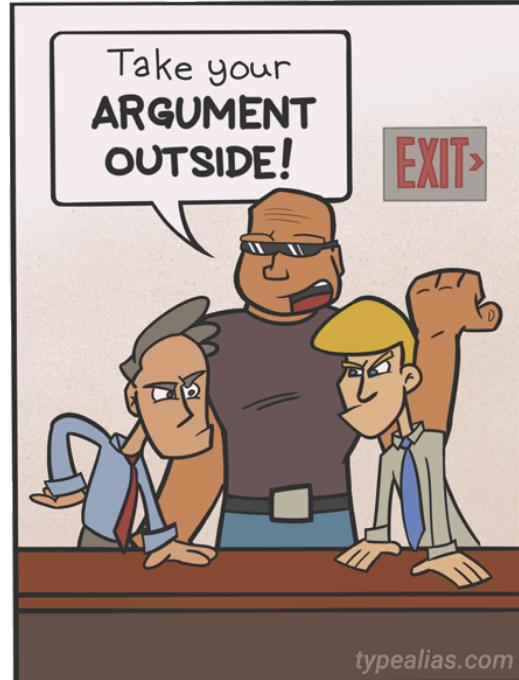
Doesn't this look nicer than Listing 2.2? Because we have a function, we don't need to write `2 * pi * radius` over and over! Instead, we just call the `circumference()` function once for each circle.

Arguments and Parameters: What's the difference?

It's easy to confuse an argument with a parameter, so it's important to understand the distinction:

1. An **argument** is a *value* that you pass to the function. For example, you might pass `5.2` into the `circumference()` function. The argument is `5.2`.
2. A **parameter** is a variable inside the function that will *hold* that value. For example, when you pass `5.2` to `circumference()`, it is assigned to the parameter called `radius`.

An argument is a value that usually comes from *outside* of the function, whereas a parameter is declared *inside* the function. As a mnemonic, simply associate the word "argument" with the word "outside", as in the cartoon to the right.



Functions with More Than One Parameter

The `circumference()` function has just *one* parameter, called `radius`, but there are times when you might need a function that has *more* than that. Let's create a function that has *two* parameters!

Even if physics class was a while ago, you probably know how to calculate *speed*. It's easy to remember, because we say it aloud all the time - "The speed limit is 100 kilometers per hour".

"Kilometers per hour" is just distance ("kilometers") divided by ("per") time ("hour").

$$\text{speed} = \frac{\text{distance}}{\text{time}}$$

In Kotlin, you use a *forward slash* to represent division. You can think of it as a fraction that fell over to the left:

$$\text{speed} = \frac{\text{distance}}{\text{time}}$$

$$\text{speed} = \cancel{\text{distance}} / \text{time}$$

$$\text{speed} = \text{distance} / \text{time}$$

To start with, let's just write some simple code to calculate the average speed of a car that has traveled 321.8 kilometers in 4.15 hours.

```
val distance = 321.8
val time = 4.15
val speed = distance / time
```

Listing 2.8 - Calculating speed without using a function.

Now, let's turn that `distance / time` expression into a function. In order to create a function for speed, we need to know two things: `distance` and `time`.

In Kotlin, when you need a function with two parameters, you can just separate those parameters with a comma, like this:

```
fun speed(distance: Double, time: Double) = distance / time
```

Listing 2.9 - Defining a function that has two parameters.

When you need to call this function, the *arguments* are also separated with a comma. For example, we can call the `speed()` function with the same values as we used above, separating those values with a comma:

```
val averageSpeed = speed(321.8, 4.15)
```

Listing 2.10 - Calling a function that has two parameters.

The result is about 77.54 kilometers per hour.

Note that the arguments here are in the *same order* as the parameters.

- Since `321.8` is the first argument, `321.8` will be assigned to the first parameter, which is `distance`.
- Since `4.15` is the second argument, `4.15` will be assigned to the second parameter, which is `time`.

In other words, the *position* of the argument matters when we call a function this way, which is why we sometimes refer to arguments like these as **positional arguments**.

```
fun speed(distance: Double, time: Double) = distance / time
val averageSpeed = speed(321.8, 4.15)
```

But this isn't the *only* way to pass arguments to a function!

Named Arguments

Rather than relying on the position of the arguments, you can instead use the name of the parameter, like this:

```
val averageSpeed = speed(distance = 321.8, time = 4.15)
```

Listing 2.11 - Calling a function using named arguments instead of positional arguments.

These are called **named arguments**. The neat thing about named arguments is that the order doesn't matter. So, you can call the function like this, with the arguments in a different order:

```
val averageSpeed = speed(time = 4.15, distance = 321.8)
```

Listing 2.12 - Using named arguments to reverse the order of two arguments.

In other words, all five of these function calls will end up with the exact same result:

```
val averageSpeed1 = speed(321.8, 4.15)
val averageSpeed2 = speed(distance = 321.8, 4.15)
val averageSpeed3 = speed(321.8, time = 4.15)
val averageSpeed4 = speed(distance = 321.8, time = 4.15)
val averageSpeed5 = speed(time = 4.15, distance = 321.8)
```

Listing 2.13 - Multiple ways to call a function with the same two arguments.

Default Arguments

In some cases, you might find yourself passing the same argument value to a function over and over again. For example, maybe you're calculating how fast:

1. A person is walking

2. Another person is biking
3. A third person is driving a car, and
4. A fourth person is flying a plane.



Everybody was moving for 2.0 hours, except for the plane, which got to its final destination in only 1.5 hours. Using our `speed()` function from above, you can calculate the speeds like this:

```
val walkingSpeed = speed(10.2, 2.0)
val bikingSpeed = speed(29.6, 2.0)
val drivingSpeed = speed(225.3, 2.0)
val flyingSpeed = speed(1368.747, 1.5)
```

Listing 2.14 - Calling a function with the same argument value. The time is 2.0 for the first three function calls.

Instead of having to pass `2.0` for the `time` parameter over and over, you could choose to make `2.0` a **default argument** when we define the function.

Let's update our `speed()` function so that the `time` parameter defaults to `2.0`.

```
fun speed(distance: Double, time: Double = 2.0) = distance / time
```

Listing 2.15 - A function with a default second argument.

Now, we can omit the argument for `time` whenever it should equal `2.0`, like this:

```
val walkingSpeed = speed(10.2)
val bikingSpeed = speed(29.6)
val drivingSpeed = speed(225.3)
val flyingSpeed = speed(1368.747, 1.5)
```

Listing 2.16 - Calling a function with a default argument.

For walking, biking, and driving, we left off the `time` argument, so those defaulted to `2.0`. But for flying, we passed `1.5`. The results for Listing 2.16 are exactly the same as those for Listing 2.14.

Easy!

But what happens when you want a default argument for the *first* parameter instead of the *second* parameter?

When a Default Argument Comes First

Our walker, biker, driver, and pilot are at it again. But this time, it's a race! Whoever gets to the finish line 42.195 kilometers away gets the prize.

Everyone finished the race, except for the airplane, which got a flat tire before it could get off the ground:

```
val walkingSpeed = speed(42.195, 8.27)
val bikingSpeed = speed(42.195, 2.85)
val drivingSpeed = speed(42.195, 0.37)
val flyingSpeed = speed(0.12, 0.01)
```

Listing 2.17 - Calling a function with the same argument value. The distance is 42.195 for the first three function calls.

Since the distance is 42.195 for everyone except the airplane, let's update our code - we'll remove the default for `time`, and instead give it a default `distance` of 42.195:

```
fun speed(distance: Double = 42.195, time: Double) =
    distance / time
```

Listing 2.18 - A function with a default first argument.

Now, since the walker, the biker, and the driver all travel the same distance, we should be able to omit the value for the first parameter, `distance`. It might be tempting to call the function like this:

```
val walkingSpeed = speed(8.27)
val bikingSpeed = speed(2.85)
val drivingSpeed = speed(0.37)
val flyingSpeed = speed(0.12, 0.01)
```

Listing 2.19 - Error: "No value passed for parameter 'time'" (does not compile)

But this causes an error: *No value passed for parameter 'time'*. Why did that happen?

Since we're using *positional arguments*, we actually ended up omitting `time` instead of `distance`.

In other words, we wanted to assign 8.27 to the `time` parameter, like this:

```
fun speed(distance: Double = 42.195, time: Double) = distance / time
                                                     ↑
val walkingSpeed = speed(8.27)
```

But we actually assigned 8.27 to the `distance` parameter, because 8.27 is the first argument, and `distance` is the first parameter:

```
fun speed(distance: Double = 42.195, time: Double) = distance / time  
val walkingSpeed = speed(8.27)
```

To tell Kotlin that we're sending the time instead, we simply use named arguments, like this:

```
val walkingSpeed = speed(time = 8.27)  
val bikingSpeed = speed(time = 2.85)  
val drivingSpeed = speed(time = 0.37)  
val flyingSpeed = speed(0.12, 0.01)
```

Listing 2.20 - Accepting the default argument for the first parameter of a function.

Now, the first parameter, `distance`, will default to **42.195** when we call `speed()` for walking, biking, and driving.

Expression Bodies and Block Bodies

So far we've been writing functions that just evaluate an [expression](#).

- Our `circumference()` function just evaluates `2 * pi * radius`
- Our `speed()` function just evaluates `distance / time`

Let's look at our code for `circumference()` again:

```
val pi = 3.14  
  
fun circumference(radius: Double) = 2 * pi * radius
```

Listing 2.21 - Our `circumference()` function, which has an expression body.

When we write a function this way, where the body is just a *single expression*, we say that the function has an **expression body**.

Kotlin also gives us a second way that we can write functions. Let's rewrite the `circumference()` function using this second way:

```
val pi = 3.14  
  
fun circumference(radius: Double): Double {  
    return 2 * pi * radius  
}
```

Listing 2.22 - Our `circumference()` function, rewritten to have a block body.

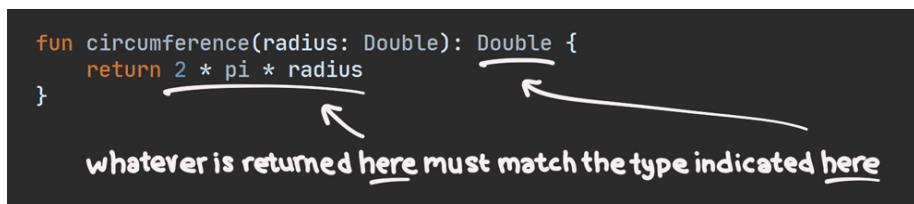
When we write a function like this, we call it a function with a **block body**.

The way you write a function with a *block body* is a little more complex than the way we've written *expression body* functions so far, so let's take a closer look at the new pieces:

First, notice the `: Double` after the parentheses. We were able to use type inference for expression bodies, but when we use a block body, we *must* specify the return type explicitly.

Next, notice the opening and closing braces: `{` and `}`. Everything between those two braces is referred to as a **code block** (which is why we call this a function with a *block body*!)

Finally, notice the word `return` inside that code block. The word `return` is a keyword that tells Kotlin that the expression that follows it is what the function should return. As with expression body functions, the value that the function returns must match the type that we said the function returns!



Functions with a block body take more typing than functions with an expression body, so why would we ever want to use them?

- They let you write *more than one line of code in the function*.
- They let you write statements inside them, not just expressions.

For example, so far we have defined `pi` *outside* of our function. But it'd be great to define it *inside* the function instead:

```
fun circumference(radius: Double): Double {
    val pi = 3.14
    return 2 * pi * radius
}
```

Listing 2.23 - A block body with multiple lines.

By moving `pi` to the inside of the function like this, it will only be accessible from inside the function. In other words, if you try to use it outside of the function, you'll get an error.

```
fun circumference(radius: Double): Double {
    val pi = 3.14
    return 2 * pi * radius
}

val tau = 2 * pi
```

Listing 2.24 - Error: "Unresolved reference: pi" (does not compile)

Throughout the rest of this book, we'll see many examples of both functions with expression bodies and functions with block bodies.

Best Practice

- When you've got a function that just evaluates a simple expression, it's conventional in Kotlin to use an *expression body* instead of a *block body*.
- Even though functions with block bodies can have as many lines of code as you want, functions are usually easier to understand when you keep them short.

Functions without a Result

There are times when you might want a function that doesn't return a result. For example, let's say we have a variable that holds a number that increases over time, named `counter`. We'll create a function named `increment()`. Every time we call that function, it'll increase `counter` by one, by writing the *statement* `counter = counter + 1`.

Functions with *expression bodies* can only contain a single expression. We cannot use a statement in an expression body. For example, we can't do this:

```
var counter = 0

fun increment() = counter = counter + 1
```

Listing 2.25 - Error: "Assignments are not expressions, and only expressions are allowed in this context"

Instead of using an expression body, it's better to use a block body for this function instead:

```
var counter = 0

fun increment() {
    counter = counter + 1
}
```

Listing 2.26 - A function with a block body that does not specify a return type.

You probably noticed that we didn't specify a return type on this function.

```
fun increment() {
    counter = counter + 1
}
```

You might be surprised to learn that, even though we didn't specify a return type, and even though this function contains no expression, this function *still returns a value!*

That's right! When you omit a return type for a function that has a block body, it *automatically* returns a special Kotlin type called **Unit**.

Unit - it's not a number or string, and you can't really do much with it. But it ends up being helpful in some cases that we'll explore when we get to generics in a future chapter.

But for now, let's wrap up the current chapter!

Summary

In this chapter, we learned:

- What a [function](#) is, and how it can help us remove duplication from our code.
- What a function [parameter](#) is, and how it [differs](#) from an [argument](#) of a function call.
- The difference between [positional](#) and [named](#) arguments.
- How to give your arguments [default values](#).
- The difference between [expression bodies](#) and [block bodies](#).
- How Kotlin will use the **Unit** type if we have [no meaningful result](#) to return from a function.

Next up in chapter 3, we'll learn all about conditionals in Kotlin, so that we can get our code to do different things in different situations.

Kotlin: An Illustrated Guide

Chapter 3

Conditionals: When and If



Different Circumstances, Different Behavior

In real life, we do different things depending on the circumstances.

For example, if it's raining outside, I'll probably grab an umbrella. If it's sunny outside, I'll grab my sunglasses. I do different things depending on the weather.

Similarly, we often need our code to do different things in different situations, so in this chapter, we'll look at Kotlin's *conditionals*.

Goldilocks and the Three Branches

You might have heard the fairy tale of *Goldilocks and the Three Bears*.

Papa Bear, Mama Bear, and Baby Bear lived in a quaint house nestled in the woods. One day, after preparing some porridge for breakfast, the three bears decided to go out for a leisurely stroll.

Meanwhile, a young girl named Goldilocks peeked in through the window and saw three bowls of porridge. Since she was hungry and saw nobody home, she decided to walk on inside and taste the porridge for herself.

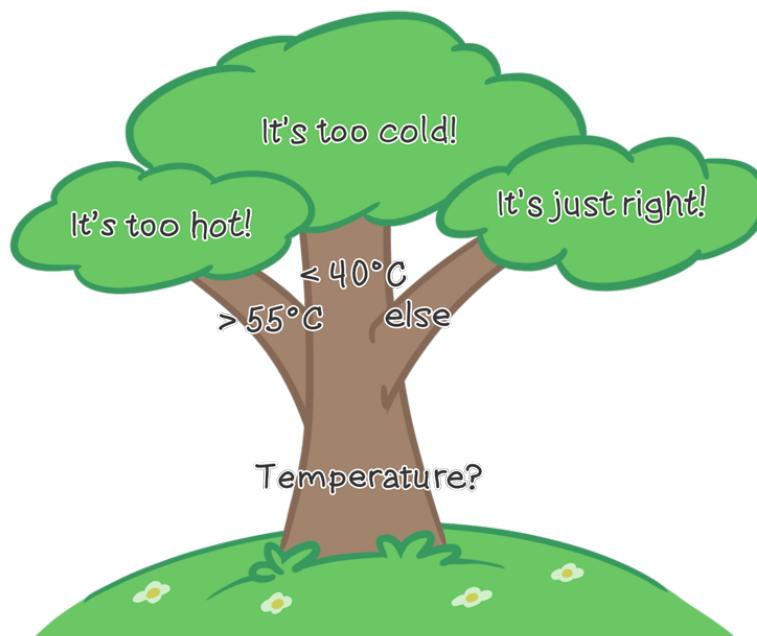
- First, she tried Papa Bear's porridge, and exclaimed, "It's too hot!"
- Then she tried Mama Bear's porridge, and complained, "It's too cold!"
- Finally, she tried Baby Bear's porridge, and found the temperature to be perfect. "It's just right!" she said, and ate the entire bowl.

Let's map out Goldilocks' reaction to the three different bowls of porridge into a simple table. Goldilocks responds differently depending on the temperature of the porridge.

1. When it's above 55°C, she says, "It's too hot!"
2. When it's below 40°C, she complains, "It's too cold!"
3. But anything in between, and she'll say, "It's just right!"

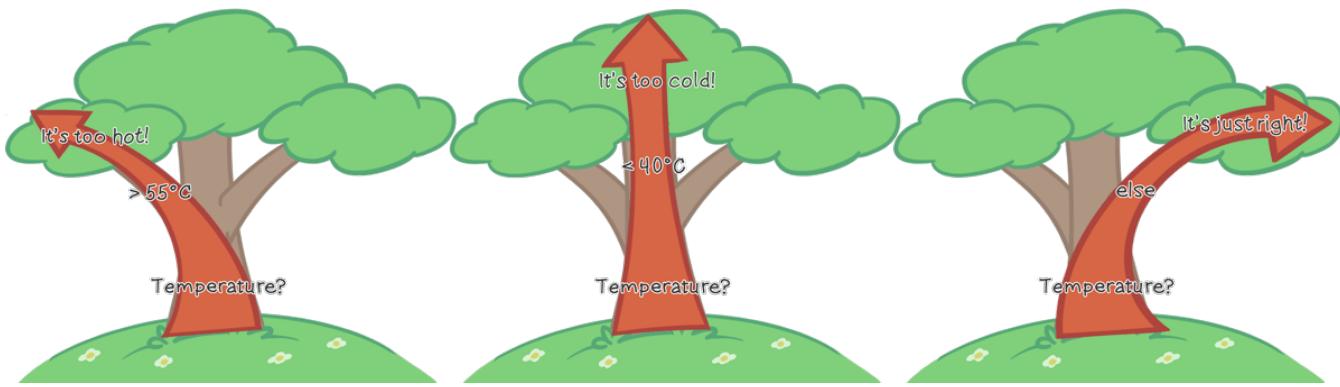
Temperature	Goldilocks' Reaction
Greater than 55°C	"It's too hot!"
Less than 40°C	"It's too cold!"
Anything else	"It's just right!"

We can write Kotlin code that will tell us which of these three things Goldilocks will say, based on the temperature of the porridge. In programming terms, the three different cases above - that is, the three different rows of the table - are called **branches**. If it helps, you can visualize the branches like you'd see on a tree:



So far, when we've run the code that we've written, *every line of code* has run. But now, we're going to start writing code instead where only one of the branches will run each time. While your code is running, the *path* that is run - including the particular branch that it takes - is referred to as the **execution path**.

Here you can see the three possible *execution paths* that our Kotlin code could take, depending on the temperature:



Code constructs that choose between different branches (e.g., different responses from Goldilocks) based on some *condition* (e.g., the temperature) are called **conditionals**.

So how can we write Kotlin code that tells us how Goldilocks will respond to each bowl of porridge? Kotlin has a couple of *conditionals* that we can use. Let's check 'em out!

Introduction to when in Kotlin

Let's look at that table again.

A table like this is easy to read and understand. We can just scan down the left-hand column, and when we find the temperature of the porridge that Goldilocks is eating, we look to the right to see how she will respond.

Temperature	Goldilocks' Reaction
Greater than 55°C	"It's too hot!"
Less than 40°C	"It's too cold!"
Anything else	"It's just right!"

Kotlin gives us a powerful conditional called `when`, which enables us to basically write that same table in our code. Here's how it looks.

```
val temperature = 48

val reaction = when {
    temperature > 55 -> "It's too hot!"
    temperature < 40 -> "It's too cold!"
    else             -> "It's just right!"
}
```

Listing 3.1 - A when expression in Kotlin

Just to show how similar this is to our table, let's add a little spacing and put the table next to it.

```
val temperature = 48

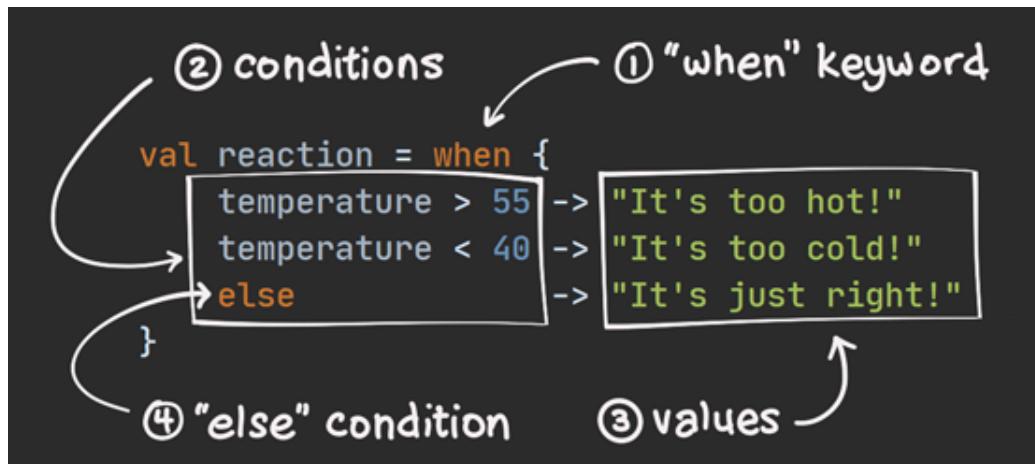
val reaction = when {
    temperature > 55 -> "It's too hot!"
    temperature < 40 -> "It's too cold!"
    else -> "It's just right!"
}
```

Temperature	Goldilocks' Reaction
Greater than 55°C	"It's too hot!"
Less than 40°C	"It's too cold!"
Anything else	"It's just right!"

As you can see, this looks almost identical to our table above! Each line between the opening brace { and the closing brace } is just a row from our table. The `reaction` variable will be assigned a *different* value depending on the value of the `temperature` variable:

- When the `temperature` is greater than 55, then `reaction` will be assigned "It's too hot!".
- When the `temperature` is less than 40, then `reaction` will be assigned "It's too cold!".
- Otherwise, `reaction` will be assigned "It's just right!".

To really understand `when`, let's look at the different parts:



1. First, there's `when`. This is a [keyword](#) that tells Kotlin that you want to do something different depending on the circumstances. Note that `when` is an [expression](#), so it is evaluated and can be assigned to a variable. In this case, it will evaluate to a [string](#), which will be assigned to the `reaction` variable.
2. Next, there are `temperature > 55` and `temperature < 40`. These are called **conditions**. Conditions are expressions that evaluate to a [Boolean](#). `temperature > 55` just asks the question, "is the temperature variable is greater than 55?" If is indeed greater than 55, then it evaluates to `true`. Otherwise it evaluates to `false`. When you need to compare numbers in Kotlin, you can use things like the greater-than sign `>` or the less-than sign `<`. The fancy term for these is **comparison operators**. See the chart below for more information.

3. When the condition evaluates to `true`, then the `when` expression will evaluate to the value on the right side of the arrow `->`.
4. At the bottom, we have `else`, which is a catch-all branch that's used in case *none* of the conditions above it match.

For the Nerds

Comparison and Equality Operators

When writing code, we sometimes need to compare one value to another. In other words, we want to know if one value is greater than, less than, or equal to another value.

Kotlin gives us a variety of comparison operators that we can use for this purpose.

Operator	Description	Example
<code><</code>	Less than	<code>x < 5</code>
<code>></code>	Greater than	<code>x > 5</code>
<code>==</code>	Equal to	<code>x == 5</code>
<code><=</code>	Less than or equal to	<code>x <= 5</code>
<code>>=</code>	Greater than or equal to	<code>x >= 5</code>
<code>!=</code>	Not equal to	<code>x != 5</code>

Each of these examples evaluates to a `Boolean` value that tells you whether the expression is true.

For example, `x < 5` evaluates to `true` when `x` is equal to 2, but `false` if `x` is equal to 9.

Technically, `==` is called the *equality operator*, `!=` is called the *inequality operator*, and the rest are called *comparison operators*.

The `when` expression is quite helpful, and you'll probably use it often. It also has some interesting characteristics. Let's look at a few of them now.

The First True Condition Wins

Goldilocks just found some more porridge in the freezer, so let's add another case for extremely cold temperatures:

```
val temperature = -5

val reaction = when {
    temperature > 55 -> "It's too hot!"
    temperature < 40 -> "It's too cold!"
    temperature < 0 -> "It's frigid!"
    else -> "It's just right!"
}
```

Listing 3.2 - Adding a fourth branch to a `when` expression, for temperatures below zero.

In this code listing, because the `temperature` variable is set to `-5`, you might expect `reaction` to be set to "It's frigid!", but actually, it would be set to "It's too cold!".

Why is that?

Because Kotlin works through the `when` cases from *top to bottom*.

1. First, `temperature > 55` evaluates to `false`, because `-5` is not greater than `55`.
2. Then, `temperature < 40` evaluates to `true`, because `-5` is indeed less than `40`.

At this point, Kotlin uses "It's too cold!", because it found a matching condition. So, the third case - `temperature < 0` - is *never evaluated*.

In other words, the first condition that evaluates to `true` will win.

It's easy to make sure that "It's frigid" is used when the temperature is below zero. Just put that case *before* the condition where the temperature is below 40 degrees, like this:

```
val temperature = -5

val reaction = when {
    temperature > 55 -> "It's too hot!"
    temperature < 0  -> "It's frigid!"
    temperature < 40 -> "It's too cold!"
    else              -> "It's just right!"
}
```

Listing 3.3 - Correctly ordering the branches so that the below-zero condition will match.

Now, when you run this code, the `temperature < 0` case will run before `temperature < 40`, so `reaction` will be "It's frigid!".

For the Nerds

If you've written code in some other languages such as Java or C, you might have used a [switch statement](#).

One important difference between a `switch` statement and Kotlin's `when` expression is that `switch` statements have *fall-through*, which means that when one condition matches, all statements below it (even statements for other conditions) will run unless you remember to use a `break` statement.

In Kotlin, `when` has no fall-through, so only one case will get executed.

Exact Matches

Sometimes, instead of checking for a *range* of values (e.g., "anything greater than 55", or "anything less than 40") we only need to check for an *exact match*.

After the Three Bears discovered how much Goldilocks liked their porridge, they decided to write a cookbook that includes all of their family recipes. They opened an online store and began selling their books. Since the book is so popular, they wanted to give discounts to people who buy multiple copies of it.

Quantity Ordered	Price Per Book
1 book	\$19.99 each
2 books	\$18.99 each
3 books	\$16.99 each
4 books	\$16.99 each
5 or more books	\$14.99 each

We *could* write the `when` statement like this:

```
val quantity = 3

val pricePerBook = when {
    quantity == 1 -> 19.99
    quantity == 2 -> 18.99
    quantity == 3 -> 16.99
    quantity == 4 -> 16.99
    else             -> 14.99
}
```

Listing 3.4 - A when expression with every condition using the equality operator (==).

However, Kotlin lets you shorten this up even more, by letting you specify a **subject**. Here's how that looks:

```
val quantity = 3

val pricePerBook = when (quantity) {
    1      -> 19.99
    2      -> 18.99
    3      -> 16.99
    4      -> 16.99
    else   -> 14.99
}
```

Listing 3.5 - Using when with a subject

In this example, `quantity` is called the *subject* of the `when` expression. The subject can be any expression, but here we just used the variable `quantity`. Because we used `quantity` as the subject, we no longer had to put `quantity ==` in each condition!

There's one more shortcut that Kotlin gives us, allowing us to make this even more concise. Since the price-per-book for 3 books is the same as that for 4 books, we can put those conditions on the same line, separated by a comma, like this:

```
val quantity = 3

val pricePerBook = when (quantity) {
    1    -> 19.99
    2    -> 18.99
    3, 4 -> 16.99
    else -> 14.99
}
```

Listing 3.6 - Separating multiple matching values (3, 4) with a comma

Every Condition Must Be Accounted For

A `when` expression in Kotlin *must* include conditions for *all possible values*. For this reason, we say that a `when` expression must be **exhaustive**. This is also the reason that you usually need to include an `else` condition at the bottom.

Sometimes you can be exhaustive even *without* an `else` condition. For example, consider a Boolean expression. As you recall, Boolean variables have only two possible values - `true` and `false`.

Kotlin *knows* that a Boolean value can only ever be `true` or `false`. So, as long as we have a branch for both of those cases, then we won't need an `else` case.

```
val isLightbulbOn = true

val reaction = when (isLightbulbOn) {
    true  -> "It's bright"
    false -> "I can't see"
}
```

Listing 3.7 - A when expression that is exhaustive without an else branch.

This isn't the only occasion when you can omit the `else` condition. We'll see this again when we learn about [enum classes](#) and sealed classes.

Quick Tip

`when` expressions are a lot easier to read when the conditions, arrows, and values are all lined up into columns, because it makes it easy to visually scan.

To help with this, IntelliJ IDEA and Android Studio include a **code formatter** that will organize the spacing of your `when` expression so that it's consistent. Below you can see how it looks without (left) and with (right) the setting turned on.

To turn this on, open the *Settings/Preferences* menu, and then go to *Editor -> Code Style -> Kotlin*. Pick the *Wrapping and Braces* tab, and then place a check mark next to *Align 'when' branches in columns*.

Once you turn this on, you can go to the *Code* menu, and choose *Reformat Code*. The file you have open currently will be reformatted according to the settings, which will now include `when` expressions that are nicely organized and easy to read!

```
val reaction = when {
    temperature > 55 -> "It's too hot!"
    temperature < 0 -> "It's frigid!"
    temperature < 40 -> "It's too cold!"
    else -> "It's just right!"
}
```

```
val reaction = when {
    temperature > 55 -> "It's too hot!"
    temperature < 0 -> "It's frigid!"
    temperature < 40 -> "It's too cold!"
    else -> "It's just right!"
}
```

The `when` expression is powerful, and as a Kotlin developer, you'll use it a lot! But it can be a bit heavy-handed for simple Boolean checks like for our lightbulb above. For these cases, Kotlin gives us a second kind of conditional.

if Expressions

We can make our lightbulb example even more concise by using an `if` expression instead of a `when` expression. `if` expressions are kind of like `when` expressions, but they have a *single* condition, and two cases - one for `true` and one for `false`.

Here's how it looks:

```
val reaction =
    if (isLightbulbOn) "It's bright" else "I can't see"
```

Listing 3.8 - An `if` expression in Kotlin.

And here are the different pieces:

```
val reaction = if (isLightbulbOn) "It's bright" else "I can't see"
                ↑           ↑           ↑
                condition   true branch  false branch
```

A diagram illustrating the structure of an `if` expression. The code is shown as `val reaction = if (isLightbulbOn) "It's bright" else "I can't see"`. Above the first `if` keyword, there is a curved arrow pointing down to it, labeled "if" keyword. Below the code, three arrows point upwards to the three main parts: the condition (`isLightbulbOn`), the true branch (`"It's bright"`), and the false branch (`"I can't see"`).

`if` is a keyword, like `when`. `isLightbulbOn` is the condition that is evaluated.

- If it evaluates to `true`, then `reaction` will be assigned "`It's bright`"
- If it evaluates to `false`, then `reaction` will be assigned "`I can't see`"

If you like, you may format this so that the branches are on different lines. For example:

```
val reaction =  
    if (isLightbulbOn)  
        "It's bright"  
    else  
        "I can't see"
```

Listing 3.9 - An if expression spanning multiple lines.

You can also use braces { and } around the branches. Doing so allows you to add statements to the branches, like this:

```
var isLightbulbOn = true  
  
val reaction =  
    if (isLightbulbOn) {  
        isLightbulbOn = false  
        "I just turned the light off."  
    } else {  
        isLightbulbOn = true  
        "I just turned the light on."  
    }
```

Listing 3.10 - An if expression with braces, allowing for statements inside.

Generally, as a Kotlin programmer, if you've got a single condition to check, you use an `if` expression. If you've got multiple conditions to check, you use a `when` expression.

It's *possible* to use `if` for more than one condition. For example, remember our price-per-book lookup in Listing 3.5? We *could* rewrite it into an `if` expression, as in Listing 3.11 below.

```
val pricePerBook =  
    if (quantity == 1)  
        19.99  
    else if (quantity == 2)  
        18.99  
    else if (quantity == 3)  
        16.99  
    else if (quantity == 4)  
        16.99  
    else  
        14.99
```

*Listing 3.11 - How **not** to use if and else in Kotlin.*

If you find yourself doing this, stop!

Use a `when` expression instead!

when and if as Statements

So far, we've used the `when` and `if` conditionals as expressions, but you can also use them as [statements](#). Here's an example of using an `if` statement in Kotlin:

```
var wattage = 0
var lightbulbIsOn = true

if (lightbulbIsOn) {
    wattage = wattage + 12
}
```

Listing 3.12 - Using `if` as a statement rather than an expression

A few interesting things to note here:

- Unlike conditional *expressions*, conditional *statements* do not require an `else` branch. In the example above, if `lightbulbIsOn` is `false`, then the single branch that's there (`wattage = wattage + 12`) will simply never run.
- We did not assign this `if` to a variable, because as you recall, you [cannot assign a statement to a variable](#).

Best Practice

So far in this book, whenever we've needed to change a variable based on its existing value, such as needing to increase `wattage` by 12, we've written it plainly like this:

```
wattage = wattage + 12
```

In Kotlin, there's a shortcut for this, and here's how it looks:

```
wattage += 12
```

In fact, this works with all of the **arithmetic operators** (+, -, *, /) that we've used so far:

`a = a + 1` can be written as `a += 1`

`b = b - 2` can be written as `b -= 2`

`c = c * 3` can be written as `c *= 3`

`d = d / 4` can be written as `d /= 4`

In Kotlin, it's more common to use this shortcut, so from now on, we'll write it this way.

Summary

In just three chapters, you've already learned a lot about Kotlin! You know about variables, types, expressions, statements, and functions. And in this chapter, you learned all about *conditionals*, including:

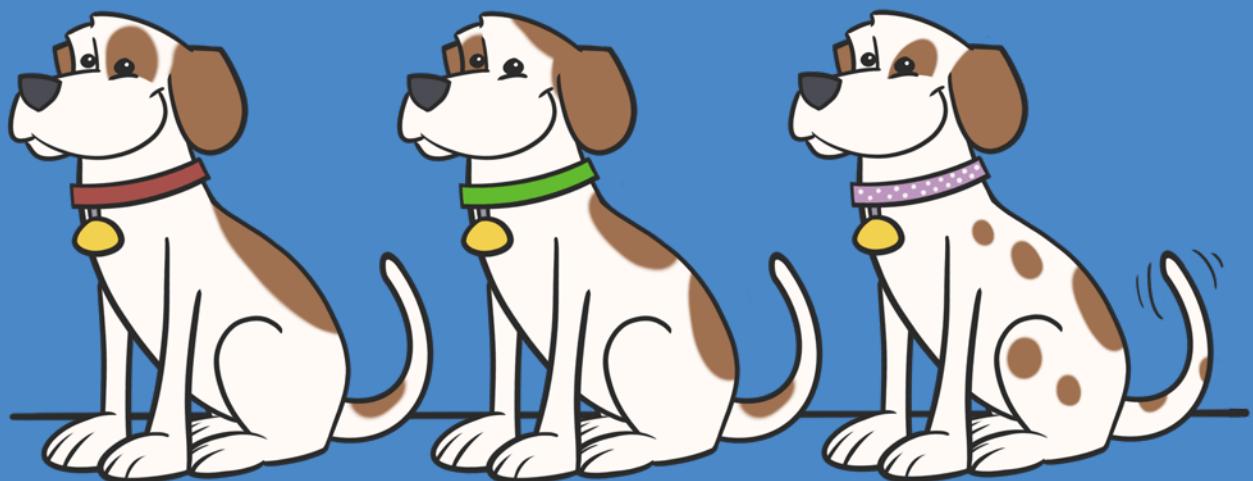
- What [branches](#) and [conditions](#) are.
- How the [when](#) expression can be used to pick a different value in different situations.
- How an [if](#) expression can be a good fit for cases where you've only got two branches.
- How you can use [when and if as statements](#).

In this book, we've used a variety of types, such as integers, strings, and Booleans. In the next chapter, we'll start putting variables and functions together to create *our very own types*, using *classes* and *objects*. These concepts will open many exciting possibilities for us to start writing more advanced programs.

Kotlin: An Illustrated Guide

Chapter 4

Introduction to Classes and Objects



Let's Create Our Own Types!

In Chapter 1, we learned about a variety of *types* in Kotlin, such as **Double**, **String**, and **Boolean**.

Get ready, because in this chapter, we're going to start creating our *very own* types! We'll put variables and functions together into a *class*.

Let's get started!

Putting Variables and Functions Together

As you might recall, we started off this book by creating [variables](#) to hold things pertaining to a circle, such as its radius and circumference. And in chapter 2, we created a [function](#) to calculate the circumference from the radius.

It all looked kind of scrambled, like this:

```
val pi = 3.14
var radius = 5.2
val circumferenceOfSmallCircle = 32.65
radius = 6.7
val circumferenceOfMediumCircle = 42.07
fun circumference(radius)
    radius = 10.0
    val circumferenceOfLargeCircle = 62.8
```

This was manageable for such a simple example, but as we come up with more and more things that we want to know about the circle, such as its diameter, area, or position, it can become quite difficult to manage all of those different variables and functions. And once we start introducing other shapes, like rectangles and triangles, it becomes even *harder* to keep things straight - for example, which functions give us the area of a circle, and which give us the area of a rectangle?

So in this chapter, we'll create a new type called a `Circle`. This way, instead of having separate variables and functions that hold a radius and a circumference, we can have a *single* variable that represents the *circle itself*.

So instead of the scrambled variables and functions above, it'll look more like this:

<code>smallCircle</code>	<code>mediumCircle</code>	<code>largeCircle</code>
<code>val pi = 3.14</code>	<code>val pi = 3.14</code>	<code>val pi = 3.14</code>
<code>val radius = 5.2</code>	<code>val radius = 6.7</code>	<code>val radius = 10.0</code>
<code>fun circumference() = 32.65</code>	<code>fun circumference() = 42.07</code>	<code>fun circumference() = 62.8</code>

In Kotlin, we can put related variables and functions together using a `class`, which is a feature that we will use to create our new `Circle` type.

Defining a Class

Let's create our very first class, a `Circle`, which will include these variables and functions:

- The `radius` variable
- The read-only `pi` variable
- The `circumference()` function

Instead of jumping into all of this at once, let's build up this class slowly... one step at a time... and look carefully at each part.

An Empty Class

First, let's define an empty class - with no variables and no functions.

```
class Circle
```

Listing 4.1 - A simple, empty class in Kotlin.

This is all it takes to create a new class called `Circle`!

When we create a class, we're creating a new type, just like `Int`, `Double`, and `String`. In other words, once we define the `Circle` class like this, we can use it anywhere that we'd normally put a type, such as a [function parameter](#).

```
fun draw(circle: Circle) {  
    // Code that draws the circle would go here  
}
```

Listing 4.2 - Using our `Circle` class as a function parameter.

By the way...

This is the first time in this series that we've used a **comment** in our code.

A section of text that starts with two forward slashes `//` is a *comment*. When Kotlin finds a comment like this, it will ignore the rest of the line, so it has no effect on how the program runs. In this book, we will sometimes use comments to describe parts of the code, or to elide parts that are irrelevant, as we're doing here.

Our First Diagram

It's often useful to *diagram* your classes, so that you can visualize them and communicate your ideas to your friends. We'll also use diagrams throughout the rest of this course to help explain concepts.

So, as we're building out this `Circle` class, we'll also diagram it at each stage, using a diagramming standard called the [Unified Modeling Language](#), or *UML* for short.

We'll start simple. To show a class, just put the name of the class in a box, like this. (I'll also add the Kotlin code alongside the diagram, so that you can compare them).



Now that we've got our new `Circle` type, we could create a variable to hold it, but since our class is completely empty, it's not particularly useful yet. So, before we do that, let's give our `Circle` a `radius`!

Adding a radius Property

In real life, every circle has a radius, so we need to make sure that every `Circle` in our code also has a `radius` variable. Here's how we can do that:

```
class Circle(var radius: Double)
```

Listing 4.3 - Kotlin class with a property.

Here, we added a new variable to the class, called `radius`, which has a type of `Double`. The value of `radius` can be changed, because of the `var` that comes before it. In Kotlin, a variable in a class like this is called a **property** of the class.

Let's also update our diagram to show that a `Circle` class has a `radius` property. To do this, we draw a horizontal line under the name of the class, and write out the name of the property and its type, almost identically to how we write it in our Kotlin code:



Now that we've got a circle class with a radius, we're ready to start using it!

Objects

In this book, we've already created lots of variables. For example, here's how we can [declare](#) and [assign](#) a variable with a `Double` type:

```
val radiusOfSmallCircle: Double = 5.2
```

Listing 4.4 - Creating a variable that holds a Double.

As mentioned, when we created the class, we made a new type called `Circle`. Just like you can have a variable that's a `Double` type, you can also have a variable that's a `Circle` type. Declaring and assigning a `Circle` variable is easy:

```
val smallCircle = Circle(5.2)
```

Listing 4.5 - Creating a variable that holds a Circle.

This creates a new variable called `smallCircle`, which is assigned a `Circle` with a radius of `5.2`.

Keep in mind - just as you can have many different `Double` values like...

- `5.2`
- `6.7`
- `10.0`

... you can also have lots of different `Circle` values like...

- A circle with a radius of `5.2`
- A circle with a radius of `6.7`
- A circle with a radius of `10.0`

But when it comes to classes, instead of calling these *values*, we usually call them **objects**.

Constructing Objects

Let's look at that code again.

```
// Declaring the class
class Circle(var radius: Double)

// Using the class
val smallCircle = Circle(5.2)
```

What's happening when we write `Circle(5.2)`?

It's kind of like we're [calling a function](#) named `Circle()` that has a parameter called `radius` and a [return type](#) of `Circle`. These kinds of functions aren't called *functions*, though. They're called **constructors** because they *construct* a new object.

Constructor Parameters

Note that when you call a constructor, you must provide an argument for every property that is listed between the constructor's opening and closing parentheses - (and). Since we put `var radius: Double` between those parentheses, we have to provide an argument of type `Double` when we call the constructor.

```
class Circle(var radius: Double)
           ↑
           ↗
           ↘
           ↓
val smallCircle = Circle(5.2)
```

Note that radius is actually filling two roles:

1. It's a **constructor parameter**. You have to provide a value for it whenever you call the constructor. (However, just like with function parameters, you can also give a constructor parameter a [default argument](#)).
2. It's a *property*. You'll be able to get the value of the *radius* from any circle object. We'll see an example of this in a moment.

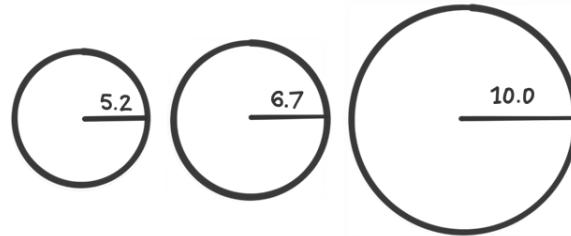
When you *create* an object, we say that you're creating an **instance** of the class. For that reason, creating an object is sometimes referred to as **instantiating** the class.

Classes vs. Objects

The difference between classes and objects can be confusing at first, so let's take a moment to clarify it.

A *class* describes the *characteristics and behavior* of some concept. If we're talking about circles, those characteristics might include its radius, diameter, circumference, and area.

An *object* is an actual particular *instance* of that thing. Here are three circle *objects*:



A circle *class* answers questions like:

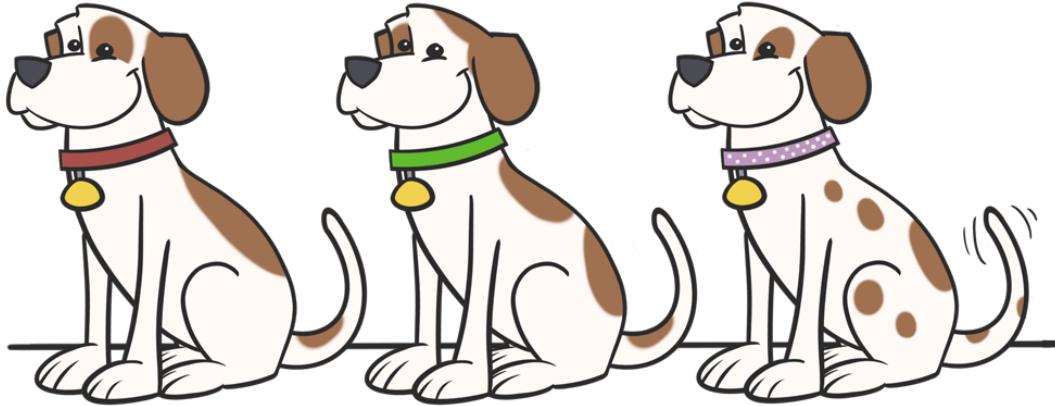
- “What does it mean for something to be a circle?”
- “What characteristics does it have?”
- “What does it do?”

A circle *object* answers questions like:

- “What is the radius of this particular circle?”
- “What is its circumference?”
- “What is its area?”

Here are a few more examples to help distinguish between classes and objects.

- You might have a *Number* class, with objects like 32,768 or 6.62607015.
- You might have a *Color* class, with objects like red, green, and blue.
- You might have a *Dog* class, with objects like Fido, Rover, or Mrs. Waggytails.



We'll see plenty more examples of classes and objects throughout the rest of this book!

Getting a Property's Value

Now that we've created a circle object, how can we get its radius?

Easy: to get the value of a property on an object, type the name of the *variable*, a *dot*, and the name of the *property*, like this:

```
val smallCircle = Circle(5.2)
val radiusOfSmallCircle: Double = smallCircle.radius
```

Listing 4.6 - Getting the value of radius.

After running that code, `radiusOfSmallCircle` will be equal to 5.2.

That takes care of `radius`. It's time to move onto the other property, `pi`.

Constant Properties

We *could* put `pi` inside the parentheses like we did for `radius`, separating them with a comma:

```
class Circle(var radius: Double, val pi: Double)
```

Listing 4.7 - Adding a second property, which you'll have to provide when calling the constructor.

But if we do this, then we'd have to provide `pi` *every time* we instantiate a circle, like this:

```
val smallCircle = Circle(5.2, 3.14)
val mediumCircle = Circle(6.7, 3.14)
val largeCircle = Circle(10.0, 3.14)
```

Listing 4.8 - Providing the value of the `pi` property when calling the constructor.

Hmm... that's not quite what we want. Since `pi` should *always* be the exact same value regardless of the particular circle, it doesn't make sense for it to be a constructor parameter.

In fact, it would be better if the calling code could *never* specify the value of `pi` when constructing a `Circle`.

To do that, we can simply move `pi` out of the parentheses, so that it looks like this:

```
class Circle(var radius: Double) {
    val pi: Double = 3.14
}
```

Listing 4.9 - A class with a body. It has two properties but only one constructor parameter.

Here, we added an opening brace `{` and a closing brace `}`. Everything between those braces is called the **body** of the class. Inside of the body, we declare `pi`, and assign it a value of `3.14`.

By moving it out of the parentheses, the `pi` property is no longer a constructor parameter, so we can continue to call the constructor with just the radius, as we did back in Listing 4.5:

```
val smallCircle = Circle(5.2)
```

Listing 4.10 - We no longer have to pass a second argument to the constructor.

Private Properties

As it is now, you can take any circle and get the value of both `radius` and `pi`:

```
val smallCircle = Circle(5.2)

val radiusOfSmallCircle = smallCircle.radius
val piFromSmallCircle = smallCircle.pi
```

Listing 4.11 - Getting the values of the `radius` and `pi` properties.

I can't think of too many reasons why code *outside* of the class would need the value of `pi`. Let's make it so that this property is *only* visible from *inside* the class. To do that, we can add the keyword `private` when we declare it.

```
class Circle(var radius: Double) {
    private val pi: Double = 3.14
}
```

Listing 4.12 - Making `pi` private, so that it is only visible from within the class body.

Now, if you try to get the value of the `pi` property from *outside* of the class body, you'll get an error:

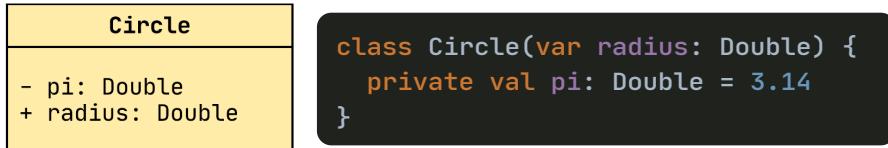
```
val smallCircle = Circle(5.2)

val radiusOfSmallCircle = smallCircle.radius
val piFromSmallCircle = smallCircle.pi
```

Listing 4.13 - Error: "Cannot access 'pi': it is private in 'Circle'" (does not compile)

Let's update our UML class diagram to include the `pi` property. We can indicate the *visibility* of a property in the diagram:

- Private properties are preceded with a `-` symbol
- Public properties are preceded with a `+` symbol



For the Nerds

Visibility Modifiers

`private` is one of a handful of keywords that can be used to tell Kotlin how “visible” you want a property or function to be. By marking `pi` with `private`, we made it so that it is only visible inside the body of the class. Any code outside of the class can't see that property.

Keywords that control the visibility of a property or function are called **visibility modifiers**.

If you don't use a visibility modifier, as is the case with `radius`, then it's the same thing as marking it as `public`, which means that you can see that property or function anywhere.

There are two other visibility modifiers - `protected` and `internal`. We'll look at those later in the series, after we cover some related concepts.

Now, we're ready to add the `circumference()` function to the class!

Adding a Member Function

When a function belongs to a class, it's often called a **method** or **member function**. Adding a method to a class is easy - just put it into the body of the class. To start with, let's just take the exact same function from Listing 2.3, and plop it verbatim into our `Circle` class:

```
class Circle(var radius: Double) {  
    private val pi: Double = 3.14  
  
    fun circumference(radius: Double) = 2 * pi * radius  
}
```

Listing 4.14 - Adding the `circumference()` function.

When first we created the `circumference()` function back in Chapter 2, it made sense for it to have a parameter called `radius`. But now that we're adding this function to the *class*, we can just refer to the value of the `radius` property instead.

In other words, instead of referring to the `radius` parameter, like this...

```
class Circle(var radius: Double) {  
    private val pi: Double = 3.14  
  
    fun circumference(radius: Double) = 2 * pi * radius  
}
```

... we can remove the `radius` parameter from the function, so that it refers to the *property*, like this...

```
class Circle(var radius: Double) {  
    private val pi: Double = 3.14  
  
    fun circumference() = 2 * pi * radius  
}
```

This introduces the concept of **scope**, which we will explore in depth in Chapter 11. For now, just be sure to *remove the parameter* from the `circumference()` function so that `2 * pi * radius` will refer to the `radius` property.

```
class Circle(var radius: Double) {  
    private val pi: Double = 3.14  
  
    fun circumference() = 2 * pi * radius  
}
```

Listing 4.15 - Removing the `radius` parameter from the `circumference()` function.

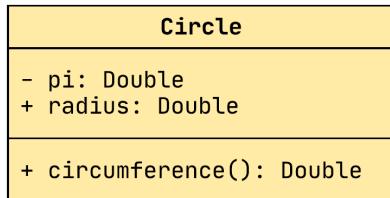
Now that `Circle` has a `circumference()` function, how can we call it?

Calling a function on an object is done similarly to how we got the value of `radius`: the name of the *variable*, a dot, and the name of the *function*.

```
val smallCircle = Circle(5.2)
val circumferenceOfSmallCircle = smallCircle.circumference()
```

Listing 4.16 - Calling a function on an object.

And before we move on, let's add `circumference()` to our diagram!



```
class Circle(var radius: Double) {
    private val pi: Double = 3.14

    fun circumference() =
        2 * pi * radius
}
```

Note that the diagram does not include the *body* of the function. In other words, `2 * pi * radius` does not appear in it. That's because class diagrams are designed to give you an idea of what data and behavior are a part of the class, without going into the specifics.

Adding More Functions

When we started off this chapter, we only had a `radius` variable and a `circumference()` function. Now that we've put those two things together into a class, it's time to fill out our `Circle` class with other things that you might want to know about a circle.

For example, we can add a function that calculates the *area* of the circle.

```
class Circle(var radius: Double) {
    private val pi: Double = 3.14

    fun circumference() = 2 * pi * radius
    fun area() = pi * radius * radius
}

val smallCircle = Circle(5.2)
val areaOfSmallCircle = smallCircle.area()
```

Listing 4.17 - Adding a second function to our class, and calling that function.

We can also add a function to calculate its *diameter*.

```
class Circle(var radius: Double) {  
    private val pi: Double = 3.14  
  
    fun circumference() = 2 * pi * radius  
    fun area() = pi * radius * radius  
    fun diameter() = 2 * radius  
}  
  
val smallCircle = Circle(5.2)  
val diameterOfSmallCircle = smallCircle.diameter()
```

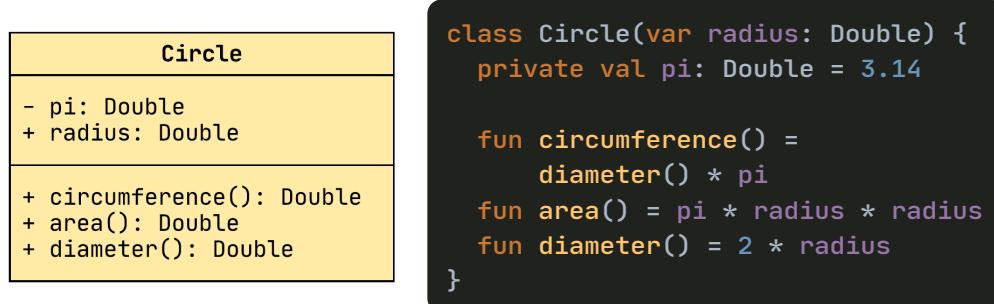
Listing 4.18 - Adding a third function to our class, and calling it.

And, we use the `diameter()` function inside `circumference()`.

```
class Circle(var radius: Double) {  
    private val pi: Double = 3.14  
  
    fun circumference() = diameter() * pi  
    fun area() = pi * radius * radius  
    fun diameter() = 2 * radius  
}
```

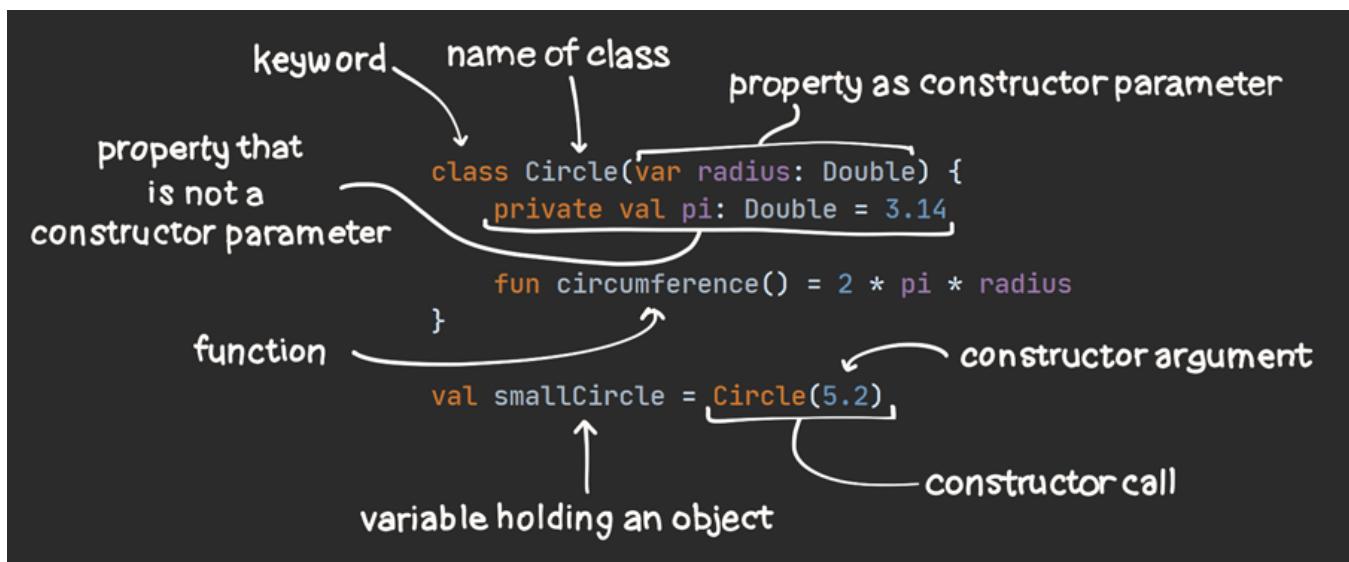
Listing 4.19 - Rewriting the `circumference()` function so that it uses the `diameter()` function.

And now, we can add these last few functions to our UML diagram.



Anatomy of a Class

Now that we've covered the basics of Kotlin classes, here's a recap of the main pieces:



Everything is an Object

Up until this chapter, we've only used built-in Kotlin types, such as `Double`, `String`, and `Boolean`. You might be surprised to learn that when we used those types, we were actually using classes and objects! Just like we used the dot to get properties and call functions on our `Circle` objects, we can also use a dot on any of these types.

Doubles as Objects

For example, objects of type `Double` have functions like `plus()` and `times()`. So instead of writing our `circumference()` function like this...

```
fun circumference() = 2 * pi * radius
```

Listing 4.20 - The `circumference()` function, using arithmetic operators.

... we can instead write it like this ...

```
fun circumference() = 2.times(pi).times(radius)
```

Listing 4.21 - The `circumference()` function, using function calls.

Kotlin developers normally use the *arithmetic operators* (`+`, `-`, `*`, `/`) in most cases, but the functions are there if you want them!

Strings as Objects

`String` objects also have some interesting properties and functions. Here are a few examples:

- The `length` property tells you how many characters (i.e., letters, numbers, and symbols) are in the string.
- `uppercase()` will force all of the letters in the string to upper case.
- `drop()` will remove characters from the beginning of the string.

Here's how that looks in code:

```
val greeting: String = "Welcome"

val numberofLettersInGreeting = greeting.length // Evaluates to 7
val loudGreeting = greeting.uppercase()           // Evaluates to "WELCOME"
val substring = greeting.drop(3)                  // Evaluates to "come"
```

Listing 4.22 - Getting a property and calling functions on a `String` object.

Booleans as Objects

Even `Boolean` variables - which are only ever `true` or `false` - are objects! For example, if you want to turn on the headlights of your car if *either* it's dark or it's raining, you can write code to do that like this:

```
val isDark: Boolean = true
val isRaining: Boolean = false

val shouldTurnOnHeadlights = isDark.or(isRaining)
val shouldStayHome = isDark.and(isRaining)
```

Listing 4.23 - Calling functions on a `Boolean` object.

Best Practice

Although it's possible to use functions like `or()` and `and()` on a Boolean variable, it's usually a better idea to use the operators `||` and `&&` instead, like this:

```
val shouldTurnOnHeadlights = isDark || isRaining
val shouldStayHome = isDark && isRaining
```

The fancy words for `||` and `&&` are **disjunction operator** and the **conjunction operator**, respectively, but almost all programmers just call them "or" and "and". The reason to favor the operators over the function calls has to do with a concept called **short-circuiting**, and here's how it works:

- When using `||`, if the expression on the left evaluates to true, Kotlin knows that the result of the whole thing must be true, so it won't bother evaluating the expression on the right.
- When using `&&`, if the expression on the left evaluates to false, Kotlin knows that the result of the whole thing must be false, so it won't bother evaluating the expression on the right.

When you're just writing a simple case like above, where you've got two Boolean variables, this won't make much of a difference, but if you've got function calls that take a long time to calculate their result, this could be a big deal.

```
val shouldGetRaise = yearsOfService > 1 && calculateSalary() < maximumSalary
```

Boolean values, conjunction, and disjunction are all part of the wonderful world of [Boolean algebra](#).

As you can see, even simple values are objects in Kotlin!

Summary

Classes are powerful, and even though we covered a lot of ground in this chapter, we've really only introduced them. They open up an entire new world of ways to represent your concepts in code. Later in this series, we'll cover more advanced concepts like abstraction and inheritance. But for now, you should be proud of your progress!

In this chapter, we finally created our very own types, using classes. We learned:

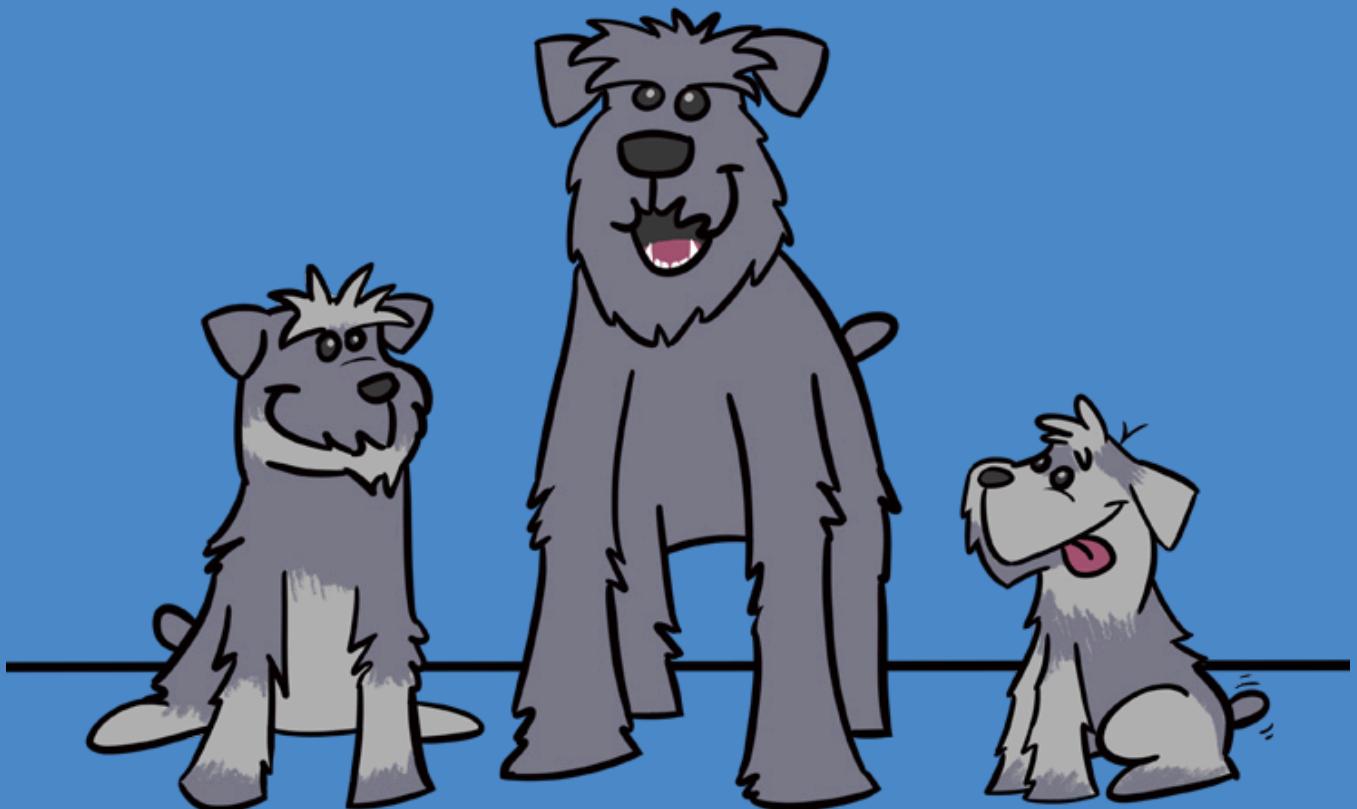
- How to [define a class](#).
- How to create an [object](#) - that is, an *instance* of a class.
- How to add [properties](#) to a class, and how to access them.
- How to add [functions](#) to a class, and how to call them.
- How to create a [UML diagram](#) for a single class.
- How to call functions and get properties from the [built-in Kotlin types](#) that we've already worked with, such as `Double`, `String`, and `Boolean`.

In the next chapter, we'll look at a special kind of class, called an enum class.

Kotlin: An Illustrated Guide

Chapter 5

Enum Classes



Sometimes limiting your options is a good thing...

In the last chapter, we created our very own type, called `Circle`, using a feature in Kotlin called a class.

In this chapter, we're going to look at a special kind of class - an enum class - which is particularly useful when you want to represent a *limited number* of values.

I hope you like schnauzers, because this chapter is chock full of 'em!

My favorite kind of dog is the Schnauzer. In fact, I've got one lying down next to me right now as I'm writing this chapter! She's a little grumpy about the click-clacking of my keyboard while she's trying to nap, but I'll make it up to her later when we play some fetch.

Anyway, let's create a **String** variable to hold the name of a pet Schnauzer like this:

```
val nameOfSchnauzer: String = "Shadow"
```

Listing 5.1 - Creating a variable to hold the name of a Schnauzer.

How many different names for Schnauzers can you think of? *Shadow*, *Rover*, and *Captain Fluffybeard* come to mind for me, but there's really no limit to the variety of names that could be given to them. The possibilities are *unlimited*!

Schnauzer *breeds*, however, are a different story! There are only 3 breeds of Schnauzer:

1. Miniature Schnauzer
2. Standard Schnauzer
3. Giant Schnauzer

So, while the number of *names* for a Schnauzer is *unlimited*, the number of *breeds* is *limited* to just 3 options.

How does all of this relate to programming?

When we're writing code, we can reduce the likelihood of errors by choosing a type that limits the range of possible values to only those that are valid.

For example, a **String** can hold practically any text that you can imagine. Since you can name your dog almost anything imaginable, it would make sense to use a **String** to hold its name.

On the other hand, a **String** would not be a great choice to hold the specific *breed* of a Schnauzer. Why? Because strings can hold an infinite number of different values, but there are only 3 possibilities that could be correct for a Schnauzer breed. In other words, when using a **String**, there's no way to guarantee that you will set a breed variable to one of those three correct values.

To demonstrate this, let's say we decided to use a String to represent the breed, like this:

```
val breedOfSchnauzer: String = "Mini"
```

Listing 5.2 - Creating a variable to hold the breed of a Schnauzer.

Now, somewhere else in our code, we might have a conditional that's checking the breed like this:

```
if (breedOfSchnauzer == "Miniature") {  
    // ... do something  
}
```

Listing 5.3 - A conditional that expected a slightly different breed name than we used above.

We might have intended that the conditional expression would be `true`, but since we incorrectly typed "Mini" instead of "Miniature", it would have evaluated to `false`... and we wouldn't have known that anything was wrong until we ran the code.

The problem is that we used a type that allows *unlimited* values (that is, the `String` type), when we needed a type that allows only a *limited* number of values. If we could create a type that only allows you to use one of the three breed names, we can make it so that the error above is not even possible!

To create a type with *limited* values in Kotlin, you use an *enumeration class*... or as we more often refer to it, an **enum class**, or just an **enum**.

Creating an Enum Class

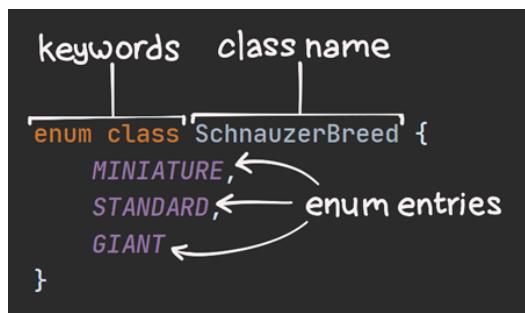
Let's create an enum class to represent Schnauzer breeds.

```
enum class SchnauzerBreed {
    MINIATURE,
    STANDARD,
    GIANT
}
```

Listing 5.4 - Creating an enum class with three options.

This enum class is named `SchnauzerBreed`, and it gives us three breed options to choose from.

Here are the main parts of an `enum class`:



1. To create an enum class, we write `enum class` rather than just `class`.
2. After that comes the name that we want to give our class - in this case, `SchnauzerBreed`.
3. Inside the body of the class, the *options* are called **enum entries**. Sometimes we call them **enum constants** instead. They're separated with commas.

Using an Enum Class

Now that we have an enum class, we can start using it. Note that you will not *construct* an object from an enum class in the same way that you would do with normal classes, or you'll get an error like this:

```
val breed: SchnauzerBreed = SchnauzerBreed()
```

Listing 5.5 - Error: "Cannot access '<init>': it is private in 'SchnauzerBreed' / Enum types cannot be instantiated" (does not compile)

Instead, you just assign a variable to one of the options, like this:

```
val breed: SchnauzerBreed = SchnauzerBreed.GIANT
```

Listing 5.6 - Assigning a variable to one of the enum options.

By using an enum class here, if we were to accidentally type `MINI` instead of `MINIATURE`, then Kotlin would give us an error before we can even run the code!

```
val breed: SchnauzerBreed = SchnauzerBreed.MINI
```

Listing 5.7 - Error: "Unresolved reference: MINI" (does not compile)

So instead of using a `String`, which can be set to just about *anything*, we limited the valid options to just *three* enum entries. By doing that, Kotlin can now help tell us when we typed something wrong, without having to even run the code!

Using Enum Classes with when Expressions

Another nice benefit of using enum classes is that Kotlin provides some added assistance when we use them with a [when conditional](#).



As you might recall, when expressions [must account for every condition](#) - that is, they must be exhaustive. Because enum classes limit the possible values, Kotlin can use this information to know when we have indeed accounted for every condition.

For example, here's some code that returns a description of a breed.

```
fun describe(breed: SchnauzerBreed) = when (breed) {
    SchnauzerBreed.MINIATURE -> "Small"
    SchnauzerBreed.STANDARD -> "Medium"
    SchnauzerBreed.GIANT     -> "Large"
}
```

Listing 5.8 - Using an enum class with a when expression.

Notice that there's no `else` condition here! Kotlin can tell that we've included *all three* of the enum entries in the `when` body, so there are no other possibilities.

If we were to omit some of the entries, Kotlin would give us an error:

```
fun describe(breed: SchnauzerBreed) = when (breed) {
    SchnauzerBreed.MINIATURE -> "Small"
    SchnauzerBreed.STANDARD -> "Medium"
}
```

Listing 5.9 - Error: "when' expression must be exhaustive" (does not compile).

To remedy this error, we would either have to provide *all* of the enum constants, as we did in Listing 5.8, or we'd have to provide an `else` condition, like this:

```
fun describe(breed: SchnauzerBreed) = when (breed) {
    SchnauzerBreed.MINIATURE -> "Small"
    SchnauzerBreed.STANDARD -> "Medium"
    else -> "Unknown"
}
```

Listing 5.10 - Providing an else branch, because not all of the enum entries are represented in the when expression.

It's great that Kotlin will give you an error when your `when` is not exhaustive. For example, whenever you add a new entry to an existing enum class, Kotlin will give you errors in all of the `when` expressions that need to be updated - so you'll know exactly what code needs to be changed!

Adding Properties and Functions to Enum Classes

As you recall from the last chapter, normal Kotlin classes allow us to put [properties and functions together](#). Is it possible to add properties and functions to an enum class?



Yes, it is! Let's start by adding a property as a [constructor parameter](#). We can do this just as we did with normal classes - by putting them between the opening (and closing) parentheses after the name of the class.

For example, we might want to include the approximate height of each breed, in centimeters. We can do that like this:

```
enum class SchnauzerBreed(val height: Int) {
    MINIATURE(33),
    STANDARD(47),
    GIANT(65)
}
```

Listing 5.11 - Adding a constructor parameter to an enum class.

Because we added a new `constructor parameter` called `height`, we also had to add a `constructor argument` to each of the enum entries. So a miniature has an approximate height of 33cm, a standard is about 47cm, and a giant is about 65cm tall.

Note that the enum entries are in fact [instances](#) of the enum class.

You can get a property off of an enum instance just like you would do with regular objects. For example, we can print the height of a breed to the screen, as shown in Listing 5.12 below.

```
enum class SchnauzerBreed(val height: Int) {  
    MINIATURE(33),  
    STANDARD(47),  
    GIANT(65)  
}
```

constructor parameter
↓
constructor arguments

```
println(SchnauzerBreed.MINIATURE.height)
```

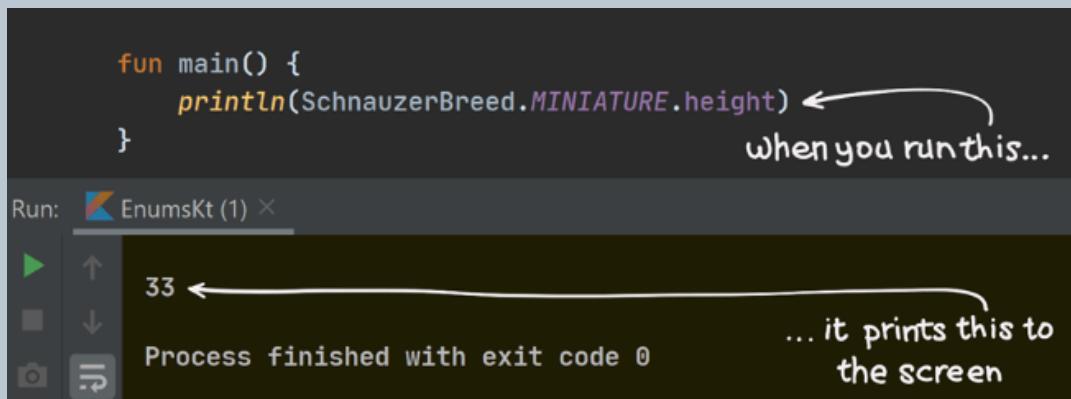
Listing 5.12 - Getting a property of an enum class

For the Nerds

`println()`

This is the first time we've used the `println()` function. It's pronounced "print line", and you use it to write text to the screen. You can pass it any kind of object, and Kotlin will do its best to write information about that object. Most often, though, you'll probably just pass it a `String` directly.

You don't have to define this function. Kotlin automatically makes this function available to you, as a part of its **standard library**.



We can also include a property that does not require a constructor argument. Let's add a property that tells us the `family` of breeds that they all belong to.

```
enum class SchnauzerBreed(val height: Int) {  
    MINIATURE(33),  
    STANDARD(47),  
    GIANT(65);  
  
    val family: String = "Schnauzer"  
}
```

Listing 5.13 - Adding a property to an enum class, without making it a constructor parameter.

The new `family` property is added to the end of the class body. It's important to note that the semicolon is required after the last enum entry! There aren't very many times in Kotlin when you *have* to use a semicolon, but this is one of them.

It's just as easy to add a function. As above, make sure you include the semicolon, and then write the function beneath the enum entries.

```
enum class SchnauzerBreed(val height: Int) {
    MINIATURE(33),
    STANDARD(47),
    GIANT(65);

    val family: String = "Schnauzer"

    fun isShorterThan(centimeters: Int) = height < centimeters
}
```

Listing 5.14 - Adding a function to an enum class.

You can get properties and call functions just as you would with any other class.

```
println(SchnauzerBreed.STANDARD.family)
println(SchnauzerBreed.STANDARD.isShorterThan(40))
```

Listing 5.15 - Using a property and function from an enum class.

Built-In Properties

In addition to any properties that you include yourself, Kotlin also automatically adds a couple of properties to all enum instances.

ordinal

All enum instances have a property called `ordinal`. You can use this to tell *where this enum entry appears in the list*. The first entry in the list has an ordinal of 0, the second has an ordinal of 1, the third has an ordinal of 2, and so on.

For example, since `STANDARD` is the second entry in the list, it has an ordinal of 1.

Yes, it starts with zero, which can be a bit confusing if you haven't done much programming before. Once we get to the topic of *collections* in Chapter 8, we'll see how *zero-based* numbering is used a lot in programming!



```
enum class SchnauzerBreed(val height: Int) {
    0 → MINIATURE(33),
    1 → STANDARD(47),
    2 → GIANT(65)
}
```

name

If you need to know the name of the enum entry as a string, you can use the `name` property. To demonstrate this, we can create a function that tells us the name of the breed, along with the height.

```
fun describe(breed: SchnauzerBreed) {  
    println(breed.name)  
    println(breed.height)  
}
```

Listing 5.16 - Using the `name` property that Kotlin automatically includes in all enum classes.

Now, we can call `describe(SchnauzerBreed.STANDARD)`, and we'll see this on the screen:

```
STANDARD  
47
```

For the Nerds

`valueOf()` - Going the other way

The `name` property can be used to get a `String` from a `SchnauzerBreed`. In some cases, though, you might want to go the other way - to get the `SchnauzerBreed` from a `String`.

To let you do this, Kotlin automatically includes a function called `valueOf()` on the enum class. Here's an example.

```
val nameOfBreed = "STANDARD"  
val breed = SchnauzerBreed.valueOf(nameOfBreed)  
  
println(breed.height) // Prints "47"
```

When this code runs, if you were to call `valueOf()` with a string that doesn't match any entries, you'll get an error.

There's more than this - you can also get each of the `entries` out of an enum class. This won't be helpful until we know how to iterate over values, which we'll cover in Chapter 8.

For now, let's wrap up this chapter!

Summary

In this chapter, we learned:

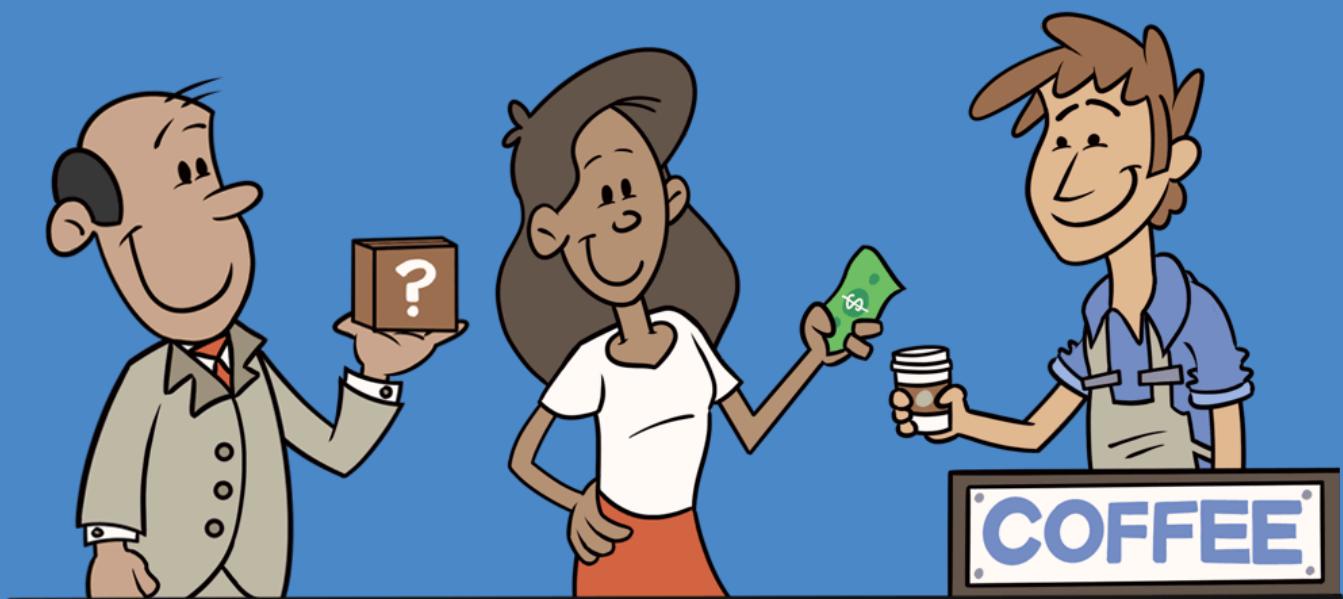
- How [limiting the range of values](#) can help make sure our programs are written correctly
- How to [create an enum class](#)
- How to [use enum classes with the "when" conditional](#)
- How to [add properties and functions](#) to an enum class
- How to use some [built-in properties](#) of enum classes

At this point, we've created many variables that each holds a value. In the next chapter we'll learn about Kotlin's ability to safely handle the *absence* of a value.

Kotlin: An Illustrated Guide

Chapter 6

Nulls and Null Safety



What do you do when there's nothing in that variable...?

So far in this course, every time that we created a variable, whether it was a **String**, an **Int**, or a **Boolean**, we assigned a *value* to it. There are times, though, when we need to create a variable that might *not* actually hold a value!

This brings us to the exciting topic of nulls!

Introduction to Nulls

James has set up a coffee stand downtown, and he's ready to start sharing his fine brew! After handing out each cup, he asks the guest to review the coffee, so that he can share the ratings with others who might be interested.

Since he wants to keep track of the ratings in a Kotlin program, he writes this simple class:

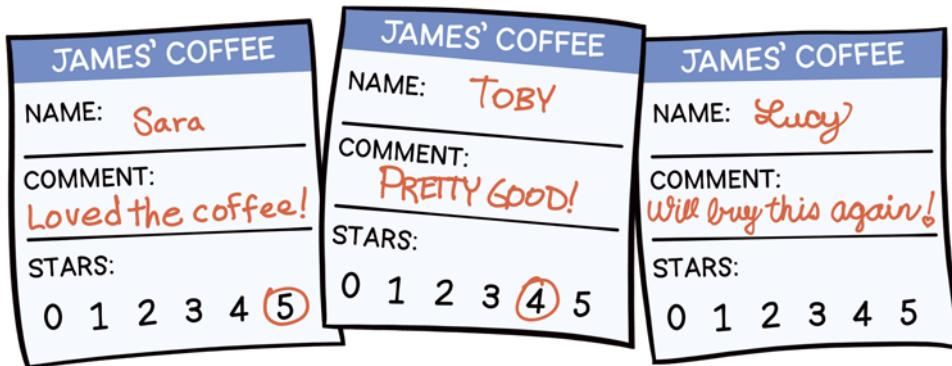
```
class CoffeeReview(  
    val name: String,  
    val comment: String,  
    val stars: Int  
)
```

Listing 6.1 - A simple class that represents what someone thought about James' coffee



The **name** is the name of the person who is reviewing his coffee, the **comment** property is used for any comment that they want to share about it, and the **stars** property holds the star rating - the number of stars that they give the coffee, between 0 and 5.

The first three guests of the day have filled out the review cards - let's see how they rated his coffee!



James instantiates some **CoffeeReview** objects to record the three reviews that he received. He starts with the first two...

```
val saraReview = CoffeeReview("Sara", "Loved the coffee!", 5)  
val tobyReview = CoffeeReview("Toby", "Pretty good!", 4)
```

*Listing 6.2 - Instantiating a few **CoffeeReview** objects.*

When he gets to Lucy's review, though, he noticed that she forgot to leave a star rating. "That's okay," he said to himself, "I'll just use the number zero since she didn't mark any stars."

```
val lucyReview = CoffeeReview("Lucy", "Will buy this again!", 0)
```

Listing 6.3 - Using a star rating of 0 because Lucy forgot to mark a star rating on the review card.

