



Peter Sommerhoff

*Foreword by Bernhard Rumpe,*  
RWTH Aachen University

# Kotlin for Android App Development



# Contents

- 1. Cover Page**
- 2. About This E-Book**
- 3. Title Page**
- 4. Copyright Page**
- 5. Contents**
- 6. Listings**
- 7. Foreword**
- 8. Preface**
  - 1. Who This Book Is For**
  - 2. How to Follow This Book**
  - 3. Book Conventions**
- 9. Acknowledgments**
- 10. About the Author**
- 11. I Learning Kotlin**
  - 1. 1 Introducing Kotlin**
    - 1. What Is Kotlin?**
    - 2. Goals and Language Concepts**
    - 3. Why Use Kotlin on Android?**
    - 4. Kotlin versus Java 8**
    - 5. Tool Support and Community**
    - 6. Business Perspective**
    - 7. Who's Using Kotlin?**
    - 8. Summary**
  - 2. 2 Diving into Kotlin**
    - 1. Kotlin REPL**
    - 2. Variables and Data Types**
    - 3. Conditional Code**
    - 4. Loops and Ranges**
    - 5. Functions**
    - 6. Null Safety**
    - 7. Equality Checks**
    - 8. Exception Handling**
    - 9. Summary**
  - 3. 3 Functional Programming in Kotlin**
    - 1. Purpose of Functional Programming**
    - 2. Functions**
    - 3. Lambda Expressions**
    - 4. Higher-Order Functions**
    - 5. Working with Collections**
    - 6. Scoping Functions**
    - 7. Lazy Sequences**
    - 8. Summary**
  - 4. 4 Object Orientation in Kotlin**

- 1. Classes and Object Instantiation**
- 2. Properties**
- 3. Methods**
- 4. Primary and Secondary Constructors**
- 5. Inheritance and Overriding Rules**
- 6. Type Checking and Casting**
- 7. Visibilities**
- 8. Data Classes**
- 9. Enumerations**
- 10. Sealed Classes**
- 11. Objects and Companions**
- 12. Generics**
- 13. Summary**

## **5. 5 Interoperability with Java**

- 1. Using Java Code from Kotlin**
- 2. Using Kotlin Code from Java**
- 3. Best Practices for Interop**
- 4. Summary**

## **6. 6 Concurrency in Kotlin**

- 1. Concurrency**
- 2. Kotlin Coroutines**
- 3. Summary**

## **12. II Kotlin on Android**

### **1. 7 Android App Development with Kotlin: Kudoo App**

- 1. Setting Up Kotlin for Android**
- 2. App #1: Kudoo, a To-Do List App**
- 3. Summary**

### **2. 8 Android App Development with Kotlin: Nutrilicious**

- 1. Setting Up the Project**
- 2. Adding a RecyclerView to the Home Screen**
- 3. Fetching Data from the USDA Nutrition API**
- 4. Mapping JSON Data to Domain Classes**
- 5. Introducing a ViewModel for Search**
- 6. Letting Users Search Foods**
- 7. Introducing Fragments I: The Search Fragment**
- 8. Introducing Fragments II: The Favorites Fragment**
- 9. Store User's Favorite Foods in a Room Database**
- 10. Fetching Detailed Nutrition Data from the USDA Food Reports API**
- 11. Integrating the Details Activity**
- 12. Storing Food Details in the Database**
- 13. Adding RDIs for Actionable Data**
- 14. Improving the User Experience**
- 15. Summary**

### **3. 9 Kotlin DSLs**

- 1. Introducing DSLs**
- 2. Creating a DSL in Kotlin**
- 3. DSL for Android Layouts with Anko**
- 4. DSL for Gradle Build Scripts**

## **5. Summary**

### **4. 10 Migrating to Kotlin**

- 1. On Software Migrations**
- 2. Leading the Change**
- 3. Partial or Full Migration**
- 4. Where to Start**
- 5. Tool Support**
- 6. Summary**

### **5. A Further Resources**

- 1. Official Resources**
- 2. Community**
- 3. Functional Programming**
- 4. Kotlin DSLs**
- 5. Migrating to Kotlin**
- 6. Testing**

### **13. Credits**

### **14. Glossary**

### **15. Index**

## About This E-Book

EPUB is an open, industry-standard format for e-books. However, support for EPUB and its many features varies across reading devices and applications. Use your device or app settings to customize the presentation to your liking. Settings that you can customize often include font, font size, single or double column, landscape or portrait mode, and figures that you can click or tap to enlarge. For additional information about the settings and features on your reading device or app, visit the device manufacturer's Web site.

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the e-book in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a “Click here to view code image” link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

# The Pearson Addison-Wesley Developer's Library



Visit [informit.com/devlibrary](http://informit.com/devlibrary) for a complete list of available publications.

The Developer's Library series from Pearson Addison-Wesley provides practicing programmers with unique, high-quality references and tutorials on the latest programming languages and technologies they use in their daily work. All books in the Developer's Library are written by expert technology practitioners who are exceptionally skilled at organizing and presenting information in a way that is useful for other programmers.

Developer's Library titles cover a wide range of topics, from open

source programming languages and technologies, mobile application developments, and web development to Java programming and more.



Make sure to connect with us!

[informit.com/socialconnect](http://informit.com/socialconnect)



# Kotlin for Android App Development

**Peter Sommerhoff**



Boston • Columbus • New York • San Francisco •  
Amsterdam • Cape Town  
Dubai • London • Madrid • Milan • Munich • Paris •  
Montreal • Toronto • Delhi • Mexico City  
São Paulo • Sydney • Hong Kong • Seoul • Singapore •  
Taipei • Tokyo

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 publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at [corpsales@pearsoned.com](mailto:corpsales@pearsoned.com) or (800) 382-3419.

For government sales inquiries, please contact [governmentsales@pearsoned.com](mailto:governmentsales@pearsoned.com).

For questions about sales outside the U.S., please contact [intlcs@pearson.com](mailto:intlcs@pearson.com).

Visit us on the Web: [informit.com/aw](http://informit.com/aw)

Library of Congress Control Number: 2018958190

Copyright © 2019 Pearson Education, Inc.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit [www.pearsoned.com/permissions/](http://www.pearsoned.com/permissions/).

ISBN-13: 978-0-13-485419-9

ISBN-10: 0-13-485419-5

**Publisher**

Mark L. Taub

**Acquisitions Editor**

Malobika Chakraborty

**Development Editor**

Sheri Replin

**Managing Editor**

Sandra Schroeder

**Full-Service Production Manager**

Julie B. Nahil

**Project Manager**

Suganya Karuppasamy

**Copy Editor**

Stephanie M. Geels

**Indexer**

Ken Johnson

**Proofreader**

Larry Sulky

**Technical Reviewers**

Miguel Castiblanco

Amanda Hill

Ty Smith

**Cover Designer**

Chuti Prasertsith

**Compositor**

codemantra

# Contents

**Listings**

**Foreword**

**Preface**

**Acknowledgments**

**About the Author**

## **I: Learning Kotlin**

### **1 Introducing Kotlin**

What Is Kotlin?

Goals and Language Concepts

Why Use Kotlin on Android?

Java on Android

Kotlin on Android

Kotlin versus Java 8

Tool Support and Community

Business Perspective

Who's Using Kotlin?

Summary

### **2 Diving into Kotlin**

Kotlin REPL

Variables and Data Types

Variable Declarations

Basic Data Types

Type Inference

Conditional Code

If and When as Statements

Conditional Expressions

Loops and Ranges

While Loops

For Loops

Functions

Function Signatures

Shorthand for Single-Expression Functions

Main Function

Default Values and Named Parameters

Extension Functions

Infix Functions

Operator Functions

Null Safety

Nullable Types

Working with Nullables

Equality Checks

Floating Point Equality

Exception Handling

Principles of Exception Handling

Exception Handling in Kotlin

Checked and Unchecked Exceptions

Summary

### **3 Functional Programming in Kotlin**

Purpose of Functional Programming

Benefits of Functional Programming

Functions

Lambda Expressions

Using Lambdas in Kotlin

Higher-Order Functions

Inlining Higher-Order Functions

## Working with Collections

### Kotlin Collections API versus Java Collections API

Instantiating Collections in Kotlin

Accessing and Editing Collections

Filtering Collections

Mapping Collections

Grouping Collections

Associating Collections

Calculating a Minimum, Maximum, and Sum

Sorting Collections

Folding Collections

Chaining Function Calls

## Scoping Functions

Using `let`

Using `apply`

Using `with`

Using `run`

Using `also`

Using `use`

Combining Higher-Order Functions

Lambdas with Receivers

## Lazy Sequences

Lazy Evaluation

Using Lazy Sequences

Performance of Lazy Sequences

## Summary

## **4 Object Orientation in Kotlin**

### Classes and Object Instantiation

## Properties

Properties versus Fields

Getters and Setters

Late-Initialized Properties

Delegated Properties

Predefined Delegates

Delegating to a Map

Using Delegated Implementations

## Methods

Extension Methods

Nested and Inner Classes

## Primary and Secondary Constructors

Primary Constructors

Secondary Constructors

## Inheritance and Overriding Rules

Interfaces

Abstract Classes

Open Classes

Overriding Rules

## Type Checking and Casting

Type Checking

Type Casting

Smart Casts

## Visibilities

Declarations in Classes or Interfaces

Top-Level Declarations

## Data Classes

Using Data Classes

Inheritance with Data Classes

[Enumerations](#)

[Sealed Classes](#)

[Objects and Companions](#)

[Object Declarations as Singletons](#)

[Companion Objects](#)

[Generics](#)

[Generic Classes and Functions](#)

[Covariance and Contravariance](#)

[Declaration-Site Variance](#)

[Use-Site Variance](#)

[Bounded Type Parameters](#)

[Star Projections](#)

[Summary](#)

## **5 Interoperability with Java**

[Using Java Code from Kotlin](#)

[Calling Getters and Setters](#)

[Handling Nullability](#)

[Escaping Clashing Java Identifiers](#)

[Calling Variable-Argument Methods](#)

[Using Operators](#)

[Using SAM Types](#)

[Further Interoperability Considerations](#)

[Using Kotlin Code from Java](#)

[Accessing Properties](#)

[Exposing Properties as Fields](#)

[Using File-Level Declarations](#)

[Calling Extensions](#)

[Accessing Static Members](#)

[Generating Method Overloads](#)

[Using Sealed and Data Classes](#)

[Visibilities](#)

[Getting a KClass](#)

[Handling Signature Clashes](#)

[Using Inline Functions](#)

[Exception Handling](#)

[Using Variant Types](#)

[The Nothing Type](#)

[Best Practices for Interop](#)

[Writing Kotlin-Friendly Java Code](#)

[Writing Java-Friendly Kotlin Code](#)

[Summary](#)

## **6 Concurrency in Kotlin**

[Concurrency](#)

[Challenges](#)

[State-of-the-Art Solutions](#)

[Kotlin Coroutines](#)

[Setup](#)

[Concepts](#)

[Suspending Functions](#)

[Coroutine Builders](#)

[Generators](#)

[Actors and Channels](#)

[Concurrency Styles](#)

[Coroutines in Practice](#)

[Under the Hood](#)

