O'REILLY®

Head First Kotlin

Grayscale Edition

For Sale in the Indian Subcontinent & Select Countries Only*

*Refer Back Cover

A Brain-Friendly Guide

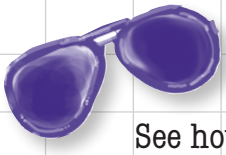
A learner's guide to Kotlin programming

Fool around in the Kotlin Standard Library

Avoid embarrassing lambda mistakes

Uncover the ins and outs of generics

Write out-of-this-world higher-order functions



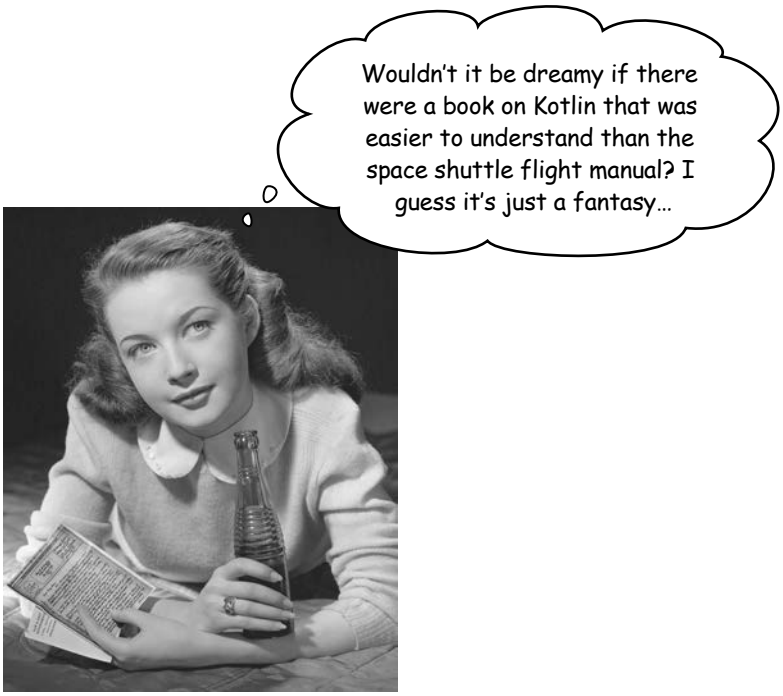
See how Elvis can change your life

Put collections under the microscope

Dawn Griffiths & David Griffiths

SPD®

Head First
Kotlin



Dawn Griffiths
David Griffiths

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

SPD®

SHROFF PUBLISHERS & DISTRIBUTORS PVT. LTD.
Mumbai Bangalore Kolkata New Delhi

Head First Kotlin

by Dawn Griffiths and David Griffiths

 ISBN: 978-1-491-99669-0
Originally printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles *(safari.oreilly.com)*. For more information, contact our corporate/institutional sales department: (800)998-9938 or *corporate@oreilly.com*.

Series Creators: Kathy Sierra, Bert Bates
Editor: Jeff Bleiel
Cover Designer: Randy Comer
Production Editor: Kristen Brown
Production Services: Jasmine Kwityn
Indexer: Lucie Haskins
Brain image on spine: Eric Freeman
Page Viewers: Mum and Dad, Laura and Aisha

Printing History:
February 2019: First Edition.

First Indian Reprint: March 2019

ISBN: 978-93-5213-807-4

Published by **Shroff Publishers & Distributors Pvt. Ltd.** B-103, Railway Commercial Complex, Sector 3, Sanpada (E), Navi Mumbai 400705 • TEL: (91 22) 4158 4158 • FAX: (91 22) 4158 4141 E-mail:spdorders@shroffpublishers.com•Web:www.shroff-publishers.com Printed at Jasmine Art Printers Pvt. Ltd., Navi Mumbai.

To the brains behind Kotlin for creating
such a great programming language.

Authors of Head First Kotlin

Dawn Griffiths has over 20 years experience working in the IT industry, working as a senior developer and senior software architect. She has written various books in the *Head First* series, including *Head First Android Development*. She also developed the animated video course *The Agile Sketchpad* with her husband, David, as a way of teaching key concepts and techniques in a way that keeps your brain active and engaged.

When Dawn's not writing books or creating videos, you'll find her honing her Tai Chi skills, reading, running, making bobbin lace, or cooking. She particularly enjoys spending time with her wonderful husband, David.

David Griffiths has worked as an Agile coach, a developer and a garage attendant, but not in that order. He began programming at age 12 when he saw a documentary on the work of Seymour Papert, and when he was 15, he wrote an implementation of Papert's computer language LOGO. Before writing *Head First Kotlin*, David wrote various other *Head First* books, including *Head First Android Development*, and created *The Agile Sketchpad* video course with Dawn.

When David's not writing, coding, or coaching, he spends much of his spare time traveling with his lovely wife—and coauthor—Dawn.

You can follow Dawn and David on Twitter at *https://twitter.com/HeadFirstKotlin*.

Table of Contents (Summary)

Table of Contents (the real thing)

Intro

Your brain on Kotlin. Here *you* are trying to *learn* something, while here your *brain* is, doing you a favor by making sure the learning doesn't *stick*. Your brain's thinking, "Better leave room for more important things, like which wild animals to avoid and whether naked snowboarding is a bad idea." So how *do* you trick your brain into thinking that your life depends on knowing how to code in Kotlin?

getting started

1 A Quick Dip

Kotlin is making waves.

From its first release, Kotlin has impressed programmers with its ***friendly syntax, conciseness, flexibility and power***. In this book, we'll teach you how to **build your own Kotlin applications**, and we'll start by getting you to build a basic application and run it. Along the way, you'll be introduced to some of Kotlin's basic syntax, such as *statements*, *loops* and *conditional branching*. Your journey has just begun...

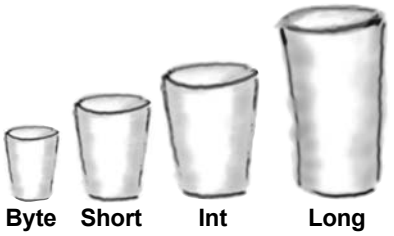
Being able to choose which platform to compile your code against means that Kotlin code can run on servers, in the cloud, in browsers, on mobile devices, and more.

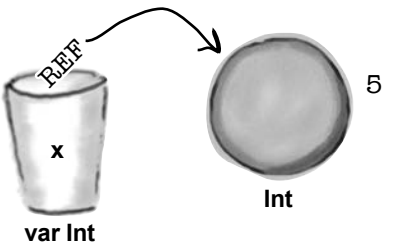
basic types and variables

2 Being a Variable

There's one thing all code depends on—variables.

So in this chapter, we're going to look under the hood, and show you ***how Kotlin variables really work***. You'll discover Kotlin's **basic types**, such as *Ints*, *Floats* and *Booleans*, and learn how the Kotlin compiler can **cleverly infer a variable's type from the value it's given**. You'll find out how to use **String templates** to construct complex Strings with very little code, and you'll learn how to create **arrays** to hold multiple values. Finally, you'll discover *why objects are so important to life in Kotlinville*.



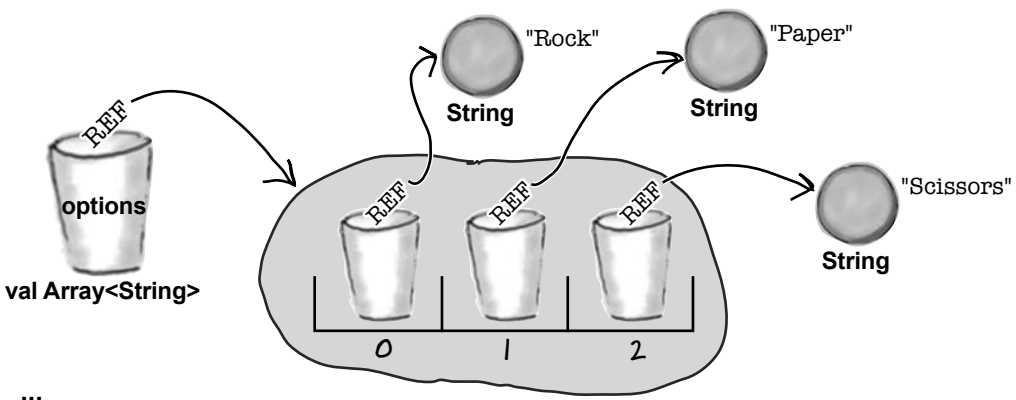


functions

3 Getting Out of Main

It's time to take it up a notch, and learn about functions.

So far, all the code you've written has been inside your application's *main* function. But if you want to write code that's **better organized** and **easier to maintain**, you need to know ***how to split your code into separate functions***. In this chapter, you'll learn *how to write functions* and *interact* with your application by building a game. You'll discover how to write compact ***single expression functions***. Along the way you'll find out how to ***iterate through ranges and collections*** using the powerful *for* loop.



classes and objects

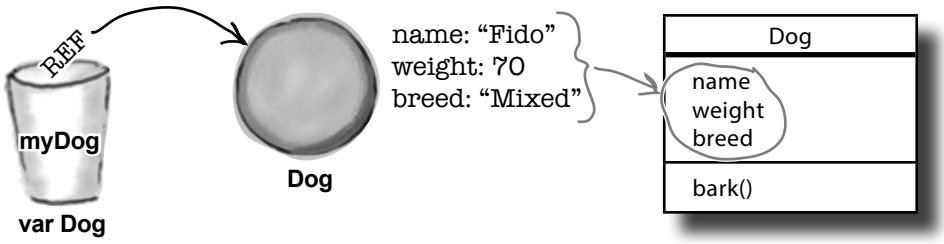
4 A Bit of Class

It's time we looked beyond Kotlin's basic types.

Sooner or later, you're going to want to use something *more* than Kotlin's basic types. And that's where ***classes*** come in. Classes are *templates* that allow you to **create your own types of objects**, and define their properties and functions. Here, you'll learn **how to design and define classes**, and how to use them to **create new types of objects**. You'll meet ***constructors***, ***initializer blocks***, ***getters*** and ***setters***, and you'll discover how they can be used to protect your properties. Finally, you'll learn how **data hiding is built into *all* Kotlin code**, saving you time, effort and a multitude of keystrokes.

One class

Many objects

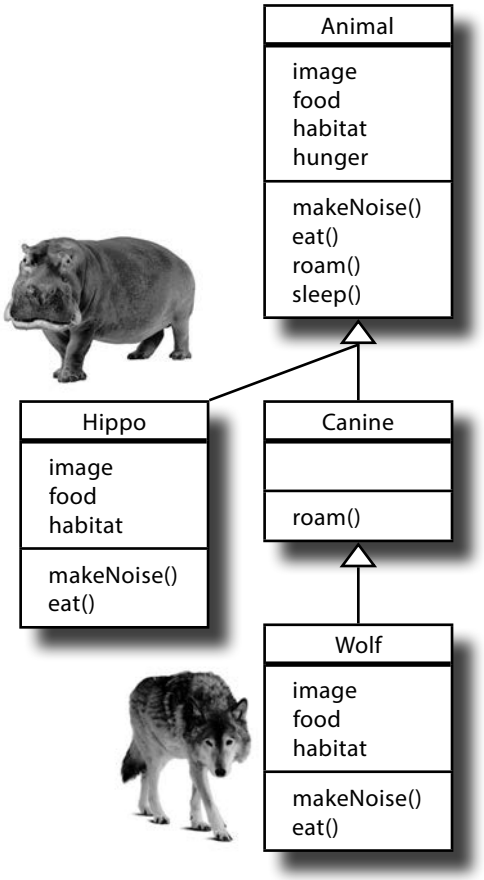


subclasses and superclasses

5 Using Your Inheritance

Ever found yourself thinking that an object's type would be perfect if you could just change a few things?

Well, that's one of the advantages of **inheritance**. Here, you'll learn how to create **subclasses**, and inherit the properties and functions of a **superclass**. You'll discover ***how to override functions and properties*** to make your classes behave the way *you* want, and you'll find out when this is (and isn't) appropriate. Finally, you'll see how inheritance helps you **avoid duplicate code**, and how to improve your flexibility with **polymorphism**.

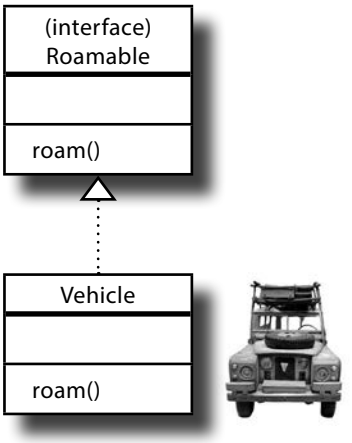


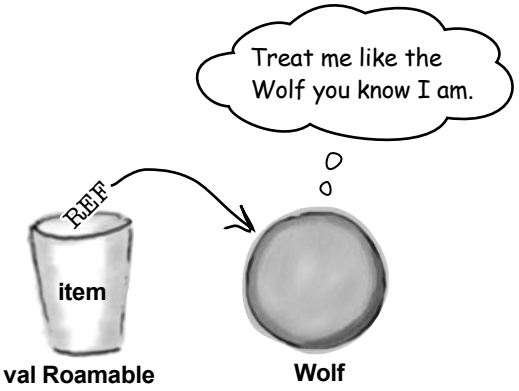
abstract classes and interfaces

6 Serious Polymorphism

A superclass inheritance hierarchy is just the beginning.