He was ready to show the reviews on a screen, so he wrote a simple function, and called it with each of the reviews that he received:

```
fun printReview(review: CoffeeReview) =
    println("${review.name} gave it ${review.stars} stars!")

println("Latest coffee reviews")
printLine("-----")
printReview(saraReview)
printReview(tobyReview)
printReview(lucyReview)
```

Listing 6.4 - Printing the coffee reviews to the screen.

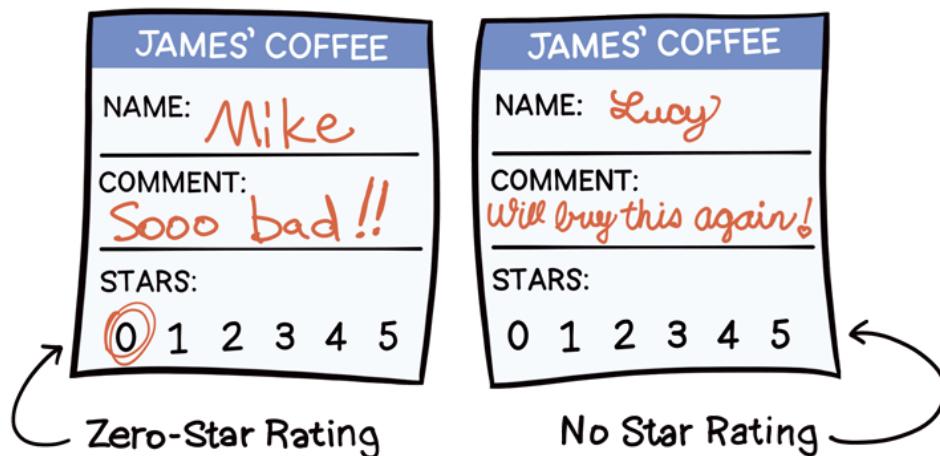
This is what showed on the screen:

```
Latest coffee reviews
-----
Sara gave it 5 stars!
Toby gave it 4 stars!
Lucy gave it 0 stars!
```

He thought that his solution would work well, but when the guests saw the review, they thought, "Wow, if Lucy didn't like the coffee, maybe it's not good. I'll go somewhere else."

Yikes!

James realized that **a zero-star rating is not the same thing as having no star rating.**



When someone doesn't leave a star rating, then James doesn't want to show a zero-star rating on the screen. Instead, he needs a way to tell Kotlin that they didn't leave a star rating at all. How can he do that?

Present and Absent Values

As you might recall from Chapter 1, a [variable](#) is like a bucket that holds a value.

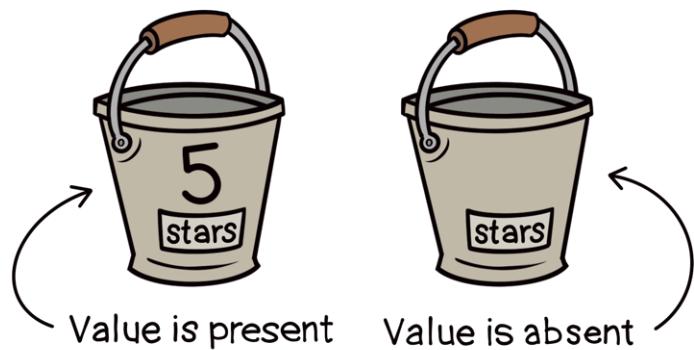
In all the code that we've written so far, we've created variables that contain a *value*. In other words, we've always had something *inside* that bucket. For example, a `stars` bucket contains an `Int`, like the number 5.



For the `CoffeeReview` class, we need a `stars` bucket that *might or might not* have a value in it. When the guest leaves a rating, the bucket needs to contain that value, but when they forget to rate it, that bucket needs to be empty.

So we want a bucket where we can either put a value inside of it, or leave it empty. This brings up two new terms:

- When a variable has a value inside of it, we'll say that the value is **present**.
- When a variable does not have a value inside of it, we'll say that the value is **absent**.



Kotlin uses a [keyword](#) called `null` to represent the absence of a value.¹ A variable that is assigned `null` is like a bucket that is empty.²

For a review where there's no star rating, we want to set `stars` to `null`. We can try giving it a `null` when we construct Lucy's `CoffeeReview`, but when we do that, we get an error:

```
val lucyReview = CoffeeReview("Lucy", "Will buy this again!", null)
```

Listing 6.5 - Error: "Null can not be a value of non-null type Int" (does not compile).

In fact, even apart from the `CoffeeReview` class, if we simply create an `Int` variable called `stars`, we can't assign `null` to it.

```
val stars: Int = null
```

Listing 6.6 - Error: "Null can not be a value of non-null type Int" (does not compile).

Why is this? In some programming languages, you can assign a `null` to any variable. That might sound like a good idea, but it can result in lots of surprises while your code is running, because we never have any guarantees that the value of a variable is present.

In order to help prevent these kinds of problems, Kotlin won't let you assign `null` to just *any* variable. Instead, you have to *clearly indicate* when a variable can be empty. How can we do this?

¹ In some Latin-based languages, the word “null” is more closely related to the number zero, but in English it more often refers to something that has no value or effect. When you see it in Kotlin, don’t think of it as the number zero; think of it as “not having a value”.

² Technically, even `null` itself can be considered a value. After all, in real life, even an empty bucket is technically full of air. So, you might hear someone say, “The value of that variable is `null`,” and that’s fine. However, since it’s easier to learn with the relatable concepts of “present” and “absent”, in this article we’ll regard `null` as the absence of a value rather than as a value itself.

Nullable and Non-Nullable Types

In Kotlin, we use different [types](#) to indicate whether a variable can or cannot be set to `null`.

All of the types that we've used so far in this series - such as `String`, `Int`, and `Boolean` - *require* you to assign an actual value. You can't just assign `null` to them, as we tried above. Since they don't let you assign `null`, we call them **non-nullable types**. In contrast, types that allow you to assign `null` are called **nullable types**.

In other words:

- When you want to *guarantee* that a variable's value will be present - that is, when the value is **required** - then give the variable a *non-nullable type*.
- When you want to *allow* a variable's value to be absent - that is, when the value is **optional** - then give the variable a *nullable type*.

In Kotlin, nullable types end with a question mark.

For example, while `Int` is a *non-nullable type*, `Int?` (ending with a question mark) is a *nullable type*.

For every non-nullable type, a corresponding nullable type exists.

Let's look at our code from Listing 6.6 again:

NON-NULLABLE	NULLABLE TYPES
<code>String</code>	<code>String?</code>
<code>Int</code>	<code>Int?</code>
<code>Boolean</code>	<code>Boolean?</code>
<code>Circle</code>	<code>Circle?</code>
<code>SchnauzerBreed</code>	<code>SchnauzerBreed?</code>

```
val stars: Int = null
```

Listing 6.7 - Error: "Null can not be a value of non-null type Int" (does not compile)

To allow this variable to accept a null, we simply change the type of the variable from the non-nullable type `Int` to the nullable type `Int?`, like this:

```
val stars: Int? = null
```

Listing 6.8 - Notice the question mark at the end of Int.

Now, `stars` can be set to `null`. Of course, it can also still be set to a normal integer value. For example:

```
val saraStars: Int? = 5
val tobyStars: Int? = 4
val lucyStars: Int? = null
```

Listing 6.9 - Setting nullable variables to either non-null values or null.

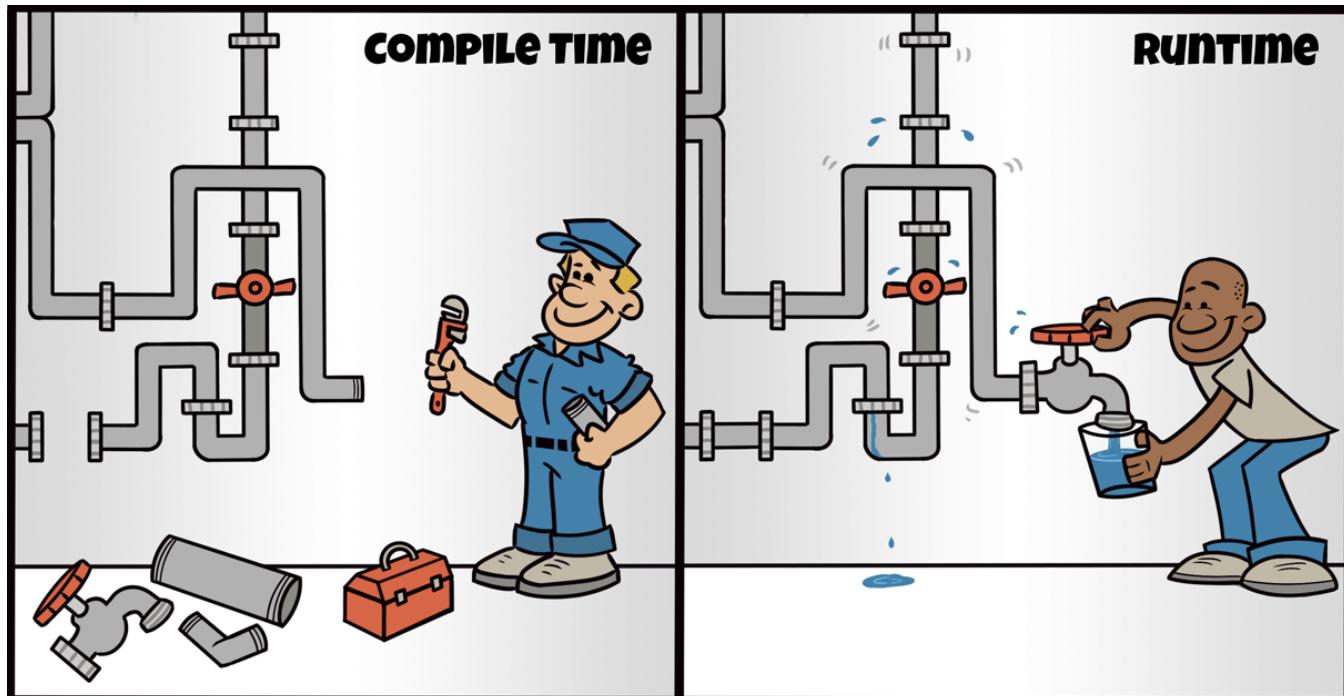
Compile Time and Runtime

The type of a variable tells us whether that variable *can* hold a null, but it cannot tell us whether it actually *does* hold a null.

This brings up an important distinction to consider. There are things we can know when Kotlin is *reading* the code, and there are things we can know when we're *running* the code.

- The point at which *Kotlin is reading our code* is called **compile time**. If we're using an IDE like IntelliJ or Android Studio, this happens while we are *writing the code*.
- The point at which our computer *runs our Kotlin code* is called **runtime**.

You can think of compile time as the point when a plumber is assembling some water pipes, whereas runtime is like the point after someone has turned on the faucet, so water is running through those pipes.



Kotlin knows the type of a variable at *compile time*, which is why it knows whether it can hold a null while we're writing the code. On the other hand, Kotlin won't know whether a variable *actually* holds a null until runtime.

In some simple cases like Listing 6.9, where we're setting the variable with a [literal](#) directly in the code, it seems obvious to us whether the value is present or absent. In fact, your IDE (like IntelliJ or Android Studio) might even warn you when you use a non-null value with a variable that's declared to be nullable - like `saraStars` and `tobyStars` above.

But values can also come from external sources, like databases, files on your hard drive, or when a user types on a keyboard. And once you start calling functions that have parameters, it's possible that the value of the argument is absent when called from one place, and present when called from another.

In order to know whether a variable *actually* has a value at runtime, we have to convert it from a nullable type to a non-nullable type. We'll see some cool tricks for this below! But first, it's important to understand the relationship between nullable and non-nullable types.

How Nullable and Non-Nullable Types are Related

Even though `Int` and `Int?` are related, they're still two different types, and you can't just use an `Int?` anywhere that you would use an `Int`. For example, a function that expects an `Int` won't work if you try to send it an `Int?` instead:

```
fun printReview(name: String, stars: Int) =
    println("$name gave it $stars stars!")

val saraStars: Int? = 5

printReview("Sara", saraStars)
```

Listing 6.10 - Error: "Type mismatch. Required Int / Found Int?" (does not compile)

Why is this so? To understand this, let's turn our attention away from the review screen, and toward the front counter, where James is taking the coffee orders!

Expecting a Nullable Type

At the moment, James is running a non-profit coffee *charity*, which provides warm beverages to anyone, even if they can't pay for it. If the guest would like to donate a payment, they can, but it's not required.

Here's a function that models this arrangement. The payment at a charity is *optional*, so we'll make the payment parameter nullable.

```
fun orderCoffee(payment: Payment?): Coffee {
    return Coffee()
}
```

Listing 6.11 - A function for ordering coffee where payment is optional, such as for a charity.

Naturally, if someone orders a coffee and provides payment, James gladly will give them a coffee!

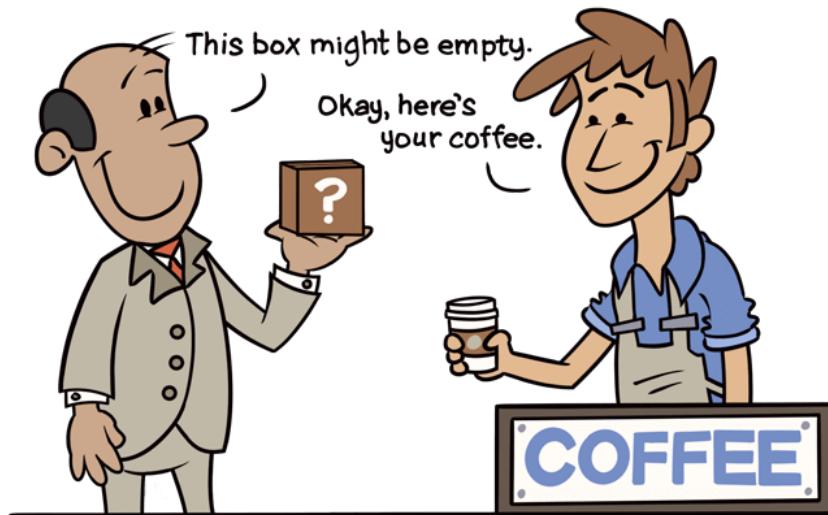


Passing a **Payment** argument to the `orderCoffee()` function is like this scenario - the guest is definitely providing payment.

```
val payment: Payment = Payment()  
val coffee = orderCoffee(payment)
```

Listing 6.12 - Providing payment for a coffee at the coffee charity.

Now, imagine that someone walks in and says, “This box *might* have a payment... or it *might* be empty. You can have whatever is inside.” James says, “Even if it’s empty, that’s fine. We’re a charity, after all. Here’s your coffee!”



When we pass a **Payment?** argument to the `orderCoffee()` function, it’s like the guest is handing James a mystery box that contains either a payment or nothing at all.

```
val payment: Payment? = Payment() // or set this to null  
val coffee = orderCoffee(payment)
```

Listing 6.13 - Maybe providing a payment for coffee... or maybe not!

To summarize, a function that has a nullable parameter like **Payment?** is like a charity - it can accept an argument that has a non-nullable type (like **Payment**), and it can also accept an argument that has a nullable type (like **Payment?**). Either one is fine.

```
val payment: Payment = Payment()  
orderCoffee(payment)  
  
val payment: Payment? = Payment()  
orderCoffee(payment)
```

```
fun orderCoffee(payment: Payment?): Coffee {  
    return Coffee()  
}
```

Expecting a Non-Nullable Type

After a while, James realized that he couldn't get enough donations to sustain the charity, so now he's running his coffee stand as a *business*. All those coffee beans cost money, and since the business doesn't run from donations, he *must* receive *payment* in order to provide coffee to the customer.

Here's a new version of `orderCoffee()` that works like a business rather than a charity. Notice that the parameter has a non-nullable type, because *payment* is now required.

```
fun orderCoffee(payment: Payment): Coffee {
    return Coffee()
}
```

Listing 6.14 - A function for ordering coffee where payment is required, such as for a business.

As before, when someone orders a coffee and provides payment, James will gladly give them a coffee.

Passing a `Payment` variable to this function is like this scenario - the guest is definitely providing payment, so everything works just fine.



```
val payment: Payment = Payment()
val coffee = orderCoffee(payment)
```

Listing 6.15 - Providing payment for a coffee at the coffee business.

Now imagine that someone orders a coffee, but instead of giving him payment, holds out a box, and says, "This box *might* have a payment... or it *might* be empty. I'll trade you whatever is inside this box for a coffee."

"No deal!" he says. "You have to actually *pay* for your coffee! I can't trade the coffee for a *chance* to receive payment. I have to actually *receive* payment!"

Passing a `Payment?` variable to this function is like this scenario - the guest is

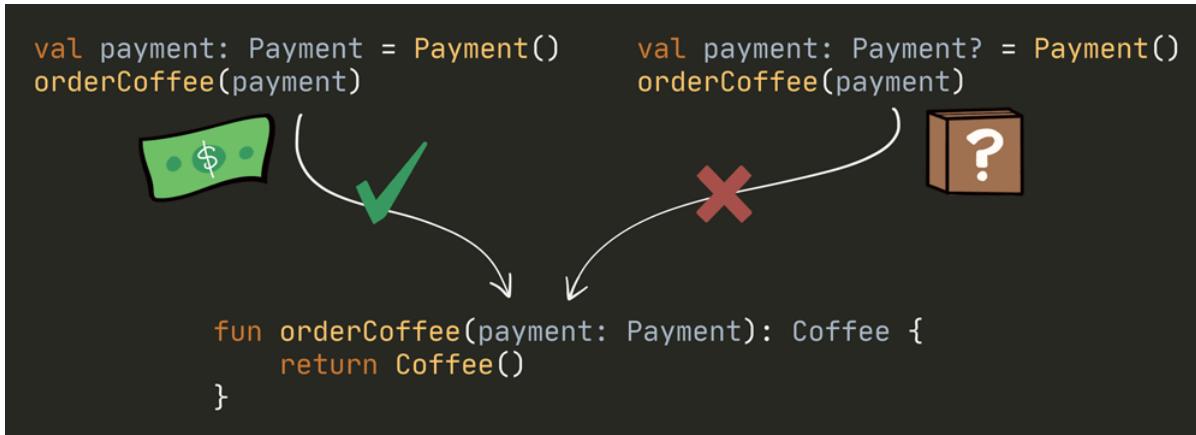


either handing James a payment or nothing at all. Just like James, Kotlin says, “No deal!” (...well, actually it says, “Type mismatch.”)

```
val payment: Payment? = Payment() // or you could set this to null  
val coffee = orderCoffee(payment)
```

Listing 6.16 - Error: "Type mismatch. Required Payment / Found Payment?" (does not compile).

When James *requires* payment, he can’t accept a payment that *might not* be there. It must be there. So a function that has a parameter of type **Payment** is like a business - it cannot accept an argument of type **Payment?**.



To summarize, you can use a non-nullable type (e.g., **Payment**) where a nullable type (e.g., **Payment?**) is expected, but not the other way around.

Now, it’s quite possible that the customer’s box *actually* has a payment inside! If only they would take the payment out of the box, then they could exchange that payment for coffee! Similarly, Kotlin gives us a few different ways to *safely* convert a nullable type to a non-nullable type. Let’s take a look!

Using Conditionals to Check for null

Here again is the function for the coffee shop business:

```
fun orderCoffee(payment: Payment): Coffee {  
    return Coffee()  
}
```

Listing 6.17 - An `orderCoffee()` function for the coffee business.

When the customer tried to pay with a box that might be empty, it looked like this:

```
val payment: Payment? = Payment()  
val coffee = orderCoffee(payment)
```

Listing 6.18 - Error: "Type mismatch. Required Payment / Found Payment?" (does not compile).

One very simple way to still order a coffee in this case is to check to see if `payment` *actually* has a value when the code is running. In other words, we can look inside the box, and if the payment is not `null`, then we can order the coffee.

```
val payment: Payment? = Payment()

if (payment != null) {
    val coffee = orderCoffee(payment)
} else {
    println("I can't order coffee today")
}
```

Listing 6.19 - Using an if conditional to see whether payment is null.

There's no error when you write this code, and `orderCoffee()` will be called when you run it. How does this work? Why can we call `orderCoffee(payment)` in Listing 6.19 but not Listing 6.18? Even though we declared `payment` to be of type `Payment?` (which is nullable), inside the `if` block, its type changes to `Payment` (which is non-nullable)! Kotlin knows that `payment` must have a value inside that block, because we checked for it! This is called a **smart cast**.

```
val payment: Payment? = Payment()

if (payment != null) {
    val coffee = orderCoffee(payment) ← Type of payment is Payment here (non-nullable)
} else {
    println("I can't order coffee today") ] Type of payment is back to Payment? again here
```

Smart casts also work with a `when` conditional, like this:

```
when (payment) {
    null -> println("I can't order coffee today")
    else -> orderCoffee(payment)
}
```

Listing 6.20 - Smart cast using a when conditional.

By the way, this isn't the only kind of smart cast that Kotlin can perform. We'll see this again in a future chapter when we cover advanced object and class concepts.

So, using a conditional in this way is like opening the box, and if there's something inside it, we order the coffee.



Otherwise, if there's nothing inside the box, we don't order the coffee.

Using a conditional to do a smart cast is just one way to convert something that's nullable to something that's non-nullable! Next, let's look at the elvis operator.



For the Nerds

In the code above, we used variables that were declared with the `val` keyword. If you try to do this kind of *smart cast* with a variable that's declared with the `var` keyword instead, you might be surprised to see that, in some cases, it won't work!

As you might remember from [Chapter 1](#), `var` makes it so that you can assign a new value to the variable, even after it's already been given one. If a variable like this is defined outside of the function where you're trying to do the smart cast, there's a possibility that its value could change between the time you check to see if it's null, and the time when you're trying to use the variable inside the `if` block. Particularly, this is a risk when you're running multiple threads of code at the same time. To prevent this problem, Kotlin will not do a smart cast in this case.

You can work around this by creating a new `val` variable inside your function, and assigning it the value of your `var` variable. We'll see other workarounds for this in [Chapter 11](#).

Using the Elvis Operator to Provide a Default Value

In the code above, we *only* ordered coffee when a value was present in the `payment` variable. It sure would be nice if we could order a coffee even when we don't have payment. For example, if our `payment` variable is null, maybe our friend can pay for us!

```
val payment: Payment? = null

if (payment != null) {
    val coffee = orderCoffee(payment)
} else {
    val coffee = orderCoffee(getPaymentFromFriend())
}
```

Listing 6.21 - If we don't have payment (that is, if `payment` is null), we get it from a friend instead.

This allows us to order coffee in either case. If `payment` actually has a value, we can use that. Otherwise, we call the `getPaymentFromFriend()` function, which returns a `Payment` value that we can use instead.

As we learned back in Chapter 3, instead of an [if statement](#) we can use an [if expression](#), which pulls the `coffee` variable outside of the `if` and `else` blocks. Let's make that small change to our code, in order to make it more concise.

```
val payment: Payment? = null

val coffee = if (payment != null) {
    orderCoffee(payment)
} else {
    orderCoffee(getPaymentFromFriend())
}
```

Listing 6.22 - Lifting the assignment of `coffee` outside of the `if` body. Compare to Listing 6.21.

We're also calling `orderCoffee()` in both branches, so let's pull that out of the `if` and `else` blocks, as well.

```
val payment: Payment? = null

val coffee = orderCoffee(if (payment != null) payment else getPaymentFromFriend())
```

Listing 6.23 - Lifting the call to `orderCoffee()` outside of the `if` body. Compare to Listing 6.22.

The code highlighted in Listing 6.23 is pretty common when dealing with nullable types: check whether a value is present... if so use that value, otherwise use some default value. To make this common expression easier, Kotlin gives us the **elvis operator** `?:`, so named because if you tip your head to the left and squint your eyes, it kind of looks like an emoticon of Elvis Presley's hair line above a pair of eyes. (You might have to use your imagination a little!)

Here's how we can use the elvis operator to make our code more concise:

```
val payment: Payment? = null

val coffee = orderCoffee(payment ?: getPaymentFromFriend())
```

Listing 6.24 - Replacing the `if` expression with an elvis expression.

This code works the same as the code in Listings 6.21, 6.22, and 6.23 - it's just shorter and easier to read. Using an elvis operator is like opening the box, and if there's something inside it, we use that.



Otherwise, if the box is empty, we get a value from somewhere else and use that instead.



Using the Not-Null Assertion Operator to Insist that a Value is Present

I hesitate to mention this one. It's dangerous, but in some rare cases, it can be a helpful option.

If you know for sure that a nullable variable will *definitely* have a value when your code is running, then you can use the **not-null assertion operator** `!!` to evaluate it to a non-nullable type. Here's how it would look when ordering coffee:

```
val payment: Payment? = Payment()  
val coffee = orderCoffee(payment!!)
```

Listing 6.25 - Using the not-null assertion operator to force a conversion from a nullable type to a non-nullable type.

The type of the `payment` variable is `Payment?`, which is nullable, but the type of the expression `payment!!` is `Payment`, which is non-nullable.

By putting `!!` after the variable name `payment`, it's like you're saying to Kotlin, "Trust me... when the code runs, `payment` will not be null!" If you're wrong about that - if the variable is indeed null, you'll get an error when the code runs:

```
val payment: Payment? = null  
val coffee = orderCoffee(payment!!) // ERROR AT RUNTIME!
```

Listing 6.26 - You can run this code, but you'll get an error when you do!

The not-null assertion operator is like reaching into the box, and if there's something inside, we use that...



Otherwise, if the box is empty...



This is why the not-null assertion operator is dangerous! In the other cases above - using a conditional to check for null, and using an elvis operator - it wasn't possible for us to get an error, because Kotlin's rules about nullable types wouldn't allow it. But here, we're foregoing that **null safety** and taking on risk that the variable might actually be null when the code is running.

Compile-Time and Runtime Errors

We can get errors during either *compile time* or *runtime*.

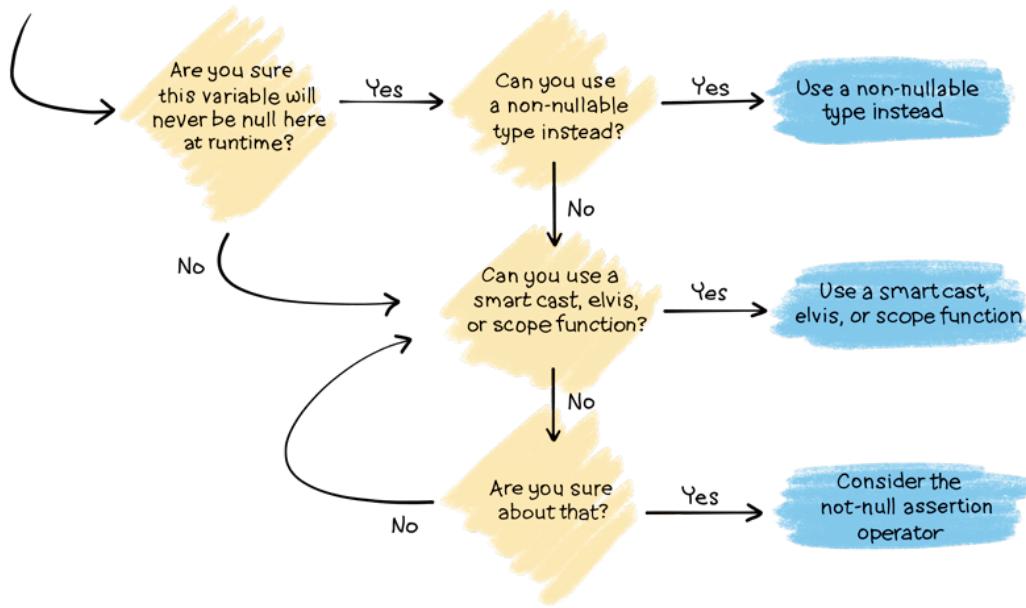
- Listing 6.18 shows an example of a compile-time error - the IDE highlights the problem while we're writing code.
- Listing 6.26 above shows code that will cause a runtime error, but unlike the compile-time error, there's no highlight to let us know that an error will happen.

As a general rule, an error during compile time is more helpful than an error during runtime, because we know about it sooner. In fact, Kotlin won't even let us run our code until we've fixed it! Runtime errors, on the other hand, are nefarious and they're often more difficult to hunt down.

When we use the *not-null assertion operator* `!!`, we're avoiding a compile-time error, but taking a risk that we could end up with a runtime error. If you're *certain* that the variable will not be null at runtime, then consider using a non-nullable type instead. If, for some reason, you can't do that, then the not-null assertion operator might be what you need. But use it only as a last resort!

The flow chart below gives some advice about when to consider using non-nullable types, smart casts, not-null assertion operators, and so on. It mentions scope functions, which are covered later, in Chapter 11.

WHEN SHOULD I USE THE NOT-NUL ASSERTION OPERATOR?



There's one more null-safety tool that Kotlin gives us. Let's check it out!

Using the Safe-Call Operator to Invoke Functions and Properties

Back in Chapter 4, we saw how objects have [functions](#) and [properties](#), and we can call those by using a dot character. For example, let's say our `Payment` class has a property that tells us what type of payment the customer is using, whether cash, a check, or a card:

```
enum class PaymentType {
    CASH, CHECK, CARD;
}

class Payment(
    val type: PaymentType = PaymentType.CASH
)
```

Listing 6.27 - Adding a `PaymentType` property to the `Payment` class.

In this case, when we get a `Payment`, we probably want to do something with it, like print out the `type`. Let's update the `orderCoffee()` function to do that.

```
fun orderCoffee(payment: Payment): Coffee {
    val paymentType = payment.type.name.toLowerCase()
    println("Thank you for supporting us with your $paymentType")
    return Coffee()
}
```

Listing 6.28 - Printing a message based on the type of payment used.

This works great when the `payment` parameter is a non-nullable `Payment` type. But when its type is a nullable `Payment?` type - as it was when James was running the charity - then we get a compile-time error:

```
fun orderCoffee(payment: Payment?): Coffee {
    val paymentType = payment.type.name.toLowerCase()
    println("Thank you for supporting us with your $paymentType")
    return Coffee()
}
```

Listing 6.29 - Error: "Only safe (?) or non-null asserted (!!.) calls are allowed on a nullable receiver of type Payment?"

Why is that?

Remember - a variable that has a nullable type - such as `Payment?` - is like a bucket that might be empty or might have a value... and we won't know whether it's empty until *runtime*! If the bucket is indeed empty while the code is running, then there would be no actual payment to get the `type` from.

In other words, if the `payment` isn't there, then neither is a payment `type`! It's not safe to get the type unless we know that the value of `payment` is present. Thankfully, Kotlin gives us a compile-time error, forcing us to deal with this fact.

In this chapter, we've learned a few tricks that could help us. For example, we could use an `if` to do a *smart cast*, like this:

```
fun orderCoffee(payment: Payment?): Coffee {
    val supportType = if (payment == null) {
        "encouragement"
    } else {
        payment.type.name.toLowerCase()
    }

    println("Thank you for supporting us with your $supportType")
    return Coffee()
}
```

Listing 6.30 - Using an if conditional to provide a supportType when payment is null

When this code runs, if `payment` is null, we'll print "Thank you for supporting us with your encouragement". Otherwise, the message will be based on the `type` of payment, such as "*Thank you for supporting us with your cash*".

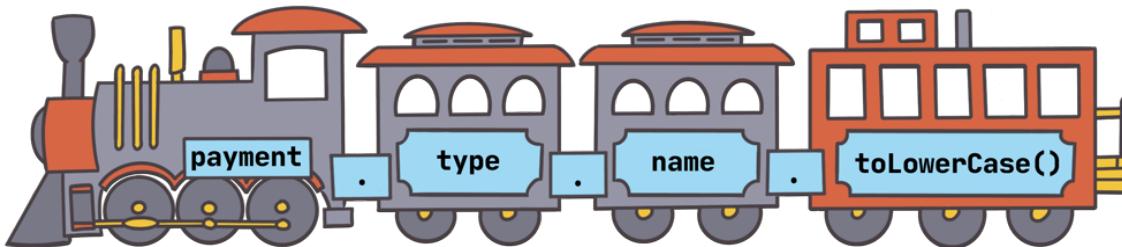
This certainly works, but it's a lot of code to write. Kotlin provides us with a **safe-call operator** `?.` that can do the same thing, just more concisely. We can use it along with the [elvis operator](#) to achieve the same thing as Listing 6.30, like this:

```
fun orderCoffee(payment: Payment?): Coffee {
    val supportType =
        payment?.type?.name?.toLowerCase() ?: "encouragement"
    println("Thank you for supporting us with your $supportType")
    return Coffee()
}
```

Listing 6.31 - Using a safe-call operator and an elvis operator instead of an if conditional.

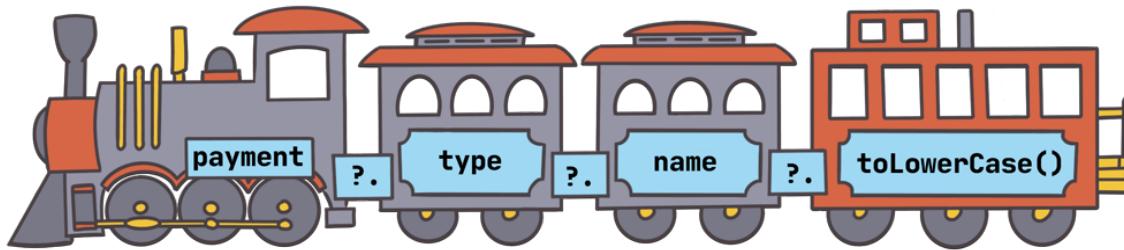
So, how does the safe-call operator work?

When we look at `payment.type.name.toLowerCase()`, it kind of looks like a train.¹

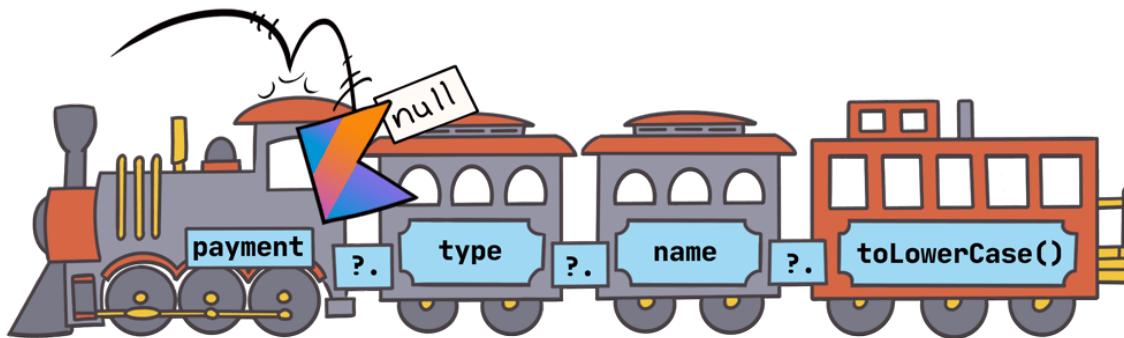


The main change in Listing 6.31 is that we replaced the train car connectors - where we previously used a dot `.`, we now use a safe-call operator `?.`:

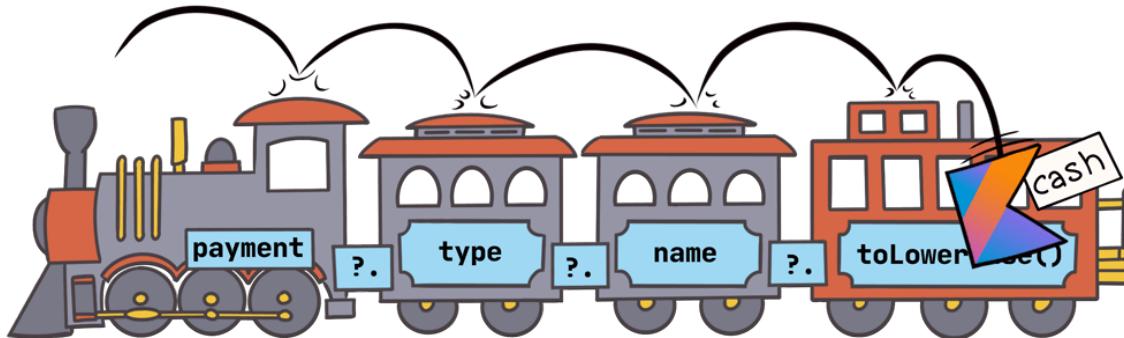
¹In fact, the term "train wreck" has been used to describe expressions like this that have many function or property calls chained together. In this book, I won't add commentary about the advantages or disadvantages of expressions like these. So, instead of calling it a train wreck, I'll just call it a train!



When Kotlin evaluates a “train” expression like this, you can imagine that it’s hopping from car to car, left to right. When the next connector is a safe-call operator `?.`, it asks, “does this car’s expression evaluate to null?” If so, then it hops off the train with a `null`.



Otherwise, it hops to the next car, repeating the process until it finds a null or jumps off the caboose with the final value.



Generally, when writing a train expression, if one of the cars has a nullable type, the *rest* of the train connectors *after* it will need to be safe-call operators rather than just dot operators. (I did say *generally* - we’ll see an exception to this when we get to extension functions in Chapter 10!)

For the Nerds

If you’re coming from a Java programming background, you might wonder how Kotlin’s nullable types compare to Java’s `Optional` type. Consider reading the article *Java Optionals and Kotlin Nulls*, which explains the similarities and differences:

<https://typealias.com/guides/java-optionals-and-kotlin-nulls/>

Summary

Congratulations! You've learned a ton in this chapter, including:

- The difference between [present and absent values](#).
- The difference between [nullable and non-nullable types](#).
- The difference between [compile time and runtime](#).
- How to use [conditionals to check for nulls](#).
- How to use the [elvis operator](#) to provide a default value.
- How to use the [not-null assertion operator](#) to insist that a value is present.
- How to use the [safe-call operator](#) to invoke functions and properties on a variable that's nullable.

Proper handling of nulls is an essential skill for every great Kotlin programmer. In the next chapter, we'll learn about another essential concept - lambdas.

Kotlin: An Illustrated Guide

Chapter 7

Lambdas and Function References



Hey, There's a Function in That Variable!

As we've seen, functions are a basic building block of Kotlin code. Throughout this series, we've written a lot of them, all using the `fun` keyword. Kotlin also gives us another way to write functions - lambdas!

To help us learn about function types, function references, and lambdas, we'll need to pay a visit to Bert's Snips & Clips.

Bert's Snips & Clips

Bert's Snips & Clips is the salon to go for a clean haircut and a nice smooth shave. Although he prides himself on his low prices, sometimes he offers coupons that give the customer \$5 off, to lower the price even further.

Once the customer has their fresh new haircut, he needs to calculate the *total cost*, so that he can charge the customer the right amount. Bert's pretty good at math, but to make this fast and easy, he created a simple Kotlin function to calculate the total.



His function needed to account for *tax* and the *five-dollars coupon*. Here's what he wrote:

```
// Tax is 9%, so we'll multiply by 109% to get the total with tax included.  
val taxMultiplier = 1.09  
  
fun calculateTotalWithFiveDollarDiscount(initialPrice: Double): Double {  
    val priceAfterDiscount = initialPrice - 5.0  
    val total = priceAfterDiscount * taxMultiplier  
  
    return total  
}
```

Listing 7.1 - A simple function to calculate the total cost when using a coupon.

If the customer has a \$20 haircut and presents a \$5 off coupon, he simply runs the function like this:

```
fun main() {  
    val total = calculateTotalWithFiveDollarDiscount(20.0)  
    println("%.2f".format(total))  
}
```

Listing 7.2 - Calling the function from Listing 7.1.

When he runs this code, it prints **\$16.35**.

For the Nerds

What's with `"%.2f".format()`?

This tells Kotlin to print the number with a dollar sign on the left, and two decimal places to the right. Because of the multiplication, `total` could end up with more than two decimal places, so this will do rounding for us.

By the way, although we're using `Double` in these examples, you should consider using a type with greater precision (e.g., `BigDecimal`) in any *real* code that deals with money.

One day, one of Bert's customers spent a *lot* of money on haircuts for the whole family, and she was

disappointed that they only had a *single* coupon for five dollars off. In order to reward his most loyal, high-paying customers, he decided to introduce a 10% off coupon. That way, the more money they spend at his salon, the more dollars would be discounted.



Alongside the original function, Bert created a second function, which accommodates the new percent-based coupon. To calculate the price after a 10% discount, he just multiplies the initial price by 90%.

```
fun calculateTotalWithFiveDollarDiscount(initialPrice: Double): Double {
    val priceAfterDiscount = initialPrice - 5.0
    val total = priceAfterDiscount * taxMultiplier

    return total
}

fun calculateTotalWithTenPercentDiscount(initialPrice: Double): Double {
    val priceAfterDiscount = initialPrice * 0.9
    val total = priceAfterDiscount * taxMultiplier

    return total
}
```

Listing 7.3 - One function for a \$5 coupon, and one for a 10% coupon.

When he looked at these functions, he realized that they're almost exactly the same, except for one small part - the part that *calculates the discounted price*.

```
fun calculateTotalWithFiveDollarDiscount(initialPrice: Double): Double {
    val priceAfterDiscount = initialPrice - 5.0
    val total = priceAfterDiscount * taxMultiplier

    return total
}

fun calculateTotalWithTenPercentDiscount(initialPrice: Double): Double {
    val priceAfterDiscount = initialPrice * 0.9
    val total = priceAfterDiscount * taxMultiplier

    return total
}
```

If only he could find a way to *pass code as an argument*... then he could consolidate his two functions down to one function that would look something like this:

```
fun calculateTotal(  
    initialPrice: Double,  
    applyDiscount: ???  
) : Double {  
    val priceAfterDiscount = applyDiscount(initialPrice)  
    val total = priceAfterDiscount * taxMultiplier  
  
    return total  
}
```

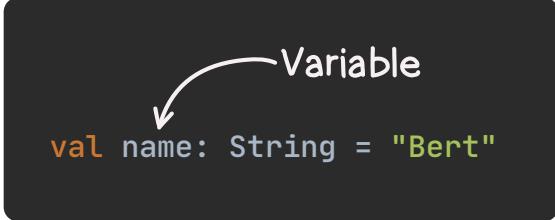
Listing 7.4 - Sketch of a function that might take a function as an argument. Doesn't actually compile.

In other words, he just wants to tell `calculateTotal()` to apply a discount *in a different way* each time he calls it. If he could pass a function as an argument to `calculateTotal()`, that would solve his problem. But passing a function as an argument to another function? How could that be possible?

With Kotlin's *function types*, it's easy!

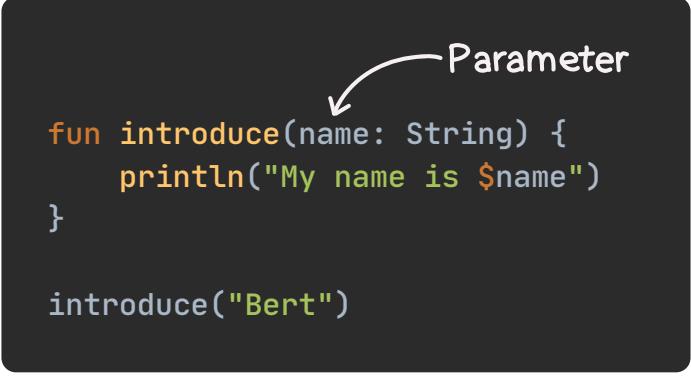
Introduction to Function Types

We saw way back in [Chapter 1](#) how we could create variables that hold *values*. For example, to create a simple `String` variable that holds text, we can do this:



```
val name: String = "Bert"
```

Similarly, [parameters](#) are variables in a function where the calling code passes in their values when [invoking](#) the function. For example:



```
fun introduce(name: String) {  
    println("My name is $name")  
}  
  
introduce("Bert")
```

So far, every time that we've assigned a variable or passed an argument to a function, we've been dealing with *values* of simple types like `String`, `Int`, and even more complex objects of our own types, like `Circle`.

In addition to assigning simple values like these, Kotlin also lets you assign *functions*. To demonstrate this, here's a simple function to calculate a flat dollar discount, like a \$5-off coupon:

```
fun discountFiveDollars(price: Double): Double = price - 5.0
```

Listing 7.5 - A simple function to take five dollars off of a price.

As you already know, you can call this function, like so:

```
val discountedPrice = discountFiveDollars(20.0) // Result is 15.0
```

Listing 7.6 - Calling the function from Listing 7.5.

In addition to *calling* this function, we can also *assign it to a variable* like this.

```
val applyDiscount = ::discountFiveDollars
```

Listing 7.7 - Assigning the function from Listing 7.5 to a variable called applyDiscount.

Notice the difference between these two lines of code. In Listing 7.6, we're assigning the *result* of a function call, but in Listing 7.7, we're assigning the *function itself*.

In the code above, `::discountFiveDollars` is a **function reference**. We call it that because it *refers* to a *function*. By assigning this function to a variable, it's kind of like giving `discountFiveDollars()` another name. Now we can call `applyDiscount()` the same way that we called `discountFiveDollars()`, and it does the same thing.¹

```
val discountedPrice = applyDiscount(20.0) // Result is 15.0
```

Listing 7.8 - Calling applyDiscount(), which will run the code from discountFiveDollars().

Whether we call `discountFiveDollars()` in Listing 7.6 above, or call `applyDiscount()` here, the result is the same: **15.0** (that is, \$15.00).

For the Nerds

You can write a function reference like `::discountFiveDollars` in many cases, but if you want a reference to a member function (that is, a function that's on an *object*), you'll need to qualify it further.

For example, if you've got an object named `couponForCustomer`, and it has a function named `discount()`, you can get a reference to it like this:

```
val applyDiscount = couponForCustomer::discount
```

As you might recall, every time that we declare a variable, that variable has a type, even if it's not explicitly written out in the code. For example:

¹ Note, however, that you cannot use named arguments when you call a function using the variable's name.

```
val name = "Bert"      // name's type is String
val hasCoupon = true   // hasCoupon's type is Boolean
val price = 12.50      // price's type is Double
```

Listing 7.9 - Examples of type inference.

So then, you might be wondering about the type of the `applyDiscount` variable.

When a variable holds a function, its type is a *combination* of all of its [parameter types](#) and its [return type](#). In the case of the `discountFiveDollars()` function, these include a single parameter of type `Double` and a return type of `Double`.

```
fun discountFiveDollars(price: Double): Double = price - 5.0
```

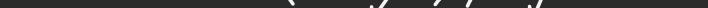
Parameter Type ↗ ↗ Return Type

A function's type can be assembled by:

1. Keeping its parentheses
 2. Keeping its types
 3. Converting the colon : to an arrow ->

Let's do that with `discountFiveDollars()`.

```
fun discountFiveDollars(price: Double): Double = price - 5.0
```



So, the type of `discountFiveDollars()` is `(Double) -> Double`. Knowing this, you can now write the type of `applyDiscount` explicitly:

```
val applyDiscount: (Double) -> Double = ::discountDollars
```

Listing 7.10 - Writing the type of applyDiscount explicitly rather than using type inference

All of the types that we've seen up until now have had no spaces in them, like `String`, `Int`, and `Double`. As you can see, **function types** like `(Double) -> Double` are a bit long, but they're easy to figure out! The parameter types go *inside the parentheses*, and the result type goes *to the right of the arrow*.

Parameter type(s) -> Result type to the right of the arrow
inside parentheses

For functions that have multiple parameters, separate the parameter types with a comma. For example, below is a function that has two parameters - a `String` and a `Double` - and it returns a `String`. When assigning this function to a variable, that variable's type will be:

```
(String, Double) -> String
```

```
fun menuItemDescription(name: String, price: Double) =
    "A $name costs $price before discounts and tax."
val describeMenuItem: (String, Double) -> String = ::menuItemDescription
```

Listing 7.11 - Demonstrates a function type that has more than one parameter.

Two functions of the same type

As you might recall, if you have *two values of the same type*, such as two `String` values, then you can assign (and reassign) those two values to the same variable.

```
var couponCode = "FIVE_BUCKS"
couponCode = "TAKE_10"
```

Listing 7.12 - Assigning and reassigning a simple value to a single variable.

Similarly, if you have *two functions of the same type* - that is, functions that have the same parameter types and return type - then you can assign (and reassign) those two functions to the same variable.

To demonstrate this, let's introduce a function to calculate the discount for a 10%-off coupon. Since it has the same parameter and result types as `discountFiveDollars()`, we can assign either of these functions to the same variable.

```
fun discountFiveDollars(price: Double): Double = price - 5.0
fun discountTenPercent(price: Double): Double = price * 0.9

var applyDiscount = ::discountFiveDollars
applyDiscount = ::discountTenPercent
```

Listing 7.13 - Assigning and reassigning a function to a single variable.

Note that the parameter *names* do not have to match. Just the types have to match. This code works just the same as the code above:

```
fun discountFiveDollars(initialPrice: Double): Double = initialPrice - 5.0
fun discountTenPercent(originalPrice: Double): Double = originalPrice * 0.9

var applyDiscount = ::discountFiveDollars
applyDiscount = ::discountTenPercent
```

Listing 7.14 - Assigning functions with different argument names to the same variable.

For functions that have multiple parameters, keep in mind that the parameters must match *in the same order*.

For example, here are two functions. Both return a String. Both accept two parameters - one String and one Double. However, since the order of those parameters doesn't match, the types of those functions don't match, so you can't assign them to the same variable, or else you get an error:

```
fun menuItemDescription(name: String, price: Double) =  
    "A $name costs $price before discounts and tax."  
  
fun sillyMenuItemDescription(price: Double, name: String) =  
    "You want a $name? It's gonna run you $price, not counting coupons, tax, and whatnot!"  
  
var describeMenuItem = ::menuItemDescription  
describeMenuItem = ::sillyMenuItemDescription
```

Listing 7.15 - Error: "Type Mismatch" (does not compile).

Well, we've successfully assigned a function to a variable. That can be helpful, but things get even more interesting when we assign a function to a *parameter*. In other words, we can pass a function as an argument to a function! This is exactly what Bert needed way back in Listing 7.4.

Passing Functions to Functions

Now that we know what function types are, let's update Bert's `calculateTotal()` function. We'll take Listing 7.4 above and make just a few small changes to it so that it accepts a parameter called `applyDiscount` that has a function type of `(Double) -> Double`.

```
fun calculateTotal(  
    initialPrice: Double,  
    applyDiscount: (Double) -> Double  
): Double {  
    // Apply coupon discount  
    val priceAfterDiscount = applyDiscount(initialPrice)  
    // Apply tax  
    val total = priceAfterDiscount * taxMultiplier  
  
    return total  
}
```

Listing 7.16 - A working version of the code in Listing 7.4.

With this code in place, Bert can call `calculateTotal()` with a function reference! Let's define a few more functions that match the type `(Double) -> Double`, and then call `calculateTotal()` with each of them:

```

fun discountFiveDollars(price: Double): Double = price - 5.0
fun discountTenPercent(price: Double): Double = price * 0.9
fun noDiscount(price: Double): Double = price

val withFiveDollarsOff = calculateTotal(20.0, ::discountFiveDollars) // $16.35
val withTenPercentOff = calculateTotal(20.0, ::discountTenPercent) // $19.62
val fullPrice        = calculateTotal(20.0, ::noDiscount)         // $21.80

```

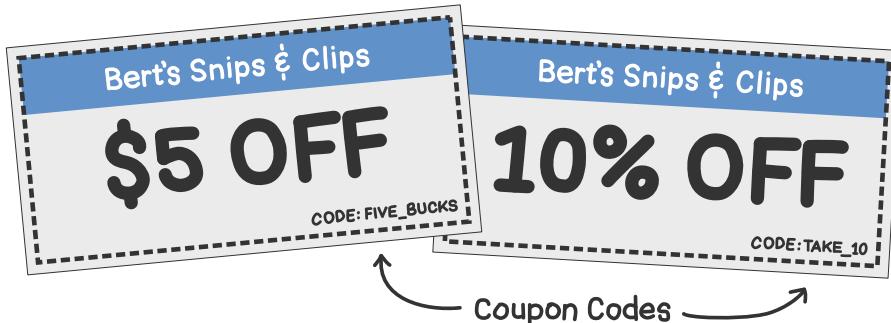
Listing 7.17 - Calling calculateTotal() with a variety of function references.

Yay! Bert is now able to calculate the total for different types of coupons, without needing to create multiple versions of `calculateTotal()` for each one!

In addition to sending a function *into* another function, you can also *return* a function from a function! Let's look at that next!

Returning Functions from Functions

Instead of typing in the *name of the function* each time he calls `calculateTotal()`, Bert would like to just enter the *coupon code* from the bottom of the coupon that he receives from the customer.



To do this, he just needs a function that accepts the coupon code and returns the right discount function. In other words, it'll have one parameter that's a `String`, and its return type will be `(Double) -> Double`.

A *when expression* makes this function easy!

```

fun discount(couponCode: String): (Double) -> Double = when (couponCode) {
    "FIVE_BUCKS" -> ::discountFiveDollars
    "TAKE_10"      -> ::discountTenPercent
    else            -> ::noDiscount
}

```

Listing 7.18 - Returning function references from a function.

And, of course, we can update Listing 7.17 to use the new `discountForCouponCode()` function.

```

val withFiveDollarsOff = calculateTotal(20.0, discount("FIVE_BUCKS")) // $16.35
val withTenPercentOff = calculateTotal(20.0, discount("TAKE_10"))     // $19.62
val fullPrice        = calculateTotal(20.0, discount("NONE"))        // $21.80

```

Listing 7.19 - Rewriting the second half of Listing 7.17 to use discount().

Functions like `calculateTotal()` and `discount()` above, which accept functions as arguments and/or return them as results, are called **higher-order functions**.

This is a higher-order function because it accepts a function.



```
fun calculateTotal(initialPrice: Double, applyDiscount: (Double) -> Double): Double {  
    // Apply coupon discount  
    val priceAfterDiscount = applyDiscount(initialPrice)  
    // Apply tax  
    val total = priceAfterDiscount * taxMultiplier  
  
    return total  
}
```

This is a higher-order function because it returns a function.



```
fun discountForCouponCode(couponCode: String): (Double) -> Double = when (couponCode) {  
    "FIVE_BUCKS" -> ::discountFiveDollars  
    "TAKE_TEN"    -> ::discountTenPercent  
    else           -> ::noDiscount  
}
```

So far, we've been able to achieve a lot with function references! They can be quite helpful when you've already written the function that you want to reference. But Kotlin also gives us another, more concise way to assign functions to variables and parameters - *lambdas*!

Introduction to Lambdas

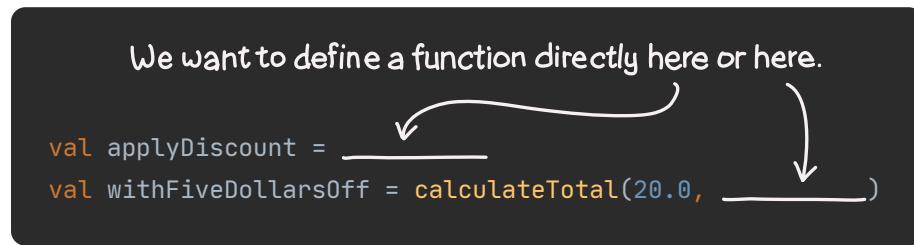
As you might recall from Chapter 1, a literal is when you *directly write out a value in your code*. For example, in Kotlin, you can create literals for basic types such as `String`, `Int`, and `Boolean`. The highlighted parts of the code below are values that are written as literals.

```
val string: String = "This is a string"  
val integer: Int = 49  
val boolean: Boolean = true
```

Listing 7.20 - Code that includes literals.

Just as you can write literals of `Strings`, `Integers`, and `Booleans`, you can also write a literal of a *function*!

"Wait," I can hear you say, "we've already been writing functions! How is this any different?" Yes, we've been writing named functions with the `fun` keyword, but we've never defined a function *directly in an expression*, such as on the right-hand side of an assignment, or directly inside a function call.



Let's take another look at the `discountFiveDollars()` function from earlier in this chapter. We defined that function and then assigned it to a variable by using a function reference. Here's what it looked like:

```

    fun discountFiveDollars(price: Double) = price - 5.0
    val applyDiscount: (Double) -> Double = ::discountFiveDollars
  
```

Listing 7.21 - Code from Listing 7.5 and 7.7 (with some changes to type inference).

Instead of defining the `discountFiveDollars()` function with the `fun` keyword, we can rewrite it as a **function literal** like this:

```

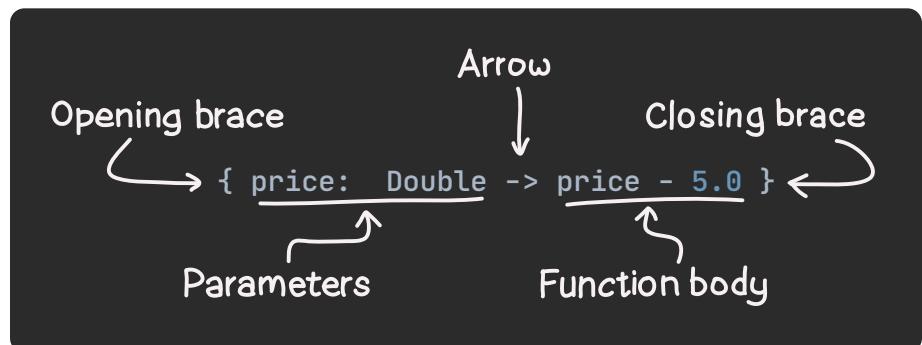
    val applyDiscount: (Double) -> Double = { price: Double -> price - 5.0 }
  
```

Listing 7.22 - Code that includes a function literal instead of a function reference.

The highlighted part of the code above is a function literal. In Kotlin, a function literal written like this is called a **lambda**.

Lambdas are functions, just like the ones we've written so far. They're simply expressed differently. To write a lambda:

- Use an opening brace `{` and a closing brace `}`.
- Write the parameters *before* the arrow `->` and the body *after* the arrow.



Once you've assigned a lambda to a variable, you can call it using the variable's name. Listing 7.21 and Listing 7.22 accomplish the same thing, but the latter does it more concisely.

Traditional Functions vs Lambdas

Both traditional functions and lambdas have *parameters* and a *body*, and evaluate to some kind of *result*. However, unlike traditional functions, the lambda itself does not have a name. Sure, you can choose to assign it to a *variable* that has a name, but the lambda itself is nameless.

For the Nerds

Anonymous Functions

Because lambdas have no name, many programming languages regard lambdas as *anonymous functions*. However, in Kotlin, the term **anonymous function** usually refers to *another* way of writing function literals. The second line below is an anonymous function:

```
val applyDiscount: (Double) -> Double =  
    fun(price: Double): Double { return price - 5.0 }
```

Anonymous functions in Kotlin are uniquely helpful for a few very particular cases. Lambdas are much more widely used and are more concise, so we'll focus on them in this book.

The lambda in Listing 7.22 indicates that the `price` parameter has a type of `Double`. Most of the time, however, Kotlin can use its [type inference](#) to figure it out. For example, we can rewrite that listing and omit the parameter's type in the lambda:

```
val applyDiscount: (Double) -> Double = { price -> price - 5.0 }
```

Listing 7.23 - Same as Listing 7.22, but using type inference on the lambda parameter.

Kotlin knows that `price` must be a `Double`, because that's what the type of `applyDiscount` says it must be. Similarly, the result of the lambda has to match.

```
val applyDiscount: (Double) -> Double = { price -> price - 5.0 }
```

So, lambdas are a concise way of creating a function right in the middle of an expression. Our lambda above is pretty small already, but we can make it even *more* concise!

The Implicit `it` parameter

In cases where there's only a *single parameter* for a lambda, you can *omit the parameter name and the arrow*. When you do this, Kotlin will automatically make the name of the parameter `it`. Let's rewrite our lambda to take advantage of this:

```
val applyDiscount: (Double) -> Double = { it - 5.0 }
```

Listing 7.24 - Rewriting Listing 7.23 to use the implicit `it` parameter.

The code here is incredibly more concise than the original `discountFiveDollars()` function!

The implicit `it` parameter is used often in Kotlin, especially when the lambda is small, like this one. In cases when the lambda is longer, as we'll see in a moment, it can be a good idea to give the parameter a name explicitly.

In future chapters, we'll also see cases where lambdas are nested inside other lambdas, which is another situation where explicit names are preferred. In many cases, though, the implicit `it` parameter can make your code easier to read.

Assigning a lambda to a variable can be helpful, but things get even more interesting when we start using lambdas with higher-order functions!

Lambdas and Higher-Order Functions

Passing Lambdas as Arguments

As we learned above, higher-order functions are those that have a function as an input (i.e., parameter) or an output (i.e., the result). Here's the code from Listing 7.16 and Listing 7.17 above, where we used function references to pass functions as arguments to the `calculateTotal()` function:

```
fun calculateTotal(
    initialPrice: Double,
    applyDiscount: (Double) -> Double
): Double {
    // Apply coupon discount
    val priceAfterDiscount = applyDiscount(initialPrice)
    // Apply tax
    val total = priceAfterDiscount * taxMultiplier

    return total
}

fun discountFiveDollars(price: Double): Double = price - 5.0
fun discountTenPercent(price: Double): Double = price * 0.9
fun noDiscount(price: Double): Double = price

val withFiveDollarsOff = calculateTotal(20.0, ::discountFiveDollars) // $16.35
val withTenPercentOff = calculateTotal(20.0, ::discountTenPercent) // $19.62
val fullPrice        = calculateTotal(20.0, ::noDiscount)         // $21.80
```

Listing 7.25 - Code from Listing 7.16 and 7.17.

It's easy to call `calculateTotal()` with a lambda instead of a function reference. Let's rewrite the last few lines of the code above to use lambdas. We'll just take the body from each corresponding function and write it as a lambda instead:

```
val withFiveDollarsOff = calculateTotal(20.0, { price -> price - 5.0 }) // $16.35
val withTenPercentOff = calculateTotal(20.0, { price -> price * 0.9 }) // $19.62
val fullPrice        = calculateTotal(20.0, { price -> price })         // $21.80
```

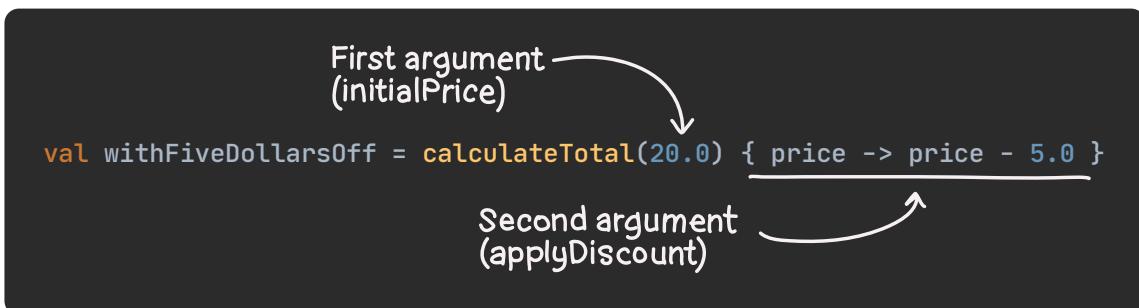
Listing 7.26 - Replacing the function references from Listing 7.25 with lambdas.

In cases where function's *last parameter* is a function type, you can move the lambda argument *outside* of the parentheses to the right, like this:

```
val withFiveDollarsOff = calculateTotal(20.0) { price -> price - 5.0 } // $16.35
val withTenPercentOff = calculateTotal(20.0) { price -> price * 0.9 } // $19.62
val fullPrice        = calculateTotal(20.0) { price -> price }        // $21.80
```

Listing 7.27 - Placing a lambda argument outside of the parentheses.

We're still sending *two* arguments to `calculateTotal()`. The first is *inside* the parentheses, and the second is *outside* to the right.



In Kotlin, writing the lambda outside of the parentheses like this is called **trailing lambda syntax**. Regardless of whether you put that last lambda argument inside the parentheses or outside, it works exactly the same. Kotlin developers usually prefer trailing lambdas, though.

Trailing lambda syntax is even more fun when the lambda is the *only* argument that you're passing to the function, because then you can *omit the parentheses completely!*

For example, here's a higher-order function with a single parameter, which has a function type:

```
fun printSubtotal(applyDiscount: (Double) -> Double) {
    val result = applyDiscount(20.0)
    val formatted = "$%.2f".format(result)
    println("A $20.00 haircut will cost you $formatted before tax.")
}
```

Listing 7.28 - A higher-order function with a single parameter, which is of a function type.

When calling `printSubtotal()`, no parentheses are needed!

```
printSubtotal { price -> price - 5.0 }
printSubtotal { price -> price * 0.9 }
```

Listing 7.29 - Calling printSubtotal() without parentheses.

In addition to using lambdas as arguments, we can also use them as function results!

Returning Lambdas as Function Results

Here's the code from Listing 7.18 above, where we returned function references:

```
fun discount(couponCode: String): (Double) -> Double = when (couponCode) {  
    "FIVE_BUCKS" -> ::discountFiveDollars  
    "TAKE_10"      -> ::discountTenPercent  
    else            -> ::noDiscount  
}
```

Listing 7.30 - Returning function references from a function. (Same as Listing 7.18).

We can very easily replace these function references with lambdas, just as we did for the function arguments in Listing 7.27.

```
fun discount(couponCode: String): (Double) -> Double = when (couponCode) {  
    "FIVE_BUCKS" -> { price -> price - 5.0 }  
    "TAKE_10"      -> { price -> price * 0.9 }  
    else            -> { price -> price }  
}
```

Listing 7.31 - Returning lambdas from a function.

Lambdas with Multiple Statements

So far, the lambdas that we've used have contained *only one simple expression* each.

Sometimes you need a lambda that has *multiple statements* in it. To do this, simply put each statement on a separate line, as you would inside any other function. Unlike a regular function, though, you won't use the return keyword to return your result. Instead, the very last line of the lambda will be the result of the call.

For example, we might want to print some debugging information inside our lambda that calculates the five-dollars-off coupon:

```
val withFiveDollarsOff = calculateTotal(20.0) { price ->  
    val result = price - 5.0  
    println("Initial price: $price")  
    println("Discounted price: $result")  
    result  
}
```

Listing 7.32 - A lambda that has multiple lines of code.

When we've got a lambda that spans multiple lines like this, it's conventional to put the parameters and arrow on the same line as the opening brace, as seen above.

Here's the same code, with some callouts indicating each part.

```
val withFiveDollarsOff = calculateTotal(20.0) { price -> Parameter and arrow
    val discountedPrice = price - 5.0
    println("Initial price: $price")
    println("Discounted price: $discountedPrice") } Multiple statements
    discountedPrice ← Result (no "return" keyword!)
}
```

Before we wrap up this chapter, we've got one more concept to cover!

Closures

Bert's salon is doing great now. Let's take a look at his code, including `calculateTotal()`, `discountForCouponCode()`, and how he ends up calling them to get the total.

```
fun calculateTotal(
    initialPrice: Double,
    applyDiscount: (Double) -> Double
): Double {
    val priceAfterDiscount = applyDiscount(initialPrice) // Apply coupon discount
    val total = priceAfterDiscount * taxMultiplier // Apply tax

    return total
}

fun discountForCouponCode(couponCode: String): (Double) -> Double = when (couponCode) {
    "FIVE_BUCKS" -> { price -> price - 5.0 }
    "TAKE_10"      -> { price -> price * 0.9 }
    else            -> { price -> price }
}

val initialPrice = 20.0
val couponDiscount = discountForCouponCode("FIVE_BUCKS")
val total = calculateTotal(initialPrice, couponDiscount)
```

Listing 7.33 - Putting it all together.

Bert noticed that when he introduces a new coupon, he needs to write another lambda. For example, if he adds a new coupon for nine dollars off, and another one for fifteen percent off, he would need to write a few more lambdas, like this:

```
fun discount(couponCode: String): (Double) -> Double = when (couponCode) {
    "FIVE_BUCKS" -> { price -> price - 5.0 }
    "NINE_BUCKS" -> { price -> price - 9.0 }
    "TAKE_10" -> { price -> price * 0.9 }
    "TAKE_15" -> { price -> price * 0.85 }
    else -> { price -> price }
}
```

Listing 7.34 - Adding more coupon codes.

That's not too bad, actually, but he decided he could make one last small improvement. There are really two main categories of coupons - dollar amount and percentages.

Dollar Amount Coupons



Percentage Coupons



He wrote these two functions to match the two categories of coupons that he identified:

```
fun dollarAmountDiscount(dollarsOff: Double): (Double) -> Double =
    { price -> price - dollarsOff }

fun percentageDiscount(percentageOff: Double): (Double) -> Double {
    val multiplier = 1.0 - percentageOff
    return { price -> price * multiplier }
}
```

Listing 7.35 - Functions that create functions.

It's important to note that these two functions *do not calculate the discount themselves*. Instead, they *create functions* that calculate the discount. This is a little easier to see in `percentDiscount()`, where we're using an explicit `return` keyword rather than an expression body.

Another neat thing here is that these lambdas use variables that are *defined outside of the lambda body*. The first one uses the `amount` variable (a parameter of the wrapping function), and the second uses the `multiplier`

variable. When a lambda uses a variable that's defined *outside of its body* like this, it's sometimes referred to as a **closure**.

Now, creating a new coupon is easy. Instead of writing the lambda inline in `discount()`, Bert can just call either `amountDiscount()` or `percentDiscount()` to create the lambda for him.

```
fun discount(couponCode: String): (Double) -> Double = when (couponCode) {  
    "FIVE_BUCKS" -> dollarAmountDiscount(5.0)  
    "NINE_BUCKS" -> dollarAmountDiscount(9.0)  
    "TAKE_10"      -> percentageDiscount(0.10)  
    "TAKE_15"      -> percentageDiscount(0.15)  
    else            -> { price -> price }  
}
```

Listing 7.36 - A small improvement to avoid some duplicated code.

Summary

Congratulations! Lambdas can be a tough concept for many programmers who haven't used them before. If you still feel a little unsure about them, it's completely fine. We'll use them a lot in upcoming chapters, and you'll get more familiar with them as you go.

In this chapter, you learned about:

- [Function types](#), like `(Int, Int) -> Int`.
- [Function references](#), which let you assign existing functions to variables and parameters.
- [Lambdas](#), which are literals for functions.
- [Higher-order functions](#), which are functions that accept a function as an argument, or return one as a result.
- [The implicit it parameter](#), which can be used when a lambda has a single parameter.
- [Multiple-line lambdas](#), which can be useful when your lambda needs more than just a single expression.

Throughout this chapter, we've seen some simple use cases for lambdas, but they *really* shine when used with *collections*. In the next chapter, we'll introduce collections, and we'll see how we can use lambdas to do all sorts of fun things with them!

Kotlin: An Illustrated Guide

Chapter 8

Collections: Lists and Sets



Who Loves to Read Books?

So far, we've only worked with variables as individual values. Writing Kotlin becomes so much more interesting once we start putting variables together in a way that we can work on them as a collection.

To learn about collections, let's visit Libby, a bright young lady who's always got a book nearby!

Libby is a voracious reader. She's always on the lookout for a great novel, so whenever someone tells her about a good book, she jots down the title on a list that she keeps on a sheet of paper. Here are the titles that are currently on her list:

Books to Read
Tea with Agatha
Mystery on First Avenue
The Ravine of Sorrows
Among the Aliens
The Kingsford Manor Mystery

Libby has been learning to write Kotlin code in her spare time, so she also wanted to write this same list in Kotlin, and print all of the titles to the screen. Here's what she wrote:

```
val book1 = "Tea with Agatha"
val book2 = "Mystery on First Avenue"
val book3 = "The Ravine of Sorrows"
val book4 = "Among the Aliens"
val book5 = "The Kingsford Manor Mystery"

println(book1)
println(book2)
println(book3)
println(book4)
println(book5)
```

Listing 8.1 - Creating multiple strings to represent titles of books.

"Hmm...", she thought. "Every time I add a new book, I have to create a new variable. And keeping the numbers in order could be hard - if I remove `book3` from the middle of the list, then `book4` and `book5` would need to be renamed to `book3` and `book4`. If only there were a better way to keep track of my list of book titles..."

Introduction to Lists

Thankfully, there's a *much* better way! Libby can create a **collection**. Let's start with one of the most common kinds of collections in Kotlin - a **list**. Creating a list is easy - just call `ListOf()` with the values that you want, separating them with commas. Let's update Libby's code so that it's using a list.

```
val booksToRead = listOf(
    "Tea with Agatha",
    "Mystery on First Avenue",
    "The Ravine of Sorrows",
    "Among the Aliens",
    "The Kingsford Manor Mystery",
)
```

Listing 8.2 - Creating a list of strings to represent titles of books.

This code looks pretty similar to Libby's handwritten list. In fact, let's compare the two!



The handwritten list and the Kotlin list have a lot in common:

1. First, they both have a *name*. In Kotlin, the name of the variable holding the list is kind of like the name of the list on paper.
2. Next, both lists have *items* in them - in this case, the titles of the books. In Kotlin, the items in a list are called **elements**.
3. Finally, both lists have the titles in a particular *order*.

In the past, we've used the `println()` function to print out the contents of a variable to the screen. You can also use `println()` with a collection variable.

```
println(booksToRead)
```

Listing 8.3 - Printing a collection variable to the screen.

When you do this, you'll see its elements in order, like this:

```
[Tea with Agatha, Mystery on First Avenue, The Ravine of Sorrows, Among the Aliens,
The Kingsford Manor Mystery]
```

Collections and Types

When working with collections in Kotlin, we have *two different types* to consider.

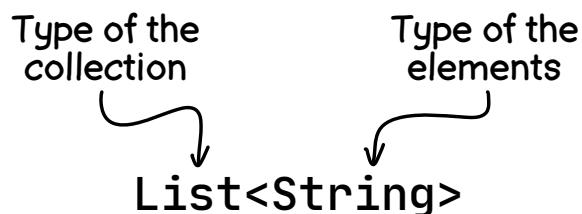
1. The type of the *collection* we're using.
2. The type of the *elements* in the collection.

These two things together determine the overall type of the collection variable. In the case of Listing 8.2:

1. The collection is a **List**.
2. The type of the elements in the collection is **String**.

```
val booksToRead = listOf(  
    "Tea with Agatha",  
    "Mystery on First Avenue",  
    "The Ravine of Sorrows",  
    "Among the Aliens",  
    "The Kingsford Manor Mystery",  
)
```

Once you know these two things, you can write the type of a collection variable easily. Just put the type of the collection first, then write the type of the elements second, between a left angle bracket < and a right angle bracket >. So the type of `booksToRead` is `List<String>`.



Let's rewrite Listing 8.2, this time explicitly including the type information for the `booksToRead` collection variable.

```
val booksToRead: List<String> = listOf(  
    "Tea with Agatha",  
    "Mystery on First Avenue",  
    "The Ravine of Sorrows",  
    "Among the Aliens",  
    "The Kingsford Manor Mystery",  
)
```

Listing 8.4 - Explicitly including the type of a collection variable.

This kind of a type is an instance of a **generic**. We will cover generics in detail in a future chapter. For now, just know how to write the type for a list, in case you need to use it as a function's parameter type or return type.

Adding and Removing an Element

Libby just heard about another great book from her friend, Rebecca! She's ready to add this new title, *Beyond the Expanse*, to her list. How can she do this?

Of course, she could just add one more argument to the end of `listOf()`. But what about adding the title *after* the list has already been created?

In Kotlin, once you've already called `listOf()` to create a list, that list can't be changed. You can't add anything to it, and you can't remove anything from it. The fancy word for "change" in programming is **mutate**, so a list that doesn't allow you to add or remove elements is called an **immutable list**.

Even though you can't add or remove elements from a regular Kotlin `List`, you can *create a new list* by putting the original list together with a new element. To do this, use the **plus operator**. That is, use `+` to connect the original list with the new item, and assign it to a new variable, like this:

```
val booksToRead = listOf(
    "Tea with Agatha",
    "Mystery on First Avenue",
    "The Ravine of Sorrows",
    "Among the Aliens",
    "The Kingsford Manor Mystery",
)

val newBooksToRead = booksToRead + "Beyond the Expanse"
```

Listing 8.5 - Creating a new list that combines a list with one new element.

In this code, `booksToRead + "Beyond the Expanse"` is an [expression](#) that evaluates to a new `List` instance. So, by the time this code is done running, we have *two collection variables* - `booksToRead` and `newBooksToRead`.

This is kind of writing the new list of titles on a second sheet of paper. That way, Libby actually ends up with *two lists* - the original list and the new list:

Books to Read	New Books to Read
Tea with Agatha	Tea with Agatha
Mystery on First Avenue	Mystery on First Avenue
The Ravine of Sorrows	The Ravine of Sorrows
Among the Aliens	Among the Aliens
The Kingsford Manor Mystery	The Kingsford Manor Mystery
	Beyond the Expanse

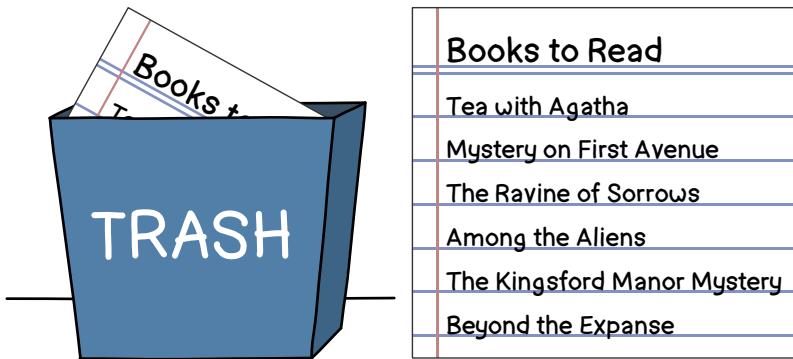
As you remember from [Chapter 1](#), a variable can be declared with either `val` or `var`, and this includes variables that hold a collection. Keep in mind, though, that declaring a collection variable with `var` does not change the fact that *the list itself is immutable*. In other words, just declaring it with `var` does not make it so that you can add or remove elements. However, `var` does let you assign *another* immutable list to it.

So, by changing the `booksToRead` variable from `val` to `var`, the new list can be assigned to the existing variable name, like this:

```
var booksToRead = listOf(  
    "Tea with Agatha",  
    "Mystery on First Avenue",  
    "The Ravine of Sorrows",  
    "Among the Aliens",  
    "The Kingsford Manor Mystery",  
)  
  
booksToRead = booksToRead + "Beyond the Expanse"
```

Listing 8.6 - Reassigning the collection to the existing booksToRead variable.

This is kind of like trashing the old paper list, and simply giving the new list the same name as the old one.



Libby's list now has six titles in it. Just when she thought she was done updating the list, she heard from Rebecca again. "You know, I read *Among the Aliens* last week. It really wasn't very good," she said. "You shouldn't bother reading that one."

Libby would like to scratch that one off her list. As you can guess, you can remove an element from a list in a similar way, but instead of the plus operator, use the **minus operator**.

```
var booksToRead = listOf(  
    "Tea with Agatha",  
    "Mystery on First Avenue",  
    "The Ravine of Sorrows",  
    "Among the Aliens",  
    "The Kingsford Manor Mystery",  
)  
  
booksToRead = booksToRead + "Beyond the Expanse"  
booksToRead = booksToRead - "Among the Aliens"
```

Listing 8.7 - Adding and removing an element from a collection.

In fact, those last two lines can be consolidated, like this:

```
booksToRead = booksToRead + "Beyond the Expanse" - "Among the Aliens"
```

Listing 8.8 - Adding and removing an element from a collection with one assignment.

Now, when Libby does `println(booksToRead)`, she sees this on the screen:

```
[Tea with Agatha, Mystery on First Avenue, The Ravine of Sorrows,
The Kingsford Manor Mystery, Beyond the Expanse]
```

"Excellent!" she thought, "My reading list is all up to date!"

List and MutableList

So far, we've been using a regular Kotlin `List`, which doesn't allow changes to it, as we saw above. Instead, we had to create a new list by using a plus or minus operator.

However, Kotlin also provides another kind of list - one that *does* allow you to change it. Since these lists do allow changes, they're called **mutable lists**, and they have a type of `MutableList`.

When using a mutable list, you can use the `add()` and `remove()` functions to add or remove elements, like this:

```
var booksToRead: MutableList<String> = mutableListOf(
    "Tea with Agatha",
    "Mystery on First Avenue",
    "The Ravine of Sorrows",
    "Among the Aliens",
    "The Kingsford Manor Mystery",
)

booksToRead.add("Beyond the Expanse")
booksToRead.remove("Among the Aliens")
```

Listing 8.9 - Creating, adding to, and removing from a mutable list.

Using a mutable list is kind of like Libby writing down her paper list with a pencil instead of a pen. She can go in and erase a title, or add another, without using another sheet of paper.

Libby is ready to start reading those books! But in order know which book to start with, she needs to know how to get a single title out of the list.

Books to Read

Tea with Agatha

Mystery on First Avenue

The Ravine of Sorrows

~~Among the Aliens~~

The Kingsford Manor Mystery

Beyond the Expanse

Heads Up: Using Plus and Minus Operators with a Mutable List

For what it's worth, you can also use the plus and minus operators on a mutable list, but keep in mind that those expressions evaluate to a regular immutable `List`, not `MutableList`. So generally, if you're using a `List`, use the plus and minus operators, and if you're using a `MutableList`, use the `add()` and `remove()` functions.

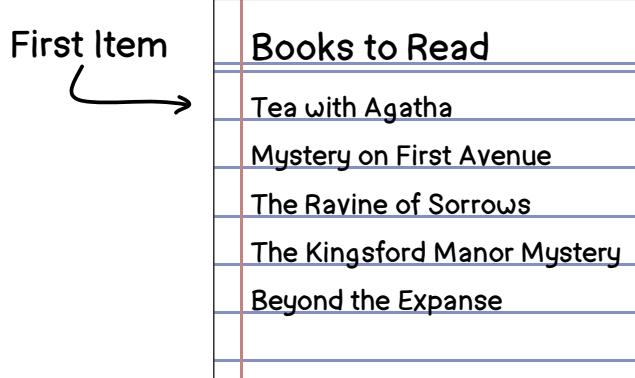
Best Practice: List vs Mutable List

`MutableList` can be convenient, and if you need to add or remove lots of elements, it can also be faster than a regular `List`. However, similar to the trade-offs of using `val` versus `var`, it's often better to stick with things that can't change, because when something can't change, you have guarantees that you wouldn't have otherwise. Sometimes this guarantee is very helpful, and other times it might not be as important. Use whatever is most appropriate for each situation, but by default, go with a regular `List`.

Getting an Element from a List

"All right, which title is first on my list?" wondered Libby. She glanced down at her handwritten page. It was easy for her to see which one was first. "Tea with Agatha," she noted. "Now how do I get the first title from the list in Kotlin?"