[Summary](#)

## **II: Kotlin on Android**

## **7 Android App Development with Kotlin: Kudoo App**

Setting Up Kotlin for Android

Using Kotlin in Android Studio

Auto-Generated Gradle Configuration

Adapting Your Gradle Configuration

Using Annotation Processors

Converting Java Code to Kotlin

App #1: Kudoo, a To-Do List App

Creating the Project

Adding the RecyclerView

Adding a Room Database

Using a ViewModel

Integrating LiveData

Adding New To-Do Items

Enabling Checking Off To-Do Items

Summary

## **8 Android App Development with Kotlin: Nutrilicious**

Setting Up the Project

Adding a RecyclerView to the Home Screen

Layout for MainActivity

Layout for RecyclerView Items

Implementing the Food Model

Implementing the RecyclerView Adapter

Adding the RecyclerView to MainActivity

Fetching Data from the USDA Nutrition API

Using Retrofit

Performing API Requests

Mapping JSON Data to Domain Classes

[Mapping JSON to DTOs](#)

[Mapping DTOs to Models](#)

[Introducing a `ViewModel` for Search](#)

[Letting Users Search Foods](#)

[Implementing a Search Interface](#)

[Introducing Fragments I: The Search Fragment](#)

[Introducing Fragments II: The Favorites Fragment](#)

[Store User's Favorite Foods in a Room Database](#)

[Fetching Detailed Nutrition Data from the USDA Food Reports API](#)

[Integrating the Details Activity](#)

[Storing Food Details in the Database](#)

[Adding RDIs for Actionable Data](#)

[Improving the User Experience](#)

[Indicating Empty Search Results](#)

[Indicate Progress in the Details Activity](#)

[Summary](#)

[9 Kotlin DSLs](#)

[Introducing DSLs](#)

[What Is a DSL?](#)

[Benefits and Drawbacks](#)

[Creating a DSL in Kotlin](#)

[DSL to Build Complex Objects](#)

[Immutability through Builders](#)

[Nesting Deeper](#)

[Introducing `@DslMarker`](#)

[Leveraging Language Features](#)

[DSL for Android Layouts with Anko](#)

[Creating Layouts Programmatically](#)

[Anko Dependencies](#)

[Creating Layouts with Anko](#)

[Adding Layout Parameters](#)

[Migrating Kudoo's AddTodoActivity to Anko Layouts](#)

[Adding Custom Views](#)

[Anko Layouts versus XML Layouts](#)

[DSL for Gradle Build Scripts](#)

[Migrating Nutrilicious to Gradle Kotlin DSL](#)

[Using buildSrc in Gradle](#)

[Benefits and Drawbacks](#)

[Summary](#)

## **10 Migrating to Kotlin**

[On Software Migrations](#)

[Risks and Benefits](#)

[Leading the Change](#)

[Getting Buy-In](#)

[Sharing Knowledge](#)

[Partial or Full Migration](#)

[Partial Migration](#)

[Full Migration](#)

[Where to Start](#)

[Test Code](#)

[Production Code](#)

[Pet Projects](#)

[Make a Plan](#)

[Tool Support](#)

[Java-to-Kotlin Converter](#)

[Adjusting Autoconverted Code](#)

[Summary](#)

**A Further Resources**

[Official Resources](#)

[Community](#)

[Functional Programming](#)

[Kotlin DSLs](#)

[Migrating to Kotlin](#)

[Testing](#)

**[Glossary](#)**

**[Index](#)**

## Listings

- [2.1 Declaring Mutable Variables](#)
- [2.2 Declaring Read-Only Variables](#)
- [2.3 Type Inference](#)
- [2.4 Conditions With `if`](#)
- [2.5 `when` for Cascades of Conditions](#)
- [2.6 Types of Conditions for `when`](#)
- [2.7 Arbitrary Boolean Conditions With `when`](#)
- [2.8 `If` Expression](#)
- [2.9 Ternary Conditional Operator with `if`](#)
- [2.10 Using `when` as an Expression](#)
- [2.11 `while` Loop](#)
- [2.12 `do-while` Loop](#)
- [2.13 `for` Loops](#)
- [2.14 Ranges in `for` Loops](#)
- [2.15 Declaring a Function](#)
- [2.16 Calling a Function](#)
- [2.17 Transforming the Function Body into a Single Expression](#)
- [2.18 Shorthand Function Notation Using Equals Sign](#)
- [2.19 Declaring a Main Function](#)
- [2.20 Defining Default Values for Parameters](#)
- [2.21 Using Named Parameters](#)
- [2.22 An Overloaded Function](#)
- [2.23 Creating and Calling Extension Functions](#)
- [2.24 Static Resolving of Extension Functions](#)
- [2.25 Importing Extension Functions](#)

- 2.26 The Infix Function `to`
  - 2.27 Calling an Infix Function
  - 2.28 Defining Own Infix Functions
  - 2.29 Defining and Calling Operators
  - 2.30 Nullable and Non-Nullable Types
  - 2.31 Accessing Members of Nullable Variables
  - 2.32 Elvis Operator
  - 2.33 Unsafe Call Operator—Member Access
  - 2.34 Unsafe Call Operator—Passing Arguments
  - 2.35 Referential and Structural Equality Checks
  - 2.36 Equality of Floating Point Values
  - 2.37 Throwing and Catching Exceptions
  - 2.38 Using `try-catch` as an Expression
  - 2.39 Using `throw` as an Expression
  - 2.40 Using `Nothing` for Non-Returning Functions
- 
- 3.1 Function Signature
  - 3.2 Function Type
  - 3.3 Full Lambda Expression
  - 3.4 Assigning a Lambda Expression
  - 3.5 Type Inference for Lambda Arguments
  - 3.6 Type Inference for Lambda Variables
  - 3.7 Implicit Argument Name `it`
  - 3.8 Function Pointers
  - 3.9 Defining a Higher-Order Function
  - 3.10 Calling a Higher-Order Function
  - 3.11 Lambdas as Last Argument
  - 3.12 Call to Higher-Order Function
  - 3.13 Inlining Higher-Order Functions
  - 3.14 Result of Inlined Function Call

- [3.15 Returning from an Enclosing Function](#)
- [3.16 Returning from an Enclosing Function—Inlined Result](#)
- [3.17 Disallowing Non-Local Returns Using `crossinline`](#)
- [3.18 Helper Functions for Creating Collections](#)
- [3.19 Indexed Access Operator for Reading](#)
- [3.20 Indexed Access Operator for Writing](#)
- [3.21 Filtering a Collection](#)
- [3.22 Searching a Collection](#)
- [3.23 Applying a Function to Each Element Using `map`](#)
- [3.24 Signature of the `map` Function](#)
- [3.25 Projections Using `map`](#)
- [3.26 Grouping Words by Length](#)
- [3.27 Turning a List into a Map](#)
- [3.28 Minimum, Maximum, and Sum by Some Metric](#)
- [3.29 Sorting Collections](#)
- [3.30 Calculating Sum and Product Using `fold`](#)
- [3.31 Chaining Functions](#)
- [3.32 Scoping with `let`](#)
- [3.33 Handling Nullables with `let`](#)
- [3.34 Using `apply`](#)
- [3.35 Initializing Objects with `apply`](#)
- [3.36 Return Value of `apply`](#)
- [3.37 Using `with` for Multiple Calls on Same Object](#)
- [3.38 Using `with` to Minimize Scope](#)
- [3.39 Using `run` with Nullables](#)
- [3.40 Using `run` for Immediate Function Application](#)
- [3.41 Using `run` to Minimize Variable Scope](#)

- 3.42 Using `run` to Turn Parameter into Receiver
  - 3.43 Using `also` for Validation
  - 3.44 Using `also` for Ancillary Operations in a Function Chain
  - 3.45 Handling `Closeables` with `use`
  - 3.46 Combining Scope Operators
  - 3.47 Combining All Higher-Order Functions
  - 3.48 Signatures of `let` and `run`
  - 3.49 Filtering and Mapping a Collection or Sequence
  - 3.50 Lazy Sequence from Scratch
  - 3.51 Lazy Sequence from Collection
  - 3.52 Lazy Sequences with `generateSequence`
  - 3.53 Using `take` and `drop`
  - 3.54 Using `take` (and `drop`) Early
  - 3.55 Eager Evaluation on Collections
  - 3.56 Output for Eager Evaluation
  - 3.57 Lazy Evaluation on Sequences
  - 3.58 Output for Lazy Evaluation
- 4.1 Class Declaration and Instantiation
  - 4.2 Adding a Property the Hard Way
  - 4.3 Using a Primary Constructor
  - 4.4 Idiomatic Way to Add a Simple Property
  - 4.5 Property Access Calls Getter or Setter
  - 4.6 Custom Getter and Setter
  - 4.7 Using Late-Initialized Properties
  - 4.8 Syntax of Delegated Properties
  - 4.9 Implementing a Custom Delegate
  - 4.10 Using Lazy Properties
  - 4.11 Using Observable Properties

- 4.12 Delegating Properties to a Map
- 4.13 Delegating to an Implementation Using `by`
- 4.14 Delegated Implementation
- 4.15 Delegating Mutable Set
- 4.16 Declaring and Calling Methods
- 4.17 Extension Methods
- 4.18 Comparing Nested and Inner Classes
- 4.19 Using a Primary Constructor
- 4.20 Adding Modifiers to a Primary Constructor
- 4.21 Upgrading Primary Constructor Parameters to Properties
- 4.22 Combining Primary and Secondary Constructors
- 4.23 Defining an Interface
- 4.24 Defining an Interface with Default Implementations
- 4.25 Inheriting from Interfaces
- 4.26 Abstract Classes and Overriding
- 4.27 Open Classes and Closed-by-Default Classes
- 4.28 Type Checks Using `is`
- 4.29 Type Casts for Nullable Types
- 4.30 Type Casts and `ClassCastException`
- 4.31 Smart Casts
- 4.32 Visibility Modifiers
- 4.33 Setting the Visibility of Constructors, Getters, and Setters
- 4.34 Declaring a Data Class
- 4.35 Generated Members of Data Classes
- 4.36 Declaring a Simple Enum Class
- 4.37 Declaring an Enum Class with Members
- 4.38 Working with Enums

- 4.39 Exhaustive `when` Expression with Enum
- 4.40 Declaring a Sealed Class with Subtypes
- 4.41 Exhaustive `when` Expression with Sealed Class
- 4.42 Simple Object Expressions
- 4.43 Object Expressions with Supertype
- 4.44 Object Expressions as Return Type
- 4.45 Object Declaration for a Registry Singleton
- 4.46 Accessing Object Members
- 4.47 A Factory as Companion Object
- 4.48 Companion Object with Factory Interface
- 4.49 Declaring and Instantiating a Generic Class
- 4.50 Generalizing the `BinaryTree` Class
- 4.51 Declaring and Using a Generic Function
- 4.52 A Generic Function that Accesses Its Type Parameter
- 4.53 A Generic Function Using Reification
- 4.54 Inlined Code of Generic Function
- 4.55 Covariance of Subclasses in Assignments
- 4.56 Covariance of Return Types
- 4.57 Invariance of Arrays in Kotlin (Type-Safe)
- 4.58 Invariance of Java Collections (Type-Safe)
- 4.59 Variance of Read-Only Collections in Kotlin (Type-Safe)
- 4.60 Invariance of Mutable Collections in Kotlin
- 4.61 Covariant Stack Class
- 4.62 An Invariant Mutable Stack Class
- 4.63 A Contravariant `Compare<T>` Type
- 4.64 Using Contravariance of `Compare<T>`
- 4.65 Another Contravariant Type `Repair<T>`