If you want to ***fully exploit polymorphism***, you need to design using **abstract classes** and **interfaces**. In this chapter, you'll discover how to use abstract classes to control which classes in your hierarchy ***can and can't be instantiated***. You'll see how they can force concrete subclasses to ***provide their own implementations***. You'll find out how to use interfaces to ***share behavior between independent classes***. And along the way, you'll learn the ins and outs of ***is***, ***as***, and ***when***.



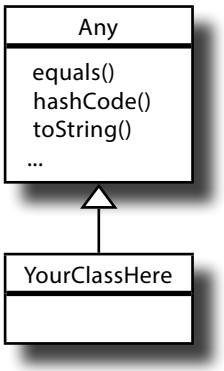


data classes

7 Dealing with Data

Nobody wants to spend their life reinventing the wheel.

Most applications include classes whose main purpose is to *store data*, so to make your coding life easier, the Kotlin developers came up with the concept of a **data class**. Here, you'll learn how data classes enable you to write code that's ***cleaner and more concise*** than you ever dreamed was possible. You'll explore the data class ***utility functions***, and discover how to ***destructure a data object into its component parts***. Along the way, you'll find out how ***default parameter values*** can make your code more flexible, and we'll introduce you to **Any**, the *mother of all superclasses*.



Data objects are considered equal if their properties hold the same values.

nulls and exceptions

8 Safe and Sound

Everybody wants to write code that's safe.

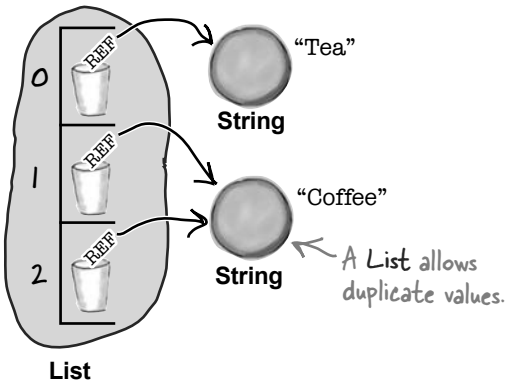
And the great news is that Kotlin was designed with *code-safety at its heart*. We'll start by showing you how Kotlin's use of **nullable types** means that you'll *hardly ever experience a NullPointerException during your entire stay in Kotlinville*. You'll discover how to make *safe calls*, and how Kotlin's **Elvis** operator stops you being *all shook up*. And when we're done with nulls, you'll find out how to **throw and catch exceptions** like a pro.

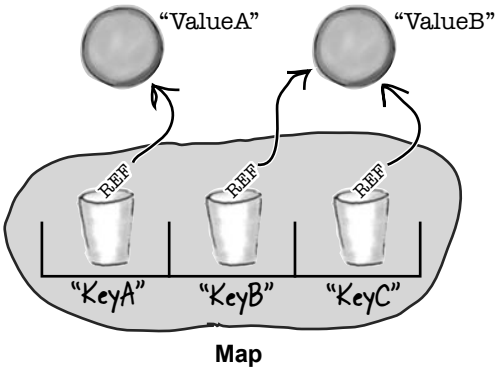
collections

9 Get Organized

Ever wanted something more flexible than an array?

Kotlin comes with a bunch of useful **collections** that give you more flexibility and greater control over how you ***store and manage groups of objects***. Want to keep a *resizeable list that you can keep adding to*? Want to *sort, shuffle or reverse its contents*? Want to *find something by name*? Or do you want something that will automatically *weed out duplicates* without you lifting a finger? If you want any of these things, or more, keep reading. It's all here...





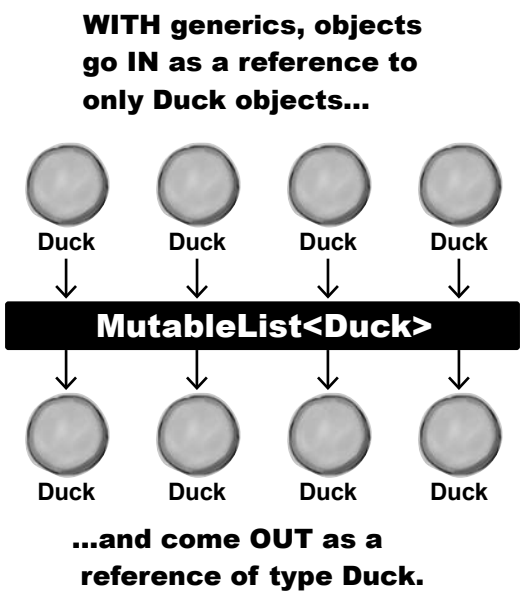
A Map allows duplicate values, but not duplicate keys.

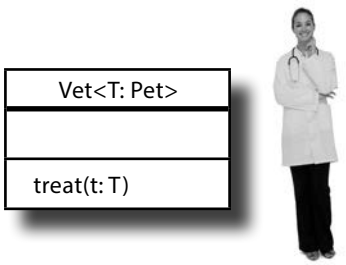
generics

10 Know Your Ins from Your Outs

Everybody likes code that's consistent.

And one way of writing consistent code that's less prone to problems is to use ***generics***. In this chapter, we'll look at how **Kotlin's collection classes use generics** to stop you from putting a Cabbage into a List<Seagull>. You'll discover when and how to ***write your own generic classes, interfaces and functions***, and how to **restrict a generic type** to a specific supertype. Finally, you'll find out ***how to use covariance and contravariance,*** putting **YOU** in control of your generic type's behavior.



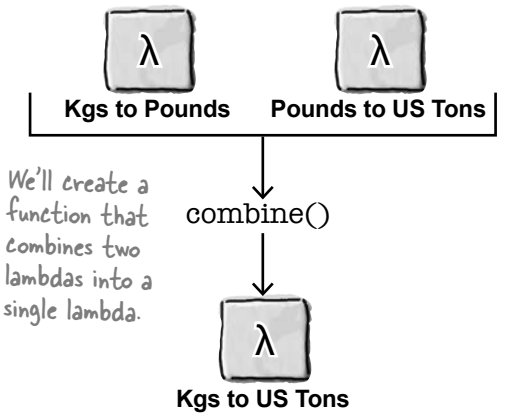


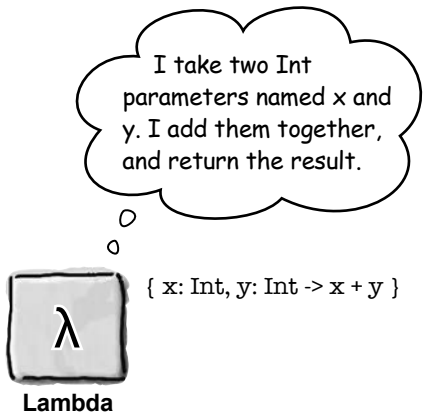
lambdas and higher-order functions

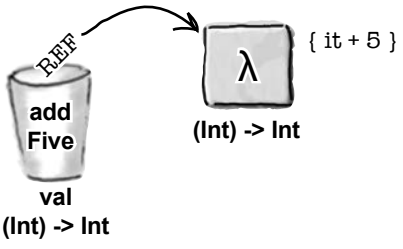
11 Treating Code Like Data

Want to write code that's even more powerful and flexible?

If so, then you need **lambdas**. A *lambda*—or *lambda expression*—is a block of code that you can pass around just like an object. Here, you'll discover ***how to define a lambda***, ***assign it to a variable***, and then ***execute its code***. You'll learn about **function types**, and how these can help you write **higher-order functions** that use lambdas for their parameter or return values. And along the way, you'll find out how a little ***syntactic sugar can make your coding life sweeter***.





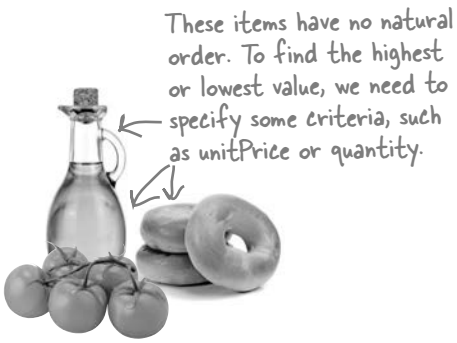


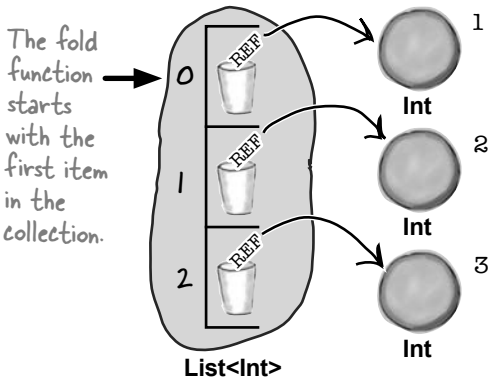
built-in higher-order functions

12 Power Up Your Code

Kotlin has an entire host of built-in higher-order functions.

And in this chapter, we'll introduce you to some of the most useful ones. You'll meet the flexible ***filter family***, and discover how they can help you trim your collection down to size. You'll learn how to ***transform a collection using map***, ***loop through its items with forEach***, and how to ***group the items in your collection using groupBy***. You'll even use ***fold*** to perform complex calculations *using just one line of code*. By the end of the chapter, you'll be able to write code more **powerful than you ever thought possible**.



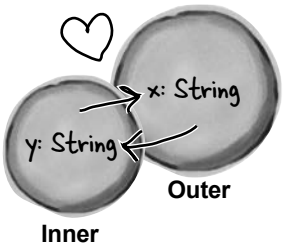


leftovers

The Top Ten Things (We Didn't Cover)

Even after all that, there's still a little more.

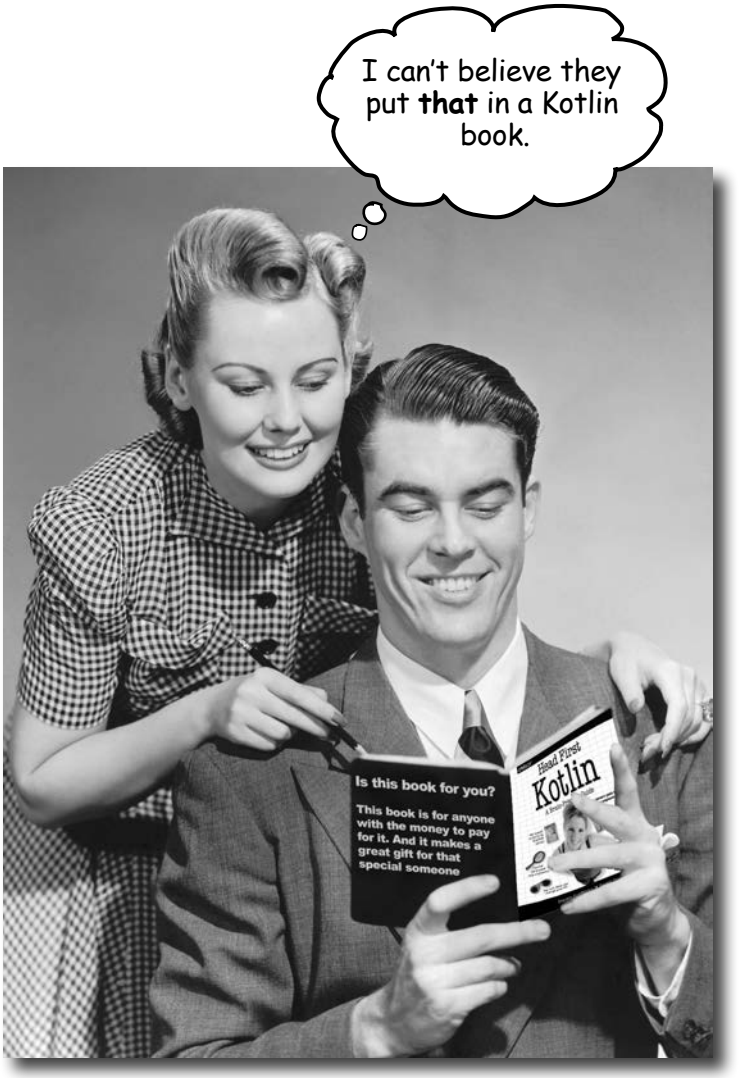
There are just a few more things we think you need to know. We wouldn't feel right about ignoring them, and we really wanted to give you a book you'd be able to lift without training at the local gym. Before you put down the book, **read through these tidbits**.



The Inner and Outer objects share a special bond. The Inner can use the Outer's variables, and vice versa.

how to use this book

Intro



In this section, we answer the burning question:
"So why DID they put that in a book on Kotlin?"

Who is this book for?

If you can answer "yes" to all of these:

1. Have you done some programming?
2. Do you want to learn Kotlin?
3. Do you prefer actually doing things and applying the stuff you learn over listening to someone in a lecture rattle on for hours on end?

this book is for you.