As mentioned earlier, the elements of a list are in a particular order, and that order is very important for getting an individual element out of the list. Here's how it works:



Each element in the list is given a number, called an **index**, based on where it is in the list. The first element has an index of 0, the second has an index of 1, the third has an index of 2, and so on.

```
    val booksToRead = listOf(  
        0 ..... "Tea with Agatha",  
        1 ..... "Mystery on First Avenue",  
        2 ..... "The Ravine of Sorrows",  
        3 ..... "The Kingsford Manor Mystery",  
        4 ..... "Beyond the Expanse"  
    )
```

It's easy to get an element out of a list once you know its index. Just call the `get()` member function, passing the index as the argument. For example, Libby can get the *first* element out of the list by calling `get(0)` like this:

```
val booksToRead = listOf(  
    "Tea with Agatha",  
    "Mystery on First Avenue",  
    "The Ravine of Sorrows",  
    "The Kingsford Manor Mystery",  
    "Beyond the Expanse"  
)  
  
val firstBook = booksToRead.get(0)  
println(firstBook) // Tea with Agatha
```

Listing 8.10 - Using the `get()` function to get a single element out of a collection.

"Great!" said Libby. "Now I can easily get a single title out of the list of books!"

Heads Up: Out-of-Bound Indexes

When you call `get()`, make sure that the index you give it is for an element that actually exists! In `booksToRead`, there are five elements, so you could call `get()` with any number including and between 0 and 4. If you were to call `get(86)`, for example, you would see an error message that says that the index was out of bounds.

For the Nerds: Why Do Indexes Start with Zero?

Kotlin follows the tradition of many other programming languages in that its indexes start with the number 0 instead of 1. One classic data structure for representing a list of values is called an array. In older programming languages, you had to manage the location of your variables and arrays in the computer's memory. A single element of an array took up a certain amount of memory - for example, an integer took up one byte of memory - and an array required a contiguous block of memory. So, if you had an array of integers, they were all physically next to each other in the computer's memory.

If you knew the starting address of the array - that is, the memory address of the first element in the array - you could get any other particular element's memory address by multiplying its index by the size of the individual elements, then adding that to the starting address.

Kotlin doesn't require its developers to manage memory this way, but it does have a few different kinds of lists, each using a different kind of data structure to hold the elements for you behind the scenes. Regardless of whether a particular list uses an array to hold its data, the indexes still always start with zero.

Rather than calling the `get()` function directly, you can use the **indexed access operator** instead, which is written with an opening bracket [and a closing bracket], with the index in the middle. The code in the following listing does the exact same thing as the code above.

```
val firstBook = booksToRead[0]
println(firstBook) // Tea with Agatha
```

Listing 8.11 - Getting a single element out of a collection, using the indexed access operator.

Kotlin developers use the *indexed access operator* much more than they use the `get()` function, so we'll be using it from now on.

Now, getting an individual item out of the list can be helpful, but collections become *especially* helpful when we want to *do something with each item in the list*. Let's see how to do that next!

Loops and Iterations

"Now, I'd like to print out the list of books to the screen," said Libby to herself. "I'll use `println(booksToRead)` for this!" Upon running that code, here's what she saw:

```
[Tea with Agatha, Mystery on First Avenue, The Ravine of Sorrows,
The Kingsford Manor Mystery, Beyond the Expanse]
```

"It's nice that I can print out the list so easily, but I'd really like to see the list *vertically*, like my handwritten list." Here's what she has in mind:

```
Tea with Agatha
Mystery on First Avenue
The Ravine of Sorrows
The Kingsford Manor Mystery
Beyond the Expanse
```

Of course, to achieve this, she could call `println()` on each element one by one, like this:

```
println(booksToRead[0])
println(booksToRead[1])
println(booksToRead[2])
println(booksToRead[3])
println(booksToRead[4])
```

Listing 8.12 - Printing each element by hand.

However, writing code like this is quite tedious. Plus, it would be easy to make a mistake by printing the elements out of order, or by accidentally printing the same element more than once. In fact, this looks a whole lot like the code back in Listing 8.1!

Instead of writing out the same code for each element in the list, what if Kotlin could just go through every element, one by one, and call `println()` on each? Thankfully, this is very easy to do in Kotlin! We can use the `forEach()` function. Here's how it looks.

```
booksToRead.forEach { element ->
    println(element)
}
```

Listing 8.13 - Using `forEach()` to print each element.

When Kotlin is running this code, it runs down through `println(element)` for the first element, then comes back up and runs down it again for the second element, then comes back up and runs down it again for the third element, and so on. By going through this line of code over and over again, it's as if it's looping in circles, like this:



That's why programming languages call this a **loop** - because, for each element in the collection, it cycles back through that code. It's also generally called **iterating**, and each time the code is run, it's called a single **iteration**.

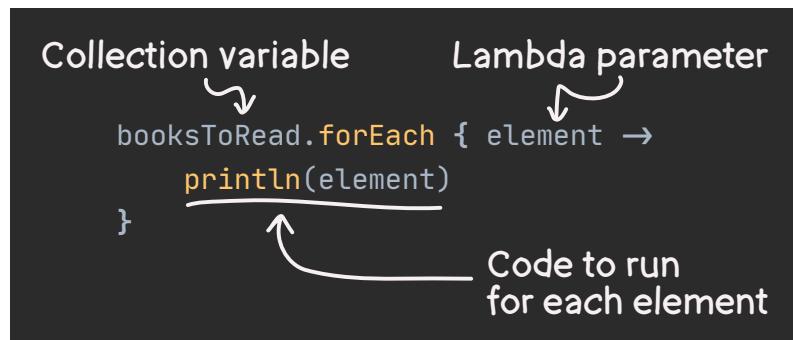
For the Nerds: Using the Term "Loop"

In a lot of the programming texts out there - including the Kotlin documentation - the term "loop" tends to be reserved for traditional programming constructs like `for` or `while`, but not used when referring to collection functions like `forEach()`. Although Kotlin does support these [traditional looping constructs](#), in this book, we'll stick with `forEach()` and similar functions, which we can put together to do all sorts of cool things, as we'll see later in this chapter.

Since we won't need to distinguish between traditional looping constructs and collection functions, and since those collection functions ultimately use a traditional `for` loop under the hood anyway, I'll be more liberal with my use of the term "loop."

Let's look a little closer at `forEach()`, to understand why we had to structure the code the way we did.

`forEach()` is a member function that exists on collection variables. It's a [higher-order function](#) that accepts a [lambda](#). That lambda is the code that you want Kotlin to run "for each" element in the collection.



Here we named the parameter `element`, but you could have named it `title` instead. Alternatively, since this lambda has only a single parameter, you can use the [implicit it parameter](#) instead, which makes it nice and concise. In fact, we can put it all on one line:

```
booksToRead.forEach { println(it) }
```

Listing 8.14 - Using the implicit it parameter with forEach().

The result in either case is exactly what Libby wanted - the book titles are printed out vertically, just like on her paper notepad!

```
Tea with Agatha  
Mystery on First Avenue  
The Ravine of Sorrows  
The Kingsford Manor Mystery  
Beyond the Expanse
```

By the way...

Instead of passing a lambda to `forEach()`, you can also pass a [function reference](#), as long as the function that you refer to has the right function type - in our case, it must have a single parameter that accepts a `String`. Since the `println()` function has a single parameter that accepts a `String`, we could have achieved the same thing as Listing 8.13 and Listing 8.14 by simply doing this:

```
booksToRead.forEach(::println)
```

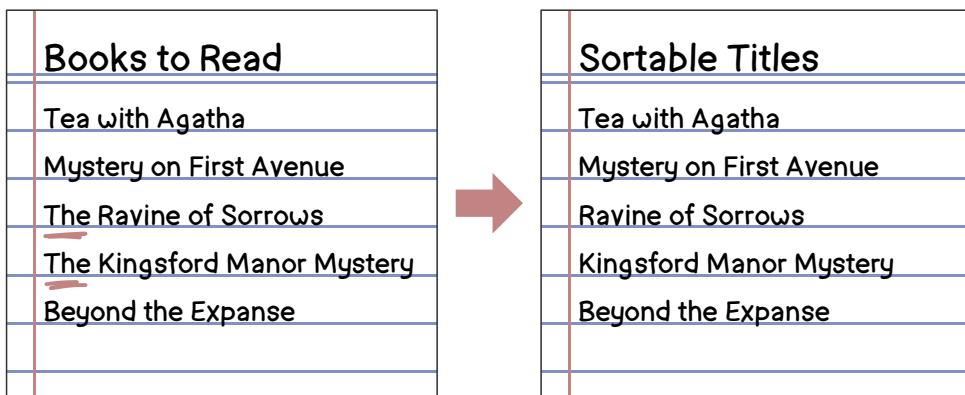
Collection Operations

Libby is ready to share her list of books with other people who are interested in what she'll be reading, starting with her friend Nolan. However, when she makes a copy of the list for him, she wants to make changes to some of the titles.

"I'd really like to remove the word 'The' from the beginning of each title," thought Libby. "That way, I'll be able to sort it alphabetically, and the titles that begin with 'The' won't clump together."

Mapping Collections: Converting Elements

Sometimes when you create a new collection from an existing one, you also want to convert one or more of the elements in some way. In Libby's case, she wants to remove the word "The" when it appears at the beginning of a title, so that they could be used for sorting.



Before doing that conversion on *all* of the titles, let's start with just one of them. **String** objects have a `removePrefix()` function, which you can use to remove words from the beginning of the string. Here's how you can use it:

```
val sortableTitle = "The Kingsford Manor Mystery".removePrefix("The ")
println(sortableTitle) // Kingsford Manor Mystery
```

Listing 8.15 - Using `removePrefix()` on a single string.

Perfect! Now all she needs is to apply this `removePrefix()` function to *each element in the list!*

"Maybe I can use `forEach()`, since I know it operates on each item in the list," thought Libby. She rolled up her sleeves, and cranked out the following code:

```
val sortableTitles: MutableList<String> = mutableListOf()

booksToRead.forEach { title ->
    sortableTitles.add(title.removePrefix("The "))
}

sortableTitles.forEach { println(it) }
```

Listing 8.16 - Manually creating a list without the word "The" at the beginning of the titles.

"Well, that works," thought Libby. "But it's a little complicated, and it's a lot of code to write..."

The reason this is complicated is that Libby wanted to create a new collection, but `forEach()` doesn't do that. It simply runs the lambda on an *existing* collection, and then returns [Unit](#).

What she really needs is a collection operation that runs the lambda and *includes the result of that lambda as an element in a new collection*.

In Kotlin, that collection operation is called `map()`. Here's how Libby can use it to remove the word "The" from the beginning of titles in the new collection:

```
val sortableTitles = booksToRead.map { title ->
    title.removePrefix("The ")
}
```

Listing 8.17 - Using map() to create a list without the word "The" at the beginning of the titles.

This code does the same thing as the previous listing (except that the result is an immutable `List` instead of a `MutableList`).

Like `forEach()`, the `map()` function calls the lambda once with each element in the list. However, unlike `forEach()`, `map()` will use the result of the lambda on each iteration to build out a new list.

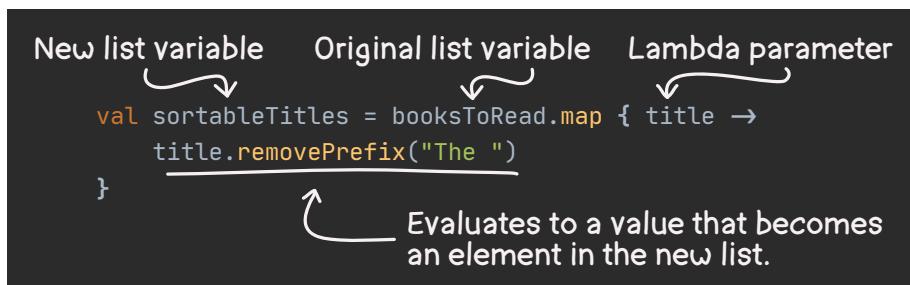
When you print each element of the list, you can see that both *The Ravine of Sorrows* and *The Kingsford Manor Mystery* have been updated so that the word "The" is not at the beginning.

```
Tea with Agatha
Mystery on First Avenue
Ravine of Sorrows
Kingsford Manor Mystery
Beyond the Expanse
```



When you print each element of the list, you can see that both *The Ravine of Sorrows* and *The Kingsford Manor Mystery* have been updated so that the word "The" is not at the beginning.

Let's take a closer look at the `map()` function:



- Similar to `forEach()`, the `map()` function is a higher-order function that takes a lambda.
- That lambda will run once for each element in the list.
- The result of the lambda will be an element in the new collection.
- The `map()` function returns that new collection.

Functions like `forEach()` and `map()` are called **collection operations**, because they're functions that perform some operation on (that is, they do something with) a collection.

"Perfect!" said Libby, "Now that the titles have been changed like I want, maybe I can sort them?"

Sorting Collections

The `forEach()` and `map()` functions are only two of many collection operations in Kotlin. Another one that can be quite helpful is called `sorted()`.

Since the `map()` function returns a collection, Libby can just call `sorted()` immediately after the call to `map()`, like this:

```
val sortedTitles = booksToRead.map { title -> title.removePrefix("The ") }.sorted()
```

Listing 8.18 - Combining map() with sorted().

When she printed out the elements of `sortedTitles`, she saw the output she was hoping for!

```

Beyond the Expanse
Kingsford Manor Mystery
Mystery on First Avenue
Ravine of Sorrows
Tea with Agatha
  
```

In order to make things easier to read, each collection operation can go on its own line, like this:

```
val sortedTitles = booksToRead
    .map { title -> title.removePrefix("The ") }
    .sorted()
```

Listing 8.19 - Formatting multiple collection operations so that they line up.

This code is identical to the previous listing except for the *formatting*. In other words, all of the letters and punctuation are exactly the same and in the same order - it's only the space between them that's different.

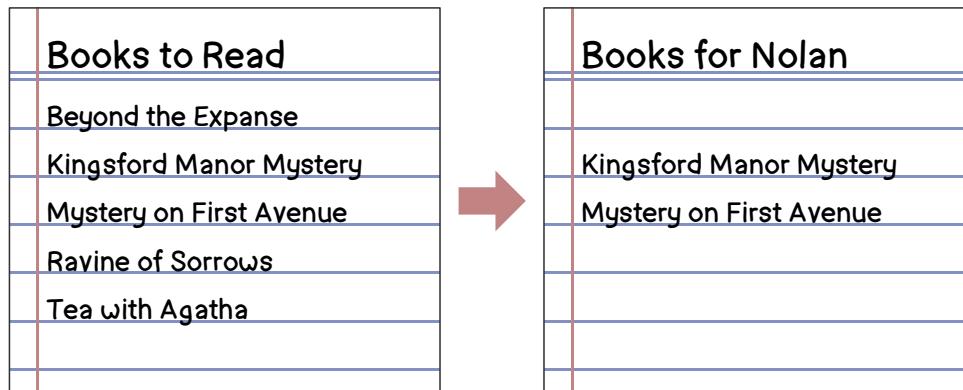
Writing the collection operations vertically like this can be helpful because it makes it easy to scan down the lines to see what collection operations are involved and what order they're in. For example, *first* the titles are mapped, and *then* the titles are sorted. For that reason, Kotlin developers often format their code this way.

Filtering Collections: Including and Omitting Elements

Libby is excited! Now she's got a list of books - sorted alphabetically - to share with Nolan.

"I can't wait to see your list of books," said Nolan. "Just remember - I only read mystery novels!"

"Only mysteries...?" repeated Libby. "Okay," she thought to herself, "the final thing I need to do is remove the titles that are not mysteries." She pulled out a sheet from her notepad, and wrote a customized list just for Nolan, omitting any title that is not a mystery.



"How can I do this in Kotlin?" she wondered.

As you probably guessed, Kotlin includes a collection operation that makes this easy, and unsurprisingly, it's called `filter()`.

Just like an air filter that blocks unwanted dust and allergens from getting through to your air conditioner system, a Kotlin list filter blocks elements that you don't want to get through to a new list!

Let's use the `filter()` function to filter down the list of books to just those that have "Mystery" in the title:

```
val booksForNolan = booksToRead
    .map { title -> title.removePrefix("The ") }
    .sorted()
    .filter { title -> title.contains("Mystery") }
```

Listing 8.20 - Using the `filter()` function to filter out unwanted elements.

The `filter()` function is similar to the `map()` function above - it takes a lambda as an argument, and that lambda will be invoked once for each title in the original list. Unlike the `map()` function, however, the lambda for `filter()` must return a `Boolean`. If it returns `true` for an element, then that element is passed into the new collection (i.e., `booksForNolan` in this case). If it returns `false`, then it's omitted from the new collection.

Here's a breakdown of how to use the `filter()` function:



New list variable Original list variable Lambda parameter

```
val booksForNolan = booksToRead.filter { title ->
    title.contains("Mystery")
}
```

Determines whether this element is included in the new list.

Printing each element of the list, here's what Libby saw:

```
Kingsford Manor Mystery
Mystery on First Avenue
```

"Great," she said. "The list is exactly like I wanted it. It includes only mysteries, and it's sorted properly!"

Collection Operation Chains

Let's look at that code again:

```
val booksForNolan = booksToRead
    .map { title -> title.removePrefix("The ") }
    .sorted()
    .filter { title -> title.contains("Mystery") }
```

Listing 8.21 - Using `map()`, `sorted()`, and `filter()` - copied from Listing 8.20.

In Kotlin, it's common to put multiple collection operations together like this, one after another. When we do this, it's called **chaining** the collection operations - each operation is like one link in the chain. In this code listing, the `map()`, `sorted()`, and `filter()` calls are *chained* together.

```
Collection operation chain {  
    val booksForNolan = booksToRead  
        .map { title → title.removePrefix("The ") }  
        .sorted()  
        .filter { title → title.contains("Mystery") }
```

Keep in mind that the operation chain is *not mutating a single list*. In fact, each of these operations creates a *new list*. The list that's created by the *final* operation, `filter()`, is the list that is assigned to the variable `booksForNolan`. The **intermediate lists** - that is, the lists that are created by the collection operations *inside* the chain - are used by the next operation in the chain, but are not assigned to any variable. It's still important to keep these intermediate lists in mind, though. This next illustration shows the list that's involved at each step in the chain.



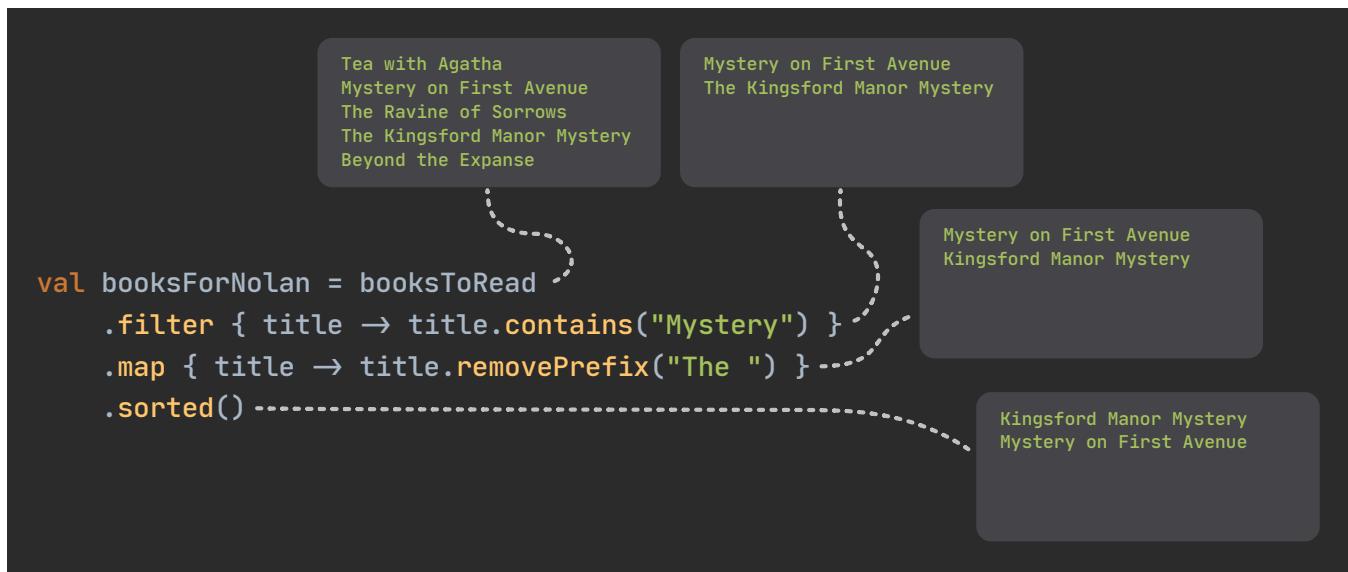
Whenever you've got a collection operation chain like this, it's helpful to consider how many elements are in each intermediate list. For example, the code in Listing 8.21 has the `filter()` call at the end of the chain. But what if it went at the *beginning* of the chain instead, like this?

```
val booksForNolan = booksToRead  
    .filter { title → title.contains("Mystery") }  
    .map { title → title.removePrefix("The ") }  
    .sorted()
```

Listing 8.22 - Moving the `filter()` call to the top of the chain.

By doing this, the intermediate list that `filter()` produces would only have two elements, in which case the `map()` function would only need to invoke its lambda twice instead of five times, and `sorted()` would only have two items to sort instead of five. In this example, the final list is the same either way, but Listing 8.22 is likely to be more efficient than Listing 8.21.

Here's an illustration showing the list involved at each step when placing the `filter()` call at the top. Notice that the intermediate lists have fewer elements than they did in the previous illustration.



On a small list like this, it's not a big deal, but on a list that has hundreds or thousands of elements, you could see how this could improve the **performance** of your code - that is, it would run much faster!

Other collection operations

Kotlin has many other collection operations that are easy to use! Just to give you an idea, here are a few others that might be helpful to you.

- `drop(3)` - The new list omits the first 3 elements from the original list.
- `take(5)` - The new list uses only the first 5 elements from the original list.
- `distinct()` - The new list will omit duplicate elements, so that each element is included only once.
- `reversed()` - The new list will have the same elements as the original, but their order will be backwards.

You can see a [more complete list of them](#) in the official Kotlin API documentation.

Introduction to Sets

Before we wrap up this chapter, it's worth noting that lists aren't the only kind of collection in Kotlin. Lists are probably the most frequently used, but another helpful collection type is called a **set**. Whereas lists are helpful for ensuring that its elements are in a particular order, sets are helpful for ensuring that *each element in it is always unique*.

For example, Nolan's favorite mystery author, Slim Chancery, has written three books, and Nolan is proud to say he's got the whole set.

Creating a set in Kotlin is just as easy as creating a list. Simply use `setOf()` or `mutableSetOf()` instead of `listOf()` or `mutableListOf()`.

```
val booksBySlim: Set<String> = setOf(  
    "The Malt Shop Caper",  
    "Who is Mrs. W?",  
    "At Midnight or Later",  
)
```

Listing 8.23 - Creating a set of strings.

When you add an element to a set that already has that value, the set will remain unchanged.

```
val booksBySlim: MutableSet<String> = mutableSetOf(  
    "The Malt Shop Caper",  
    "Who is Mrs. W?",  
    "At Midnight or Later",  
)  
  
booksBySlim.add("The Malt Shop Caper")  
  
println(booksBySlim)  
// [The Malt Shop Caper, Who is Mrs. W?, At Midnight or Later]
```

Listing 8.24 - Adding an element to a set that already contains that element. The set does not include it a second time.

Note that a set *does not guarantee the order of its elements* when you print them out or use a collection operation on it. It's *possible* that the elements will be in the same order that you added them, but don't depend on it!

Because sets don't have any particular order to their elements, their elements do not have indexes. For that reason, sets do not even include a `get()` function!

The key takeaway is that:

1. Lists have elements in a guaranteed order, and can contain duplicates.
2. Sets have elements in no particular order, and are guaranteed not to contain duplicates.

Also, you can convert a list into a set, or the other way around. Simply use `toSet()` or `toList()`. Just remember that if you convert a list to a set, you'll lose duplicate elements, and the order could possibly be different!

```
val bookList = listOf(  
    "The Malt Shop Caper",  
    "At Midnight or Later",  
    "The Malt Shop Caper",  
)  
  
val bookSet = bookList.toSet() // bookSet has two elements  
val anotherBookList = bookSet.toList() // anotherBookList also has two elements
```

Listing 8.25 - Converting between lists and sets.

Best Practice: Lists or Sets?

When deciding whether to use a list or a set, consider *how you intend to use the collection*. In the examples above, both the lists and the sets included book titles, but the deciding factor was *how the collection would be used*.

It made sense to use a list for the reading list, because Libby wants to read the books in a particular order, and it's possible that she might want to read the same book more than once.

For the books by Slim Chancery, a set made sense because the order didn't matter, but we wanted to make sure there were no duplicates in it - Slim doesn't write the same book twice! On the other hand, if we wanted to ensure that his books were in order by title or publication date, for example, a list would have been a better choice.

Summary

Up until this chapter, we've only worked with individual variables. By using collections like lists and sets, we're able to do things with entire groups of values, which opens a whole new world of possibilities! In this chapter, you learned about collections, including:

- How to [create a list](#).
- How to create lists by [adding or removing](#) an element from another list.
- The difference between immutable and [mutable collections](#).
- How to [get a single element](#) out of a list.
- [Collection operations](#), like `filter()`, `map()`, and `sorted()`.
- The difference between a list and a [set](#).

We discovered how easy it is to get an element from a list by its index. However, sometimes you want an easy way to get an element by *some other information* about it. For example, instead of getting a book by its positional index, you might want to get a book by its ISBN (that long number above the barcode on the back). In the next chapter, we'll learn about another way to group elements that will make it easy to do this!

Kotlin: An Illustrated Guide

Chapter 9

Collections: Maps



Creating Associations Between Two Values

In the last chapter, we saw how collections, such as lists and sets, can be used to do things that couldn't be done easily with separate, individual variables.

In this chapter, we're going to look at one more kind of collection - a map, which can hold many associations between values.

Let's check it out!

The Right Tool for the Job

"You gotta use the right tool for the job." That's what Mr. Andrews taught his young son Jim, who was just starting to learn how to become a handyman like his old man. "When you've got a nail, you need to use a hammer, not a screwdriver."

In order to help Jim pick out the right tool, he sketched out a table of the different hardware and tools in his toolbox.

Hardware	Tool to Use
Nail	Hammer
Hex Nut	Wrench
Hex Bolt	Wrench
Slotted Screw	Slotted Screwdriver
Phillips Screw	Phillips Screwdriver

"Now, with this table, you can easily look up what tool you need. Just scan down the left-hand column for the hardware you need to work with, and then scan across to see the right tool to use."

Which tool should be used with a slotted screw?



Hardware	Tool to Use
Nail	Hammer
Hex Nut	Wrench
Hex Bolt	Wrench
Slotted Screw	Slotted Screwdriver
Phillips Screw	Phillips Screwdriver

A Slotted Screwdriver!

Tables like this show that there are **associations** between things - a nail is associated with a hammer, a hex nut is associated with a wrench, and so on. In this chapter, we'll build out Kotlin's equivalent of a table like this, but before we do, let's start by creating a single association.

Associating Data

One simple way to associate two values is to use a class called `Pair`. The [constructor](#) of this class has two parameters. You can call its constructor with *any* two objects, regardless of their types. In our case, let's associate two `String` objects - one for a nail, and one for a hammer.

```
val association = Pair("Nail", "Hammer")
```

Listing 9.1 - Creating a simple association with the `Pair` class.

`Pair` is a very simple class that has two properties, `first` and `second`, which you can use to get the values back out of it. Below is a UML diagram showing the `Pair` class. The types of `first` and `second` depend on the types of the arguments you give it when calling its constructor, so in this diagram, we will just use `A` and `B` as placeholders for the actual types.

`Pair<A, B>`

+ `first: A`
+ `second: B`

The `first` property will be whatever the `first` argument was when you called the constructor, and the `second` will be whatever the `second` argument was.

```
println(association.first) // Nail
println(association.second) // Hammer
```

Listing 9.2 - Printing the `first` and `second` properties of the `Pair`.

Easy, right?

Now, instead of calling the constructor of the `Pair` class, it's sometimes more natural to use a function called `to()`, which will call the `Pair` constructor for you. This function can be called on *any object at all*. Let's update our code so that it uses the `to()` function.

```
val association = "Nail".to("Hammer")
```

Listing 9.3 - Creating a `Pair` with the `to` function.

When reading this code, you might say, "When I have a Nail, then I should go `to` a Hammer." Both Listing 9.1 and Listing 9.3 do the same thing - they create a `Pair` where the left-hand value is assigned to the property called `first` and the right-hand value is assigned to the property called `second`.

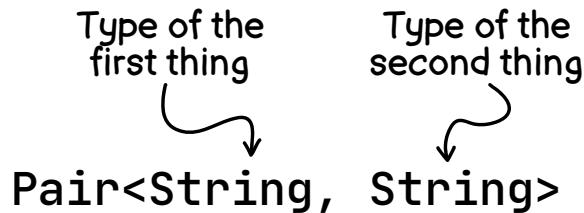
The `to()` function also has a special characteristic about it that lets you use it without the punctuation! So, you can also create this same `Pair` like this:

```
val association = "Nail" to "Hammer"
```

Listing 9.4 - Creating a `Pair` with the `to` function, without the punctuation.

Notice that this is the same as Listing 9.3 above, except that the `.`, the `(`, and the `)` are all missing. When a function lets you call it this way, it's called an **infix function**. We won't see infix functions often, but it's important to know that they exist so that they won't confuse you when you see code like this.

So far, we've used [type inference](#) so that we don't have to write out the type of the `association` variable. As with `List` and `Set` in the previous chapter, the type of a `Pair` variable depends on the type of the things that it contains. Since both "Nail" and "Hammer" have type `String`, the type of the `association` variable is `Pair<String, String>`.



And of course, we can specify the type explicitly like this:

```
val association: Pair<String, String> = "Nail" to "Hammer"
```

Listing 9.5 - Explicitly specifying the type of the association variable.

Now that we've successfully made a single association, we can do the same thing for the rest of the tools... and then put them all together into a map!

Map Fundamentals

Let's look at Mr. Andrews' table again.

Hardware	Tool to Use
Nail	Hammer
Hex Nut	Wrench
Hex Bolt	Wrench
Slotted Screw	Slotted Screwdriver
Phillips Screw	Phillips Screwdriver

In Kotlin, a table like this is called a **map**. You might be familiar with maps like street maps and treasure maps, but that's not what we're talking about here.

The term comes from the world of mathematics, where a map defines a *correspondence* between elements of sets. Similarly, Kotlin maps define an association between each item in the left-hand column, and the corresponding item in the right-hand column.

Before we create our first map, let's cover a few important terms:

- The items in the left-hand column of the table are called **keys**.
- The items in the right-hand column are called **values**.
- The association of a key and a value within a map is called a map **entry**.

There's an important rule to keep in mind - *each key in a map is unique*. However, the values can be duplicated.

A table illustrating a map structure. The left column is labeled "Hardware" and the right column is labeled "Tool to Use". The rows contain pairs of hardware items and their corresponding tools. Red arrows point from the left side of the table to the word "Keys" and from the right side to the word "Values".

Hardware	Tool to Use
Nail	Hammer
Hex Nut	Wrench
Hex Bolt	Wrench
Slotted Screw	Slotted Screwdriver
Phillips Screw	Phillips Screwdriver

A table illustrating a map structure. The left column is labeled "Hardware" and the right column is labeled "Tool to Use". The rows contain pairs of hardware items and their corresponding tools. Red arrows point from the left side of the table to the word "Entries".

Hardware	Tool to Use
Nail	Hammer
Hex Nut	Wrench
Hex Bolt	Wrench
Slotted Screw	Slotted Screwdriver
Phillips Screw	Phillips Screwdriver

In other words, you cannot have duplicate items in the left-hand column, but it's fine in the right-hand column.

You can see this in the table below, where the left-hand column items are unique, but the wrench appears twice in the right-hand column.

A table illustrating a map structure. The left column is labeled "Hardware" and the right column is labeled "Tool to Use". The rows contain pairs of hardware items and their corresponding tools. Red annotations include "No duplicates in this column" pointing to the Hardware column, "Duplicates allowed in this column" pointing to the Tool to Use column, and "Duplicated value!" pointing to the second row where the Wrench is listed twice.

Hardware	Tool to Use
Nail	Hammer
Hex Nut	Wrench
Hex Bolt	Wrench
Slotted Screw	Slotted Screwdriver
Phillips Screw	Phillips Screwdriver

Creating a Map with `mapOf()`

Now that we understand the main concepts, it's time to create our first map! To do this, we'll use the `mapOf()` function, passing in a `Pair` for each association that we want in the map.

```
val toolbox = mapOf(  
    "Nail" to "Hammer",  
    "Hex Nut" to "Wrench",  
    "Hex Bolt" to "Wrench",  
    "Slotted Screw" to "Slotted Screwdriver",  
    "Phillips Screw" to "Phillips Screwdriver",  
)
```

Listing 9.6 - Creating a map.

If you read the last chapter, you'll notice that this looks similar to `listOf()` and `setOf()`, except that all of the elements here have two pieces - the *key* and the *value*, which are joined together in a `Pair`.

By the way...

Remember - `to` is just another way to create a `Pair`. We could also have created the map like this:

```
val toolbox = mapOf(  
    Pair("Nail", "Hammer"),  
    Pair("Hex Nut", "Wrench"),  
    // ... and so on ...  
)
```

You can see the similarities between the Kotlin map and Mr. Andrews' table when you place them side by side:

```
val toolbox = mapOf(  
    "Nail" to "Hammer",  
    "Hex Nut" to "Wrench",  
    "Hex Bolt" to "Wrench",  
    "Slotted Screw" to "Slotted Screwdriver",  
    "Phillips Screw" to "Phillips Screwdriver",  
)
```

Hardware	Tool to Use
Nail	Hammer
Hex Nut	Wrench
Hex Bolt	Wrench
Slotted Screw	Slotted Screwdriver
Phillips Screw	Phillips Screwdriver

Just as with lists, sets, and other variables, you can use `println()` to print out the contents of a map.

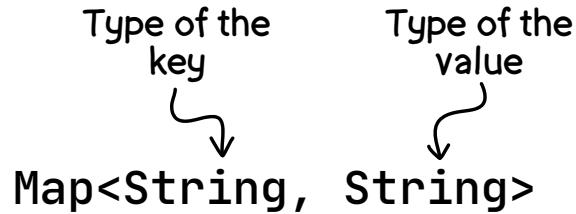
```
println(toolbox)
```

Listing 9.7 - Code to print the contents of a map to the screen.

When you print this out, the entries of the map appear between braces. The keys are to the left of the equal signs, and the values are to the right.

```
{Nail=Hammer, Hex Nut=Wrench, Hex Bolt=Wrench, Slotted Screw=Slotted Screwdriver,  
Phillips Screw=Phillips Screwdriver}
```

Just like with `Pair`, the type of a `Map` variable depends on the type of the key and the type of the value.



So, we could write out the type explicitly like this:

```
val toolbox: Map<String, String> = mapOf(  
    "Nail" to "Hammer",  
    "Hex Nut" to "Wrench",  
    "Hex Bolt" to "Wrench",  
    "Slotted Screw" to "Slotted Screwdriver",  
    "Phillips Screw" to "Phillips Screwdriver",  
)
```

Listing 9.8 - Explicitly specifying the type of the `toolbox` variable.

Looking Up a Value

The most common thing that you'll need to do with a map is to look up a value. When Jim has a nail, for example, he needs to look up which tool to use. Just as Jim would find the nail in the left-hand column and find the corresponding tool next to it, Kotlin can give you the value when you provide it a key. You can use the `get()` function to do this.

```
val tool = toolbox.get("Nail")  
println(tool) // Hammer
```

Listing 9.9 - Getting a value from a map by the value's corresponding key.

Similar to lists, you can also use the [indexed access operator](#) with maps to get a value.

```
val tool = toolbox["Nail"]  
println(tool) // Hammer
```

Listing 9.10 - Using the indexed access operator to get a value from a map.

If you call `get()` (or use the indexed access operator) with a key that does not exist in the map, it will return a `null`. This means the `get()` function returns a [nullable](#) type rather than a non-nullable type! In Listing 9.9 and 9.10, it returns a `String?` rather than a `String`.

You can use the null-safety tools you learned about in [Chapter 6](#) (such as the [elvis operator](#)) to get it back to a non-null type. Alternatively, you can call `getValue()` instead of `get()`. `getValue()` will return a non-null type, but be warned - if you give it a key that does not exist, you'll see an error message and your code will stop running.

```
val tool = toolbox.getValue("Nail")
println(tool) // Hammer

val anotherTool = toolbox.getValue("Wing Nut") // Error at runtime
```

Listing 9.11 - Using the `getValue()` function to get a value with a non-null type.

You can also use `getOrDefault()` to provide a default value if the key doesn't exist. If Mr. Andrews doesn't have a tool for a particular piece of hardware, he'll just need to tighten it by hand!

```
val tool = toolbox.getOrDefault("Hanger Bolt", "Hand")
```

Listing 9.12 - Using `getOrDefault()` to provide a default value if the key does not exist in the map.

Modifying a Map

As with the other collection types, maps come in two flavors of mutability - `MutableMap` and an immutable `Map`. The mutable variety allows you to change its contents, whereas an immutable map requires you to create a *new* map instance that you can assign to a new or existing variable.

Let's look at how to change a `MutableMap` first. To start with, we'll need to use `mutableMapOf()` to create the map, instead of just `mapOf()`, which we used back in Listing 9.6.

```
val toolbox = mutableMapOf(
    "Nail" to "Hammer",
    "Hex Nut" to "Wrench",
    "Hex Bolt" to "Wrench",
    "Slotted Screw" to "Slotted Screwdriver",
    "Phillips Screw" to "Phillips Screwdriver",
)
```

Listing 9.13 - Creating a mutable map.

To add a new entry, you can use the `put()` function, where the first argument is the key, and the second argument is the value.

```
toolbox.put("Lumber", "Saw")
```

Listing 9.14 - Adding a new entry to a mutable map using the `put()` function.

Just like with the `get()` function, though, Kotlin developers typically use the *indexed access operator* instead of calling the `put()` function directly. The following code accomplishes the same thing as Listing 9.14.

```
toolbox["Lumber"] = "Saw"
```

Listing 9.15 - Adding a new entry to a mutable map using the indexed access operator.

You can also *change an existing value* exactly the same way. Just provide a key that already exists.

```
toolbox["Hex Bolt"] = "Nut Driver"
```

Listing 9.16 - Changing an existing value in a mutable map.

Finally, you can remove an entry using the `remove()` function.

```
Toolbox.remove("Lumber")
```

Listing 9.17 - Removing an entry from a map.

Note that although you can change a value, you *cannot change a key*. Instead, you can remove a key and insert a new entry.

```
toolbox.remove("Phillips Screw")
toolbox["Cross Recess Screw"] = "Phillips Screwdriver"
```

Listing 9.18 - Removing an entry and reinserting it with a new key, to simulate changing a key.

Immutable Maps

As with immutable lists and sets, you can use the *plus* and *minus operators* on an immutable map. Remember, doing so will create *new map instances*, which you'd typically assign to a variable.

The following code demonstrates the same operations as we did above, but on an immutable map. Notice that we use the `var` keyword here so that we can assign each result back to the same `toolbox` variable!

```
var toolbox = mapOf(  
    "Nail" to "Hammer",  
    "Hex Nut" to "Wrench",  
    "Hex Bolt" to "Wrench",  
    "Slotted Screw" to "Slotted Screwdriver",  
    "Phillips Screw" to "Phillips Screwdriver",  
)  
  
// Add an entry  
toolbox = toolbox + Pair("Lumber", "Saw")  
  
// Update an entry  
toolbox = toolbox + Pair("Hex Bolt", "Nut Driver")  
  
// Remove an entry  
toolbox = toolbox - "Lumber"  
  
// Simulate changing a key  
toolbox = toolbox - "Phillips Screw"  
toolbox = toolbox + Pair("Cross Recess Screw", "Phillips Screwdriver")
```

Listing 9.19 - Adding, changing, and removing entries by replacing an immutable map.

Map Operations

As with `List` and `Set`, `Map` objects have operations that can be performed on them, and some of them will look very familiar. Let's start with the `forEach()` function.

forEach()

The `forEach()` function is almost identical to the one found on `List` and `Set` objects. It takes a lambda that you can use to do something with each entry in the map. Since maps store *entries*, the parameter of the lambda will be of type `Map.Entry`.

`Map.Entry` is very similar to the `Pair` class that we looked at earlier in this chapter - it has two properties on it, but instead of being named `first` and `second`, they're named `key` and `value`.

Here's how you can use the `forEach()` function on a `Map`.

Map.Entry<K, V>
+ key: K
+ value: V

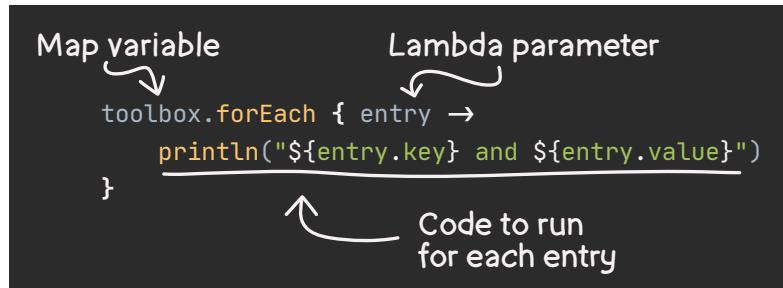
```
toolbox.forEach { entry ->  
    println("Use a ${entry.value} on a ${entry.key}")  
}
```

Listing 9.20 - Using the `forEach()` function on a map.

When you run this code, here's what you'll see.

```
Use a Hammer on a Nail
Use a Wrench on a Hex Nut
Use a Wrench on a Hex Bolt
Use a Slotted Screwdriver on a Slotted Screw
Use a Phillips Screwdriver on a Phillips Screw
```

Because it's so similar to the `forEach()` that you saw in the last chapter, you should be able to identify the main parts. Here they are:



Filtering

Similar to lists and sets, you can `filter` a map. Keep in mind that, just as we saw with lists, this function doesn't modify an existing map - it creates a new map instance, so you'll probably want to assign the result to a variable. Let's filter down the toolbox to just screwdrivers.

```
val screwdrivers = toolbox.filter { entry ->
    entry.value.contains("Screwdriver")
}
```

Listing 9.21 - Using the `filter()` function on a map.

The result is a new Map that contains only the screwdrivers.

Hardware	Tool to Use
Nail	Hammer
Hex Nut	Wrench
Hex Bolt	Wrench
Slotted Screw	Slotted Screwdriver
Phillips Screw	Phillips Screwdriver



Hardware	Tool to Use
Slotted Screw	Slotted Screwdriver
Phillips Screw	Phillips Screwdriver

In this case, we filtered on the `value`, but you can just as easily filter by the `keys`.

```
val screwdrivers = toolbox.filter { entry ->
    entry.key.contains("Screw")
}
```

Listing 9.22 - Filtering a map based on the entry's key.

Mapping

Yes, you can map a **Map**! Simply use the `mapKeys()` and `mapValues()` functions to convert its keys or values. Just like with the collection operations we looked at in the last chapter, you can create an [operation chain](#). Let's map both keys and values in one chain.

```
val newToolbox = toolbox
    .mapKeys { entry -> entry.key.replace("Hex", "Flange") }
    .mapValues { entry -> entry.value.replace("Wrench", "Ratchet") }
```

Listing 9.23 - Mapping both keys and values of a map.

Hardware	Tool to Use
Nail	Hammer
Hex Nut	Wrench
Hex Bolt	Wrench
Slotted Screw	Slotted Screwdriver
Phillips Screw	Phillips Screwdriver



Hardware	Tool to Use
Nail	Hammer
Flange Nut	Ratchet
Flange Bolt	Ratchet
Slotted Screw	Slotted Screwdriver
Phillips Screw	Phillips Screwdriver

Map objects have many other operations on them, which you can explore in Kotlin's [API docs](#). However, there's one more operation that we'll examine before continuing - `withDefault()`.

Setting Default Values

As we saw earlier, you can use `getOrDefault()` to gracefully handle cases where the key does not exist. However, this can quickly get out of control if you use the same default every time...

```
val tool = toolbox.getOrDefault("Hanger Bolt", "Hand")
val another = toolbox.getOrDefault("Dowel Screw", "Hand")
val oneMore = toolbox.getOrDefault("Eye Bolt", "Hand")
```

Listing 9.24 - Using `getOrDefault()` can get out of hand when used at every call site.

Instead, you can use an operation called `withDefault()`, which will return a new map based on the original. In this new map, whenever you call `getValue()` with a key that doesn't exist, it will invoke a lambda and return the result. Here's how it looks:

```
toolbox = toolbox.withDefault { key -> "Hand" }
```

Listing 9.25 - Creating a new map that has a default value.

Now, instead of providing the default every time you try to get a value (as done in Listing 9.24 above), you can just call `getValue()` normally.

This is great, because if you ever wanted to change the default, you could make the change in one spot instead of many!

```
val tool = toolbox.getValue("Hanger Bolt")
val another = toolbox.getValue("Dowel Screw")
val oneMore = toolbox.getValue("Eye Bolt")
```

Listing 9.26 - Calling `getValue()` instead of `getOrDefault()`, because the new map has a default value.

Keep in mind that this works with `getValue()` but *not* with `get()` or the indexed access operator, which will continue to return `null` if the key is not found!

Now you know how to create and change maps, get values out of them, and use collection operations on them. But things get really fun when you start using maps in conjunction with other *collections*! Let's look at that next.

Creating a Map from a List

We've created maps by hand using the `mapOf()` function. It's also possible to create maps that are based on *existing list or set collections*. With just a few important functions, you can slice and dice your data in many different ways! In order to do this, of course, we will need a list to start with.

Instead of using a simple `String` to represent the tools in Mr. Andrews' toolbox, let's create a [class](#), so that it can hold the name of the tool, its weight in ounces, and the corresponding hardware that it works with.

```
class Tool(
    val name: String,
    val weightInOunces: Int,
    val correspondingHardware: String,
)
```

Listing 9.27 - Creating a class to hold more information about a tool.

Now, let's create a list of `Tool` objects, so that they include the tools from Mr. Andrews' toolbox.

```
val tools = listOf(  
    Tool("Hammer", 14, "Nail"),  
    Tool("Wrench", 8, "Hex Nut"),  
    Tool("Wrench", 8, "Hex Bolt"),  
    Tool("Slotted Screwdriver", 5, "Slotted Screw"),  
    Tool("Phillips Screwdriver", 5, "Phillips Screw"),  
)
```

Listing 9.28 - Creating a list of tools.

Now that we have a list, we're ready to create some maps from it!

Associating Properties from a List of Objects

You can use the `associate()` function to create a map from a list of objects. To start with, let's use `associate()` to create a map similar to the one in Listing 9.6:

```
val toolbox = tools.associate { tool ->  
    tool.correspondingHardware to tool.name  
}
```

Listing 9.29 - Creating a map by associating two properties in the elements of a list.

Hopefully you're starting to feel more comfortable with collection operations at this point. For each element in the list, the `associate()` function will invoke the lambda given to it. The lambda returns a key-value `Pair`, which contains the key and value that you want in the resulting map.

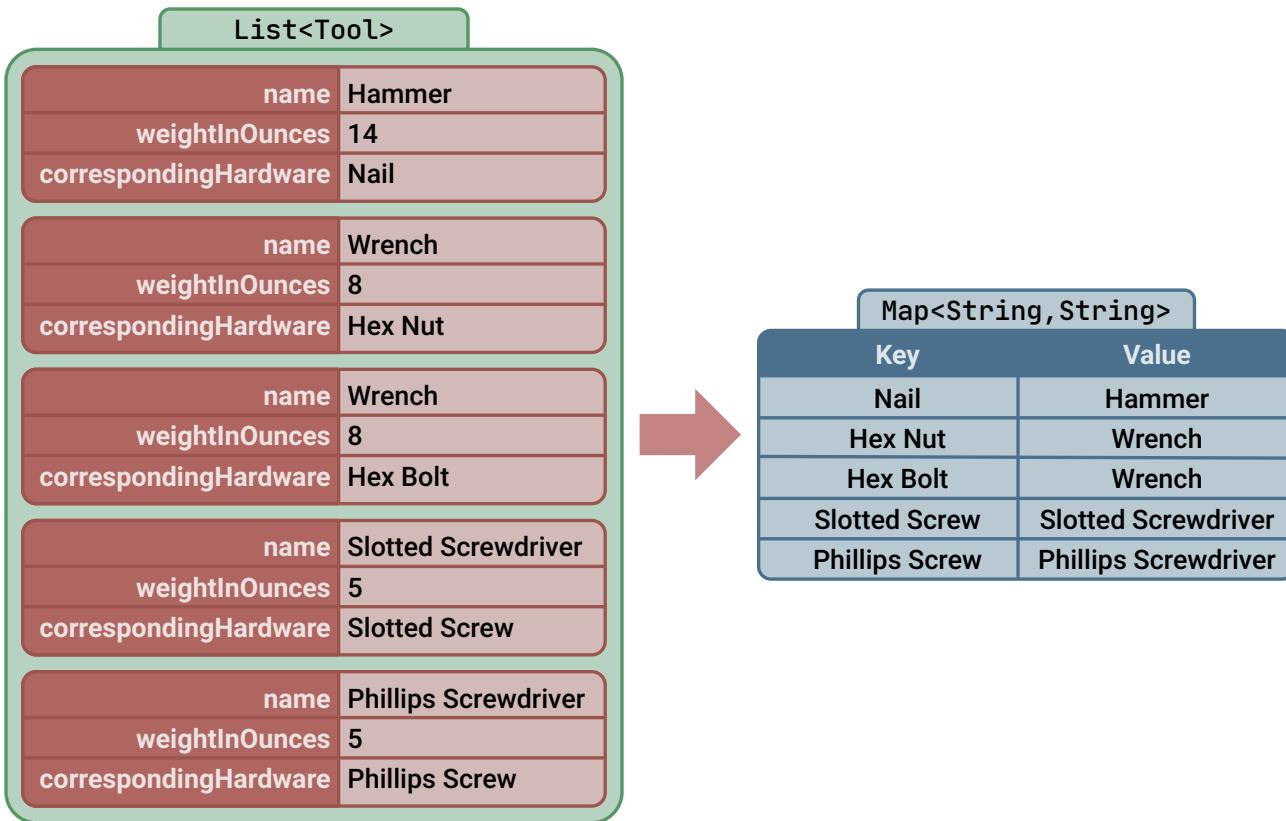
Here's a breakdown of the `associate()` function.

New map variable Original list variable Lambda parameter

```
val toolbox = tools.associate { tool ->  
    tool.correspondingHardware to tool.name  
}
```

Pair to insert as an entry in the new map.

And here's the effect that it has in Listing 9.29:



Often, the number of elements in the resulting map will be the same as the number of elements in the original list. In some cases, it could have fewer. Because the keys in a map are all unique, if you try to add a key that already exists, it will *overwrite* the existing value.

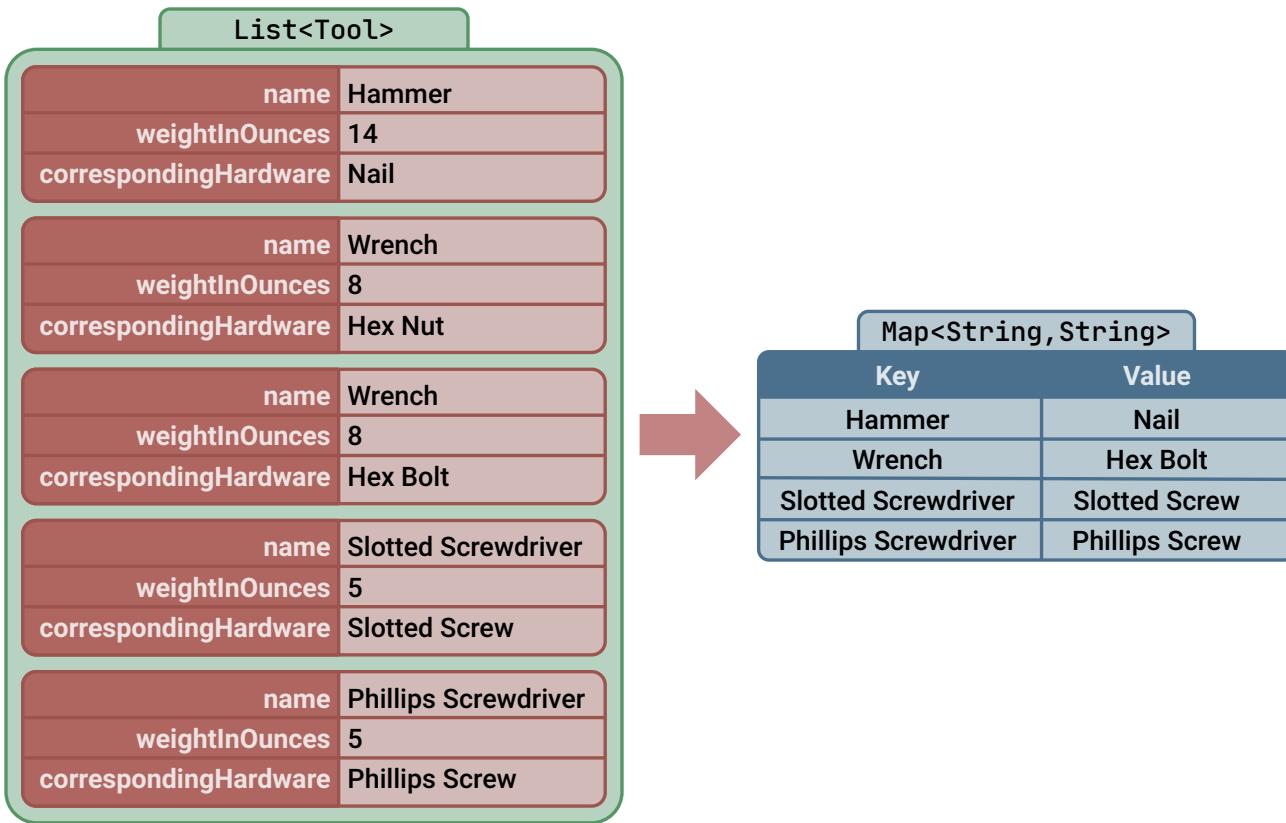
For example, let's reverse the key and value in the lambda in Listing 9.29, so that the tool name is the key, and the hardware is the value.

```
val toolbox = tools.associate { tool ->
    tool.name to tool.correspondingHardware
}
```

Listing 9.30 - Associating the properties in the reverse order from Listing 9.29.

The original list has two `Tool` objects with a `name` of `Wrench`, so when `associate()` encounters the *first* one, it's added to the map, but when it encounters the *second*, it replaces the first value.

So, the resulting map only includes `Hex Bolt` rather than `Hex Nut`, because of the two, `Hex Bolt` came last.



So in this case, there are fewer entries in the map than elements in the original list. That is, there are only 4 entries in the map, compared with 5 elements in the list.

Other Association Functions

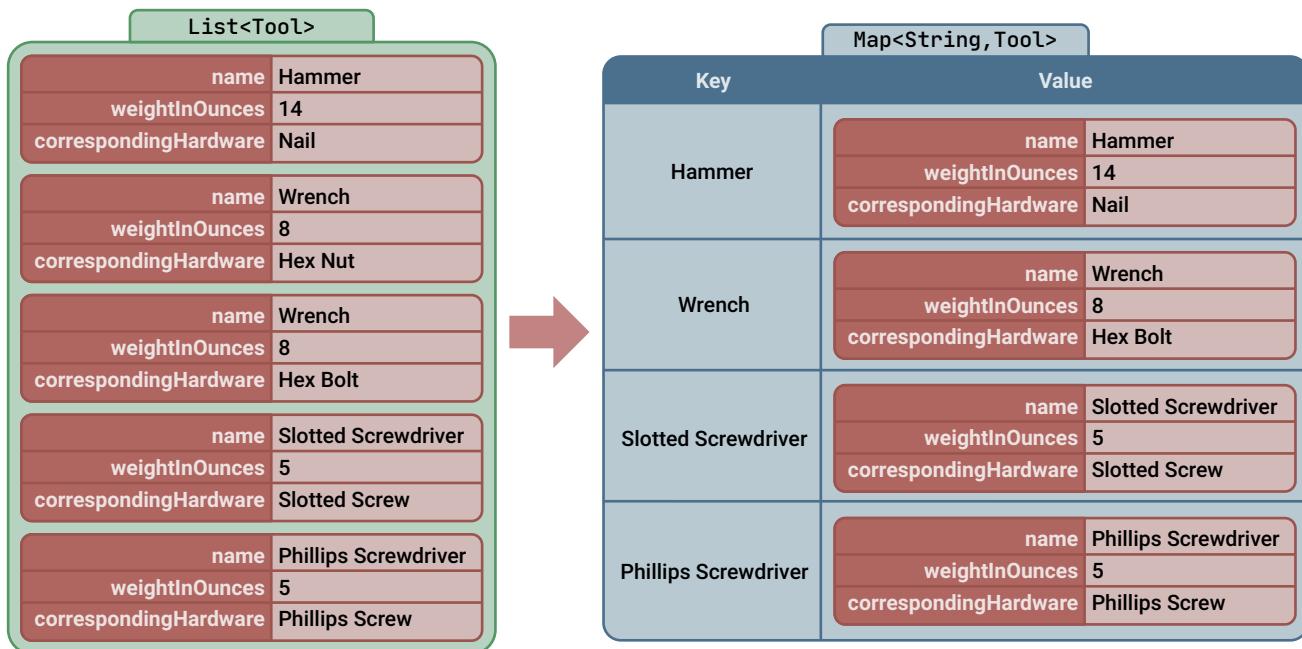
There are a few other variations of the `associate()` function that are good to know. These are especially helpful if you want the original list element to be either the *key* or the *value* in the resulting map.

For example, if you want to create a map where the keys are the tool names and the value is the `Tool` object, you can use `associateBy()`. The lambda of this function returns just the *key*. The original list element itself will be the value.

```
val toolsByName = tools.associateBy { tool -> tool.name }
```

Listing 9.31 - Calling associateBy() to create a map where the original list elements are the map's values.

This has the effect that's depicted in the following illustration.



With this map, you can easily get a tool by its name!

```
val hammer = toolsByName["Hammer"]
```

Listing 9.32 - Getting a Tool from a map by its name.

Inversely, if you want to create a map where the keys are the `Tool` object and the value is specified in the lambda, you can use the `associateWith()` function. The lambda of this function returns the *value*, and the original list element will be the key.

```
val toolWeightInPounds = tools.associateWith { tool ->
    tool.weightInOunces * 0.0625
}
```

Listing 9.33 - Calling associateWith() to create a map where the original list elements are the map's keys.

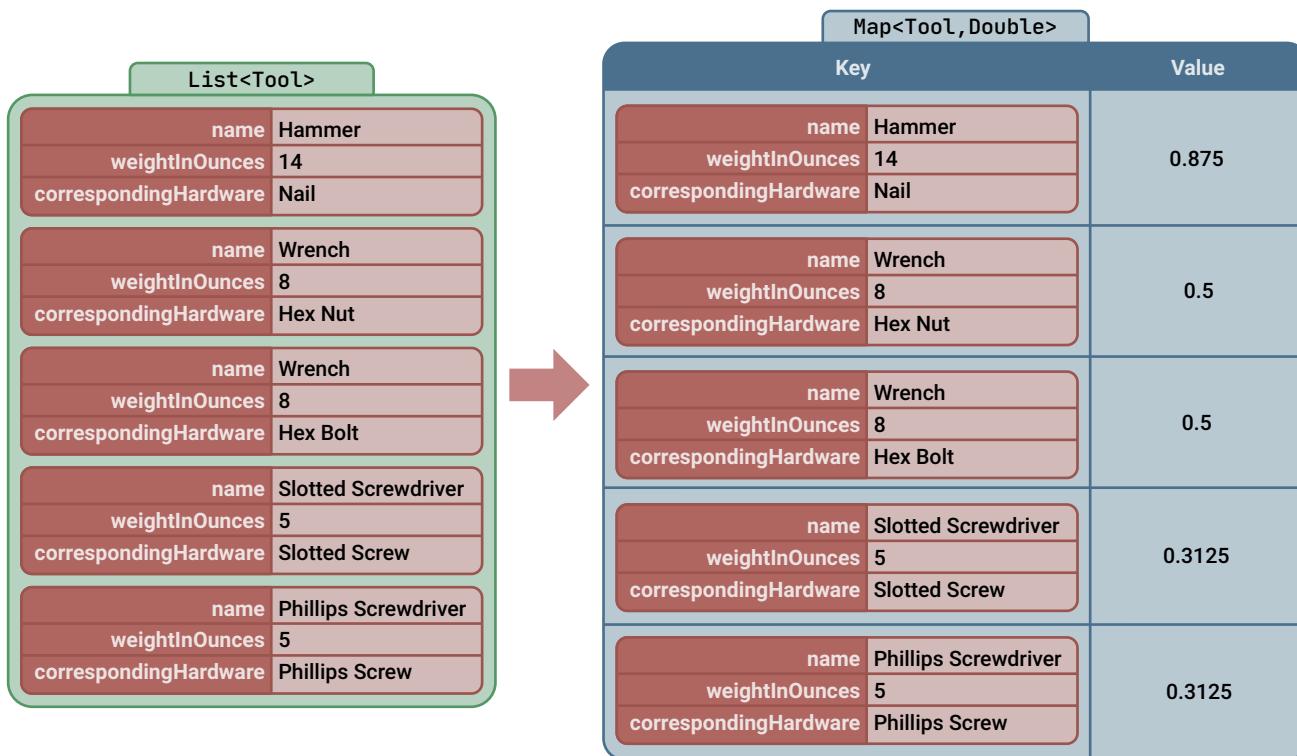
For the Nerds

The `associateBy()` function also has a version that accepts two arguments - one lambda that returns the key, and one lambda that returns the value.

It accomplishes the same thing as the regular `associate()` function, but it can sometimes run faster on large lists because it doesn't create a temporary `Pair` object for each element like `associate()` does.

Here's how you would use it to create the same map as in Listing 9.29:

```
val toolbox = tools.associateBy({ it.correspondingHardware }, { it.name })
```



To get the weight of a hammer, you'd need to have a hammer object already.

```
val hammerWeightInPounds = toolWeightInPounds[hammer]
```

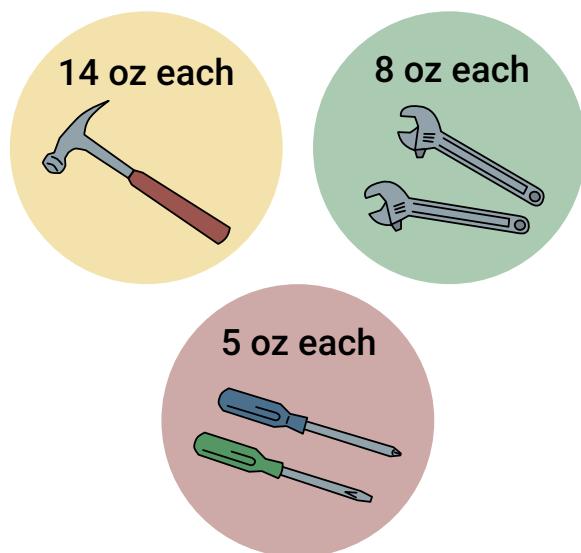
Listing 9.34 - Using an object as a key.

Grouping List Elements into a Map of Lists

Sometimes when you've got a list, you want to split it up into multiple smaller lists, based on some characteristic. For example, we can take the `tools` list and split it up by weight.

To do this, we can use the `groupBy()` function. This function will run the provided lambda for each element in the list.

Elements for which the lambda returns the same result will be assembled into a list, and inserted into a map.



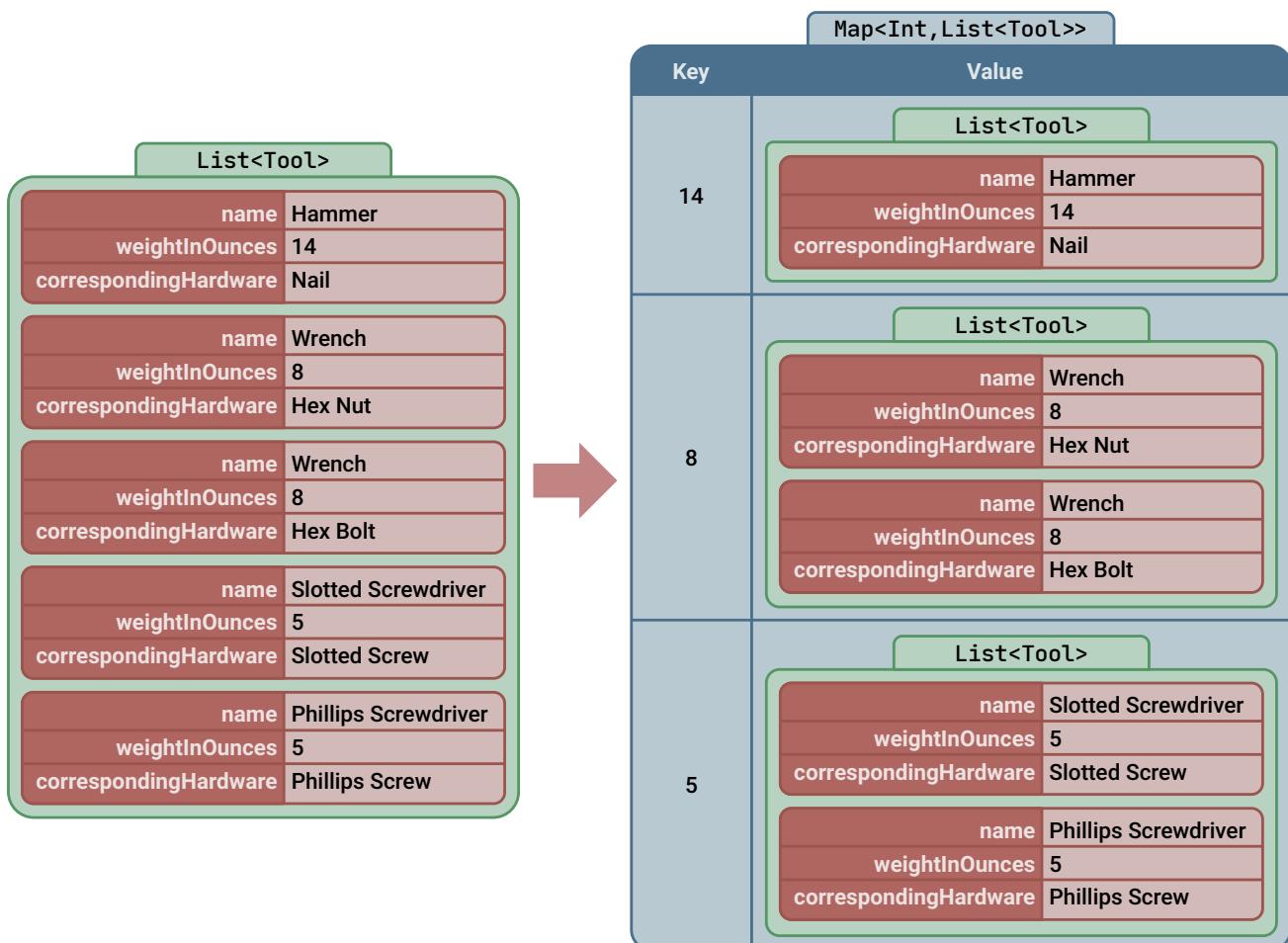
```
val toolsByWeight = tools.groupBy { tool ->
    tool.weightInOunces
}
```

Listing 9.35 - Using the `groupBy()` function to group a list of tools by their weight.

The result is a Map with one list of tools that weigh 14 ounces, *another* list of tools that weigh 8 ounces, and a *third* list of tools that weigh 5 ounces.

The map's *key* is the weight in ounces, and the map's *value* is a list of tools that have that weight.

This illustration shows the effect that this operation has.



Here's a breakdown of the `groupBy()` function:



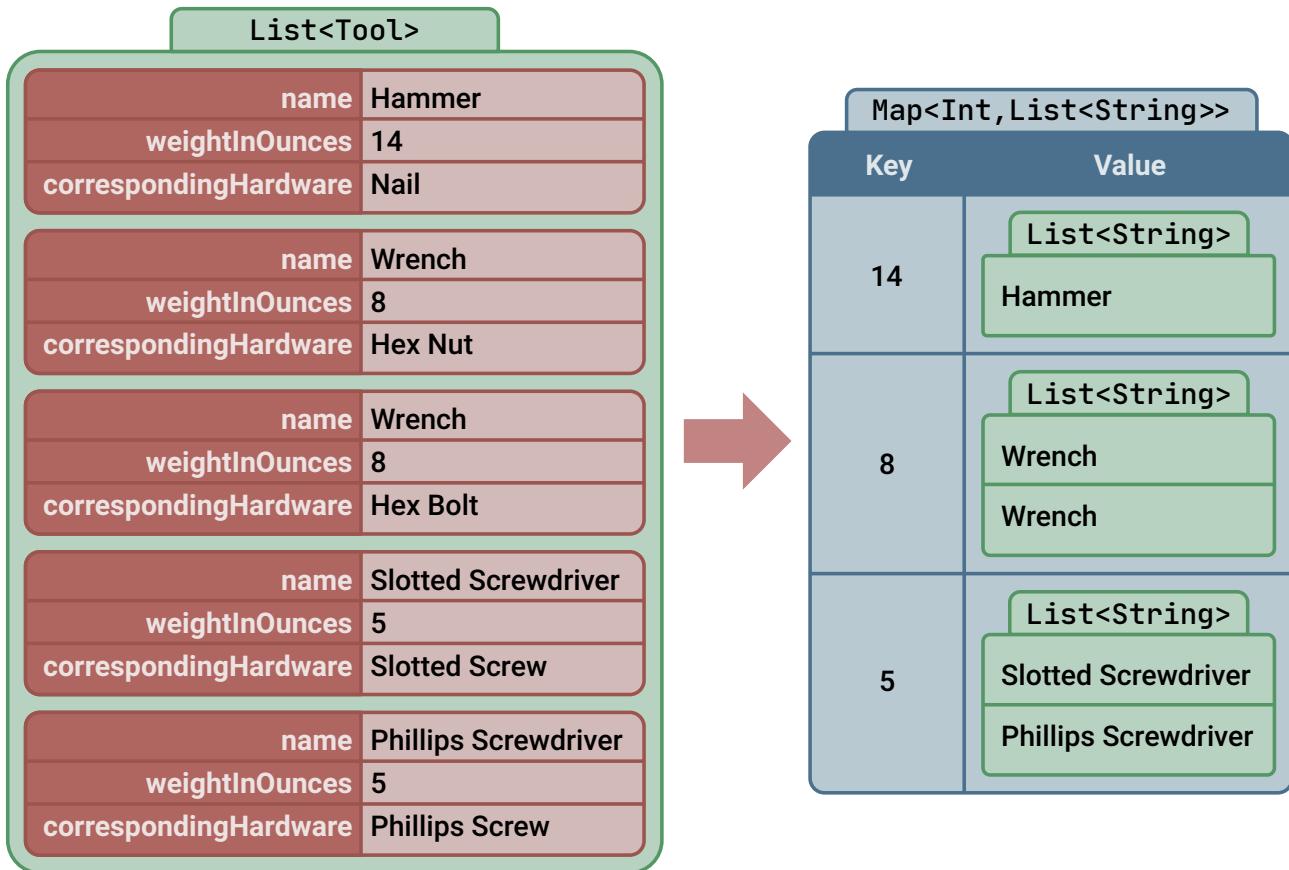
In case you want something *other* than the original list element in the resulting lists, you can also call this function with a *second* argument. For that one, give it a lambda that returns whatever you want in the resulting list.

For example, if you only want the *names* of the tools in those lists, you can do this:

```
val toolNamesByWeight = tools.groupBy(
    { tool -> tool.weightInOunces },
    { tool -> tool.name }
)
```

Listing 9.36 - Calling the groupBy() function with two arguments instead of one.

Here's the effect that this operation has:



Summary

Well, Jim is well on his way to becoming a great handyman like his father. And with the knowledge you've gained over the past nine chapters, you're well on your way to becoming a great Kotlin developer! Here's what you learned in this chapter:

- How to [associate two values](#) with a `Pair`.
- How to [create a simple map](#) with `mapOf()` and `mutableMapOf()`.
- How to [look up a value](#) in a map by its key.
- How to [modify a map](#).
- How to [perform operations](#) on a map.
- How to [associate keys and values](#) from an existing list of objects.
- How to [group elements](#) from an existing list of objects.

Now that you've learned about collections - like lists, sets, and maps - you've opened up many possibilities in your code. In the next chapter, we'll look at Receivers and Extensions.

Kotlin: An Illustrated Guide

Chapter 10

Receivers and Extensions



Tack New Functions and Properties onto Any Class!

Kotlin extensions can be used to add new functions and properties to existing classes - even to classes that you didn't write! In this chapter, we'll cover explicit receivers, implicit receivers, extension functions, and extension properties.

Buckle up!

Standalone Functions and Object Functions

Way back in [Chapter 2](#), we learned how to create functions. Here's a very simple function that puts single quotes at the beginning and the end of a `String`:

```
fun singleQuoted(original: String) = "'$original'"
```

Listing 10.1 - A simple function to wrap a string in single quotes.

As you recall, this function can be called easily, like this:

```
val title = "The Robots from Planet X3"
val quotedTitle = singleQuoted(title)

println(quotedTitle) // 'The Robots from Planet X3'
```

Listing 10.2 - Calling a simple function.

And then in [Chapter 4](#), we learned that `objects` can contain functions, too.

For example, `String` objects have a function named `uppercase()` that returns the same string, but with all uppercase letters. You can call it like this:

```
val title = "The Robots from Planet X3"
val loudTitle = title.uppercase()

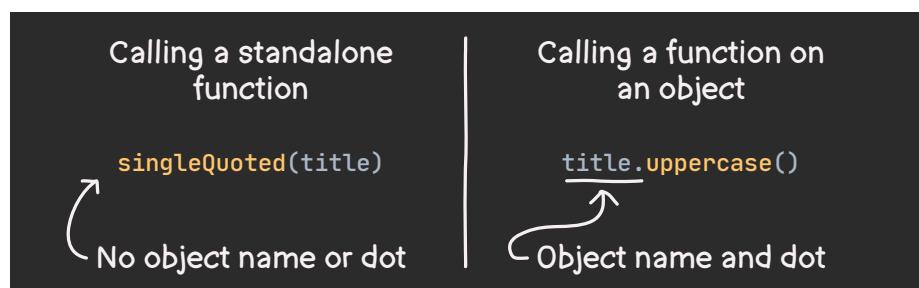
println(loudTitle) // THE ROBOTS FROM PLANET X3
```

Listing 10.3 - Calling a function on an object.

When you call a function that's on an object like this, you prefix the function call with the *name of the object* and a dot. For this reason, this way of writing a function call is called **dot notation**.

So, we have two different *categories* of functions here:

- Functions that *stand alone*, apart from an object.
- Functions that are called on an *object*.



It's easy to call a standalone function. It's also easy to call a function on an object.

However, things become more difficult when you *combine* calls to these two different types of functions in one place.

Take a look at this code, which calls one standalone function (`singleQuoted()`), and calls two functions with dot notation (`removePrefix()` and `uppercase()`).

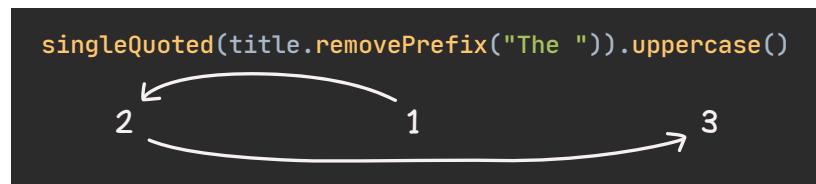
```
singleQuoted(title.removePrefix("The ")).uppercase()
```

Listing 10.4 - Calling three functions to change a string - two functions are on an object and another is a standalone function.

Can you figure out what order will these functions will be called in?

1. First, `removePrefix()` is called.
2. Then, the result from that call will be used as an argument to the `singleQuoted()` function.
3. Finally, `uppercase()` will be called on the string object that is returned from `singleQuoted()`.

Visually, our minds have to process this expression by bouncing around - starting in the middle, then moving to the left, then moving to the right.



Imagine trying to read a book like this!



It would be easier to read and understand the code if *all three* of these function calls worked the same way, so that we could read them in a single direction.

For example, if you could use *dot notation* to call the `singleQuoted()` function - just like you do with `removePrefix()` and `uppercase()` - then it would be very easy to follow. Here's what that would look like:

```
val newTitle = title.removePrefix("The ").singleQuoted().uppercase()
```

Listing 10.5 - Calling three functions, each of them is a function on an object.

Since `singleQuoted()` isn't a part of the `String` class, this code doesn't actually work yet. But you can certainly see how much easier it is to read and understand it, because the functions are called in the same order that you read them.

You can simply follow the code from *left to right*:

```
title.removePrefix("The ").singleQuoted().uppercase()
    1 → 2 → 3
```

These calls could also be arranged vertically, one per line, like this:

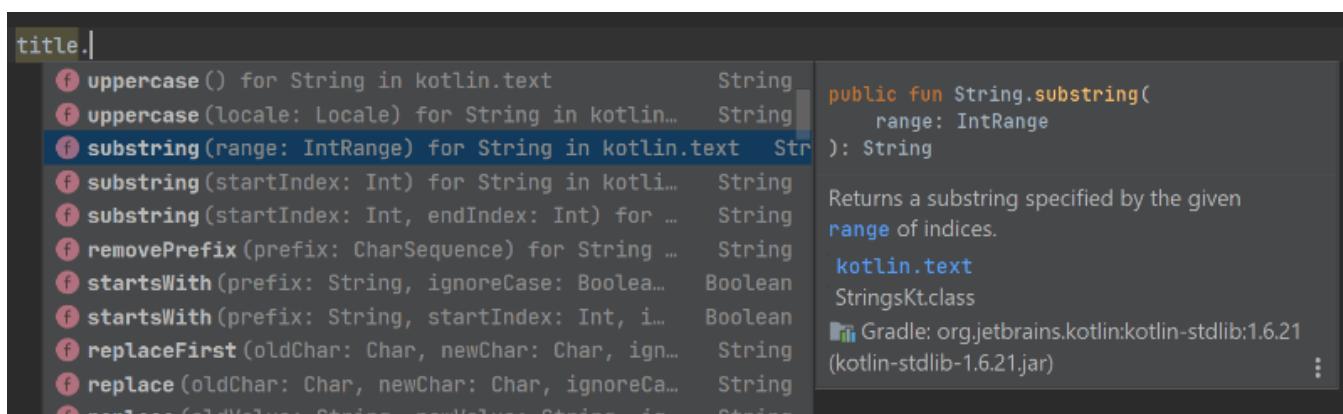
```
val newTitle = title
    .removePrefix("The ")
    .singleQuoted()
    .uppercase()
```

Listing 10.6 - Arranging a call chain vertically.

Again, it's natural to read - from top to bottom.¹

Besides making the function calls consistent and easy to read, there are times when dot notation just fits well with a Kotlin developer's expectations. By convention, if a function primarily does something *to* an object or *with* an object, then we often expect that function to exist on the object.

Also, if you're using an IDE like IntelliJ or Android Studio, functions that are "on an object" are easier to discover. If you've got a `String` object, and you wonder what functions can be called on it, just type the dot, and you'll see a list of the available functions! This is a great way to explore classes that you're not as familiar with.



So, in this chapter, our goal is to change `singleQuoted()` so that it can be called with a dot, like this:

```
val newTitle = title.singleQuoted()
```

Listing 10.7 - How we want the call site to look - calling `singleQuoted()` on an object instead of as a standalone function.

Let's start by looking more closely at the similarities and differences between standalone functions, and those that are called on an object.

They're Not So Different After All

These two categories of functions - standalone functions and functions that are called on an object - have more in common than you might think. Yes, the way that you have to *write* the code - that is, the **syntax** - to call the function is a little different in each case:

¹When the functions are called in the same order as you'd naturally read them (that is, left to right, top to bottom), developers often refer to this as a **fluent interface**. However, Martin Fowler and Eric Evans, who came up with that term, clarify that using chained function calls is only *part* of what makes an interface fluent. Read more thoughts about fluent interfaces from Martin Fowler here: <https://www.martinfowler.com/bliki/FluentInterface.html>

<code>singleQuoted(title)</code>		<code>title.uppercase()</code>
----------------------------------	--	--------------------------------

But in concept, they're actually very similar:

- They both start off with a string.
- They both return a new string that is based on the original string.

From that standpoint, it's almost as if each of these functions takes a `String` argument. The difference is only in *where you put that argument* when you call the function.

<code>singleQuoted(title)</code>		<code>title.uppercase()</code>
<code>title</code> argument	↑	↑ <code>title</code> argument

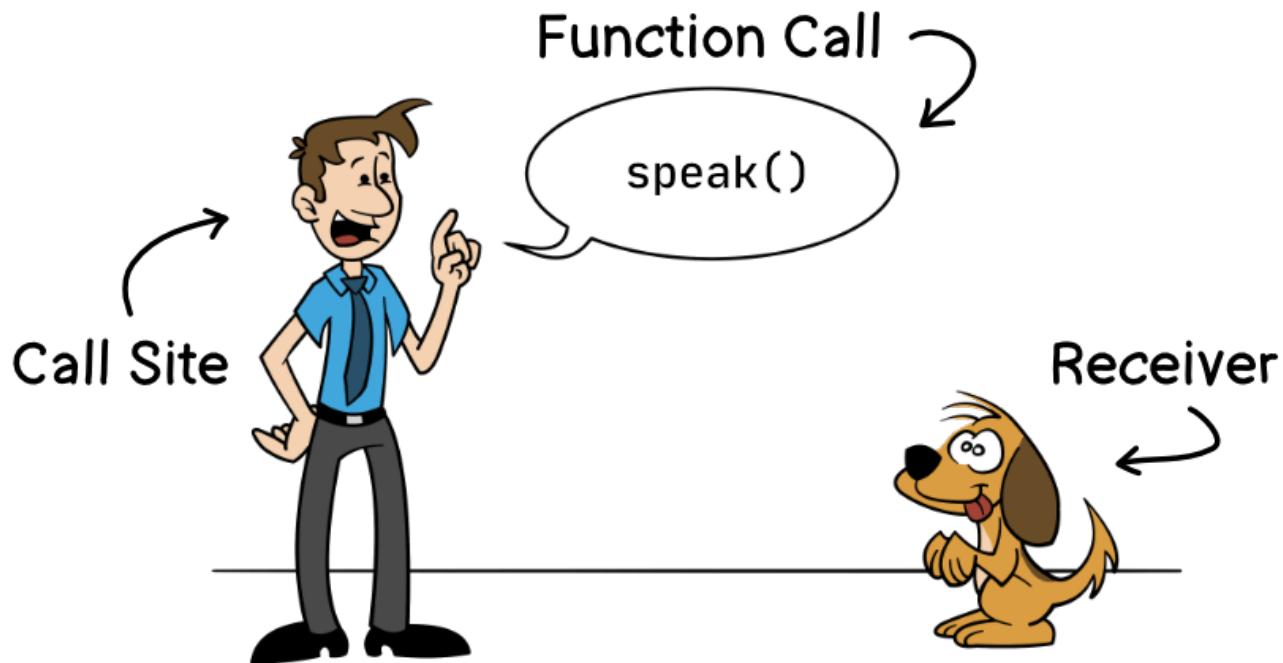
When calling a function using a dot, the *object to the left of the dot* is called the **receiver** of the function call. Receivers are an important concept for this chapter, but they're also important for understanding upcoming concepts like scope functions and more advanced lambdas, so let's dig in!