- [4.66 Use-Site Covariance in Assignments](#)
- [4.67 Use-Site Contravariance in Assignments](#)
- [4.68 Function Without Use-Site Variance](#)
- [4.69 Use-Site Variance Increases Function Flexibility](#)
- [4.70 Bounded Type Parameter](#)
- [4.71 Multi-Bounded Type Parameter](#)
- [4.72 Using Star Projections](#)
- [4.73 Star Projections for Variant Types](#)
- [5.1 Using Java Libraries from Kotlin](#)
- [5.2 Calling Java Getters and Setters](#)
- [5.3 Handling Platform Types in Kotlin](#)
- [5.4 Using Nullability Annotations](#)
- [5.5 Handling Name Clashes](#)
- [5.6 Calling a Variable-Argument Method](#)
- [5.7 Using Java Methods as Operators](#)
- [5.8 Using SAM Types from Kotlin](#)
- [5.9 SAM Conversion in Function Arguments](#)
- [5.10 Calling Getters and Setters](#)
- [5.11 Custom Method Name Using `@JvmName`](#)
- [5.12 Exposing a Property as a Field using `@JvmField`](#)
- [5.13 File-Level Declarations in Kotlin](#)
- [5.14 Using `@JvmName` on File Level](#)
- [5.15 Calling Top-Level Extensions](#)
- [5.16 Calling Class-Local Extensions](#)
- [5.17 Generating Static Fields on the JVM](#)
- [5.18 Using `@JvmStatic` to Generate Static Methods](#)
- [5.19 Generating Overloads with `@JvmOverloads`](#)
- [5.20 Working with Sealed Classes](#)
- [5.21 Working with Data Classes](#)

[5.22 Getting KClass and Class References](#)

[5.23 JVM Name Clash](#)

[5.24 Resolving JVM Name Clashes](#)

[5.25 Calling Inline Functions \(from Kotlin\)](#)

[5.26 Generating `Throws` Clauses](#)

[5.27 Difference in Decompiled Java Code](#)

[5.28 Mapping Declaration-Site Variance to Use Site](#)

[5.29 Adjusting Wildcard Generation](#)

[5.30 Using the `Nothing` Type](#)

[6.1 Asynchrony with Callbacks](#)

[6.2 Asynchrony with Futures](#)

[6.3 Asynchrony with `async-await` Pattern \(C#\)](#)

[6.4 Asynchrony with Coroutines](#)

[6.5 Declaring a Suspending Function](#)

[6.6 Function Signatures for Callbacks and Futures](#)

[6.7 Asynchrony with Coroutines](#)

[6.8 Naming Asynchronous Functions](#)

[6.9 Cannot Call Suspending Function from  
Nonsuspending Function](#)

[6.10 Suspending Functions and Loops](#)

[6.11 Suspending Functions and Exception Handling](#)

[6.12 Suspending Functions and Higher-Order Functions](#)

[6.13 Using `runBlocking`](#)

[6.14 Signature of `runBlocking` \(Simplified\)](#)

[6.15 Idiomatic Way to Use `runBlocking`](#)

[6.16 Launching a New Coroutine](#)

[6.17 Waiting for a Coroutine](#)

[6.18 Launching 100,000 Coroutines](#)

[6.19 Cancelling a Coroutine](#)

- 6.20 Noncooperative Coroutine Cannot Be Cancelled
- 6.21 Making Coroutine Cooperative Using `isActive`
- 6.22 Starting Asynchronous Tasks
- 6.23 Running Asynchronous Functions Synchronously
- 6.24 Idiomatic Asynchrony
- 6.25 Handling Exceptions from Asynchronous Calls
- 6.26 Launching a Coroutine on the UI Thread (Android)
- 6.27 Definition of UI Context on Android
- 6.28 Default Coroutine Dispatcher
- 6.29 Dispatching Onto New Thread
- 6.30 Unconfined Dispatcher
- 6.31 Creating a Child Coroutine
- 6.32 Comparing Dispatchers
- 6.33 Switching Context Inside Coroutine
- 6.34 Timeout for Asynchronous Calls
- 6.35 Handling Timeouts
- 6.36 Child Coroutines and Cancellation
- 6.37 Binding Coroutines to Activity #1: The `JobHolder` Interface
- 6.38 Binding Coroutines to Activity #2: The Activity
- 6.39 Binding Coroutines to Activity #3: Accessing Job from Views
- 6.40 `COROUTINE_NAME` and `COROUTINE_EXCEPTION_HANDLER`
- 6.41 `COROUTINE_CONTEXT` Combination Operator
- 6.42 Combining Contexts
- 6.43 Accessing Context Elements
- 6.44 Starting Coroutines Lazily
- 6.45 Combining Coroutines and `CompletableFuture`

[6.46 Creating a Suspending Sequence \(Generator\)](#)

[6.47 Declaration of `buildSequence`](#)

[6.48 Declaration of `SequenceBuilder` with Restricted Suspension](#)

[6.49 A Simple Actor](#)

[6.50 Actor that Keeps Reading from its Channel](#)

[6.51 Actor with Conflated Channel](#)

[6.52 Actor with Multiple Channels](#)

[6.53 Multiple Actors on Same Channels](#)

[6.54 Creating a Producer](#)

[6.55 Actor with Custom Message Type](#)

[6.56 Transformation between Concurrency Styles](#)

[6.57 Logging Current Thread \(and Coroutine\)](#)

[6.58 Enabling Coroutine Debugging on Android](#)

[6.59 Implementation of `future` Coroutine Builder \(Simplified\)](#)

[6.60 Custom Await Function \(Simplified\)](#)

[6.61 Wrapping Suspending Function for Java Interop](#)

[6.62 Suspending Function to be Examined](#)

[6.63 Step 1: Adding `Continuation` Parameter](#)

[6.64 Step 2: Removing `suspend` Modifier](#)

[6.65 Step 3: Changing Return Type to Any?](#)

[6.66 Step 4: State Machine \(Simplified\)](#)

[7.1 Project's `build.gradle` File](#)

[7.2 Module's `build.gradle` File](#)

[7.3 Add Separate Kotlin Sources Directory to Gradle \(Optional\)](#)

[7.4 Dependencies for Targeting JDK 7 or 8](#)

[7.5 Dependencies for Kotlin Reflection and Test Libraries](#)

7.6 Configuring kapt for Annotation Processing

(Example: Dagger 2)

7.7 RecyclerView Layout

7.8 RecyclerView Item Layout

7.9 TodoItem Data Class

7.10 RecyclerViewAdapter Signature

7.11 Custom ViewHolder

7.12 Enabling Experimental Kotlin Android Extensions

7.13 ViewHolder Without findViewById

7.14

```
RecyclerViewAdapter.onCreateViewHolder  
( )
```

7.15

```
RecyclerViewAdapter.onBindViewHolder()
```

7.16 RecyclerViewAdapter.getItemCount()

7.17 Complete RecyclerViewAdapter

7.18 Setting Up the RecyclerView with the Adapter

7.19 Adjusting onCreate

7.20 Gradle Dependencies for Room

7.21 Enabling the Kotlin Annotation Processor Plugin

7.22 TodoItem as an Entity

7.23 TodoItemDao for Database Access

7.24 AppDatabase

7.25 Populating the AppDatabase with Sample Data

7.26 Gradle Dependencies for Kotlin Coroutines

7.27 Using the Database from MainActivity

7.28 Gradle Dependencies for ViewModel (and  
LiveData)

7.29 Class Header of TodoViewModel

[7.30 Complete TodoViewModel](#)

[7.31 Integrating the TodoViewModel into MainActivity](#)

[7.32 Extension to Retrieve View Models](#)

[7.33 Returning LiveData from the DAO](#)

[7.34 Returning LiveData from the ViewModel](#)

[7.35 Observing the LiveData from MainActivity](#)

[7.36 Adding setItems to the RecyclerViewAdapter](#)

[7.37 Layout for AddTodoActivity](#)

[7.38 Layout for the FloatingActionButton](#)

[7.39 Setting Up the FloatingActionButton](#)

[7.40 Adjusting onCreate\(\)](#)

[7.41 Implementing the AddTodoActivity](#)

[7.42 Enabling Navigating Up from AddTodoActivity to MainActivity](#)

[7.43 Assigning Event Handlers in the RecyclerView Adapter](#)

[7.44 Assigning Event Handlers in the RecyclerView Adapter](#)

[8.1 Bottom Navigation Menu](#)

[8.2 String Resources for Bottom Navigation Menu](#)

[8.3 MainActivity Setup for the Bottom Navigation Menu](#)

[8.4 MainActivity Layout](#)

[8.5 Layout for RecyclerView Items](#)

[8.6 Resources for RecyclerView Item Layout](#)

[8.7 Food Data Class](#)

[8.8 Enabling the Experimental Kotlin Android Extensions](#)

[8.9 RecyclerView Adapter](#)

[8.10 Setting Up the RecyclerView in MainActivity](#)

[8.11 Hard-Coding the Sample Data](#)

[8.12 Calling the Setup Method in onCreate](#)

[8.13 Gradle Dependencies for Network and API Access](#)

[8.14 Constants Used for Retrofit](#)

[8.15 Adding API Keys to Your Gradle Properties](#)

[8.16 Adding a Build Config Field](#)

[8.17 Building the Retrofit Object](#)

[8.18 Building the HTTP Client](#)

[8.19 Building the HTTP Client](#)

[8.20 Encapsulating Interceptor Creation](#)

[8.21 Creating the Interceptor that Adds the API Key to the Query](#)

[8.22 Defining the Retrofit Interface to Access the \*Search API\*](#)

[8.23 Building the \*Search API\* Object](#)

[8.24 Enabling Internet Access](#)

[8.25 Gradle Dependencies for Kotlin Coroutines](#)

[8.26 Coroutine Dispatcher for Network Calls](#)

[8.27 Performing an API Request](#)

[8.28 Gradle Dependencies for Moshi](#)

[8.29 Enabling the Kotlin Annotation Processor](#)

[8.30 \*Search API\* JSON Format](#)

[8.31 Wrapper DTOs for the \*Search API\*](#)

[8.32 Food DTO for the \*Search API\*](#)

[8.33 Returning the DTO from Retrofit Calls](#)