This is NOT a reference book. Head First Kotlin is a book designed for **learning**, not an encyclopedia of Kotlin facts.

Who should probably back away from this book?

If you can answer "yes" to any of these:

1. Is your programming background limited to HTML only, with no scripting language experience?

   (If you've done anything with looping, or if/then logic, you'll do fine with this book, but HTML tagging alone might not be enough.)

2. Are you a kick-butt Kotlin programmer looking for a *reference* book?

3. Would you rather have your toenails pulled out by 15 screaming monkeys than learn something new? Do you believe a Kotlin book should cover *everything*, especially all the obscure stuff you'll never use, and if it bores the reader to tears in the process, then so much the better?

this book is ***not*** for you.

[Note from Marketing: this book is for anyone with a credit card or a PayPal account]

We know what you're thinking

"How can *this* be a serious Kotlin book?"

"What's with all the graphics?"

"Can I actually *learn* it this way?"

"Do I smell pizza?"

We know what your *brain* is thinking

Your brain craves novelty. It's always searching, scanning, *waiting* for something unusual. It was built that way, and it helps you stay alive.

So what does your brain do with all the routine, ordinary, normal things you encounter? Everything it *can* to stop them from interfering with the brain's *real* job—recording things that *matter*. It doesn't bother saving the boring things; they never make it past the "this is obviously not important" filter.

How does your brain *know* what's important? Suppose you're out for a day hike and a tiger jumps in front of you—what happens inside your head and body?

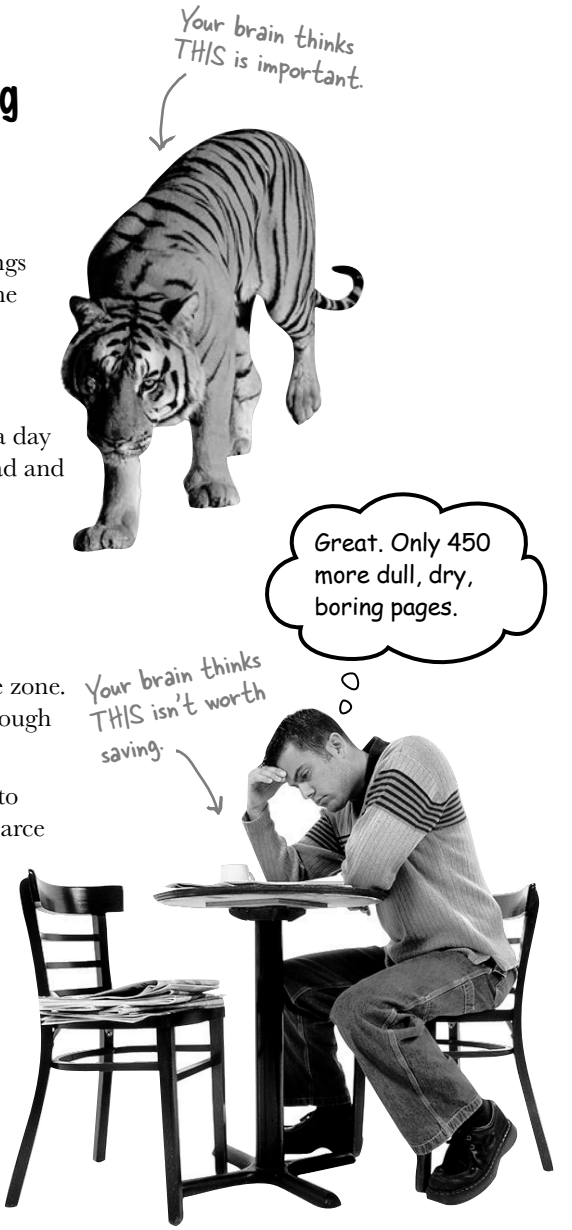
Neurons fire. Emotions crank up. *Chemicals surge*.

And that's how your brain knows…

This must be important! Don't forget it!

But imagine you're at home or in a library. It's a safe, warm, tiger-free zone. You're studying. Getting ready for an exam. Or trying to learn some tough technical topic your boss thinks will take a week, ten days at the most.

Just one problem. Your brain's trying to do you a big favor. It's trying to make sure that this *obviously* unimportant content doesn't clutter up scarce resources. Resources that are better spent storing the really *big* things. Like tigers. Like the danger of fire. Like how you should never have posted those party photos on your Facebook page. And there's no simple way to tell your brain, "Hey brain, thank you very much, but no matter how dull this book is, and how little I'm registering on the emotional Richter scale right now, I really *do* want you to keep this stuff around."



We think of a "Head First" reader as a learner.

So what does it take to *learn* something? First, you have to *get* it, then make sure you don't *forget* it. It's not about pushing facts into your head. Based on the latest research in cognitive science, neurobiology, and educational psychology, *learning* takes a lot more than text on a page. We know what turns your brain on.

Some of the Head First learning principles:

Make it visual. Images are far more memorable than words alone, and make learning much more effective (up to 89% improvement in recall and transfer studies). It also makes things more understandable. **Put the words within or near the graphics** they relate to, rather than on the bottom or on another page, and learners will be up to *twice* as likely to solve problems related to the content.

Use a conversational and personalized style. In recent studies, students performed up to 40% better on post-learning tests if the content spoke directly to the reader, using a first-person, conversational style rather than taking a formal tone. Tell stories instead of lecturing. Use casual language. Don't take yourself too seriously. Which would *you* pay more attention to: a stimulating dinner-party companion, or a lecture?

Get the learner to think more deeply. In other words, unless you actively flex your neurons, nothing much happens in your head. A reader has to be motivated, engaged, curious, and inspired to solve problems, draw conclusions, and generate new knowledge. And for that, you need challenges, exercises, and thought-provoking questions, and activities that involve both sides of the brain and multiple senses.

Get—and keep—the reader's attention. We've all had the "I really want to learn this, but I can't stay awake past page one" experience. Your brain pays attention to things that are out of the ordinary, interesting, strange, eye-catching, unexpected. Learning a new, tough, technical topic doesn't have to be boring. Your brain will learn much more quickly if it's not.

Touch their emotions. We now know that your ability to remember something is largely dependent on its emotional content. You remember what you care about. You remember when you *feel* something. No, we're not talking heart-wrenching stories about a boy and his dog. We're talking emotions like surprise, curiosity, fun, "what the…?", and the feeling of "I rule!" that comes when you solve a puzzle, learn something everybody else thinks is hard, or realize you know something that "I'm more technical than thou" Bob from Engineering *doesn't*.

Metacognition: thinking about thinking

If you really want to learn, and you want to learn more quickly and more deeply, pay attention to how you pay attention. Think about how you think. Learn how you learn.

Most of us did not take courses on metacognition or learning theory when we were growing up. We were *expected* to learn, but rarely *taught* to learn.

But we assume that if you're holding this book, you really want to learn how to code in Kotlin. And you probably don't want to spend a lot of time. If you want to use what you read in this book, you need to *remember* what you read. And for that, you've got to *understand* it. To get the most from this book, or *any* book or learning experience, take responsibility for your brain. Your brain on *this* content.

The trick is to get your brain to see the new material you're learning as Really Important. Crucial to your well-being. As important as a tiger. Otherwise, you're in for a constant battle, with your brain doing its best to keep the new content from sticking.

So just how *DO* you get your brain to treat Kotlin like it was a hungry tiger?

There's the slow, tedious way, or the faster, more effective way. The slow way is about sheer repetition. You obviously know that you *are* able to learn and remember even the dullest of topics if you keep pounding the same thing into your brain. With enough repetition, your brain says, "This doesn't *feel* important to him, but he keeps looking at the same thing *over* and *over* and *over*, so I suppose it must be."

The faster way is to do ***anything that increases brain activity,*** especially different *types* of brain activity. The things on the previous page are a big part of the solution, and they're all things that have been proven to help your brain work in your favor. For example, studies show that putting words *within* the pictures they describe (as opposed to somewhere else in the page, like a caption or in the body text) causes your brain to try to makes sense of how the words and picture relate, and this causes more neurons to fire. More neurons firing = more chances for your brain to *get* that this is something worth paying attention to, and possibly recording.

A conversational style helps because people tend to pay more attention when they perceive that they're in a conversation, since they're expected to follow along and hold up their end. The amazing thing is, your brain doesn't necessarily *care* that the "conversation" is between you and a book! On the other hand, if the writing style is formal and dry, your brain perceives it the same way you experience being lectured to while sitting in a roomful of passive attendees. No need to stay awake.

But pictures and conversational style are just the beginning…

Here's what WE did:

We used ***pictures***, because your brain is tuned for visuals, not text. As far as your brain's concerned, a picture really *is* worth a thousand words. And when text and pictures work together, we embedded the text *in* the pictures because your brain works more effectively when the text is *within* the thing it refers to, as opposed to in a caption or buried in the body text somewhere.

We used ***redundancy***, saying the same thing in *different* ways and with different media types, and *multiple senses*, to increase the chance that the content gets coded into more than one area of your brain.

We used concepts and pictures in ***unexpected*** ways because your brain is tuned for novelty, and we used pictures and ideas with at least *some* ***emotional*** *content*, because your brain is tuned to pay attention to the biochemistry of emotions. That which causes you to *feel* something is more likely to be remembered, even if that feeling is nothing more than a little ***humor***, ***surprise***, or ***interest.***

We used a personalized, ***conversational style***, because your brain is tuned to pay more attention when it believes you're in a conversation than if it thinks you're passively listening to a presentation. Your brain does this even when you're *reading*.

We included ***activities***, because your brain is tuned to learn and remember more when you ***do*** things than when you *read* about things. And we made the exercises challenging-yet-doable, because that's what most people prefer.

We used ***multiple learning styles***, because *you* might prefer step-by-step procedures, while someone else wants to understand the big picture first, and someone else just wants to see an example. But regardless of your own learning preference, *everyone* benefits from seeing the same content represented in multiple ways.

We include content for ***both sides of your brain***, because the more of your brain you engage, the more likely you are to learn and remember, and the longer you can stay focused. Since working one side of the brain often means giving the other side a chance to rest, you can be more productive at learning for a longer period of time.

And we included ***stories*** and exercises that present ***more than one point of view,*** because your brain is tuned to learn more deeply when it's forced to make evaluations and judgments.

We included ***challenges***, with exercises, and by asking ***questions*** that don't always have a straight answer, because your brain is tuned to learn and remember when it has to *work* at something. Think about it—you can't get your *body* in shape just by *watching* people at the gym. But we did our best to make sure that when you're working hard, it's on the *right* things. That ***you're not spending one extra dendrite*** processing a hard-to-understand example, or parsing difficult, jargon-laden, or overly terse text.

We used ***people***. In stories, examples, pictures, etc., because, well, *you're* a person. And your brain pays more attention to *people* than it does to *things*.

Cut this out and stick it on your refrigerator.

Here's what YOU can do to bend your brain into submission

So, we did our part. The rest is up to you. These tips are a starting point; listen to your brain and figure out what works for you and what doesn't. Try new things.

1 Slow down. The more you understand, the less you have to memorize.

Don't just *read*. Stop and think. When the book asks you a question, don't just skip to the answer. Imagine that someone really *is* asking the question. The more deeply you force your brain to think, the better chance you have of learning and remembering.

2 Do the exercises. Write your own notes.

We put them in, but if we did them for you, that would be like having someone else do your workouts for you. And don't just *look* at the exercises. **Use a pencil.** There's plenty of evidence that physical activity *while* learning can increase the learning.

3 Read "There Are No Dumb Questions."

That means all of them. They're not optional sidebars, ***they're part of the core content!*** Don't skip them.

4 Make this the last thing you read before bed. Or at least the last challenging thing.

Part of the learning (especially the transfer to long-term memory) happens *after* you put the book down. Your brain needs time on its own, to do more processing. If you put in something new during that processing time, some of what you just learned will be lost.

5 Talk about it. Out loud.

Speaking activates a different part of the brain. If you're trying to understand something, or increase your chance of remembering it later, say it out loud. Better still, try to explain it out loud to someone else. You'll learn more quickly, and you might uncover ideas you hadn't known were there when you were reading about it.

6 Drink water. Lots of it.

Your brain works best in a nice bath of fluid. Dehydration (which can happen before you ever feel thirsty) decreases cognitive function.

7 Listen to your brain.

Pay attention to whether your brain is getting overloaded. If you find yourself starting to skim the surface or forget what you just read, it's time for a break. Once you go past a certain point, you won't learn faster by trying to shove more in, and you might even hurt the process.

8 Feel something.

Your brain needs to know that this *matters*. Get involved with the stories. Make up your own captions for the photos. Groaning over a bad joke is *still* better than feeling nothing at all.

9 Write a lot of code!

There's only one way to learn Kotlin: **write a lot of code**. And that's what you're going to do throughout this book. Coding is a skill, and the only way to get good at it is to practice. We're going to give you a lot of practice: every chapter has exercises that pose a problem for you to solve. Don't just skip over them—a lot of the learning happens when you solve the exercises. We included a solution to each exercise—don't be afraid to **peek at the solution** if you get stuck! (It's easy to get snagged on something small.) But try to solve the problem before you look at the solution. And definitely get it working before you move on to the next part of the book.