Introduction to Receivers

A well-trained dog knows how to bark on command. When you tell your dog Fido to "speak", you're sending him a command, and he is the receiver of that command.



Similarly, when you call a function on an object, that object is the receiver of that function call.



Let's flesh this out further with some code. Here's a simple Dog class, followed by some code to tell the dog to speak.

```
class Dog {  
    fun speak() {  
        println("BARK!")  
    }  
}  
  
val fido = Dog()  
fido.speak()
```

Listing 10.8 - A simple Dog class, with code telling it to speak.

Since `fido` is the dog you're telling to `speak()`, `fido` is the receiver. Easy, right?

Now, sometimes your dog doesn't need to be told to speak. Sometimes he will choose to bark on his own. (In fact, sometimes you can't get him to *stop* barking... ask me how I know!) Let's update the `Dog` class so that Fido will bark whenever he starts playing.

`fido.speak()`
↑
Receiver

```
class Dog {  
    fun speak() {  
        println("BARK!")  
    }  
    fun play() {  
        this.speak()  
    }  
}
```

Listing 10.9 - Adding a `play()` function to the Dog class.

Here, the `play()` function calls the `speak()` function. As you might recall from [Chapter 4](#), the keyword `this` refers to the same object that `play()` is called upon. In other words, if you call `fido.play()`, then `speak()` will be called on the `fido` object. In Listing 10.9, the receiver of the `speak()` function call is `this`.

You might also remember that you can omit `this.`, so the following code works the same as the code in the previous listing.

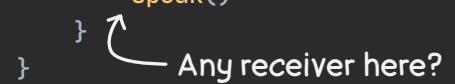
```
class Dog {
    fun speak() {
        println("BARK!")
    }
    fun play() {
        speak()
    }
}
```

Listing 10.10 - Omitting `this.` when calling `speak()` from inside `play()`.

Now, there's no object name or dot before `speak()`; just the function name. Does this mean that there's no receiver here?

In fact, there *is* a receiver here! Remember - any time that a function is called *on an object*, that object is the receiver. Because `speak()` is being called on a `Dog` object, that object is the receiver. Inside the `play()` function, you can include `this.` before `speak()`, or you can omit it. The result is the same either way, and the receiver is the same either way.

```
class Dog {
    fun speak() {
        println("BARK!")
    }
    fun play() {
        speak()
    }
}
```



So, `speak()` has a receiver here! It's just not *explicitly* stated in the code. It's *implied*.

That's why this is called an **implicit receiver**.

Contrast this with the **explicit receiver** in Listing 10.8 above. The following shows two call sites for `speak()` - one that's using an implicit receiver, and one that's using an explicit receiver.

Wow, that's a lot of information about receivers, but we can summarize it like this:

- A receiver is an object whose function you are calling.
- It can either be *explicit*, as seen when calling a function with a dot, or...
- It can be *implicit*, such as when one function calls another function inside the same class.

Now that we know about receivers, we can use this knowledge to get back to our original goal - updating the `singleQuoted()` function, so that we can call it with a dot.

```
class Dog {
    fun speak() {
        println("BARK!")
    }
    fun play() {
        speak()
    }
}
```




```
val fido = Dog()
fido.speak()
```



By the way: Sometimes you need this

As mentioned above, you can often omit this, but there are times where it's needed. If two functions have the same *signature* - that is, the same name, parameter types, and return type - then this can be helpful for choosing the right one. We'll see more examples of this in the future.

Introduction to Extension Functions

As it's currently written, the `singleQuoted()` function has a single parameter, called `original`, which is the string that will be wrapped with quotes. All we need to do now is to update the function so that it has a *receiver* instead of a normal function parameter.

What we have: `singleQuoted(title)`
↓
What we want: `title.singleQuoted()`

When you want to be able to call a function with a dot, one way to do this is to *add the function to the class*. However, you can't always do this. The `String` class is part of the Kotlin standard library, so you can't just open up its code and write a new function in it!

Thankfully, Kotlin provides a way to *extend* a class with your own functions, which can be called with a dot. These are called **extension functions**.

Let's look at the `singleQuoted()` function that we wrote way back at the beginning of this chapter.

```
fun singleQuoted(original: String) = "'$original'"
```

Listing 10.11 - The original function from Listing 10.1.

Let's change the `original` parameter to be the receiver, so that `singleQuoted()` will be an extension function. It's easy to do:

1. First, prefix the function name with the *type* of the receiver that you want, and add a dot. In this case, we want a receiver that's a `String`.
2. Second, refer to the receiver using `this` inside the function body.

Here's how `singleQuoted()` looks after making these changes:

```
fun String.singleQuoted() = "'$this'"
```

Listing 10.12 - An extension function that does the same thing as the previous listing.

In this code:

- `String` is the **receiver type**. By specifying this as `String`, you'll be able to call `singleQuoted()` on any object that is a `String`.
- `this` is the **receiver parameter**. It refers to whatever object `singleQuoted()` is called upon, so if you call `title.singleQuoted()`, then `this` will refer to the `title` object.

Receiver Type Receiver Parameter
↓ ↘
`fun String.singleQuoted() = "'$this'"`

You can easily convert a regular function to an extension function:

1. Put the type of the parameter before the function name, and add a dot.
2. Anywhere you used that parameter, rename it to `this`.
3. Finally, remove the original parameter from between the parentheses.

```
fun singleQuoted(original: String) = "'$original'"  
        ↓          ↓          ↓  
fun String.singleQuoted() = "'$this'"
```

With these changes, whenever you call this function, you *must* call it with a receiver, like this:

```
val quotedTitle = title.singleQuoted()
```

Listing 10.13 - How to call `singleQuoted()`, now that it's an extension function.

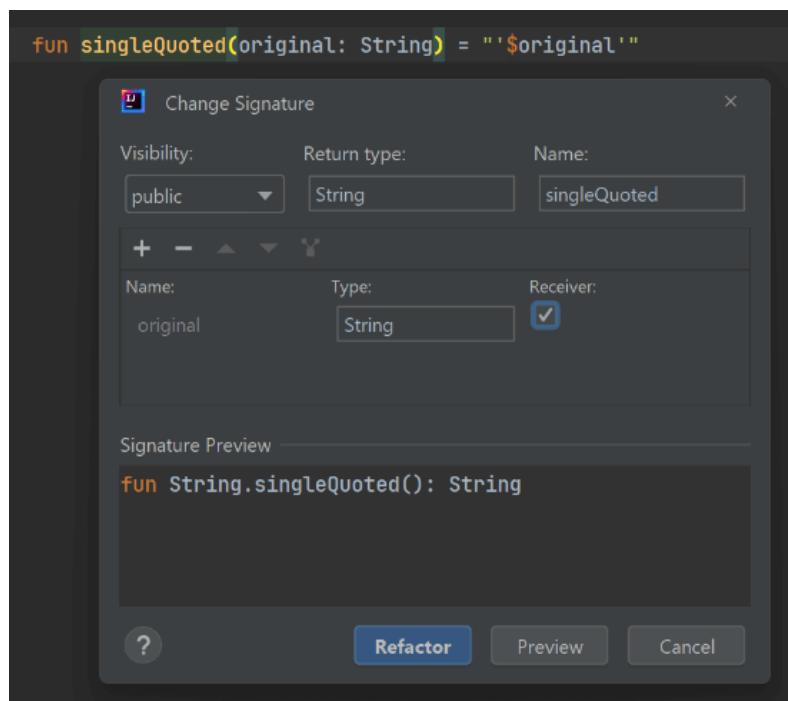
And now, it's easy to insert this function call into the middle of a call chain:

```
val title = "The Robots from Planet X3"  
val newTitle = title  
    .removePrefix("The ")  
    .singleQuoted()  
    .uppercase()  
  
// 'ROBOTS FROM PLANET X3'
```

Listing 10.14 - Adding a call to `singleQuoted()` in the middle of a call chain.

By the way, if you're using IntelliJ or Android Studio, you can also convert a function to an extension function by using the *refactoring* tools. To do this, right-click the function name, then choose "Refactor" and "Change Signature". From there, check mark the parameter that you want to convert to a receiver.

Extension functions are quite common in Kotlin code. Kotlin's standard library includes many extension functions, too. In fact, you might be surprised to learn that both `removePrefix()` and `uppercase()` are not actually members of the `String` class - they're extension functions, too!



Extensions are a great way to give an existing type some new functionality, especially for classes where you can't edit the class itself. Just keep in mind that extensions cannot access **private** members of a class. So, even though an extension function is *called* the same way as a member function, it doesn't have access to all of the same things that a [member function](#) does!

Nullable Receiver Types

What happens when you want to call an extension function on a nullable object? You'll get an error message.

```
val title: String? = null
val newTitle = title.singleQuoted()
```

Listing 10.15 - Error: "Only safe (?) or non-null asserted (!!.) calls are allowed on a nullable receiver of type String?"

As you might remember from [Chapter 6](#), you can work around this by using the [safe-call operator](#) (?) so that `singleQuoted()` is only called when `title` is *not* null.

```
val title: String? = null
val newTitle = title?.singleQuoted()
```

Listing 10.16 - Using a safe-call operator when the object is nullable.

Kotlin also gives you another option, though - you can create an extension function that has a *nullable receiver type*. For example, instead of making the receiver type a non-nullable `String`, you can make it a nullable `String?`, like this:

```
fun String?.singleQuoted() =
    if (this == null) "(no value)" else "'$this'"
```

Listing 10.17 - Creating an extension function for a nullable receiver type.

Inside this function, `this` is nullable. If this version of `singleQuoted()` is called on a null, then it returns a string that says `(no value)`. Otherwise, it works like the previous version of `singleQuoted()`, as in Listing 10.12.

When an extension function has a nullable receiver type, you don't *have* to call it with a safe-call operator. You can call it with a regular dot operator instead.

```
val title: String? = null
val newTitle = title.singleQuoted()

println(newTitle) // (no value)
```

Listing 10.18 - Calling an extension function that has a nullable receiver type using a normal dot operator.

On the other hand, you *could* still choose to call it with the safe-call operator if you want, but in that case, the function will *only* be called if the receiver is not null. For example, the only difference between the following listing and the previous listing is that we changed from a regular dot operator to the safe-call operator. The result is that `newTitle` is `null` rather than `(no value)`.

```
val title: String? = null
val newTitle = title?.singleQuoted()

println(newTitle) // null
```

Listing 10.19 - Calling an extension function that has a nullable receiver type using a safe-call operator.

So, choose carefully between a dot operator and a safe-call operator, based on your expectations.

Extension Properties

In addition to extension *functions*, you can also create extension *properties*. However, you can't use an extension property to actually store additional values inside a class. For example, it's not possible to add an ID number to a `String`. Still, they can be helpful for making small calculations.

Let's create an extension property that tells us if a `String` is longer than 20 characters.

```
val String.isLong: Boolean
    get() = this.length > 20
```

Listing 10.20 - A simple extension property.

Just as with an extension function, an extension property specifies the *receiver type*, and the *receiver parameter* is available as `this`.

As mentioned before, when you're calling a function or property on an *implicit receiver*, you don't need to include `this.`, so you could also write this property without it:

```
val String.isLong: Boolean
    get() = this.length > 20
```

```
val String.isLong: Boolean
    get() = length > 20
```

Listing 10.21 - Omitting `this.` inside the extension property.

You can use this property the same way as you'd use any property:

```
val string = "This string is long enough"
val isItLong = string.isLong
```

Listing 10.22 - Using an extension property.

By the way: What about Context Receivers?

You might have come across the term **context receiver**.

Context receivers allow you to take one or more *implicit receivers* at a function call site, and make them

available to the function that is being called. In practice, this means you can have a function that has *multiple* receiver parameters, rather than just one, as you have with extension functions.

At the time of writing, context receivers are a *prototype* feature, so it's quite possible they'll undergo some significant changes before they're ready for use in real Kotlin projects. For that reason, I'm not covering them here yet. Many of us in the Kotlin community are excited about them, though!

Summary

In this chapter, you learned:

- The difference between [standalone functions and object functions](#).
- All about explicit and implicit [receivers](#).
- How to create an [extension function](#).
- How to create an extension function that has a [nullable receiver type](#).
- How to create an [extension property](#).

In the next chapter, we'll learn about *Scopes and Scope Functions*. Kotlin developers use scope functions frequently, and in some cases, they can even be a helpful replacement for extension functions.

Kotlin: An Illustrated Guide

Chapter 11

Scopes and Scope Functions



Dot Notation Without Your Own Extensions

In the last chapter, we learned how to create extension functions, which can be called using dot notation.

In this chapter, we'll learn about the five scope functions, which are particular functions (four of which are extension functions) that Kotlin gives you out of the box. Before we can understand scope functions, though, it helps to first understand scopes.

Introduction to Scopes

In Kotlin, a **scope** is a section of code where you can declare new variables, functions, classes, and more. In fact, every time that you've declared a new variable, you've declared it inside of *some* scope. For example, when we create a new Kotlin file (one that ends in `.kt`) and add a variable to it, that variable is declared within the file's scope.

```
val pi = 3.14
```

Listing 11.1 - Declaring a variable in a top-level file scope.

In this case, the variable `pi` is declared within that file's top-level scope.

```
val pi = 3.14
```

Top-Level File Scope

When we declare a *class* in that same file, the body of that class creates *another scope* - one that is contained *within* the top-level scope of the file. Let's create a `Circle` class in that file, and add a `diameter` property to it.

```
val pi = 3.14

class Circle(var radius: Double) {
    val diameter = radius * 2
}
```

Listing 11.2 - Adding a `Circle` class to create another scope.

Now we can identify two scopes in this file:

1. The top-level file scope, where the `pi` variable and the `Circle` class are declared.
2. The class body scope of the `Circle` class, where `diameter` is declared.

We can add new things like variables, functions, or classes to either of these scopes.

```
val pi = 3.14
```

Top-Level File Scope

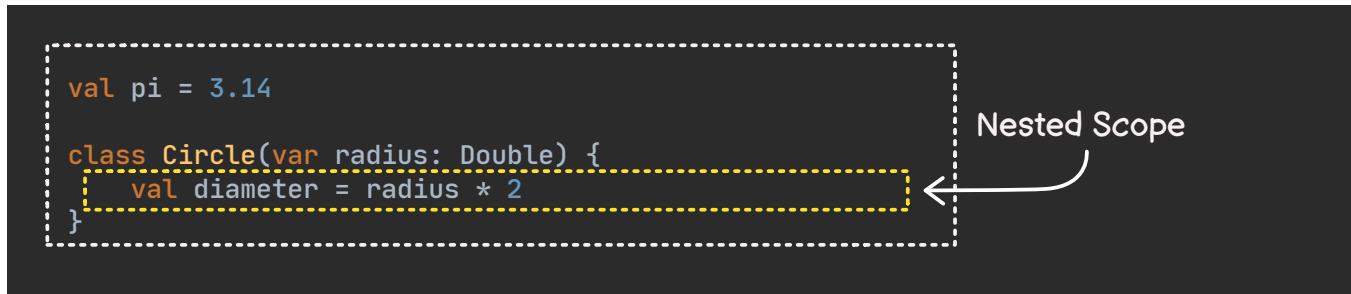
```
class Circle(var radius: Double) {
    val diameter = radius * 2
}
```

Class Body Scope

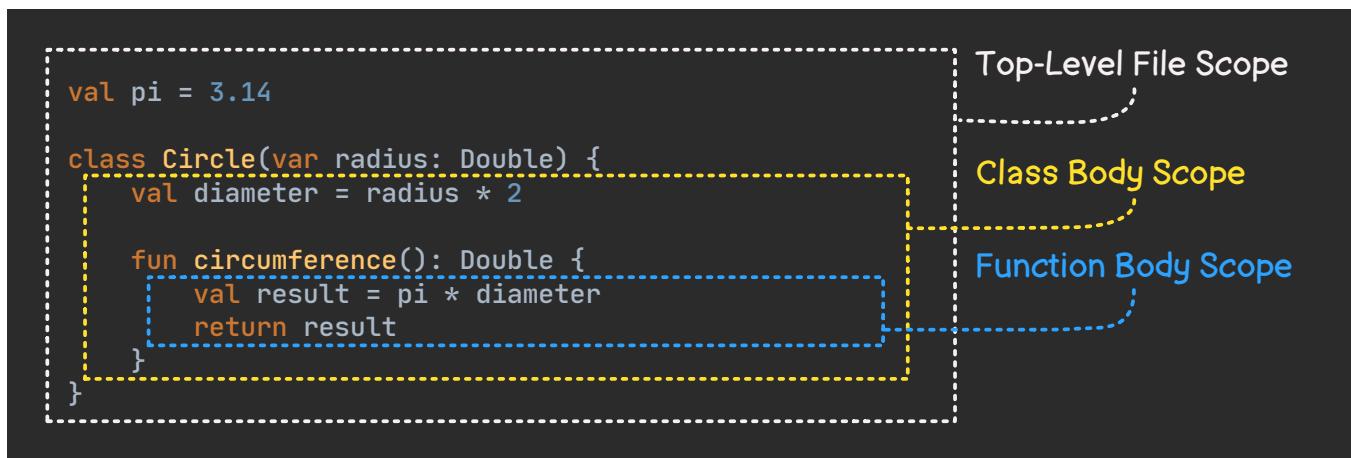
By the way: Parameter Scopes

Technically, there's a third scope here. The parameter list of the `Circle` constructor (where `radius` is declared) also has its own scope. For simplicity, we'll mostly ignore parameter scopes in this chapter.

When one scope is contained within another, we call it a **nested scope**. In the example above, the body of the `Circle` class is a scope that is *nested* within the scope of the file.

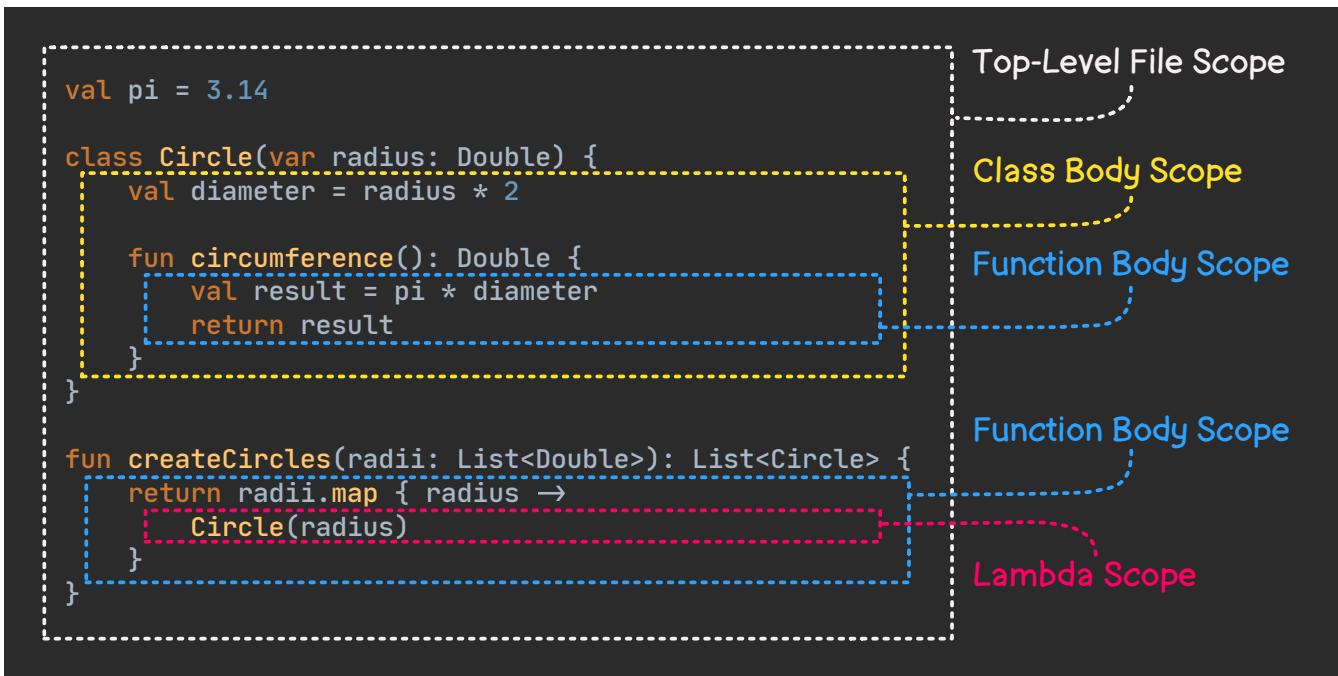


We can take it even further. If we add a function to that class, that function's body creates yet *another* a scope that's nested within the class scope, which is nested within the original scope!



This third scope is yet *another* place where we can add new variables, functions, and classes.

A new scope is also created when you write a [lambda](#). Let's add a new function that uses a lambda. This will add a scope for the function body and a scope for the lambda.



In the code above, we can now identify *five* scopes. We can add a new variable, function, or class within any one of them!¹

You might have noticed that, other than the outer-most scope, each scope here begins with an opening brace `{` and ends with a closing brace `}`. This isn't a hard-and-fast rule (for example, some functions have [expression bodies](#), and therefore no braces), but can serve as a helpful way to generally identify scopes. This also means that if we indent the code consistently, it makes it easier to tell where each new scope is being introduced.

One of the most important things about scopes is that they affect *where* you can use the things that are declared inside of them. Let's look at that next!

Scopes and Visibility

As you grow up speaking a language natively, you gradually develop a sense for the rules of that language. When you were a child, your parents might have gently corrected your grammar here and there, and over time, you were eventually able to *intuit* the rules, even if you couldn't always explain them to someone.



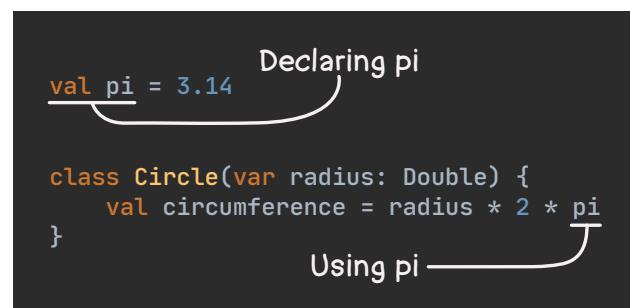
Similarly, as you've been writing Kotlin code, you've developed a sense for *when* you can use a particular variable, function, or class. **Visibility** is the term used to describe where in the code you can or cannot use something (such as a variable, function, or class). As with your native language, you can also *intuit* these visibility rules. However, you'll be a more productive Kotlin developer if you actually *know* what those rules are.

In a moment, we'll look at two kinds of scopes, and how they affect visibility. As you read this next section, keep in mind the difference between *declaring* something and *using* it. A variable, function, or class is [declared](#)

¹There are actually more than five scopes here. As mentioned above, parameter lists have their own scopes, but you'll typically only declare parameters there. Also, depending on whether a class contains other things like secondary constructors, enum classes, and companion objects, there could be a "static" scope. In order to stay focused on the main concepts of scopes, we'll ignore those for this chapter. If you're curious, you can read all about them in the *Declarations* chapter of the Kotlin language specification.

at the point where you introduce it with the `val`, `var`, `fun`, or `class` keyword. When you evaluate a variable, call a function, and so on, you're *using* it.

Visibility describes where in the code you can *use* something, and that is determined by where in the code that thing was *declared*. We'll see plenty of examples of this in the next sections of this chapter.



Statement Scopes

There are two different kinds of scopes in Kotlin, and the kind of the scope affects the visibility of the things declared inside of it. Let's start with **statement scopes**, which are easiest to understand. The visibility rule for a statement scope is simple:

You can only use something that was declared in a statement scope *after* the point where it was declared.

For example, a function body has a statement scope. Inside that function body, if you try to use a variable that hasn't yet been declared, you'll get a compiler error. In the following code, we declare a `diameter()` function inside the `circumference()` function, but try to use it *before* it was declared.

```

class Circle(val radius: Double) {
    fun circumference(): Double {
        val result = pi * diameter()
        fun diameter() = radius * 2
        return result
    }
}

```

Listing 11.3 - Error: Unresolved reference: diameter

We can't call the `diameter()` function at this point in the code, because the function is declared later in the code (that is, on the next line) inside this statement scope.

To correct this error, we just need to move the line that declares `diameter()` so that it comes *before* the line that uses it.

```

class Circle(val radius: Double) {
    fun circumference(): Double {
        fun diameter() = radius * 2
        val result = pi * diameter()
        return result
    }
}

```

Listing 11.4 - Declaring the diameter() function before using it in a statement scope.

So, something that is declared inside a statement scope can only be used *after* the point that it was declared. Easy!

As we saw here, a function body is one example of a statement scopes. Other examples include constructor bodies, lambdas, and Kotlin Script files (when your file ends in `.kts`).

Declaration Scopes

A second kind of scope in Kotlin is called a **declaration scope**. Unlike statement scopes, things declared within a declaration scope can be used from a point in the code *either before or after* that declaration.

A class body is an example of a declaration scope. We can update the `Circle` class from the previous listing so that the `diameter()` function is declared in the class body (a declaration scope) instead of the `circumference()` function body (a statement scope). We'll also change `circumference` to a property to make that line similar to the `result` assignment in Listing 11.3.

```
class Circle(val radius: Double) {  
    val circumference = pi * diameter()  
    fun diameter() = radius * 2  
}
```

Listing 11.5 - Using the `diameter()` function before declaring it in a declaration scope.

Now, even though `diameter()` is declared *after* `circumference()`, everything compiles and runs just fine.

So, in declaration scopes, things can be used either before or after the point where they are declared.

A notable exception to this rule is that variables and properties that are declared *and* used in the same declaration scope must still be declared *before* they are used.

For example, if we were to simply change both `circumference()` and `diameter()` to properties without changing their order, we'll get an error from the compiler.

```
class Circle(val radius: Double) {  
    val circumference = pi * diameter  
    val diameter = radius * 2  
}
```

Listing 11.6 - Error: Variable `diameter` must be initialized.

As we saw, a class body is one example of a declaration scope. The top level of a regular Kotlin file (when it ends in `.kt`) is another example.

Nested Scopes and Visibility

In general, if you want to know what variables, functions, and classes are available to you inside a particular scope, you can "crawl your way out" from that scope, toward the outermost scope - the one at the file level. As you're crawling:

1. If you crawl into a statement scope, you can only use things declared earlier in the scope.

2. However, if you crawl into a declaration scope, you can use things declared either earlier or later in the scope.

Let's demonstrate how this works. We'll start with a file that has the code in Listing 11.7 on the next page.

Which variables are visible at the comment?

To answer that question, we can start by identifying which of the scopes in this listing are *statement scopes* and which are *declaration scopes*.

Remember...

- Function bodies and lambdas have a statement scope.
- Class bodies and the file itself have declaration scopes.

```
val pi = 3.14

fun main() {
    val radii = listOf(1.0, 2.0, 3.0)

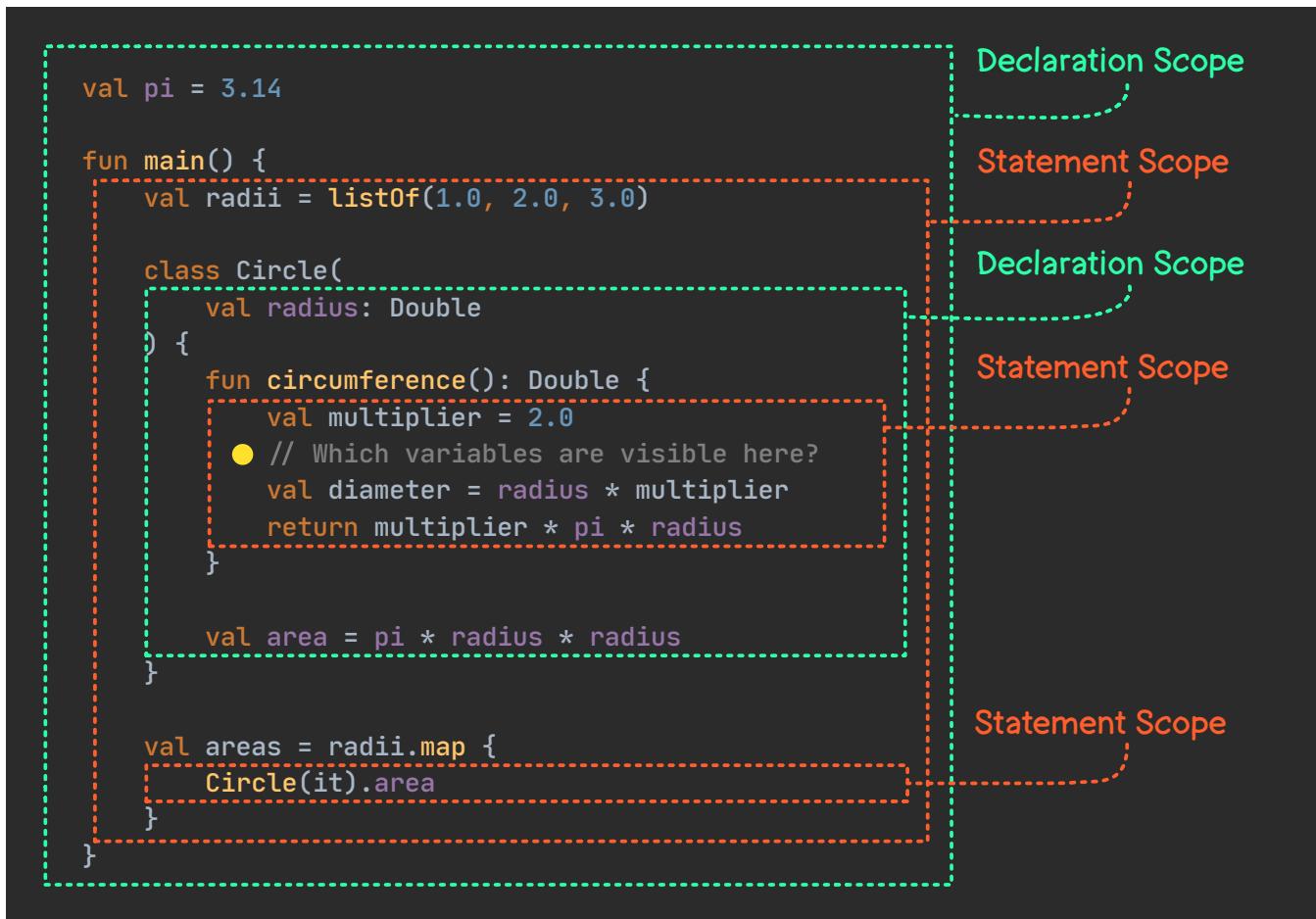
    class Circle(
        val radius: Double
    ) {
        fun circumference(): Double {
            val multiplier = 2.0
            // Which variables are visible here?
            val diameter = radius * multiplier
            return multiplier * pi * radius
        }

        val area = pi * radius * radius
    }

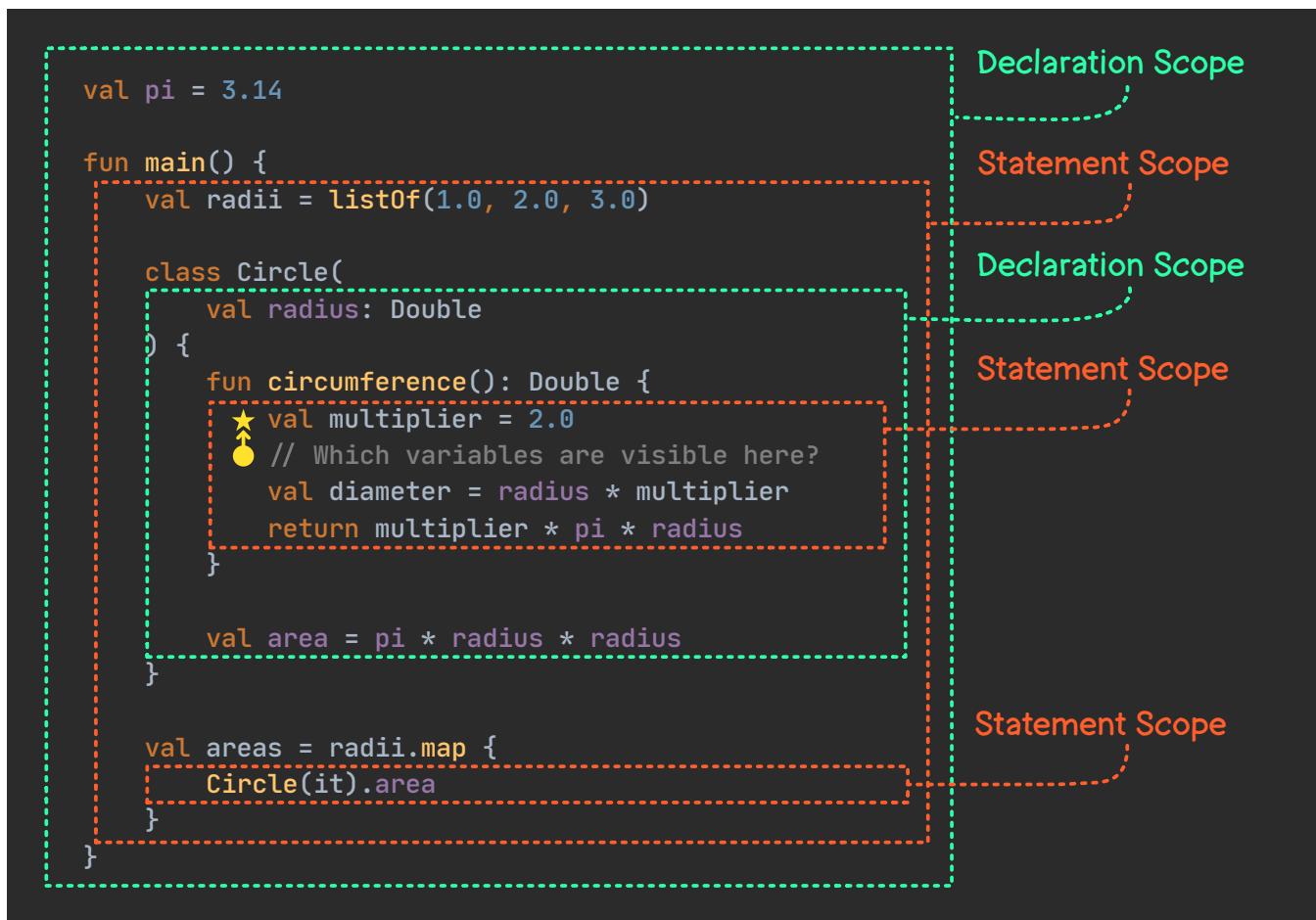
    val areas = radii.map {
        Circle(it).area
    }
}
```

Listing 11.7 - Code that will be used to demonstrate scope crawling.

The following illustrates the different declaration and statement scopes in the code. We're starting at the yellow dot, which we'll call the *starting point*, and our goal is to figure out which variables are visible at that point in the code.

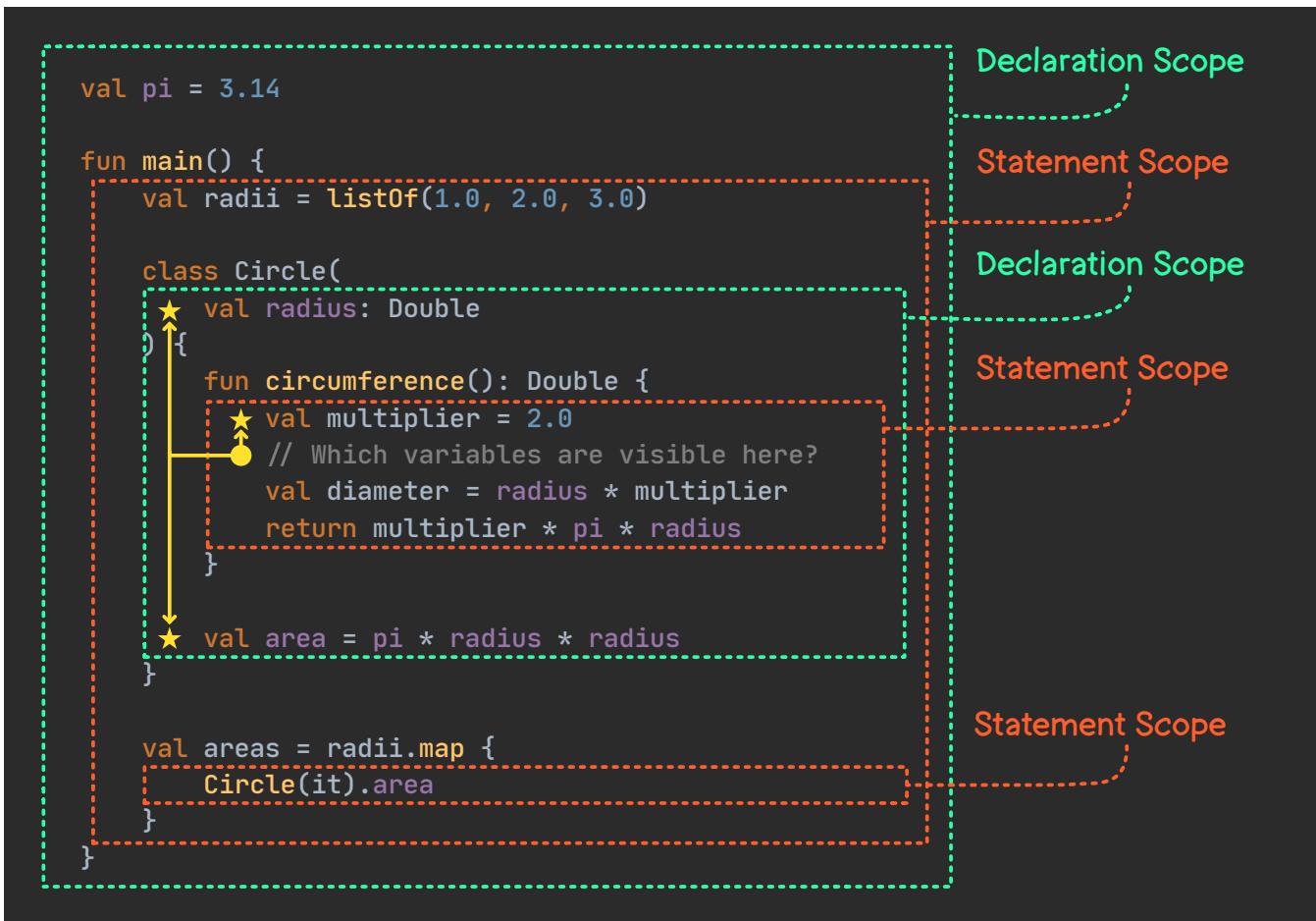


The starting point is inside a *statement scope*. Because things declared in a statement scope can only be used after they have been declared, we start by scanning only upward - not downward.



When we scan upward, we run into the `multiplier` variable. So, the `multiplier` variable is visible at the starting point. The `diameter` variable, however, is not visible, because it's declared *after* the starting point.

Having scanned this statement scope, we're now ready to crawl into the next scope *outward* - that is, we crawl into the scope that contains the `circumference()` function. This is a class body, which has a *declaration scope*. With a declaration scope, we scan both upward *and* downward.



Scanning in both directions, we come across both the `radius` parameter, which is declared *before* the function, and `area`, which is declared *after* the function. So, both of these variables are also visible at the starting point.

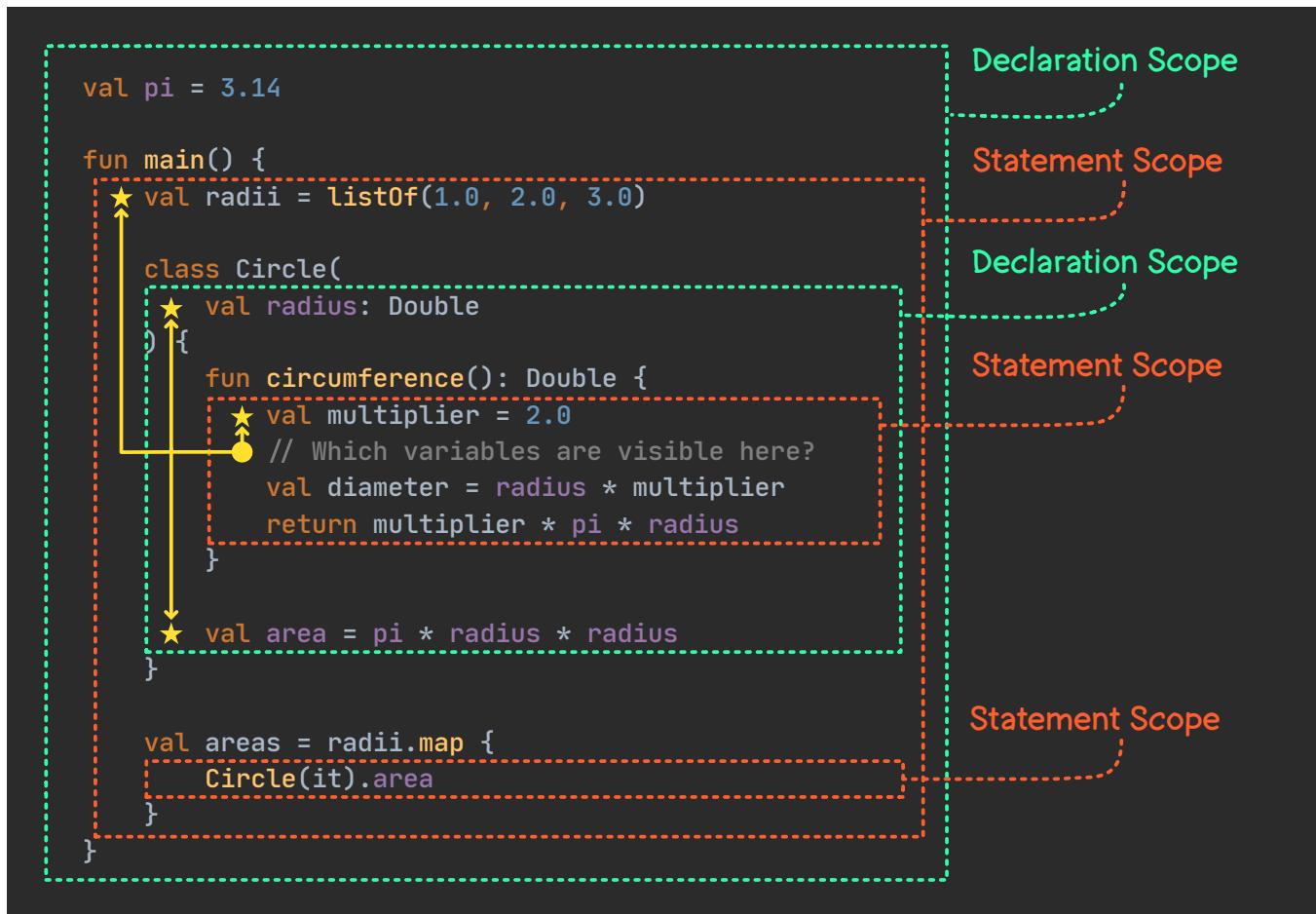
By the way: Parameter Scopes, Again!

For simplicity, I'm treating the parameter lists (such as the one containing the `radius` constructor parameter here) as part of the function or class body that they pertain to. Technically, a parameter list has its own scope, which is actually *adjacent* to its corresponding function or class body scope.

However, these parameter scopes are **linked** to their body scope, which makes the parameters visible inside that body.

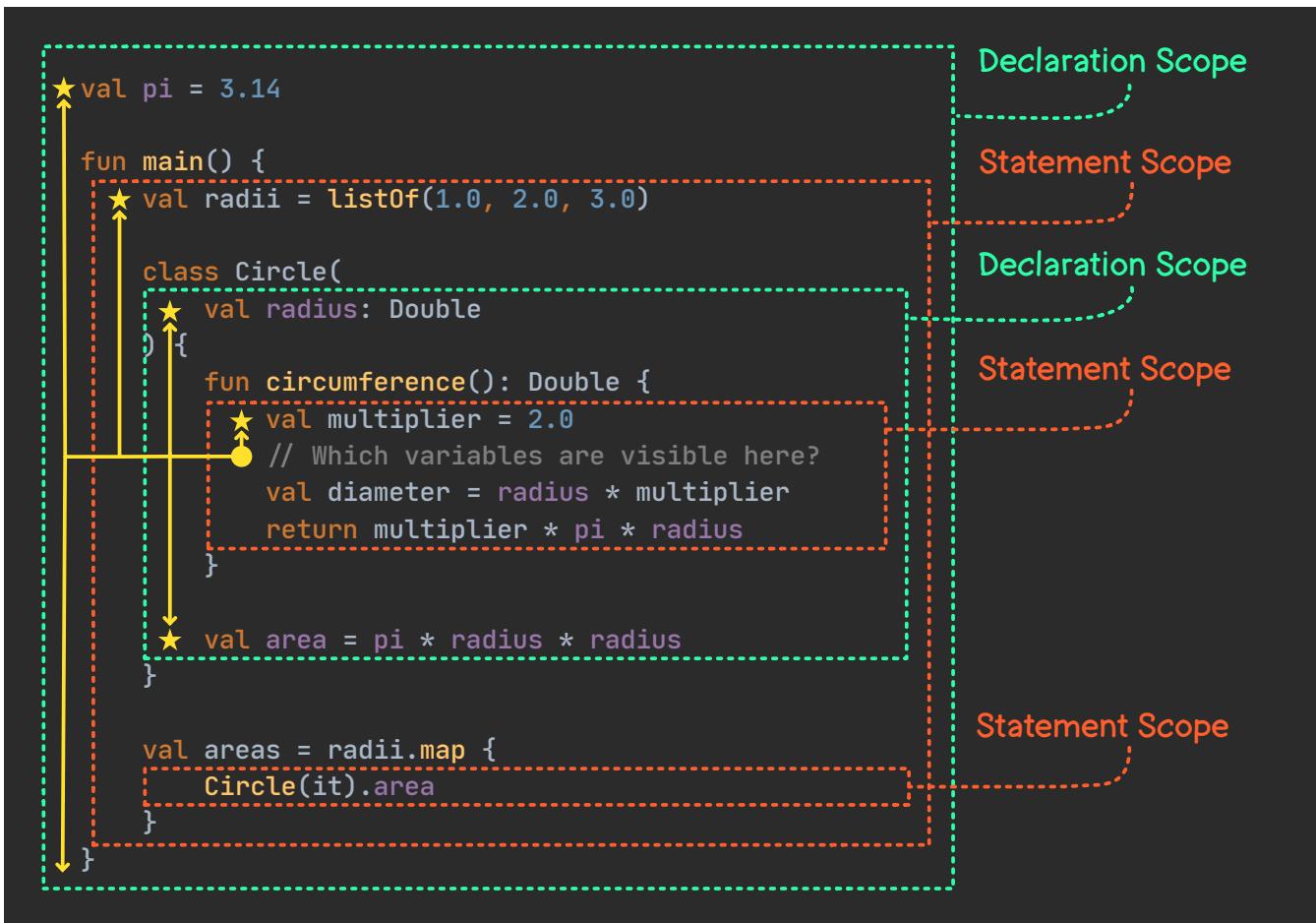
We'll see other examples of linked scopes in the future as we explore *inheritance* in the upcoming chapters.

Next, we crawl out to the containing scope - that is, the scope of the `main()` function. Because this is a function body (and therefore has a statement scope), we crawl only upward.



Scanning upward, we run into the `radii` variable, which is also visible at the starting point. However, the `areas` variable is not visible, because it's below.

Finally, we crawl out to the top-level file scope. Files have a declaration scope, so we scan both directions.



The `pi` variable is found when scanning upward, and there are no variables found when scanning downward in this scope.

So, the answer to our question, “which variables are visible here?” is:

- `pi`
- `radii`
- `radius`
- `multiplier`
- `area`

Although we concerned ourselves only with variables here, note that the same scanning approach works for other things as well, such as functions and classes. So, at the starting point, you can also:

- ... call the `main()` function.
- ... instantiate a new `Circle` object.
- ... call the `circumference()` function (from inside itself).

Again, you’ve probably developed an intuition around most of these rules. And as you write Kotlin from day to day, you’ll normally just rely on the compiler and IDE (such as IntelliJ or Android Studio) to tell you whether

something is visible to you at a particular point in code. Still, it's helpful to know the rules, so that you can structure your code the right way, ensuring that each thing has the visibility you want it to have!

Best Practice

It's usually a good idea to give things the *least* amount of visibility necessary. This is especially true for variables that are declared with the `var` keyword. When these kinds of variables are declared at the top-level scope, they can be evaluated and assigned from *anywhere* in your code. This can make it difficult to know when and why the value is changing as your code runs.

By limiting its visibility, there are fewer places in the code that are able to use or change the value. Troubleshooting becomes much easier when you have less code to sift through!

Now that we've got a solid understanding of scopes and visibility, we're ready to dive into *scope functions*!

Introduction to Scope Functions

There are five functions in Kotlin's standard library that are designated as scope functions. Each of them is a [higher-order function](#) that you typically call with a lambda, which introduces a new statement scope. The point of a scope function is to take an existing object - called a **context object**¹ - and represent it in a particular way inside that new scope.

Let's start with a simple example - a scope function called `with()`.



with()

When you need to use the same variable over and over again, you can end up with a lot of duplication. For example, suppose we need to update an address object.

```
address.street1 = "9801 Maple Ave"
address.street2 = "Apartment 255"
address.city = "Rocksteady"
address.state = "IN"
address.postalCode = "12345"
```

Listing 11.8 - Updating many properties of an address object.

When *writing* this code, it's tedious to type `address` on each line, and when *reading* this code, seeing `address` on each line doesn't really make it any easier to understand. This duplication actually *detracts* from the important thing on each line - the property that is being updated.

¹The term **context object** is used in the official Kotlin documentation for scope functions, so I'm using it here as well. If you're an Android developer, this could be confusing, since Android has a specific `Context` class. Keep in mind that these are two entirely different concepts. You can use a scope function with any object.

We can use the `with()` scope function to introduce a new scope where the `address` becomes an [implicit receiver](#). Here's how it looks:

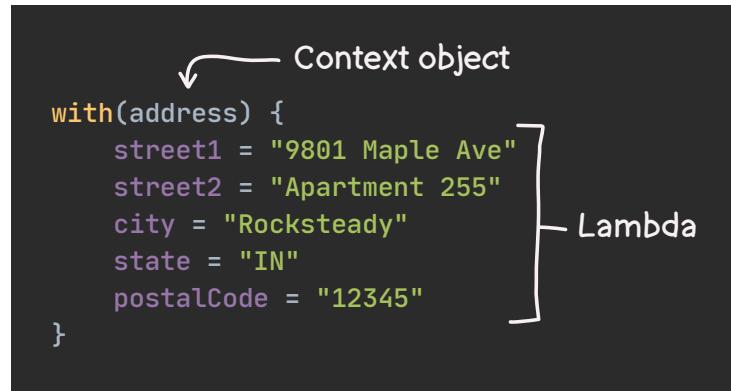
```
with(address) {  
    street1 = "9801 Maple Ave"  
    street2 = "Apartment 255"  
    city = "Rocksteady"  
    state = "IN"  
    postalCode = "12345"  
}
```

Listing 11.9 - Using `with()` so that each property assignment does not need to be prefixed with `address`.

The `with()` function is called with two arguments:

1. The object that you want to become an implicit receiver. This is the *context object*. Here, it's `address`.
2. A lambda in which the context object will be the implicit receiver.

As you probably recall from the [last chapter](#), an implicit receiver can be used with no variable name at all, so inside the lambda above, we can assign values to `street1`, `street2`, and so on, without prefixing the name of the variable as we had to do in Listing 11.8.



So again, the `with()` scope function introduces a new scope (the lambda) in which the context object is represented as an implicit receiver.

The remaining four scope functions are all [extension functions](#). Next, let's look at one called `run()`, because it's very similar to `with()`.

run()

The `run()` function works the same as `with()`, but it's an extension function instead of a normal, top-level function.¹ This means we'll need to invoke it with [dot notation](#). Let's rewrite Listing 11.9 so that it uses `run()` instead of `with()`.

Other than the first line, you'll notice that this code listing is identical to the previous one. The context object is passed as a receiver instead of a regular function argument, but the lambda is the same.

¹There's actually also a top-level function called `run()`, which can be helpful when you need to squeeze multiple statements into a spot where Kotlin expects a single expression. We're not going to cover that version of the function in this chapter, though.

```
address.run {
    street1 = "9801 Maple Ave"
    street2 = "Apartment 255"
    city = "Rocksteady"
    state = "IN"
    postalCode = "12345"
}
```

Listing 11.10 - Using run() so that each property assignment does not need to be prefixed with address.



Even though `run()` and `with()` are very similar, `run()` does have some different characteristics because it's an extension function. For instance, we saw in the last chapter how extension functions can be inserted into a call chain.

In fact, instead of defining your own extension functions for use in a call chain, you can often use a scope function like `run()`.

For example, in the last chapter, we created a very simple extension function called `singleQuoted()` (in Listing 10.12), and called it in the middle of a call chain (in Listing 10.14), like this:

```
val title = "The Robots from Planet X3"
val newTitle = title
    .removePrefix("The ")
    .singleQuoted()
    .uppercase()

// 'ROBOTS FROM PLANET X3'
```

Listing 11.11 - Code assembled from listings in Chapter 10.

Because `singleQuoted()` is so simple (it's just a single expression!) we can remove the `singleQuoted()` function entirely, and replace it with a simple call to `run()`, like this:

```
val title = "The Robots from Planet X3"
val newTitle = title
    .removePrefix("The ")
    .run { '''$this''' }
    .uppercase()

// 'ROBOTS FROM PLANET X3'
```

Listing 11.12 - Replacing the `singleQuoted()` extension function with a call to the `run()` function.

The `run()` function returns the result of its lambda, so the code in this listing works identically to Listing 11.11 above. Of course, if you need to make a string single quoted in lots of places in your code, you'd be better off sticking with the `singleQuoted()` extension function. That way, if you need to change the way it works, you can fix it in one spot instead of lots of places. If you've only got a single call site, though, a scope function can be a good option!

Another advantage of using `run()` instead of `with()` is that you can use the [safe-call operator](#) to handle cases where the context object might be null. We'll look at this more closely toward the end of this chapter.

For now, the important things to remember about `run()` are that:

1. Inside the lambda, the context object is represented as the *implicit receiver*.
2. The `run()` function returns the result of the lambda.

Next, let's take a look at another scope function, called `let()`.

Context object is
implicit receiver

Returns result
of lambda

`obj.run { }`

let()

`let()` might be the most frequently-used scope function. It's very similar to `run()`, but instead of representing the context object as an *implicit receiver*, it's represented as the *parameter* of its lambda.

Let's rewrite the previous listing to use `let()` instead of `run()`.

```
val title = "The Robots from Planet X3"
val newTitle = title
    .removePrefix("The ")
    .let { titleWithoutPrefix -> "'$titleWithoutPrefix'" }
    .uppercase()

// 'ROBOTS FROM PLANET X3'
```

Listing 11.13 - Single-quoting a title with the `let()` function.

This is very similar to the previous listing, but instead of using `this`, we used a lambda parameter called `titleWithoutPrefix`.

This parameter name is pretty long. Let's change it to use the [implicit it](#), so that it will be nice and concise.

```
val title = "The Robots from Planet X3"
val newTitle = title
    .removePrefix("The ")
    .let { ''$it'' }
    .uppercase()

// 'ROBOTS FROM PLANET X3'
```

Listing 11.14 - Single-quoting a title with the `let()` function, using the implicit `it` lambda parameter.

As with `run()` and `with()`, the `let()` function returns the result of the lambda.

Context object is implicit receiver	Context object is lambda parameter
Returns result of lambda	<code>obj.run { }</code>

By the way: `let()` vs `map()`

In some ways, the `let()` function is similar to the `map()` function that we learned about in [Chapter 8](#) when we looked at collection operations. However, whereas the `map()` function maps each *element* in the receiver, the `let()` function maps the receiver itself.

A scope function that's similar to `let()` is called `also()`. Let's look at that next.

`also()`

As with `let()`, the `also()` function represents the context object as the lambda parameter, too. However, unlike `let()`, which returns the result of the lambda, the `also()` function returns the *context object*. This makes it a great choice for inserting into a call chain when you want to do something "on the side" - that is, without changing the value at that point in the chain.

For example, we might want to print out the value at some point in the call chain. Here's the code from Listing 11.11, with `also()` inserted after the call that removes the prefix.

```
val title = "The Robots from Planet X3"
val newTitle = title
    .removePrefix("The ")
    .also { println(it) } // Robots from Planet X3
    .singleQuoted()
    .uppercase()

// 'ROBOTS FROM PLANET X3'
```

Listing 11.15 - Using also() in a call chain to print out a value.

The `also()` call here prints out the result of `title.removePrefix("The ")`, without interfering with the rest of the call chain. Regardless of whether we include or omit the line with the `also()` call, the `singleQuoted()` call will be called upon the same value - `"Robots from Planet X3"`.

By the way, as you might remember from [Chapter 7](#), you can use a [function reference](#) instead of a lambda, so we could choose to write the previous code listing like this:

```
val title = "The Robots from Planet X3"
val newTitle = title
    .removePrefix("The ")
    .also(::println) // Robots from Planet X3
    .singleQuoted()
    .uppercase()

// 'ROBOTS FROM PLANET X3'
```

Listing 11.16 - Using also() with a function reference.

Here's how `also()` fits in among `run()` and `let()`:

	Context object is implicit receiver	Context object is lambda parameter
Returns result of lambda	<code>obj.run { }</code>	<code>obj.let { }</code>
Returns context object		<code>obj.also { }</code>

As you can see, we've roughly created a chart, and there's one spot that's empty. Let's fill in that last spot as we look at the final scope function, `apply()`.

apply()

Like `also()`, the `apply()` function returns the *context object* rather than the result of the lambda. However, like `run()`, the `apply()` function represents the context object as the *implicit receiver*. We can update Listing 11.15 to use `apply()` instead of `also()`, and it would do the same thing.

```
val title = "The Robots from Planet X3"
val newTitle = title
    .removePrefix("The ")
    .apply { println(this) } // Robots from Planet X3
    .singleQuoted()
    .uppercase()

// 'ROBOTS FROM PLANET X3'
```

Listing 11.17 - Using `apply()` to print out the value in a call chain. This works, but most Kotlin developers favor `also()` for this situation.

However, in practice, Kotlin developers would typically prefer to use `also()` in this case. The `apply()` function really shines when you want to customize an object after you construct it. For example, after you call a constructor, you might want to set some other property on that object, or call one of its functions to initialize it - that is, to make the object ready for use.

```
val dropTarget = DropTarget().apply {
    addDropTargetListener(myListener)
}
```

Listing 11.18 - Using `apply()` to initialize a `DropTarget` immediately after constructing it. It's constructed and initialized in a single expression.

With this, we can fill out the remaining spot on the chart:

As you can see, the scope functions are all similar, but they differ on two things:

1. How they refer to the context object.
2. What they return.

	Context object is implicit receiver	Context object is lambda parameter
Returns result of lambda	<code>obj.run { }</code>	<code>obj.let { }</code>
Returns context object	<code>obj.apply { }</code>	<code>obj.also { }</code>

I've omitted `with()` from this chart, because it's the same thing as `run()`, except that it's a traditional function instead of an extension function.

With all these scope functions to choose from, how do you know which one to use?

Choosing the Most Appropriate Scope Function

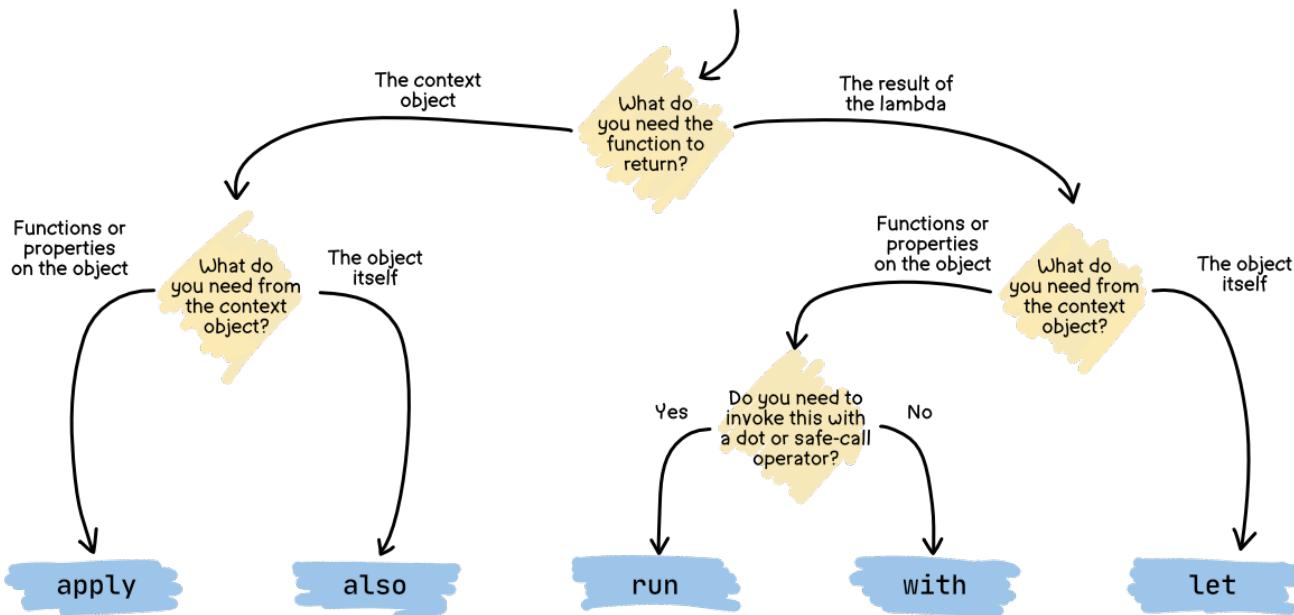
When deciding which scope function to use, start by asking yourself, “what do I need the scope function to return?”.

1. If you need the lambda result, narrow your options down to either `let()` or `run()`.
2. If you need the context object, narrow your options down to either `also()` or `apply()`.

After that, choose between the remaining two options based on your preference for how to represent the context object inside the lambda. If you need to use functions or properties on the object but *not* the object itself, then `run()` or `apply()` would probably be a good fit. Otherwise, `let()` or `also()` generally would be a good way to go.

You can use the flowchart on the next page to help you decide which scope function to use in different situations.

WHICH KOTLIN SCOPE FUNCTION SHOULD I USE?



One other caveat - it's possible to create a variable or lambda parameter with the same name as a variable from an outer scope, and it's usually best to avoid that. Let's look at that next.

Shadowing Names

When a nested scope declares a name for a variable, function, or class that's also declared in an outer scope, we say that the name in the outer scope is **shadowed** by the name in the inner scope. Here's very simple a simple example, where both a book and a chapter have a `title`.

```
class Book(val title: String) {
    fun printChapter(number: Int, title: String) {
        println("Chapter $number: $title")
    }
}
```

Inside the function body here -
this property is shadowed by
this parameter because they
have the same name.

It's perfectly valid to shadow names like this, but there are a few things to keep in mind.

First, when you read code like this, it's possible to get confused, thinking that you're referring to the name from the *outer* scope.

Second, it is more difficult - and sometimes impossible - to refer to a variable that's declared in an outer scope from an inner scope that has shadowed that variable's name.

Sometimes there are solutions - in the example above, you could still refer to the book's title with `this.title`. In cases when the shadowed variable is at the top level, you can refer to it by prefixing it with the package name. But in some cases, your only option might be to rename one of the two names.

So in general, it's best to avoid shadowing.

Shadowing and Implicit Receivers

An interesting form of shadowing happens when an *implicit receiver* is shadowed by the *implicit receiver* of a nested scope! And it works differently depending on whether you include or omit the `this` prefix!

For example, let's say we've got classes and objects for a `Person` and a `Dog`.

```
class Person(val name: String) {
    fun sayHello() = println("Hello!")
}

class Dog(val name: String) {
    fun bark() = println("Ruff!")
}

val person = Person("Julia")
val dog = Dog("Sparky")
```

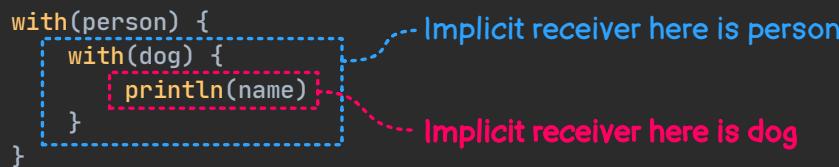
Listing 11.19 - Two classes, and two objects, which will be used to demonstrate shadowing of implicit receivers.

Now, we can shadow the implicit receiver if we nest one `with()` call inside another `with()` call, like this:

```
with(person) {  
    with(dog) {  
        println(name)  
    }  
}
```

Listing 11.20 - Shadowing the `person` implicit receiver with the `dog` implicit receiver.

In the outer scope, the implicit receiver is `person`, but in the inner scope, the implicit receiver is `dog`:



As you'd probably guess, `name` inside that innermost scope refers to the `name` of the `dog` object, so it's Sparky. You can also call `bark()` on that object.

```
with(person) {  
    with(dog) {  
        println(name) // Prints Sparky from the dog object  
        bark()        // Calls bark() on the dog object  
    }  
}
```

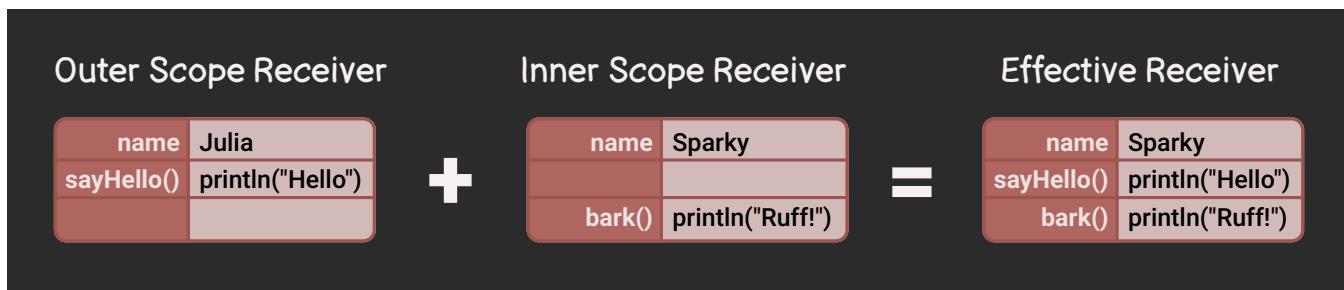
Listing 11.21 - Invoking a property and a function on the `dog` implicit receiver.

But here's the fun part - In that same scope, you can also call `sayHello()` on the `person` object without explicitly prefixing it with `person`!

```
with(person) {  
    with(dog) {  
        println(name) // Prints Sparky from `dog`  
        bark()        // Calls bark() on `dog`  
        sayHello()    // Calls sayHello() on `person`  
    }  
}
```

Listing 11.22 - Invoking a function on the `person` implicit receiver from a scope where `dog` is the primary implicit receiver.

So, in this example, both `person` and `dog` contribute to the implicit receiver in that innermost scope. You can visualize it like this:



In other words, the effective implicit receiver becomes a combination of *all* implicit receivers from the innermost scope to the outermost scope. When a name conflict exists (for example, both **Person** and **Dog** have a **name** property), priority is given to the inner scope.

Shadowing, Implicit Receivers, and this

Now, all of that is true when you use the implicit receiver *without* the `this` keyword. However, if you refer to the implicit receiver *with* the prefix `this`, it will *only* have the functions and properties from the implicit receiver of the innermost scope.

In the example above, this means that `this` will refer to the `dog` *without* any contributions from the `person` object.

To demonstrate this, let's try adding `this.` before `name`, `bark()`, and `sayHello()`:

```
with(person) {  
    with(dog) {  
        println(this.name) // Prints Sparky from `dog`  
        this.bark()        // Calls bark() on `dog`  
        this.sayHello()    // Compiler error here!  
    }  
}
```

Listing 11.23 - Error: Unresolved reference: sayHello

As you can see, it works fine for `this.name` and `this.bark()`, but `this.sayHello()` gives us an error, because `this` only refers to the dog.

So, just remember:

1. When using **this**, it will *only* refer to the exact implicit receiver in that scope.
 2. When omitting **this**, the effective receiver is a combination of the implicit receivers, from the innermost to the outermost scope.

Before we wrap up this chapter, let's look at how scope functions are used with null-safety features in Kotlin.

Scope Functions and Null Checks

Other than `with()`, all of the scope functions are extension functions. As with all extension functions, you can use the [safe-call operator](#) when calling them, so that they're only actually called if the receiver is not null, as we saw in the [previous chapter](#).

The safe-call operator is often used with scope functions. In fact, many Kotlin developers use `let()` with the safe-call operator to run a small block of code whenever the object is not null. For example, when we first learned about nulls in [Chapter 6](#), we needed to make sure the code would only order coffee when the customer had a payment. Here's a code snippet inspired by Listing 6.19 from that chapter.

```
if (payment != null) {
    orderCoffee(payment)
}
```

Listing 11.24 - Using a conditional to call `orderCoffee()` only when `payment` is present, roughly copied from Chapter 6.

There's absolutely nothing wrong with writing the code this way. However, it's also common for Kotlin developers to write this code with a scope function and safe-call operator, like this:

```
payment?.let { orderCoffee(it) }
```

Listing 11.25 - Using a scope function to call `orderCoffee()` only when `payment` is present.

It's good to be able to recognize both ways of expressing this. The second way is especially helpful when you need to insert it into a call chain, of course.

In some cases, you might also have an `else` with your conditional, like this:

```
if (payment != null) {
    orderCoffee(payment)
} else {
    println("I can't order coffee today")
}
```

Listing 11.26 - A simple `if-else` conditional that checks for nulls.

To get the same effect with a scope function, you can use an [elvis operator](#) to express the `else` case, like this:

```
payment?.let { orderCoffee(it) }
?: println("I can't order coffee today")
```

Listing 11.27 - Rewriting the `if-else` null-check conditional so that it uses a scope function.

Good ol' fashioned `if/else` conditionals are easy to understand for most developers, though, so consider starting there, only using the scope function / safe-call / elvis approach for null checks when it fits better with the surrounding context, such as inside a call chain.

Summary

This chapter covered a lot of ground, including:

- What a [scope](#) is, and how it affects [visibility](#) of things like variables, functions, and classes.
- The difference between [statement scopes](#) and [declaration scopes](#).
- The five scope functions - [with](#), [run](#), [let](#), [also](#), and [apply](#).
- Guidance about how to choose the [most appropriate scope function](#).
- How [shadowing](#) affects names and receivers.
- How to use scope functions for [null checks](#).

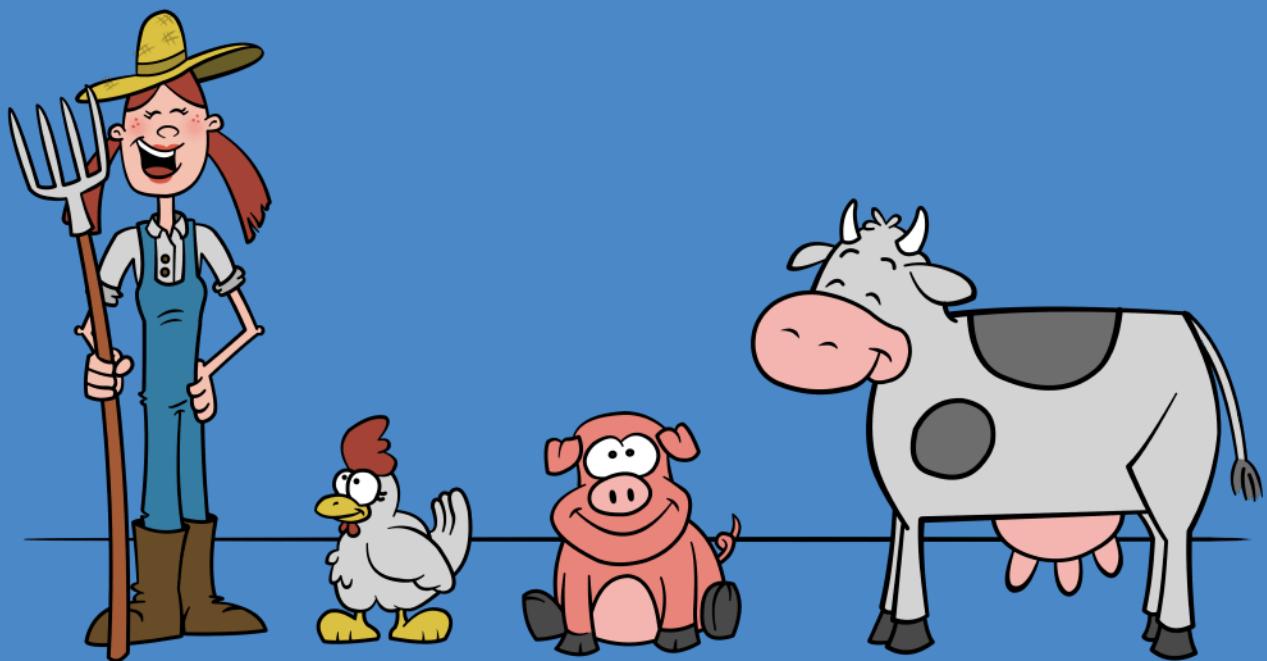
Code written with scope functions can be easier to read, but don't overdo it! If you use scope functions everywhere, or if you start using one scope function inside the lambda of another, it can actually make your code more difficult to understand. Used properly, though, scope functions can be immensely helpful.

In the next chapter, we'll start looking at abstractions, including interfaces, subtypes, and supertypes.

Kotlin: An Illustrated Guide

Chapter 12

Introduction to Interfaces



Types and Supertypes and Subtypes - Oh, my!

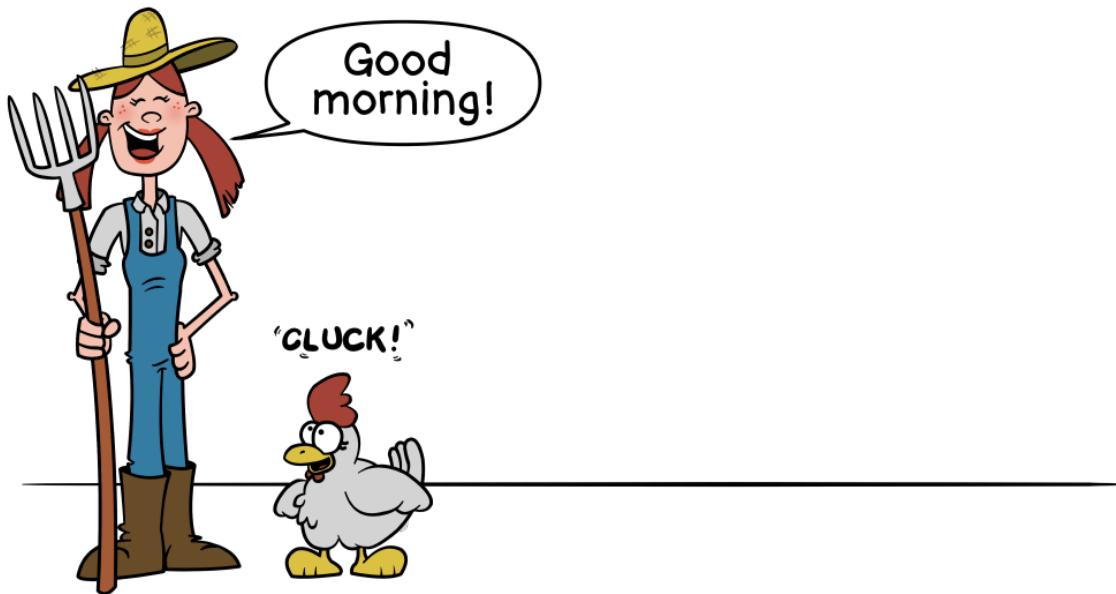
Ever since Chapter 1, we used a variety of built-in Kotlin types, like `Int`, `String`, and `Bool`. Then in Chapter 4, we introduced our own custom types, like `Circle`, by writing classes.

In this chapter, we'll dive into *interfaces*, which will allow you to declare that a single object has more than one type at a time! Interfaces are a powerful language feature, and when used well, they can take your code to the next level.

We've got lots of fun stuff to cover, so let's get started!

Sue Starts a Farm

Sue just bought a lot of land out in the country, and she's ready to start her farm! To kick things off, she got her very first chicken, named Henrietta. Every morning, Sue greets Henrietta, and Henrietta faithfully clucks a greeting back to Sue.



Let's write some Kotlin code to represent Henrietta the chicken and Sue the farmer.

```
class Chicken(val name: String, var number0fEggs: Int = 0) {  
    fun speak() = println("Cluck!")  
}  
  
class Farmer(val name: String) {  
    fun greet(chicken: Chicken) {  
        println("Good morning, ${chicken.name}!")  
        chicken.speak()  
    }  
}
```

Listing 12.1 - Classes to represent a chicken and a farmer.

Now, we can [instantiate](#) the two classes, and Sue can greet Henrietta.

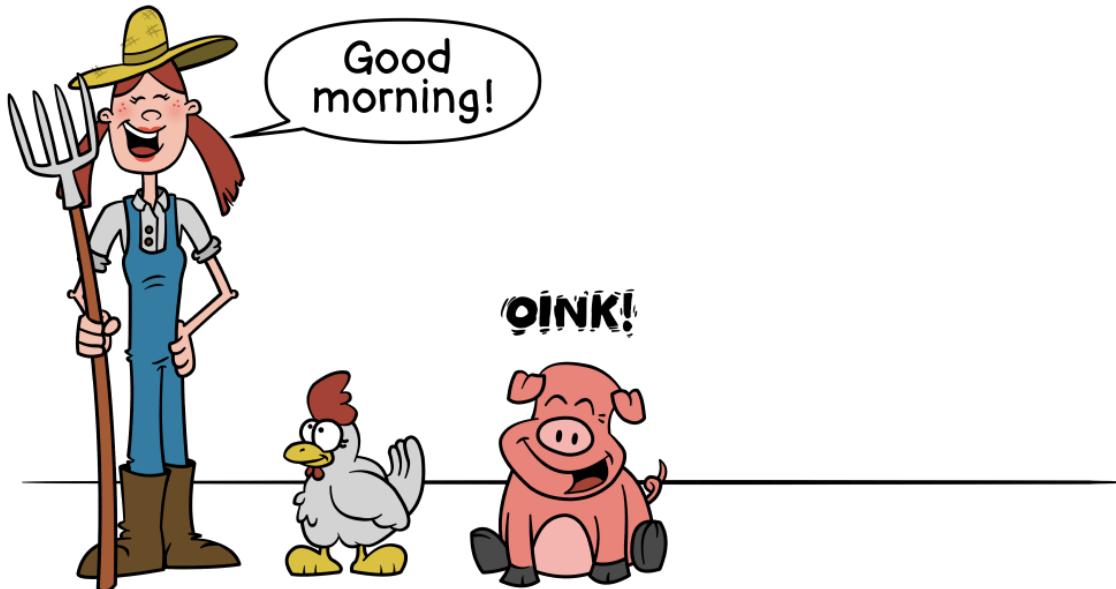
```
val sue = Farmer("Sue")  
val henrietta = Chicken("Henrietta")  
  
sue.greet(henrietta)
```

Listing 12.2 - Instantiating classes, and calling the greet() function.

When you run this, you get the following output:

Good morning, Henrietta!
Cluck!

Now that Sue's farm is off to a good start, she's ready to add another animal resident - this time it's a pig! Farmer Sue will also greet her pig, Hamlet, every morning!



```
class Pig(val name: String, val excitementLevel: Int) {
    fun speak() {
        repeat(excitementLevel) {
            println("Oink!")
        }
    }
}
```

Listing 12.3 - Adding a Pig class.

This looks very similar to the `Chicken` class. It also has a `name` property and a `speak()` function, but it doesn't have a `numberOfEggs` property. Instead, it has an `excitementLevel` property that determines how many "oink" sounds it makes.

We can update the script so that Sue greets both the chicken and the pig, but when we do that, we'll get a compile-time error:

```
val sue = Farmer("Sue")
val henrietta = Chicken("Henrietta")
val hamlet = Pig("Hamlet", 1)

sue.greet(henrietta)
sue.greet(hamlet)
```

Listing 12.4 - Error: Type mismatch: Required Chicken. Found Pig.

Well, that makes sense, of course. The `greet()` function takes a `Chicken`, not a `Pig`.



To remedy this, we can add a new function that takes a `Pig`. We can still name the new function `greet()`, but instead of accepting a `Chicken`, this one will accept a `Pig`.

When we create a function that has the same name as another function but different parameter types, it's called **overloading** the function. Let's create an overload for `greet()` that accepts a `Pig` object.

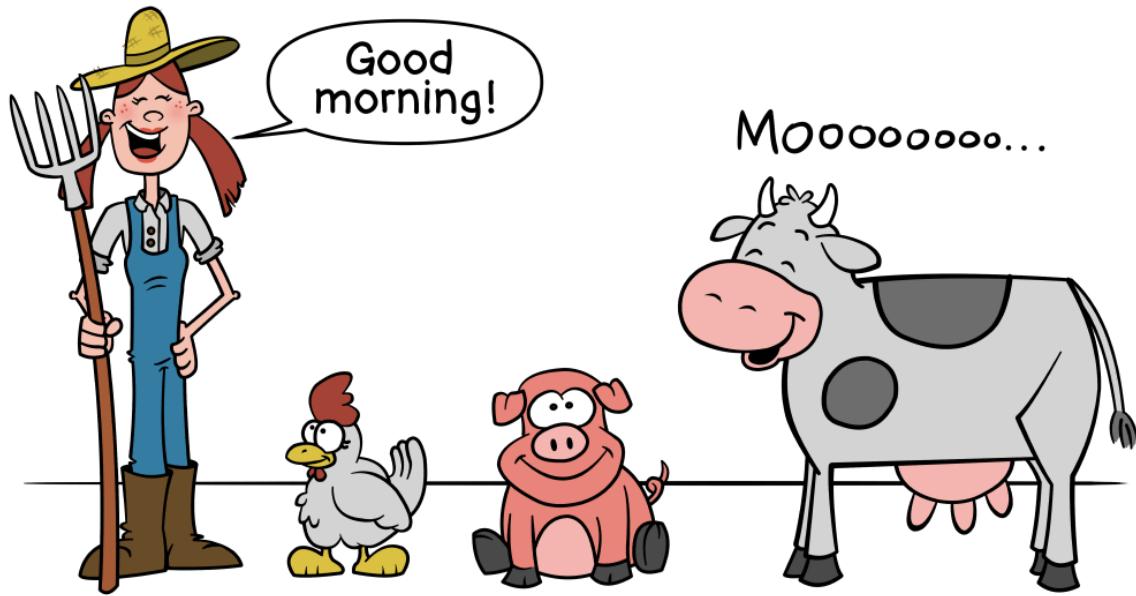
```
class Farmer(val name: String) {
    fun greet(chicken: Chicken) {
        println("Good morning, ${chicken.name}!")
        chicken.speak()
    }

    fun greet(pig: Pig) {
        println("Good morning, ${pig.name}!")
        pig.speak()
    }
}
```

Listing 12.5 - Adding a `greet(pig)` function - which is very similar to `greet(chicken)`.