- 8.34 Secondary Constructor to Map DTO to Model
- 8.35 Displaying Mapped Data in RecyclerView
- 8.36 Removing the Sample Data
- 8.37 Gradle Dependencies for Architecture Components
- 8.38 Step 1: Implementing the SearchViewModel
- 8.39 Step 2: Implementing the SearchViewModel
- 8.40 Using the SearchViewModel in  
MainActivity
- 8.41 getViewModel Extension Function
- 8.42 Using the SearchViewModel in  
MainActivity
- 8.43 Encapsulating the Logic For Requests
- 8.44 Menu Resource for Search Interface
- 8.45 String Resource for Search Menu
- 8.46 Searchable Configuration
- 8.47 Setting Up Search in the Android Manifest
- 8.48 Inflating and Setting Up the Search Menu
- 8.49 Handling Search Intents
- 8.50 Layout for the SearchFragment
- 8.51 Layout for the MainActivity
- 8.52 Overriding Lifecycle Methods for the Search  
Fragment
- 8.53 getViewModel Extension for Fragments
- 8.54 Accessing UI Elements Safely
- 8.55 Adjusting the RecyclerView Setup
- 8.56 Setting Up the SwipeRefreshLayout
- 8.57 Adding a Property for the Fragment
- 8.58 Extension Function to Include Fragments
- 8.59 Including a Fragment into the UI

- 8.60 Extension to Store Fragment in Activity's State
- 8.61 Restoring an Existing Fragment
- 8.62 Adding a Fragment Tag
- 8.63 Adjusting `onCreate`
- 8.64 Delegating Searches to the Search Fragment
- 8.65 Handling Swipe Refresh
- 8.66 Layout for the Favorites Fragment
- 8.67 Resources for the Layout
- 8.68 Implementing the Favorites Fragment
- 8.69 Moving Common Logic into `MainActivity`
- 8.70 Removing Duplicated Code from the Fragments
- 8.71 Implementing the Bottom Navigation Menu
- 8.72 Retaining the Last Search Results
- 8.73 Making Food an Entity
- 8.74 Adding the DAO to Access Favorites
- 8.75 Adding the AppDatabase
- 8.76 The View Model to Access Favorite Foods
- 8.77 Coroutine Dispatcher for Database Operations
- 8.78 Adjusting the Adapter to Handle Star Icon Clicks
- 8.79 Creating the Adapter with Click Listener
- 8.80 Toggling Favorites on Star Icon Click
- 8.81 Extension Function to Create Toasts
- 8.82 Favorites Fragment with View Model
- 8.83 Search Fragment
- 8.84 Adding a New Endpoint to the API Interface
- 8.85 JSON Format for Nutrition Details
- 8.86 DTO Wrappers
- 8.87 Details DTOs
- 8.88 Domain Classes for Food Details

- 8.89 Making a Test Request
- 8.90 Horizontal Divider Layout
- 8.91 Divider Dimensions
- 8.92 Details Activity Layout
- 8.93 String Resources For Headlines
- 8.94 Adding an Item Click Listener to the Adapter
- 8.95 Defining the Click Handler for List Items
- 8.96 Identifier for the Intent Extra
- 8.97 View Model for Details
- 8.98 Adding the View Model to the **DetailsActivity**
- 8.99 Using the View Model in the Details Activity
- 8.100 Displaying the Data
- 8.101 String Resource Indicating No Data
- 8.102 Providing Up Navigation
- 8.103 Defining the Entities
- 8.104 Type Converter for `List<Nutrient>`
- 8.105 Type Converter for `NutrientType`
- 8.106 DAO for Food Details
- 8.107 Extending the `AppDatabase`
- 8.108 Enabling Destructive Database Migration in Development
- 8.109 Details Repository as Single Source of Truth
- 8.110 Using the Repository in the View Model
- 8.111 Classes to Represent RDIs
- 8.112 Storing RDIs Statically
- 8.113 Using the Amount Class in Nutrient
- 8.114 Mapping `WeightUnit` to String And Vice Versa
- 8.115 Displaying a Nutrient

- [8.116 Calculating the Percentage of RDI](#)
- [8.117 Calculating the Percentage of RDI](#)
- [8.118 Showing a Snackbar to Indicate Empty Search](#)
- [8.119 Extension Function to Show Snackbar](#)
- [8.120 Adding a `ProgressBar` to the Layout](#)
- [8.121 Showing and Hiding the Progress Bar](#)
- [9.1 User DSL](#)
- [9.2 DSL Entry Point](#)
- [9.3 DSL Entry Point—Improved](#)
- [9.4 User Data Class](#)
- [9.5 Adding an Address to a User](#)
- [9.6 Shortcomings of Current DSL](#)
- [9.7 Immutable Data Classes](#)
- [9.8 User Builder](#)
- [9.9 Address Builder](#)
- [9.10 Adjusted Entry Point to the DSL](#)
- [9.11 Allowing Multiple Addresses](#)
- [9.12 Syntax of Dedicated Addresses Block](#)
- [9.13 Implementing Dedicated Addresses Block](#)
- [9.14 User DSL Annotation](#)
- [9.15 Preventing Nested Calls to Entry Point \(in `UserBuilder`\)](#)
- [9.16 Using Variables in Your DSL](#)
- [9.17 Improving DSL Readability](#)
- [9.18 Improving DSL Performance](#)
- [9.19 Creating a Layout Programmatically the Hard Way](#)
- [9.20 Anko Metadependency](#)
- [9.21 Dependencies for Anko Layouts](#)
- [9.22 Simple Anko Layout](#)

- [9.23 Adding Layout Parameters with Anko](#)
  - [9.24 Activity Layout with Anko](#)
  - [9.25 Modularized Activity Layout Using an Anko Component](#)
  - [9.26 Custom Frame Layout](#)
  - [9.27 Integrating a Custom Layout into Anko](#)
  - [9.28 `settings.gradle` \(Groovy\)](#)
  - [9.29 `settings.gradle.kts` \(Kotlin\)](#)
  - [9.30 `buildscript` Block \(Groovy\)](#)
  - [9.31 `buildscript` Block \(Kotlin\)](#)
  - [9.32 `allprojects` Block \(Groovy and Kotlin\)](#)
  - [9.33 Delete Task \(Groovy\)](#)
  - [9.34 Delete Task \(Kotlin\)](#)
  - [9.35 Applying Plugins \(Kotlin\)](#)
  - [9.36 Android Setup \(Kotlin\)](#)
  - [9.37 Adding Dependencies \(Kotlin\)](#)
  - [9.38 Enabling Experimental Android Extensions \(Kotlin\)](#)
  - [9.39 Gradle Config in `buildSrc`](#)
  - [9.40 `buildSrc/build.gradle.kts`](#)
  - [9.41 Using the Gradle Config](#)
- [10.1 Subtle Difference](#)

## Foreword

“The limits of my language mean the limits of my world.” This quote goes back to the Austrian-British philosopher Ludwig Wittgenstein. With the steadily increasing demand for new software in all the domains that now go digital, it is important that we software developers are efficient and effective. One important prerequisite for this is of course the use of a programming language that best fits the problem.

Java was and still is a dominant language for a variety of domains for over a decade now. It was so successful, originally, because its programs could be downloaded over the internet and executed on the customer’s computer in a secure sandbox, and later because of the virtual machine (VM) that allows the developer to abstract away from the ever-varying nasty technical details. Furthermore, type safety is at the core of the language and is enforced by the VM. The next generation of languages currently build on these achievements.

Kotlin is among the most prominent languages of this next generation. The language itself has a high potential to become, at least in some domains, one of the most important languages of the next decade. This has a number of reasons, both technical and related to marketing.

Kotlin is able to stand out among the currently existing young programming languages due to its rapid growth in popularity, especially since becoming an official language for Android app development backed by Google in May 2017. This gave Kotlin an enormous push and was the reason the discussion in various computer science curricula started to move from the currently used Java to the even more elegant Kotlin.

Kotlin already provides a mighty tool set, allowing the development of industrial-scale software at an enormous pace. In addition, it is based on the Java VM and thus can easily be executed on a wide range of systems. As a consequence, Kotlin is also binary compatible with Java and thus can reuse

Java libraries of all kinds. This immediate availability of libraries is key for a new language.

But foremost, Kotlin is at the same time a strongly typed language and offers efficient language constructs for both object-oriented and higher-order functional programming. Having a strong type system enables developers to prevent a number of tedious errors automatically. This is because a type system guides us to use APIs correctly from the beginning, including autocompletion in editors, quick access to documentation, effective code navigation, and sophisticated code refactoring. However, Kotlin also relaxes this type system in various ways to allow effective development. For example, it provides a type-safe downcast along an inheritance hierarchy (if necessary). One of the best integrated concepts is that an object type by definition does not include “null.” If needed, the “null” value needs to be added explicitly, mimicking an `Optional` type.

Kotlin also provides various language concepts for higher-order functions, thus allowing the switch to a functional programming style for certain parts of the system. The integration of object-oriented and functional programming styles gives us a powerful and effectively programmable language—even though, to ensure appropriate use of these styles, it is important to master both.

In this book, Peter Sommerhoff takes a practical approach to teaching Kotlin by providing a larger set of code listings that demonstrate the discussed language features and by guiding readers through the development of two Android apps step by step. This language introduction covers both Kotlin itself and how to use it on Android. Peter finds a good balance between what is essential and what can be left to readers for their further journey, and thus this book is an efficient yet comprehensible source for starting programming with Kotlin.

Enjoy reading.

—*Bernhard Rumpe*

*Aachen, Germany*

*August 2018*

## Preface

### WHO THIS BOOK IS FOR

This book is for everyone who has at least one year of programming experience and is intended to deliver the basic constructs of a programming language. You do *not* need any prior experience with Kotlin or Java. Some basic familiarity with Android is helpful for the practical part of this book, but it is not required.

This book also assumes you're familiar with some basic terms surrounding software development. The glossary can help if you're unfamiliar with a certain term that is used without explanation.

In short, you fulfill the prerequisites to follow this book if any of the following are true.

- You have a year of programming experience.
- You're a Java or Android developer.
- You're a Kotlin developer who wants to use the language to build Android apps.
- You have attended an introductory programming class at university.

This is by no means an exhaustive list. Most important, you should be eager to learn Kotlin and then put it into practice by building two Android apps.

### How This Book Is Structured

This book first covers Kotlin in detail to teach you all essential language features, and then moves on to using Kotlin in the context of Android. Although Kotlin is introduced comprehensively, this is by no means a language reference but rather a practical book focusing on gaining hands-on experience.

**Part I**, Learning Kotlin, introduces the Kotlin programming language.

- [Chapter 1](#), Introducing Kotlin, gives an overview of Kotlin and motivation to learn this language.

- [Chapter 2](#), Diving into Kotlin, then covers the basic language constructs such as control flow, functions, and exceptions. It also focuses on the underlying principles of Kotlin and how they are reflected in the language design.
- [Chapter 3](#), Functional Programming in Kotlin, shows how Kotlin incorporates functional programming concepts, how you can use them, and what their potential benefits are.
- [Chapter 4](#), Object Orientation in Kotlin, deals with object orientation in Kotlin and interesting features the language provides to write reusable and concise code. It also again highlights selected language design decisions.
- [Chapter 5](#), Interoperability with Java, discusses common issues, solutions, and best practices for interoperability with Java.
- [Chapter 6](#), Concurrency in Kotlin, covers concurrency in Kotlin and is therefore mostly about coroutines, Kotlin's answer to asynchronous and concurrent programming.

[Part II](#), Kotlin on Android, lets you put what you learned in [Part I](#) into practice by building two Android apps with Kotlin.