Read me

This is a learning experience, not a reference book. We deliberately stripped out everything that might get in the way of learning whatever it is we're working on at that point in the book. And the first time through, you need to begin at the beginning, because the book makes assumptions about what you've already seen and learned.

We assume you're new to Kotlin, but not to programming.

We assume that you've already done some programming. Maybe not a lot, but we'll assume you've already seen things like loops and variables in some other language. And unlike a lot of other Kotlin books, we don't assume that you already know Java.

We begin by teaching some basic Kotlin concepts, and then we start putting Kotlin to work for you right away.

We cover the fundamentals of Kotlin code in Chapter 1. That way, by the time you make it all the way to Chapter 2, you are creating programs that actually do something. The rest of the book then builds on your Kotlin skills, turning you from *Kotlin newbie* to *Kotlin ninja master* in very little time.

The redundancy is intentional and important.

One distinct difference in a Head First book is that we want you to *really* get it. And we want you to finish the book remembering what you've learned. Most reference books don't have retention and recall as a goal, but this book is about *learning*, so you'll see some of the same concepts come up more than once.

The code examples are as lean as possible.

We know how frustrating it is to wade through 200 lines of code looking for the two lines you need to understand. Most examples within this book are shown within the smallest possible context, so that the part you're trying to learn is clear and simple. So don't expect the code to be robust, or even complete. That's *your* assignment for after you finish the book, and it's all part of the learning experience.

The exercises and activities are NOT optional.

The exercises and activities are not add-ons; they're part of the core content of the book. Some of them are to help with memory, some are for understanding, and some will help you apply what you've learned. So don't skip the exercises! Your brain will thank you for it.

The Brain Power exercises don't have answers.

Not printed in the book, anyway. For some of them, there *is* no right answer, and for others, part of the learning experience is for *you* to decide if and when your answers are right. In some of the Brain Power exercises, you will find hints to point you in the right direction.

The technical review team

Technical reviewers:

Ingo Krotzky is a trained health information technician who has been working as a database programmer/software developer for contract research institutes.

Ken Kousen is the author of the books *Modern Java Recipes* (O'Reilly), *Gradle Recipes for Android* (O'Reilly) and *Making Java Groovy* (Manning), as well as O'Reilly video courses in Android, Groovy, Gradle, advanced Java and Spring. He is a regular speaker on the No Fluff, Just Stuff conference tour and a 2013 and 2016 JavaOne Rock Star, and has spoken at conferences all over the world. Through his company, Kousen I.T., Inc., he has taught software development training courses to thousands of students.

Acknowledgments

Our editor:

Heartfelt thanks to our awesome editor **Jeff Bleiel** for all his work and help. We've truly valued his trust, support, and encouragement. We've appreciated all the times he pointed out when things were unclear or needed a rethink, as it's led to us writing a much better book.

The O'Reilly team:

A big thank you goes to **Brian Foster** for his early help in getting *Head First Kotlin* off the ground; **Susan Conant**, **Rachel Roumeliotis** and **Nancy Davis** for their help smoothing the wheels; **Randy Comer** for designing the cover; the **early release team** for making early versions of the book available for download; and **Kristen Brown, Jasmine Kwityn, Lucie Haskins and the rest of the production team** for expertly steering the book through the production process, and for working so hard behind the scenes.

Friends, family and colleagues:

Writing a *Head First* book is always a rollercoaster, and we've truly valued the kindness and support of our friends, family and colleagues along the way. Special thanks go to **Jacqui**, **Ian**, **Vanessa**, **Dawn**, **Matt**, **Andy**, **Simon**, **Mum**, **Dad**, **Rob** and **Lorraine**.

The without-whom list:

Our awesome technical review team worked hard to give us their thoughts on the book, and we're so grateful for their input. They made sure that what we covered was spot on, and kept us entertained along the way. We think the book is much better as a result of their feedback.

Finally, our thanks to **Kathy Sierra** and **Bert Bates** for creating this extraordinary series of books, and for letting us into their brains.

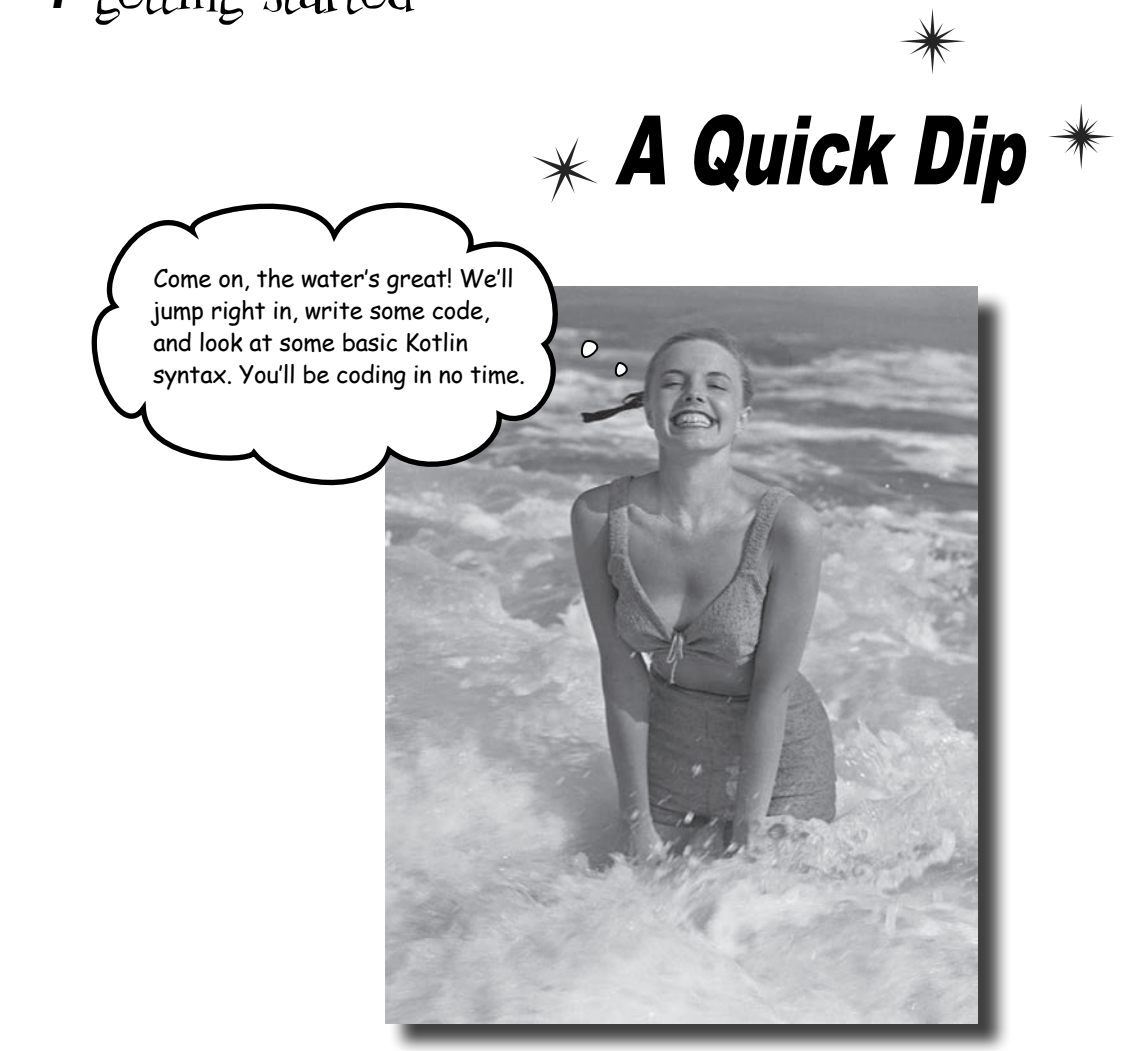
O'Reilly

For almost 40 years, O'Reilly Media has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers.

For more information, please visit *http://oreilly.com*.

1 getting started



A Quick Dip

Kotlin is making waves.

From its first release, Kotlin has impressed programmers with its ***friendly syntax, conciseness, flexibility and power***. In this book, we'll teach you how to **build your own Kotlin applications**, and we'll start by getting you to build a basic application and run it. Along the way, you'll be introduced to some of Kotlin's basic syntax, such as *statements*, *loops* and *conditional branching*. Your journey has just begun...

Welcome to Kotlinville

Kotlin has been taking the programming world by storm. Despite being one of the youngest programming languages in town, many developers now view it as their language of choice. So what makes Kotlin so special?

Kotlin has many modern language features that make it attractive to developers. You'll find out about these features in more detail later in the book, but for now, here are some of the highlights.

It's crisp, concise and readable

Unlike some languages, Kotlin code is very concise, and you can perform powerful tasks in just one line. It provides shortcuts for common actions so that you don't have to write lots of repetitive boilerplate code, and it has a rich library of functions that you can use. And as there's less code to wade through, it's quicker to read, write and understand, leaving you more time to do other things.

You can use object-oriented AND functional programming

Can't decide whether to learn object-oriented or functional programming? Well, why not do both? Kotlin lets you create object-oriented code that uses classes, inheritance and polymorphism, just as you can in Java. But it also supports functional programming, giving you the best of both worlds.

The compiler keeps you safe

Nobody likes unsafe, buggy code, and Kotlin's compiler puts a lot of effort into making sure your code is as clean as possible, preventing many of the errors that can occur in other programming languages. Kotlin is statically typed, for example, so you can't perform inappropriate actions on the wrong type of variable and crash your code. And most of the time, you don't even need to explicitly specify the type yourself as the compiler can infer it for you.

So Kotlin is a modern, powerful and flexible programming language that offers many advantages. But that's not the end of the story.

Kotlin virtually eliminates the kinds of errors that regularly occur in other programming languages. That means safer, more reliable code, and less time spent chasing bugs.

You can use Kotlin nearly everywhere

Kotlin is so powerful and flexible that you can use it as a general-purpose language in many different contexts. This is because you can ***choose which platform to compile your Kotlin code against***.

Java Virtual Machines (JVMs)

Kotlin code can be compiled to JVM (Java Virtual Machine) bytecode, so you can use Kotlin practically anywhere that you can use Java. Kotlin is 100% interoperable with Java, so you can use existing Java libraries with it. If you're working on an application that contains a lot of old Java code, you don't have to throw all the old code away; your new Kotlin code will work alongside it. And if you want to use the Kotlin code you've written from inside Java, you can do so with ease.

Being able to choose which platform to compile your code against means that Kotlin code can run on servers, in the cloud, in browsers, on mobile devices, and more.

Android

Alongside other languages such as Java, Kotlin has first-class support for Android. Kotlin is fully supported in Android Studio, and you can make the most of Kotlin's many advantages when you develop Android apps.

Client-side and server-side JavaScript

You can also transpile—or translate and compile—Kotlin code into JavaScript, so that you can run it in a browser. You can use it to work with both client-side and server-side technology, such as WebGL or Node.js.

Native apps

If you want to write code that will run quickly on less powerful devices, you can compile your Kotlin code directly to native machine code. This allows you to write code that will run, for example, on iOS or Linux.

In this book, we're going to focus on creating Kotlin applications for JVMs, as this is the most straightforward way of getting to grips with the language. Afterwards, you'll be able to apply the knowledge you've gained to other platforms.

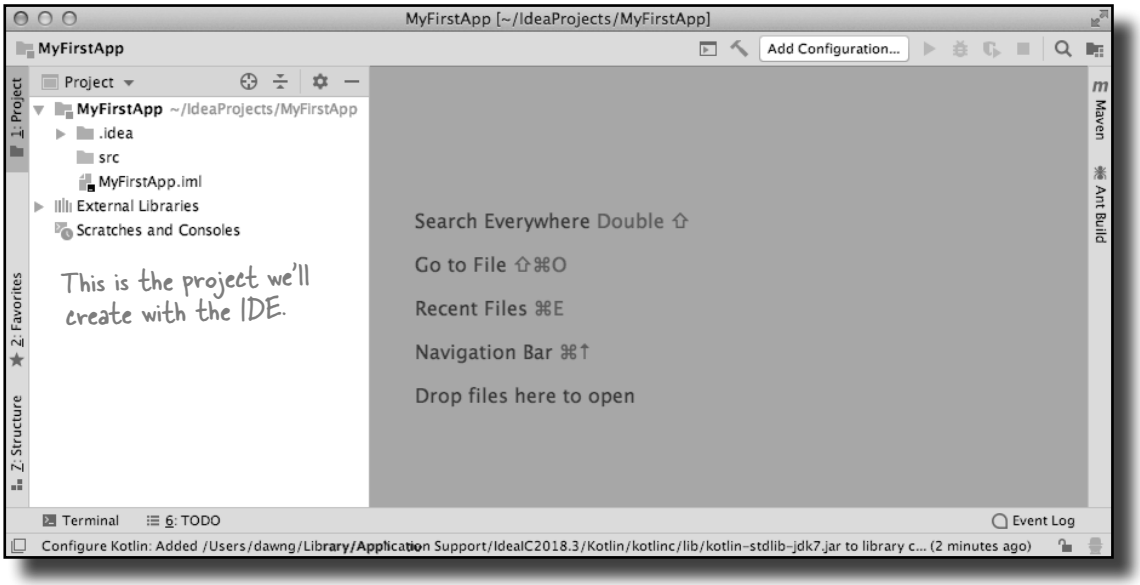
Let's dive in.