With this update, Listing 12.4 now compiles and runs as expected.

Now that Sue's farm is doing great with a chicken and pig, she's ready to add a cow! As with the other animals, she greets her cow, Dairy Godmother, every morning, and hears a "Moo" in response! Let's create a class for a Cow.



```
class Cow(val name: String) {  
    fun speak() = println("Moo!")  
}
```

Listing 12.6 - A Cow class, which also has a name property and a speak() function.

As with the pig, when we try to make Sue greet the cow, we get an error.

```
val sue = Farmer("Sue")  
val henrietta = Chicken("Henrietta")  
val hamlet = Pig("Hamlet", 1)  
val dairyGodmother = Cow("Dairy Godmother")  
  
sue.greet(henrietta)  
sue.greet(hamlet)  
sue.greet(dairyGodmother)
```

Listing 12.7 - Error: None of the following functions can be called with the arguments supplied...

Again, to fix this, we can add a new overload to greet the pig.

```
class Farmer(val name: String) {  
    fun greet(chicken: Chicken) {  
        println("Good morning, ${chicken.name}!")  
        chicken.speak()  
    }  
  
    fun greet(pig: Pig) {  
        println("Good morning, ${pig.name}!")  
        pig.speak()  
    }  
  
    fun greet(cow: Cow) {  
        println("Good morning, ${cow.name}!")  
        cow.speak()  
    }  
}
```

Listing 12.8 - Adding a third `greet()` function. This one accepts a `Cow` object.

Yikes! This is becoming unwieldy! Every time Sue adds a new kind of farm animal, we have to create *another* overload of the `greet()` function. That's a shame, because Farmer Sue has some big plans! She wants to add a donkey, a goat, and a llama! That means *even more* overloads...

All of these functions are so similar. In fact, the only difference is the name and type of the parameter.

These are all the same except for the name and type of the parameter:

```
fun greet(chicken: Chicken) {  
    println("Good morning, ${chicken.name}!")  
    chicken.speak()  
}  
  
fun greet(pig: Pig) {  
    println("Good morning, ${pig.name}!")  
    pig.speak()  
}  
  
fun greet(cow: Cow) {  
    println("Good morning, ${cow.name}!")  
    cow.speak()  
}
```

Instead of adding a new function for each new farm animal, it'd be amazing if the `greet()` function could work with *any* kind of farm animal, whether it's a chicken, pig, cow, donkey, goat, llama, or anything else.

In other words, we want something like this:

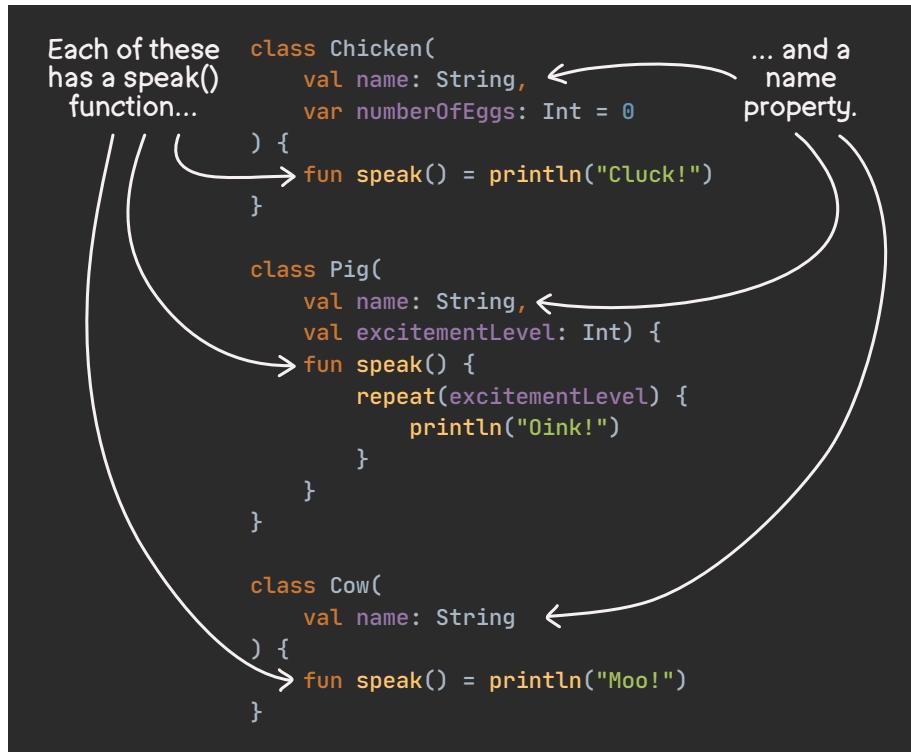
```
class Farmer(val name: String) {
    fun greet(animal: FarmAnimal) {
        println("Good morning, ${animal.name}!")
        animal.speak()
    }
}
```

List 12.9 - What we want: a function that can accept any kind of farm animal.

Thankfully, Kotlin gives us an easy way to do this - with **interfaces**!

Introducing Interfaces

When we look at the animal classes, we can see that they all look very similar - they all have a `name` property and a `speak()` function.



Any new animals that join Susan's farm will also have a `name` and a `sound` that they make when they speak to her. So, we can introduce a new `type` called `FarmAnimal`, and give it a `name` property and a `speak()` function.

As you might recall, so far, we've always created new types by using the `class` keyword. However, instead of introducing the `FarmAnimal` type with a `class`, we'll use an *interface*.

Like a `class`, an interface describes what `properties` and `functions` a type has. However, unlike a class, you don't have to actually *include* any function bodies! For example, we can create the `FarmAnimal` interface like this:

```
interface FarmAnimal {  
    val name: String  
    fun speak()  
}
```

Listing 12.10 - An interface that can represent any kind of farm animal.

Notice that we didn't add any body to the `speak()` function here.

One difference between classes and interfaces, though, is that we can't [instantiate](#) an interface. For example, this won't work:

```
val donkey = FarmAnimal("Phyllis")  
donkey.speak()
```

Listing 12.11 - Error: Interface FarmAnimal does not have constructors.

This makes sense - after all, since the `FarmAnimal` has no function body for `speak()`, what would we expect `donkey.speak()` to actually do?

So then, if interfaces cannot be *instantiated*, how can we use them?

We *update our existing classes*, to tell Kotlin that each of them is a `FarmAnimal` *in addition to* being a `Chicken`, `Pig`, or `Cow`. To start with, let's update the `Cow` class, to mark it as a `FarmAnimal`:

```
class Cow(override val name: String) : FarmAnimal {  
    override fun speak() = println("Moo!")  
}
```

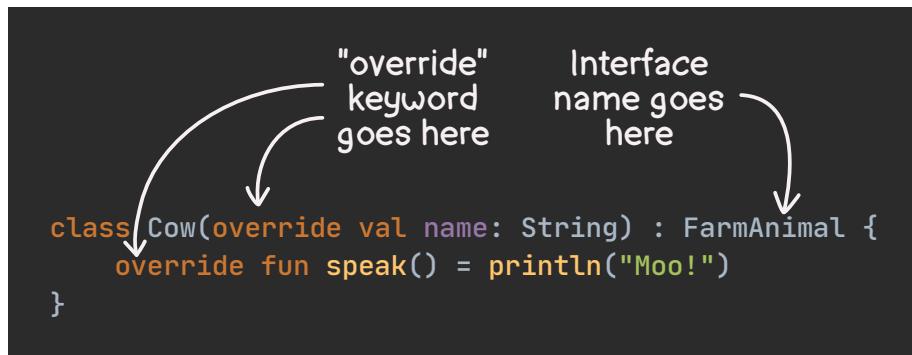
Listing 12.12 - Updating the Cow class so that it implements the FarmAnimal interface.

When a class is marked with an interface like this, we say that the class **implements** the interface. In other words, the `FarmAnimal` interface says that each implementing class *must* include a `name` property and a `speak()` function, but says nothing about the sound the animal should make when `speak()` is called... just that the function must exist. The class, however, provides the implementation - that is, it has a `speak()` function that *does* have a function body.

A class that implements an interface needs to include a few things:

- It must declare that it implements the interface. To do this, add a colon and the name of the interface between the primary constructor and the opening brace of the class body.¹
- It must add the `override` keyword to each property and function that the class implements from the interface. This keyword tells Kotlin that this property or function corresponds to the one from the interface.

¹ In cases where the class has no constructor or body, it'd be as easy as writing `class Chicken : FarmAnimal`.



Once we make these same changes to `Chicken` and `Pig`, we can proceed to remove all those `greet()` functions on the `Farmer` class, and replace them with a single function, as we did in Listing 12.9 (repeated here):

```
class Farmer(val name: String) {
    fun greet(animal: FarmAnimal) {
        println("Good morning, ${animal.name}!")
        animal.speak()
    }
}
```

Listing 12.13 - Updating the `greet()` function to work with any implementations of `FarmAnimal`.

And now, we can call `greet()` with any class that implements the `FarmAnimal` interface. So, as new donkeys, goats, and llamas are added to the farm, we'll just need to declare that they implement the `FarmAnimal` interface, and Farmer Sue can greet them, without any new overloads... in fact, without any changes to the `Farmer` class.

Let's look closer at the relationship between classes and the interfaces that they implement, to understand why this works.

Subtypes and Supertypes

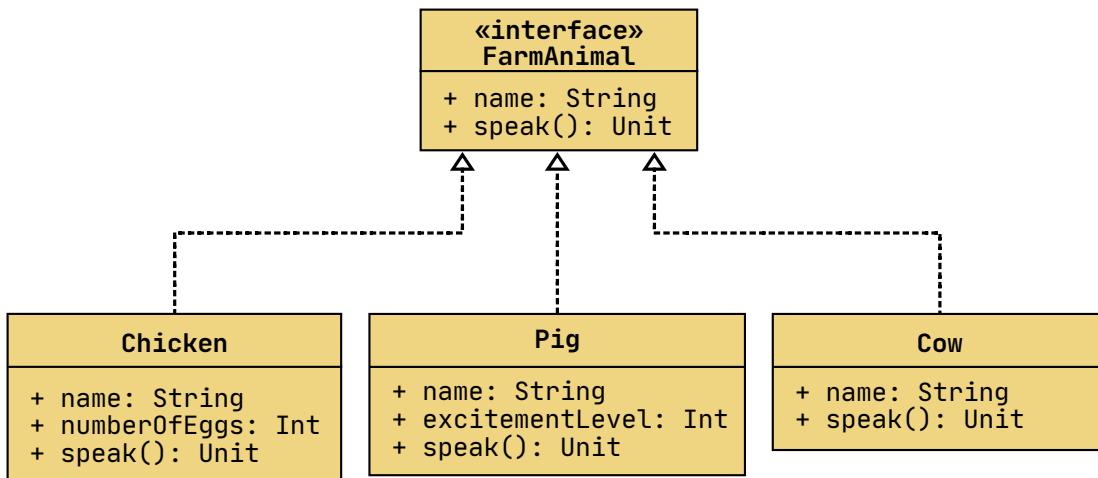
Any tangible, real-life object can usually be categorized in multiple ways. For example, if someone points to a chicken and asks you what it is, you might say "it's a *chicken*", or you might say, "it's a *farm animal*." One is more specific, and one is more general - but both are correct.

This idea of *specific* and *general* types also applies in Kotlin. Now that the `Chicken` class is marked as a `FarmAnimal`, all chicken objects are now *both* a `Chicken` (a more specific type) and `FarmAnimal` (a more general type).

When we're talking about types in Kotlin...

- A class that implements an interface is called a **subtype** of that interface, because the class is a more specific (or, "lower" - and therefore "sub") type.
- Conversely, an interface is called a **supertype** of a class that implements it, because an interface is a more general (or, "higher" - and therefore "super") type.

Back in Chapter 4, we created some UML diagrams to describe our classes. Here's a simple diagram showing that subtype/supertype relationship.



Subtypes and Substitution

A specific type is suitable when someone requests a more general type. For example, if Farmer Sue asks you for a "farm animal" and you give her a chicken, she would be satisfied because a chicken is indeed a kind of farm animal. Alternatively, you could give her a cow or a pig. She would be satisfied with any of those, because each of those is a kind of farm animal.

Similarly, in Kotlin, *you can use a subtype anywhere that the code expects a supertype*. So for instance, if a variable, property, or function expects a **FarmAnimal**, you can give it a **Chicken**, **Cow**, or **Pig**. We already saw this with the `greet(animal)` function in Listing 12.13, but this also applies to variables.

To demonstrate this, we can explicitly specify the type of a variable as **FarmAnimal**, but assign it a **Chicken**.

```
val henrietta: FarmAnimal = Chicken("Henrietta")
```

Listing 12.14 - Explicitly specifying a supertype.

Similarly, we can create a `list` of **FarmAnimal** objects, and give it a **Chicken**, a **Cow**, and a **Pig**. Here's a revised version of Listing 12.7 that uses a `List`.

In both of these cases, we declared a type of **FarmAnimal**, but were able to provide a **Chicken**, **Pig**, or **Cow**, because each of those classes implements **FarmAnimal**.

```
val sue = Farmer("Sue")

val animals: List<FarmAnimal> = listOf(
    Chicken("Henrietta"),
    Pig("Hamlet", 1),
    Cow("Dairy Godmother"),
)

animals.forEach { sue.greet(it) }
```

*Listing 12.15 - Using **FarmAnimal** in a `List` to put multiple implementations into a single collection.*

However, when you assign an object with a more specific type (e.g., a `Chicken` object) to a more general variable or parameter (e.g., one whose type is a `FarmAnimal`), you lose the ability to do specific things with it. For example, the `Chicken` class has a `numberOfEggs` property. You can use this property just fine when the object is assigned to a `Chicken` variable, like this:

```
val henrietta: Chicken = Chicken("Henrietta")
henrietta.numberOfEggs = 1
```

Listing 12.16 - Explicitly specifying the subtype instead of the supertype.

However, after simply changing the type of this variable from a `Chicken` to a `FarmAnimal`, you can't do anything with `numberOfEggs`:

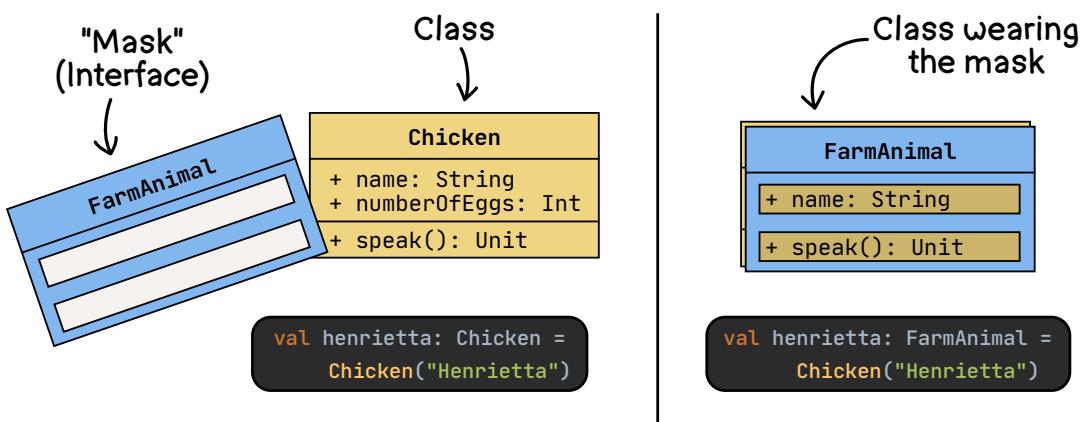
```
val henrietta: FarmAnimal = Chicken("Henrietta")
henrietta.numberOfEggs = 1
```

Listing 12.17 - Error: Unresolved reference: numberOfEggs

Why is that?

When an object of a *specific* type is assigned to a variable of a more *general* type, it's kind of like the object is wearing a mask. That mask hides the things that are declared in the specific type, but lets you see through to the properties and functions that are declared in the more general type.

For example, when assigning a `Chicken` object to a `Chicken` variable (as in Listing 12.16), the `Chicken` class isn't wearing a mask, so you can see all of its properties and functions. However, in Listing 12.17, the `Chicken` object is assigned to a `FarmAnimal` variable, so it's wearing a `FarmAnimal` mask, which only lets Kotlin see the things declared in the `FarmAnimal` interface - `name` and `speak()`.



Because it's wearing a mask, the properties and functions that are declared in `FarmAnimal` are visible, but any properties or functions declared in `Chicken` are hiding behind the mask, so they can't be seen. In some cases, this might prevent you from doing something you want to do. For example, if we update the `greet()` function so that Farmer Sue also says how many eggs she sees, we'll get an error at compile time.

```
val chicken: Chicken = Chicken("Henrietta")
farmer.greet(chicken)

class Farmer(val name: String) {
    fun greet(animal: FarmAnimal) {
        println("Hello, ${animal.name}!")
        println("You have ${animal.numberOfeGgs} eggs today!")
        animal.speak()
    }
}
```

Listing 12.18 - Error: Unresolved reference: numberOfeGgs.

It's good that Kotlin prevents us from doing this. After all, if we were to pass a `Cow` object instead of a `Chicken` object, the `Cow` would have no `numberOfeGgs` property, so it wouldn't make sense to print out a line about eggs at all!

So, how can we get Kotlin to print the line about eggs *only* when the animal actually *is* a Chicken? To do that, we need the `animal` object to cast aside its mask!

Casting

If you want to use a property or function that's declared on a subtype, *you have to tell the object to take off that metaphorical mask first*. Changing the type, such as from `FarmAnimal` to `Chicken`, is called **casting**. There are a few ways to do this.

Smart Casts

One common way to cast the type is to use a [conditional](#) with the `is` keyword, like this:

```
fun greet(animal: FarmAnimal) {
    println("Hello, ${animal.name}!")
    if (animal is Chicken) {
        println("I see you have ${animal.numberOfeGgs} eggs today!")
    }
    animal.speak()
}
```

Listing 12.19 - Using an if conditional with is to perform a smart cast.

Now, *inside* the body of that conditional, the type of `animal` becomes `Chicken`. But outside of that body, it's still a `FarmAnimal`.

```

fun greet(animal: FarmAnimal) {
    println("Hello, ${animal.name}!")
    if (animal is Chicken) {
        println("I see you have ${animal.numberOfEggs} eggs today!")
    }
    animal.speak()
}

```

The code above illustrates smart casts with annotations:

- An annotation points to the first `println` statement: "Type of animal is FarmAnimal here".
- An annotation points to the second `println` statement: "Type of animal is Chicken here".
- An annotation points to the `animal.speak()` call: "Type of animal is FarmAnimal again here".

This is called a **smart cast**. If this looks familiar, it's because we already saw a smart cast in [Chapter 6](#) when we looked at Kotlin's null-safety features. It's the same thing here, but instead of casting from a nullable to a non-null type, we're casting from a `FarmAnimal` to a `Chicken`.

Smart casts can only be used when Kotlin can be *certain* that the value won't change between the conditional and the expression where it's used. For example, in the code above, it's not possible for the value of `animal` to be reassigned, so Kotlin knows it's safe to use a smart cast here.

However, in other situations, it's entirely possible for the value to change after the conditional was evaluated. For example, Kotlin won't smart cast a `var` property of a class, because it's possible that other code might be running at the same time, and that code could reassign a new value to that property.. (So far, we haven't written any code that runs concurrently like that, but we'll see that in a future chapter about coroutines!)

Explicit Casts

Smart casts are an easy way to cast a type, but you can also cast them explicitly yourself. To do this, you can use the `as` keyword. Here's how that looks:

```

fun greet(animal: FarmAnimal) {
    println("Hello, ${animal.name}!")

    val chicken: Chicken = animal as Chicken
    println("I see you have ${chicken.numberOfEggs} eggs today!")

    animal.speak()
}

```

Listing 12.20 - Explicitly casting to a `Chicken`. This is an 'unsafe cast'.

The problem with this is that if `animal` is *not* actually a `Chicken` - for example, if you called `greet()` with a `Cow`, then you'll get an error at runtime. That's why this is sometimes called an **unsafe cast**.

Alternatively, you can use the `as?` keyword (with the question mark), which is called a **safe cast**. Here's how it looks.

```
fun greet(animal: FarmAnimal) {  
    println("Hello, ${animal.name}!")  
  
    val chicken: Chicken? = animal as? Chicken  
    chicken?.let { println("I see you have ${it.numberOfEggs} eggs today!") }  
  
    animal.speak()  
}
```

Listing 12.21 - An explicit safe cast.

The `as?` operator will *try* to cast the object to the specified type. It will evaluate to one of two things:

- If that object actually *is* that specified type, then it evaluates to that object.
- Otherwise, it evaluates to null.

In the code above, if `greet()` is called with a `Chicken` object, then `chicken` would be the same object instance as `animal`, but would have a compile-time type of `Chicken?`. In other words, it took off the mask... but you still have a nullable type to deal with. On the other hand, if `greet()` is called with a `Cow` object, then the `chicken` variable would be null.

Keep in mind that because `as?` evaluates to a [nullable type](#), you've got to use [null-safety tooling](#) in order to deal with the null. For example, in Listing 12.21, we used a [scope function for a null check](#).

A smart cast tends to be the more elegant approach in many cases, so if you find yourself needing to do a cast, that's a great place to start. Consider using `as` or `as?` only if it fits the situation well.

Best Practice: Careful with Casting

Many developers prefer to avoid casting when possible, because it can make things harder to manage later.

For example, in Listing 12.13, the `greet()` function only knew about the `Animal` interface, but didn't need to know anything about its subtypes. This was nice because changes you make to the `Chicken` class wouldn't require you to make changes to the `greet()` function.

In Listing 12.21, though, a change to the `Chicken` class might require a change to the `greet()` function - such as if `numberOfEggs` were to be renamed.

Multiple Interfaces

It's possible for a class to implement more than one interface. To demonstrate this, let's split up the `FarmAnimal` interface into two separate interfaces - one for the `speak()` function and one for the `name` property:

```
interface Speaker {
    fun speak()
}

interface Named {
    val name: String
}
```

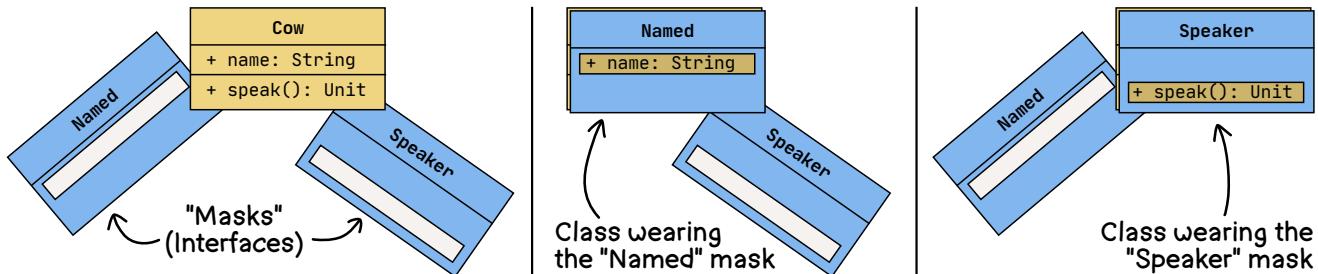
Listing 12.22 - Two interfaces, split out from the FarmAnimal interface.

To update the classes so that they implement both of these interfaces, simply separate the names of the interfaces with a comma, like this:

```
class Cow(override val name: String) : Speaker, Named {
    override fun speak() = println("Moo!")
}
```

Listing 12.23 - A class that implements multiple interfaces.

When you split things up into multiple interfaces like this, it's as if you're creating multiple "masks", each of which exposes only a small part of the class.



This makes it possible to use the type in a broader variety of situations. For example, you could imagine collecting a roster of everyone on the farm, including Farmer Sue. The `Farmer` class already has a `name` property, so we can easily update it to implement the `Named` interface.

```
class Farmer(override val name: String) : Named {
    // (eliding the class body for now)
}
```

Listing 12.24 - Updating the `Farmer` class so that it implements the `Named` interface.

With that change, now we can collect a list of everyone on the farm!

```
val roster: List<Named> = listOf(
    Farmer("Sue"),
    Chicken("Henrietta"),
    Pig("Hamlet", 1),
    Cow("Dairy Godmother")
)
```

Listing 12.25 - Creating a list of `Named` objects, including both the farmer and the animals.

By splitting the `FarmAnimal` interface into `Speaker` and `Named` interfaces, though, we've done away with the `FarmAnimal` interface. That means the `greet()` function doesn't work any more.

```
class Farmer(override val name: String) : Named {
    fun greet(animal: FarmAnimal) {
        println("Good morning, ${animal.name}!")
        animal.speak()
    }
}
```

Listing 12.26 - Error: Unresolved reference: FarmAnimal

Let's fix that next!

Interface Inheritance

In order to get the `greet()` function to work again, we could simply reintroduce the `FarmAnimal` interface, like this...

```
interface Speaker {
    fun speak()
}

interface Named {
    val name: String
}

interface FarmAnimal {
    val name: String
    fun speak()
}
```

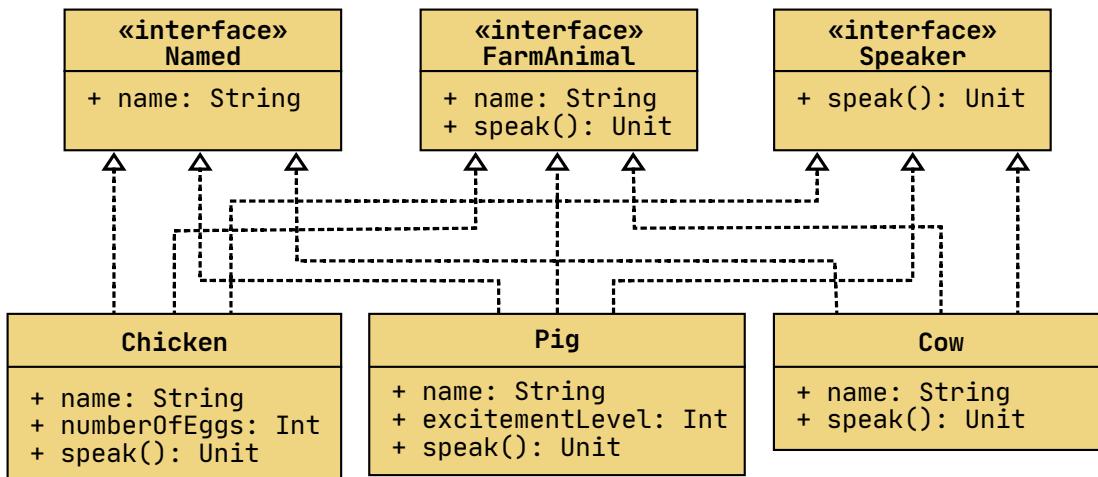
Listing 12.27 - Duplicating the Speaker and Named interfaces in FarmAnimal.

... and then update the classes so that they implement all three of these interfaces, like this:

```
class Cow(override val name: String) : Speaker, Named, FarmAnimal {
    override fun speak() = println("Moo!")
}
```

Listing 12.28 - Implementing three interfaces in one class.

This results in a diagram that looks like this.



Wow - there are lots of lines going everywhere! When a diagram is this confusing to look at, it usually means there's room for improvement in our code. Although this three-interface approach works, Kotlin gives us a more concise way to do this: an interface can **inherit** from other interfaces.¹

When this happens, it *automatically includes* all of the properties and functions from the interfaces that it inherits from. For example, we can update the code from Listing 12.27 so that **FarmAnimal** inherits from both the **Speaker** and **Named** interfaces, like this.²

As in Listing 12.27, a class that implements this **FarmAnimal** interface will still need to have a **name** property and a **speak()** function on it.

```

interface Speaker {
    fun speak()
}

interface Named {
    val name: String
}

interface FarmAnimal : Speaker, Named
  
```

Listing 12.29 - Interface extension.

However, by inheriting from the **Speaker** and **Named** interfaces, **FarmAnimal** is now a subtype of them! This means that every **FarmAnimal** is also a **Speaker** and a **Named**. Now we can remove the **Speaker** and **Named** declarations from Listing 12.28 above, because they come along automatically as a part of **FarmAnimal**:

```

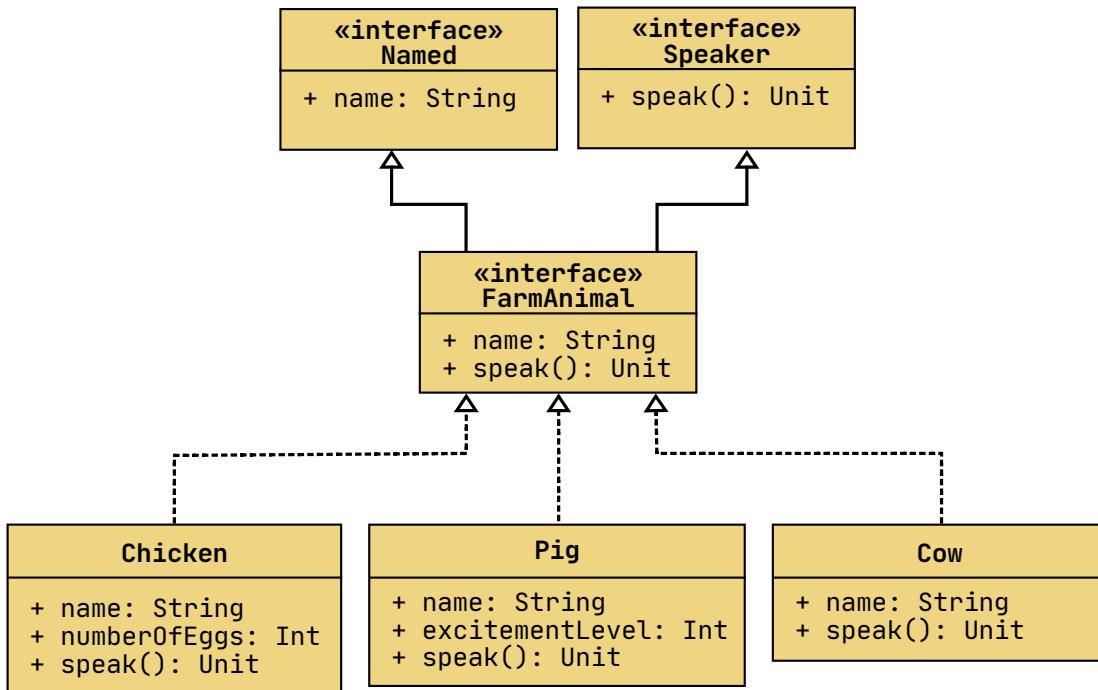
class Cow(override val name: String) : FarmAnimal {
    override fun speak() = println("Moo!")
}
  
```

*Listing 12.30 - Declaring that **Cow** implements **FarmAnimal**. It's implied that it also implements **Named** and **Speaker**, because **FarmAnimal** extends those two interfaces.*

Even though this class only *declares* that it's a **FarmAnimal**, it's also still of type **Named** and **Speaker** as well. The result is a UML diagram that looks like this:

¹Some languages call this *extending* another interface, but since this term could easily be confused with *extensions*, Kotlin developers prefer to call this *inheriting*. Inheritance applies to more than just interfaces, as we'll see in Chapter 14.

²The **FarmAnimal** interface here does not include a body, but often when you extend an interface, you would give it one. That way, **FarmAnimal** would inherit **name** and **speak()**, and then add some more properties or functions of its own.



This diagram looks much better!

Now these classes can be used in a lot of situations! For example, because `Cow` is a subtype of `FarmAnimal`, `Named`, and `Speaker`, a `Cow` object can be sent as an argument to any of the following functions:

```
fun milk(cow: Cow) = // ...
fun feed(animal: FarmAnimal) = // ...
fun introduce(name: Named) = // ...
fun listenTo(speaker: Speaker) = // ...
```

Listing 12.31 - A Cow object can be used wherever a Cow, FarmAnimal, Named, or Speaker type is accepted.

Default Implementations

Default Functions in Interfaces

Most of the time, interfaces themselves do not contain any code – they don't usually include function bodies. However, you *can* actually include a function body, in order to provide a **default implementation**. This default implementation will be used if the class doesn't provide its own implementation of that function. For example, let's update the `Speaker` interface so that it has a default implementation of the `speak()` function.

```
interface Speaker {
    fun speak() {
        println("...")
    }
}
```

Listing 12.32 - A default implementation in an interface.

Now, we can create a new class that implements the `FarmAnimal` interface, but omits the `speak()` function!

```
class Snail(override val name: String) : FarmAnimal
```

Listing 12.33 - A class that uses the default implementation for the speak() function.

This `Snail` class has no class body, let alone a `speak()` function. However, we can still call the `speak()` function on a `Snail`, and when we do that, it'll print ..., suggesting that the snail doesn't say much!



```
val snail = Snail("Slick")
snail.speak() // prints "..."
```

Listing 12.34 - Calling the default implementation of speak().

Default Properties in Interfaces

You can also provide a default implementation for properties, but you can't just directly assign a value. For example, this won't work:

```
interface Named {
    val name: String = "No name"
}
```

Listing 12.35 - Error: Property initializers are not allowed in interfaces

Instead, you can create a **getter** for the property. Some programming languages call this a *computed property*.

```
interface Named {
    val name: String get() = "No name"
}
```

Listing 12.36 - A property getter.

A getter is essentially an underlying function that is called whenever you get the property's value. So when you do `println(something.name)`, it calls that `get()` function and evaluates to the result of that function. Here, we're simply returning the value, "No name".

Now that `FarmAnimal` has a default implementation for *both* `name` and `speak()`, we can create a class that implements the interface, but has no properties or functions at all.

```
class UnknownAnimal : FarmAnimal

val unknown = UnknownAnimal()
```

Listing 12.37 - A class that implements FarmAnimal, using the default implementation for both name and speak().

Default implementations can be especially helpful when adding a new property or function to an existing interface. For example, if we were to add a new `nickname` property to the `FarmAnimal` interface, then the `Chicken`, `Pig`, and `Cow` classes would *all* need to be updated to have that new property, or else you'd get a compile-time error. However, if the `FarmAnimal` interface had a default implementation for that property (for example, it could be set to `null` by default), then the code would compile successfully, even with no other changes to the classes. Then, if cows were the *only* ones who ever had nicknames, you could implement that property *only* in the `Cow` class.

Summary

Well, Sue's farm is now in great shape! As she adds more and more animals, she'll be able to greet them all with ease. This chapter introduced the concept of an interface, and covered these topics:

- [Subtypes and supertypes](#).
- [Casting types](#) with [smart casts](#), [unsafe casts](#), and [safe casts](#).
- Implementing [multiple interfaces](#).
- [Inheriting from an interface](#).
- [Default implementations](#).

In the next chapter, we'll see how we can use interfaces to easily delegate function and property calls to other objects.

Kotlin: An Illustrated Guide

Chapter 13

Introduction to Class Delegation



Don't Duplicate Those Functions!

Back in Chapter 2, we introduced functions as a way to prevent us from duplicating expressions. Once we start working with classes, we might notice that it's also possible to end up duplicating our *functions*.

There are lots of ways to reuse our functions instead of duplicating them. In this chapter, we'll see how we can use interfaces with Kotlin's class delegation feature to reuse the code from our functions and properties.

Roger is out for dinner at a restaurant, when the waiter walks up and asks, "Can I start you off with something to drink, sir?"

"I'd like a soda, please," says Roger.

"One moment, sir." The waiter walks back to a counter in the kitchen, fills a glass of soda, and sets it down on the table. "Are you ready to order your meal, sir?"

Roger replies, "Yes, I'd like the salmon on rice, please."

"I'll get that in for you," says the waiter. He walks back to the kitchen again. This time, though, he doesn't prepare the order himself. Instead, he hands it off to the chef, who skillfully prepares the delicious dinner. Once it's ready, the chef hands the meal to the waiter, who returns and places it on the table for Roger.

Delegation in Restaurants and Code

As this story shows, sometimes a waiter can fulfill an order without involving the chef - such as when Roger ordered his beverage - but in other cases, such as when Roger ordered an entrée, the waiter has to hand off the order to the *chef* to fulfill.

Similarly, in Kotlin, sometimes an object is fully capable of fulfilling a request (such as a function call) on its own, and in other cases, it might need to hand off the request to *another object*.

Whether this happens in real life or in Kotlin, it's called **delegation**.

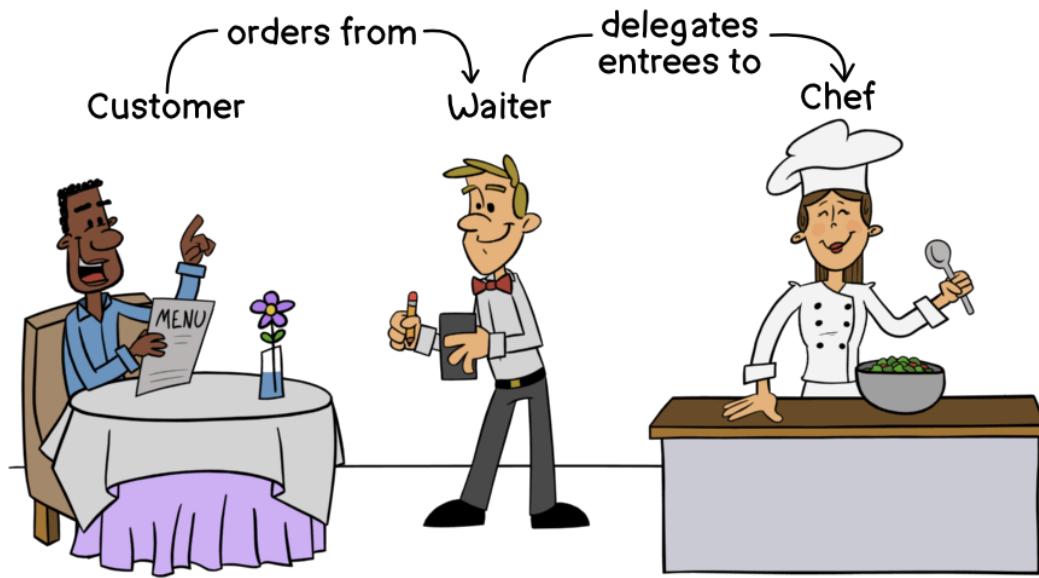
By the way: Delegation vs. Forwarding

Technically, what you're going to see in this chapter is more precisely known in the broader programming community as "forwarding" rather than "delegation". However, in the Kotlin world, it's always referred to as *delegation*, so we'll continue to use that term here.

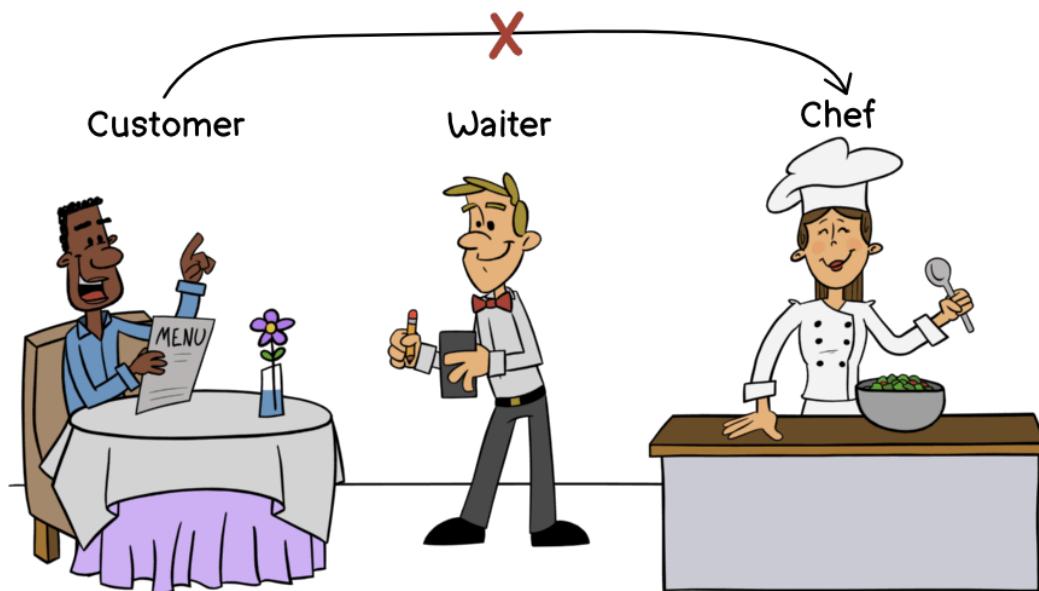
Manual Delegation

Let's review the relationship between the customer, the waiter, and the chef.

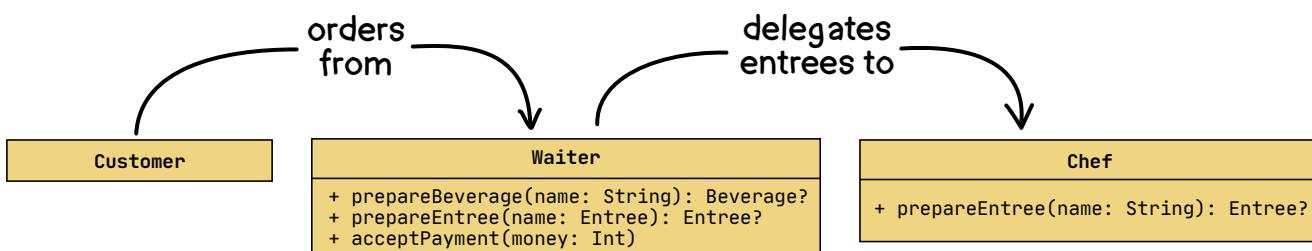
1. The *customer* places an order with the *waiter*.
2. When the order is for an entrée, the *waiter* delegates the entrée preparation to the *chef*.



Notice that customers never interact with the chef *directly* - they only ever interact with the *waiter*, who will interact with the chef on the customers' behalf.



We can model this customer-waiter-chef relationship in Kotlin. To get started, let's replace the *drawings* above with *boxes*, which will roughly convert our illustration into a simple UML [class diagram](#).



Now we're ready to create classes to represent the waiter and the chef, along with some enum classes for the beverages and entrées.

```
class Chef {  
    fun prepareEntree(name: String): Entree? = when (name) {  
        "Tossed Salad"    -> Entree.TOSSED_SALAD  
        "Salmon on Rice" -> Entree.SALMON_ON_RICE  
        else              -> null  
    }  
}  
  
class Waiter(private val chef: Chef) {  
    // The waiter can prepare a beverage by himself...  
    fun prepareBeverage(name: String): Beverage? = when (name) {  
        "Water"   -> Beverage.WATER  
        "Soda"    -> Beverage.SODA  
        else      -> null  
    }  
  
    // ... but needs the chef to prepare an entree  
    fun prepareEntree(name: String): Entree? = chef.prepareEntree(name)  
  
    fun acceptPayment(money: Int) = println("Thank you for paying for your meal")  
}  
  
enum class Entree { TOSSED_SALAD, SALMON_ON_RICE }  
enum class Beverage { WATER, SODA }
```

Listing 13.1 - Modeling out the `Chef` and `Waiter` classes.

In the code above, the `prepareEntree()` function in `Waiter` simply calls the same function on the `chef` object, sending along the same `name` argument that it received. This is manual delegation.

The word **delegate** can be either a noun or a verb, with a slight difference in pronunciation. In the listing above, the `chef` object here is called a **delegate** (noun), because the `Waiter` **delegates** (verb) the `prepareEntree()` call to it.

Now we've got a class for the waiter and the chef. However, we won't bother creating a class for the customer. Instead, the code in the next listing will act like a customer, calling functions on a `Waiter` object.

```
val waiter = Waiter(Chef())  
  
val beverage = waiter.prepareBeverage("Soda")  
val entree = waiter.prepareEntree("Salmon on Rice")
```

Listing 13.2 - Simulating the behavior of a customer.

Congratulations! You've already created a simple, manual delegation relationship between the `Waiter` and the `Chef`. As we'll see in a moment, delegation can be even easier than this!

Before we move on, though, you might have noticed that both `Waiter` and `Chef` have a function called `prepareEntree()`, and those two functions have the same parameter types and return type.

```

class Chef {
    fun prepareEntree(name: String): Entree? = when (name) {
        "Caesar Salad" -> Entree.CAESAR_SALAD
        "Salmon on Rice" -> Entree.SALMON_ON_RICE
        else -> null
    }
}

class Waiter(private val chef: Chef) {
    // The waiter can prepare a beverage by himself...
    fun prepareBeverage(name: String): Beverage? = when (name) {
        "Water" -> Beverage.WATER
        "Soda" -> Beverage.SODA
        else -> null
    }

    // ...but needs the chef to prepare an entree
    fun prepareEntree(name: String): Entree? = chef.prepareEntree(name)

    fun acceptPayment(money: Int) = println("Thank you for paying for your meal")
}

```

As we saw in the last chapter, when this happens, we can create an [interface](#). Let's update our code from Listing 13.1 so that `Waiter` and `Chef` both implement the same interface. Remember, when we do this, we also need to mark the `prepareEntree()` function as `override` in those two classes.

```

interface KitchenStaff {
    fun prepareEntree(name: String): Entree?
}

class Chef : KitchenStaff {
    override fun prepareEntree(name: String): Entree? = when (name) {
        "Tossed Salad" -> Entree.TOSSED_SALAD
        "Salmon on Rice" -> Entree.SALMON_ON_RICE
        else -> null
    }
}

class Waiter(private val chef: Chef) : KitchenStaff {
    fun prepareBeverage(name: String): Beverage? = when (name) {
        "Water" -> Beverage.WATER
        "Soda" -> Beverage.SODA
        else -> null
    }

    override fun prepareEntree(name: String): Entree? = chef.prepareEntree(name)

    fun acceptPayment(money: Int) = println("Thank you for paying for your meal")
}

```

Listing 13.3 - Updating `Chef` and `Waiter` to implement a new `KitchenStaff` interface.

Great! We've now got a delegation relationship between the `Waiter` and the `Chef`, plus we've created an interface for `prepareEntree()` function that they both share.

Delegating More Function Calls

Now, writing one function to manually call `chef.prepareEntree()` isn't too bad, but there are plenty of other cases when the waiter might delegate to the chef. For example:

- Getting a list of the chef's specials for the day
- Preparing an appetizer
- Preparing a dessert
- Sending the customer's compliments along to the chef

It's easy to update the interface to include these things:

```
interface KitchenStaff {  
    val specials: List<String>  
    fun prepareEntree(name: String): Entree?  
    fun prepareAppetizer(name: String): Appetizer?  
    fun prepareDessert(name: String): Dessert?  
    fun receiveCompliment(message: String)  
}
```

Listing 13.4 - Adding more members to the `KitchenStaff` interface.

However, we also have to update the `Chef` and `Waiter` classes to implement the new property and functions. As for how the `Chef` class might implement those functions... we can leave that to our imaginations. We can easily see how this affects the `Waiter` class, though:

```
class Waiter(private val chef: Chef) : KitchenStaff {  
    // These first two functions are the same as before  
    fun prepareBeverage(name: String): Beverage? = when (name) { /* ... */ }  
    fun acceptPayment(money: Int) = println("Thank you for paying for your meal")  
  
    // Manually delegating to the chef for all of these things:  
    override val specials: List<String> get() = chef.specials  
    override fun prepareEntree(name: String) = chef.prepareEntree(name)  
    override fun prepareAppetizer(name: String) = chef.prepareAppetizer(name)  
    override fun prepareDessert(name: String) = chef.prepareDessert(name)  
    override fun receiveCompliment(message: String) = chef.receiveCompliment(message)  
}
```

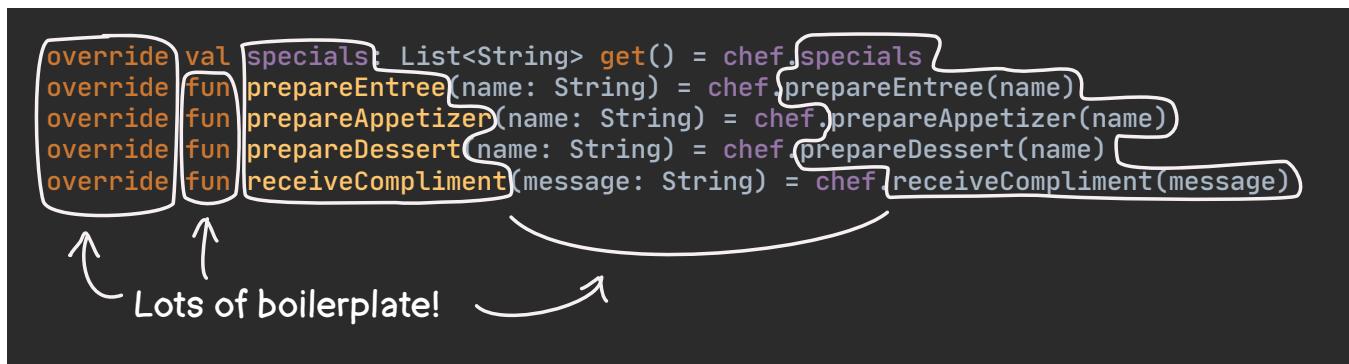
Listing 13.5 - Overriding the new properties and functions in the `Waiter` class.

As more and more properties and functions are added to the `KitchenStaff` interface, manually delegating from the `Waiter` to the `Chef` becomes tedious to write, tedious to read, and makes it more likely that we could accidentally get something wrong.

When you look at `override` functions above, you'll notice a lot of repeated text in the code:

- The `override fun`
- The name of the function on the left and the right side of the line
- The parameter names on the left and also on the right

The pattern is the same on each line. Code that has the same repeated pattern like this is called **boilerplate**.¹ Even using Kotlin's [expression body](#) functions, as we're doing here, the boilerplate is really starting to pile up.



Thankfully, Kotlin makes it easy to do this kind of delegation, without having to write it all by hand!

Easy Delegation, the Kotlin Way

Instead of writing all of the boilerplate for delegation manually, we can just use Kotlin's class delegation feature. To do this, we just need to do two things:

1. Indicate which functions and properties to delegate
2. Indicate which object they should be delegated to

The functions and properties are specified by the name of the *interface* that contains them, and the delegate is specified using the `by` keyword. Once you do this, you can remove the manual delegation.

For example, to delegate all of the properties and functions in the `KitchenStaff` to the `chef` object, we can simply write this:

```
class Waiter(private val chef: Chef) : KitchenStaff by chef {
    fun prepareBeverage(name: String): Beverage? = when (name) { /* ... */ }
    fun acceptPayment(money: Int) = println("Thank you for paying for your meal")
}
```

Listing 13.6 - Automatically delegating to the `chef` object.

¹The term **boilerplate** comes from the old days of the printed newspaper industry, when publishing syndicates would send local newspapers stories on metal plates that were ready for them to print. Those metal plates looked like the plating used when producing steam boilers. Those stories were also often fluff articles that lacked original content, which is how the term got its meaning. ("boilerplate," Merriam-Webster.com Dictionary, <https://www.merriam-webster.com/dictionary/boilerplate>).

Easy, right? This code works the same as Listing 13.5; it's just written differently. With this change, all of the properties and functions that were manually delegated to `chef` are *completely omitted* from the `Waiter` class body, but it's still delegating them.

The only functions left inside this class are `prepareBeverage()` and `acceptPayment()` - the two functions that `Waiter` handles himself. In other words, when you look at this class, you can really focus on what's *unique* about the `Waiter` class instead of having to see everything that it delegates to `chef`.

By simply including `by chef`, Kotlin does a lot of work for us. When you look at the first line of Listing 13.6, you can read it like:

`Waiter` implements `KitchenStaff` by delegating to `chef`.

Or, illustrated:

```
class Waiter(private val chef: Chef) : KitchenStaff by chef
```

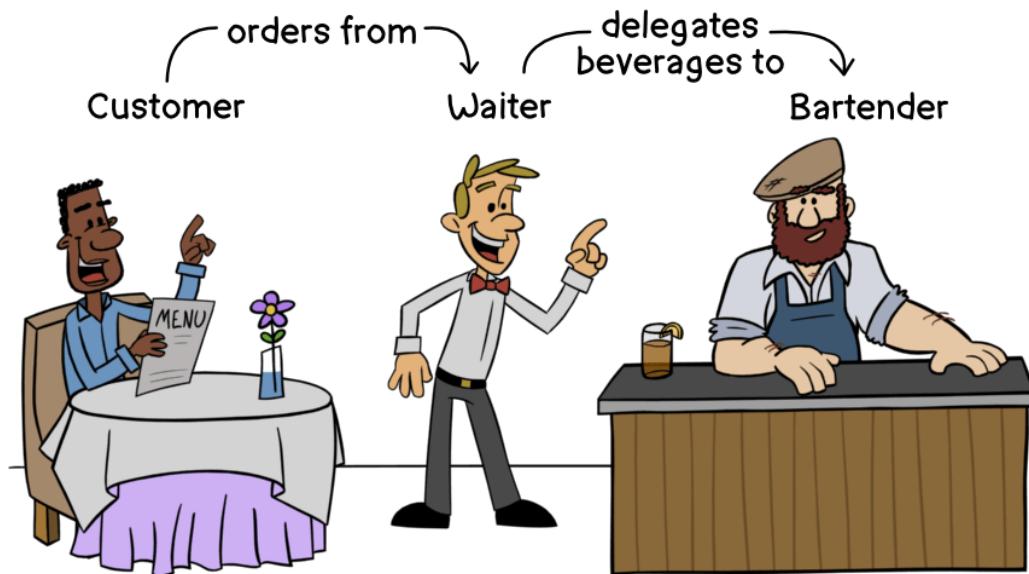
By the way...

Note that since we no longer needed a *member instance* of `chef`, we *could* even remove `private val` from the constructor. However, we're going to need it again later in this chapter, so let's leave it there for now.

Multiple Delegates

Good news - the restaurant just opened up its new beverage bar, offering fancy drinks like peach iced tea and tea-lemonade. Instead of preparing the beverages himself, the waiter will now delegate beverage preparation to the bartender.

To start with, let's add a new interface and class for the bartender. We'll also update the `Beverage` enum class with the new beverage options.



```

interface BarStaff {
    fun prepareBeverage(name: String): Beverage?
}

class Bartender: BarStaff {
    override fun prepareBeverage(name: String): Beverage? = when (name) {
        "Water"           -> Beverage.WATER
        "Soda"            -> Beverage.SODA
        "Peach Tea"       -> Beverage.PEACH_ICED_TEА
        "Tea-Lemonade"   -> Beverage.TEA_LEMONADE
        else              -> null
    }
}

enum class Beverage { WATER, SODA, PEACH_ICED_TEА, TEA_LEMONADE }

```

Listing 13.7 - Adding an interface and class for the bartender.

Now, we want the Waiter to delegate the beverage preparation to the bartender. To start, let's add a bartender property to the Waiter, and manually delegate to it, just like we did previously with the chef.

```

class Waiter(
    private val chef: Chef,
    private val bartender: Bartender
) : KitchenStaff by chef, BarStaff {
    override fun prepareBeverage(name: String) = bartender.prepareBeverage(name)
    fun acceptPayment(money: Int) = println("Thank you for paying for your meal")
}

```

Listing 13.8 - Manually delegating to the bartender object.

Here, we've got Kotlin *automatically* delegating everything in `KitchenStaff` to the `chef`, and we're *manually* delegating `prepareBeverage()` to the `Bartender`. This is fine, but Kotlin also allows us to use `by` to delegate to *more than one object at a time*, and it works just like you'd expect. Simply add `by bartender` after `BarStaff`, like this:

```
class Waiter(  
    private val chef: Chef,  
    private val bartender: Bartender  
) : KitchenStaff by chef, BarStaff by bartender {  
    fun acceptPayment(money: Int) = println("Thank you for paying for your meal")  
}
```

Listing 13.9 - Automatically delegating to the `bartender` object.

With this, we now have a class that delegates to two different classes. As in Listing 13.8, the body of the `Waiter` class shows only what's unique to the `Waiter` class. All of the delegation boilerplate is tucked away behind those simple `by` declarations!

Overriding a Delegated Call

As it's getting later, the restaurant is getting crowded, and the chef is busy trying to prepare food for all of the customers. So, the chef says to the waiter, "The salad is easy to prepare. From now on, if you get an order for a tossed salad, just take care of it yourself. I'll still handle the fancier meals."

As we've seen, when we use Kotlin's class delegation, all of the properties and functions from the interface are automatically sent to the designated object. However, you can also choose *not* to automatically delegate one or more *particular* properties or functions from the interface.

Let's update our Kotlin code so that the `Waiter` can prepare the salad by himself. To do this, we can include the `prepareEntree()` function in `Waiter` again, and manually delegate it to the `chef` *only when needed*. Here's how that looks:



```
class Waiter(  
    private val chef: Chef,  
    private val bartender: Bartender  
) : KitchenStaff by chef, BarStaff by bartender {  
    override fun prepareEntree(name: String): Entree? =  
        if (name == "Tossed Salad") Entree.TOSSED_SALAD else chef.prepareEntree(name)  
  
    fun acceptPayment(money: Int) = println("Thank you for paying for your meal")  
}
```

Listing 13.10 - Using both automatic and manual delegation with the `chef` object.

When a function from `KitchenStaff` is included in the `Waiter` class, Kotlin will use that function instead of the sending it to the delegate. However, we can still choose to manually delegate to it, as we're doing here in Listing 13.10. When the customer orders anything other than a tossed salad, we manually delegate to the `chef`.

So, we can combine Kotlin's class delegation *with* manual delegation.

Managing Conflicts

When the meal is fantastic, sometimes the customer tells the waiter to send along compliments to the chef. The `KitchenStaff` has a function called `receiveCompliment()`. So, when this function is called on a `Waiter` object, it's sent along to its `chef` object. Inside the `Chef` class, we can simply print out that the compliment was received, like this:

```
interface KitchenStaff {  
    // ... disregarding other properties and functions ...  
    fun receiveCompliment(message: String)  
}  
  
class Chef : KitchenStaff {  
    // ... disregarding other properties and functions ...  
    override fun receiveCompliment(message: String) =  
        println("Chef received a compliment: $message")  
}
```

Listing 13.11 - Implementing `receiveCompliment()` inside the `Chef` class.

And of course, calling this function on the `waiter` will send it along to the `chef`.

```
val waiter = Waiter(Chef(), Bartender())  
  
val beverage = waiter.prepareBeverage("Water")  
val entree = waiter.prepareEntree("Salmon on Rice")  
  
waiter.receiveCompliment("The salmon entree was fantastic!")
```

Listing 13.12 - Calling `receiveCompliment()` on the `waiter` object, which is forwarded to the `chef`.

The chef isn't the only one who might receive a compliment, though. The bartender makes an excellent peach iced tea, and sometimes the customers want to send along their compliments to the bartender, too!

Let's update the `BarStaff` interface and the `Bartender` class so that he can also receive a compliment.

```
interface BarStaff {  
    fun prepareBeverage(name: String): Beverage  
    fun receiveCompliment(message: String)  
}  
  
class Bartender: BarStaff {  
    // ... disregarding other properties and functions ...  
    override fun receiveCompliment(message: String) =  
        println("Bartender received a compliment: $message")  
}
```

Listing 13.13 - Updating the `BarStaff` interface and `Bartender` class to receive a compliment.

Now, *both* `Chef` and `Bartender` include a function called `receiveCompliment()` - and they both have the same parameter types and return type.

```
interface KitchenStaff {  
    val specials: List<String>  
    fun prepareEntree(name: String): Entree?  
    fun prepareAppetizer(name: String): Appetizer?  
    fun prepareDessert(name: String): Dessert?  
    fun receiveCompliment(message: String)  
}  
  
interface BarStaff {  
    fun prepareBeverage(name: String): Beverage  
    fun receiveCompliment(message: String)  
}
```

same function name, parameter, and return type

So, when our code calls `waiter.receiveCompliment()`, what should Kotlin do with it? Should it send the compliment to the `chef` object, or to the `bartender` object? Or maybe both?

It's up to you as the programmer to decide. In fact, in a situation like this, where you try to use class delegation for two interfaces that have the same property or function, you'll get a compiler error like this:

```
Class 'Waiter' must override public open fun receiveCompliment(message: String): Unit  
defined in Waiter because it inherits many implementations of it.
```

To resolve this, we'll need to include this function in the `Waiter` class. For example, we could check to see if the `message` includes the word "entree" or "beverage", and manually delegate to the `chef` or `bartender` accordingly. Then, as a last resort, the waiter can receive the compliment himself.

```

class Waiter(
    private val chef: Chef,
    private val bartender: Bartender
) : KitchenStaff by chef, BarStaff by bartender {
    override fun receiveCompliment(message: String) = when {
        message.contains("entree") -> chef.receiveCompliment(message)
        message.contains("beverage") -> bartender.receiveCompliment(message)
        else -> println("Waiter received compliment: $message")
    }

    override fun prepareEntree(name: String): Entree? =
        if (name == "Tossed Salad") Entree.TOSSED_SALAD else chef.prepareEntree(name)

    fun acceptPayment(money: Int) = println("Thank you for paying for your meal")
}

```

Listing 13.14 - Manually delegating `receiveCompliment()` to either the `chef` or the `bartender`; or handling it directly.

And now, the customer can send compliments along to the chef, the bartender, or the waiter!

```

val waiter = Waiter(Chef(), Bartender())

waiter.receiveCompliment("The salmon entree was fantastic!")
waiter.receiveCompliment("The peach tea beverage was fantastic!")
waiter.receiveCompliment("The service was fantastic!")

```

Listing 13.15 - Sending compliments to the `chef`, the `bartender`, and the `waiter`.

```

Chef received a compliment: The salmon entree was fantastic!
Bartender received a compliment: The peach tea beverage was fantastic!
Waiter received compliment: The service was fantastic!

```

So far, we've used class delegation to easily forward property calls and function calls from a `waiter` object to the `chef` and `bartender` objects. Although waiters, chefs, and bartenders illustrate the concept of delegation with concrete examples, delegation is also often used for a different purpose - to share *general* code across multiple *specific* types. So before we wrap up this chapter, let's see how that works!

Delegation for General and Specific Types

As we saw in the [last chapter](#), interfaces often represent *general types* like `FarmAnimal`, and classes that implement interfaces often represent more *specific types* like `Chicken`, `Pig`, and `Cow`. In some cases, you might have some general code that you want to *share* among many different specific types.

For example, all farm animals will want to eat, although the particular food that they eat might be a little different depending on what kind of animal they are. In this next code listing, we have three farm animals, each with its own `eat()` function.

```
class Cow {
    fun eat() = println("Eating grass - munch, munch, munch!")
}

class Chicken {
    fun eat() = println("Eating bugs - munch, munch, munch!")
}

class Pig {
    fun eat() = println("Eating corn - munch, munch, munch!")
}

Cow().eat()      // Eating grass - munch, munch, munch!
Chicken().eat() // Eating bugs - munch, munch, munch!
Pig().eat()      // Eating corn - munch, munch, munch!
```

Listing 13.16 - Three farm animals, all eating something different.

You probably noticed some boilerplate code here again. In fact, other than their names, the only difference between each of these classes is the *food* in the string. The `eat()` function is *general*, and we can share it across these *specific* classes with delegation. It's quick and easy, so let's start with a simple interface for someone who eats.

```
interface Eater {
    fun eat()
}
```

Listing 13.17 - An interface for any object that can eat.

Let's implement this interface with a class that makes that munching sound.

```
class Muncher(private val food: String) : Eater {
    override fun eat() = println("Eating $food - munch, munch, munch!")
}
```

Listing 13.18 - A class that implements the `Eater` interface, making a munching sound.

By using class delegation, we can make it so that all of the animals share this implementation of the `eat()` function. Notice that there's a lot less repeated code now:

```
class Cow : Eater by Muncher("grass")
class Chicken : Eater by Muncher("bugs")
class Pig : Eater by Muncher("corn")

Cow().eat()      // Eating grass - munch, munch, munch!
Chicken().eat() // Eating bugs - munch, munch, munch!
Pig().eat()      // Eating corn - munch, munch, munch!
```

Listing 13.19 - Using delegation so that all animals can use the same `eat()` function.

As it turns out, the pigs on this farm like to scarf down their dinner much faster than the cows and chickens, who usually just graze. So instead of sharing the `Muncher` code with them, we can just implement the `eat()` function directly in `Pig`, like this:

```
class Cow : Eater by Muncher("grass")
class Chicken : Eater by Muncher("bugs")
class Pig : Eater {
    override fun eat() = println("Scarfing down corn - NOM NOM NOM!!!")
}

Cow().eat()      // Eating grass - munch, munch, munch!
Chicken().eat() // Eating bugs - munch, munch, munch!
Pig().eat()      // Scarfing down corn - NOM NOM NOM!!!
```

Listing 13.20 - Implementing the `eat()` function directly in the `Pig` class.

Alternatively, we can accomplish the same thing by creating another class, and using it as the delegate of the `Pig`. Here's how that looks:

```
class Scarfer(private val food: String) : Eater {
    override fun eat() = println("Scarfing down $food - NOM NOM NOM!!!")
}

class Cow : Eater by Muncher("grass")
class Chicken : Eater by Muncher("bugs")
class Pig : Eater by Scarfer("corn")
```

Listing 13.21 - Creating a second implementation of `Eater` that is used by the `Pig` class.

So, class delegation can be used to share some *general* code - such as how to `eat()` - across different specific types - like `Cow`, `Chicken`, and `Pig` classes.

This is only one way of sharing code across different classes. In the next chapter, we'll look at *abstract* and *open* classes, which can also be used to share code.

Summary

In this chapter, you learned:

- What [delegation](#) is.
- How to [manually delegate](#) from one object to another.
- How to use Kotlin's [class delegation](#) to automatically delegate.
- How to [override particular functions](#) that would otherwise be delegated.
- How to [resolve conflicts](#) when two delegates provide an implementation for the same function.

- How to [use delegation for general and specific types](#).

As mentioned above, the next chapter will cover *abstract* and *open* classes, which can also be used to share code among multiple classes.

Kotlin: An Illustrated Guide

Chapter 14

Abstract and Open Classes

VR0000M!



There's More Than One Way to Avoid Duplication

In Chapter 12, we saw how we could use interfaces to create subtypes, and in the last chapter, we saw how we could use *delegation* with interfaces in order to share general code among specific classes.

In this chapter, we'll learn how we can extend *open and abstract classes* to accomplish these same things with a different approach. Each approach has its advantages and disadvantages, so we'll also look at how they compare to one another.

Modeling a Car

Let's start by modeling a simple car that can increase its speed with an `accelerate()` function.

```
class Car {  
    private var speed = 0.0  
    private fun makeEngineSound() = println("Vrrrrrrr...")  
  
    fun accelerate() {  
        speed += 1.0  
        makeEngineSound()  
    }  
}
```

Listing 14.1 - A simple class to represent a car.

Each time that we call the `accelerate()` function, the car will increase its speed by `1.0`¹ and make an engine sound.

```
val myCar = Car()  
myCar.accelerate()
```

Listing 14.2 - Instantiating the `Car` class and calling its `accelerate()` function.

Vrrrrrrr...

This works great for many cars because they make a "Vrrrrrrr..." sound.



But wait... here comes Old Man Keaton in his rundown, puttering clunker of a car. Instead of a smooth "Vrrrrrrr..." sound, it sounds like "putt-putt-putt"!

¹To keep things simple, I'm not including a unit of speed. If it helps, feel free to imagine that the speed is in kilometers per hour, miles per hour, meters per second, or any other unit you like!

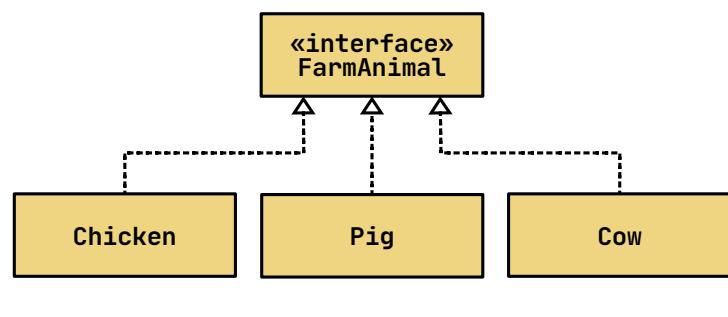


And whizzing by him is Rico in his 1969 muscle car! Go, Rico!

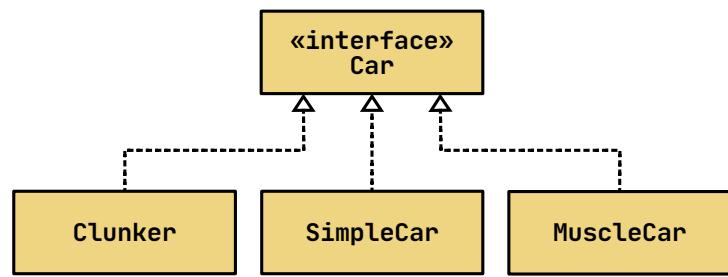


Well, it seems like "Vrrrrrr..." is a fine sound for lots of cars, but we'll need to allow different kinds of cars to have different engine sounds. How can we accomplish this in Kotlin?

This problem sounds familiar. In [Chapter 12](#), in order to create multiple animal classes that could each have its own sound, we created an interface called `FarmAnimal`, and classes for each specific animal, like `Chicken`, `Pig`, and `Cow`.



If that approach worked for animals making sounds, can it also work for cars and their engine sounds? Could we convert our `Car` class into an interface, like `FarmAnimal`, and create subtypes for the different kinds of cars?



Well, it's possible to convert the `Car` class into an interface, but it brings with it some rather significant and undesirable changes. Here's how it looks as an interface, compared to the original class. Can you spot the differences?

New Car Interface

```
interface Car {  
    var speed: Double  
    fun makeEngineSound() =  
        println("Vrrrrrr...")  
  
    fun accelerate() {  
        speed += 1.0  
        makeEngineSound()  
    }  
}
```

Original Car Class

```
class Car {  
    private var speed = 0.0  
    private fun makeEngineSound() =  
        println("Vrrrrrr...")  
  
    fun accelerate() {  
        speed += 1.0  
        makeEngineSound()  
    }  
}
```

Here are the changes we had to make when converting `Car` into an interface:

- Visibility** - An interface cannot have `private` members, so we had to make `speed` and `makeEngineSound()` public. This means that code *outside* of `Car` would be able to set the speed, without going through the `accelerate()` function. Similarly, it'd be possible to call `makeEngineSound()` without accelerating.
- State** - Although an interface can declare a property, it can't contain *state*. In other words, it can't itself contain a *value* for that property. So, we had to remove the `= 0.0` from the `speed` property. The implementing classes will have to initialize it, instead.
- Instantiation** - An interface cannot be instantiated. To work around this, we would have to introduce a class that [implements](#) the `Car` interface, and instantiate *that*, instead.

Those are some concerning changes, so it'd be great if we could create a subtype *without* introducing them. Thankfully, in addition to creating subtypes from an interface, Kotlin also allows us to create subtypes from a *class*. But we can't create a subtype from just *any* old class. By default, a class is **final**, which means that Kotlin will not allow us to create subtypes from it.

Instead, we have to modify the [class](#) declaration so that Kotlin knows we want to create subtypes from it. One way to do this is with an **abstract class**.

Introduction to Abstract Classes

Abstract classes are a lot like interfaces, but they can include private functions, private properties, and state. To create an abstract class, just add the `abstract` modifier when declaring it. As you can see below, the only difference between the new abstract class and the original class is the word `abstract` at the beginning.

New Abstract Car Class

```
abstract class Car {
    private var speed = 0.0
    private fun makeEngineSound() =
        println("Vrrrrrrr...")

    fun accelerate() {
        speed += 1.0
        makeEngineSound()
    }
}
```

Original Car Class

```
class Car {
    private var speed = 0.0
    private fun makeEngineSound() =
        println("Vrrrrrrr...")

    fun accelerate() {
        speed += 1.0
        makeEngineSound()
    }
}
```

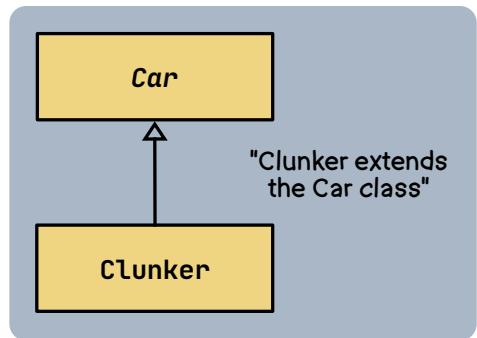
So far, so good! Just like with our original `Car` code, the `speed` property is private and initialized to `0.0`. The `makeEngineSound()` function is also private.

There's one problem, though. As with the interface version of `Car` above, we can't directly instantiate this abstract class - we can only instantiate its subtypes. Later in this chapter, we'll remedy this, but for now, let's see how we can create a subtype from our new abstract class!

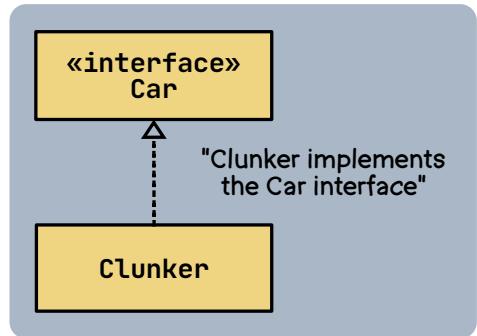
Extending Abstract Classes

Now that we've got this abstract `Car` class, we can create a subtype from it.

When we create a subtype class from an *abstract class*, we usually say that the subtype class **extends** the abstract class.

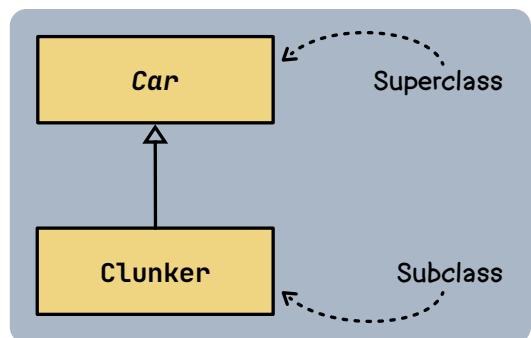


As you might recall from Chapter 12, this differs from when we create a subtype class from an *interface*, in which case we say that the class *implements* the interface.



There are two more terms to keep in mind when creating a subtype from a class:

- The class being extended is called a **superclass**,
- The class that's extending it is called a **subclass**.



So, how can we create a subclass from `Car`? Thankfully, the syntax for subtyping a *class* looks a whole lot like the syntax for subtyping an *interface*!

Subtyping a Car Class

```
class Clunker : Car() {  
    // ...  
}
```

Subtyping a Car Interface

```
class Clunker : Car {  
    // ...  
}
```

The only difference between these two is the *parentheses*. Why is it that we need parentheses when we subtype a class, but not when we subtype an interface? Because classes have constructors, but interfaces don't. The parentheses here are *invoking the constructor* of the `Car` class.

This is much easier to see if we were to add a [constructor parameter](#), so let's add one for the rate of acceleration.

```
abstract class Car(private val acceleration: Double = 1.0) {  
    private var speed = 0.0  
    private fun makeEngineSound() = println("Vrrrrrrr...")  
  
    fun accelerate() {  
        speed += acceleration  
        makeEngineSound()  
    }  
}
```

Listing 14.3 - Adding an `acceleration` property to the `Car` class as a constructor parameter.

Now, when we create our `Clunker` subclass, we can pass it a rate of acceleration that's slower than the default of `1.0`. Now that we're passing a constructor argument, it's easier to see that we're calling the constructor with those parentheses!

```
class Clunker : Car(0.25)
```

Listing 14.4 - Instantiating a `Car` class with a literal for the acceleration.

In this case, we hard-coded the acceleration to `0.25` for all clunkers. If we don't want all clunkers to have the same acceleration, we could also add it as a constructor parameter of the `Clunker` class, and just *relay* the argument to the constructor of the `Car` class.

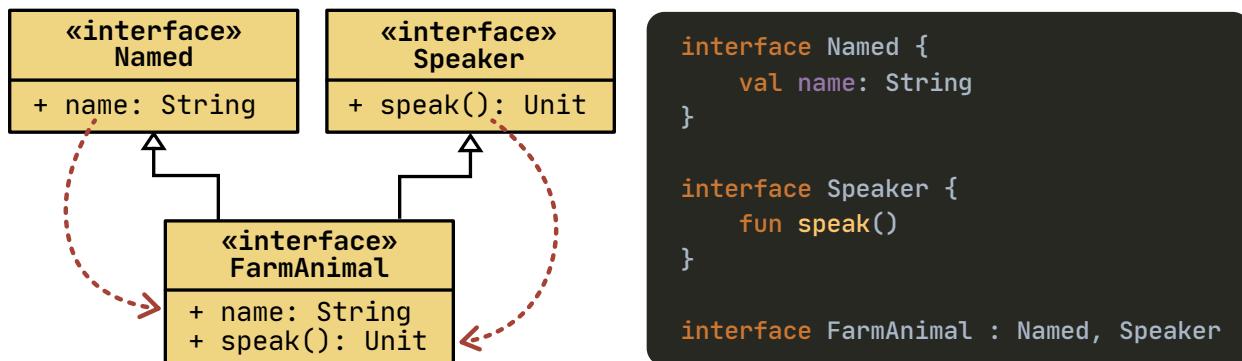
```
class Clunker(acceleration: Double) : Car(acceleration)

val clunker = Clunker(0.25)
```

Relying constructor arguments from a subclass (e.g., `Clunker`) to a superclass (e.g., `Car`) is a very common thing to do. Note that `acceleration` in `Clunker`'s constructor does *not* include the `val` or `var` keyword - we're just passing it along to the `Car` class, which will store it as a property.

Inheritance

Back in [Chapter 12](#), we learned how one interface can *inherit* the functions and properties of another interface. As the example that we used in that chapter, a `FarmAnimal` has a name and it can speak, so we were able to inherit from both a `Named` interface and a `Speaker` interface.

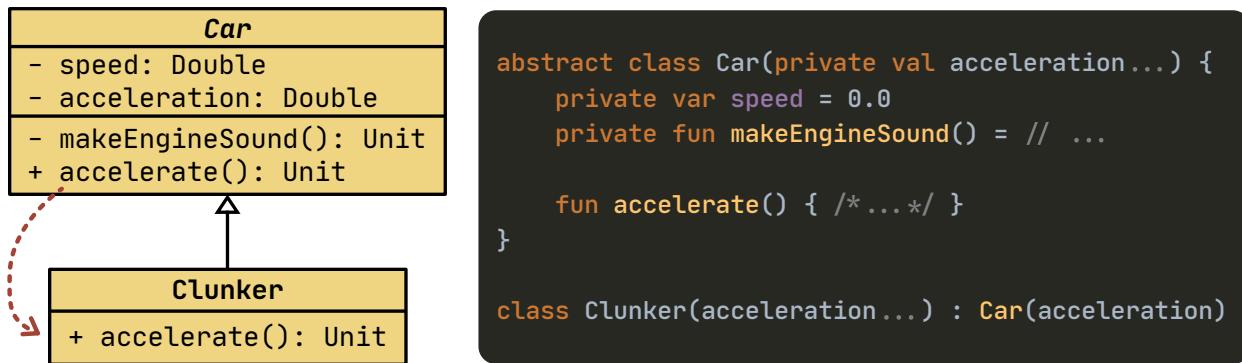


In the code above, the `FarmAnimal` interface inherits a few things:

- It gains the `name` property from the `Named` interface.
- It gains the `speak()` function from the `Speaker` interface.

So even though the `FarmAnimal` interface doesn't explicitly declare those members, it inherited them from the other interfaces.

Similarly, when extending an abstract class, the subclass will inherit the functions and properties from the superclass. So, the `Clunker` subclass contains a function called `accelerate()`, even though it's not explicitly declared in its class body, because it inherits the function from `Car`.



For what it's worth, it technically also contains `acceleration`, `speed`, and `makeEngineSound()`, but since those are private, they won't be visible to the subclass. We'll see how to work around this in a moment.

Interface and Implementation

To understand inheritance, it can be helpful to distinguish between two parts of a class:

1. The visible function and property signatures (that is, their names, parameter types, and return types) of a class make up its **interface**. This term can be confusing, because languages like Kotlin also include a code element called an **interface**, which we covered in [Chapter 12](#). So, when we talk about the “interface of a class”, it’s not always clear whether we’re talking about the public surface area of the class, or an actual **interface** in the class declaration.
2. The code in the *body* of a function or property is called its **implementation**.

To help visualize this, let’s look at the code for one of the first classes we wrote, way back in [Chapter 4](#) - a Circle.

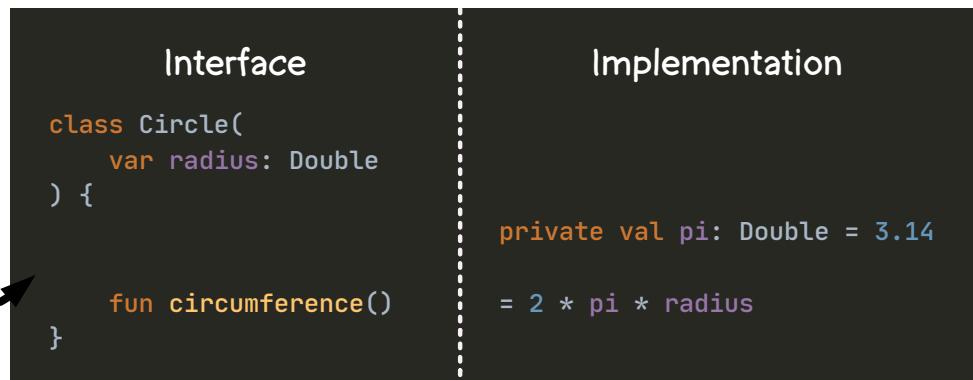
```

class Circle(
    var radius: Double
) {
    private val pi: Double = 3.14

    fun circumference() = 2 * pi * radius
}
  
```

Listing 14.5 - A simple Circle class that we created in Chapter 4.

Now, let’s take the exact same code, but indent the implementation off to the right, in order to help distinguish between the *interface* and *implementation*.



- We can think of an interface as the way a class looks from the “outside” - its name, its properties and their types, its functions and their parameter and return types, and so on.
- The implementation, on the other hand, is what a class looks like from the “inside” - its functions and properties that are *not* externally visible, the *body* of its functions and properties, and so on. Essentially, its inner workings.

So, when we say that a class is *implementing* an **interface** (referring to the code element here), what we really mean is that it’s providing an *implementation* - that is, the code in the body - for each function and property that the **interface** declares.

Speaker interface

```
interface Speaker {
    fun speak()
}
```

Cow's Interface

```
class Cow : Speaker {
    override fun speak()
}
```

Cow's Implementation

```
= println("Moo!")
```

When a class inherits from an **interface** or a class, what *exactly* does it inherit? The interface, or the implementation?