- In [Chapter 7](#), Android App Development with Kotlin: Kudo App, you will create a simple to-do list app purely in Kotlin while following best practices, such as using Android's Architecture Components, recycler views, and coroutines.
- [Chapter 8](#), Android App Development with Kotlin: Nutrilicious, then guides you through a more complex app that offers nutrition data to users. The data is fetched from a third-party API, mapped to an internal data representation, and then presented to the user. This app introduces more common tools and best practices for Android such as using Retrofit for network calls, splitting up your app into fragments, and using a repository as a single source of truth for data.
- [Chapter 9](#), Kotlin DSLs, covers how to create simple domain-specific languages purely in Kotlin by combining its language features in a clever way. These DSLs can improve code readability and reliability.
- [Chapter 10](#), Migrating to Kotlin, provides guidance for migrating from Java to Kotlin, including how to evaluate whether Kotlin is a good fit, common obstacles, and practices that have helped other companies adopt Kotlin successfully.

## HOW TO FOLLOW THIS BOOK

Unless you already have a strong understanding of Kotlin, you should definitely go through Part I before attempting to build the apps in Part II. You may skip Chapter 5, which covers interoperability with Java, but it is useful to understand what happens under the hood and how Kotlin compiles to Java bytecode. Also, even if you already know Kotlin, you may want to read Chapter 6 if you’re not familiar with coroutines yet.

In Part II, you may skip Chapter 9 covering domain-specific languages (DSLs) because it is not a prerequisite for another chapter. However, DSLs are a popular feature of Kotlin that can improve your code, and it’s a good way to recap interesting language features from Part I. Finally, if you’re not interested in migrating an existing app to Kotlin or adopting Kotlin at your company, you may skip Chapter 10.

For all resources, updates, and news related to this book, please visit its companion website at

<https://kotlinandroidbook.com>

The site hosts runnable and editable versions of all listings in this book for you to work with, lists all related resources and GitHub repositories, and presents any updates and corrections for the book. In short, it’s your companion while reading this book and applying it to your work. The main GitHub repository for this book can also be found directly at [github.com/petersommerhoff/kotlin-for-android-app-development](https://github.com/petersommerhoff/kotlin-for-android-app-development).

## BOOK CONVENTIONS

This book follows several conventions that are used consistently throughout the book.

- Unless stated otherwise, all listings in this book assume they are run as a Kotlin script file (with .kts extension) and therefore don’t use a main function. Chapters 7 and 8 are excluded from this because they assume your code runs on Android.
- In Part I, notes marked as “Java Note” are intended for Java developers and typically compare a concept to its Java counterpart. You can safely

ignore them if you don't know Java.

- I aimed to use all terms in this book accurately and consistently. If you don't know a term, it may be covered in the Glossary.

Register your copy of *Kotlin for Android App Development* on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to [informit.com/register](http://informit.com/register) and log in or create an account. Enter the product ISBN (9780134854199) and click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access any available bonus materials. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

## Acknowledgments

I was very fortunate to have a diverse audience of reviewers for this book who were able to provide feedback from different perspectives—from students to professors, and from programming beginners to senior developers and technical leads.

First and foremost, I want to thank Professor Bernhard Rümpe, who holds the Chair for Software Engineering at RWTH Aachen University, for his continuous support during the creation of this work, which was created in cooperation with his chair. I also want to thank Katrin Hölldobler, Research Assistant at the Chair for Software Engineering, for her valuable feedback on several chapters of the manuscript.

Since the practical part of this book covers Android app development, I also had the great pleasure to work with three Android developers who reviewed this book. Special thanks go to Miguel Castiblanco, Software Architect at Globant LLC, and to Ty Smith, Tech Lead Manager at Uber, for their detailed notes that helped me improve not only the hands-on Android development part but also the rest of the book. I also thank Amanda Hill, Senior Android Engineer at PAX Labs, Inc., for her feedback on several chapters of this book that helped me further improve the draft.

Most of this book covers the Kotlin programming language itself, and I was fortunate to have two JetBrains employees take a look at parts of this book, too. First, I want to thank Hadi Hariri, Developer Advocate at JetBrains, for reviewing an early version of this book’s outline. Second, thanks go to Roman Elizarov, Kotlin Libraries Team Lead at JetBrains, for looking over [Chapter 6](#), which covers concurrency and Kotlin’s coroutines.

Last but not least, I would like to thank my love, Yifan, for her continuous support and her detailed feedback on [Part I](#), which helped me greatly improve its readability and understandability.



## About the Author

**Peter Sommerhoff** is a software developer with a passion for teaching. He's an online teacher and the founder of CodeAlong.TV (<https://www.codealong.tv/>), where he teaches software development and design to students from around the globe. Kotlin was his first topic to teach when he started out in 2016. Now, he's teaching various topics in online courses and live trainings to an audience of over 35,000 motivated learners.

Peter holds a master's degree in computer science from RWTH Aachen University in Germany. Apart from learning and teaching, he enjoys biking tours and playing badminton, as well as cooking for friends and family.

You can follow his activity primarily on Twitter (<https://twitter.com/petersommerhoff>) and on YouTube (<https://www.youtube.com/c/PeterSommerhoff>), where he shares educational content to help you become a better software developer.

|

# Learning Kotlin

# 1

## Introducing Kotlin

Mark Twain *The secret to getting ahead is getting started.*

This chapter provides an overview of the Kotlin programming language, the principles on which it is built, and the benefits it brings compared to Java, with particular focus on writing Android apps. We will also introduce the ecosystem and community around Kotlin and online resources for the language.

### WHAT IS KOTLIN?

Kotlin is a statically typed programming language that is completely open source and free to use. Kotlin code can compile down to Java bytecode and thus can run on the Java virtual machine (JVM) and on Android.<sup>1</sup> Apart from this, it can be compiled down to JavaScript<sup>2</sup> and can even run on embedded devices and iOS<sup>3</sup> devices. The big picture is to be able to target all these platforms with a single language and share parts of your code base between them. In this book, we focus only on Kotlin targeting Java bytecode, particularly on Android.

1. <https://developer.android.com/>

2. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/>

3. <https://developer.apple.com/ios/>

Kotlin was developed by JetBrains,<sup>4</sup> the Czech software tool company that develops the IntelliJ integrated development environment (IDE), among others. Kotlin was designed to be completely interoperable with Java, meaning that Kotlin code

can use any Java classes and libraries and vice versa. Even though Kotlin is very similar to Java in many regards, its language design is much cleaner and uses the expertise that was gained in software development and programming language design since the release of the first version of Java in 1996. For instance, checked exceptions have shown drawbacks for large-scale software (so Kotlin only has unchecked exceptions), Java tends to require much boilerplate code (so Kotlin avoids this), and inheritance comes at the cost of very tight coupling (so Kotlin classes are closed for inheritance by default). In short, Kotlin does not carry much legacy, which allows for a clean language that incorporates best practices from the start.

4. <https://www.jetbrains.com/>

## GOALS AND LANGUAGE CONCEPTS

The main goal of Kotlin is to provide a pragmatic, concise, interoperable, and safe programming language.

- **Pragmatic** means that Kotlin is built to develop real-world enterprise software. The language design aims to address common issues of large-scale software development and continuously incorporates industry feedback.
  - **Concise** code is supported by several concepts including type inference, data classes, and lambda expressions. Conciseness can greatly improve readability and thus maintainability of a code base, which is essential because code is read much more frequently than it is written.
  - **Interoperability** with Java is indispensable for Kotlin in order to be usable wherever Java is used today. This includes Android, server-side code, backend development, and desktop applications. Kotlin itself also makes heavy use of this by reusing and extending the Java standard library, such as its Collection application programming interface (API).<sup>5</sup> Similarly, it interoperates with JavaScript when targeting that.
5. An API is a set of well-defined interfaces for reuse.
- **Safe** refers to the fact that Kotlin inherently prevents many software errors by its language design. This is achieved by enforcing several best practices, such as explicitly designing nonfinal classes for inheritance, and most prominently by providing null safety. In Kotlin, each type is either explicitly nullable or can never be `null`, which greatly helps prevent `NullPointerExceptions`. When interoperating with Java, extra care must be taken to avoid them.

What differentiates Kotlin from many other modern JVM languages such as Scala,<sup>6</sup> Ceylon,<sup>7</sup> and Clojure<sup>8</sup> is that it focuses on enterprise software and is not an academic endeavor. Instead, it is developed and maintained by JetBrains, who make heavy use of Kotlin themselves to develop their stack of IDEs and other tools.

6. <https://scala-lang.org/>
7. <https://www.ceylon-lang.org/>
8. <https://clojure.org/>

Additionally, when compared to a language like Scala, Kotlin is more lightweight because it is intentionally constrained. There are several reasons for this. First, this improves toolability because fewer language concepts need to be supported by the compiler, IDEs, and other tools. Second, it increases simplicity, thus making the language easier to use and code easier to read for developers. Third, solution paths are constrained, thus making it clearer how to solve a particular problem.

## WHY USE KOTLIN ON ANDROID?

At the Google I/O 2017 developer conference, Google<sup>9</sup> announced official support for Kotlin as a language for Android app development; it joins Java and C++. This decision was certainly motivated by Google's lawsuit with Oracle over patents on the Java APIs, but this section focuses on why Kotlin may be preferable to Java from a language perspective.

9. [https://www.google.com/intl/en\\_de/about/](https://www.google.com/intl/en_de/about/)

Kotlin can run on the JVM and is fully compatible with Java 6, which makes it a viable language for writing Android apps. Due to its similarity with Java, Android developers can learn Kotlin's syntax and understand its semantics quickly. This not only means that the effort involved in switching to Kotlin is comparably low, it also means that it entails less risk. Performance-wise, using Kotlin instead of Java comes at no additional cost; the bytecode is executed just as fast as any

Java bytecode. Additionally, the Kotlin runtime is fairly small, so it does not add much weight to the application.

## Java on Android

To understand how Kotlin fits into the Java and Android ecosystems, we briefly recap the history and current situation for Android app development with Java. We focus on Java 6 and above because that is what is currently relevant for Android.

Java 6 already supports many crucial language features it inherits from Java 5, including generic types, enumeration types, annotations, variable-argument parameters, for-each loops, and static imports. Additionally, Java 6 itself adds significant performance improvements in the core platform, the compiler, and synchronized code. Generally, each language update entails performance improvements, a more robust core platform, and various language enhancements. Therefore, adopting a newer Java version typically improves productivity and allows writing more concise and maintainable code.

Unfortunately, this is not always possible if a target platform (or Android device in this case) is not compatible with the desired Java version, which is why Android developers still have to rely mostly on Java 7 at the time of writing.

In order to support at least 96% of Android devices today, you have to target Android 4.4 (API level 19). To reach at least 98% support, Android 4.2 (API level 17) must be targeted; and to support 100% of devices, it is necessary to target Android 2.3.3 (API level 10) from 2010. You can find the latest numbers in Android's Distribution Dashboard.<sup>10</sup>

10. <https://developer.android.com/about/dashboards/>

On any device running Android 4.4 or lower, Android applications are run on the Dalvik virtual machine, which is part of the operating system. Java bytecode is not executed directly but first translated to a Dalvik executable. Since Android 4.4, Dalvik has been replaced by the Android Runtime (ART), which is the only included runtime in Android versions 5.0 and above. Apart from the runtime, which Java features are available also depends on which

toolchain is used. For instance, the D8 dexer that replaced DX in Android Studio 3.1 allows you to use some Java 8 features such as lambda expressions. For these two reasons (runtime and toolchain), there is no direct mapping from Android versions to supported Java versions; each language feature may or may not be supported by ART or your toolchain. Over time, additional Java language features become incorporated into Android but with a significant delay. With Kotlin, you can have language features even beyond those from Java 8 today.