Even though we're building applications for Java Virtual Machines, you don't need to know Java to get the most out of this book. We're assuming you have some general programming experience, but that's it.

What we'll do in this chapter

In this chapter, we're going to show you how to build a basic Kotlin application. There are a number of steps we're going to go through to do this:

1. **Create a new Kotlin project.**
   We'll start by installing IntelliJ IDEA (Community Edition), a free IDE that supports Kotlin application development. We'll then use the IDE to build a new Kotlin project:



2. **Add a function that displays some text.**
   We'll add a new Kotlin file to the project, then write a simple `main` function that will output the text "Pow!"

3. **Update the function to make it do more.**
   Kotlin includes basic language structures such as statements, loops and conditional branching. We'll use these to change our function so that it does more.

4. **Try out code in the Kotlin interactive shell.**
   Finally, we'll look at how to try out snippets of code in the Kotlin interactive shell (or REPL).

We'll install the IDE after you've tried the following exercise.

Sharpen your pencil

We know we've not taught you any Kotlin code yet, but see if you can guess what each line of code is doing. We've completed the first one to get you started.

```
val name = "Misty"          Declare a variable named 'name' and give it a value of "Misty".
val height = 9

println("Hello")
println("My cat is called $name")
println("My cat is $height inches tall")

val a = 6
val b = 7
val c = a + b + 10
val str = c.toString()

val numList = arrayOf(1, 2, 3)
var x = 0
while (x < 3) {
    println("Item $x is ${numList[x]}")
    x = x + 1
}

val myCat = Cat(name, height)
val y = height - 3
if (y < 5) myCat.miaow(4)

while (y < 8) {
    myCat.play()
    y = y + 1
}
```

Sharpen your pencil Solution

We know we've not taught you any Kotlin code yet, but see if you can guess what each line of code is doing. We've completed the first one to get you started.

```
val name = "Misty"          Declare a variable named 'name' and give it a value of "Misty".
val height = 9              Declare a variable named 'height' and give it a value of 9.

println("Hello")            Prints "Hello" to the standard output.
println("My cat is called $name")      Prints "My cat is called Misty".
println("My cat is $height inches tall")   Prints "My cat is 9 inches tall".

val a = 6                   Declare a variable named 'a' and give it a value of 6.
val b = 7                   Declare a variable named 'b' and give it a value of 7.
val c = a + b + 10          Declare a variable named 'c' and give it a value of 23.
val str = c.toString()      Declare a variable named 'str' and give it a text value of "23".

val numList = arrayOf(1, 2, 3)   Create an array containing values of 1, 2 and 3.
var x = 0                   Declare a variable named 'x' and give it a value of 0.
while (x < 3) {             Keep looping as long as x is less than 3.
    println("Item $x is ${numList[x]}")   Print the index and value of each item in the array.
    x = x + 1               Add 1 to x.
}                           This is the end of the loop.

val myCat = Cat(name, height)   Declare a variable named 'myCat' and create a Cat object.
val y = height - 3          Declare a variable named 'y' and give it a value of 6.
if (y < 5) myCat.miaow(4)   If y is less than 5, the Cat should miaow 4 times.

while (y < 8) {             Keep looping as long as y is less than 8.
    myCat.play()            Make the Cat play.
    y = y + 1               Add 1 to y.
}                           This is the end of the loop.
```

You are here.

Build application
Add function
Update function
Use REPL

Install IntelliJ IDEA (Community Edition)

The easiest way of writing and running Kotlin code is to use IntelliJ IDEA (Community Edition). This is a free IDE from JetBrains, the people who invented Kotlin, and it comes with everything you need to develop Kotlin applications, including:

A code editor
The code editor offers code completion to help you write Kotlin code, and formatting and color highlighting to make your code easier to read. It also gives you hints for improving your code.

Build tools
You can compile and run your code using quick and easy shortcuts.

There are many more features too, all there to make your coding life easier.

Kotlin REPL
You have easy access to the Kotlin REPL, which lets you try out code snippets outside your main code.

Version control
IntelliJ IDEA interfaces with major version control systems such as Git, SVN, CVS and more

To follow along with us in this book, you need to install IntelliJ IDEA (Community Edition). You can download the IDE here:

https://www.jetbrains.com/idea/download/index.html ← Make sure you choose the option to download the free Community Edition of IntelliJ IDEA.

Once you've installed the IDE, open it. You should see the IntelliJ IDEA welcome screen. You're ready to build your first Kotlin application.

This is the IntelliJ IDEA welcome screen.

Build application
Add function
Update function
Use REPL

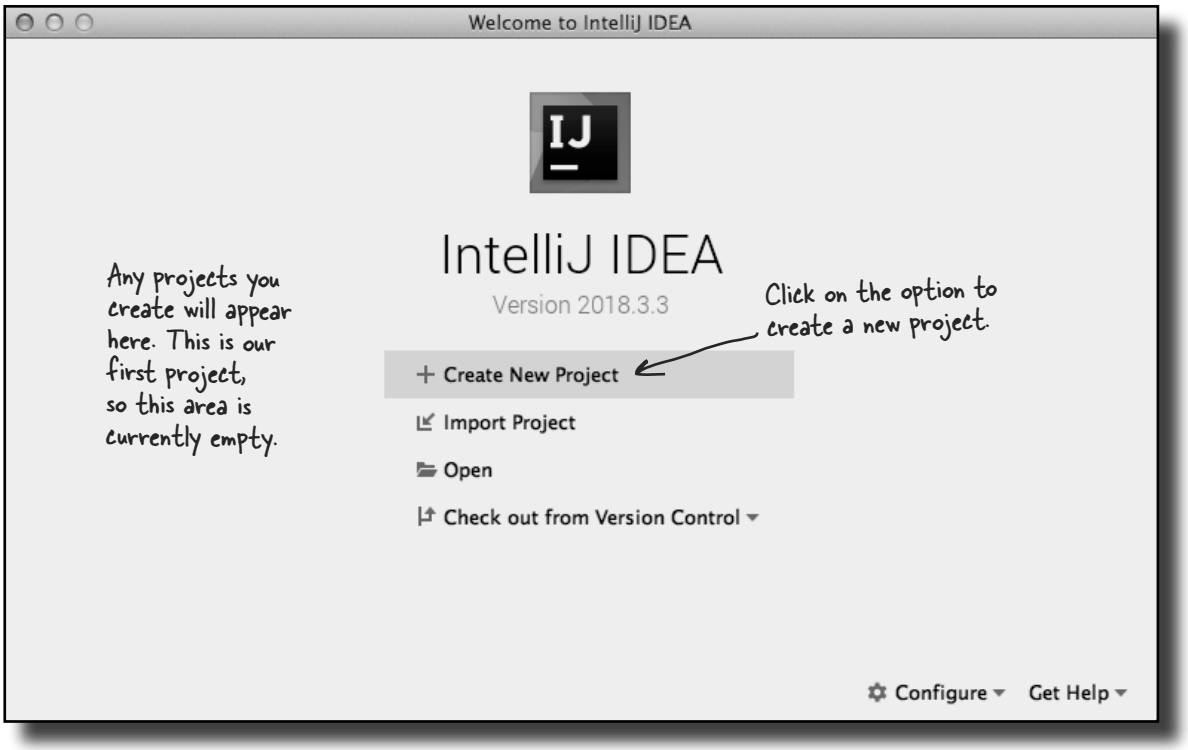
Let's build a basic application

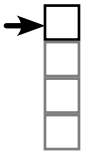
Now that you've set up your development environment, you're ready to create your first Kotlin application. We're going to create a very simple application that will display the text "Pow!" in the IDE.

Whenever you create a new application in IntelliJ IDEA, you need to create a new project for it. Make sure you have the IDE open, and follow along with us.

1. Create a new project

The IntelliJ IDEA welcome screen gives you a number of options for what you want to do. We want to create a new project, so click on the option for "Create New Project".





Build application
Add function
Update function
Use REPL

Building a basic application (continued)

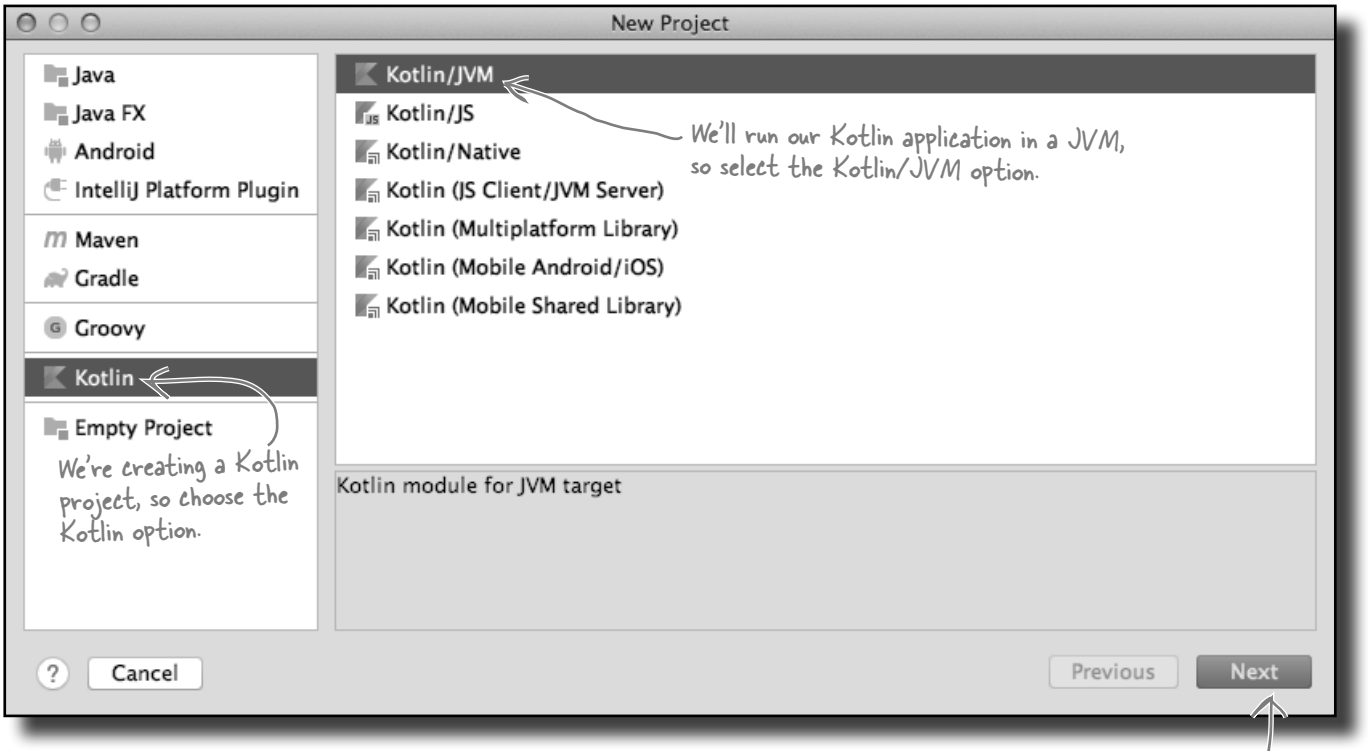
2. Specify the type of project

Next, you need to tell IntelliJ IDEA what sort of project you want to create.

IntelliJ IDEA allows you to create projects for various languages and platforms, such as Java and Android. We're going to create a Kotlin project, so choose the option for "Kotlin".

You also need to specify which platform you want your Kotlin project to target. We're going to create a Kotlin application with a JVM target, so select the Kotlin/JVM option. Then click on the Next button.

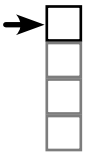
There are other options too, but we're going to focus on creating applications that run against a JVM.



We'll run our Kotlin application in a JVM, so select the Kotlin/JVM option.

We're creating a Kotlin project, so choose the Kotlin option.

Click on the Next button to proceed to the next step.