- In some cases, it just inherits the *interface* - that is, the function and property signatures. For example, when an **interface** does not include a default implementation, the class inherits the interface, but must provide its own implementation, such as in the `Cow` code above.
- In other cases, it also inherits the *implementation* of those functions and properties. For example, when an interface includes a default implementation, the inheriting class can inherit that implementation, such as in the following code.

```
interface Speaker {
    fun speak() = println("...")
}

class Cow : Speaker
```

Listing 14.6 - Inheriting a default implementation from an interface.

As we’ll see in a moment, these two things hold true when extending an abstract class, as well.

When a subclass inherits an *implementation* from its superclass, it might also have an opportunity to replace or augment the implementation that the superclass provides. This is called **overriding**,¹ and it’s how we can customize the sound of a `Clunker`’s engine! Let’s look at overriding next.

Overriding Members

¹ As you might recall, we did the same kind of thing when we used class delegation in the last chapter, and the term “override” is the same as we used then.

Just like when using delegation, you can [override](#) functions and properties from an abstract class in order to specialize the behavior of the subclass - such as to give a `Clunker` a special engine sound! We can't just add the `override` keyword, though, or we'll get a compiler error.

```
class Clunker(acceleration: Double) : Car(acceleration) {
    override fun makeEngineSound() = println("putt-putt-putt")
}
```

Listing 14.7 - Error: makeEngineSound defined in Clunker has no access to makeEngineSound defined in Car, so it cannot override it.

The problem here is that the `makeEngineSound()` has a `private` [visibility modifier](#) in the superclass, as in Listing 14.3 above. When a function or property is `private`, it's *so private* that even its own subclasses can't see it! We can fix that with a different visibility modifier.

Protected Visibility

A function marked as `private` in the superclass isn't visible in its subclasses. And if you can't see it, you can't override it! Of course, one option is to remove the `private` modifier, which would make it a public function, but if we do that, then it would be possible to make the engine sound without calling the `accelerate()` function, like this:

```
val car = Clunker(0.25)
car.makeEngineSound()
```

Listing 14.8 - Bypassing the accelerate() function, calling the makeEngineSound() function directly.

We only want the car to make an engine sound when *accelerating*. It'd be great if we could make it so that the `makeEngineSound()` function is visible to *subclasses*, but not to any *other* code. For these situations, Kotlin provides another visibility modifier, called `protected`. Let's update `makeEngineSound()` so that it's `protected`:

```
abstract class Car(private val acceleration: Double = 1.0) {
    private var speed = 0.0
    protected fun makeEngineSound() = println("Vrrrrrrr...")

    fun accelerate() {
        speed += acceleration
        makeEngineSound()
    }
}
```

Listing 14.9 - Adding the protected visibility modifier to the makeEngineSound() function, so that it is visible to subclasses.

A function or property marked as `protected` will be visible to both the current class (e.g., `Car`) and its subclasses (e.g., `Clunker`), but invisible to code everywhere else. With this, `makeEngineSound()` is now visible in the `Clunker` subclass. Are we ready to override it now?

```
class Clunker(acceleration: Double) : Car(acceleration) {
    override fun makeEngineSound() = println("putt-putt-putt")
}
```

Listing 14.10 - Error: 'makeEngineSound' in 'Car' is final and cannot be overridden.

We're still getting a compiler error! Remember how classes are *final* by default? Well, it's the same thing with functions... by default, a function is *final*. In other words, it can't be overridden in subclasses unless we *explicitly state* that it's allowed. There are two ways to do this.

Abstract Functions and Properties

The first way is to add the **abstract** modifier to the function or property. When a function is marked with **abstract**...

- It *cannot* be implemented in the abstract class, and...
- It *must* be implemented in the subclass... unless the subclass itself is also abstract!

So, let's remove the function body from `makeEngineSound()`, and add the **abstract** modifier to it.

```
abstract class Car(private val acceleration: Double = 1.0) {
    private var speed = 0.0
    protected abstract fun makeEngineSound() // no body allowed here!

    fun accelerate() {
        speed += acceleration
        makeEngineSound()
    }
}
```

*Listing 14.11 - Adding an **abstract** modifier to the `makeEngineSound()` function, and removing its implementation.*

With this, we can *finally* override the `makeEngineSound()` function:

```
class Clunker(acceleration: Double) : Car(acceleration) {
    override fun makeEngineSound() = println("putt-putt-putt")
}
```

Listing 14.12 - Overriding the `makeEngineSound()` function in the `Clunker` subclass.

And now we can instantiate and accelerate a clunker...

```
val clunker = Clunker(0.25)
clunker.accelerate()
```

Listing 14.13 - Instantiating and accelerating a `Clunker`.

...which makes that *putt-putt-putt* sound that follows Old Man Keaton around everywhere he goes!

putt-putt-putt

Again, marking a function or property as **abstract** means that each non-abstract subclass *must* implement it. But what if you want **Car** to have a default implementation for **makeEngineSound()**, so that subtypes don't *have* to override it? For this, we have to turn to a different modifier, which we'll explore next.

Open Functions and Properties

The second way to allow subclasses to override a function or property is to mark it as **open**. Open members can have a default implementation in the superclass, so that subclasses don't *have* to provide their own implementation. But they can if they want to. Let's change our **makeEngineSound()** function so that it's **open** instead of **abstract**, and add the body to that function again.

```
abstract class Car(private val acceleration: Double = 1.0) {  
    private var speed = 0.0  
    protected open fun makeEngineSound() = println("Vrrrrrrr...")  
  
    fun accelerate() {  
        speed += acceleration  
        makeEngineSound()  
    }  
}
```

Listing 14.14 - Changing the `makeEngineSound()` function from `abstract` to `open`, which involves adding the body to the function again.

With this change, we can run the code from Listing 14.13 again, and we'll get the exact same result, because **Clunker** still overrides the **makeEngineSound()** function.

Let's introduce *another* subclass that does *not* override it.

```
class SimpleCar(acceleration: Double) : Car(acceleration)
```

Listing 14.15 - A subclass of `Car` that does not override the `makeEngineSound()` function.

When we instantiate it, and call **accelerate()**...

```
val car = SimpleCar(1.2)  
  
car.accelerate()
```

Listing 14.16 - Instantiating and calling `accelerate()` on a `SimpleCar` class.

...it'll use the default engine sound of "Vrrrrrrr..."

Vrrrrrrr...

So, to summarize, abstract classes can be extended by other classes. Their functions and properties can be:

- **abstract**, in which case they have no body in the abstract class, but subclasses *must* implement them.
- **open**, in which case they have a body in the abstract class, but subclasses *may* override them.
- Final (i.e., neither **abstract** nor **open**), in which case subclasses *cannot* override them.

There's still one problem with our code. As mentioned earlier, like an interface, an abstract class doesn't let you instantiate it directly.

```
val myCar = Car()
```

Listing 14.17 - Error: Cannot create an instance of an abstract class.

Instead, you have to instantiate one of its subclasses. To fix this, instead of making `Car` an *abstract* class, we can consider making it an **open class**.

Introduction to Open Classes

An open class is a class that can be both extended *and* instantiated directly. We can change our `Car` class from an abstract class to an open class by simply replacing the keyword **abstract** with the keyword **open**:

```
open class Car(private val acceleration: Double = 1.0) {  
    // ...  
}
```

*Listing 14.18 - Using the **open** modifier on the `Car` class.*

With this simple change, we can now instantiate a `Car` directly.

```
val myCar = Car()
```

Listing 14.19 - Directly instantiating a `Car`.

There's a catch, though - while an open class can have functions and properties that are either **open** or final, it cannot contain any that are **abstract**. That makes sense, though - imagine if this open `Car` class had an abstract function called `honk()`, which naturally could have no body. Now, if we were to instantiate and call `honk()` on the car, what could we possibly expect to happen?

So again, open classes cannot contain **abstract** members. Next, let's look at how we can use a visibility modifier to give subclasses special access to the functions and properties of a superclass.

Getter and Setter Visibility Modifiers

Let's create another subclass of `Car`. This one's a muscle car, and the sound of its engine depends on how fast it's going. Unfortunately, when we try to reference the `speed` variable, we get a compiler error:

```
class MuscleCar : Car(5.0) {
    override fun makeEngineSound() = when {
        speed < 10.0 -> println("Vroooooom")
        speed < 20.0 -> println("Voooooooooom")
        else           -> println("Vooooooooooooooooooooom!")
    }
}
```

Listing 14.20 - Error: Cannot access 'speed': it is invisible (private in a supertype) in MuscleCar.

The problem is that, in the `Car` class, the `speed` property has a `private` visibility modifier on it.

```
open class Car(private val acceleration: Double = 1.0) {
    private var speed = 0.0
    // ...
}
```

Listing 14.21 - The `private` visibility modifier here prevents subclasses from seeing the `speed` property.

As we saw earlier, we can use the `protected` modifier so that subclasses can see the `speed` property.

```
open class Car(private val acceleration: Double = 1.0) {
    protected var speed = 0.0
    // ...
}
```

Listing 14.22 - Changing the visibility of the `speed` property from `private` to `protected`.

With this change, our `MuscleCar` code from Listing 14.20 now compiles just fine!

Let's not celebrate just yet though. With this change, subclasses can now *bypass* the `accelerate()` function, and directly set the speed to anything they want!

```
class Clunker(acceleration: Double) : Car(acceleration) {
    override fun makeEngineSound() {
        println("putt-putt-putt")
        speed = 999.0 // Yikes! Shouldn't be able to increase the
                      // speed without calling accelerate()!
    }
}
```

Listing 14.23 - Bypassing the `accelerate()` function by setting the value of `speed` directly.

What we really want here is to let the subclasses *get* the `speed` value but prevent them from *setting* it. Thankfully, in Kotlin, a property's *getter* can have a different visibility modifier from its *setter*. The syntax can seem a little unnatural at first, but here's how it looks:

```
open class Car(private val acceleration: Double = 1.0) {
    protected var speed = 0.0
        private set
    // ...
}
```

Listing 14.24 - Making the visibility of the `speed` property `protected` on its getter and `private` on its setter.

This code indicates that:

- The `speed` property is `protected`, so subclasses of `Car` can get its value.
- The `speed` property's *setter* visibility is `private`, which means only the `Car` class itself can set the value.

By the way, if you prefer to keep everything on one line, you can just use a semicolon to separate them, like this:

```
open class Car(private val acceleration: Double = 1.0) {
    protected var speed = 0.0; private set
    // ...
}
```

Listing 14.25 - Setting different visibility for a property's getter and setter on a single line.

For what it's worth, this is one of two occasions when I might use a semicolon in Kotlin. The other is when adding functions to an [enum class](#).

Combining Interfaces and Abstract/Open Classes

As we saw in [Chapter 12](#), a class can implement multiple interfaces. It's also possible to implement interfaces and extend a class. To do this, just separate the names of the interfaces and/or superclass with a comma, like this:

```
class NamedCar(override val name: String) : Car(3.0), Named
```

Listing 14.26 - Declaring that `NamedCar` both extends a class and implements an interface.

The biggest critical difference between interfaces and abstract/open classes is that **a subclass can only extend one class**. Implement as many interfaces as you want, but you're stuck with no more than one superclass.¹ This is why interfaces can be much more flexible than abstract and open classes.

So, when should we use interfaces, and when should we use abstract or open classes?

¹Like many other programming languages, Kotlin does not allow multiple class inheritance because of the ambiguity created when two superclasses have different implementations of the same function. For more information about this, see *The Diamond Problem* in Wikipedia's article on Multiple Inheritance.

Comparing Interfaces, Abstract Classes, and Open Classes

Between interfaces, abstract classes, and open classes, there are a lot of options for creating subtypes, and it can be hard to know which option is the most appropriate for different circumstances. Software design decisions like this are the subject of many books (and many debates!).

Although software analysis and design aren't in scope for this book, it's still worth summarizing the significant characteristics of each option, so I've included the following handy-dandy chart to help get you pointed in the right direction!

Characteristic	Interface	Abstract Class	Open Class
Can inherit from it?	Yes	Yes	Yes
Can inherit from multiple?	Yes	No	No
Can be instantiated directly?	No	No	Yes
Can include non-implemented members?	Yes	Yes	No
Can include default implementation?	Yes	Yes	Yes

Subclasses and Substitution

As [mentioned](#) back in Chapter 12, we can use a subtype anywhere that the Kotlin code expects a supertype. This is true not only for interfaces, but also for abstract and open classes. So, we can [explicitly specify](#) the type of a variable as a *superclass* (e.g., `Car`), while actually assigning an instance of a *subclass* (e.g., `MuscleCar`).

```
val car: Car = MuscleCar()
```

Listing 14.27 - Assigning a MuscleCar to a variable that has a type of Car.

The same holds true for calling a function.

```
fun drive(car: Car) {  
    // ...  
}  
  
drive(MuscleCar())
```

Listing 14.28 - Sending a MuscleCar object to a function that has a Car parameter.

If a function has a parameter of type `Car`, it will happily receive a `MuscleCar`, because - by definition - the subclass has at least all the same functions and properties as its superclass. It could have *more* than its superclass, but it will never have fewer.

This ability to use a subtype where a supertype is expected, along with the ability of the subtypes to override functions and properties, is called polymorphism¹. It's a big word that, apart from software development, probably doesn't mean anything to you (unless you happen to be a biologist), but it's still important to know, because it's considered one of the pillars of object-oriented programming.

¹ More precisely, this is called **subtype polymorphism**. There's another kind called "parametric polymorphism", which we typically just refer to as "generics".

Class Hierarchies

So far, every subclass we've created has been a final class, but it's entirely possible for a subclass itself to *also* be an abstract or open class. For example, a clunker that doesn't drive at all might be classified as a "junker". To accommodate this, we could make `Clunker` an open class, and extend it with a new class called `Junker`.

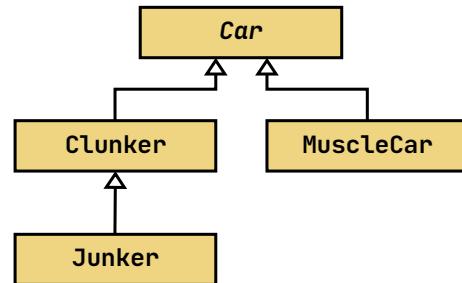
```
open class Clunker(acceleration: Double) : Car(acceleration) {
    override fun makeEngineSound() = println("putt-putt-putt")
}

class Junker : Clunker(0.0)
```

Listing 14.29 - Updating `Clunker` so that it is both a subclass and a superclass.

Now, `Clunker` is *both* a subclass of `Car` and a superclass of `Junker`. Once you've got more than a few classes, it can be helpful to visualize the relationships of the different classes with a UML class diagram, as shown on the right.

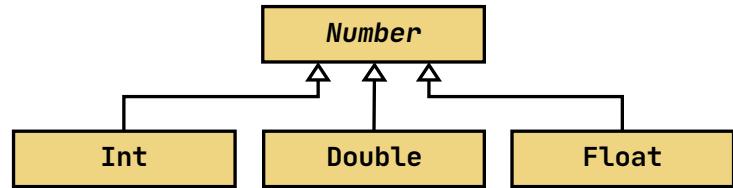
This visualization makes it easy to see how these classes are related in a **class hierarchy**, where the general classes are toward the top, and as you go down the diagram, the classes become more specific. The **depth** of a class hierarchy is determined by how many layers of classes there are in the hierarchy. In the diagram above, we see three layers of depth.



Generally, it's a good idea to limit the depth of a class hierarchy to only a few layers. Otherwise, it gets hard to keep track of which superclasses are providing the different functions and properties, which subclasses are depending on them, and in what ways they're depending on them.

The Any Type

Supertypes and subtypes are not limited to our own classes and interfaces. Many of the classes in Kotlin's standard library implement interfaces and extend abstract or open classes. For example, the basic number types like `Int`, `Double`, and `Float` are all subclasses of an abstract class called `Number`.

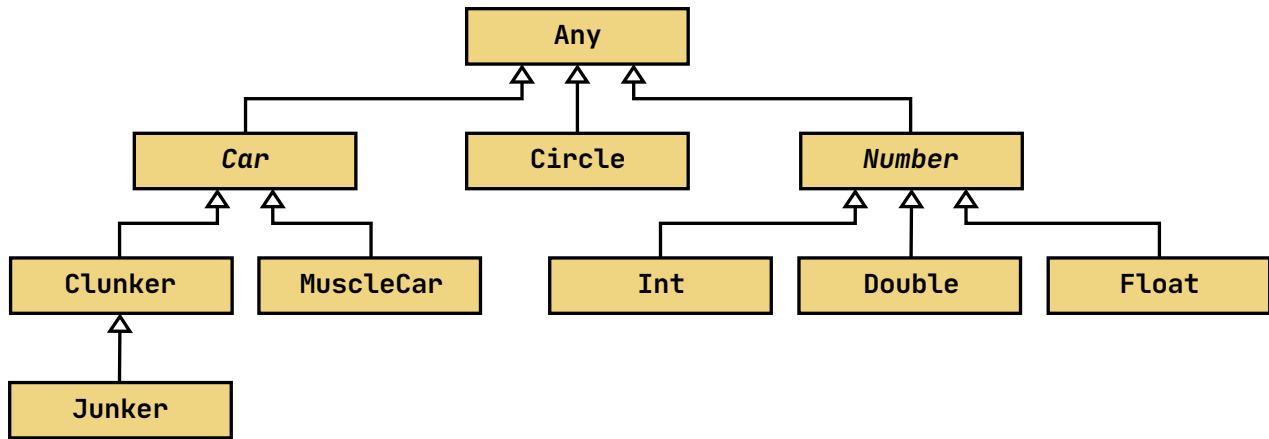


In fact, every Kotlin class that you write will have at least one superclass. For example, way back in Listing 4.1, we created the simplest class possible:

```
class Circle
```

Listing 14.30 - The simplest class, from Chapter 4.

Even though this class doesn't *explicitly* extend a class, it still *implicitly* extends an open class called `Any`. This class is at the very top of the class hierarchy in Kotlin, so even classes that are otherwise unrelated have the `Any` class in common with each other.



The `Any` class provides a few essential functions that are inherited by all other classes - `equals()`, `hashCode()`, and `toString()`. These three functions can be overridden, but it's not very often that we need to do so, because Kotlin has a special kind of class that will override those functions for us, with the implementations that we usually need. We'll learn all about that in the next chapter, as we explore *data classes*!

Summary

Congratulations for completing this large chapter! Here's what you learned:

- What [abstract classes](#) are.
- What it means to [extend](#) a class.
- How functions and properties are [inherited](#) from superclasses.
- The difference between the concepts of [interface and implementation](#).
- How to [override](#) functions and properties from a superclass.
- How [protected visibility](#) gives subclasses access to the members of a superclass.
- The difference between [abstract members](#) and [open members](#).
- How [open classes](#) can be instantiated and include default implementations.
- How getters and setters can have different [visibility modifiers](#).
- How one class can both [extend a class and implement interfaces](#).
- How classes form a [class hierarchy](#).
- How the [Any](#) type is at the top of the class hierarchy in Kotlin.

In Chapter 15, we'll look at data classes and destructuring.

Kotlin: An Illustrated Guide

Chapter 15

Data Classes and Destructuring



Lots of Power with Just One Modifier!

At the end of the last chapter, we saw how all objects in Kotlin inherit three functions from an open class called `Any`. Those functions are `equals()`, `hashCode()`, and `toString()`.

In this chapter, we're going to learn about *data classes*, which are super-powered classes that are especially helpful when you've got an immutable class that mainly just holds properties.

In order to best understand data classes, let's first visit each of the three functions above, and see what's involved when we override them!

Overriding equals()

Reference Equality

Two fathers were crossing paths at the local park, with their daughters in tow, when each accidentally let go of his daughter's hand.



Both of these girls were named Fiona. However, despite having the *same* name, they were two *different* girls, so when the fathers turned around, it was important that each collected his own daughter, not just the first girl he saw who was named Fiona!



Similarly, in Kotlin, there are times when we want to check to see if an object is the instance that we want. To do this, we can use the equality operator, which is two adjacent equal signs `==`. Let's demonstrate this by creating a class to represent the two girls, and instantiate an object for each.

```
class Child(val name: String)

val fiona1 = Child("Fiona")
val fiona2 = Child("Fiona")

println(fiona1 == fiona2) // false
```

Listing 15.1 - Two variables are not considered equal, even though their objects have the same property values.

When using the equality operator on these two objects, we see that they are *not* equal. In other words, they're two *different* `Child` objects.

That's exactly what we want! After all, these two girls are different children, even though they both have the same name.

`val fiona1 =`



`val fiona2 =`



Of course, if we assign the one `Child` object to two different variables, then the equality operator will indicate that those two variables are equal, because they both refer to the same exact object instance.

```
val fiona1 = Child("Fiona")
val fiona2 = fiona1

println(fiona1 == fiona2) // true
```

Listing 15.2 - Two variables are considered equal when they refer to the same object.

In the code above, `fiona1` and `fiona2` are both assigned the same object, so they're equal.

This kind of equality is sometimes called **reference equality**,¹ because as long as the two variables *refer* to the same object instance, then they will be considered equal.

By default, Kotlin uses reference equality when we use the equality operator to compare two objects.

`val fiona1 =`



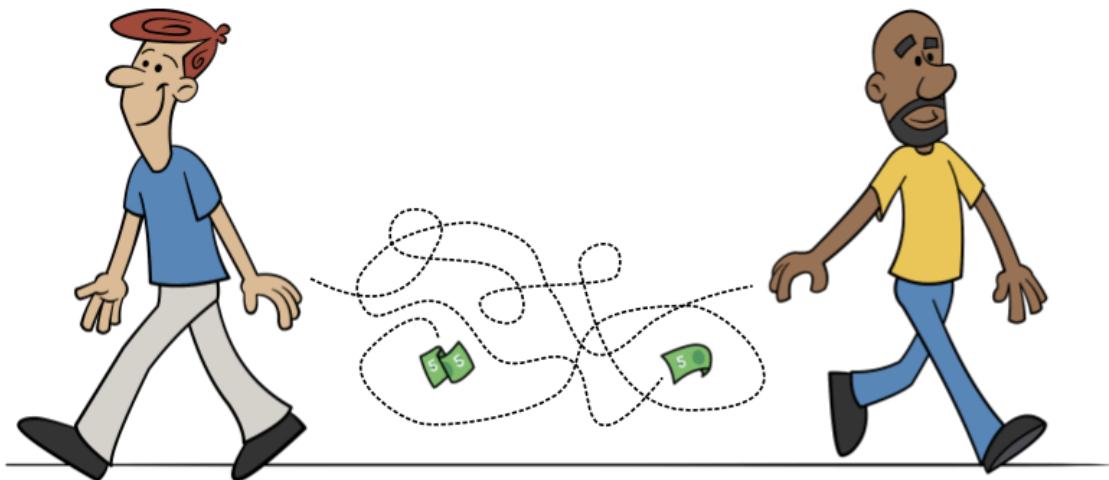
`val fiona2 =`



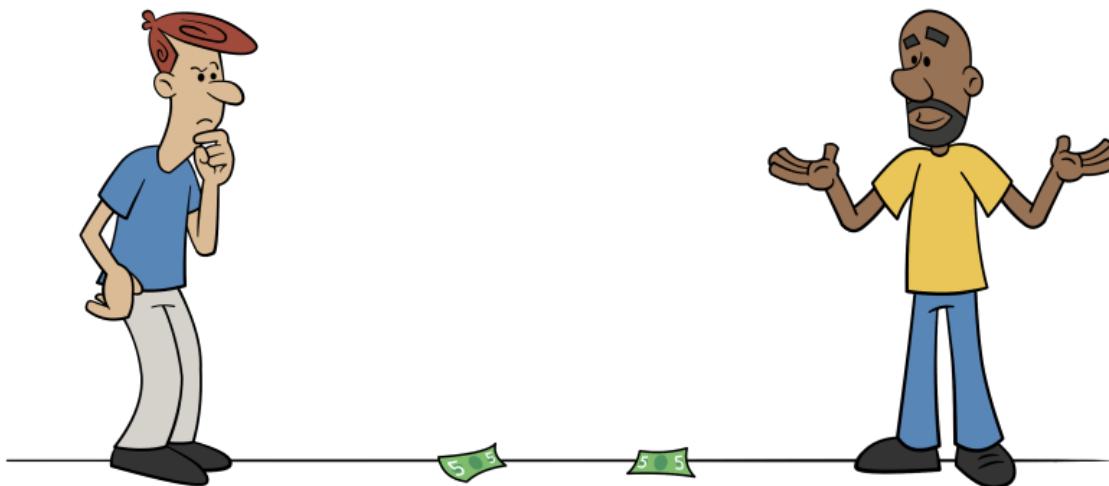
Value Equality

The next day, the two fathers were again crossing paths in the park. This time, they each accidentally let go of a *five-dollar bill* at the same time.

¹This is sometimes also called **identity equality**.

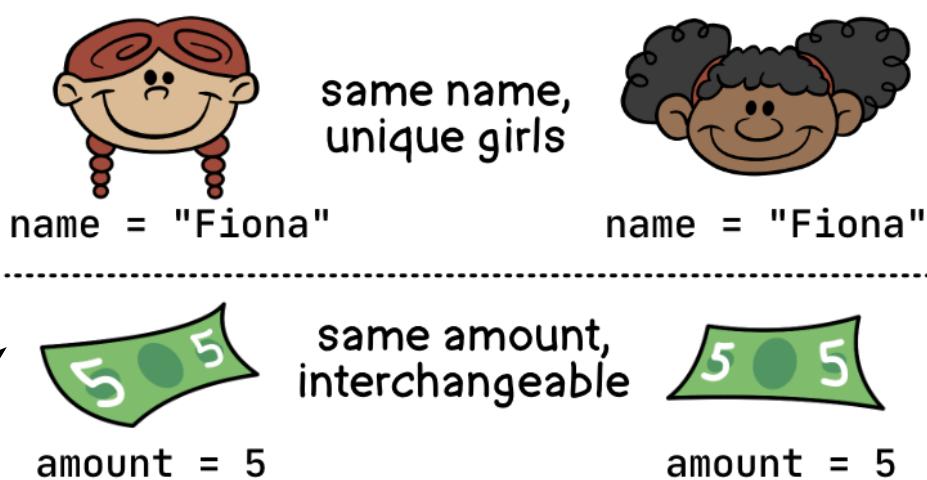


When they turned around, they looked at the dollar bills on the ground, but they weren't sure which bill belonged to which person.



However, each bill is worth the same amount - five dollars. Unlike the previous day when it was important for each father to pick up his own exact daughter, in *this* case, it doesn't matter whether each person picks up the *exact* bill that he dropped. As long as each of them picks up one of the five-dollar bills, it's fine, because the two bills are *equal* to one another.

In other words, some things are interchangeable as long as they have certain *characteristics* that are the same.



Similarly, in Kotlin, when objects are considered equal based on their property values rather than their identity, it's often called **value equality**.¹

Since we would consider two five-dollar bills to be equal to each other, we would probably want a `DollarBill` class to have *value equality* rather than *reference equality*. The following code is roughly the same as Listing 15.1 above, but with a `DollarBill` class instead of a `Child` class.

```
class DollarBill(val amount: Int)

val bill1 = DollarBill(5)
val bill2 = DollarBill(5)

// We want this to be true!
println(bill1 == bill2) // false
```

Listing 15.3 - How can we make it so that two objects are equal based on their property values?

As you can see, when we run this, we get `false`. How can we get this code to print `true`?

Under the hood, when we use the equality operator, it calls into the `equals()` function to determine whether the two objects are considered equal. The implementation of `equals()` that is handed down from the `Any` class will check to see if the two variables *refer to the same object*, which is why reference equality is the default. So if we want the equality operator to act differently for a particular class, we'll need to override the `equals()` function in that class.

In the case of this `DollarBill` class, one simple way to achieve this is to manually delegate the `equals()` call to the `amount` property. Let's try overriding this function in the `DollarBill` class. Note that in the `Any` class, this function has a parameter of type `Any?`, and a return type of `Boolean`, so we'll use the same types in our `equals()` function here.²

```
class DollarBill(val amount: Int) {
    override fun equals(other: Any?) =
        amount.equals(other.amount)
}
```

Listing 15.4 - Error: Unresolved reference: amount

Well, that didn't work. The problem is that the `other` parameter has a type of `Any?`, so that any two objects in Kotlin can be compared to each other, even if their types don't match. Since `other` might not actually be a `DollarBill` object, it won't necessarily have a property named `amount`.

To fix this, we'll need to first check whether the type of `other` is `DollarBill`.

- If it is, then we can just use a smart cast to compare the `amount` values.
- If it isn't, then we can just return `false`.

¹You might also hear this referred to as **content equality** or **structural equality**.

²When overriding a function, we often use the same parameter types and return types that are specified by the same function in the superclass. However, Kotlin also allows you to specify a more general parameter type or a more specific return type. This feature is called **variance**. We'll learn more about this in Chapter 19 when we look at how variance works for generics.

```
class DollarBill(val amount: Int) {  
    override fun equals(other: Any?) =  
        if (other is DollarBill) amount.equals(other.amount) else false  
}
```

Listing 15.5 - Properly overriding the `equals()` function to achieve value equality.

With this change, two `DollarBill` objects will be equal to one another as long as...

1. They're both instances of `DollarBill`.
2. They both have the same value for the `amount` property.

When we run the code again, we'll see that `bill` and `bill2` are now considered equal!

```
val bill1 = DollarBill(5)  
val bill2 = DollarBill(5)  
  
println(bill1 == bill2) // true
```

Listing 15.6 - These two `DollarBill` objects are considered equal because we overrode the `equals()` function.

By the way: The Referential Equality Operator

Even when you override the `equals()` function to give a class *value equality*, you can still check to see whether two variables refer to the same instance.

To do this, you can use the **referential equality operator** instead of the regular equality operator. The referential equality operator is three contiguous equal signs instead of two. Continuing with the code from Listing 15.5 and Listing 15.6 above:

```
println(bill1 == bill2) // true - i.e., according to `equals()` function  
println(bill1 === bill2) // false - i.e., not the same object instance
```

So, by overriding the `equals()` function, we were able to give our `DollarBill` class value equality instead of reference equality! However, there's *another* problem that shows up when we try to use this class with certain collection types. This brings us to the `hashCode()` function.

Overriding `hashCode()`

Little Fiona is starting a collection of contemporary US dollar bills. In order to complete her collection, she's going to need one bill of each of the seven denominations: \$1, \$2, \$5, \$10, \$20, \$50, and \$100.

As you recall from [Chapter 8](#), multiple objects can be stored in a collection type called a Set, which guarantees that each of its elements is unique. In other words, if you try to add an object to a set that it already contains, nothing will change.



We can use a Set to keep track of Fiona's dollar bills.

```
val denominations = mutableSetOf<DollarBill>()
```

Listing 15.7 - Creating a mutable set to hold unique DollarBill instances.

Fiona has collected bills of three different denominations so far: \$1, \$2, and \$5. We can add those to her collection, and print out the size, just to make sure it's got three unique items in it.

```
val denominations = mutableSetOf<DollarBill>()

denominations.add(DollarBill(1))
denominations.add(DollarBill(2))
denominations.add(DollarBill(5))

println(denominations.size) // 3
```

Listing 15.8 - Adding three unique DollarBill objects to a mutable set.

Perfect!

One day, Fiona finds a one-dollar bill, but can't remember if she already collected that one. What happens when we try to add that second one-dollar bill to the set?

```
val denominations = mutableSetOf<DollarBill>()

denominations.add(DollarBill(1))
denominations.add(DollarBill(2))
denominations.add(DollarBill(5))
denominations.add(DollarBill(1)) // duplicate entry!

println(denominations.size) // 4
```

Listing 15.9 - The set includes a DollarBill with the same amount as another in the set.

Yikes! Instead of rejecting the duplicate, the `denominations` set happily included it as a fourth element! Why did this happen? After all, we overrode the `equals()` function in the `DollarBill` class back in Listing 15.5!

Set itself is just an [interface](#), and when we call `mutableSetOf()` it returns an [implementation](#) of `Set`, called `LinkedHashSet`.

A `LinkedHashSet` does not *primarily* use the `equals()` function to determine whether it already contains an object. Instead, it starts with the `hashCode()` function, and only calls `equals()` when an object with the same hash code already exists in that set.¹

This is why we're supposed to override `hashCode()` any time we override `equals()`. In our case, since we're already delegating to the `amount` property for `equals()`, we can just do the same for `hashCode()`.

```
class DollarBill(val amount: Int) {
    override fun equals(other: Any?) =
        if (other is DollarBill) amount.equals(other.amount) else false

    override fun hashCode() = amount.hashCode()
}
```

Listing 15.10 - Overriding the `hashCode()` function.

Easy! With this change, when we run Listing 15.9 again, the set will correctly disregard the duplicate one-dollar bill, resulting in a size of 3 instead of 4.

```
val denominations = mutableSetOf<DollarBill>()

denominations.add(DollarBill(1))
denominations.add(DollarBill(2))
denominations.add(DollarBill(5))
denominations.add(DollarBill(1)) // duplicate entry!

println(denominations.size) // 3 - Success!
```

Listing 15.11 - The set now omits the duplicate entry, because we overrode the `hashCode()` function.

Well, our `DollarBill` class is *almost* doing everything we want. Before we see how data classes can make our lives easier, let's override that third and final function from the `Any` class, `toString()`.

Overriding `toString()`

Although we've used the `println()` function frequently, we haven't examined it closely yet. As you know, we can pass an argument to this function, and it prints it out to the screen. We've often passed it a string, like this:

```
println("Hello, Kotlin!")
```

Listing 15.12 - Passing a string to the `println()` function.

However, we aren't limited to passing strings; we can pass literally *any* object to this function! For example, we can send it an instance of our `DollarBill` class.

¹ Note that this same problem also applies to maps, because `mutableMapOf()` returns a hash-based implementation called `HashMap`.

```
println(DollarBill(100))
```

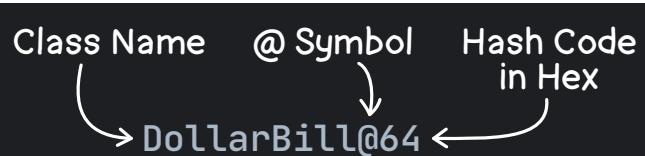
Listing 15.13 - Passing an object to the `println()` function.

When we run this, we'll see this printed out:

```
DollarBill@64
```

When we pass an object to the `println()` function, it will call into the `toString()` function on that object. By default, this function returns a string that looks like the one above. This string has three parts to it:

1. The name of the class
2. An @ sign
3. The hash code of the object, in hexadecimal¹



As you recall, the `DollarBill` class is delegating the `hashCode()` call to the `amount` integer in Listing 15.10. For an integer, the hash code is simply the value of the integer itself. So, if we assign the value 100 to an integer variable, its hash code is also 100. However, since `toString()` converts that hash code to hexadecimal, it appears as 64.

The `toString()` function in this class would be much more helpful if it were to show the name of the class and the value of the `amount` property in *decimal* rather than *hexadecimal*. While we're at it, it'd be helpful to include a label indicating what that number represents (i.e., "amount").

Let's override the `toString()` function and return a string that includes those things.

```
class DollarBill(val amount: Int) {
    override fun equals(other: Any?) =
        if (other is DollarBill) amount.equals(other.amount) else false

    override fun hashCode() = amount.hashCode()

    override fun toString() = "DollarBill(amount=$amount)"
}
```

Listing 15.14 - Overriding the `toString()` function.

After making this change, when we call `println()` with a `DollarBill` object, we'll get more helpful output.

```
println(DollarBill(100))
```

Listing 15.15 - Calling `println()` with a `DollarBill` object.

```
DollarBill(amount=100)
```

¹Most of us have grown up using numbers where each digit has one of ten possible values - 0 through 9. Since there are ten possible values, this is sometimes called the Base-10 or decimal numeral system. Hexadecimal is a numbering system where each digit can have up to 16 values, so it's also called Base-16. After 0-9 come A, B, C, D, E, and F. For example, the number "C" in hexadecimal represents the number 12 in decimal, and the number 10 in hexadecimal represents decimal 16.

Fantastic!

Well, we've made a lot of changes to our `DollarBill` class now. Let's review everything that we did.

1. We made it so that two instances of `DollarBill` are equal to each other, as long as they have the same `amount` values. In other words, we gave it *value equality* instead of *reference equality*.
2. We made it so that they are also treated as equal in sets and maps.
3. We created a more helpful implementation of `toString()`.

We achieved all of these things by overriding the three functions from the `Any` class - `equals()`, `hashCode()`, and `toString()`. Now that we understand why we might want to override these functions, it's finally time to see how data classes can make our lives much easier!

Introduction to Data Classes

After all the changes we made, here's the code that we ended up with:

```
class DollarBill(val amount: Int) {  
    override fun equals(other: Any?) =  
        if (other is DollarBill) amount.equals(other.amount) else false  
  
    override fun hashCode() = amount.hashCode()  
    override fun toString() = "DollarBill(amount=$amount)"  
}
```

Listing 15.16 - The full code for the `DollarBill` class, with all three overrides.

The `DollarBill` example has been rather simple. After all, it only has a *single* property. What would happen if we tried to achieve the same things for a class that includes multiple properties? Implementing `toString()` would be straightforward. However, `equals()` would be a bit more difficult, and `hashCode()` would be even *more* involved!

The good news is that whether our class has a single property or a dozen, Kotlin can *automatically* accomplish everything that we've already done in this chapter - overriding `equals()`, `hashCode()`, and `toString()` - and all we have to do is declare our class to be a **data class**. To do this, we just add the keyword `data` before the class declaration, like this:

```
data class DollarBill(val amount: Int)
```

Listing 15.17 - Declaring a data class.

In this code, we haven't provided an override for `equals()`, `hashCode()`, or `toString()`. In fact, this `DollarBill` doesn't even have a *class body* at all! And yet, in just one line, this class does *everything* that Listing 15.16 does!

```
val bill1 = DollarBill(100)
val bill2 = DollarBill(100)

bill1 == bill2           // true
mutableSetOf(bill1, bill2).size // 1
println(bill1)           // DollarBill(amount=100)
```

Listing 15.18 - Demonstrating that a data class accomplishes everything that our three overrides did.

Again, the `DollarBill` class only includes a *single* property, but data classes can easily provide structural equality and a nice `toString()` result for classes that have *multiple* properties. For example, here's an `Address` class with three properties.

```
data class Address(
    val street: String,
    val city: String,
    val postalCode: String
)
```

Listing 15.19 - A data class with three properties.

As this next code listing demonstrates, instances of `Address` use value equality, and they print out nicely.

```
val address1 = Address("123 Maple Ave", "Berrytown", "56789")
val address2 = Address("123 Maple Ave", "Berrytown", "56789")

address1 == address2           // true
mutableSetOf(address1, address2).size // 1
println(address1)
// Address(street=123 Maple Ave, city=Berrytown, postalCode=56789)
```

Listing 15.20 - Using a data class that has three properties.

We've seen how data classes give us a lot of power, automatically generating useful implementations of `equals()`, `hashCode()`, and `toString()`. The superpowers don't stop there, though! Data classes also include a function called `copy()`, which we'll look at next.

Copying Data Classes

The properties of a data class can be declared with either `val` or `var`, but Kotlin developers tend to use data classes primarily for *immutable* data. In other words, it's common to use only `val` properties in a data class.

As you know, when a class has a mutable property, we can simply assign it a new value. For example, here we have a data class that represents a book. Its `title` property is read-only, but its `price` property is mutable.

```
data class Book(val title: String, var price: Int)
```

Listing 15.21 - A simple data class to represent a book.

When the price of the book increases, we can just set its new value.

```
val book = Book("The Malt Shop Caper", 18)

// The price just went up!
book.price = 20
```

Listing 15.22 - Changing the price of a book object.

It's easy enough to change the value of the `price` property when it's declared with `var`. But what about properties that are declared with `val`?

Naturally, we can't change the value of a `val` property, but instead, we can create a *copy* of the entire object, substituting the new value in that copy. This approach is similar to the one we used in [Chapter 8](#) when we created new lists by [adding an element](#) to an existing list with the plus operator.

Let's update the `Book` class so that the `price` property is declared with `val`.

```
data class Book(val title: String, val price: Int)
```

Listing 15.23 - Changing the `price` property so that it's read-only.

With this change, when the price goes up, we can create a new variable called `newBook`, which has the *same title* as the original, but with the *new price*.

```
val book = Book("The Malt Shop Caper", 18)

// The price just went up!
val newBook = Book(book.title, 20)
```

Listing 15.24 - Instantiating a second `Book` variable with the title from the first.

Now, this is easy enough to do with just *two* properties, but the more properties we add, the more tedious it becomes to make a copy. To demonstrate this, let's add four more properties to the `Book` class.

```
data class Book(
    val title: String,
    val price: Int,
    val author: String,
    val width: Int,
    val height: Int,
    val isbn: String,
)

val book = Book("The Malt Shop Caper", 18, "Slim Chancery", 6, 9, "020516918K")

// The price just went up!
val newBook = Book(book.title, 20, book.author, book.height, book.width, book.isbn)
```

Listing 15.25 - Manually copying an object that has lots of properties.

This is a lot of [boilerplate](#), and it's easy to accidentally get something mixed up when relaying the values from the old object to the new one. In the code above, did you notice that the height and width values got swapped?

To avoid all that boilerplate and reduce the likelihood of these kinds of errors, data classes have a powerful function called `copy()`. For the book example above, instead of manually relaying each property to the `Book` constructor, we can just call `copy()` on the original book, and give it a new value for `price`, like this.

```
val newBook = book.copy(price = 20)
```

Listing 15.26 - Using the `copy()` function that is included in data classes.

The `copy()` function has a parameter for each property in the data class. Since our `Book` data class has 6 properties, its `copy()` function has a total of 6 parameters, each of which defaults to the current value of the property.



```
data class Book(           book.copy(  
    val title: String,      _____> title: String = ...,  
    val price: Int,         _____> price: Int = ...,  
    val author: String,     _____> author: String = ...,  
    val width: Int,         _____> width: Int = ...,  
    val height: Int,        _____> height: Int = ...,  
    val isbn: String        _____> isbn: String = ...  
)
```

When calling the `copy()` function, simply include named arguments for any properties that you want to change, and omit arguments for any properties that you want to stay the same. In Listing 15.26, we only provided the `price` parameter, so `newBook` will have a price of `20`, but all other properties will have the same values that they had in the original `book` object.

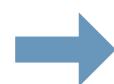
As you can see, the `copy()` function is an incredibly powerful way to work with immutable classes, allowing our code to effectively change data, without actually mutating the individual properties!

There's one more feature that data classes provide, which is the ability to *destructure* its properties. Let's dive in!

Destructuring

When we've got a lot of values that all pertain to some concept, it usually makes sense to put them together. For example, if we've got a title, price, author, width, height, and an ISBN, we usually don't want to deal with them as *individual* variables. Instead, we want them grouped all together in one structure, such as a `Book` class.

```
val title: String  
val price: Int  
val author: String  
val width: Int  
val height: Int  
val isbn: String
```



Book
+ title: String
+ price: Int
+ author: String
+ width: Int
+ height: Int
+ isbn: String

Here's the `Book` data class that we used earlier, which groups together all of the variables mentioned above.

```
data class Book(
    val title: String,
    val price: Int,
    val author: String,
    val width: Int,
    val height: Int,
    val isbn: String,
)
```

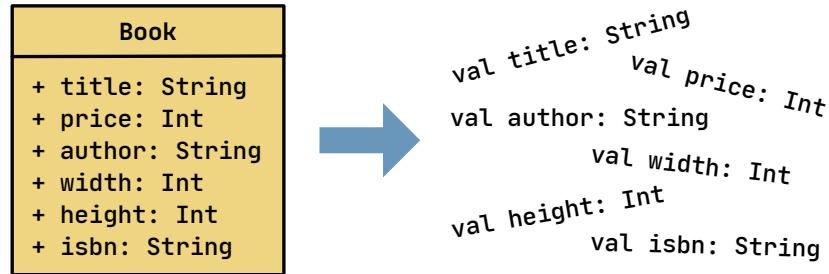
Listing 15.27 - The Book class from Listing 15.25.

When we assemble values into a single class like this, it's clear that these different values are all associated with one another and that, taken together, they represent the concept of a *book*. This usually makes it easier for developers to understand - for example, it's clear that the `title` represents the title of a book, not the title of a movie. This also makes it convenient to pass all of these properties from one function to another - instead of passing *each* property as a separate parameter, we can just pass the `Book` object as a whole.¹

So again, it often makes sense to put associated values *together* into a structure.

Sometimes, however, it makes sense to separate the individual values back out of the structure, so that they're stored in individual variables.

This is called **destructuring**. Of course, we could do this by hand.



For example, in the following code, we pull out all six properties from the book object into separate variables, some of which have names that differ from the original properties.

```
val title = book.title
val cost = book.price
val author = book.author
val widthInInches = book.width
val heightInInches = book.height
val isbn = book.isbn
```

Listing 15.28 - Manually destructuring an object into multiple variables.

When working with a data class, rather than manually extracting each property to a variable, we can use a **destructuring assignment** to pull out each property to a variable *automatically*. To do this, rather than declaring a *single* variable name, declare multiple variable names with commas, and put them all inside parentheses, like this:

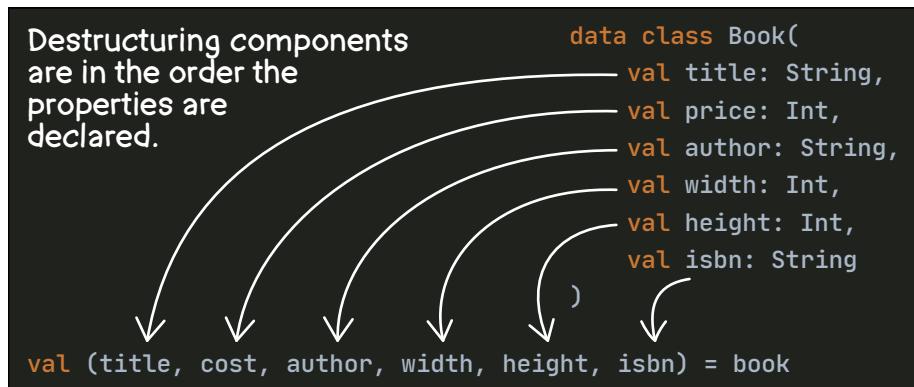
```
val (title, cost, author, widthInInches, heightInInches, isbn) = book
```

Listing 15.29 - A destructuring assignment.

¹Although this is convenient, it's also worth considering how many of the object's values are actually needed in the function. When you pass more values than a function needs, it's called **stamp coupling**, and it can limit the reusability of the function. For example, if the function only needs a title, it might be best to pass only the title rather than the whole book object, because then it could work for more than just book titles - perhaps it will also work for movie and song titles.

This does the same thing as the code in Listing 15.28, but all in a single assignment statement.

Note that the values will be assigned based on the *order* that the properties appear in the primary constructor of the data class. In the case of Listing 15.29 above, this means the `title` variable will be assigned the value of `book.title`, the `cost` variable will be assigned the value of `book.price`, and so forth.



Each of the values that come out of a destructuring assignment - `title`, `price`, `author`, and so on - is called a **component**.¹

Heads Up: Destructuring Risks

When using destructuring assignments, be careful to declare the variables in the correct order. It's easy to mix up the order of the components, such as accidentally swapping the `height` and `width`. One way to help mitigate this risk is to declare the types of each component explicitly, like this:

```
val (title: String, cost: Int, author: String) = // ... and so on ...
```

Naturally, this only helps as long as the types of the components are different. For example, in the code above, if you accidentally put `cost` before `title`, you'd get a compiler error, but if you were to accidentally swap `title` and `author`, there would be no error to call your attention to the mix-up!

Finally, note that we don't have to assign *all* of the components when using a destructuring assignment. For example, if we only need to pull the `title` and `price` out of a `book` object, we can choose to only include two variable names within the parentheses.

```
val (title, cost) = book
```

Listing 15.30 - Omitting some values from a destructuring assignment.

Destructuring and the Standard Library

¹The term "component" is broad and can refer to many things within the disciplines of software development and system architecture. Here, we're just using the term in the very narrow context of destructuring.

Destructuring doesn't only apply to our own data classes. Some of the types in Kotlin's standard library can also be destructured. For example, back in [Chapter 9](#), we used a class called `Pair`, which can be used with destructuring assignments.

```
val association = "Nail" to "Hammer"
val (hardware, tool) = association
```

Listing 15.31 - Code from Listing 9.4

One of the most common ways to use destructuring is with *lambda parameters*. For example, instead of an individual association, let's say we've got a *map* of the hardware and tools.

```
val toolbox = mapOf(
    "Nail" to "Hammer",
    "Bolt" to "Wrench",
    "Screw" to "Screwdriver"
)
```

Listing 15.32 - A map of hardware and the corresponding tools.

When we loop over these tools, the lambda parameter has a type called `Map.Entry`, which has two properties - a `key` and a `value`. Here's how we used it way back in Listing 9.20.

```
toolbox.forEach { entry ->
    println("Use a ${entry.value} on a ${entry.key}")
}
```

Listing 15.33 - Printing the value and key of an Entry object.

A `Map.Entry` object can be destructured, so instead of using `entry` as a parameter for this lambda, we can use parentheses and two variable names, like this:

```
toolbox.forEach { (hardware, tool) ->
    println("Use a $tool on a $hardware")
}
```

Listing 15.34 - Destructuring an Entry object.

When we do this, the argument to this lambda will get destructured into two different variables - the first is the entry's key, and the second is the entry's value. In the code above, the key will be assigned to a variable named `hardware` and the value will be assigned to a variable named `tool`.

Destructuring here is a great idea, because `hardware` and `tool` are terms related to the *problem* that our code solves. We might say that these terms are in the **problem domain** or *business domain*. Contrast these terms with those like `entry`, `key`, and `value`, which are more focused on the data structures that we're using to implement a solution. We might say that those terms are in the **technical domain**.

Now let's consider the case where we want to print out only the *tools* in the toolbox, but not the corresponding *hardware*. Naturally, we could just do this:

```
toolbox.forEach { (hardware, tool) ->
    println("Found a $tool")
}
```

Listing 15.35 - Using only one variable from a destructured lambda parameter.

Here, the `hardware` variable is irrelevant to the lambda. A value is assigned to it, but we're not using it, so it's just noise. In other words, it's something we have to read when looking at this code, but it doesn't affect the way the code works. In cases like this, where one of the components isn't needed, we can substitute an underscore `_` for the name of the irrelevant variable, like this:

```
toolbox.forEach { (_, tool) ->
    println("Found a $tool")
}
```

Listing 15.36 - Using an underscore to avoid creating a variable for the first value when destructuring.

With that change, we no longer have to concern ourselves with a `hardware` variable that isn't used.

By the way...

Underscores are most helpful when we want to omit one of the *earlier* components, while keeping one or more of the *later* components. For example, if we want to print out only the hardware instead of only the tools (the inverse case of Listing 15.36), we *could* use an underscore, like this:

```
toolbox.forEach { (hardware, _) ->
    println("We can service a $hardware")
}
```

However, since we don't have to provide a variable for each component, we could opt to just specify one variable inside the parentheses, like this:

```
toolbox.forEach { (hardware) ->
    println("We can service a $hardware")
}
```

Destructuring Non-Data Classes

Destructuring isn't limited to data classes. Any object can be destructured, as long as its class includes the right functions. The secret is to add functions called `component1()`, `component2()`, `component3()`, and so on. These are called `componentN()` functions. For example, here's the `Child` class from Listing 15.1 at the beginning of this chapter, but this one adds a new property for the child's age.

```
class Child(val name: String, val age: Int)
```

Listing 15.37 - A simple class that represents a child.

If we want to add the ability to destructure this class, we don't have to convert it to a data class. Instead, we can simply add functions called `component1()` and `component2()`, where each one returns one of the properties.

```
class Child(val name: String, val age: Int) {  
    operator fun component1() = name  
    operator fun component2() = age  
}
```

Listing 15.38 - Adding componentN() functions to the Child class.

With this, we can now use destructuring to pull out the child's name and age.

```
val children = listOf(  
    Child("Fiona", 5),  
    Child("Jack", 7)  
)  
  
children.forEach { (name, age) ->  
    println("$name is $age years old.")  
}
```

Listing 15.39 - Destructuring Child objects.

Note that we had to include the `operator` modifier on the two functions in Listing 15.38. This is the first time we've created an **operator function**. When a function includes the `operator` modifier, it can still be called like any other function, but it also serves some special purpose - and the particular purpose that it serves depends upon the name of the function. When a function is named as they are in Listing 15.38 (i.e., like `componentN()`), that special purpose is that the function will be used when the object is destructured.

By the way: Other Operator Functions

Kotlin includes a lot of operator functions that can enhance your classes with new abilities. For example, you can include an operator function named `add()`, and your objects will work with the plus sign.

```
val tenDollarBill = fiveDollarBill + fiveDollarBill
```

The full list of [operator functions](#) can be found in the official Kotlin documentation. I also put together a [video](#) that shows off lots of amusing things you can do with them!

We can even create an [extension function](#) to add destructuring to a class for which we *don't* have the source code. For example, with a little ingenuity, we can create our own extension functions to destructure a `Double` into its integer and fractional parts.

```
operator fun Double.component1() = toString().split(".").first().toInt()
operator fun Double.component2() = toString().split(".").last().toInt()

val (integral, fractional) = 108.245
println(integral) // 108
println(fractional) // 245
```

Listing 15.40 - Using extension functions to enable a `Double` object to be destructured.

So, destructuring can be helpful in certain situations, and data classes are one way to easily add a destructuring capability to our classes!

Limitations of Data Classes

As we've seen in this chapter, data classes give us a lot of convenience!

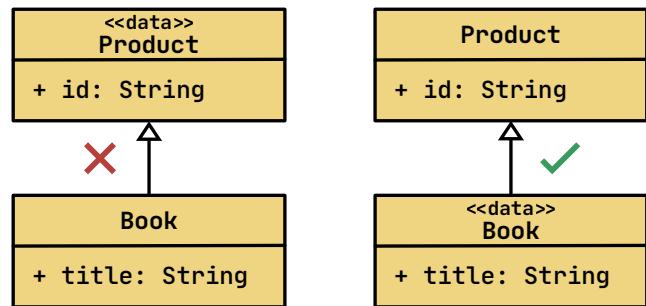
- They create `equals()` and `hashCode()` functions for *value equality*.
- They create nice-looking `toString()` output.
- They make it easy to create copies of an object.
- They can be used with destructuring assignments.

However, we also give up a few things when we declare a class to be a data class, so let's take a quick look at those.

Data Classes and Inheritance

The first and most significant disadvantage of a data class is that it cannot be extended by another class. In other words, you cannot add the `abstract` or `open` modifier to a data class. However, the data class itself *can* extend another class.

Even though a data class can extend another class, if that superclass requires constructor arguments, things get complicated quickly, because of the second disadvantage, which relates to constructor parameters.



Constructor Parameters

The second disadvantage is that all of the constructor parameters in a data class *must* be [property parameters](#). In other words, each one must be declared with either `val` or `var`. This means it's not possible to add a constructor parameter that is *only* relayed to a superclass constructor.

```
open class Product(val id: String)

data class Book(id: String, val title: String) : Product(id)
```

Listing 15.41 - Error: Data class primary constructor must only have property (val/var) parameters.

Also, a data class must have *at least one* parameter in its primary constructor.

Finally, while a data class *may* have properties that are not part of its constructor, they will not be regarded in any of the functions that are generated. For example, in the following code, the `serialNumber` property is not in the primary constructor.

```
data class DollarBill(val amount: Int) {
    var serialNumber: String? = null
}
```

Listing 15.42 - This data class includes a property that is not part of the primary constructor.

A property like `serialNumber` won't be considered in `equals()`, `hashCode()`, or `toString()`, because it's declared in the body of the class, but not in the primary constructor. It won't be possible to call `copy()` with it, and it won't be used for destructuring assignments. To demonstrate this, the following code shows how two `DollarBill` objects are considered equal, even though they have different values for the `serialNumber`.

```
val bill1 = DollarBill(5).apply { serialNumber = "QB12345678T" }
val bill2 = DollarBill(5).apply { serialNumber = "IE87654321C" }

println(bill1 == bill2) // true, despite different serial numbers
```

Listing 15.43 - Demonstrating that serialNumber does not affect equality.

Even with these disadvantages, data classes are tremendously helpful, especially when dealing with immutable classes that are primarily meant to hold properties.

By the way: Data Objects

It makes sense that a data class must have at least one *property parameter* in its constructor, because if it has none, then doesn't really have any data to hold... at least, not any data that affects equality, `toString()`, copying, or destructuring. However, there are still some particular situations where you might want a data class that has no primary constructor parameters, mainly for parity with other data classes in a sealed type hierarchy (which we'll cover in the next chapter). For those cases, you can use a `data object` instead of a `data class`. You can read about [data objects](#) in the official Kotlin documentation.

Summary

In this chapter, we learned all about data classes and destructuring, including:

- The difference between [reference equality](#) and [value equality](#).
- How to manually override [equals\(\)](#) and [hashCode\(\)](#) to achieve value equality.
- How to manually override [toString\(\)](#) so that our classes work well with [println\(\)](#).
- How to [declare a data class](#) so that we don't have to manually override [equals\(\)](#), [hashCode\(\)](#), or [toString\(\)](#).
- How to use the [copy\(\)](#) function to make it easier to create an object based on another object's values.
- How to use a [destructuring assignment](#) when working with data classes.
- How to [enhance a non-data class](#) to enable destructuring assignments.
- How to use [extension functions to enable destructuring assignments](#).
- The [limitations of data classes](#).

In the next chapter, we'll introduce another class modifier, which will enable us to account for every possible subclass of an interface or abstract class.

Kotlin: An Illustrated Guide

Chapter 16

Sealed Types



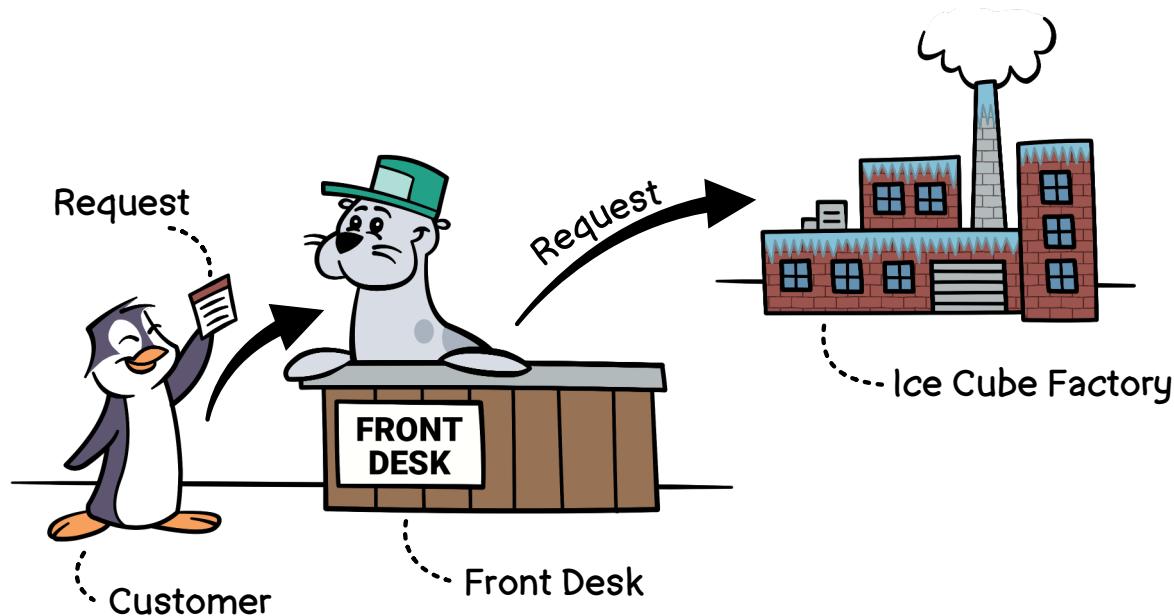
Let's make the compiler even more helpful by limiting our types!

In Chapter 5, we saw how limiting our options could be a good thing. In that chapter, we used enum classes to limit our *values*, which allows Kotlin to ensure that we account for all possibilities in a when expression.

We can get a similar benefit for our *types* by using sealed interfaces and classes. In this chapter, we'll visit *Cecil's Ice Shop* to learn all about sealed types.

Let's get started!

In the frigid lands of the antarctic, there's a store called *Cecil's Ice Shop*, a thriving business where the locals can buy containers of ice cubes in three different sizes. It's a simple operation - when customers want to place an order or request a refund, they show up to the front desk and fill out a request form. From there, the front desk sends the request off to their ice cube factory, which handles the fulfillment.



Cecil, the store's owner, is also modeling out his operations in Kotlin code. To start with, he created an [enum class](#) to represent those three sizes of ice cube packages.

```
enum class Size { CUP, BUCKET, BAG }
```

Listing 16.1 - An enum representing three sizes of ice cube packages.

Next, for the order and refund requests, he created an interface called `Request`. The front desk deals with lots of requests each day, so in order to keep track of them all, each one has a unique ID number. So likewise, his `Request` interface has a property called `id`.

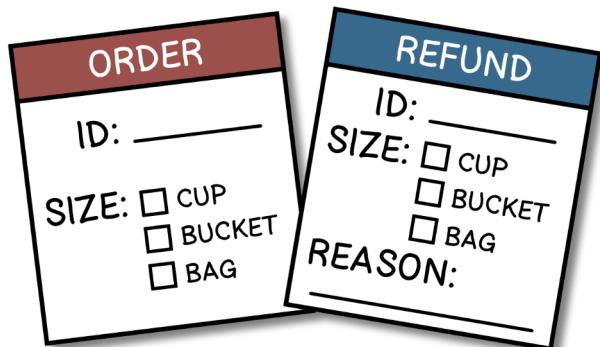
```
interface Request {
    val id: Int
}
```

Listing 16.2 - A simple Request interface with one property.

Next, he added two classes that implement that interface - one for placing an order, and one for requesting a refund.

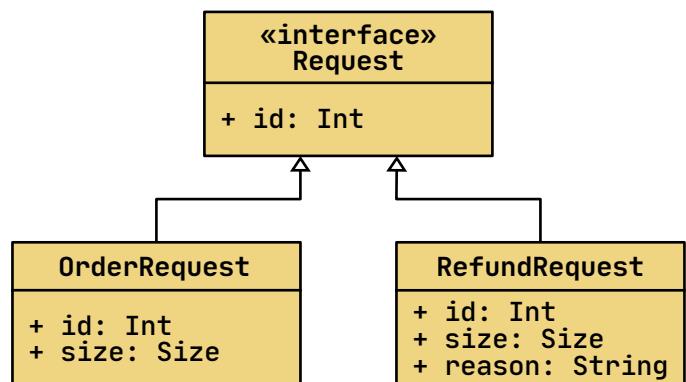
```
class OrderRequest(override val id: Int, val size: Size) : Request
class RefundRequest(override val id: Int, val size: Size, val reason: String) : Request
```

Listing 16.3 - Two classes implementing the Request interface.



Then, in his Kotlin code, Cecil created a `FrontDesk` object to receive a `Request`. The front desk records that it received the request, by printing out its ID number.

After that, he uses a `when` conditional to do a smart cast, and sends the request along to the correct function at the *ice cube factory*, where the request will be fulfilled.



```

object FrontDesk {
    fun receive(request: Request) {
        println("Handling request #${request.id}")
        when (request) {
            is OrderRequest -> IceCubeFactory.fulfillOrder(request)
            is RefundRequest -> IceCubeFactory.fulfillRefund(request)
        }
    }
}
  
```

Listing 16.4 - An object representing the front desk, which can receive requests.

Speaking of the ice cube factory, Cecil isn't too concerned about exactly *how* it handles the orders and refunds. So, in his Kotlin code, he created an `IceCubeFactory` that just prints out a message as each request is being fulfilled.

```

object IceCubeFactory {
    fun fulfillOrder(order: OrderRequest) = println("Fulfilling order #${order.id}")
    fun fulfillRefund(refund: RefundRequest) = println("Fulfilling refund #${refund.id}")
}
  
```

Listing 16.5 - An object representing the ice cube factory, which prints out the orders that it receives.

With this simple Kotlin code, a customer can now order a cup of ice! The front desk receives the order, and forwards it on to the ice cube factory, where the order will be fulfilled.

```

val order = OrderRequest(123, Size.CUP)
FrontDesk.receive(order)
  
```

Listing 16.6 - Passing an order to the front desk, which is forwarded to the factory.

```

Handling request #123
Fulfilling order #123
  
```

And of course, Cecil's code also allows a customer to request a refund, simply by giving the front desk a refund request.

```
val refund = RefundRequest(456, Size.CUP)
FrontDesk.receive(refund)
```

Listing 16.7 - Passing a refund request to the front desk, which is forwarded to the factory.

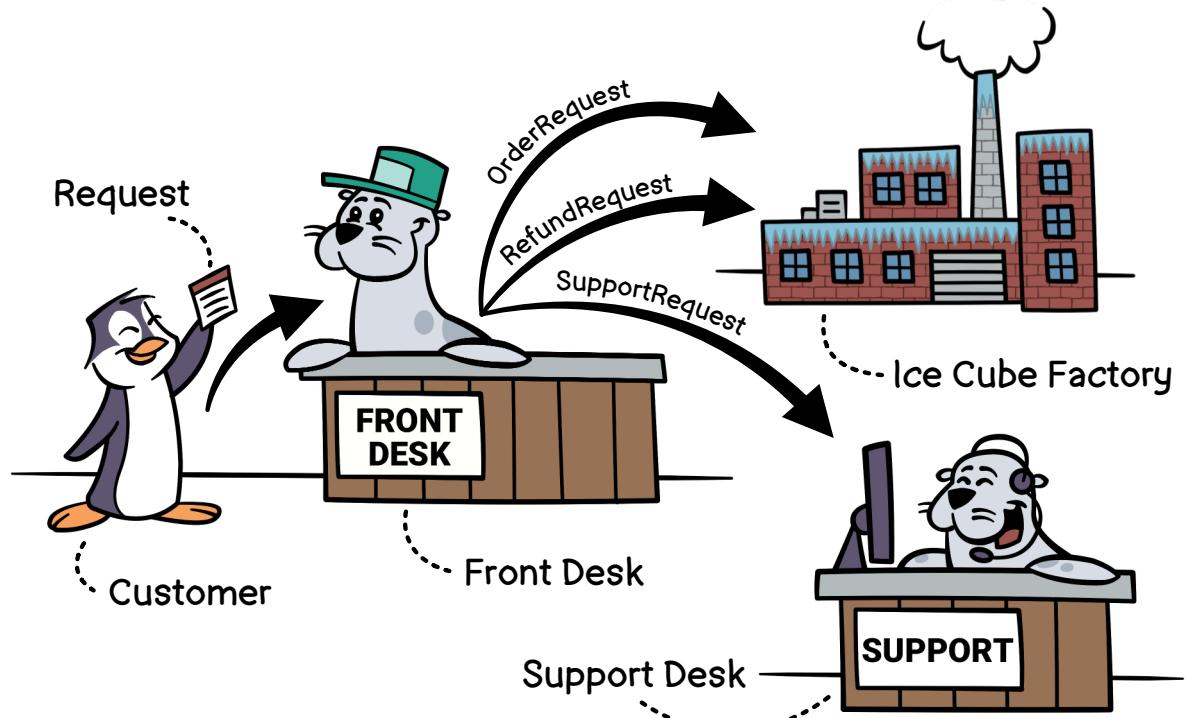
```
Handling request #456
Fulfilling refund #456
```

With this code, Cecil's Ice Shop continued fulfilling orders and refunds, making many satisfied customers... until one day when Cecil needed to add one more request type!

Adding Another Type

One day, a customer named Wallace needed help opening his bag of ice. With that massive body and those slippery flippers, it's hard to blame a walrus for not being able to open that bag!

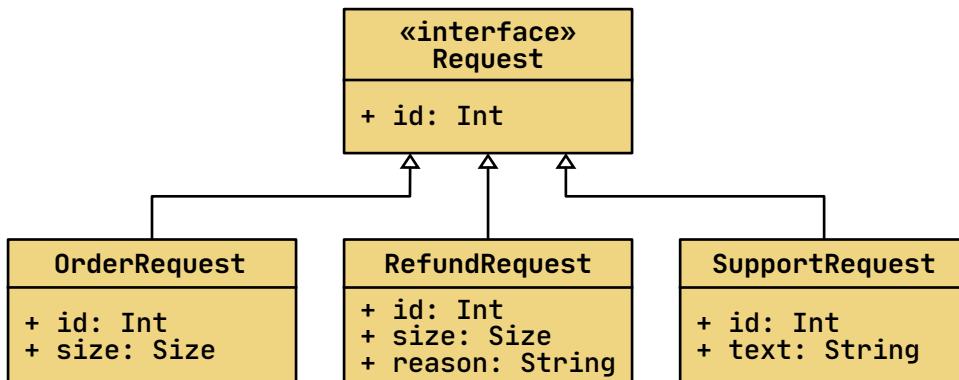
Cecil decided that it was time to start providing support for his customers. In order to help customers like Wallace, Cecil came up with plans to add a new kind of request, called a *support request*. Customers who need help could just fill out a support request with text about the problem that they have, and hand it to the front desk. The front desk would forward it on to a new help desk.



Naturally, it was easy for Cecil to add a new request type in his Kotlin code. He just created a new class called `SupportRequest`, which implemented the `Request` interface. In addition to the `id` property, this new class added only a `text` property, where customers can write a description of the help that they need.

```
class SupportRequest(override val id: Int, val text: String) : Request
```

Listing 16.8 - A third class that implements the Request interface.



As with the ice cube factory, instead of including details about how the help desk *actually* helps, Cecil's code simply included a `println()` statement to log that the request was received.

```
object HelpDesk {
    fun handle(request: SupportRequest) = println("Help desk is handling ${request.id}")
}
```

Listing 16.9 - An object that handles SupportRequest objects. Here it just prints out that it was received.

Great! Cecil hit the “Run” button in his IDE, and his shop was up and running again. Wallace submitted his help request, and was told that someone from the help desk would follow up with him.

```
val request = SupportRequest(789, "I can't open the bag of ice!")
FrontDesk.receive(request)
```

Listing 16.10 - Passing a support request to the front desk.

A few days later, though, Wallace returned, as grumpy as ever. “Why hasn’t anyone contacted me about my support request?” he asked.

Embarrassed, Cecil combed through his program’s output for Wallace’s support request ID number to see what happened. To his surprise, he only found one line about it:

```
Handling request #789
```

“The front desk recorded that it received the request, but that was all! The help desk apparently never received it!” noted Cecil. What happened? Cecil pulled up the code for the front desk again.

```
object FrontDesk {
    fun makeRequest(request: Request) {
        println("Handling request #${request.id}")
        when (order) {
            is OrderRequest -> IceCubeFactory.fulfill(request)
            is RefundRequest -> IceCubeFactory.fulfill(request)
        }
    }
}
```

Listing 16.11 - The front desk does not include a when branch for SupportRequest.

"Of course!" he cried, "I forgot to add a branch to the when conditional for the new `SupportRequest` type!" He would have slapped his forehead, but his flipper couldn't reach his head.

Cecil noticed how easy it is to forget to add a branch to his when conditional when he adds a new subtype. He mused, "I've only got *one* when in my code right now. How much *easier* would it be to forget a branch if I had even *more* of them throughout my code! It's too bad that I only discovered this problem after a *customer* complained about it!"

Instead of waiting for a customer to report a problem with his code, it'd be great if Kotlin could tell him right away, with a compile-time error message. In other words, if he forgets to add a branch to a when, he'd love to know *before* the code ever runs - and *well before* any customer could be affected! Thankfully, Kotlin has a feature that can solve this problem!

Introduction to Sealed Types

As you might recall, when a conditional accounts for *every* possible case, then we say that the conditional is [exhaustive](#). As we saw back in [Chapter 5](#), we can use enum classes to ensure that our when conditionals are exhaustive. For example, if Cecil wants to describe the different sizes of ice packages, he could write something like this:

```
when (size) {
    Size.CUP    -> println("A 12-ounce cup of ice")
    Size.BUCKET -> println("A bucket with 1 quart of ice")
    Size.BAG     -> println("A bag with 1 gallon of ice")
}
```

Listing 16.12 - An exhaustive when conditional.

If one of these branches were missing from this when statement, then Kotlin would give a compiler error.

```
when (size) {
    Size.CUP    -> println("A 12-ounce cup of ice")
    Size.BAG     -> println("A bag with 1 gallon of ice")
}
```

Listing 16.13 - Error: "when expression must be exhaustive, add necessary BUCKET branch or else branch" (does not compile)

This is exactly the kind of compiler error that Cecil would love to see, but instead of a `when` conditional that checks the *value* of a variable, his `when` conditional is checking the *type* of a variable.

```
when (request) {
    is OrderRequest -> IceCubeFactory.fulfill(request)
    is RefundRequest -> IceCubeFactory.fulfill(request)
}
```

Listing 16.14 - How can we get this when conditional to create a similar compiler error?

So, how can Cecil tell Kotlin that he wants this `when` statement to be exhaustive, to make sure that there's a branch for each subtype of the `Request` interface? The secret is to use a feature called a **sealed type**. Using a sealed type is easy - we just add a modifier called `sealed` to our interface or class declarations.

To demonstrate this, let's update Cecil's `Request` interface so that it's sealed. The `sealed` modifier goes just before the `interface` keyword, as shown here.

```
sealed interface Request {
    val id: Int
}
```

Listing 16.15 - Updating the Request interface so that it's a sealed interface.

When a type like `Request` is sealed, Kotlin will keep track of all its direct subtypes. That way, Kotlin can know when you've been exhaustive in a conditional that checks subtypes.

In fact, just by adding the `sealed` modifier to the `Request` interface, it caused a compiler error on the `when` statement.

```
object FrontDesk {
    fun receive(request: Request) {
        println("Handling request #${request.id}")
        when (request) {
            is OrderRequest -> IceCubeFactory.fulfillOrder(request)
            is RefundRequest -> IceCubeFactory.fulfillRefund(request)
        }
    }
}
```

Listing 16.16 - Error: 'when' expression must be exhaustive, add necessary 'is SupportRequest' branch or 'else' branch instead.

Perfect! Just like Cecil wanted, Kotlin now alerts him when he forgets a branch, and since he gets this alert at compile time, he can fix it before any customers are affected. Speaking of fixing it, that's also easy to do - Cecil just adds a branch for `SupportRequest`, and the compiler error goes away.

```
object FrontDesk {
    fun receive(request: Request) {
        println("Handling request #${request.id}")
        when (request) {
            is OrderRequest -> IceCubeFactory.fulfillOrder(request)
            is RefundRequest -> IceCubeFactory.fulfillRefund(request)
            is SupportRequest -> HelpDesk.handle(request)
        }
    }
}
```

Listing 16.17 - Adding a branch for SupportRequest fixes the compiler error.

For what it's worth, this compiler error can also be satisfied by using an `else` branch. However, in the code above, Cecil needs the smart cast in order to send it to the help desk.

And now, the help desk is receiving support requests! Cecil can rest easy, knowing that the help desk is taking care of customers like Wallace.

Sealed Classes

In the example above, we added the `sealed` modifier to an *interface* declaration. However, it's also possible to add it to a *class* declaration. For example, instead of requiring customers to enter an `id` number on each request, Cecil could change `Request` to an abstract class, and automatically assign a random number to it. Since interfaces can't hold state, Cecil would need to change the interface to a class, like this.

```
sealed class Request {
    val id: Int = kotlin.random.Random.nextInt()
}
```

Listing 16.18 - Converting a sealed interface to a sealed class.

With this simple change, he's now using a sealed class instead of a sealed interface. Naturally, this change implies a few updates to the subclasses - like removing the `id` property and calling `Request`'s constructor.

```
class OrderRequest(val size: Size) : Request()
class RefundRequest(val size: Size, val reason: String) : Request()
class SupportRequest(val text: String) : Request()
```

Listing 16.19 - Updating the subtypes so that they extend the Request class (instead of implementing the old Request interface).

Note that a `sealed class` is, by definition, also an `abstract class`. This means that you can't directly instantiate it - you can only instantiate one of its subclasses. The `sealed` modifier also implies the `abstract` modifier. Although it's not an error to include both of them, doing so is redundant and unnecessary. So if you use the `sealed` modifier, omit the `abstract` modifier.

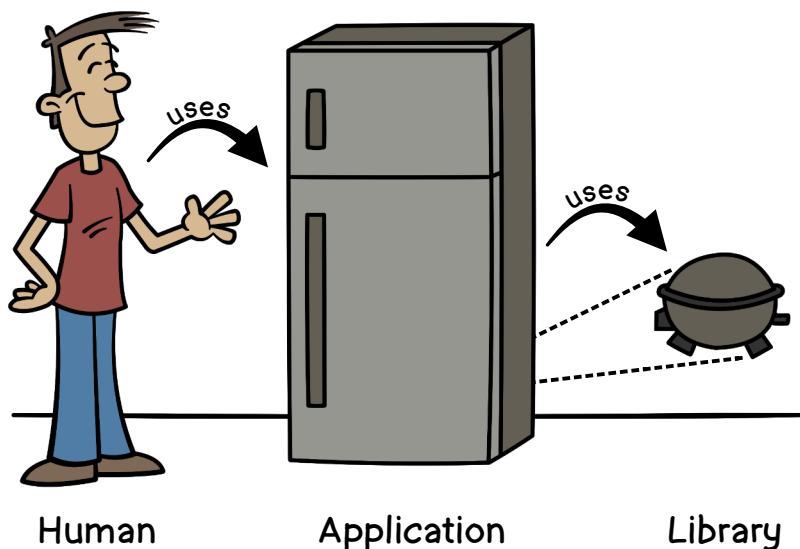
Why Is the sealed Modifier Required At All?

Now, you might be wondering why we need to add the `sealed` modifier to our interface or class declaration. Why can't Kotlin be exhaustive in those `when` statements without it? We'll answer that question, but first, let's talk about refrigerators.

You probably use a refrigerator all the time. You make sure it's plugged in, then you open the door, put something inside for the refrigerator to keep cold, and then you close the door again. A refrigerator is a household *appliance* - it's a piece of equipment that's designed for *humans* to interact with.

Now consider a *compressor*. A compressor is a major *component* of a refrigerator. Without it, a refrigerator won't keep your food cold. However, as a human, you don't *directly* interact with a compressor. You use a refrigerator, and the *refrigerator* uses its compressor.

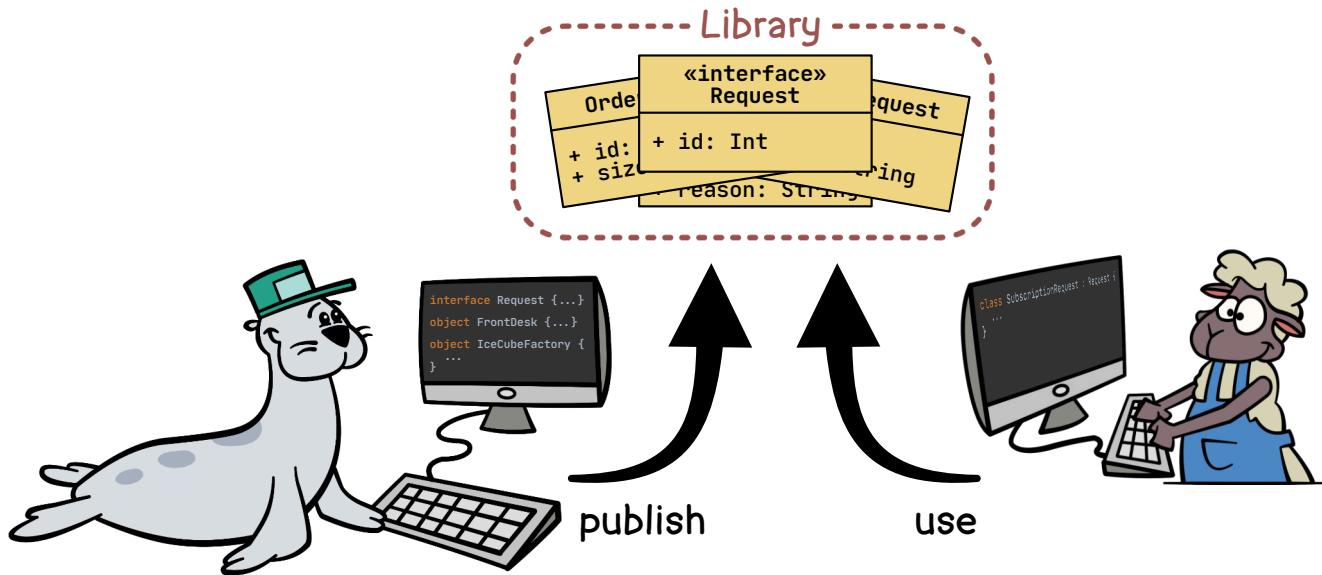
Similar to refrigerators, some code that we write is intended for a *human* to interact with directly. Instead of calling this kind of software an *appliance*, we call it an **application**. On the other hand, similar to a compressor, *other* programs that we write are not intended to be used directly by humans - it's intended that they'll be used as a *component* of an application. This kind of software component is called a **library**.



Throughout this book, you've already been using a library, called the *Kotlin Standard Library*. This library includes basic classes and interfaces, functions that we used for collection processing, and lots more. In fact, just like you can't do much with a refrigerator if it's missing its compressor, you can't do much with a Kotlin project if you don't include the standard library!

It's possible to create a library from your own code, so that other developers can use it.

For example, Cecil could take his code, compile it, and bundle it up into a library that includes his `Request` interface, its subclasses, and the `FrontDesk` and `IceCubeFactory` objects.



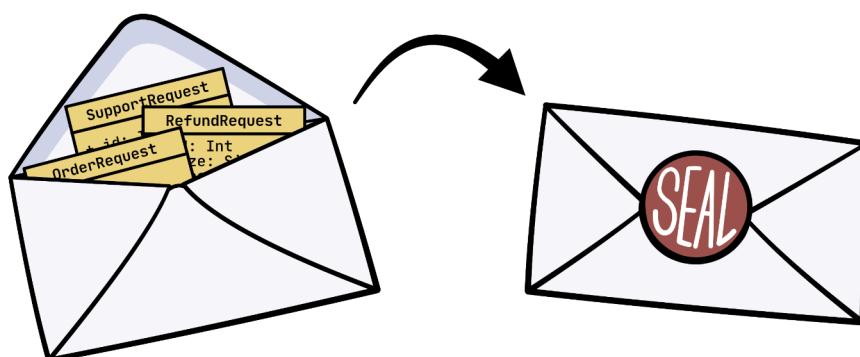
Then, if Bert from *Bert's Snips & Clips* (see [Chapter 7](#)) wants to use that interface, he could include Cecil's library in his code.

When Bert uses Cecil's library, he'd be able to see that there's a `Request` interface, and could create his own subclass of it. For example, he might create a `SubscriptionRequest`, where his customers could subscribe to his mailing list for coupons!

However, the library was *already compiled* before Bert started using it. So, the `FrontDesk` code assumed there would only be *three* subclasses, because that's how many there were when it was compiled. But Bert created a fourth! So, *at the point when Cecil builds his library*, there's no way for Kotlin to know all of the subclasses that Bert - or any other developers - might also create in the future when using that library.