### **Kotlin on Android**

When developing apps for Android nowadays, you are mostly tied to Java 7. With Android Studio 3.0 and later, all Java 7 features and several Java 8 features, most prominently lambda expressions, method references, and default interface methods, are available for all API levels. Still, the more devices you want to support, the fewer language features are available and the harder it becomes to write maintainable and high-quality code. In addition, you will always have to wait for new Java features to become supported on Android. For instance, Java 8 *Streams* only work with API level 24+ at the time of writing, whereas Kotlin's equivalent *Sequences* work irrespective of API level.

With Kotlin, you are not tied to any Java version, and you can use all features developers are longing for in Java 8—plus many more. Functional-style programming with higher-order functions and lambda expressions is incorporated from the start in Kotlin. This leads to several syntactic advantages in the language design when compared with lambdas in Java. Powerful APIs for collections and I/O operations in Kotlin supersede what you could achieve with Streams in Java 8, and default interface methods are a part of Kotlin as well.

## KOTLIN VERSUS JAVA 8

Java 8 was a huge language update that really changed the way you write code in Java. This is not the case to that extent for every Java language update. For instance, Java 9 mainly changes the way developers package and deploy their code (due to the module system) but not so much how they think about problems and write code. As you may have noticed from the language concepts discussed above, Kotlin has several useful language features that Java 8—and even Java 9, 10, or 11—do not provide. This includes nullable types to provide null safety, comprehensive type inference, extension functions, smart casts, named and default parameters, and more. Most of these can make your code more concise and expressive at the same time; features like these will be introduced in [Part I](#) of this book.

With Kotlin, you have a strong standard library at your disposal in addition to the Java standard library, which is also available for use. The Kotlin standard library encompasses many functional-style methods to work with collections and I/O streams, many useful predefined extension functions, and a variety of third-party libraries to simplify working with JavaFX,<sup>11</sup> Android,<sup>12</sup> databases,<sup>13</sup> and more. In this book, you will learn how to use the Anko library for Android app development.

11. TornadoFX, <https://github.com/edvin/tornadofx>

12. Anko, <https://github.com/Kotlin/anko>

13. Exposed, <https://github.com/JetBrains/Exposed>

## TOOL SUPPORT AND COMMUNITY

Kotlin is designed to be highly toolable from its roots. For instance, static typing enables sophisticated refactorings, compile-time checks, and autocompletions. It is not surprising that toolability is not an afterthought in Kotlin, considering that JetBrains is a tool company. IDEs that have strong support for Kotlin include IntelliJ IDEA, Android Studio, Eclipse, and NetBeans. They facilitate writing idiomatic Kotlin code by suggesting better ways to rewrite parts of the code, among other things.

Still, myriad tools, libraries, and frameworks have been developed for Java over the last decades, and not all of them work with Kotlin as smoothly as they do with Java, such as static analysis tools and linters, the build tool Buck,<sup>14</sup> or the mocking library Mockito.<sup>15</sup> Fortunately, most existing tools and libraries work smoothly with Kotlin, for instance Gradle,<sup>16</sup> Retrofit,<sup>17</sup> and Android architecture components. On Android specifically, Google is pushing Kotlin-specific APIs, such as Android KTX,<sup>18</sup> that help developers write code more effectively.

14. <https://buckbuild.com/>

15. <https://site.mockito.org/>

16. <https://gradle.org/>

17. <http://square.github.io/retrofit/>

18. <https://developer.android.com/kotlin/ktx>

There is an active community around Kotlin that acts via various channels in order to give feedback to the Kotlin team and help shape the way the language evolves. The Slack channel<sup>19</sup> has very active discussions and is a good place to ask your questions while learning Kotlin. There is also a discussion forum<sup>20</sup> that often features more advanced discussions concerning language design decisions and the future direction of Kotlin. Among others, there is also a Reddit subreddit specifically for Kotlin<sup>21</sup> as well as various talks about the language. An overview of all community resources is available on the Kotlin website.<sup>22</sup>

19. <http://slack.kotlinlang.org/>

20. <https://discuss.kotlinlang.org>
21. <https://www.reddit.com/r/Kotlin/>
22. <https://kotlinlang.org/community/>

## BUSINESS PERSPECTIVE

Because the intent is for Kotlin to be usable wherever Java runs, it is important to make the transition easy. Indeed, the effort involved in learning Kotlin is comparably low for developers using Java or a similar language because many language concepts are the same or very similar, such as classes, interfaces, and generics. However, other language concepts may not be familiar to developers using a language without functional programming features, such as lambda expressions and streams. Lastly, some language concepts are not known from most other languages, such as non-nullable types and extension functions.

This is one of the common reasons why adopting a new language always introduces a certain risk. You have to expect a productivity slowdown at least in the beginning of introducing a new language before productivity can finally increase. During this transition time, businesses must also provide the time and resources necessary for the training that developers will need.

Additionally, adoption of a new language is inherently hard to reverse. If you decide to make the transition from Java by converting several Java files from your flagship product to Kotlin and make changes to that code, it will take time to rewrite (or decompile and refactor) this code back to Java. Ideally, try out Kotlin in a noncritical environment such as a pet project to familiarize yourself with the syntax and tooling without introducing unnecessary risk. [Chapter 10](#), Migrating to Kotlin, discusses best practices for adopting Kotlin in detail.

Tooling for Kotlin includes IntelliJ IDEA and Android Studio, where the Kotlin plugin comes with a Java-to-Kotlin converter, which can be used for two main purposes. First, it allows the developer to move a Java code base to Kotlin faster. However, the converted code will typically need manual changes to be free of errors and more extensive work to

resemble idiomatic Kotlin code. Second, the converter can be used for learning purposes. Since developers understand their Java code, they can compare that to the Kotlin code generated in order to learn how the different concepts and language elements are mapped to one another. This is another reason why Java developers can learn Kotlin fairly quickly.

Remember, you may want to keep a copy of the original Java file.

Once developers are familiar with the Kotlin language, performance can potentially increase significantly due to the possibility to express more with less code and the fact that Kotlin enforces many best practices by design. For instance, correctly using nullable types increases robustness of the code by preventing `NullPointerExceptions`. Similarly, classes are closed for inheritance by default, thus enforcing the principle to explicitly design for inheritance before allowing it. In total, this leads to a code base that is less prone to errors and costs less to maintain.

## WHO'S USING KOTLIN?

Hundreds of companies have already incorporated Kotlin into their technology stack, from young startups to large corporations. Besides JetBrains and Google, who obviously use them intensively, these include Pinterest,<sup>23</sup> Slack,<sup>24</sup> Uber,<sup>25</sup> Netflix,<sup>26</sup> WordPress,<sup>27</sup> Udacity,<sup>28</sup> Evernote,<sup>29</sup> Lyft,<sup>30</sup> and Trello.<sup>31</sup> Developers from many of these contribute to the Kotlin community by sharing their experiences with the language and what they found to be best practices.

23. <https://www.pinterest.com/>

24. <https://slack.com/>

25. <https://www.uber.com/>

26. <https://www.netflix.com/>

27. <https://wordpress.com/>

28. <https://udacity.com>

29. <https://evernote.com/>

30. <https://www.lyft.com/>

31. <https://trello.com/>

Although Kotlin gained popularity through being an official language for Android development, its goal is to be deployable virtually everywhere. There are three compilation targets of the language that indicate the platforms on which Kotlin can run:

- **Kotlin/JVM**, which targets the JVM and Android by compiling Kotlin code to Java bytecode. This allows Kotlin to be used anywhere Java is used, including in the backend, on the server, on Android, or for desktop applications. This book covers Kotlin/JVM, with a particular focus on Android, but the chapters introducing Kotlin itself are useful irrespective of the platform on which you want to use it.
- **Kotlin/JS**, where Kotlin compiles to JavaScript code to target web applications and anything that runs in the browser. Here, Kotlin interoperates with JavaScript similarly to how it interoperates with Java on the JVM.
- **Kotlin/Native**, which allows Kotlin to compile to native binaries that don't require any virtual machine like the JVM. This enables the use of Kotlin to develop iOS apps and embedded systems in general, and it interoperates with other native code as well.

This is the big picture the Kotlin team is aiming for. Work needs to be done in all of these branches, especially Kotlin/Native. Still, Kotlin is a stable language that already offers many benefits over languages like Java, JavaScript, or C.<sup>32</sup> This book focuses on its benefits for Android app development.

32. C Programming Language, <http://www.bell-labs.com/usr/dmr/www/chist.html>

## SUMMARY

Being a statically typed language with object-oriented as well as functional language elements, Kotlin is similar to Java 8 in many regards. However, additional language features allow the avoidance of lots of boilerplate code, thus increasing conciseness and readability. Kotlin is not an academic project and does not try to invent anything new, but instead combines what is known to work well to solve real problems in large-scale software development.

Kotlin runs on the JVM and can be used anywhere Java is used today without losing performance or increasing the runtime significantly. Its tool support and focused libraries, such as the Anko library for Android, facilitate many development tasks.

Java and Android developers can typically get up to speed with Kotlin quickly and then use all mentioned features that go beyond Java 8, 9, 10, or even 11.

## Diving into Kotlin

*When you catch a glimpse of your potential, that's when passion is born.*

Zig Ziglar

In this chapter, you'll familiarize yourself with the syntax, features, and underlying concepts of the Kotlin programming language. Apart from basics like data types and control flow, you'll learn how to avoid `null` in Kotlin, how to create different types of functions, how to check for equality of two variables, and how to handle exceptions.

### KOTLIN REPL

To try out simple code snippets, Kotlin provides a read-eval-print-loop (REPL). The REPL evaluates a given piece of code directly and prints its result. This is useful to quickly try out how to use a language feature or a third-party library or to share code snippets with colleagues. Thus, it speeds up the feedback loop and helps learn the language faster.

With the Kotlin plugin activated, every IDE should contain the Kotlin REPL. In Android Studio and IntelliJ, you can launch it from the menu under *Tools*, then *Kotlin*, and then *Kotlin REPL*. Alternately, you can create Kotlin script files (with `.kts` as file extension) in an IntelliJ project<sup>1</sup> to run the code examples from this and the following chapters. The REPL and script files allow you to run your code without a main function. Note that you have to adjust the file extension

manually to `.kts` if you create a regular Kotlin file with `.kt` extension. Lastly, you can use the online editor on the Kotlin website,<sup>2</sup> but you must then write your code inside a main function.

1. Create a Kotlin/JVM project via *File, New, Project...*, and choose *Kotlin/JVM*.
2. <https://try.kotlinlang.org/>

**Note**

All listings in this book are written as if they were inside a Kotlin script file, ending in `.kts` as opposed to `.kt`. For the sake of brevity, some listings only illustrate a point but may require additional code to actually run. But most are runnable as a script as shown.