Build application
Add function
Update function
Use REPL

Building a basic application (continued)

3. Configure the project

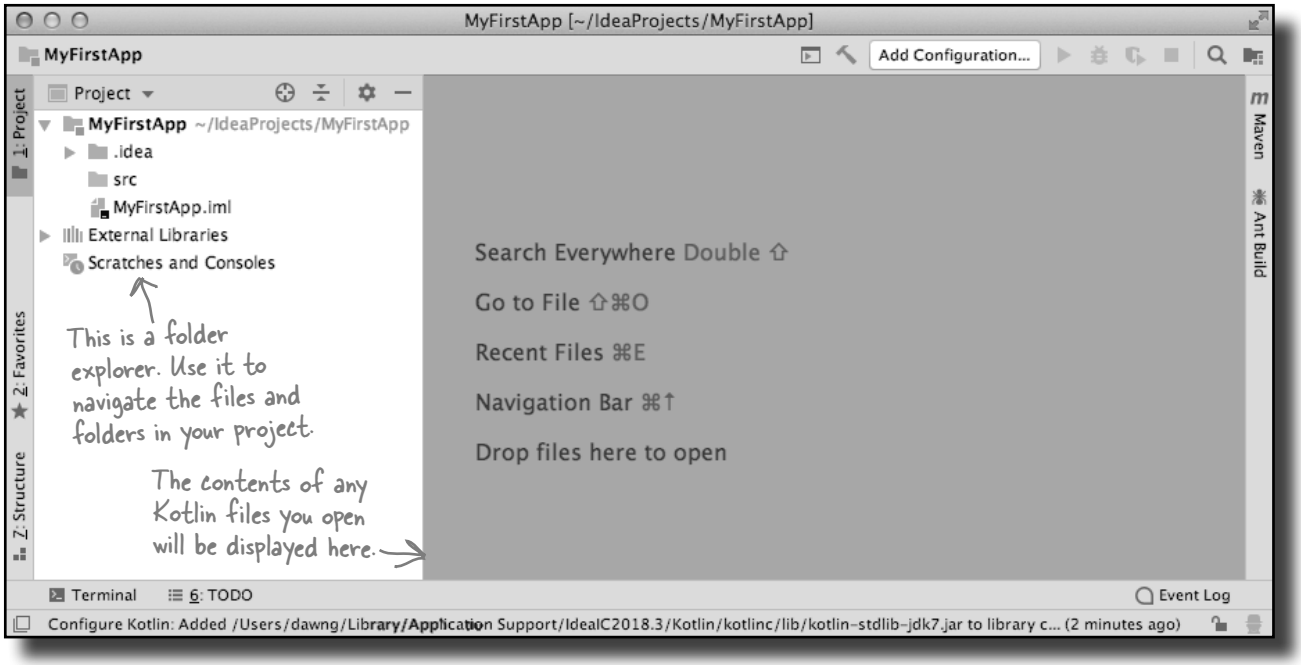
You now need to configure the project by saying what you want to call it, where you want to store the files, and what files should be used by the project. This includes which version of Java should be used by the JVM, and the library for the Kotlin runtime.

Name the project "MyFirstApp", and accept the rest of the defaults.

When you click on the Finish button, IntelliJ IDEA will create your project.

You've just created your first Kotlin project

After you've finished going through the steps to create a new project, IntelliJ IDEA sets up the project for you, then displays it. Here's the project that the IDE created for us:

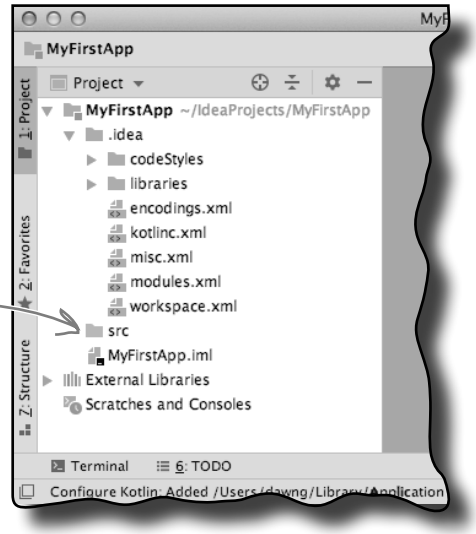


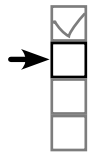
As you can see, the project features an explorer which you can use to navigate the files and folders that make up your project. IntelliJ IDEA creates this folder structure for you when you create the project.

The folder structure is comprised of configuration files that are used by the IDE, and some external libraries that your application will use. It also includes a *src* folder, which is used to hold your source code. You'll spend most of your time in Kotlinville working with the *src* folder.

The *src* folder is currently empty as we haven't added any Kotlin files yet. We'll do this next.

Any Kotlin source files you create need to be added to the src folder.



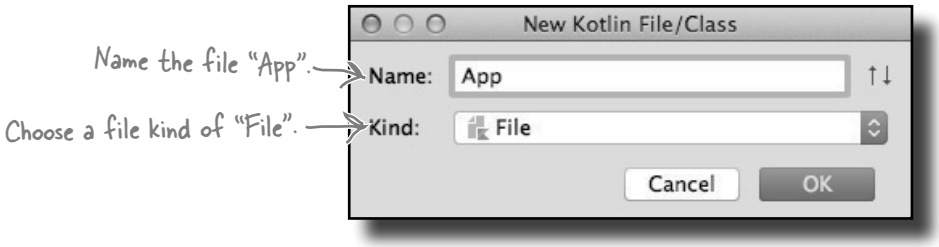


Build application
Add function
Update function
Use REPL

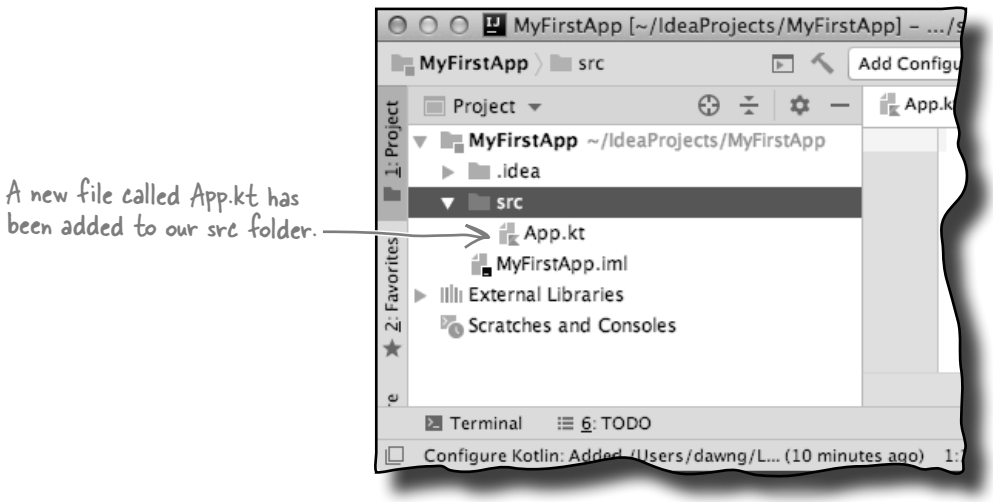
Add a new Kotlin file to the project

Before you can write any Kotlin code, you first need to create a Kotlin file to put it in.

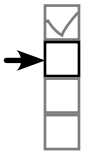
To add a new Kotlin file to your project, highlight the *src* folder in IntelliJ IDEA's explorer, then click on the File menu and choose New → Kotlin File/Class. You will prompted for the name and type of Kotlin file you want to create. Name the file "App", and choose File from the Kind option, like this:



When you click on the OK button, IntelliJ IDEA creates a new Kotlin file named *App.kt*, and adds it to the *src* folder in your project:



Next, let's look at the code we need to add to *App.kt* to get it to do something.



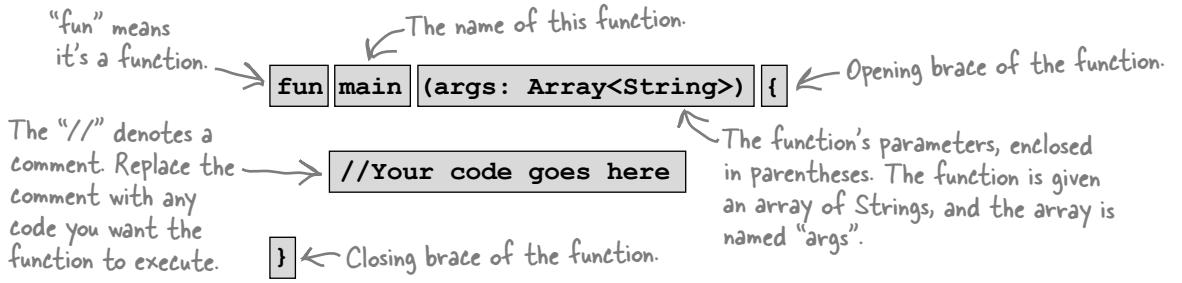
Build application
Add function
Update function
Use REPL

Anatomy of the main function

We're going to get our Kotlin code to display "Pow!" in the IDE's output window. We'll do this be adding a function to *App.kt*.

Whenever you write a Kotlin application, you *must* add a function to it called `main`, which starts your application. When you run your code, the JVM looks for this function, and executes it.

The `main` function looks like this:

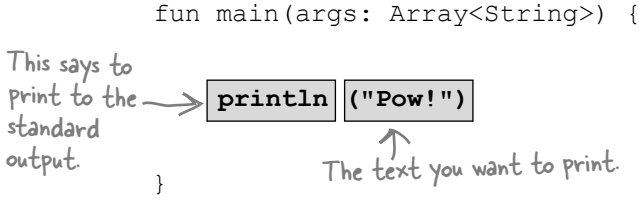


The function begins with the word **`fun`**, which is used to tell the Kotlin compiler that it's a function. You use the `fun` keyword for each new Kotlin function you create.

The `fun` keyword is followed by the name of the function, in this case **`main`**. Naming the function `main` means that it will be automatically executed when you run the application.

The code in the braces `()` after the function name tells the compiler what arguments (if any) the function takes. Here, the code `args: Array<String>` specifies that the function accepts an array of `String`s, and this array is named `args`.

You put any code you want to run between the curly braces `{}` of the `main` function. We want our code to print "Pow!" in the IDE, and we can do that using code like this:



`println("Pow!")` prints a string of characters, or `String`, to the standard output. As we're running our code in an IDE, it will print "Pow!" in the IDE's output pane.

Now that you've seen what the function looks like, let's add it to our project.

Parameterless main functions

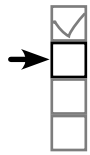
If you're using Kotlin 1.2, or an earlier version, your `main` function *must* take the following form in order for it to start your application:

```
fun main(args: Array<String>) {
    //Your code goes here
}
```

From Kotlin1.3, however, you can omit `main`'s parameters so that the function looks like this:

```
fun main() {
    //Your code goes here
}
```

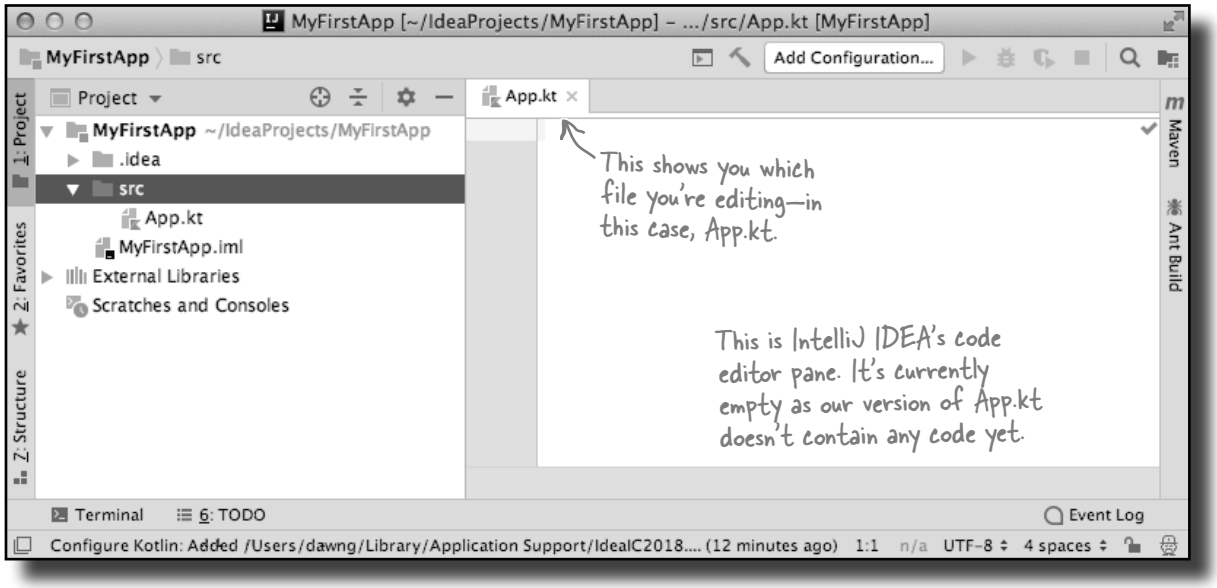
Through most of this book, we're going to use the longer form of the `main` function because this works for all versions of Kotlin.



Build application
Add function
Update function
Use REPL

Add the main function to App.kt

To add the `main` function to your project, open the file *App.kt* by double-clicking on it in IntelliJ IDEA's explorer. This opens the code editor, which you use to view and edit files:



Then, update your version of *App.kt* so that it matches ours below:

```
fun main(args: Array<String>) {
    println("Pow!")
}
```

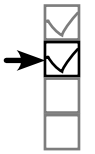
Let's try running our code to see what happens.

there are no Dumb Questions

Q: Do I have to add a `main` function to every Kotlin file I create?

A: No. A Kotlin application might use dozens (or even hundreds) of files, but you may only have *one* with a `main` function—the one that starts the application running.