So instead, by adding the `sealed` modifier to the interface, it prevents Bert from being able to add another subtype of `Request` when he uses the library. Cecil can still add one, of course, but anyone *using* the library will be unable to do so.

Marking `Request` as `sealed` is kind of like taking its three subclasses, putting them in an envelope, and then "sealing" the envelope so that anyone else who gets that envelope can't put anything else inside!



In summary, if you want exhaustive subtype matching, you'll need to include the `sealed` modifier, regardless of whether you're building an application or a library.

Restrictions of a Sealed Type's Subtype

As we've seen, sealed types are helpful when you want Kotlin to ensure that you exhaustively match subtypes in a `when` conditional. By design, they come with a few restrictions. Specifically, every direct subtype of a sealed interface or class...

1. ... must be declared in the same *code base*. In other words, if you were to create a library out of your code, anyone using that library would be working in a different code base, and would not be able to subtype it.
2. ...must be declared in the same *package*. Even in the same Kotlin project, the subtypes of a sealed type must all be in the same exact package as the sealed type itself.

For what it's worth, these rules are relaxed compared to what they were back in Kotlin 1.0. Back then, only sealed *classes* were supported (sealed *interfaces* were added in Kotlin 1.5), and all subclasses had to be declared *inside the class body* of the sealed class!

Note that these limitations apply only to *direct* subtypes of the sealed type. If you want to create a subtype of a subtype of a sealed type, you can do so, even if the sealed type is in a library that you're using.

This class is sealed, so its subtypes must be in the same code base and package.

However, these subclasses are not sealed, so their subtypes are not subject to those restrictions.

```
sealed class Request {
    val id: Int = kotlin.random.Random.nextInt()
}

open class OrderRequest(...) : Request()
open class RefundRequest(...) : Request()
open class SupportRequest(...) : Request()
```

For example, Bert can't create a new *direct* subtype of `Request`. However, he could create a subclass of `SupportRequest`, as long as Cecil had marks it as open or abstract. Why are *direct* subtypes restricted but *secondary* subtypes allowed?

Well, let's look at the `FrontDesk` code again.

```
object FrontDesk {
    fun receive(request: Request) {
        println("Handling request #${request.id}")
        when (request) {
            is OrderRequest -> IceCubeFactory.fulfillOrder(request)
            is RefundRequest -> IceCubeFactory.fulfillRefund(request)
            is SupportRequest -> HelpDesk.handle(request)
        }
    }
}
```

Listing 16.20 - The `FrontDesk` code from Listing 16.17.

Let's say Bert is using Cecil's library, and he adds a new direct subtype of `Request`, called `SubscriptionRequest`. In this code, if `FrontDesk.receive()` is called with an instance of `SubscriptionRequest`, *none* of the branches in this `when` conditional would match, so this conditional wouldn't *actually* be exhaustive. So, Kotlin does not allow that.

```
object FrontDesk {    ↗  
    fun receive(request: Request) {        If this could be a SubscriptionRequest...  
        println("Handling request #${request.id}")  
        when (request) {  
            is OrderRequest → IceCubeFactory.fulfillOrder(request)  
            is RefundRequest → IceCubeFactory.fulfillRefund(request)  
            is SupportRequest → HelpDesk.handle(request)  
        }  
    }  
}
```

...then none of these cases would match!

Now, let's say he creates a subtype of `SupportRequest` called `CouponSupportRequest`. In this case, when `FrontDesk.receive()` is called with an instance of `CouponSupportRequest`, then the third branch would match, because `CouponSupportRequest` is a more specific kind of `SupportRequest`. So, the conditional is still exhaustive in this situation.

So again, the two restrictions above apply only to *direct* subtypes, because secondary subtypes won't break the integrity of the conditionals.

Sealed Types vs Enum Classes

As mentioned earlier, it was way back in [Chapter 5](#) that we first saw how Kotlin could tell us when our `when` conditionals are exhaustive, without the need for an `else` branch, as shown here.

```
enum class SchnauzerBreed { MINIATURE, STANDARD, GIANT }  
  
fun describe(breed: SchnauzerBreed) = when (breed) {  
    SchnauzerBreed.MINIATURE -> "Small"  
    SchnauzerBreed.STANDARD -> "Medium"  
    SchnauzerBreed.GIANT -> "Large"  
}
```

Listing 16.21 - Combines the code from Listings 5.4 and 5.8.

And as we've seen in *this* chapter, Kotlin can do the same thing for sealed types.

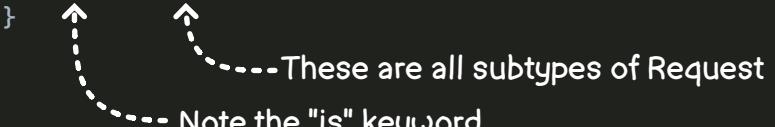
```
when (request) {
    is OrderRequest -> IceCubeFactory.fulfillOrder(request)
    is RefundRequest -> IceCubeFactory.fulfillRefund(request)
    is SupportRequest -> HelpDesk.handle(request)
}
```

Listing 16.22 - Exhaustive matching for sealed types.

It's tempting to see that similarity and conclude that sealed types are just a more sophisticated kind of enum class, but that comparison is usually more confusing than helpful. Sealed types and enum classes have some critical differences that are important to know.

First, there's a difference between *what the conditional is checking*. With a sealed type, your conditional is checking *subtypes* of the sealed type.

```
when (request) {
    is OrderRequest -> IceCubeFactory.fulfillOrder(request)
    is RefundRequest -> IceCubeFactory.fulfillRefund(request)
    is SupportRequest -> HelpDesk.handle(request)
}
```



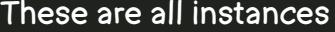
These are all subtypes of Request



Note the "is" keyword

With an enum class, on the other hand, the conditional is not checking *types* - it's checking *values*.

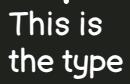
```
fun describe(breed: SchnauzerBreed) = when (breed) {
    SchnauzerBreed.MINIATURE -> "Small"
    SchnauzerBreed.STANDARD -> "Medium"
    SchnauzerBreed.GIANT -> "Large"
}
```



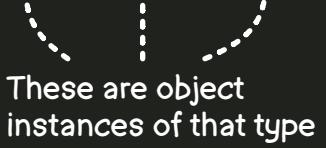
These are all instances

That's because each entry inside an enum class is an *object*, not a *class*.

```
enum class SchnauzerBreed { MINIATURE, STANDARD, GIANT }
```



This is the type



These are object instances of that type

Second, enum classes have a variety of built-in properties and functions that sealed classes don't have. For example:

- You can get the `ordinal` property of an enum entry, but subtypes of a sealed type have no order.

- Enum classes provide the `entries()` function, which allows you to easily iterate over its entries. A sealed type has no such function for its subtypes.¹

For these reasons, it's best *not* to think of sealed types as a more sophisticated kind of enum class. They achieve a similar effect in conditionals, but otherwise, they have different characteristics that give each an advantage in different situations.

If you find yourself trying to decide between using a sealed type or an enum class, ask yourself what it is that you're trying to limit. If you need to limit *values*, then use an enum class. If you need to limit *types*, then use a sealed type.

Let's take the example of schnauzer dog breeds from Chapter 5. If you want to represent the three valid *breeds* of a schnauzer, then an enum class works well. The *type* is just `SchnauzerBreed`, and its values are *limited* to `MINIATURE`, `STANDARD`, and `GIANT`.

```
// SchnauzerBreed instances are limited to three:  
enum class SchnauzerBreed { MINIATURE, STANDARD, GIANT }
```

Listing 6.23 - An enum class is a good choice for representing a schnauzer breed.

On the other hand, if you want to represent actual *schnauzers* - that is, the dogs themselves rather than the breed - then consider a sealed type. This allows you to *limit the types* to just three subtypes...

```
// Subtypes of Schnauzer are limited to three:  
sealed class Schnauzer(val name: String, val sound: String)  
class MiniatureSchnauzer(name: String) : Schnauzer(name, "Yip! Yip!")  
class StandardSchnauzer(name: String) : Schnauzer(name, "Bark!")  
class GiantSchnauzer(name: String) : Schnauzer(name, "Ruuuuffff!")
```

Listing 16.24 - A sealed type is a good choice for representing individual schnauzer dogs.

...but it allows you to create an *unlimited* number of instances.

```
// No limit on how many Schnauzer instances you can create:  
val dogs = listOf(  
    MiniatureSchnauzer("Shadow"),  
    StandardSchnauzer("Agent"),  
    MiniatureSchnauzer("Scout"),  
    GiantSchnauzer("Rex"),  
    GiantSchnauzer("Brutus")  
    // ... as many as you want ...  
)
```

Listing 16.25 - Creating multiple instances of different schnauzer breeds.

Both enum classes and sealed types are important, each in its own way!

¹Generally, you shouldn't need to iterate over the subtypes of a sealed type. Technically, however, it's possible to do with Kotlin's reflection library.

Summary

Well, Cecil's Ice Shoppe is doing great now, handling orders, refunds, and even support tickets! In this chapter, we learned:

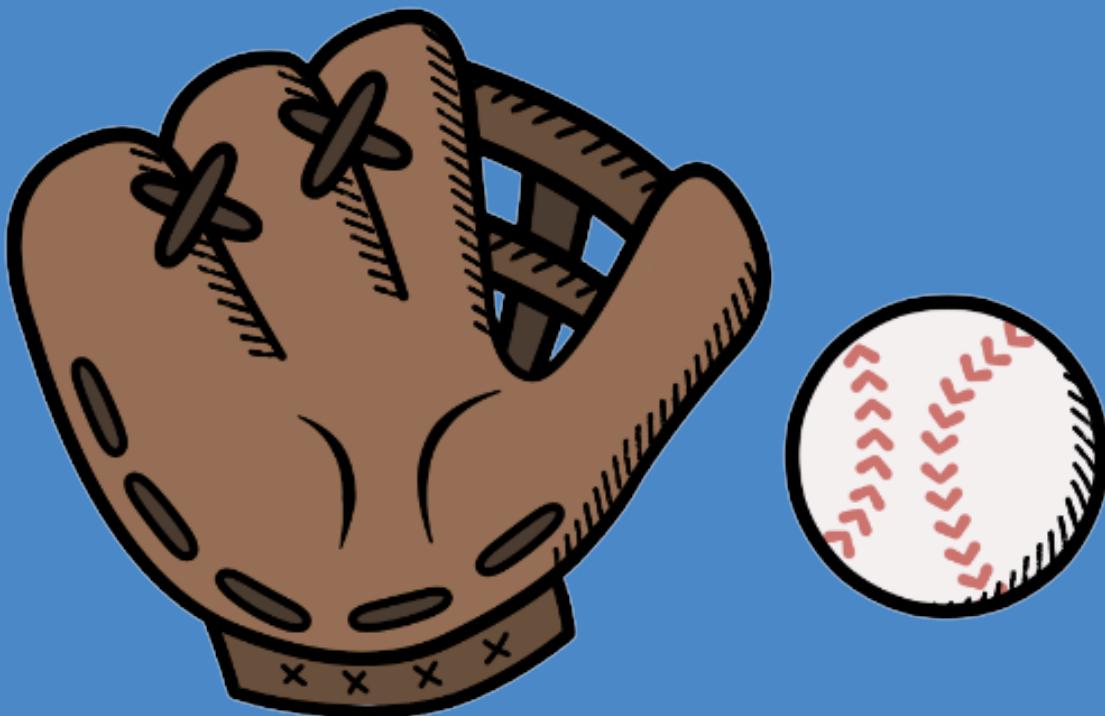
- How normal types will *not* be exhaustively matched in a conditional.
- How sealed types *will* be exhaustively matched in a conditional.
- Why Kotlin requires you to use the `sealed` modifier for this feature.
- How the `sealed` modifier can be applied to both `interface` and `class` declarations.
- The restrictions imposed on the subtypes of sealed types.
- The differences between sealed types and enum classes.

In this chapter, we saw a few examples of *compile-time* errors. In the next chapter, we'll look at different ways that we can handle errors that happen while our code is *running*!

Kotlin: An Illustrated Guide

Chapter 17

Handling Runtime Exceptions



That's a good plan... but what happens with things go wrong?

In real life, when we decide to do something, we primarily think in terms of a successful experience. For example, if you're driving to your friend's house for dinner, you might look up the directions on a mapping app, make sure you've got enough gasoline in the car, and leave at the right time in order to arrive when dinner is hot. If everything goes according to plan, you'll arrive on time.

Things don't always go according to plan, though. If you get a flat tire while you're on the way, you'll have to replace it with the spare, get to a garage, and buy a new tire. By the time you do all of that, dinner could be cold, and you might even have to cancel your plans.

Things can go wrong in our Kotlin code, too. In this chapter we'll learn how to handle problems that arise while our program is running!

Much like in real life, when our Kotlin code runs, things might not go according to plan. Our functions usually represent our plan - we tell Kotlin, "In general, follow this plan." If anything goes wrong, then we'll do something else, as an *exception* to that plan.

For this reason, when something unexpected happens in our code, we call it an **exception**. In this chapter, we'll learn all about how we can handle those exceptions!

Problems at Runtime

We've seen how we can get errors at two different times - either at [compile time or runtime](#). As Cecil discovered in the [last chapter](#), errors at compile time are quite helpful, because you can fix them before someone *using* your application can run into it.

However, not every problem can be detected at compile time. As an example, here's a function that can convert a number (e.g., 3) to its ordinal (e.g., "third").

```
val ordinals = listOf("zeroth", "first", "second", "third", "fourth", "fifth")
fun ordinal(number: Int) = ordinals.get(number)
```

Listing 17.1

Looking at this function, it's easy to see that it only supports ordinals for numbers up to 5. However, there's no way the Kotlin compiler can be sure that the **number** argument will be within those bounds. We can easily call this function with a number that's too high.

```
fun main() {
    val place = ordinal(9)
}
```

Listing 17.2

Running this `main()` function results in the following error message.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 9 out of
bounds for length 6
        at java.base/java.util.Arrays$ArrayList.get(Arrays.java:4165)
        at MainKt.ordinal(Main.kt:3)
        at MainKt.main(Main.kt:7)
        at MainKt.main(Main.kt)
```

The *plan* was to simply return the ordinal for the number, but things didn't go according to plan. Passing **9** to this function resulted in an *exception* to the plan. Since the *compiler* can't prevent us from calling `ordinal()` with an argument that's out of bounds, we'll have to handle this problem at *runtime* instead.

There are many similar problems that can't be detected at compile time. For example:

- At compile time, Kotlin can't know whether a [map](#) will include a particular key. (See Listing 9.11).

- At compile time, Kotlin can't know what values we might get when we ask a database for data.
- At compile time, Kotlin can't know what a user might type into the keyboard when prompted. We could ask the user for a zip code, but might get a phone number instead.

Because of this, we need to know how to handle things that can go wrong at runtime. To do that, we first need to understand the *call stack*.

The Call Stack

When your Kotlin program is running, code in one place usually calls code in another place, which in turn might call code in yet another place.

To demonstrate this, let's add a function that will *announce* the ordinal of a task that you're planning to do. Instead of naming this function `announce()`, we'll abbreviate it to `annc()`, which will fit better on some of the diagrams that we'll see in a moment.

```
val ordinals = listOf("zeroth", "first", "second", "third", "fourth", "fifth")
fun ordinal(number: Int) = ordinals.get(number)

fun annc(number: Int, task: String): String {
    val ordinal = ordinal(number)
    return "The $ordinal thing I will do is $task."
}

fun main() {
    val first = annc(1, "clean my room")
    // "The first thing I will do is clean my room."
}
```

Listing 17.3

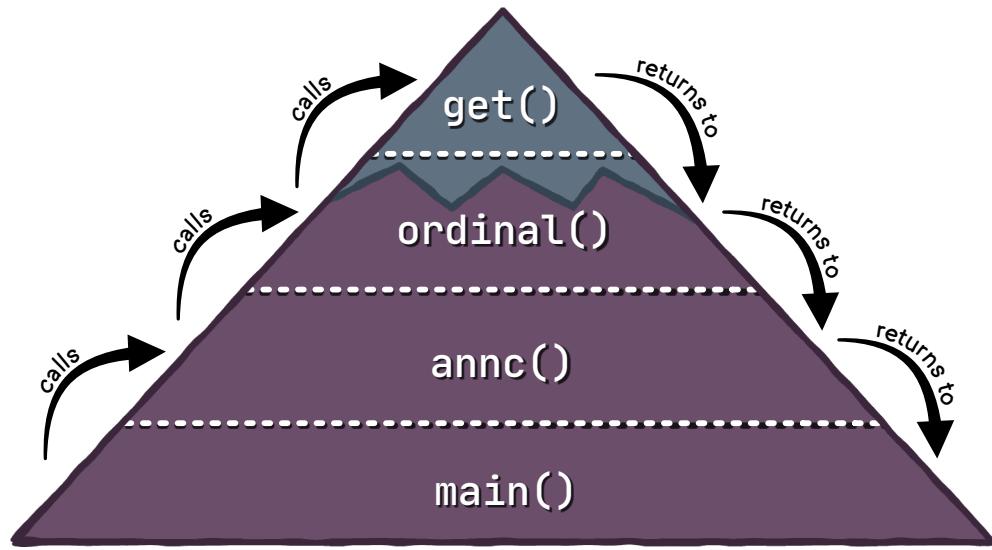
This code has a simple [execution path](#). Starting in the `main()` function...

1. The `main()` function calls the `annc()` function.
2. The `annc()` function then calls the `ordinal()` function.
3. The `ordinal()` function calls the `get()` function of the list.
4. The `get()` function returns its result to the `ordinal()` function.
5. The `ordinal()` function returns its result to the `annc()` function.
6. The `annc()` function returns its result to the `main()` function.

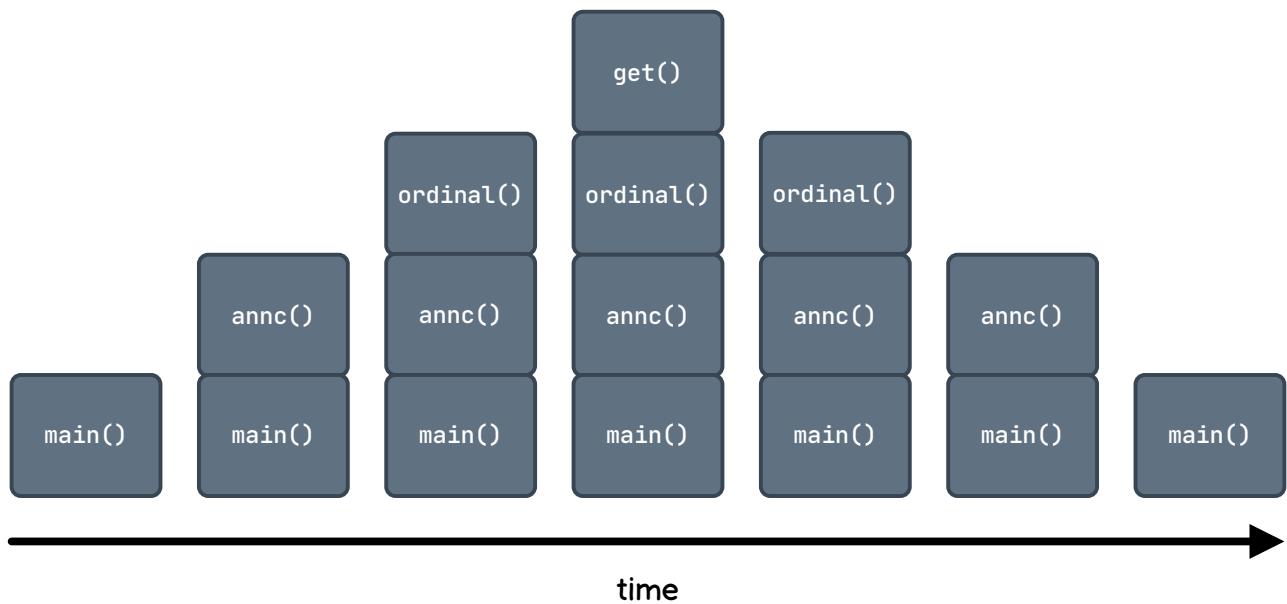
And of course, when the `main()` function ends, the program successfully finishes running.

It's kind of like our program is climbing a mountain. It starts inside the `main()` function at ground level, and for each function call along the way, it climbs higher on the mountain. Eventually, it's in the `get()` function, at

which point each function returns its value in turn, as the program works its way back down the mountain, back into the `main()` function.



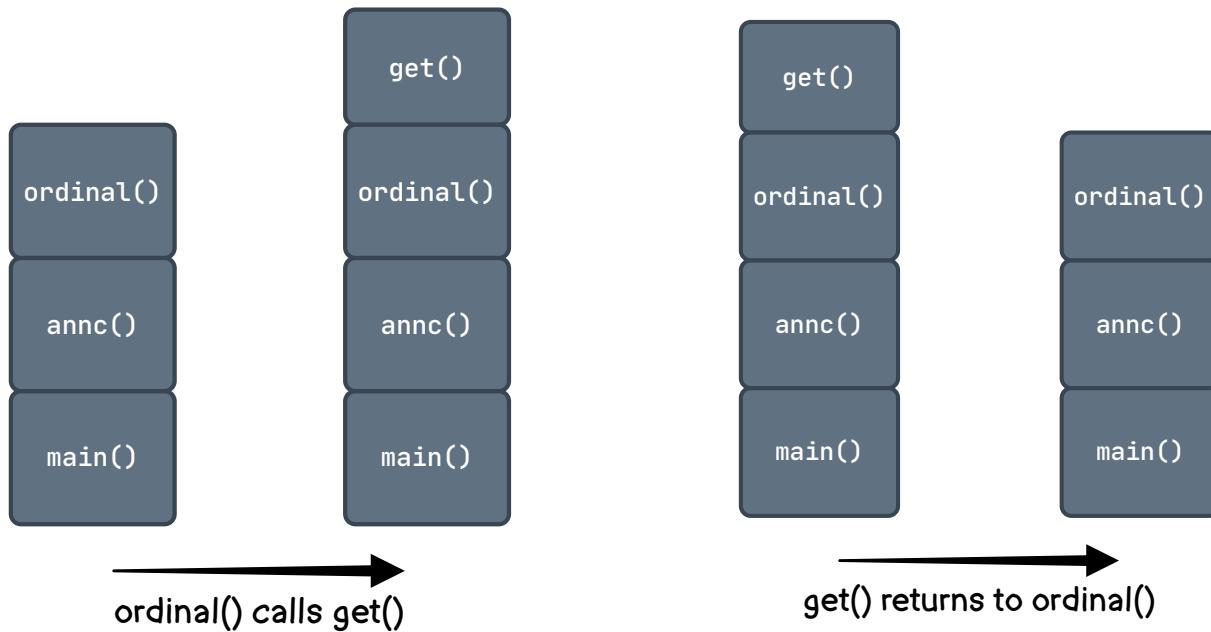
Rather than a drawing of a mountain, a simpler depiction might be a stack of boxes.



The diagram above still roughly has the shape of a mountain. Each little box represents a function. As the program's execution progresses (from left to right in this diagram), each function either calls another function, or it returns a value to the function that called it.

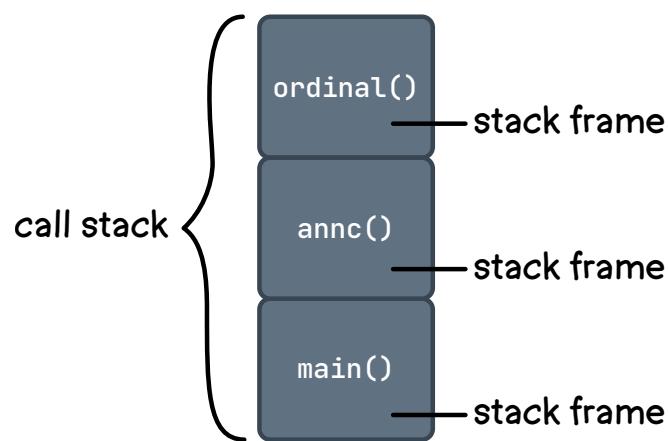
When one function calls another, then the pile of boxes at the next step will include that next function on top of the function that called it.

And when a function returns, the pile of boxes at the next step will remove that function from the top of the stack.



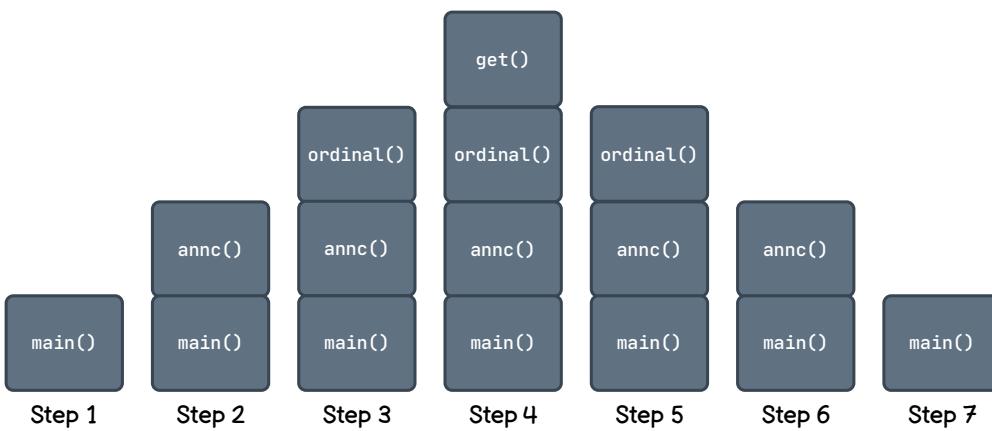
Note that those are the *only* two ways that the stack will change! A box can never be added or removed from anywhere except the *top* of the stack!

In programming, this stack of function calls is referred to as the **call stack**, or sometimes just **the stack**. Each box in the stack is a **stack frame**.



So, the diagram on the next page shows what the call stack looks like at each step as the program is running.

Our example from Listing 17.3 is pretty simple - we just have a few functions, each calling the next, so over time, the call stack looks like a single mountain.



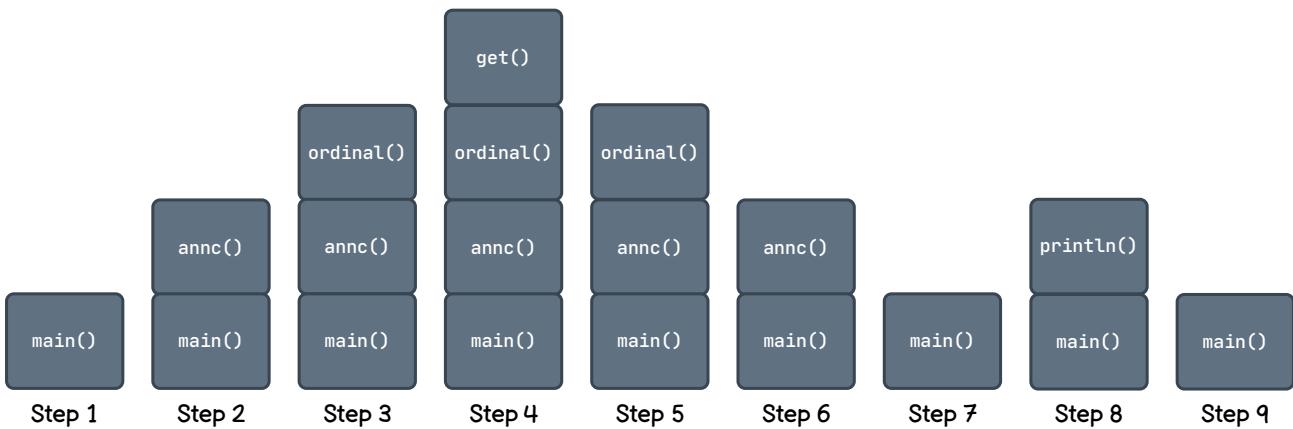
Usually, though, it'll look more like a mountain range. For example, let's add a call to `println()` in the `main()` function.

```
fun main() {
    val first = anncc(1, "clean my room")
    println(first)
}
```

Listing 17.4

The first thing I will do is clean my room.

With this small change, the call stack over time looks like this:



For what it's worth, the `println()` function itself technically calls *other* functions, so if we were to fully chart out all of those calls, the mountain range would look even more complex than it does above!

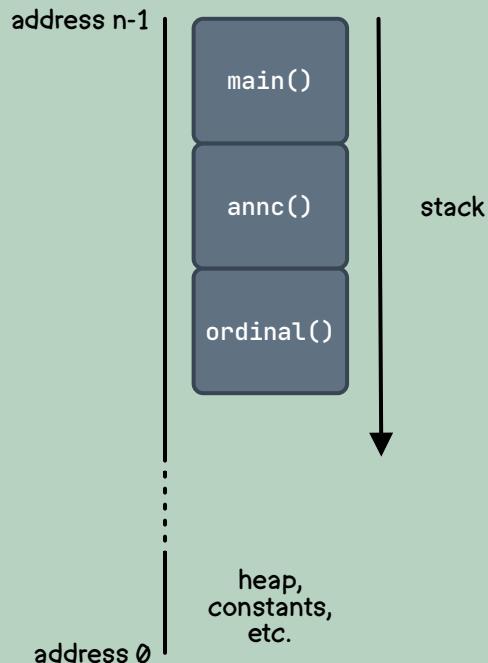
One last thing to note before we move on - in Listing 17.3, we've only got *functions* calling other *functions*, but a call stack will often include other things. For example, when you call a class' constructor, the properties of that class will also be initialized, and when that happens, it's also part of the call stack. Similarly, you might get a value from a *property* that has a custom [getter function](#). These would also have their own frames on the call stack.

By the way: Top and Bottom of the Stack

The *call stack* isn't just a mental construct - each stack frame takes up actual space in the memory on your computer. Each slot of memory has an address, starting at zero and going all the way up to however much memory is allocated to your program. Interestingly, in most computer systems, the call stack actually starts at the *highest* memory address - that is, the slot whose address is the biggest number. Then, when a frame is added to the stack, it gets an address just *below* the previous frame's address.

On the other hand, in regular life, when you and I think about a stack - whether it's a stack of pancakes, dishes, or chairs - we put the newest item on the top of the stack, so the stack grows upward. However, because of the way a call stack takes up space in memory, starting at the highest memory address, developers usually consider it to grow downward - i.e., toward memory address zero.

This can lead to quite a bit of confusion. So when a programmer says that something's at the *top* of the call stack, sometimes they really mean that it's at the *first* frame in the stack (i.e., the `main()` function in Kotlin). Other times, though, a developer who says "top of the stack" might mean the *last* frame, as found in some of Java's documentation.



So now that we know all about the call stack, what does this have to do with exceptions and error messages?

Call Stacks, Exceptions, and Error Messages

Let's update the `main()` function. Instead of passing it the number `1`, let's pass it the number `9`, which will be out of bounds for the `ordinal()` function.

```
fun main() {
    val task = annC(9, "clean my room")
    println(task)
}
```

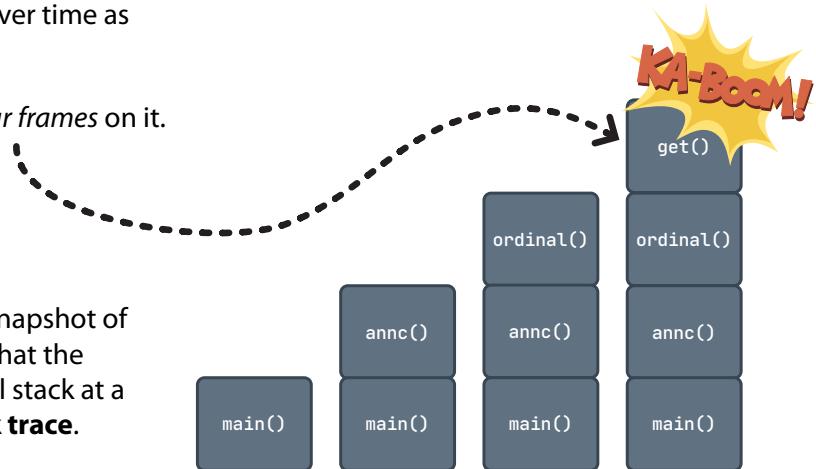
Listing 17.5

When this runs, instead of seeing "The ninth thing I will do is clean my room", we get an `ArrayIndexOutOfBoundsException`, just like we did back in Listing 17.2. Here's that error message again:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 9 out of
bounds for length 6
    at java.base/java.util.Arrays$ArrayList.get((Arrays.java:4165)
    at MainKt.ordinal(Main.kt:3)
    at MainKt.main(Main.kt:7)
    at MainKt.main(Main.kt)
```

Let's examine how the call stack changes over time as this code is running.

When things blow up, the call stack has *four frames* on it.



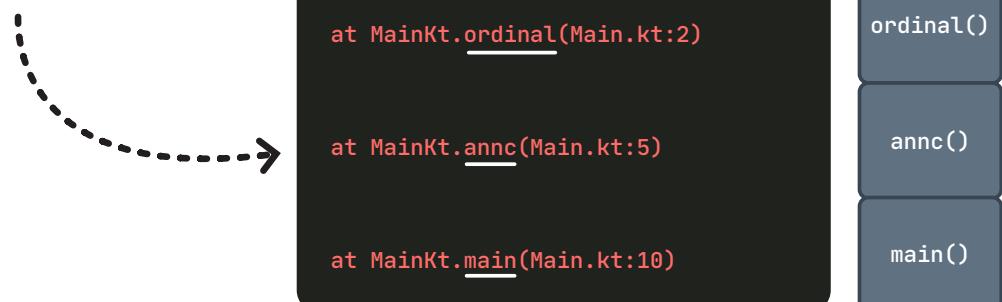
The error message that we get includes a snapshot of what the call stack looked like at the time that the exception happened. A snapshot of the call stack at a particular point in time is known as a **stack trace**.

This exception...
... happened in this function

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
Index 9 out of bounds for length 6
    at java.base/java.util.Arrays$ArrayList.get((Arrays.java:4165)
    at MainKt.ordinal(Main.kt:2)
    at MainKt.annc(Main.kt:5)
    at MainKt.main(Main.kt:10)
    at MainKt.main(Main.kt)
```

stack trace { }

Note that, not counting the bottom line of the stack trace, it shows the same thing as our stack of boxes above.



For each frame in the stack trace, we can see the *name of the function*, the *name of the file* where that function is, and the *line number* of code where the next function was called.

```
at java.base/java.util.Arrays$ArrayList.get(NSArray.java:4165)
at MainKt.ordinal(Main.kt:2)
at MainKt.annC(Main.kt:5)
at MainKt.main(Main.kt:10)
at MainKt.main(Main.kt)
```

It's easy to understand the stack trace, but let's take it one line at a time.

```
at java.base/java.util.Arrays$ArrayList.get(NSArray.java:4165)
```

This line tells us that the error happened in a function called `get()`, which is inside a class called `ArrayList`. This class belongs to Java's API, which was used by Kotlin's standard library, and the error happened at line 4165. (Yes, that's a lot of lines!)

```
at MainKt.ordinal(Main.kt:2)
```

Next, we can see that the `get()` function (from the previous line) was called from our `ordinal()` function, at line 2 inside a file that I called `Main.kt`.

```
at MainKt.annC(Main.kt:5)
```

After that, `ordinal()` (from the previous line) was called by `annC()` on line 5 of `Main.kt`.

```
at MainKt.main(Main.kt:10)
```

And `annC()` was called by the `main()` function on line 10 of the same file.

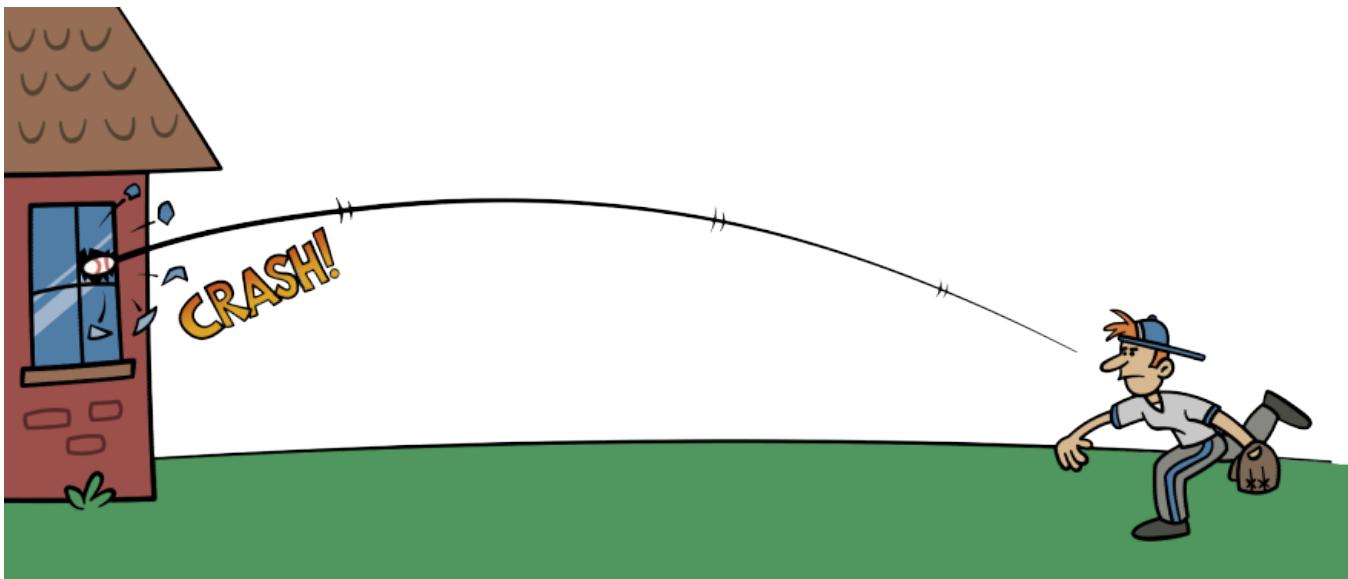
```
at MainKt.main(Main.kt)
```

And this final line simply shows us that everything started in the `Main.kt` file.

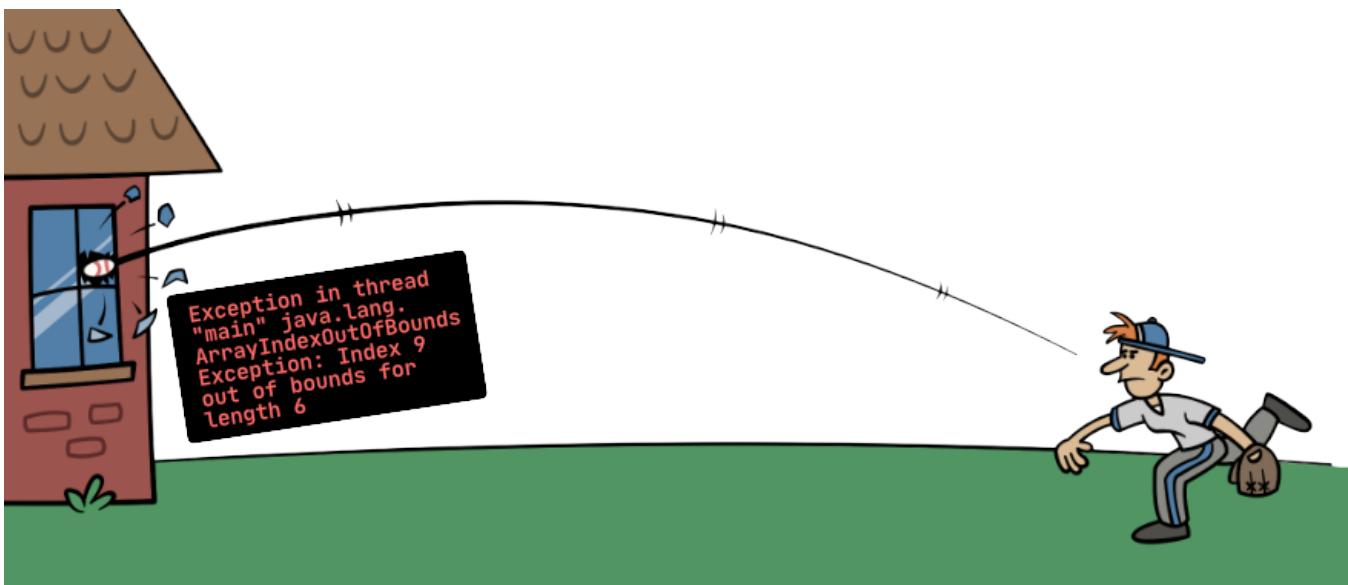
The stack trace can be helpful when you're trying to figure out what went wrong, but it also shows us where we could add code to try to prevent the program from abruptly terminating.

Catching Exceptions

When an exception arises, it's kind of like a kid throwing a baseball toward a window. If nobody is there to catch that baseball, it'll crash into the window.



Similarly, in our Kotlin code, when an exception is thrown, if nobody catches the exception, an error message will be printed out, and the program will crash.



Usually, we do *not* want our program to crash in response to an exception. To demonstrate why, let's replace our single task with a *list* of tasks, and print out an announcement for each one.

```
fun main() {
    val tasks = listOf(1 to "clean my room", 9 to "take out trash", 5 to "feed the dog")
    tasks.forEach { (number, task) ->
        println(annnc(number, task))
    }
}
```

Listing 17.6

When this runs, we'll see the following output:

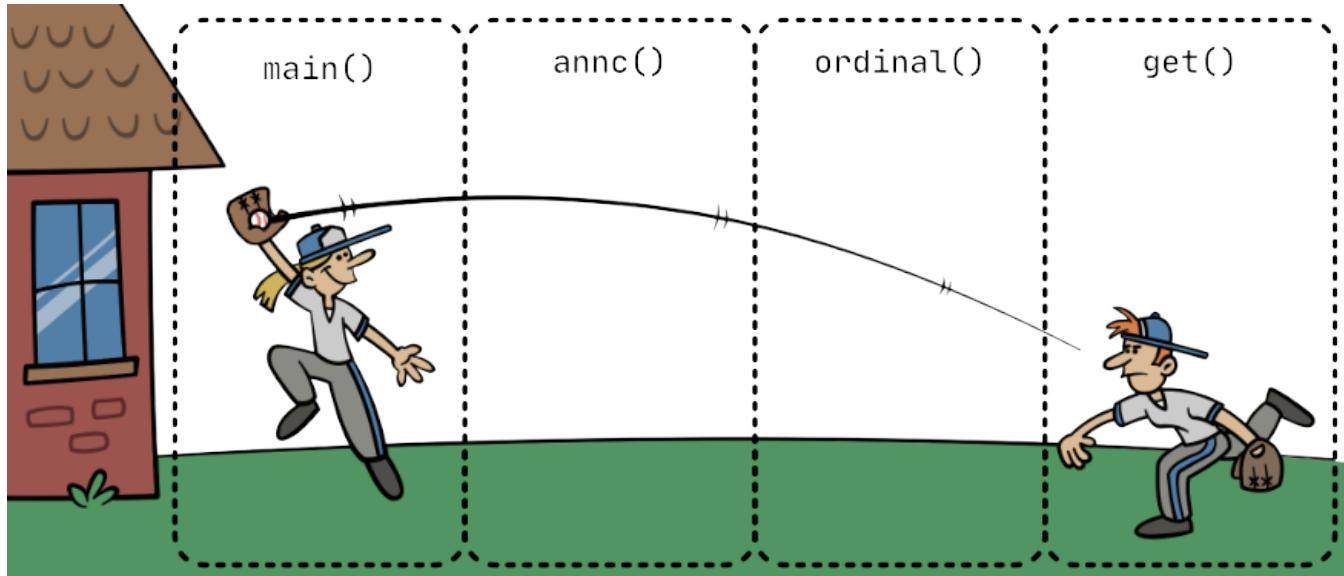
```
The first thing I will do is clean my room.
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 9 out of bounds
for length 6
at java.base/java.util.Arrays$ArrayList.get(Arrays.java:4165)
at MainKt.ordinal(Main.kt:2)
at MainKt.annc(Main.kt:5)
at MainKt.main(Main.kt:11)
at MainKt.main(Main.kt)
```

The first task was printed to the screen, but the second task caused an exception, so the program aborted. Because the program stopped running, it never printed out the task about feeding the dog! Instead, it'd be great if the program would keep running, even when there's an exception.

To keep the things running, we'll need to *catch* that exception. In other words, we need someone to catch that baseball before it crashes through the window!

So, how can we *catch* an exception? Well, we can put a baseball player at any frame of the call stack. Starting from the frame where it's thrown, the exception will pass by each stack frame, one by one, all the way through the `main()` function where the program started. At each step along the way, if there's a ball player in that frame, he or she can catch the exception, preventing the window from breaking.

For example, we can put a catcher in the `main()` function, and she'll catch the exception just before it hits the window.



Let's add some code to catch an exception! To do this, we'll use a **try expression**, which looks like this.

```
try {           "try" keyword
    // do something exception-prone here
} catch (exception: Exception) {           exception parameter
    // handle the exception here
}
```

This expression has a few key pieces to it:

1. It starts with the `try` keyword.
2. After that comes the `try` block. This is where we put the exception-prone code.
3. After that is the `catch` keyword.
4. Inside the parentheses is the **exception parameter**, which is declared with a name and the type of the exception that we want to catch. We'll learn more about this in a moment.
5. After that comes the `catch` block. This is where we put the code that should run when an exception is caught.

To start with, let's just put our catcher (that is, a try expression) in the `main()` function, where she can catch the exception just before it breaks the window. When we call `ann()`, an exception could end up getting thrown. Since that call is *exception-prone*, we'll put it *inside* the `try` block. Then, inside the `catch` block, we'll simply print out that something went wrong. Here's what we end up with:

```
fun main() {  
    val tasks = listOf(1 to "clean my room", 9 to "take out trash", 5 to "feed the dog")  
    tasks.forEach { (number, task) ->  
        try {  
            println(ann(number, task))  
        } catch (exception: Exception) {  
            println("Something went wrong!")  
        }  
    }  
}
```

Listing 17.7

When we run this again, we'll still get an exception while processing "take out trash", but this time, the exception will be *handled* by the `catch` block. As a result, we see this:

```
The first thing I will do is clean my room.  
Something went wrong!  
The fifth thing I will do is feed the dog.
```

With this change, we prevented the program from crashing with an error message! That means we *still get to see the task about feeding the dog!* After processing all three tasks, the program ends successfully.

Printing out "Something went wrong!" is better than nothing, but there's much more we can do with exceptions when we catch them.

Exceptions Are Objects

An exception is an object, and we can use its functions and properties, just like any other object.

In Listing 17.7 above, our catch block declared an exception parameter named `exception` and that parameter's type is `Exception`.

```
try {
    println(announcement(number, task))
} catch (exception: Exception) {
    println("Something went wrong!")
}
```

Although the variable name `exception` was used here, most developers prefer using very small variable names for exceptions, such as `ex` or just `e`.

```
try {
    println(annnc(number, task))
} catch (e: Exception) {
    println("Something went wrong!")
}
```

Listing 17.8

This variable is visible inside the `catch` block, so you can interact with it as needed. For example, every `Exception` object has a nullable property called `message`, so let's update our `println()` to include that message.

```
try {
    println(annnc(number, task))
} catch (e: Exception) {
    println("Something went wrong! ${e.message}")
}
```

Listing 17.9

Now when this runs, instead of just seeing that *something* went wrong, we can see *what it was* that went wrong - in this case, that index 9 was out of bounds for an array that has a length of 6.

```
The first thing I will do is clean my room.
Something went wrong! Index 9 out of bounds for length 6
The fifth thing I will do is feed the dog.
```

In addition to the `message` property, `Exception` objects include a stack trace, which you can print out with the `printStackTrace()` function.¹

We're not confined to using the `Exception` class itself, though. We can [extend](#) it, and include any functions or properties that we want! Before that will be helpful, though, we'll first need to learn how to throw our own exceptions.

¹This function is available when your Kotlin code is targeting the Java Virtual Machine or a native platform, but not currently available when targeting JavaScript.

By the way...

You can even use [collection operations](#) to loop over and process the stack trace, like this:

```
try {
    println(annC(number, task))
} catch (e: Exception) {
    e.stackTrace
        .reversed()
        .drop(1)
        .joinToString(" -> ") { "${it.methodName}()" }
        .let { println("Error: $it") }
}
```

```
The first thing I will do is clean my room.
Error: main() -> annC() -> ordinal() -> get()
The third thing I will do is feed the dog.
```

Throwing Exceptions

In addition to *catching* exceptions, we can also *throw* them ourselves. Since an exception is just an object, we can instantiate one like usual - by calling its constructor. We don't *have* to pass any arguments to its constructor, but it can be helpful to give it a message describing the problem.

```
val exception = Exception("No cleaning allowed on holidays!")
```

Listing 17.10

Once you have an instance of an exception, you can **throw** the exception with the `throw` keyword, so we could do this:

```
throw exception
```

Listing 17.11

However, the *stack trace* of an exception is determined at the point where the exception is *instantiated*, not the point at which it is *thrown*. For that reason, we don't usually instantiate and store an exception in a variable like we're doing in Listing 17.10. Instead, it's typical to instantiate an exception *when you throw it*, like this.

```
throw Exception("No cleaning allowed on holidays!")
```

Listing 17.12

Let's update our `annC()` function so that if the `task` includes the word "clean", then we'll throw this exception.

```
fun annC(number: Int, task: String): String {
    if ("clean" in task) throw Exception("No cleaning allowed on holidays!")
    val ordinal = ordinal(number)
    return "The $ordinal thing I will do is $task."
}
```

Listing 17.13

Running this with the `try` expression from Listing 17.9 results in this output:

```
Something went wrong! No cleaning allowed on holidays!
Something went wrong! Index 9 out of bounds for length 6
The fifth thing I will do is feed the dog.
```

Our `try` expression is now handling at least two different cases of exceptions:

1. An exception that is thrown when the task includes the word "clean".
2. An exception that is thrown when the index is out of bounds.

At the moment, we're handling both of those exceptions the same way, but we might prefer to handle each of those cases *differently*. To distinguish between these two different kinds of exceptions, we can make sure they have different types.

Exception Types

Although we can instantiate the `Exception` class directly, as we did in Listing 17.13, it's often helpful to extend it for each unique category of exception that we throw. For example, we can create a subclass of `Exception` that's dedicated to holidays, when we aren't allowed to clean.

```
class HolidayException(val task: String) : Exception("$task' is not allowed on holidays")
```

Listing 17.14

Now we can throw a `HolidayException` instead of a general `Exception`.

```
fun annC(number: Int, task: String): String {
    if ("clean" in task) throw HolidayException(task)
    // ...
}
```

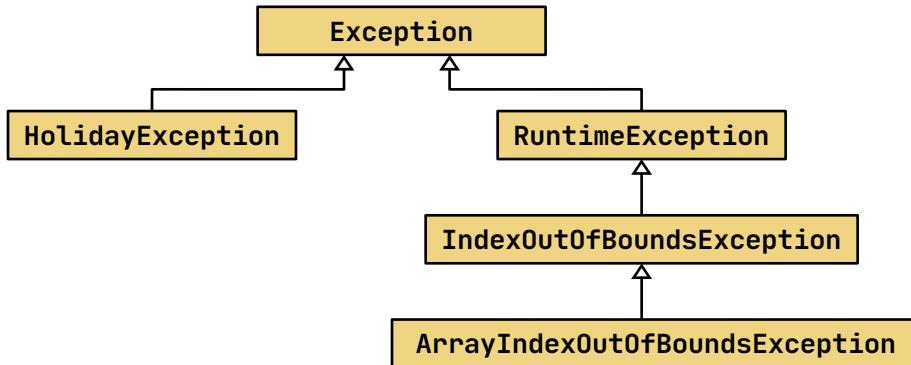
Listing 17.15

With this change, this program now deals with two kinds of exceptions:

- `ArrayIndexOutOfBoundsException`, which could be thrown from the `get()` function of the list.

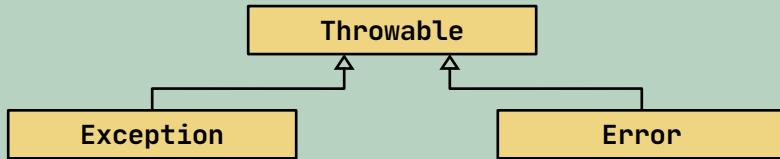
- `HolidayException`, which could be thrown from our `ann()` function.

Each of these is a subclass of the `Exception` class, although `ArrayIndexOutOfBoundsException` goes through a few intermediate classes. Here's a UML class diagram showing how these classes are related.



By the way: Throwables and Errors

`Exception` objects aren't the only things that can be thrown. In fact, `Exception` is a subclass of a class called `Throwable`. Any instance of `Throwable` or its subclasses can be used with the `throw` keyword. There's a second category of throwables that's a sibling to `Exception`, called `Error`.



Whereas `Exception` usually represents a deviation from the normal flow of code, an `Error` typically represents a more severe condition - one which we can't recover from, such as when the system runs out of memory. Although it's possible to catch these, there's usually not much you can do with them. So, you'll rarely need to catch anything other than exceptions.

Let's look at our try expression again.

```

try {
    println(ann(number, task))
} catch (e: Exception) {
    println("Something went wrong! ${e.message}")
}
  
```

Listing 17.16

The type of the exception parameter tells Kotlin which *kinds* of exceptions should be handled in this catch block. This is like telling the ball players to only catch balls of a certain color, and ignore the others.

In Listing 17.16, the type is `Exception`, so this catch block will handle any exceptions that have a type of `Exception`, *including any of its subtypes*. Because `ArrayIndexOutOfBoundsException` and `HolidayException` are both subtypes of `Exception`, this code will catch both of those exceptions.

However, we can change this type to something more specific. For example, let's change it to `HolidayException`.

```
try {
    println(annC(number, task))
} catch (e: HolidayException) {
    println("Something went wrong! ${e.message}")
}
```

Listing 17.17

After making this change, we can run the code, and see in our output that the try expression caught and handled the `HolidayException`, but it did *not* catch the `ArrayIndexOutOfBoundsException` caused by the second task.

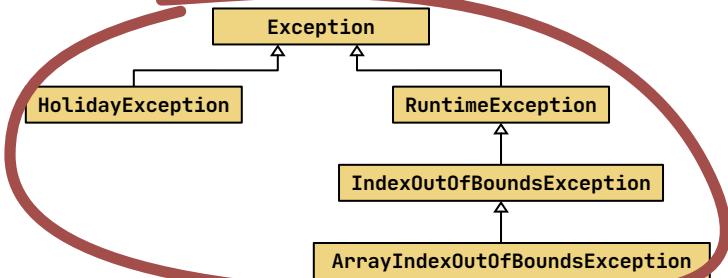
```
Something went wrong! 'clean my room' is not allowed on holidays
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
Index 9 out of bounds for length 6
... (stack trace follows) ...
```

So, the *type* of the exception parameter determines *which* exceptions are caught. Just remember that the catch block will handle the specified type, *including* any of its subtypes.

This code...

```
try {
    // ...
} catch (e: Exception) {
    // ...
}
```

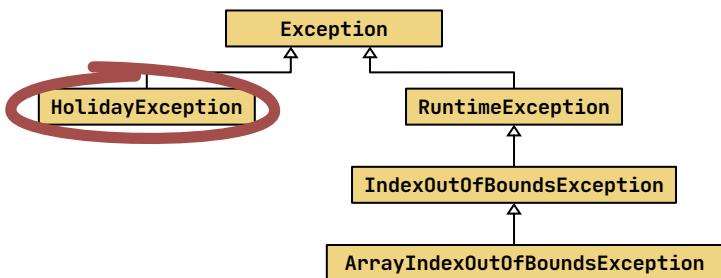
... will catch all of these exception types:



This code...

```
try {
    // ...
} catch (e: HolidayException) {
    // ...
}
```

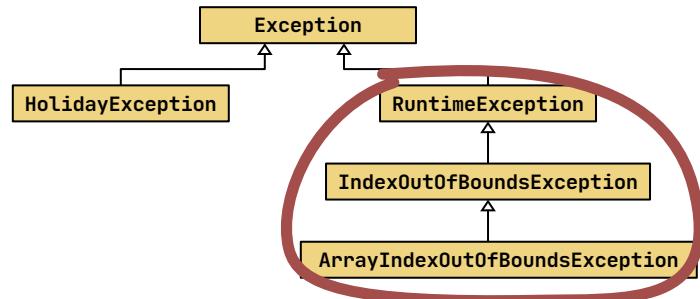
... will catch just this exception type:



This code...

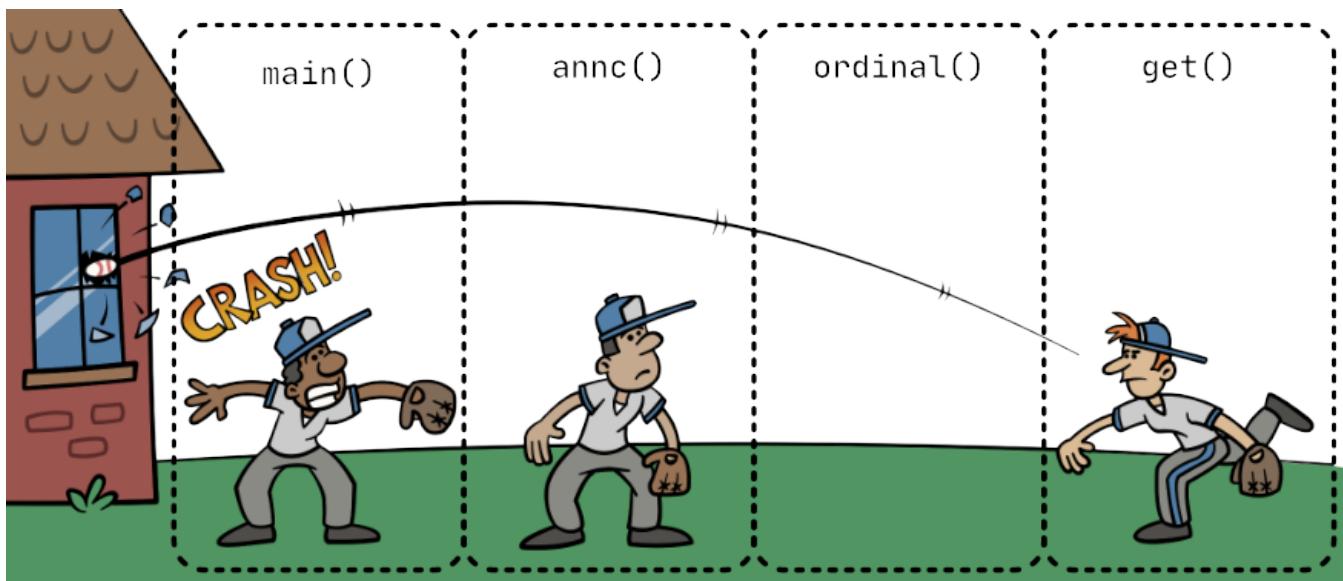
```
try {  
    // ...  
} catch (e: RuntimeException) {  
    // ...  
}
```

... will catch these exception types:



Any exception types that the catch block does *not* catch will simply continue to the next stack frame, where another try expression could have a chance to catch and handle it.

As always, if nobody catches the exception, it'll crash through the window.



Handling Multiple Exception Types Differently

Sometimes you might want to handle multiple types of exceptions in a single try expression, but do something *different* with each one of them. For example, instead of catching a `HolidayException` and ignoring an `ArrayIndexOutOfBoundsException`, you might want the `main()` function to handle them both, but print out something different in each case.

To do this, we can simply append additional catch blocks as needed, like this.

```
try {
    println(annC(number, task))
} catch (e: HolidayException) {
    println("It's a holiday! I'm not going to ${e.task} today!")
} catch (e: ArrayIndexOutOfBoundsException) {
    println("I can't count that high!")
}
```

Listing 17.18

```
It's a holiday! I'm not going to clean my room today!
I can't count that high!
The fifth thing I will do is feed the dog.
```

Note that when there's more than one catch block, the *first block with the matching exception type wins* - and any following blocks will be disregarded, even if the type of its exception parameter matches. Because of this, it's a good idea to think about the order of the catch blocks. For example, we *could* add a catch block for `Exception` at the very top:

```
try {
    println(annC(number, task))
} catch (e: Exception) {
    println("I wasn't expecting this!")
} catch (e: HolidayException) {
    println("It's a holiday! I'm not going to ${e.task} today!")
} catch (e: ArrayIndexOutOfBoundsException) {
    println("I can't count that high!")
}
```

Listing 17.19

However, if we do this, the two catch blocks that follow will be pointless. Since `HolidayException` and `ArrayIndexOutOfBoundsException` are *subclasses* of `Exception`, they'll always only match the first block!

```
I wasn't expecting this!
I wasn't expecting this!
The fifth thing I will do is feed the dog.
```

So instead, if a single try expression is going to include cases for both an exception class *and* one or more of its *subclasses*, put the subclasses above their corresponding superclasses, like this:

```
try {
    println(annC(number, task))
} catch (e: HolidayException) {
    println("It's a holiday! I'm not going to ${e.task} today!")
} catch (e: ArrayIndexOutOfBoundsException) {
    println("I can't count that high!")
} catch (e: Exception) {
    println("I wasn't expecting this!")
}
```

Listing 17.20

With this change, a `HolidayException` will be handled in the first catch block, an `ArrayIndexOutOfBoundsException` will be handled in the second, and any other kind of exception will be handled in the third block.

Evaluating a Try Expression

Throughout this chapter, we've referred to try-catch as an *expression*. As you might recall, the difference between a statement and an expression is that an expression can be [evaluated](#) - in other words, it can be reduced to a value. Since try expressions are indeed expressions, we can assign them to variables, pass them to functions, or return them from functions.

For example, currently, our try expression in the `main()` function is calling `println()` in each case.

```
try {
    println(annC(number, task))
} catch (e: HolidayException) {
    println("It's a holiday! I'm not going to ${e.task} today!")
} catch (e: ArrayIndexOutOfBoundsException) {
    println("I can't count that high!")
}
```

Listing 17.21

Instead of calling `println()` everywhere, we can simply let this try expression evaluate to a `String`. To do this, we just remove each `println()`, and assign the result to a variable. Then we can print that variable.

```
val words: String = try {
    annC(number, task)
} catch (e: HolidayException) {
    "It's a holiday! I'm not going to ${e.task} today!"
} catch (e: ArrayIndexOutOfBoundsException) {
    "I can't count that high!"
}

println(words)
```

Listing 17.22

This works exactly like you'd expect:

- If there are no exceptions, the result of `annC()` will be assigned to the `words` variable.
- If there's a `HolidayException`, then the string in that block will be assigned to `words`. ("It's a holiday!...")
- If there's an `ArrayIndexOutOfBoundsException`, then "I can't count that high!" will be assigned to `words`.
- If there's any other exception, it won't be caught here.

Try-Catch-Finally

One day, there was a friendly neighbor named Sarah. She takes pride in having a lush, green, well-manicured lawn. Twice a week, she would turn on the outdoor faucet to start watering her lawn. One day, after turning on the faucet, she noticed that the sprinkler wasn't spinning like it should.

Realizing that the sprinkler was broken, she went into the house and ordered a new one from an online store. Then, she continued with everything else that she needed to do that day. Unfortunately, by the end of the day, her lawn was flooded, because she never turned off the faucet!

Just like Sarah, sometimes you need to make sure you wrap up a task, even if something goes wrong! The following code models her experience:

```
try {
    faucet.turnOn()
    watch(sprinkler) // SprinklerBrokenException is thrown here
    faucet.turnOff() // this never runs!
} catch (e: SprinklerBrokenException) {
    store.orderNewSprinkler()
}
```

Listing 17.23

This code seems reasonable, but there's a problem with it - if the `watch()` function throws a `SprinklerBrokenException`, then `faucet.turnOff()` never runs!

To help with cases like this, you can include a block called `finally` at the end of a `try` expression. A `finally` block will run after the rest of the expression is processed, regardless of whether an exception was thrown. Here's how it looks:

```
try {
    faucet.turnOn()
    watch(sprinkler) // SprinklerBrokenException is thrown here
} catch (e: SprinklerBrokenException) {
    store.orderNewSprinkler()
} finally {
    faucet.turnOff() // This will run, even when the sprinkler breaks!
}
```

Listing 17.24

With this code:

- If no exception is thrown, then `faucet.turnOff()` will run after `watch(sprinkler)` completes.
- If a `SprinklerBrokenException` is thrown, then `faucet.turnOff()` will run after `store.orderNewSprinkler()` runs.
- If any other kind of exception is thrown, then `faucet.turnOff()` will run before the exception makes its way toward the start of the stack.

A `finally` block is typically most helpful with resources that need to be closed. For example, if you want your program to read a file on your computer, you'll want to make sure that you close it when you're done with it, even if an exception is thrown while processing it.

By the way: Closeable Resources

When your Kotlin code targets the JVM, you might use a resource that implements the `Closeable` interface, such as a `FileReader`. When doing this, rather than using `try-finally` directly, you can use an extension function called `use()`. It will automatically close the resource, regardless of whether an exception is thrown.

```
FileReader(file).use { reader ->
    // Use the FileReader here
}
```

Note that you can use `finally` with `try`, even when you don't have any `catch` blocks! For example, you might want the exception to be handled by another `try-expression`, closer to the start of the stack, but still need to make sure you turn the faucet off.

```
try {
    faucet.turnOn()
    watch(sprinkler)
} finally {
    faucet.turnOff()
}
```

Listing 17.25

At minimum, a try expression must have a catch block or a finally block.

Try expressions are a bit verbose. They tend to take up a lot of space on the screen, and they result in an execution path that can be difficult to follow, because it's not always obvious *where* an exception will be handled. Kotlin includes another approach to exception handling, which is also worth considering.

A Functional Approach to Exception Handling

Kotlin's standard library includes a function named `runCatching()`. Internally, it uses a try-expression, but by using this function, we can often avoid manually writing a try-expression in our own code. Using this function is easy. To start with, simply call it with the exception-prone code, and assign its result to a variable, like this:

```
fun main() {
    val tasks = listOf(1 to "clean my room", 9 to "take out trash", 5 to "feed the dog")
    tasks.forEach { (number, task) ->
        val result = runCatching { ann(number, task) }
    }
}
```

Listing 17.26

`runCatching()` will return a `Result` object, which will contain one of two things - either a successful result, or an exception. In the code above, this means...

1. If `ann()` completes successfully, then it will contain the result that `ann()` returned.
2. If `ann()` throws an exception, then it will contain that exception.

The `Result` object has a few functions on it that can be used to work with the result or exception. An easy way to work with it is to use its `getOrDefault()` function. For example, the following code does the same thing as Listing 17.8:

```
val result = runCatching { ann(number, task) }
val text = result.getOrDefault("Something went wrong!")
println(text)
```

Listing 17.27

In this code, if `annC()` completes successfully, then its result will be assigned to the `text` variable. Otherwise the string "Something went wrong!" will be assigned to it. This works just fine in many cases, but if we need the default to be based on a property of the `exception` - for example, if we want to include the `message` of the exception - then we'll need something else.

For cases like these, we can use `getOrDefault()`. This function takes a lambda where you can operate on the exception object.¹ For example, the code below works the same as Listing 17.9.

```
val result = runCatching { annC(number, task) }
val text = result.getOrDefault { "Something went wrong! ${it.message}" }
println(text)
```

Listing 17.28

The `Result` class includes a number of other functions that can be used to transform its value or exception. These functions can be chained together, much like a collection operation chain.

`runCatching()` and `Result` can make some code easier to read than their equivalent `try` expressions. However, if we write the `try`-expression ourselves, we get more control - we can catch specific kinds of exceptions, and can add a `finally` block if it's needed.

Also, developers who come from a Functional Programming background often want more advanced functionality than the `Result` class provides, such as the ability to catch only certain kinds of exceptions, or the ability to accumulate multiple exceptions. If that's you, you'll probably want to check out the [Arrow](#) Functional Programming libraries for Kotlin.

Summary

You've done an "exceptional" job learning all about exceptions in this chapter, including:

- Why we can end up with [problems at runtime](#).
- What a [call stack](#) is, and how to [read a stack trace](#).
- How to [catch an exception](#), and what you can do with it once you've got one.
- How to [throw](#) our own exceptions.
- How to catch [different types of exceptions](#).
- How [try-catch is an expression](#), so it can be assigned to variables.
- How to use a [finally](#) block to make sure something happens, regardless of success or failure.
- How to use [runCatching\(\)](#) for a more functional approach to exception-handling.

In the next chapter we'll finally take a closer look at generic types!

¹The type of this object isn't actually `Exception`; it's `Throwable`. See the *Throwables and Errors* aside above for more information about how they relate to one another.

Kotlin: An Illustrated Guide

Chapter 18

Generics



If parameters are so helpful for functions, imagine what they can do for types!

All the way back in Chapter 8, when we introduced collections, we saw our first generic type: `List<String>` ... then we saw more generics in Chapter 9 when we looked at the `Pair` and `Map` classes. In order to stay focused on learning about collection types, we glossed over the details about these classes.

We've put off learning about them for long enough, though! It's finally time for us to gain a full understanding of generics, so buckle up!

Mugs and Beverages

Jennifer's bakery café offers delightful, sweet pastries and hot beverages, which you can enjoy at a dainty table or in a cozy chair.



As Jennifer's business has been picking up lately, she reached out to her brother Eric, who has been learning Kotlin, to model her operations for her. Eric decided to start with the beverage menu, which includes light, medium, and dark roast coffees, which customers receive in a ceramic mug. He decided to use [enum class](#) to represent the coffee options, and a simple [class](#) to represent the mug. He also added a [drink\(\)](#) function, which prints a line whenever a customer drinks the coffee.

```
enum class Coffee { LIGHT_ROAST, MEDIUM_ROAST, DARK_ROAST }
class Mug(val beverage: Coffee)

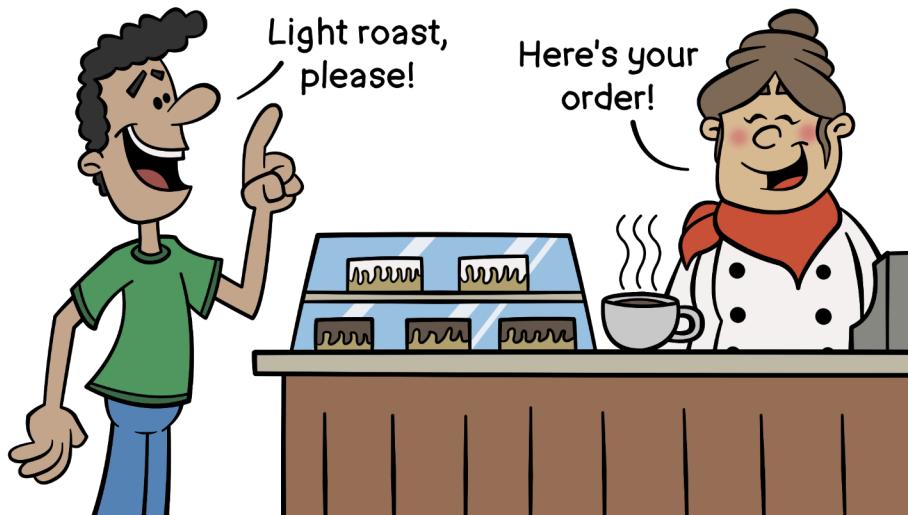
fun drink(coffee: Coffee) = println("Drinking coffee: $coffee")
```

Listing 18.1 - Modeling the coffee and mugs.

Now, whenever a customer orders a coffee, Eric can just instantiate a Mug with the right kind of coffee.

```
val mug =
    Mug(Coffee.LIGHT_ROAST)
```

Listing 18.2 - Instantiating a Mug with the right kind of coffee.



And when he takes a sip of his coffee, Eric can call the `drink()` function with the beverage in the mug.



```
drink(mug.beverage) // Drinking coffee: LIGHT_ROAST
```

Listing 18.3 - Calling the `drink()` function with the beverage that's in the mug.

One day, Jennifer decided that it was time to expand her beverage menu to include hot tea. As with the coffee, Eric chose to model these new tea options as an enum class. He also created an [overload](#) of the `drink()` function that accepts tea.

```
enum class Tea { GREEN_TEА, BLACK_TEА, RED_TEА }
fun drink(tea: Tea) = println("Drinking tea: $tea")
```

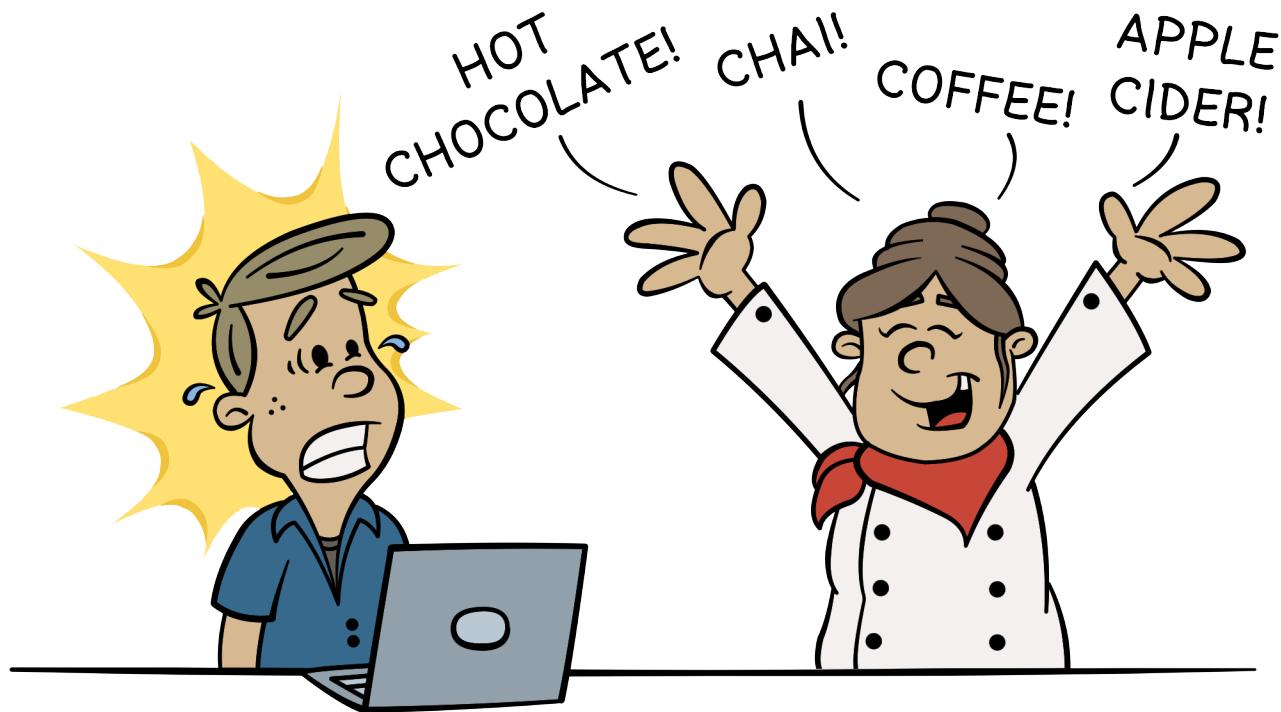
Listing 18.4 - A new enum class representing tea, plus an overload of the `drink()` function.

The `Mug` class also needed some work. After all, you can't pass an instance of `Tea` to a `Mug` that accepts only `Coffee`. Eric thought about it. "Well, I guess I can just create another mug class, just for tea." So he renamed `Mug` to `CoffeeMug`, and added another class, named `TeaMug`.

```
class CoffeeMug(val coffee: Coffee)
class TeaMug(val tea: Tea)
```

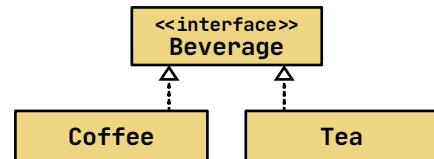
Listing 18.5 - Renaming `Mug` to `CoffeeMug`, and introducing a new `TeaMug` class.

As he finished typing, Jennifer walked up to him and said, "Have I mentioned that I'm going to be expanding my beverage menu next week? I'll be adding *hot chocolate, apple cider, and more!*"



Eric grimaced. "The more *beverages* my sister adds to the menu, the more *mug classes* I'm going to have to create. Things are going to get out of hand quickly. How can I make a single `Mug` class that can hold *any* kind of beverage?"

So he thought about how a [subtype can be used](#) anywhere that the compiler expects a supertype. "That's it!" he exclaimed, "I'll create an interface for the general concept of a beverage, and update the tea and coffee classes so that they're subtypes!" Having recently learned about [sealed types](#), he decided to make the `Beverage` interface sealed. He also updated the `Mug` class so that its property's type is `Beverage`. Here's what he ended up with.



```
sealed interface Beverage
enum class Tea : Beverage { GREEN_TEAS, BLACK_TEAS, RED_TEAS }
enum class Coffee : Beverage { LIGHT_ROASTS, MEDIUM_ROASTS, DARK_ROASTS }

class Mug(val beverage: Beverage)
```

Listing 18.6 - Creating a sealed interface for a beverage type.

After making this change, he was able to create instances of `Mug` with either `Coffee` or `Tea`.

```
val mugOfCoffee = Mug(Coffee.LIGHT_ROAST)
val mugOfTea = Mug(Tea.BLACK_TEAS)
```

Listing 18.7 - Instantiating a mug of coffee and a mug of tea.

"Huzzah! I've got a single mug class that can hold *any* kind of beverage!" When he ran his Kotlin code, though, he was alarmed to see some compiler errors - the call to the `drink()` functions no longer worked!

```
fun drink(coffee: Coffee) = println("Drinking coffee: $coffee")
fun drink(tea: Tea) = println("Drinking tea: $tea")

drink(mugOfCoffee.beverage)
drink(mugOfTea.beverage)
```

Listing 18.8 - Error: None of the functions can be called with the arguments supplied...

Why isn't this working? Let's take a closer look.

Declared Types, Actual Types, and Assignment Compatibility

As we learned back in [Chapter 12](#), objects have more than one type at a time. For example, a `Coffee` object has a type of `Coffee`, but it also has a type of `Beverage` and `Any`. This means it can be assigned to a variable that is declared with any of those types.

```
val coffee: Coffee = Coffee.MEDIUM_ROAST
val beverage: Beverage = Coffee.MEDIUM_ROAST
val anything: Any = Coffee.MEDIUM_ROAST
```

Listing 18.9 - Assigning `Coffee.MEDIUM_ROAST` to variables that have different types.

As a result, there can be a difference between the type of the *variable* and the type of the *object inside that variable*. This brings up a few distinguishing terms.

- The type that a variable has been declared with is known as its **declared type**.
- The most specific type of the object *inside* a variable is known as its **actual type** or its **runtime type**.

```
val coffee: Coffee = Coffee.MEDIUM_ROAST
val beverage: Beverage = Coffee.MEDIUM_ROAST
val anything: Any = Coffee.MEDIUM_ROAST
```

↑
Declared Types ↑
Actual Type is
Coffee in each case

When we assign an object to a variable, property, or parameter, the *actual* or *runtime type* is irrelevant. Only the *declared type* matters. For example, in the following code listing, the second line fails.

```
val beverage: Beverage = Coffee.MEDIUM_ROAST
val coffee: Coffee = beverage
```

Listing 18.10 - Error: Type mismatch. Required: Coffee. Found: Beverage.

Even though, as you and I read this code, we know that the *actual type* of the object inside the `beverage` variable at runtime will definitely be `Coffee`, its *declared type* is `Beverage`. And since a variable whose type is `Beverage` could *possibly* hold objects other than `Coffee` - it could hold a `Tea` object, for example - Kotlin won't let us perform this assignment.

In order for an assignment to succeed, the type of an expression on the right-hand side of the equal sign must be the same as the declared type on the left-hand side, or one of its subtypes.

Assignment compatibility is the term we use to describe whether the object on the right-hand side can be assigned to the variable's type on the left-hand side. When it *can* be assigned, we say that the object is **assignment-compatible** with that type.

Even though we've been talking specifically about literal assignments involving an equal sign, it's important to note that when we call a function with an argument, we're *effectively assigning an object to the function's parameter*, so all of the same rules apply.

The type of this expression...

val beverage: Beverage = Coffee.MEDIUM_ROAST

...must be Beverage or one of its subtypes.

fun drink(coffee: Coffee) =
println("Drinking coffee: \$coffee")

Let's look at the relevant parts of Eric's code again.

```
fun drink(coffee: Coffee) = println("Drinking coffee: $coffee")
fun drink(tea: Tea) = println("Drinking tea: $tea")

class Mug(val beverage: Beverage)

val mugOfCoffee = Mug(Coffee.LIGHT_ROAST)
drink(mugOfCoffee.beverage)
```

Listing 18.11 - Error: None of the following functions can be called with the arguments supplied.

Regardless of the **beverage** property's *actual type* at runtime (e.g., **Coffee** in this example), it can be assigned neither to a parameter of type **Coffee**, nor to a parameter of type **Tea**, because its *declared type* is **Beverage**. In other words, **beverage** is not assignment-compatible with either of the **drink()** overloads. This is why Eric's code is failing.

In order for this to compile without errors, he would need to [cast](#) the **beverage** property back to the **Coffee** type, in order to make it assignment-compatible with one of the **drink()** overloads.

```
val mugOfCoffee = Mug(Coffee.LIGHT_ROAST)
drink(mugOfCoffee.beverage as Coffee)
```

Listing 18.12 - Casting the Beverage to Coffee before calling drink().

Although this works, this means that *every time* Eric gets the **beverage** property off of a **Mug**, if he needs to assign it to the more specific **Coffee** or **Tea** type, he would need to cast it. Every time!

It'd be great if he could have a single **Mug** class that could work with any kind of **Beverage**, and avoid the need to cast the **beverage** property all the time. Thankfully, this can be solved with **generic** types!

Introduction to Generic Types

Declaring a Generic Type

For a long time now, we've utilized [functions](#) to reuse expressions. By calling them with different [arguments](#), we get different results. For example, we created this function to figure out the circumference of a circle back in Listing 2.3.

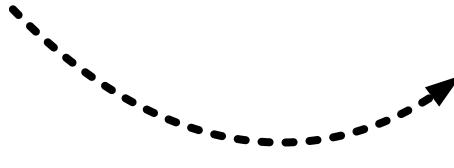
```
fun circumference(radius: Double) = 2 * pi * radius
```

Listing 18.13 - From Listing 2.3 in Chapter 2.

A function with a parameter like this is kind of like a fill-in-the-blank expression.

2 * pi * _____

Whenever you call it, you give it an argument to put into that blank.



circumference(2.0)	circumference(3.0)
2 * pi * <u>2.0</u>	2 * pi * <u>3.0</u>

So a function's parameter is basically a blank that you can fill in with a *value*.

What if there were something similar for *types*? For example, what if the type of the `beverage` property were a blank that we could fill in with different types?

class Mug(val beverage: _____)

Imagine all the different types we could put there!

class Mug(val beverage: <u>Coffee</u>)	class Mug(val beverage: <u>Tea</u>)
class Mug(val beverage: <u>AppleCider</u>)	
class Mug(val beverage: <u>HotChocolate</u>)	

Well, just like *functions* can have *parameters*, *types* can have **type parameters**.

To add a *type parameter* to a class, we can put the name of the type parameter inside angle brackets < and >, to the right of the name of the class, like this:

Declaring a type parameter

```
class Mug<BEVERAGE_TYPE>(val beverage: BEVERAGE_TYPE)
```

Using a type parameter



Here, we named the type parameter `BEVERAGE_TYPE`, but it's more typical to give them names that are just one letter long, such as `T`.