In any case,<sup>3</sup> a runnable full-fledged version is available from the GitHub repository.<sup>4</sup> Additionally, the book's companion website allows you to run all listings directly inside your browser.<sup>5</sup>

3. <https://github.com/petersommerhoff/kotlin-for-android-app-development>
4. <https://www.kotlinandroidbook.com/listings>

## VARIABLES AND DATA TYPES

This section covers variable declarations, basic data types, how they are mapped to Java types in the bytecode, and how you can use type inference to write more concise code.

### Variable Declarations

In Kotlin, there are two keywords to declare a variable. The first is called **var** and creates a mutable variable, meaning that you can reassign the variable to a new value any number of times, as illustrated in Listing 2.1.

#### Listing 2.1 Declaring Mutable Variables

[Click here to view code image](#)

```
var mercury: String = "Mercury" // Declares a mutable variable
mercury = "Venus" // Can be reassigned
```

In Kotlin, the **var** keyword stands in front of the variable name, followed by a colon, the variable type, and the assignment. You'll learn to write this more concisely later using type inference.

Kotlin promotes the use of immutability wherever appropriate, so you should use the **val** keyword whenever possible to create read-only variables as in Listing 2.2.

#### **Listing 2.2 Declaring Read-Only Variables**

[Click here to view code image](#)

```
val mercury: String = "Mercury" // Declares a read-only variable

mercury = "Venus" // Compile-time error
```

This way, the variable cannot be reassigned after its declaration, thus the compile-time error when trying to reassign `mercury` to a different value. For local variables like this, you can also split up declaration and initialization. In any case, once a read-only variable is assigned a value, it cannot be reassigned.

##### **Note**

Whether the variable itself is mutable (can be reassigned) or read-only (can only be assigned once) has nothing to do with whether the object *stored inside* the variable is mutable. For instance, Listing 2.1 presents a mutable reference to an immutable string object.

Similarly, you could have a `val` referencing a mutable object. Thus, just using `val` is *not* enough to take advantage of immutability in your code—the referenced object must also be immutable.

## **Immutability Matters**

Kotlin promotes immutability both in the way the language is designed and in its communication of coding conventions. This is because Kotlin focuses on building large software systems by bringing together best practices that proved beneficial in the industry—one of which is immutability.

Immutable variables eliminate many common sources of errors, especially in multithreaded code such as on Android, because side effects and unwanted data manipulations are not possible for immutable variables, leading to more robust code that is also easier to reason about. Immutability is covered in detail in Chapter 3, Functional Programming in Kotlin,

because it is one of the underlying principles of functional programming.

### **Basic Data Types**

The basic data types in Kotlin are similar to those from other languages. They are not considered primitive types because they are objects at runtime. An overview of basic types in Kotlin, their memory usage, and their range is given in [Table 2.1](#).

Table 2.1 Data Types in Kotlin

Data Type	Bits	Range
Bool	1	{ true, false }
Byte	8	-128 .. 127
Short	16	-32768 .. 32767
Int	32	-2147483648 .. 2147483647
Long	64	-18446744073709551616 .. 18446744073709551615
Floating	32	1.4e-45 .. 3.4028235e38
Double	64	4.9e-324 .. 1.7976931348623157e308
Char	16	16-bit Unicode characters
String	16 * length	Unicode strings

All number types represent signed numbers<sup>5</sup> meaning the range includes negative and positive numbers. Also, all strings are immutable, so if a mutable string variable gets reassigned, a new `String` object is created.

5. Kotlin 1.3 introduced additional unsigned integer types like `UShort`, `UInt`, and `ULong`.

Keep in mind that floating point numbers invariably have limited precision. For instance, while `0.000000001` –

`0.00000000005` evaluates correctly to `9.5E-10`, an additional zero causes a result of `9.500000000000001E-11`. Thus, never compare `Float` and `Double` values for equality but rather compare with an adequate tolerance. Also, treat computation results with a grain of salt as imprecisions may have led to numerical errors.

## Mapping of Basic Types

When using Kotlin for Android development, the code will end up as Java bytecode. Thus, it is important to understand how the basic types are mapped between Kotlin and Java.

- When receiving data from Java, primitive Java types will be mapped to their equivalent Kotlin type at compile-time, for example `char` to `kotlin.Char` and `int` to `kotlin.Int`. This allows you to work with primitive values coming from Java the same way as with Kotlin's own types.
- Conversely, at runtime, Kotlin's basic types are mapped to their Java counterparts, for instance `kotlin.Int` to `int` and `kotlin.Double` to `double`. This is possible because Kotlin's types are by default non-nullable, meaning they can never be `null`. Hence, they can always be mapped to a primitive Java type instead of a boxed type such as `java.lang.Integer`.

There are more rules for mapped types that are introduced once you know more of Kotlin's language features and libraries.

## Type Inference

One of the language features of Kotlin that is used throughout idiomatic code is type inference. This allows you to skip the type in variable declarations whenever the Kotlin compiler can infer it (which it often can). Thus, you can declare variables as in [Listing 2.3](#).

### Listing 2.3 Type Inference

[Click here to view code image](#)

---

```
val mercury = "Mercury"      // Inferred type: String
val maxSurfaceTempInK = 700   // Inferred type: Int
val radiusInKm = 2439.7      // Inferred type: Double
```



---

The first statement is semantically equivalent to the variable declaration in [Listing 2.2](#), where the type `String` was specified explicitly. With type inference, you are able to write more concise code that focuses on what is relevant—and in cases where the type may not be clear to the reader this way, you can still explicitly specify it. There is also an option in Android Studio to show inferred types, which is especially useful when starting out.

Note that in Kotlin, you cannot pass an `Int` into a function that expects a `Float` or `Double` as you can do with primitive types in Java. Kotlin does not automatically convert between these types. However, the standard library has helpers like `toInt`, `toDouble` and so forth that you can use for these conversions.

**Java Note**

Java 10 introduced local-variable type inference with Java Development Kit (JDK) Enhancement Proposal (JEP) 286.<sup>6</sup> Before that, Java only supported type inference for generic type parameters with the diamond operator (since Java 7).<sup>7</sup>

6. <http://openjdk.java.net/jeps/286>

7. <https://docs.oracle.com/javase/7/docs/technotes/guides/language/type-inference-generic-instance-creation.html>

## CONDITIONAL CODE

In Kotlin, `if` and `when` are used for conditional control flow. Both are expressions, meaning that they have a value. One consequence is that they can be assigned to a variable. But you can also ignore their value, effectively using them like statements (which don't carry a value).

## If and When as Statements

For conditional control flow, Kotlin provides the keywords **if** and **when**. These allow running certain parts of your code only if a given condition is fulfilled. The syntax for **if** conditions is the same as in languages like C or Java, and the **when** keyword is similar to **switch** in these languages, but more powerful.

Conditions with **if** can be written as in [Listing 2.4](#).

**Listing 2.4** Conditions with **if**

[Click here to view code image](#)

```
if (mercury == "Mercury") {  
  
    println("Universe is intact.")  
  
} else if (mercury == "mercury") {  
  
    println("Still all right.")  
  
} else {  
  
    println("Universe out of order.")  
  
}
```

The **else-if** and **else** branches are optional. Conditions can be composed using `&&` (“and”) and `||` (“or”), and `!` (“not”) can be used to negate a condition. If a condition is composed of multiple atomic conditions using `&&` or `||`, each of them is evaluated lazily. In the condition `a && b` for instance, the second atomic condition `b` is not evaluated if `a` is already evaluated to **false** because then the whole condition must be **false**. This is called *short-circuiting* and aims to improve performance.

In addition to **if**, Kotlin also features powerful **when** conditions. These can be used to define different behavior for a given set of distinct values. In this way, **when** conditions are a more concise alternative to writing cascades of conditions. Thus, you can replace the **if** condition from [Listing 2.4](#) with a

**when** condition as in Listing 2.5 to save two lines of code (~29%) compared with cascading **else-if** branches.

**Listing 2.5 when for Cascades of Conditions**

[Click here to view code image](#)

```
when (mercury) {  
    "Mercury" -> println("Universe is intact.")  
    "mercury" -> println("Still all right.")  
    else -> println("Universe out of order.")  
}
```

First, the variable to test against is passed into parentheses and the **when** block is enclosed by curly braces, followed by each case with a condition on the left-hand side and its associated code on the right-hand side, separated by an arrow (->). If the left-hand value matches the variable value, its corresponding right-hand side is executed and no further cases are probed.

Although similar to **switch** from other languages, this is more powerful because it can perform more types of checks, as shown in Listing 2.6.

**Listing 2.6 Types of Conditions for when**

[Click here to view code image](#)

```
when(maxSurfaceTempInK) {  
    700          -> println("This is Mercury's maxi  
    0, 1, 2       -> println("It's as cold as it get  
    in 300..699    -> println("This temperature is al  
    !in 0..300     -> println("This is pretty hot")  
    earthSurfaceTemp() -> println("This is earth's aver  
    is Int         -> println("Given variable is of  
    else           -> {  
        // You can also use blocks of code on the right-  
    }
```

```
    println("Default case")  
}  
}  
}
```

This demonstrates that you can combine different kinds of checks inside the same **when** condition. Let us go through these one by one.

1. The first case inside the **when** block checks whether `maxSurfaceTempInK` equals 700.
2. The second case checks if at least one of several comma-separated values matches the variable. If so, the right-hand side is executed.
3. The third condition checks if the given variable is in the range of numbers from 300 to 699 (both bounds inclusive) using the **in** keyword. The range is created using Kotlin's shorthand operator `n..m` that internally calls `n.rangeTo(m)`, which returns an `IntRange`. In Android Studio, you can go to the declaration of such operators just like any other function to see what it does.
4. The fourth condition is similar but checks whether the variable is *not* in a given range.
5. In the fifth case, you see that it is also possible to call functions on the left-hand side to specify the value against which to test. If the function would return an iterable object or a range, you can use it with **in**, for instance `in earthSurfaceTempRange() -> ...`
6. The sixth condition checks the variable type, in this case whether it's an `Int`. Type checking is discussed in more detail in [Chapter 4, Object Orientation in Kotlin](#).
7. Lastly, the **else** keyword is used to define the default case that is executed only if none of the previous conditions could be evaluated to `true`. As you can see, it is possible to define a larger code block on the right-hand side for each case by using curly braces, as demonstrated in [Listing 2.6](#). If you do, the last line then defines the value of the **when** expression, as explained in the following section covering conditional expressions.

**Note**

Notice that there are no `break` statements; after the first case matches, its right-hand side is executed and no more subsequent cases are checked. Thus, the code behaves as if there was an implicit `break` statement included at the end of each case, preventing the common mistake of forgetting a `break` statement.

To replace arbitrary cascades of conditions, you can omit the variables in parentheses and then use any Boolean expressions on the left-hand sides, as in [Listing 2.7](#).