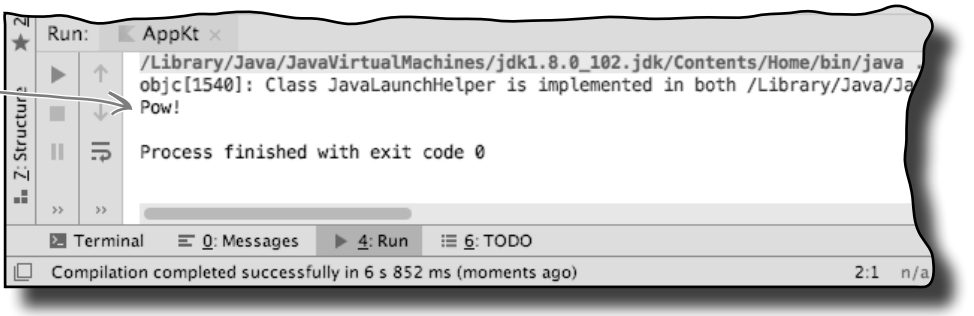
Test drive



You run code in IntelliJ IDEA by going to the Run menu, and selecting the Run command. When prompted, choose the AppKt option. This builds the project, and runs the code.

After a short wait, you should see "Pow!" displayed in an output window at the bottom of the IDE like this:

Here's the output text in the IDE.

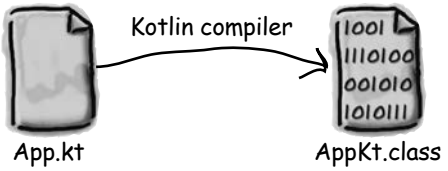


What the Run command does

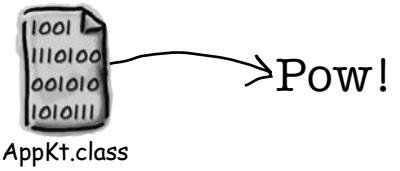
When you use the Run command, IntelliJ IDEA goes through a couple of steps before it shows you the output of your code:

1. **The IDE compiles your Kotlin source code into JVM bytecode.** Assuming your code has no errors, compiling the code creates one or more class files that can run in a JVM. In our case, compiling *App.kt* creates a class file called *AppKt.class*.

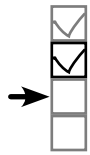
It specifically compiles our source code into JVM bytecode because when we created the project, we selected the JVM option. Had we chosen to run it in another environment, the compiler would have compiled it into code for that environment instead.



2. **The IDE starts the JVM and runs AppKt.class.** The JVM translates the *AppKt.class* bytecode into something the underlying platform understands, then runs it. This displays the `String` "Pow!" in the IDE's output window.



Now that we know our function works, let's look at how we can update it to make it do more.



Build application
Add function
Update function
Use REPL

What can you say in the main function?

Once you're inside the `main` function (or any other function, for that matter), the fun begins. You can say all the normal things that you say in most programming languages to make your application do something.

You can get your code to:

Do something (statements)

```
var x = 3
val name = "Cormoran"
x = x * 10
print("x is $x.")
//This is a comment
```

Do something again and again (loops)

```
while (x > 20) {
    x = x - 1
    print(" x is now $x.")
}
for (i in 1..10) {
    x = x + 1
    print(" x is now $x.")
}
```

Do something under a condition (branching)

```
if (x == 20) {
    println(" x must be 20.")
} else {
    println(" x isn't 20.")
}
if (name.equals("Cormoran")) {
    println("$name Strike")
}
```

We'll look at these in more detail over the next few pages.

Syntax Up Close

Here are some general syntax hints and tips for while you're finding your feet in Kotlinville:

* A single-lined comment begins with two forward slashes:

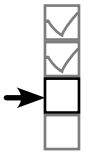
```
//This is a comment
```

* Most white space doesn't matter:

```
x           =           3
```

* Define a variable using `var` or `val`, followed by the variable's name. Use `var` for variables whose value you want to change, and `val` for ones whose value will stay the same. You'll learn more about variables in Chapter 2:

```
var x = 100
val serialNo = "AS498HG"
```



Build application
Add function
Update function
Use REPL

Loop and loop and loop...

Kotlin has three standard looping constructs: `while`, `do-while` and `for`. For now we'll just focus on `while`.

The syntax for `while` loops is relatively simple. So long as some condition is true, you do everything inside the loop *block*. The loop block is bounded by a pair of curly braces, and whatever you need to repeat needs to be inside that block.

If you just have one line of code in the loop block, you can omit the curly braces.

The key to a well-behaved `while` loop is its *conditional test*. A conditional test is an expression that results in a boolean value—something that is either *true* or *false*. As an example, if you say something like "While *isIceCreamInTub* is *true*, keep scooping" you have a clear boolean test. There is either ice cream in the tub, or there isn't. But if you say "While *Fred*, keep scooping", you don't have a real test. You need to change it to something like "While *Fred* is hungry, keep scooping" in order for it to make sense.

Simple boolean tests

You can do a simple boolean test by checking the value of a variable using a comparison operator. These include:

< (less than)

> (greater than)

== (equality) You use two equals signs to test for equality, not one.

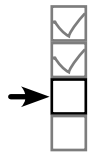
<= (less than or equal to)

>= (greater than or equal to)

Notice the difference between the assignment operator (a single equals sign) and the equals operator (two equals signs).

Here's some example code that uses boolean tests:

```
var x = 4 //Assign 4 to x
while (x > 3) {
    //The loop code will run as x is greater than 4
    println(x)
    x = x - 1
}
var z = 27
while (z == 10) {
    //The loop code will not run as z is 27
    println(z)
    z = z + 6
}
```



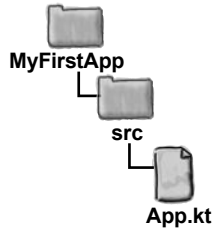
Build application
Add function
Update function
Use REPL

A loopy example

Let's update the code in *App.kt* with a new version of the `main` function. We'll update the `main` function so that it displays a message before the loop starts, each time it loops, and when the loop has ended.

Update your version of *App.kt* so that it matches ours below (our changes are in bold):

```
fun main(args: Array<String>) {
    println("Row!")                          <- Delete this line, as it's no longer needed.
    var x = 1
    println("Before the loop. x = $x.")
    while (x < 4) {
        println("In the loop. x = $x.")
        x = x + 1                            <- This prints out the value of x.
    }
    println("After the loop. x = $x.")
}
```



Let's try running the code.

Test drive

Run the code by going to the Run menu, and selecting the Run 'AppKt' command. The following text should appear in the output window at the bottom of the IDE:

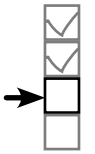
```
Before the loop. x = 1.
In the loop. x = 1.
In the loop. x = 2.
In the loop. x = 3.
After the loop. x = 4.
```

Now that you've learned how `while` loops and boolean tests work, let's look at `if` statements.

print vs. println

You've probably noticed us switching between **`print`** and **`println`**. What's the difference?

`println` inserts a *new* line (think of `println` as print new line) while **`print`** keeps printing to the *same* line. If you want each thing to print out on its own line, use `println`. If you want everything to stick together on the same line, use `print`.



Build application
Add function
Update function
Use REPL

Conditional branching

An `if` test is similar to the boolean test in a `while` loop except instead of saying "***while*** there's still ice cream..." you say "***if*** there's still ice cream..."

So that you can see how this works, here's some code that prints a `String` if one number is greater than another:

If you just have one line of code in the if block, you can leave out the curly braces.

```
fun main(args: Array<String>) {
    val x = 3
    val y = 1
    if (x > y) {
        println("x is greater than y")
    }
    println("This line runs no matter what")
}
```

This line is only executed if x is greater than y.

The above code executes the line that prints "*x* is greater than *y*" only if the condition (`x` is greater than `y`) is true. Regardless of whether it's true, though, the line that prints "This line runs no matter what" will run. So depending on the values of `x` and `y`, either one statement or two will print out.

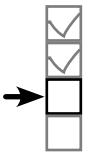
We can also add an `else` to the condition, so that we can say somthing like, "*if* there's still ice cream, keep scooping, *else* (otherwise) eat the ice cream then buy some more".

Here's an updated version of the above code that includes an `else`:

```
fun main(args: Array<String>) {
    val x = 3
    val y = 1
    if (x > y) {
        println("x is greater than y")
    } else {
        println("x is not greater than y")
    }
    println("This line runs no matter what")
}
```

This line is only executed if the condition x > y is not met.

In most languages, that's pretty much the end of the story as far as using `if` is concerned; you use it to execute code *if* conditions have been met. Kotlin, however, takes things a step further.



Build application
Add function
Update function
Use REPL

Using if to return a value

In Kotlin, you can use `if` as an **expression**, so that it returns a value. It's like saying "*if* there's ice cream in the tub, return one value, else return a different value". You can use this form of `if` to write code that's more concise.

Let's see how this works by reworking the code you saw on the previous page. Previously, we used the following code to print a `String`:

```
if (x > y) {
    println("x is greater than y")
} else {
    println("x is not greater than y")
}
```

When you use if as an expression, you MUST include an else clause.

We can rewrite this using an `if` expression like so:

```
println(if (x > y) "x is greater than y" else "x is not greater than y")
```

The code:

```
if (x > y) "x is greater than y" else "x is not greater than y"
```

is the `if` expression. It first checks the `if`'s condition: `x > y`. If this condition is *true*, the expression returns the `String` "x is greater than y". Otherwise (`else`) the condition is *false*, and the expression returns the `String` "x is not greater than y" instead.

The code then prints the value of the `if` expression using `println`:

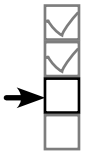
```
println(if (x > y) "x is greater than y" else "x is not greater than y")
```

If x is greater than y, the code prints "x is greater than y". If x is not greater than y, the code prints "x is not greater than y" instead.

So if `x` is greater than `y`, "x is greater than y" gets printed. If it's not, "x is not greater than y" gets printed instead.

As you can see, using an `if` expression in this way has the same effect as the code you saw on the previous page, but it's more concise.

We'll show you the code for the entire function on the next page.



Build application
Add function
Update function
Use REPL

Update the main function

Let's update the code in *App.kt* with a new version of the `main` function that uses an `if` expression. Replace the code in your version of *App.kt* so that it matches ours below:

```
fun main(args: Array<String>) {
    var x = 1
    println("Before the loop. x = $x.")
    while (x < 4) {
        println("In the loop. x = $x.")
        x = x + 1
    }
    println("After the loop. x = $x.")
    val x = 3
    val y = 1
    println(if (x > y) "x is greater than y" else "x is not greater than y")
    println("This line runs no matter what")
}
```

Delete these lines.

Let's take the code for a test drive.

Test drive

Run the code by going to the Run menu, and selecting the Run 'AppKt' command. The following text should appear in the output window at the bottom of the IDE:

```
x is greater than y
This line runs no matter what
```

Now that you've learned how to use `if` for conditional branching and expressions, have a go at the following exercise.

Code Magnets

Somebody used fridge magnets to write a useful new **main** function that prints the `String` "YabbaDabbaDo". Unfortunately, a freak kitchen whirlwind has dislodged the magnets. Can you piece the code back together again?

You won't need to use all of the magnets.

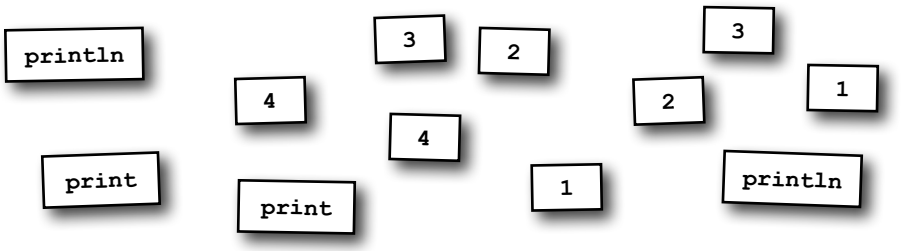
```
fun main(args: Array<String>) {
    var x = 1

    while (x <          ) {

        ..........(if (x == ..........) "Yab" else "Dab")

        ..........("ba")

        x = x + 1
    }
    if (x ==          ) println("Do")
}
```



Answers on page 29.