```
class Mug<T>(val beverage: T)
```

Listing 18.14 - A type parameter whose name is only one letter long.

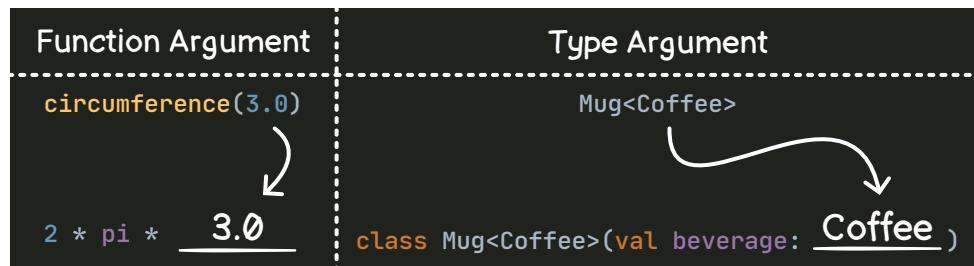
With this, the `Mug` class can now be used with different types! Let's see how to do that next.

Using a Generic Type

When we call a function that has a parameter, we have to supply an *argument* for that parameter. Likewise, when creating an instance of a `Mug`, we'll have to supply a **type argument** for the type parameter named `T`. To do this, we can put the type argument in angle brackets next to the name of the class, like this:

Type argument
`val mug = Mug<Coffee>(Coffee.LIGHT_ROAST)`

Much like calling a function with an argument is effectively the same as replacing the parameter with that value, creating a type with a type argument is effectively like filling in the type parameter with that type.¹



Even though a type argument must be supplied when constructing a generic class, we usually don't have to write it out ourselves, because Kotlin can use its [type inference](#) to figure it out. For example, since the constructor of `Mug` takes an argument whose type is `T`, Kotlin knows that if you create an instance with `Mug` and pass it a `Coffee` object, then the type argument for this `Mug` instance should be `Coffee`.

Because this constructor parameter uses the type parameter `T`...
... and because we're using a constructor argument whose type is `Coffee`...
... then the inferred type argument here is also `Coffee`.

Note that the type of this `mug` variable is *not* `Mug<T>`. It's `Mug<Coffee>`, and we can [explicitly specify](#) its type like this:

```
val mug: Mug<Coffee> = Mug(Coffee.LIGHT_ROAST)
```

Listing 18.15 - Explicitly specifying the type of the mug variable.

¹ Note that the right side of the illustration shows what the "effective" class would be. In other words, by creating a `Mug<Coffee>` type, it's as if we had declared another class called `Mug<Coffee>` whose `beverage` type is `Coffee`. However, we don't actually *write* that code - by creating the generic class in Listing 18.14, Kotlin has all it needs to create the type for us!

It's important to be able to distinguish between `Mug<T>` and `Mug<Coffee>`. A class that has a *type parameter*, such as `Mug<T>`, is known as a **generic type**. A generic whose type parameter has been filled with a type argument is known as a **parameterized type**.

The great thing about making the `Mug` class generic is that the `beverage` property will *retain its specific type*.

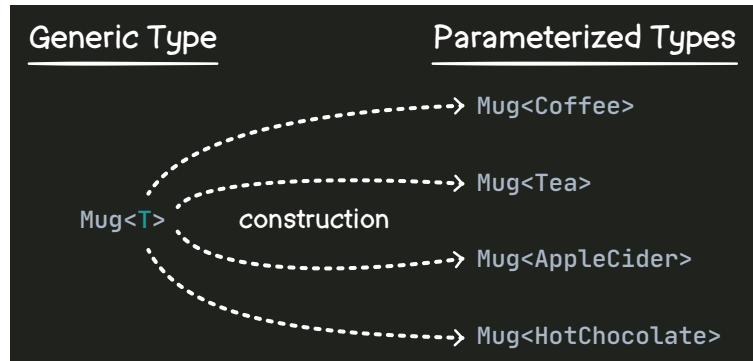
- When getting the `beverage` property from a `Mug<Coffee>`, its type will be `Coffee`.
- When getting the `beverage` property from a `Mug<Tea>`, its type will be `Tea`.

Because of this, there's no need to cast it when using it with code that expects a specific type!

For example, in the following code, we don't need to cast the `beverage` property to `Tea`, because that's what its type is already. So, the call to `drink(mug.beverage)` works just fine.

```
val mug: Mug<Tea> = Mug(Tea.GREEN_TEAL)
drink(mug.beverage)
```

Listing 18.16 - The type of `mug.beverage` here is `Tea`, so there is no need to cast it.



After Eric made all of these changes, the customers were able to enjoy their tea and coffee. He was able to use a single `Mug` class, and never needed to cast the `beverage` property.

He was about to discover some surprises with his code, though! Let's find out what happened next.

Type Parameter Constraints

One day, Jennifer decided to replace all of the café's ceramic mugs with fancy new temperature-controlled mugs that will keep the customers' tea and coffee at just the right drinking temperature.

She looked at Eric and said, "The mug will need to set its temperature based on the beverage inside it. If there's tea in the mug, then it should set the temperature to 140 degrees. If there's coffee in there, then it should set it to only 135 degrees."

Eric rolled up his sleeves and got to work. He updated all of the `Beverage` types, so that coffee and tea can each have its own ideal temperature.

```
sealed interface Beverage {
    val idealTemperature: Int
}

enum class Tea : Beverage {
    GREEN_TEAL, BLACK_TEAL, RED_TEAL;
    override val idealTemperature = 140
}

enum class Coffee : Beverage {
    LIGHT_ROAST, MEDIUM_ROAST, DARK_ROAST;
    override val idealTemperature = 135
}
```

Listing 18.17 - Adding an `idealTemperature` property to each beverage type.

Next, in order to represent the temperature-controlled mug, he added a new `temperature` property to his `Mug` class, and assigned it to the beverage's ideal temperature. To his surprise, he ended up with a compile-time error!

```
class Mug<T>(val beverage: T) {
    val temperature = beverage.idealTemperature
}
```

Listing 18.18 - Error: Unresolved reference: idealTemperature.

It seems that the `Mug` class is unable to see the new `idealTemperature` property that he added. As Eric wondered about this, a customer walked up to Jennifer and asked why his mug had a string in it!

"A string? It's supposed to be a beverage!" Jennifer cried. She shot a glance to her brother, who looked at the customer. It was true, there was a string in his mug!

Eric looked back down at his computer screen and hammered out some more code. Sure enough, it was possible to put a `String` in a `Mug`. In fact, as the `Mug` code is currently written, literally *anything* can be stuffed inside it!



```
val mugOfString: Mug<String> = Mug("How did this get in the mug?")
val mugOfInt: Mug<Int> = Mug(5)
val mugOfBoolean: Mug<Boolean> = Mug(true)
val mugOfEmptiness: Mug<Any?> = Mug(null)
```

Listing 18.19 - Stuffing a String, an Int, a Boolean, and a null into a mug.

So Eric now had two problems:

1. The `Mug` class can't see the `idealTemperature` property of its `Beverage`. (Listing 18.18)
2. The `Mug` class can hold an object of any type, but it should only hold a `Beverage`. (Listing 18.19)

Thankfully, the solution to both of these problems is the same - Eric needs to constrain the type parameter, so that only Beverage types can go inside it.

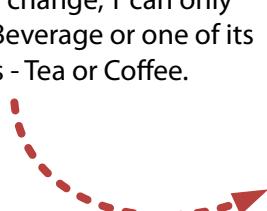
To do this, he can add a **type parameter constraint**, which will ensure that the type argument is of a particular type. To do this, after the name of the type parameter, add a colon and

Type parameter constraint

```
class Mug<T : Beverage>(val beverage: T) {
    val temperature = beverage.idealTemperature
}
```

the name of the type that will serve as the **upper bound** for that parameter. For example, to add an upper bound constraint of **Beverage**, we can write this.

With this change, T can only ever be Beverage or one of its subtypes - Tea or Coffee.



Using any other type as a type argument will cause a compiler error.

```
val mugOfString: Mug<String> = Mug("This won't work any more!")
```

Listing 18.20 - Error: Type argument is not within its bounds.

Also, now that Kotlin knows that **beverage** will be some kind of **Beverage**, it's possible to access any properties or functions that the **Beverage** type includes! Because the **Beverage** type includes the **idealTemperature** property, the following code now works.

```
class Mug<T : Beverage>(val beverage: T) {
    val temperature = beverage.idealTemperature
}
```

Listing 18.21 - The code from Listing 18.18 now compiles without errors.

If we don't specify the upper bound, Kotlin will assume a default of **Any?**, which means it'll accept any type, as we saw in Listing 18.19. If we know for sure that only certain types should be used for a type argument, it's usually a good idea to give it a type parameter constraint so that the compiler will enforce those rules.

Well, we've learned a lot about generics from Jennifer's bakery café. There are more things we can do with generics, though! Let's take a look at more ways they can be used.

Generics in Practice

Using Type Parameters

So far, we've only used a type parameter for a [property parameter](#), but we can use a type parameter almost anywhere that we might normally write a type within the class body.

Frequently, they're used for function parameters and return types, as shown here.

```
class Dish<T>(private var food: T) {
    fun replaceFood(newFood: T) {
        println("Replacing $food with $newFood")
        food = newFood
    }

    fun getFood(): T = food
}
```

Listing 18.22 - Using a type parameter in multiple places throughout a class.

Generics with Multiple Type Parameters

Our examples so far have only included one type parameter, but it's possible for a class to have more than one. For example, a combination order could include one type parameters for food and one type parameter for a beverage. When declaring them, just use a different name for each type parameter, and separate them with commas, like this.

```
class ComboOrder<T : Food, U : Beverage>(val food: T, val beverage: U)

val combo: ComboOrder<Pastry, Tea> = ComboOrder(Pastry.MUFFIN, Tea.GREEN_TEALISTING 18.23 - A class with two type parameters.
```

Don't go crazy with it, though! It's rare to use more than two or three type parameters in a generic type.¹

Generic Interfaces and Superclasses

In addition to classes, interfaces can also be generic. For example, here's a generic interface with one property.

```
interface Dish<T> { val food: T }
```

Listing 18.24 - A generic interface.

When *implementing* this class, we can substitute actual types for the type parameter. For example, we can create a `BowlOfSoup` class and instance like this.

```
class BowlOfSoup(override val food: Soup) : Dish<Soup>

val bowlOfSoup: BowlOfSoup = BowlOfSoup(Soup.TOMATO)
```

Listing 18.25 - Implementing the generic interface from Listing 18.24.

Alternatively, the implementing class can itself declare a type parameter, and relay that to the interface, as shown here.

```
class Bowl<F>(override val food: F) : Dish<F>

val bowlOfSoup: Bowl<Soup> = Bowl(Soup.TOMATO)
```

Listing 18.26 - Relaying a type parameter to the supertype.

Similarly, abstract and open classes can also be generic, and extending them works like you would expect.

¹ Once in a while you might find a library with lots of type parameters in a single generic type. For example, the [Arrow](#) functional programming library includes a class called `Tuple10` that includes ten type parameters. Most of the time, you shouldn't need that many in your own code, though.

```
abstract class Dish<T>(val food: T)
class BowlOfSoup(food: Soup) : Dish<Soup>(food)
class Bowl<F>(food: F) : Dish<F>(food)
```

Listing 18.27 - Extending an abstract generic class.

Just be sure to provide a type argument when calling the superclass' constructor. And again, that type argument can be a type parameter on the subclass, as is the case on the third line in this code listing.

Generic Functions

We've seen how both classes and interfaces can be generic, and how their functions can utilize those type parameters. However, functions can also declare their *own* type parameters. This is especially common for top-level functions - those which are declared outside of a class.

In this case, the type parameter is declared between the `fun` keyword and the name of the function. For example, we can make a top-level function that wraps the constructor of `Mug`.

```
fun <T : Beverage> serve(beverage: T): Mug<T> = Mug(beverage)
```

Listing 18.28 - Declaring a generic function.

When calling this function, we can specify the type argument explicitly by putting it in angle brackets to the right of the function name.

```
val mug = serve<Coffee>(Coffee.DARK_ROAST)
```

Listing 18.29 - Calling a generic function.

But again, Kotlin can usually infer the type, in which case you can omit it.

```
val mug = serve(Coffee.DARK_ROAST)
```

Listing 18.30 - Calling a generic function without explicitly specifying the type argument.

We can even create generic [extension functions](#) where the `receiver` type is a type parameter. For example, we can change the `serve()` function into an extension function like this:

```
fun <T : Beverage> T.pourIntoMug() = Mug(this)
val mug = Coffee.DARK_ROAST.pourIntoMug()
```

Listing 18.31 - Declaring and calling a generic extension function, which uses the type parameter as the receiver type.

We've now got a good understanding of the range of possibilities with generics. Kotlin's standard library also includes lots of generic types and functions, and we've already seen some of them throughout this book. Now that we understand more about what generics are and how they work, let's review a few of them!

Generics in the Standard Library

List and Set

Back in [Chapter 8](#), we created **List** and **Set** collections using functions like `listOf()` and `mutableSetOf()`, which are generic functions. As a refresher, here's how we can use the `listOf()` function to create a list of menu items.

```
val menu = listOf("Bagel", "Croissant", "Muffin", "Crumpet")
```

Listing 18.32 - Creating a list of menu items.

Much like the `serve()` function above, the code in the `listOf()` function ends up calling the constructor of a generic class called `ArrayList`. The `ArrayList` class implements the `List` interface, which is also generic.

Most of the time, we just allow Kotlin's type inference to fill in the type arguments for us, but we could explicitly specify them like this:

```
val menu: List<String> = listOf<String>("Bagel", "Croissant", "Muffin", "Crumpet")
```

Listing 18.33 - Explicitly specifying the type argument for Listing 18.32.

The type argument to the `listOf()` function is `String` (regardless of whether we explicitly specify it or allow Kotlin to infer the type based on the function's arguments). The return type of this function depends on the type argument it was called with. Since it was called with a type argument of `String`, the return type is `List<String>`.

Pair

`Pair` is a generic [data class](#) that we came across in [Chapter 9](#). It has two type parameters, which determine the type of the two elements that it contains. As you might recall, we can create an instance of `Pair` by calling its constructor:

```
val pair = Pair("Crumpet", "Tea")
```

Listing 18.34 - Instantiating a Pair.

Again, we use type inference most of the time, but we *could* explicitly specify the two type arguments like this.

```
val pair: Pair<String, String> = Pair<String, String>("Crumpet", "Tea")
```

Listing 18.35 - Explicitly specifying the type arguments for Listing 18.34.

This listing looks much more intimidating than the previous listing, so it's usually a good idea to just allow Kotlin's type inference to do its thing.

The second way to create a `Pair` is to use the `to()` infix function, like this.

```
val pair = "Crumpet" to "Tea"
```

Listing 18.36 - Creating a Pair with the to infix function.

The `to()` function is a generic *extension* function, similar to the `pourIntoMug()` function we created above, except that it has two type arguments. The `to()` function exists to make our code read more naturally, so you should never explicitly specify the type arguments when calling it. Just to help demystify the magic, though, this is how you could do that.

```
val pair = "Crumpet".to<String, String>("Tea")
```

Listing 18.37 - Explicitly specifying the type arguments for Listing 18.36.

We've seen why generics are helpful, how to create them, how to use them, and a few examples of how they're used in the standard library. Before we wrap up this chapter, let's look at some of the trade-offs involved when we use them.

Trade-Offs of Generics

As we've seen, generics are a great way to reuse a class or interface, without needing to cast its properties or function results. They come with some trade-offs, though! Here are a few to consider.

Assignment Compatibility of Generic Types

Back in Listing 18.6 we had a version of the `Mug` class that was not generic. This is what it looked like:

```
class Mug(val beverage: Beverage)
```

Listing 18.38 - The non-generic version of Mug from Listing 18.6.

With this code, any `Mug` object can be assigned to any `Mug` variable, regardless of what kind of beverage it holds. For example, we can declare a `Mug` variable, and assign it a mug of coffee or a mug of tea.

```
val mugOfCoffee: Mug = Mug(Coffee.DARK_ROAST)
val mugOfTea: Mug = Mug(Tea.RED_TEAS)

var mug: Mug = mugOfCoffee
mug = mugOfTea
```

Listing 18.39 - A variable with the Mug type from Listing 18.38 can be assigned either a mug of coffee or a mug of tea.

Now let's consider a generic version of this class.

```
class Mug<T : Beverage>(val beverage: T)
```

Listing 18.40 - A generic version of the Mug class.

When we use this class, we'll end up with parameterized types such as `Mug<Coffee>` and `Mug<Tea>`. By default, these parameterized types are not assignment-compatible. For example, it's not possible to assign an instance of `Mug<Tea>` to a variable declared with `Mug<Coffee>`.

```
val mugOfCoffee: Mug<Coffee> = Mug(Coffee.DARK_ROAST)
val mugOfTea: Mug<Tea> = Mug(Tea.RED_TEAE)

var mug: Mug<Coffee> = mugOfCoffee
mug = mugOfTea
```

Listing 18.41 - Error: Type Mismatch. Required: Mug<Coffee>. Found: Mug<Tea>.

This isn't surprising. What can catch some developers by surprise, though, is that it's *also* not possible to assign `mugOfCoffee` or `mugOfTea` to a variable that has type `Mug<Beverage>`.

```
val mugOfCoffee: Mug<Coffee> = Mug(Coffee.DARK_ROAST)
val mugOfTea: Mug<Tea> = Mug(Tea.RED_TEAE)

var mug: Mug<Beverage> = mugOfCoffee
```

Listing 18.42 - Error: Type Mismatch: Required: Mug<Beverage>. Found: Mug<Coffee>

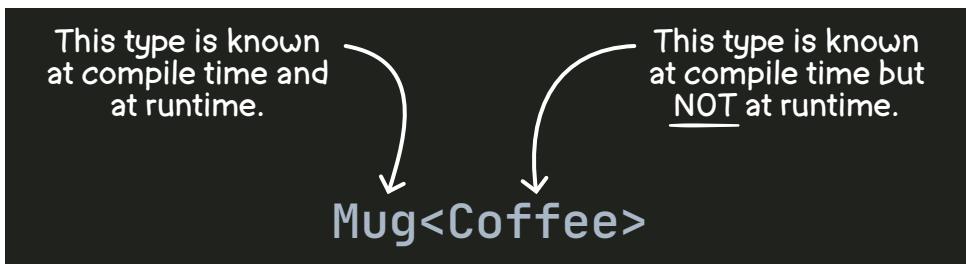
There are some ways that we can work around this, as we'll see throughout the next chapter! Note that this assignment does actually work when assigning directly from a constructor call.

```
val mug: Mug<Beverage> = Mug(Coffee.DARK_ROAST)
```

Listing 18.43 - Listing 18.42 compiles just fine when directly assigning the result of a constructor call.

Type Erasure

Probably the most significant trade-off is called **type erasure**. Although an object's type arguments are known at compile-time, they are *erased* before your code runs. In other words, an object's type arguments are *not* known at *runtime*.



Let's look at a few ways that type erasure can affect your code.

Checking the Type of Type Arguments

One consequence of type erasure is that it's not possible to use `is` to check the type of a parameterized type's *type argument* at runtime. For example, the following code tries to determine whether the `mug` instance has a type of `Mug<Tea>` or `Mug<Coffee>`. This results in a compiler error.

```
val mug: Mug<Beverage> = Mug(Coffee.MEDIUM_ROAST)

when (mug) {
    is Mug<Tea> -> println("Sipping on tea: ${mug.beverage}!")
    is Mug<Coffee> -> println("Sipping on coffee: ${mug.beverage}!")
}
```

Listing 18.44 - Error: Cannot check for instance of erased type: Mug<Tea>.

However, it *is* still possible to check the type of a *property* that was declared with a type parameter (such as `beverage`), so this works just fine:

```
val mug: Mug<Beverage> = Mug(Coffee.MEDIUM_ROAST)

when (mug.beverage) {
    is Tea -> println("Sipping on tea: ${mug.beverage}!")
    is Coffee -> println("Sipping on coffee: ${mug.beverage}!")
}
```

Listing 18.45 - Checking the type of the beverage rather than the type of the mug.

Function Overloads on JVM

Kotlin code can target different kinds of computer systems and environments. Most often, a Kotlin project targets the Java Virtual Machine (JVM), and if you've been following along with the code in this book, that's probably what you've been doing. However, you can also use Kotlin to create programs that run natively on different platforms, like Windows, Linux, Mac, and so on. In fact, you can even create Kotlin code that compiles down to JavaScript!

Once in a while, there's a limitation that affects some of these platforms, but not others. When it comes to type erasure, Kotlin code that targets the JVM has a limitation that doesn't affect native or JavaScript targets: it's not possible to use [function overloads](#)

where the functions' parameters differ only on their *type arguments*. For example, in the following code, the only difference between the signatures of these two functions is the type argument of their parameters. Compiling this code on the JVM produces an error.

These two function signatures are the same except for the type argument here...

```
fun drinkFrom(mug: Mug<Tea>) = println("Drinking tea")
fun drinkFrom(mug: Mug<Coffee>) = println("Drinking coffee")
```

... and here.

Even with these trade-offs, generics are incredibly helpful, so it's important to know about them!

Summary

As the seasons changed, so did the menu at Jennifer's bakery café, but thanks to Eric's newfound understanding of Kotlin generics, adapting to these changes became a piece of cake! And now that you know all about generics, you'll be able to adapt to changes, too! Here's a quick review of what we learned:

- The [problem](#) that generics solve.
- How to [declare](#) and [use](#) a generic type.
- How type parameter [constraints](#) can ensure that only the right kinds of type arguments are used.
- How to use [generic interfaces and superclasses](#).
- How to declare and use [generic functions](#).
- How generics are [used in the standard library](#).
- [Trade-offs](#) of using generics.

As we saw in this chapter, the subtyping of generics doesn't always work like we expect - `Mug<Coffee>` is not naturally a subtype of `Mug<Beverage>`. However, with a few small changes, we can make that happen. Stay tuned for the next chapter where we cover the fascinating topic of generic variance!

Kotlin: An Illustrated Guide

Chapter 19

Generic Variance (DRAFT)

Cover Image
Coming Soon!

Hey! Thanks for checking out this draft of Chapter 19!

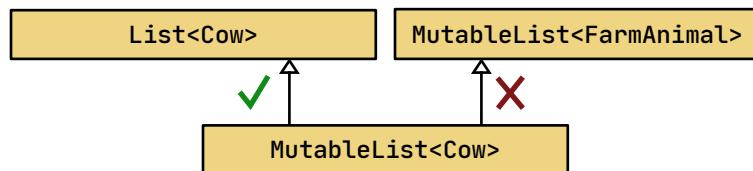
Although this chapter is not yet in its final form, the main text is done, and it's ready for reader feedback! There are a few things to keep in mind about draft chapters:

- The illustrations are still on the way. Instead of final, polished illustrations, you'll see sketches that convey the main idea I'm going for.
- Typically, when referring to a topic already covered earlier in the book, I include a link for convenience. Those links aren't present in a draft chapter like this one.
- Also, the book's index does not include topics from the draft.
- Code listings have numbers, but their captions will come later.
- I'd love to hear your feedback about this chapter! At the top of each page is a link to an online version of this chapter where you can add comments.

Thanks again!

As we learned in Chapter 12, we can substitute a subtype anywhere that a supertype is declared.

When it comes to generics, though, things don't always work the way we might expect. While `MutableList<Cow>` is a subtype of `List<Cow>`, it's not a subtype of `MutableList<FarmAnimal>`.



With a little bit of magic, though, we can finagle parameterized types into becoming the subtypes that we need them to be. To do that, we first need to learn about *covariance* and *contravariance*, so let's jump in!

Covariance

Parker works for the local Parks & Recreation Department. Since the playground is often full of parents with hungry children, he thought, "Let's add a vending machine in the pavilion." So he spoke with Vinnie, a guy who owns a nearby business that sets up and stocks vending machines.

"I need a vending machine where kids can put in a *coin* and get back a *snack*", said Parker. Vinnie agreed, so they drew up a contract.

Once the paperwork was signed, Vinnie installed the new vending machine at the park, where customers could insert a coin and receive a random snack, like trail mix, a bag of gummy bears, or a candy bar. The next day, when Parker saw a kid insert a nickel and receive a bag of trail mix from the machine, he smiled to himself, knowing all was well.



A few weeks later, Parker saw Vinnie replacing the machine. Parker was a bit nervous about the change, so he spoke to him. "This new machine is still going to *accept a coin* and *return a snack*, right?"

"Yes", began Vinnie, "Trail mix and gummy bears are in short supply, though, so this vending machine is *only* going to return a candy bar. That's still okay, right?"

Parker thought about it, and finally replied, "As long as it *accepts a coin* and *returns a snack*. A candy bar is a kind of snack, so that should be fine."

After the new machine was substituted for the old one, a child came up, inserted a coin, and received a candy bar. Parker smiled, glad that the new machine did what it was supposed to.



A few weeks passed. One day, Vinnie was at it *again*, replacing the machine with yet *another* one. A few minutes after the installation was complete, an irate father complained to Parker, "What's wrong with the new vending machine? My hungry kid put in a coin and instead of getting a snack, he got a toy action figure - *he can't eat a toy!*"

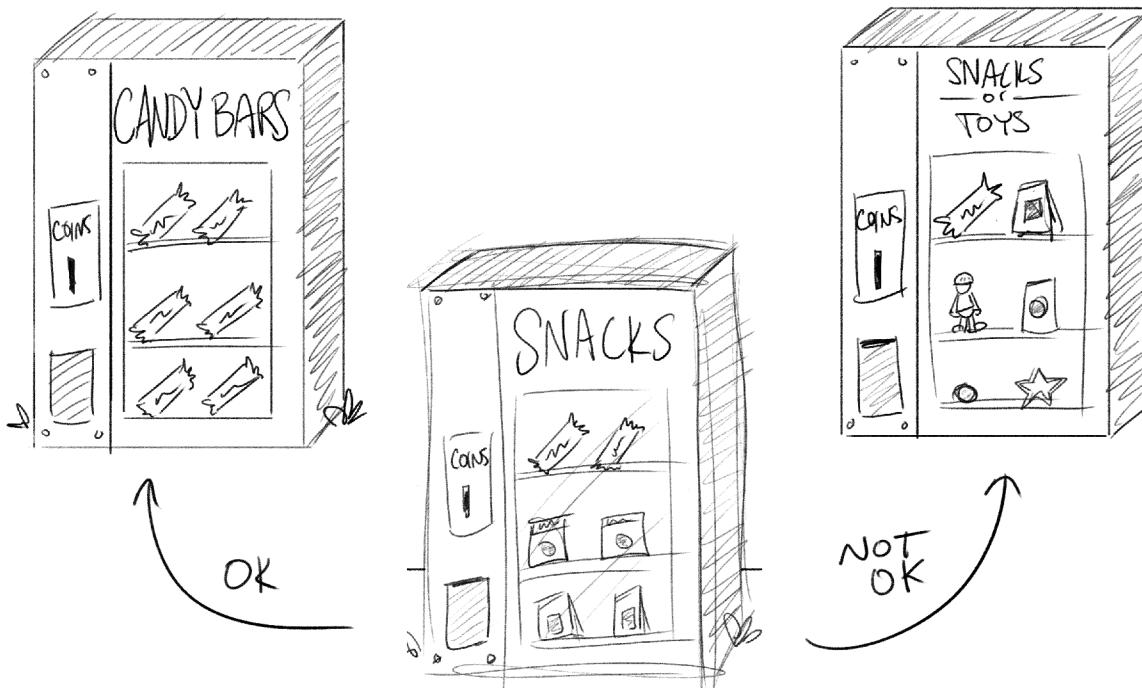


Parker quickly caught up with Vinnie again. When asked about the new machine, Vinnie replied, "Oh, I thought I'd switch out the old vending machine with a new one that returns *either* a snack or a toy."

"No, no, no! The contract said the machine would accept a *coin* and return a *snack*. Toys are not snacks!" Realizing that this new machine didn't do what Parker needed, Vinnie agreed to resolve the issue.

As we can see from this story, some substitutions work, but others don't.

- When Vinnie replaced the original vending machine with one that only returns candy bars, everything was fine, because candy bars are still a kind of snack that kids can eat.
- However, when he replaced it with the machine that returns either a snack *or* a toy, he broke his contract - the machine no longer returned *only* snacks.

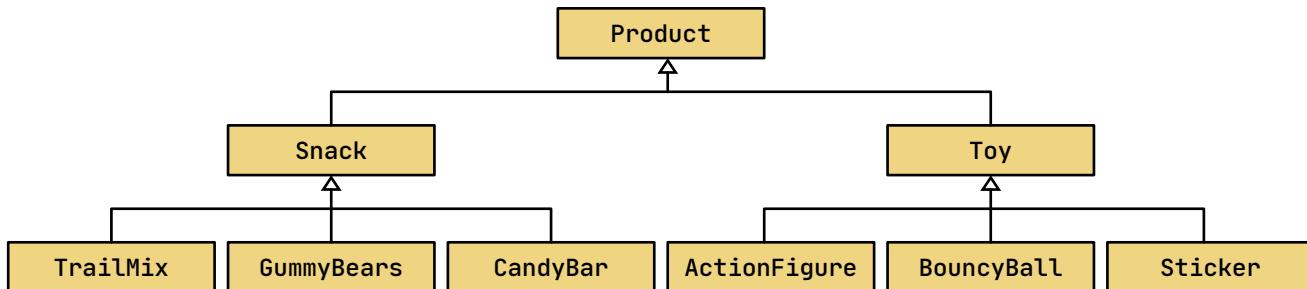


This brings up some important points for us to explore about *substitution* in Kotlin. To get started, let's model the first vending machine that Vinnie provided - the one that accepted a coin and returned a snack.

```
open class VendingMachine {
    open fun purchase(money: Coin): Snack = randomSnack()
}
```

Listing 19.1 - A simple VendingMachine class that accepts a Coin and returns a Snack.

Although this chapter won't include definitions for **Snack** and its related types, here's a class diagram that shows them with their type hierarchies.



Now let's model Vinnie's *second* vending machine by extending the one above. This one looks almost identical to it, but instead of a random snack, it only ever returns a candy bar.

```
class CandyBarMachine : VendingMachine() {  
    override fun purchase(money: Coin): Snack = CandyBar()  
}
```

Listing 19.2 - Modeling Vinnie's second vending machine by extending the first.

Any code that expects a `VendingMachine` can work with a `CandyBarMachine`, because it's a subtype of `VendingMachine` - it still *accepts a coin and returns a snack*, just like `VendingMachine` does. Because of this, we can substitute a `CandyBarMachine` for a `VendingMachine`. In other words, we can assign a `CandyBarMachine` to a variable declared as a `VendingMachine`.

```
val machine: VendingMachine = CandyBarMachine()  
  
supertype ----- subtype
```

Similarly, we can send it as an argument to a function that expects a `VendingMachine`.

```
supertype -----  
fun purchaseSnackFrom(machine: VendingMachine) = machine.purchase(Dime())  
  
val snack = purchaseSnackFrom(CandyBarMachine()) subtype  
-----
```

As we saw in the story, a `CandyBar` is a kind of `Snack`, so it's also safe to *declare* that the `CandyBarMachine` only returns `CandyBar` objects. Let's take the code from Listing 19.2 and update the return type.

```
class CandyBarMachine : VendingMachine() {  
    override fun purchase(money: Coin): CandyBar = CandyBar()  
}
```

Listing 19.3 - Declaring that CandyBarMachine only returns CandyBar objects.

As the story revealed, this only works one way. When Vinnie tried to replace the vending machine with one that returned a product that could be *either* a snack or a toy, he broke his contract. Kids expect to be able to eat anything that the vending machine provides, but toys aren't edible!

The same principle holds true in Kotlin - a subclass can't return a more *general* type. For example, if we create a `ToyOrSnackMachine` that tries to return any kind of `Product`, we'll get a compile-time error.

```
class ToyOrSnackMachine : VendingMachine() {  
    override fun purchase(money: Coin): Product = randomToyOrSnack()  
}
```

Listing 19.4 - Error: Return type is 'Product', which is not a subtype of overridden

By the way...

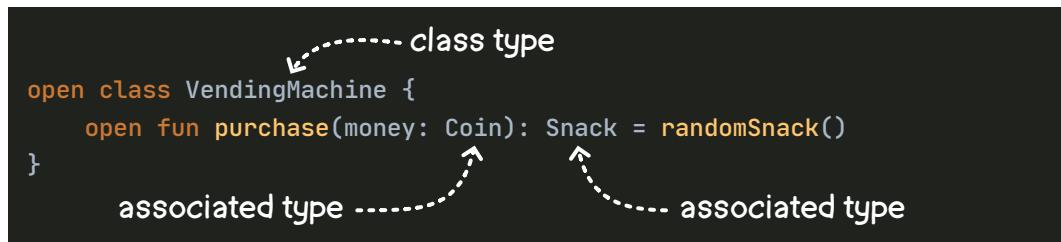
When a `CandyBarMachine` is assigned to a variable declared as `CandyBarMachine`, it'll return the candy bar as a `CandyBar`. However, when a `CandyBarMachine` is assigned to a `VendingMachine` variable, it'll return the candy bar as a `Snack`. Of course, you can still cast it from `Snack` to `CandyBar`.

```
val candyBarMachine: CandyBarMachine = CandyBarMachine()
val vendingMachine: VendingMachine = CandyBarMachine()

val candyBar: CandyBar = candyBarMachine.purchase(Dime())
val snack: Snack = vendingMachine.purchase(Dime())
```

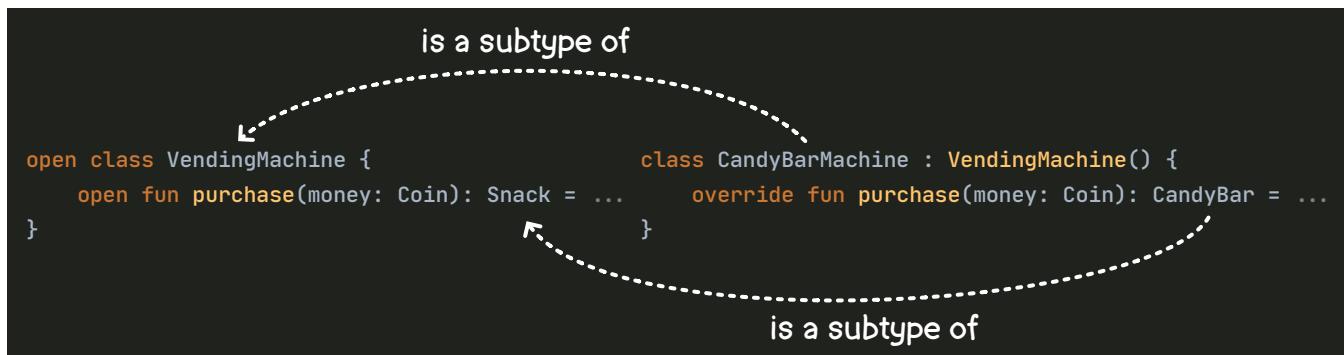
Most classes and interfaces associate with other types. Associated types can appear in a variety of places - as function parameter types, function return types, property types, and so on. For example, the `VendingMachine` class associates with two other types:

- `Coin` - the type of the parameter in `purchase()`
- `Snack` - the return type of `purchase()`



The `VendingMachine` class can have supertypes and subtypes. Similarly, the associated types `Coin` and `Snack` can also have their own supertypes and subtypes.

The nature of the relationship between the type hierarchy of `VendingMachine` and the type hierarchy of its associated types can be described by **variance**. As we look at the code of `VendingMachine` and `CandyBarMachine` side by side, we can see that relationship emerge.



Specifically, we can see that as we make a more *specific* `VendingMachine`, we can return a more *specific* `Snack` from the `purchase()` function. Because they become more specific *together*, this kind of variance is called

covariance, where “co-” is a prefix that means “together”. When talking about variance, developers normally say it one of these ways:

- “A type is covariant *with regard to* its return types”
- “A type is covariant *on* its return types”

To sum up this section, just remember that within a subtype (e.g., `CandyBarMachine`) a function can return a more *specific* type (e.g., `CandyBar`) than declared in its supertype, but not a more *general* type.

The adventures of Parker and Vinnie continue, though! Brace yourself - Vinnie is about to try substituting a few more vending machines!

Contravariance

A few weeks later, Parker saw Vinnie replacing the machine yet again. “We’re upgrading our machines so that they accept both coins *and* bills,” he explained.

Parker thought about it. “Well, I guess that’s fine. As long as the machine still *accepts a coin* and *returns a snack*.” After the new machine was installed, a kid walked up, inserted a dime, and got a snack from the machine. “Great,” Parker thought to himself, “Everything is still good”.

Things continued well over the next few weeks. Even though this new vending machine had a bill acceptor, none of the kids ever used it, because they only ever had coins.



Wouldn't you know it - a few weeks later, Parker saw Vinnie replacing the machine one more time. Parker shrugged it off and continued on his way.

Ten minutes later, though, a young girl was crying next to the snack machine. "What's wrong?" he asked her.

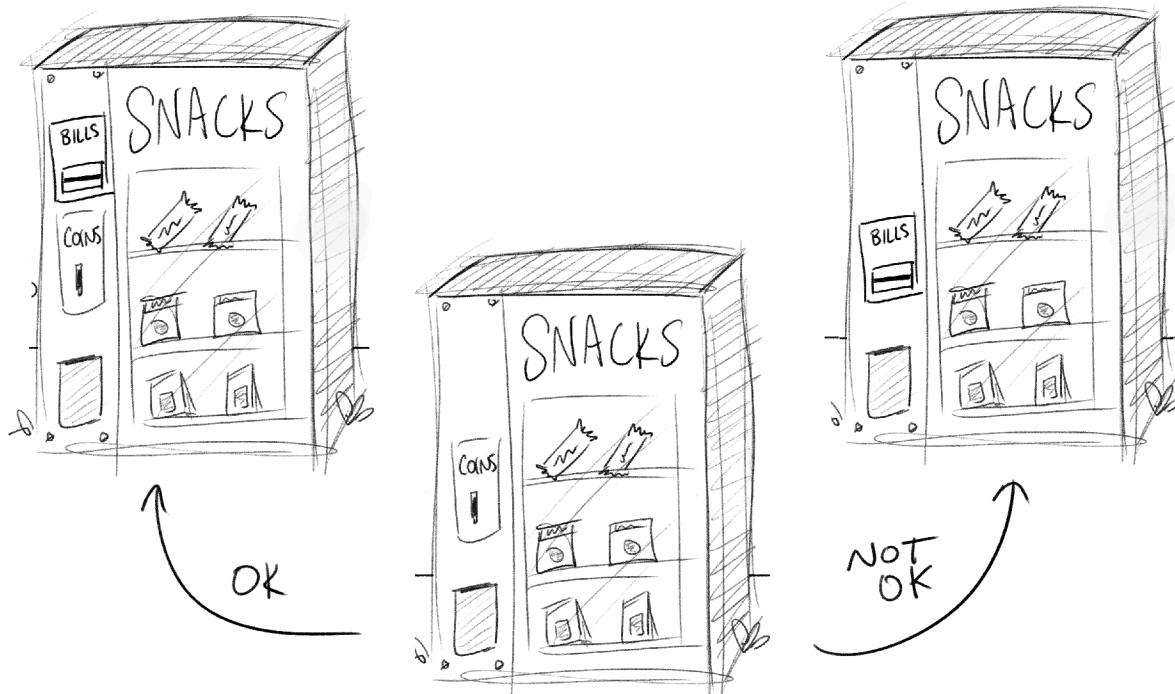
"I wanted a snack. I have a nickel, but this machine only accepts a dime!" Parker looked at the new vending machine, and sure enough, it *only* had a slot for dimes, but not any other kind of coin.



Parker complained to Vinnie again. "No, no, no! The contract said the machine would *accept a coin and return a snack*. A nickel is a kind of coin, so the machine still needs to accept it!" Embarrassed, Vinnie put back the previous vending machine.

This story shows again that some substitutions work, and some don't.

- When Vinnie replaced the original vending machine with the one that accepted both coins and bills, everything still worked, because it still accepted coins. The kids only ever had coins on them, so they never actually *used* the bill acceptor, but there was no harm in the machine having one, as long as it could still accept coins.
- However, when Vinnie replaced it with the machine that accepted only dimes, he broke his contract - the machine no longer accepted nickels, quarters, or any other kind of coin.

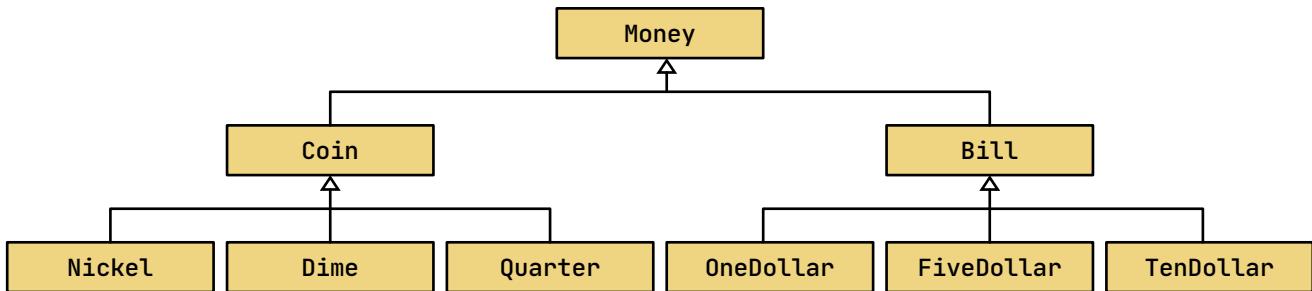


This illustrates additional points about substitution. Let's take a look at the original `VendingMachine` we created back in Listing 19.1.

```
open class VendingMachine {  
    open fun purchase(money: Coin): Snack = randomSnack()  
}
```

Listing 19.5 - The `VendingMachine` class from Listing 19.1.

As with `Snack`, this chapter won't include any class definitions for `Coin` and its type hierarchy, but here's a class diagram showing how they relate.



As we saw, it was safe for Vinnie to replace the coin-based vending machine with one that can accept *either* coins or bills. Likewise, it would be safe for Kotlin to allow a subtype to declare a more *general* parameter type.

But guess what? If we try changing the parameter type from `Coin` to `Money`, we'll get a compiler error!

```
class AnyMoneyVendingMachine : VendingMachine() {
    override fun purchase(money: Money): Snack = randomSnack()
}
```

Listing 19.6 - Error: 'purchase' overrides nothing

Again, this would be perfectly safe for Kotlin's type system to allow. So why does this cause a compiler error?

As you might recall, Kotlin allows us to overload a function - a class or interface can have multiple functions that have the same name, as long as their parameter types differ. In order to support this feature, Kotlin won't allow us to change the type of a function parameter in subtypes, as we're trying to do above.

One way to fix this is to simply remove the `override` keyword, which means we'll be *overloading* the function instead of *overriding* it.

```
class AnyMoneyVendingMachine : VendingMachine() {
    fun purchase(money: Money): Snack = randomSnack()
}
```

Listing 19.7 - Overloading the purchase() function from the VendingMachine superclass.

Just keep in mind that when we do this, we end up with *two* `purchase()` functions, each with its own body - one in `AnyMoneyVendingMachine` and one in `VendingMachine`.

So Kotlin's overloading feature is getting in the way. Overloading only applies to functions declared with the `fun` keyword, though, so to work around this, we can change `purchase()` from a function to a property that has a function type, like this.

```
open class VendingMachine {
    open val purchase: (Coin) -> Snack = { randomSnack() }
}
```

Listing 19.8 - Rewriting purchase() as a property with a function type.

With this change, we can rewrite `AnyMoneyVendingMachine` to override that property.

```
class AnyMoneyVendingMachine : VendingMachine() {
    override val purchase: (Coin) -> Snack = { randomSnack() }
}
```

Listing 19.9 - Overriding the purchase property in the subclass.

And finally, we can replace the `Coin` parameter type with `Money`.

```
class AnyMoneyVendingMachine : VendingMachine() {
    override val purchase: (Money) -> Snack = { randomSnack() }
}
```

Listing 19.10 - Replacing the parameter with a more general type.

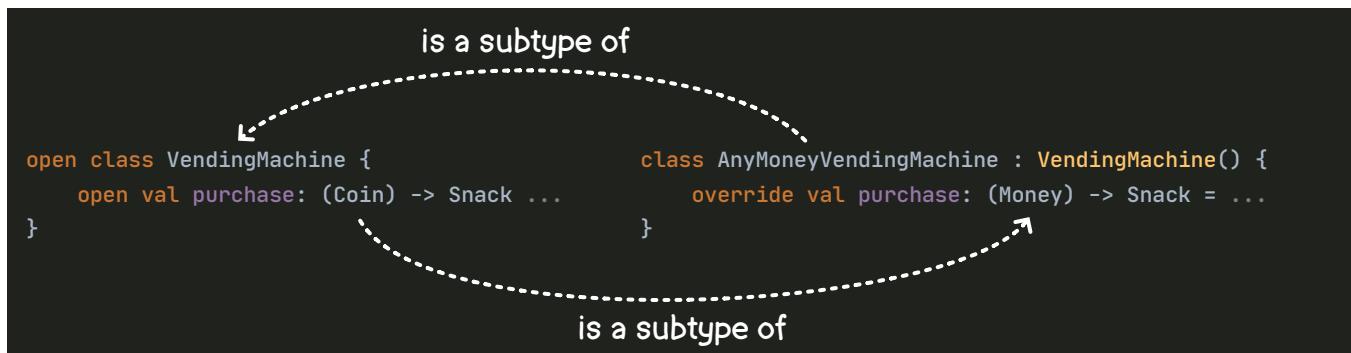
With this, we can assign an instance of `AnyMoneyVendingMachine` to a variable whose type is `VendingMachine`. When assigned to a `VendingMachine` variable, it can only accept a coin. But when assigned to an `AnyMoneyVendingMachine`, it can accept either a coin or a bill.

```
val vendingMachine: VendingMachine = AnyMoneyVendingMachine()
val anyMoneyMachine: AnyMoneyVendingMachine = AnyMoneyVendingMachine()

val snack1: Snack = vendingMachine.purchase(Dime())
val snack2: Snack = anyMoneyMachine.purchase(Dime())
val snack3: Snack = anyMoneyMachine.purchase(OneDollarBill())
```

Listing 19.11 - The object's type affects which argument types the `purchase()` function accepts.

As we did earlier, we can put these two classes side by side and discover the nature of the variance between the class type and the return type of `purchase()`.



This time, we can see that as we make a more *specific* `VendingMachine`, we can accept a more *general* parameter type in `purchase`. The arrows above are pointing in *opposite* directions, so this kind of variance is called **contravariance**.

As you recall from the story, when Vinnie tried to replace the vending machine with one that accepted a more *specific* kind of coin, he broke his contract. Similarly, we can't create a subclass that accepts a more *specific* type than its superclass did. For example, if we update `purchase` so that its parameter type is `Dime`, we'll get a compiler error.

```
class AnyMoneyVendingMachine : VendingMachine() {
    override val purchase: (Dime) -> Snack = { randomSnack() }
}
```

Listing 19.12 - Error: Property type is (Dime) -> Snack, which is not a subtype of overridden (Coin) -> Snack.

So, we learned from this story that a subtype can declare that it accepts a more *general* type, but not a more *specific* type.

Now that we understand variance - including covariance and contravariance - its time to revisit what we learned, and see how these concepts apply to generics.

What Makes a Subtype a Subtype?

Whenever Vinnie replaced one vending machine with another, everything was fine as long as the new machine did *everything* that the contract said it would. Specifically, the contract stated that the vending machine must *accept a coin* and *return a snack*. If the new machine did those things, then it was a suitable substitute. On the two occasions when it broke the contract, it was not a suitable substitute.

So what makes a subtype a subtype? *The ability to substitute it for one of its supertypes.* A subtype must *fully support* the contract of its supertype. Specifically, this means it must obey the following three rules:¹

1. The subtype must have all of the same *public properties and functions* as its supertype.
2. Its function *parameter types* must be the *same as or more general than* the ones in its supertype.
3. Its function *return types* must be the *same as or more specific than* the ones in its supertype.

By the way: Variance and Properties

To keep things simple, this chapter focuses mainly on variance as it relates to function parameter types and return types. However, variance does still apply to properties as well.

- A property that's declared with `val` can be overridden and given a more specific type, just as you can do with a function return type.
- A property that's declared with `var` can also be overridden, of course, but its type must be *exactly the same* as what's declared in its supertype - you can't replace it with a more specific or more general type.

We usually think of a subtype as a class that extends another class, an interface that extends another interface, or a class that implements an interface. In all three of these cases, Kotlin's type system will ensure that the subclass follows the three rules above. However, when it comes to parameterized types - such as `VendingMachine<Snack>` and `VendingMachine<CandyBar>` - we can't *explicitly* declare that one type is a subtype of another.

To help demonstrate this, let's convert `VendingMachine` to a generic class. As a part of this change, we'll add a `snack` constructor parameter, which is the snack that will get returned when the `purchase()` function is called.²

¹The goal of this chapter is to explain generic variance, so these three rules are focused on the structure of the types. More strictly speaking, though, the behaviors of the classes should be compatible, as well. (See Liskov, B., & Wing, J. M. (1994). A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6), 1811-1841). Also, Meilir Page-Jones observes these three rules more generally through the lens of preconditions, postconditions, and class invariants. (Page-Jones, M. (2000). *Fundamentals of Object-Oriented Design in UML*. Addison-Wesley Professional. p. 283).

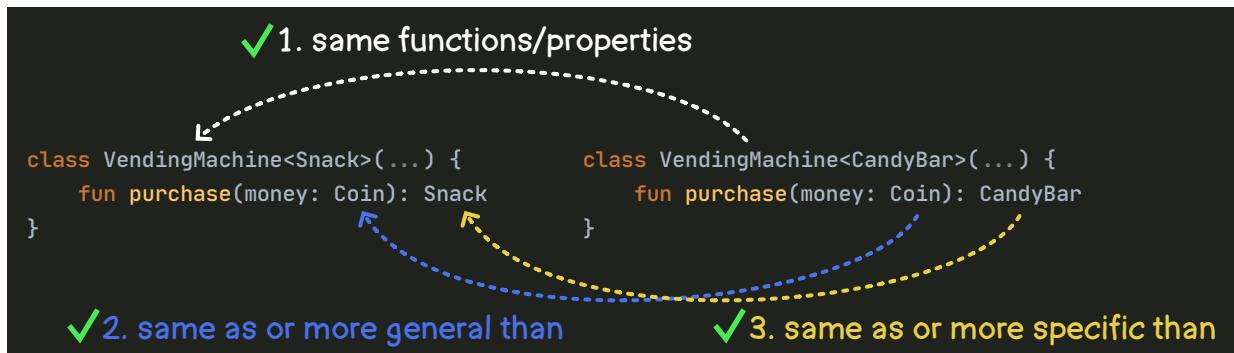
²Since the return type of `purchase()` varies depending on the type argument, we can't just use `randomSnack()` any more - the snack has to have the same type as the type argument. For example, in `VendingMachine<CandyBar>`, the `purchase()` function must return a `CandyBar`, not a `Snack`. Setting this value in the constructor is a simple way to do this. If you prefer, you could change the `snack` property to a function type that constructs new snacks. e.g., `private val snack: () -> T`.

```
class VendingMachine<T : Snack>(private val snack: T) {
    fun purchase(money: Coin): T = snack
}
```

Listing 19.13 - A generic version of the VendingMachine class.

As we learned in the last chapter we can create parameterized types from this generic, such as `VendingMachine<Snack>` and `VendingMachine<CandyBar>`. Even though we can't explicitly declare that `VendingMachine<CandyBar>` is a subtype of `VendingMachine<Snack>`, it would be perfectly safe for this to be the case, because it wouldn't break the contract - all three rules above would be satisfied.

We can visualize this by imagining what the *effective* parameterized type would look like. In other words, let's take the generic `VendingMachine` above, and everywhere that the type parameter appears, we'll replace it with the type argument. Are all three rules satisfied?



So, `VendingMachine<CandyBar>` fully satisfies the contract of `VendingMachine<Snack>`, which means it's safe for it to be one of its subtypes. This won't just happen automatically, though. For example, if we try assigning it to a variable declared as `VendingMachine<Snack>`, we'll get a compiler error.

```
val candyBarMachine: VendingMachine<CandyBar> = VendingMachine(CandyBar())
val vendingMachine: VendingMachine<Snack> = candyBarMachine
```

Listing 19.14 - Error: Type Mismatch. Required: VendingMachine<Snack>. Found: VendingMachine<CandyBar>.

So, `VendingMachine<CandyBar>` won't be a subtype of `VendingMachine<Snack>` until we tell Kotlin our intent by doing one more thing.

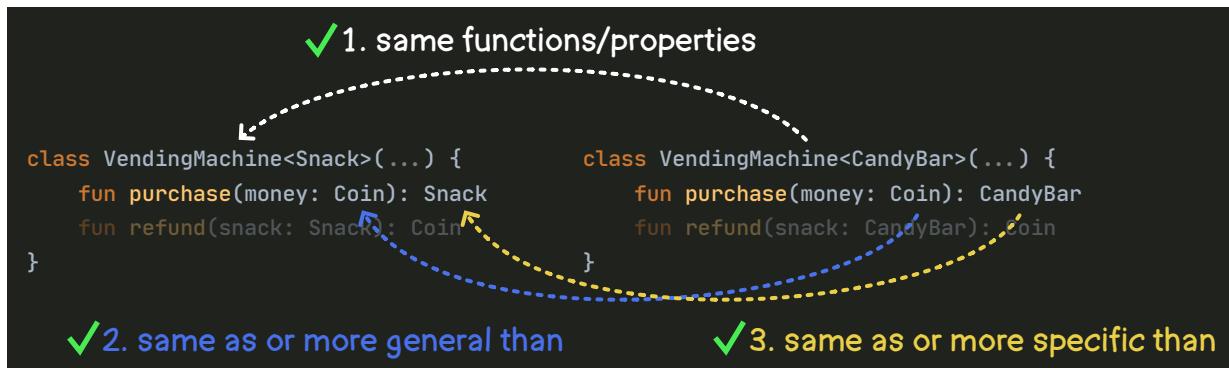
Variance Modifiers

As we saw above, `VendingMachine<CandyBar>` fully satisfies the contract of `VendingMachine<Snack>`, so it *should* be possible for it to be a subtype of it. However, a type parameter can *potentially* be used in lots of places throughout the body of a generic type. It could be used as the return type of a function, as the parameter of a function, as the type of a property, and so on. Let's consider what would happen if we were to add a `refund()` function to the `VendingMachine` interface.

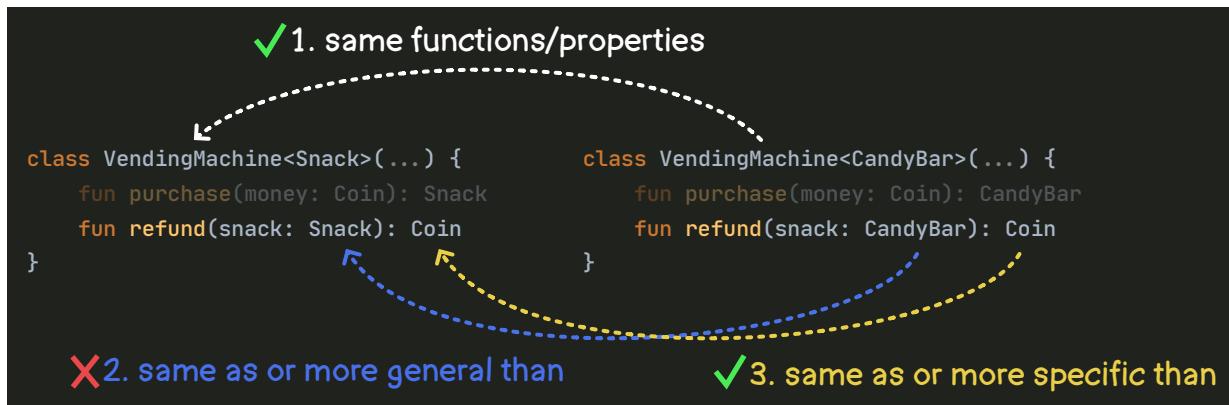
```
class VendingMachine<T : Snack>(private val snack: T) {
    fun purchase(money: Coin): T = snack
    fun refund(snack: T): Coin = Dime()
}
```

Listing 19.5 - Adding a `refund()` function to the `VendingMachine` class.

In this case, the type parameter `T` appears *both* as a function result type *and* as a function parameter type. Would this version of `VendingMachine<CandyBar>` satisfy the contract of `VendingMachine<Snack>`? Let's compare the effective parameterized types again. First, let's look at the `purchase()` function.



As before, when it comes to the `purchase()` function, `VendingMachine<CandyBar>` satisfies the contract of `VendingMachine<Snack>`. So far so good. Now let's look at the `refund()` function.



Yikes! Even though the *return type* of `refund()` satisfies the contract, the *parameter type* does not, because `CandyBar` is not the “same as or more general than” `Snack` - it’s more *specific*. So, if `VendingMachine<T>` includes the `refund()` function, then `VendingMachine<CandyBar>` *cannot* be a subtype of `VendingMachine<Snack>`.

If Kotlin is going to treat `VendingMachine<CandyBar>` as a subtype of `VendingMachine<Snack>`, we have to promise Kotlin that we *won’t* use this type parameter as a function parameter type, like we did in `refund()`. Instead, it can only appear in **out-positions** - in other words, as a function return type, or as the type of a read-only property.

To make this promise, we can add a **variance modifier** to the type parameter. Kotlin has two variance modifiers - let's check them out.

The out modifier

The first variance modifier is named **out**, and it's our way of telling Kotlin that this type parameter will only appear in an *out-position*. Let's add the **out** modifier to type parameter T.

```
class VendingMachine<out T : Snack>(private val snack: T) {  
    fun purchase(money: Coin): T = snack  
}
```

*Listing 19.16 - Adding an **out** modifier to a type parameter.*

This simple change is all that's needed to get the code from Listing 19.14 to compile without errors - **VendingMachine<CandyBar>** is now a subtype of **VendingMachine<Snack>**!

```
val candyBarMachine: VendingMachine<CandyBar> = VendingMachine(CandyBar())  
val vendingMachine: VendingMachine<Snack> = candyBarMachine
```

Listing 19.17 - The code from Listing 19.14 now compiles without errors.

As we saw earlier in this chapter, a type is *covariant* with its associated function *result types*. So by ensuring that this type parameter is *only used as a result type*, we know that it's safe for the **VendingMachine** type to be covariant with T.

Again, the **out** modifier is a promise to Kotlin that we will only use the type parameter in the *out-position*. If we try to use this same type parameter in an **in-position** - that is, as a function parameter type - then we'll get a compiler error. To demonstrate this, we can pop in the **refund()** function from Listing 19.15, and watch as the compiler calls us out for breaking our promise.

```
class VendingMachine<out T : Snack>(private val snack: T) {  
    fun purchase(money: Coin): T = snack  
    fun refund(snack: T): Coin = Dime()  
}
```

*Listing 19.18 - Error: Type parameter T is declared as **out** but occurs in **in** position.*

So, by adding the **out** modifier to the type parameter, we've got the *advantage* that **VendingMachine<CandyBar>** is now a subtype of **VendingMachine<Snack>**, but we've got the *disadvantage* that we can no longer use T in an *in-position*.

The in modifier

As you probably guessed, Kotlin includes a complement to the **out** modifier, called **in**. But before we look at it, let's shake things up in the **VendingMachine** class. Instead of a type parameter for its *snack* type, let's use the type parameter for its *money* type.

```
class VendingMachine<T : Money> {
    fun purchase(money: T): Snack = randomSnack()
}
```

Listing 19.19 - Changing the type parameter to represent the money instead of the snack.

Similar to before, we can try to assign an instance of `VendingMachine<Money>` to `VendingMachine<Coin>`, but we'll get an error.

```
val moneyVendingMachine: VendingMachine<Money> = VendingMachine()
val coinVendingMachine: VendingMachine<Coin> = moneyVendingMachine
```

Listing 19.20 - Error: Type Mismatch. Required: VendingMachine<Coin>. Found: VendingMachine<Money>.

Even though `T` is only being used in the *in-position*, we have to declare this to Kotlin with the `in` variance modifier.

```
class VendingMachine<in T : Money> {
    fun purchase(money: T): Snack = randomSnack()
}
```

Listing 19.21 - Adding an in variance modifier to the type parameter.

With this change, the code from Listing 19.20 successfully compiles.

```
val moneyVendingMachine: VendingMachine<Money> = VendingMachine()
val coinVendingMachine: VendingMachine<Coin> = moneyVendingMachine
```

Listing 19.22 - The code from Listing 19.20 now compiles without any errors.

As with the `out` modifier, we've made a trade-off here: when we declare a type parameter with the `in` modifier, we're promising Kotlin that it will only ever appear in the *in-position*. Again, if we add a `refund()` function, we would need `T` in the *out-position* - as that function's return type - which would cause a compiler error.

```
class VendingMachine<in T : Money>(<private val money: T>) {
    fun purchase(money: T): Snack = randomSnack()
    fun refund(snack: Snack): T = money
}
```

Listing 19.23 - Error: Type parameter T is declared as 'in' but occurs in 'out' position in type T.

In summary:

- The `out` modifier can be used to ensure that the type parameter will only appear publicly in an *out-position*, which makes it safe for covariance.
- Conversely, the `in` modifier can be used to ensure that it will only appear publicly in an *in-position*, so that it's safe for contravariance.

In order to keep things simple, we've only used one type parameter at a time so far in this chapter. It's entirely possible to have multiple type parameters, though, and when we do, we can use a variance modifier on each one. Let's see how that looks.

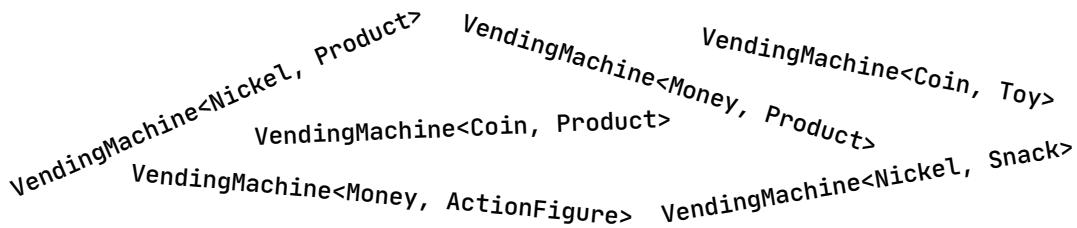
Variance on Multiple Type Parameters

It's important to note that generic variance doesn't describe the type as a whole - it describes the relationship of the *type to one of its type parameters*. So, it can be covariant on one and contravariant on another. Instead of a single type parameter for either the **Money** or **Product**, let's include one for each.

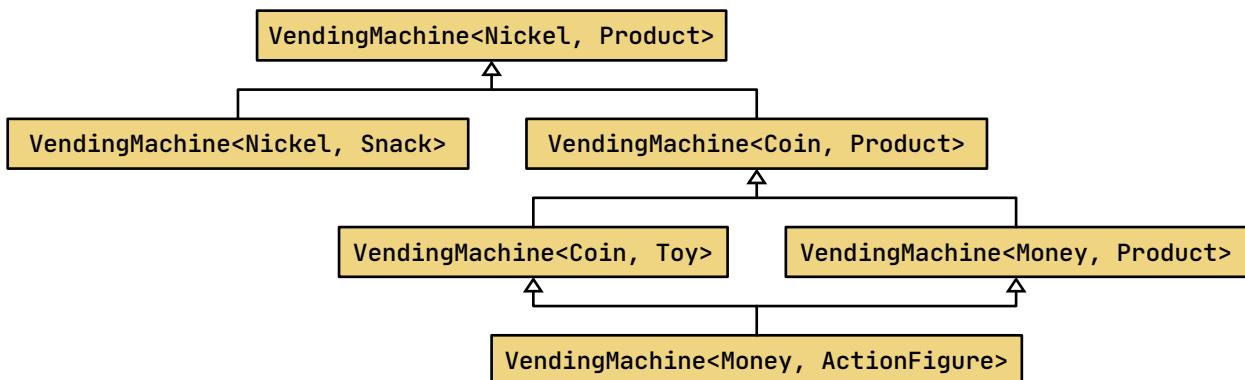
```
class VendingMachine<in T : Money, out R: Product>(private val product: R) {  
    fun purchase(money: T): R = product  
}
```

Listing 19.24 - Variance on multiple type parameters in one class.

In this code, `VendingMachine` is *contravariant* with respect to `T`, and *covariant* with respect to `R`. From this generic class, we could instantiate a wide range of parameterized types. Some of them would include:



With two type parameters, all those types of money, and all those types of product, it's easy to get lost in which of the parameterized types are subtypes of the others. Just remember that a subtype must *fully support* the contract of its supertype. Here's how the type hierarchy looks for the parameterized types above.



It's helpful to know that we can apply variance modifiers to multiple type parameters. For the rest of this chapter, though, we'll go back to a single type parameter in order to keep the examples easy to follow.

We've seen how the `in` and `out` modifiers create variance, but we've also seen the trade-offs - type parameters declared with `in` can't be used as a result type, and those declared with `out` can't be used as a function

parameter type. But sometimes, those trade-offs just won't cut it. When that's the case, we can still get some of the benefits of variance by using *type projections*.

Type Projections

So far, we've been able to get `VendingMachine<CandyBar>` to be a subtype of `VendingMachine<Snack>` by using variance modifiers on our type parameters. Let's update the code from Listing 19.16 so that it works with any kind of `Product` and any kind of `Money`.

```
class VendingMachine<out T : Product>(private val product: T) {
    fun purchase(money: Money): T = product
}
```

Listing 19.25 - Updating the VendingMachine to work with any kind of Product and any kind of Money.

In this code, we've got `out` on the type parameter again. Because we put this variance modifier where the type parameter is *declared*, this is called **declaration-site variance**. We can't always use declaration-site variance, though. For example, what if we really, truly need that `refund()` function from Listing 19.18?

```
class VendingMachine<T : Product>(private val product: T) {
    fun purchase(money: Money): T = product
    fun refund(product: T): Money = Dime()
}
```

Listing 19.26 - Adding the refund() function to our new VendingMachine class.

We can't use the `out` variance modifier, because `T` is used in an in-position in `refund()`. And we can't use the `in` modifier, because it's used in an out-position in `purchase()`. Are we just out of luck, here? Does this mean that `VendingMachine<CandyBar>` will never be considered a subtype of `VendingMachine<Snack>`?

Thankfully, Kotlin provides a second option. Instead of using a variance modifier on a type *parameter*, we can use it on a type *argument*. To demonstrate this, let's start with a function that accepts a `VendingMachine<Snack>`.

```
fun getSnackFrom(machine: VendingMachine<Snack>): Snack {
    return machine.purchase(Dime())
}
```

Listing 19.27 - A function that accepts a VendingMachine<Snack>.

Since there are no variance modifiers on the type parameter in Listing 19.26, `VendingMachine<CandyBar>` is not a subtype of `VendingMachine<Snack>`, so we won't be able to call the function with an instance of `CandyBarMachine`.

```
val candyBarMachine: VendingMachine<CandyBar> = VendingMachine(CandyBar())
getSnackFrom(candyBarMachine)
```

Listing 19.28 - Error: Type Mismatch. Required: VendingMachine<Snack>; Found: VendingMachine<CandyBar>.

Now let's add an `out` modifier to the type *argument* in the `getSnackFrom()` function, like this.

```
fun getSnackFrom(machine: VendingMachine<out Snack>): Snack {  
    return machine.purchase(Dime())  
}
```

Listing 19.29 - Adding the `out` variance modifier to a type argument.

With this change, the code from Listing 19.28 compiles successfully!

```
val candyBarMachine: VendingMachine<CandyBar> = VendingMachine(CandyBar())  
getSnackFrom(candyBarMachine)
```

Listing 19.30 - The code from Listing 19.28 now compiles successfully.

When we put a variance modifier on a type *argument* rather than a type *parameter*, we create variance at the place in our code where we're *using* the type (e.g., `getSnackFrom()`) instead of where we *declared* it (e.g., `VendingMachine`). For this reason, we call this **use-site variance**.

Just like declaration-site variance, use-site variance comes with trade-offs. As we'll see in a moment, because the `machine` parameter has the `out` modifier, we won't be able to call the `refund()` function inside the body of this function. Thankfully, the body of this function has no need for the `refund()` function, so this trade-off is entirely acceptable here!

```
fun getSnackFrom(machine: VendingMachine<out Snack>): Snack {  
    return machine.purchase(Dime())  
}
```

} This function body
can't call `refund()`, but
it also has no need to.

Keep in mind that *declaration-site variance* applies to the whole project, but *use-site variance* works only in a specific part of the project where we put the variance modifier on the type argument. In the example code above, it only works for the `getSnackFrom()` function. So, if other functions in our project still need to use the `refund()` function, that's completely fine. In those places, `VendingMachine<CandyBar>` simply wouldn't be able to be a subtype of `VendingMachine<Snack>`.

Out-Projections

It's important to note that, within the body of this function, `machine` does *not* have the type `VendingMachine<Snack>`. It has the type `VendingMachine<out Snack>`, which is a **type projection**. Since this type argument has the `out` modifier on it, this particular kind of type projection is called an **out-projection**.

What exactly is a projection?

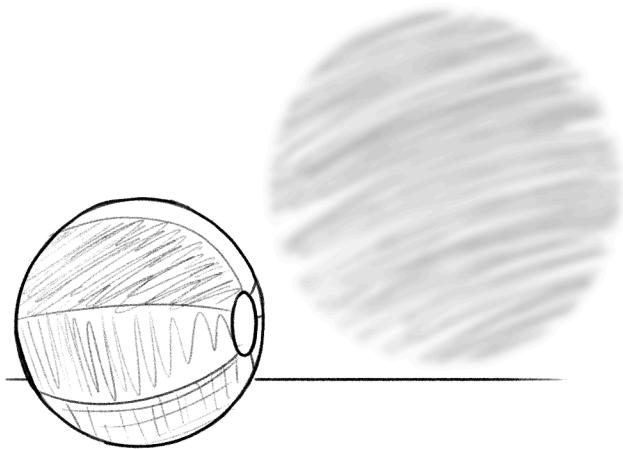
Think of a projection like the shadow of a ball. If you shine a light on it, it casts a shadow onto the wall. The ball itself is a sphere that has three dimensions, but the ball's shadow on the wall is a circle that has only two dimensions. The shadow still *resembles* the ball, but it's missing its depth.¹

Similarly, when we create a type projection, it's kind of like we're removing some of the "depth" of that object by limiting the types of its function inputs and outputs. For example, in the `getSnackFrom()` function above,

we created a type projection from

`VendingMachine<Snack>`, called `VendingMachine<out Snack>`,

which looks a lot like the original, except that the `refund()` function no longer accepts a `Snack`. Instead, it accepts a type called `Nothing`!



Original Generic Class

```
class VendingMachine<T : Product>(...) {
    fun purchase(money: Money): T = ...
    fun refund(product: T): Money = ...
}
```

Effective Parameterized Type

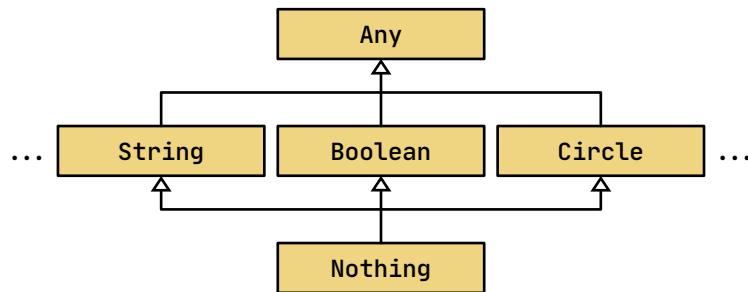
```
class VendingMachine<Snack>(...) {
    fun purchase(money: Money): Snack = ...
    fun refund(product: Snack): Money = ...
}
```

Effective Out-Projected Type

```
class VendingMachine<out Snack>(...) {
    fun purchase(money: Money): Snack = ...
    fun refund(product: Nothing): Money = ...
}
```

What exactly is `Nothing`? As mentioned in Chapter 14, every type in Kotlin is a *subtype* of a class named `Any`. Similarly, every type in Kotlin is also a *supertype* of a class called `Nothing`.

¹The word "projection" in the term "type projection" is technically rooted in mathematics, but the idea is the same.



The `Nothing` class can never be instantiated, because it only has a `private` constructor. Since it's impossible to create an instance of `Nothing`, we'll never be able to call `refund()` here in this function.

By the way: Nothing

The `Nothing` type sounds useless, but besides being helpful for type projections, you can also use it as a return type for a function that will never return - that is, a function that will always throw an exception.

It's also interesting to note that in Kotlin, `null` has a type, and that type is `Nothing?`.

So, an out-projection is created by adding the `out` modifier to a type argument. It looks similar to the original parameterized type, but everywhere that the type argument appears in an in-position, it is replaced with the `Nothing` type.

In-Projections

As you can probably guess, we can also use an `in` modifier to create an **in-projection**, like in this function.

```
fun getRefundFrom(machine: VendingMachine<in CandyBar>): Coin {  
    return machine.refund(CandyBar())  
}
```

Listing 19.31 - A function with a parameter whose type is an in-projection.

As with the out-projection above, inside the body of this function, we end up with a projection of `VendingMachine<CandyBar>`. But this time, instead of affecting the functions' *parameter* types, it's the functions' *result* types that have been affected.

With an in-projection, everywhere that the type argument appears in an out-position, it is forced to `Any?`. It's entirely possible to call these functions and get a result. If we want to do anything useful with that result, though, we'll probably need to cast it back to its more specific type.

Original Generic Class

```
class VendingMachine<T : Product>(...) {
    fun purchase(money: Money): T = ...
    fun refund(product: T): Money = ...
}
```

Effective Parameterized Type

```
class VendingMachine<Snack>(...) {
    fun purchase(money: Money): Snack = ...
    fun refund(product: Snack): Money = ...
}
```

Effective In-Projected Type

```
class VendingMachine<in Snack>(...) {
    fun purchase(money: Money): Any? = ...
    fun refund(product: Snack): Money = ...
}
```

So, out-projections and in-projections are two kinds of type projections. Out-projections create covariance by forcing the type argument in in-positions to `Nothing`, and in-projections create contravariance by forcing the type argument in out-positions to `Any?`.

Sometimes we want a function to accept all instances of a generic type, regardless of their type arguments. For these cases, Kotlin includes one more kind of projection.

Star Projections

Remember Vinnie? Well, at the end of the month, he services all of his vending machines, performing basic maintenance in order to keep them finely-tuned and working well. This operation doesn't involve any `Money` or `Product` - he just needs to get in there and tighten a few bolts.

Here's an updated version of `VendingMachine` that includes a function called `tune()`. As you can see, this new function doesn't use a type parameter at all - neither as a function parameter nor as a return type.

```
class VendingMachine<T : Snack>(private val snack: T) {
    fun purchase(money: Coin): T = snack
    fun refund(snack: T): Coin = Dime()
    fun tune() = println("All tuned up!")
}
```

Listing 19.32 - Adding a function that does not use the type parameter.

All vending machines need to be tuned up. The type of the `Snack` that they return is completely irrelevant.

In cases like these, we might want a function to accept literally any kind of type created from `VendingMachine`. Kotlin makes this easy with a special kind of type projection called a **star-projection**. To create a star-projection, rather than using a variance modifier, simply use an asterisk * (that is, a “star”) in place of the type argument. For example, here’s a function that will accept any kind of `VendingMachine`, regardless of its type argument.

```
fun service(machine: VendingMachine<*>) {  
    print("Tuning up $machine... ")  
    machine.tune()  
}
```

Listing 19.33 - A function with a parameter whose type is a star-projection.

This function can be called with any kind of `VendingMachine`, regardless of its type argument.

```
service(VendingMachine(CandyBar()) // Works with VendingMachine<CandyBar>  
service(VendingMachine(TrailMix()) // Works with VendingMachine<TrailMix>  
service(VendingMachine(GummyBears()) // Works with VendingMachine<GummyBears>
```

Listing 19.34 - Calling the `service()` function with different kinds of vending machines.

A star-projection looks like the original parameterized type, but:

- Anywhere that the type parameter was used in an in-position, it’ll be replaced with the `Nothing` type.
- Anywhere that the type parameter was used in an out-position, it’ll be replaced with the type parameter’s upper bound. Remember - if no upper bound was specified, it’ll be `Any?` by default.

<p>Original Generic Class</p> <pre>class VendingMachine<T : Product>(...) { fun purchase(money: Money): T = ... fun refund(product: T): Money = ... }</pre>	<p>Effective Parameterized Type</p> <pre>class VendingMachine<Snack>(...) { fun purchase(money: Money): Snack = ... fun refund(product: Snack): Money = ... }</pre>	<p>Effective Star-Projected Type</p> <pre>class VendingMachine<*>(...) { fun purchase(money: Money): Product = ... fun refund(product: Nothing): Money = ... }</pre>
--	--	---

In summary, star-projections are useful when you want to accept any instance of a generic type, regardless of its type arguments.

Variance in the Standard Library

Now that we've learned all about covariance, contravariance, variance modifiers, and type projections, we can better understand why certain types in the standard library work the way that they do. We opened this chapter by presenting the fact that `MutableList<Cow>` is a subtype of `List<Cow>`, but it's not a subtype of `MutableList<FarmAnimal>`. Why is this?

- `MutableList<Cow>` is a subtype of `List<Cow>` because `MutableList` extends the `List` interface. That's just regular interface inheritance.
- `MutableList<Cow>` is *not* a subtype of `MutableList<FarmAnimal>` because it allows you to both read *and* modify its elements, which means its type parameter appears in both the in-position *and* out-position. As a result, it can't have a variance modifier on it. This is similar to our `VendingMachine` in Listing 19.26.

As we've seen, the collection types in Kotlin usually come in two flavors - a read-only type (e.g., `List`) and a mutable type (e.g., `MutableList`). The read-only types will have the `out` modifier on their type parameters, so `List<Cow>` will be a subtype of `List<FarmAnimal>`. However, the mutable types *won't* have any variance modifiers, so `MutableList<Cow>` will *not* be a subtype of `MutableList<FarmAnimal>`. As you now know, though, you can use a type projection to work around this when it makes sense!

Summary

As Parker sat on a bench, watching the families running around in the park, he thought about the differences in the vending machines that Vinnie had installed. Now that the two of them understood which vending machines would satisfy the contract, they both felt much more comfortable about any replacements they might need to make in the future. Vinnie walked up to the vending machine and inserted a dime. He turned to Parker, who was still sitting on the bench. "Want a snack?" he asked him.

Congratulations on working your way through this chapter! Here's a recap of what we learned:

- How covariance describes the relationship between a type and its function return types.
- How contravariance describes the relationship between a type and its function parameter types.
- How we can use variance modifiers on type parameters to create declaration-site variance.
- How we can use variance modifiers on type arguments to create use-site variance.
- How collection types in the standard library use variance modifiers.

Keep playing with these concepts in your Kotlin projects to help solidify your understanding! In the next chapter, we'll introduce the very exciting topic of *coroutines*! See you then!

More is on the Way!

Hey, thanks again for supporting me as I'm finishing up the last few parts of this book! There's **at least one more chapter on the way**, including a topic I can't wait to cover...

Coroutines!

Meanwhile, you can keep up with the rest of my antics here:

- [YouTube](#)
- [Twitter/X](#)
- [LinkedIn](#)
- [Instagram](#)



Index

A

absent 81
abstract class 238
abstract function 249
abstract property 249
actual type 326
anonymous function 110
Any 255
application 288
argument 28–29
arithmetic operator 49
assignment 13
assignment compatibility 327
association 141

B

boilerplate 228
Boolean (type) 19
branch 40

C

call stack 300
cast 212

chain 134
class 53
 abstract 238
 data 257
 open 238
 override 247
class body 59
class delegation 221
closure 114
code block 35
collection 117
 filter operation 133
 map operation 130
 operation 129
 sort operation 132
comment 54
comparison operator 42
compile time 82
condition 42
conditional 41
conjunction operator 65

constructor parameter 57

context object 187

D

data class 257

copy() 268

inheritance 276

declaration 13

declared type 326

default argument 31

delegate 225

delegation 223

override 231

destructuring 270

component 272

destructuring assignment 271

disjunction operator 65

dot notation 163

duplication 25

E

element 120

elvis operator 89

enum class 70

entry 70

vs sealed type 291

enum constant 70

equals() 259

evaluation 16

exception parameter 307

execution path 41

exhaustive 46, 285

expression 16

extend 243

extension 161

extension function 169

extension properties 172

F

final 242

function 26

block body 34

expression body 34

generic 334

infix 142

function body 26

function literal 109

function reference 98, 103

function type 102

G

generic 321

generic function 334

generic type 330

H

hashCode() 263

hierarchy 255

depth 255

higher-order function 108

intermediate 135

literal 17

loop 127

I

identifier 13

if 47

immutable list 122

implement 208

implementation 246

implicit it parameter 110

index 125

indexed access operator 127

input 26

instance 57

instantiate 57

integer (type) 19

interface 200, 246

default implementation 218

inheritance 216, 245

iteration 127

M

map 139

entry 143

immutable 148

key 143

value 143

member function 60

method 60

minus operator 123

mutable list 124

mutate 122

N

named argument 31

non-nullable type 82

not-null assertion operator 91

null 77

nullable type 82

K

keyword 13

O

object 56

open class 238

open function 250

open property 250

lambda 98

library 288

list 117

operator function 275

optional 82

output 26

overload 204

P

parameter 26, 29

parameterized type 330

pass 28

plus operator 122

positional argument 31

present 81

problem domain 273

property 55

protected 248

R

receiver 161

 explicit 168

 implicit 168

 nullable 171

receiver parameter 169

receiver type 169

reference equality 259

required 82

result 26

return type 26

runtime 82

runtime type 326

S

safe-call operator 93

scope 174

 declaration 180

 nested 177

 statement 179

scope function 174

 also() 191

 apply() 192

 let() 190

 run() 188

 with() 187

sealed type 279

 restrictions 290

 sealed class 287

 sealed interface 286

 vs enum class 291

set 136

shadowing 194

short-circuiting 65

smart cast 88–89

stack 300

stack frame 300

stack trace 303

statement 16

static typing 20

string (type) 19

subclass 244

substitution 210

subtype 209

superclass 244

supertype 209

val (keyword) 14

value 13

value equality 260

variable 12–13

assigning 13

declaring 13

read-only 14

T

technical domain 273

`toString()` 265

trailing lambda syntax 112

try expression 306

type 18

actual 326

declared 326

generic 330

parameterized 330

runtime 326

Unit 36

type argument 329

type erasure 337

type inference 18

type parameter 328

type parameter constraint 331

W

when 41

subject 45

U

upper bound 332

V