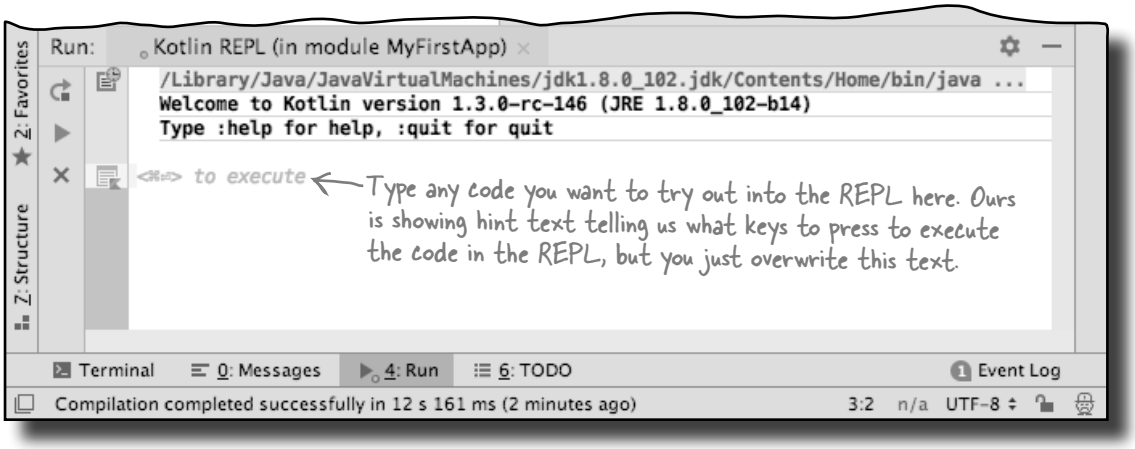
- [x] Build application
- [x] Add function
- [x] Update function
- [ ] **Use REPL**

Using the Kotlin interactive shell

We're nearly at the end of the chapter, but before we go, there's one more thing we want to introduce you to: the Kotlin interactive shell, or REPL. The REPL allows you to quickly try out snippets of code outside your main code.

REPL stands for Read–Eval–Print Loop, but nobody ever calls it that.

You open the REPL by going to the Tools menu in IntelliJ IDEA and choosing Kotlin → Kotlin REPL. This opens a new pane at the bottom of the screen like this:

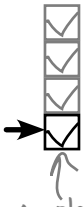


To use the REPL, simply type the code you want to try out into the REPL window. As an example, try adding the following:

```
println("I like turtles!")
```

Once you've added the code, execute it by clicking on the large green Run button on the left side of the REPL window After a pause, you should see the output "I like turtles!" in the REPL window:

You can add multi-line code snippets to the REPL



Build application
Add function
Update function
Use REPL

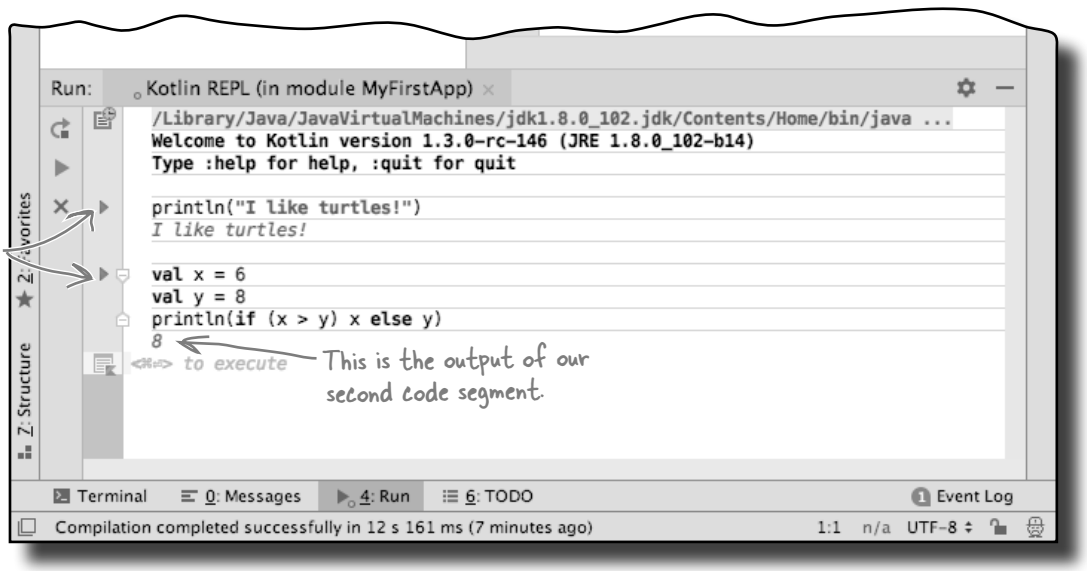
We've completed all the steps for this chapter.

As well as adding single-line code snippets to the REPL, as we did on the previous page, you can try out code segments that take up multiple lines. As an example, try adding the following lines to the REPL window:

```
val x = 6
val y = 8
println(if (x > y) x else y)
```

This prints the larger of two numbers, x and y.

When you execute the code, you should see the output `8` in the REPL like this:



These look like small versions of the execute button, but they're not. They indicate which blocks of code you've executed.

This is the output of our second code segment.

It's exercise time

Now that you've learned how to write Kotlin code and seen some of its basic syntax, have a go at the following exercises. Remember, if you're unsure, you can try out any code snippets in the REPL.

BE the Compiler

Each of the Kotlin files on this page represents a complete source file. Your job is to play like you're the compiler and determine whether each of these files will compile. If they won't compile, how would you fix them?

A

```
fun main(args: Array<String>) {
    var x = 1
    while (x < 10) {
        if (x > 3) {
            println("big x")
        }
    }
}
```

B

```
fun main(args: Array<String>) {
    val x = 10
    while (x > 1) {
        x = x - 1
        if (x < 3) println("small x")
    }
}
```

C

```
fun main(args: Array<String>) {
    var x = 10
    while (x > 1) {
        x = x - 1
        print(if (x < 3) "small x")
    }
}
```

BE the Compiler Solution

Each of the Kotlin files on this page represents a complete source file. Your job is to play like you're the compiler and determine whether each of these files will compile. If they won't compile, how would you fix them?

A

```
fun main(args: Array<String>) {
    var x = 1
    while (x < 10) {
        x = x + 1
        if (x > 3) {
            println("big x")
        }
    }
}
```

This will compile and run with no output, but without a line added to the program, it will run forever in an infinite "while" loop.

B

```
fun main(args: Array<String>) {
    val var x = 10
    while (x > 1) {
        x = x - 1
        if (x < 3) println("small x")
    }
}
```

This won't compile. x has been defined using val, which means that its value can't change. The code therefore can't update the value of x inside the "while" loop. To fix, change val to var.

C

```
fun main(args: Array<String>) {
    var x = 10
    while (x > 1) {
        x = x - 1
        print(if (x < 3) "small x" else "big x")
    }
}
```

This won't compile as it uses an if expression with no else clause. To fix, add the else clause.

Mixed Messages

A short Kotlin program is listed below. One block of the program is missing. Your challenge is to match the candidate block of code (on the left), with the output that you'd see if the block were inserted. Not all the lines of output will be used, and some lines of output may be used more than once. Draw lines connecting the candidate blocks of code with their matching output.

```
fun main(args: Array<String>) {
    var x = 0
    var y = 0
    while (x < 5) {
        [                    ]   <- The candidate code goes here.
        print("$x$y ")
        x = x + 1
    }
}
```

Candidates:

Match each candidate with one of the possible outputs.

```
y = x - y
```

```
y = y + x
```

```
y = y + 3
if (y > 4) y = y - 1
```

```
x = x + 2
y = y + x
```

```
if (y < 5) {
    x = x + 1
    if (y < 3) x = x - 1
}
y = y + 3
```

Possible output:

```
00 11 23 36 410
```

```
00 11 22 33 44
```

```
00 11 21 32 42
```

```
03 15 27 39 411
```

```
22 57
```

```
02 14 25 36 47
```

```
03 26 39 412
```

Mixed Messages Solution

A short Kotlin program is listed below. One block of the program is missing. Your challenge is to match the candidate block of code (on the left), with the output that you'd see if the block were inserted. Not all the lines of output will be used, and some lines of output may be used more than once. Draw lines connecting the candidate blocks of code with their matching output.

```
fun main(args: Array<String>) {
    var x = 0
    var y = 0
    while (x < 5) {
        [                    ]

        print("$x$y ")
        x = x + 1
    }
}
```

Candidates:

```
y = x - y
```

```
y = y + x
```

```
y = y + 3
if (y > 4) y = y - 1
```

```
x = x + 2
y = y + x
```

```
if (y < 5) {
    x = x + 1
    if (y < 3) x = x - 1
}
y = y + 3
```

Possible output:

```
00 11 23 36 410
```

```
00 11 22 33 44
```

```
00 11 21 32 42
```

```
03 15 27 39 411
```

```
22 57
```

```
02 14 25 36 47
```

```
03 26 39 412
```

Code Magnets Solution

Somebody used fridge magnets to write a useful new **main** function that prints the `String` "YabbaDabbaDo". Unfortunately, a freak kitchen whirlwind has dislodged the magnets. Can you piece the code back together again?

You won't need to use all of the magnets.

```
fun main(args: Array<String>) {
    var x = 1

    while (x < 3 ) {

        print (if (x == 1 ) "Yab" else "Dab")

        print ("ba")

        x = x + 1
    }
    if (x == 3 ) println("Do")
}
```

println

4

4

2

2

1

println

You didn't need to use these magnets.

Your Kotlin Toolbox

You've got Chapter 1 under your belt and now you've added Kotlin basic syntax to your toolbox.

You can download the full code for the chapter from https://tinyurl.com/HFKotlin.

BULLET POINTS

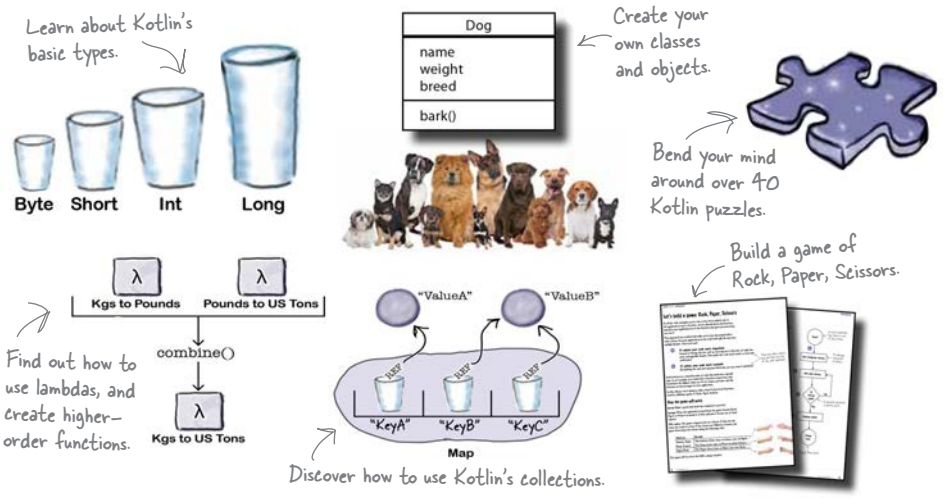
- Use `fun` to define a function.
- Every application needs a function named `main`.
- Use `//` to denote a single-lined comment.
- A `String` is a string of characters. You denote a `String` value by enclosing its characters in double quotes.
- Code blocks are defined by a pair of curly braces `{ }`.
- The assignment operator is *one* equals sign `=`.
- The equals operator uses *two* equals signs `==`.
- Use `var` to define a variable whose value may change.
- Use `val` to define a value whose value will stay the same.
- A `while` loop runs everything within its block so long as the conditional test is *true*.
- If the conditional test is *false*, the `while` loop code block won't run, and execution will move down to the code immediately after the loop block.
- Put a conditional test inside parentheses `( )`.
- Add conditional branches to your code using `if` and `else`. The `else` clause is optional.
- You can use `if` as an expression so that it returns a value. In this case, the `else` clause is mandatory.

O'REILLY®

Head First Kotlin

What will you learn from this book?

Head First Kotlin is a complete introduction to coding in Kotlin. This hands-on book helps you learn the Kotlin language with a unique method that goes beyond syntax and how-to manuals, and teaches you how to think like a great Kotlin developer. You'll learn everything from language fundamentals to collections, generics, lambdas, and higher-order functions. Along the way, you'll get to play with both object-oriented and functional programming. If you want to really understand Kotlin, this is the book for you.



"Clear, intuitive, and easy to understand. If you're new to Kotlin, this is an excellent introduction."

— Ken Kousen
Official Kotlin Trainer, certified by JetBrains

"*Head First Kotlin* will definitely help you come to grips fast, build a solid foundation, and (re)gain your joy in writing code."

— Ingo Krotzky
Kotlin learner

"At last! Learn Kotlin without knowing Java. Simple, concise and fun, this is the book I've been waiting for."

— Dr. Matt Wenham
data scientist and Python coder

Why does this book look so different?

Based on the latest research in cognitive science and learning theory, *Head First Kotlin* uses a visually rich format to engage your mind, rather than a text-heavy approach that puts you to sleep. Why waste your time struggling with new concepts? This multi-sensory learning experience is designed for the way your brain really works.

For sale in the Indian Subcontinent (India, Pakistan, Bangladesh, Sri Lanka, Nepal, Bhutan, Maldives) and African Continent (excluding Morocco, Algeria, Tunisia, Libya, Egypt, and the Republic of South Africa) only. Illegal for sale outside of these countries

KOTLIN / PROGRAMMING

oreilly.com
twitter.com/headfirstlabs
facebook.com/HeadFirst

MRP: ₹ 1,625.00

SPD®
SHROFF PUBLISHERS & DISTRIBUTORS PVT. LTD.

ISBN: 978-93-5213-807-4

9 789352 138074

First Edition/2019/Paperback/English