#### Listing 2.7 Arbitrary Boolean Conditions with when

[Click here to view code image](#)

```
when {  
    age < 18 && hasAccess -> println("False positive")  
    age > 21 && !hasAccess -> println("False negative")  
    else -> println("All working as expected")  
}
```

## Conditional Expressions

In Kotlin, both **if** and **when** are expressions, meaning that they have a value. If that value is not used, they are equivalent to statements, which carry no value. However, using them as expressions is a powerful feature that can, for instance, be used to avoid **null** in some cases.

In both **if** and **when** expressions, the value of each case (and therefore the whole expression) is defined by the last line in the corresponding code block. Thus, [Listing 2.5](#) can be rewritten to use **if** as an expression as in [Listing 2.8](#).

#### Listing 2.8 If Expression

[Click here to view code image](#)

```
val status = if (mercury == "Mercury") { // 'if' is  
    "Universe is intact" // Expression value in case  
} else {  
    "Universe out of order" // Expression value in case  
}
```

Here, the **if** expression's value is assigned to a **status** variable. The **println** statements are removed in order to return the string value instead of printing it right away. The **else-if** branch is removed for the sake of brevity but any number of **else-if** branches can be used. Notice the use of

type inference by leaving out the type declaration of `status`. In case the `if` block and `else` block have different types, Kotlin currently infers `Any` as the overall type of the `if` expression. `Any` is the superclass of all other (non-nullable) classes in Kotlin.

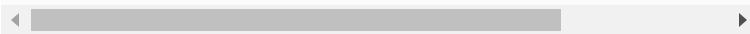
Because `if` is an expression in Kotlin, there is *no* extra ternary conditional operator (`result = condition ? thenThis : elseThis`) because `if` expressions fulfill this purpose. Consider [Listing 2.9](#) as a more concise version of [Listing 2.8](#).

#### **Listing 2.9 Ternary Conditional Operator with if**

[Click here to view code image](#)

---

```
val status = if (mercury == "Mercury") "Intact" else
```



**Note**

The above way to use `if` as a ternary conditional operator uses the fact that curly braces are optional for a branch if it only consists of a single expression, plus the fact that line breaks are also optional. It works the same way when introducing line breaks or adding curly braces.

One conclusion of this is that this structure may also be used with an arbitrary number of `else-if` branches. However, this is uncommon because it quickly becomes unreadable.

**When** expressions are very similar: The last line in each case block defines the corresponding value of the `when` expression if that block is executed. For instance, you can rewrite [Listing 2.6](#) to use strings on the right-hand sides so that the `when` expression has a string value that can be assigned to a variable, as shown in [Listing 2.10](#).

#### **Listing 2.10 Using when as an Expression**

[Click here to view code image](#)

---

```
val temperatureDescription = when(maxSurfaceTempInK)  
    700 -> "This is Mercury's maximum surface temperatu  
    // ...  
    else -> {
```

```
// More code...

"Default case" // Expression value if 'else' cas
}

}

◀ ▶
```

Here, the expression value for each case can either be directly defined on the right-hand side if no additional code should be executed, or it can be defined in the last line of a code block as in the **else** branch.

## LOOPS AND RANGES

For repetitive control flow, Kotlin provides **for** loops, **while** loops, and **do-while** loops. **While** and **do-while** loops work the same way as in Java, whereas **for** loops are similar to foreach loops known from many other languages.

### While Loops

**While** loops and **do-while** loops are used to repeat a block of code as long as a given condition is true. The condition is defined in parentheses following the **while** keyword, as portrayed in Listing 2.11.

**Listing 2.11** while Loop

[Click here to view code image](#)

---

```
val number = 42

var approxSqrt = 1.0

var error = 1.0

while (error > 0.0001) { // Repeats code block until
    approxSqrt = 0.5 * (approxSqrt + number / approxSqr
    error = Math.abs((number - approxSqrt*approxSqrt) /
}

}
```

In the first example, the **while** loop keeps repeating its block of code as long as its condition is **true**, that is, as long as the error is still greater than **0.0001**. The code approximates the square root of a given number with the Babylonian method up to a tolerance of **0.0001**. Note that **java.util.Math** can be used seamlessly.

The point of the **do-while** loop is that it executes its code block at least once initially, and only then checks its condition. Listing 2.12 demonstrates a common use case for this type of loop.

#### **Listing 2.12 do-while Loop**

[Click here to view code image](#)

```
do {  
  
    val command = readLine()      // Reads user command  
  
    // Handle command...  
  
} while (command != ":quit") // Repeats code block until user quits
```

This reads user input, for example in a command-line tool, until the user wants to quit. Generally, **do-while** loops are used if you want to guarantee at least one initial iteration.

## **For Loops**

The **for** loop in Kotlin resembles foreach loops known from other languages and can be used to iterate over any type that provides an iterator. Most prominently, this includes ranges, collections, and strings, as demonstrated in Listing 2.13.

#### **Listing 2.13 for Loops**

[Click here to view code image](#)

```
for (i in 1..100) println(i)      // Iterates over a range  
  
for (i in 1 until 100) println(i) // Iterates over a range, not including 100
```

```
for (planet in planets)           // Iterates over a collection

    println(planet)

for (character in "Mercury") {     // Iterates over a string

    println("$character, ")

}
```

The first line uses the syntax `1..100` to create an `IntRange` to iterate over, as you saw in `when` expressions. Thus, the first loop prints the elements from 1 to 100. More versatile uses of ranges in `for` loops are discussed in the next subsection. The second example uses `until` to create a range where the upper bound is exclusive. The third loop iterates over a collection of planets; collections are discussed in detail in [Chapter 3](#). Lastly, you can use the `for` loop to iterate over strings character by character.

Note that the last `for` loop shows another interesting feature of Kotlin, called *string interpolation*. This means that you can insert the value of a variable into a string by prefixing the variable with a \$ sign as in `$character`. More complex expressions must be separated from the surrounding string by curly braces, for instance `"Letter: ${character.toUpperCase()}"`.

## Using Ranges in for Loops

After the initial example of `for` loops with ranges, you may be wondering how to iterate over more complex structures of elements, such as the numbers from 100 down to 1, maybe even in steps of 5. Kotlin provides the helper functions `downTo` and `step` to easily create such ranges, as shown in [Listing 2.14](#).

**Listing 2.14 Ranges in for Loops**

[Click here to view code image](#)

```
for (i in 100 downTo 1) println(i)          // 100, 99  
for (i in 1..10 step 2) println(i)          // 1, 3, 5  
for (i in 100 downTo 1 step 5) println(i)  // 100, 95
```



The syntax here is interesting because it uses infix functions. Calling `100 downTo 1` is equivalent to calling `100.downTo(1)`. This is possible because `downTo` is declared as an infix function, so that the function can stand between its two arguments. Similarly, `1..10 step 2` is equivalent to `(1..10).step(2)` and the third range could be rewritten as `100.downTo(1).step(5)`. Infix functions thus allow writing more readable code with less clutter from parentheses. Notice that infix functions must have two parameters. The next section covers how to create and work with your own functions, including infix functions.

## FUNCTIONS

Functions are a powerful language feature in Kotlin. This section starts with the basics of creating and calling functions, and then shows how to use default values for parameters and how to define special types of functions such as extension functions, infix functions, and operator functions.

### Function Signatures

A function's signature is the part of a function declaration that defines its name, parameters, and return value. Together with a function body, it defines the function completely. In Kotlin, a function declaration is denoted as in [Listing 2.15](#).

[Listing 2.15 Declaring a Function](#)

[Click here to view code image](#)

```
fun fib(n: Int): Long {
```



```
return if (n < 2) {  
    1  
} else {  
    fib(n-1) + fib(n-2) // Calls 'fib' recursively  
}  
}
```

This is an (inefficient) implementation of the well-known Fibonacci sequence. Here, `fun fib(n: Int): Long` is the function signature. It begins with the `fun` keyword used to define functions in Kotlin, followed by the function name, its parameters in parentheses, a colon, and the return value. Notice the consistency with variable declarations in Kotlin, where the type comes after the name as well.

In the function body, an `if` expression is used instead of placing a `return` in each branch. This is not necessary but would allow you to write the function body in a single line by using it like a ternary conditional operator. You'll see this in the next subsection.

The `fib` function can be called by writing the function name with its arguments in parentheses, as in [Listing 2.16](#). This would also work if you declared `fib` inside a regular Kotlin file (not a script file) but outside of a main function. Then the `fib` function is a *top-level* function—meaning that it is declared directly on *file level*, not inside any other declaration (such as a class or another function). In this book, the terms *top-level* and *file-level* are used synonymously.

#### [Listing 2.16 Calling a Function](#)

[Click here to view code image](#)

---

```
val fib = fib(7) // Assigns value 21L ('L' indicates
```

The value of the function call is stored in a variable called `fib` that implicitly has the type `Long`. Notice that the variable

and function names do not clash because their use can always be differentiated by the compiler due to the parentheses used in function calls.

**Note**

For this section, keep in mind that the term *parameter* refers to what is used at the declarationsite as part of the function signature, whereas *arguments* are the actual values used at the call-site. So, in the declaration of `fib` in [Listing 2.15](#), `n` is a parameter of type `Int` and `7` is the corresponding argument in [Listing 2.16](#).

## Shorthand for Single-Expression Functions

For single-expression functions, Kotlin provides a shorthand that looks almost like a variable declaration.

Let's use this to simplify the `fib` function in two steps.

[Listing 2.17](#) first reduces the expression in the function body to a single expression (and a single line).

### [Listing 2.17 Transforming the Function Body into a Single Expression](#)

[Click here to view code image](#)

```
fun fib(n: Int): Long {  
    return if (n < 2) 1 else fib(n-1) + fib(n-2) // Use  
}  
◀ ▶
```

With this, you can remove the `return` and directly assign the expression as the value of the function, as in [Listing 2.18](#).

### [Listing 2.18 Shorthand Function Notation Using Equals Sign](#)

[Click here to view code image](#)

```
fun fib(n: Int): Long = if (n < 2) 1 else fib(n-1) +  
◀ ▶
```

Here, `fib` uses Kotlin's shorthand notation by defining the function body using an equals sign. This works for functions that contain only a single expression because you can then assign the expression value as the value of the function. The explicit return type is still required here because type inference would run into infinite recursion. But in general, you can make

use of type inference for the return type when using this shorthand notation.

## Main Function

The special main function defines the entry point into a program. The easiest way to create one is to use a top-level function as in [Listing 2.19](#). This must be done in a regular Kotlin file, not a script file.

### [Listing 2.19 Declaring a Main Function](#)

[Click here to view code image](#)

```
fun main(args: Array<String>) { // A main function in a regular Kotlin file

    println("Hello World!")

}
```

The main function is declared like any other top-level function and accepts an array of strings that captures command-line arguments. The missing return type indicates that this is a void function. In Kotlin, void is represented by the `Unit` type to ensure that every function call has a value and can, for instance, be assigned to a variable. `Unit` is the default return type, so if you don't define an explicit return type for a function, it returns `Unit` (like the main function).

#### Note

A main function allows you to run code such as print statements in normal Kotlin files (with `.kt` file extension) instead of using Kotlin scripts (`.kts` files). Listings in this book still assume a script file and therefore don't use a main function unless noted otherwise.

## Default Values and Named Parameters

In Kotlin, you can define default values for function parameters. This makes them optional, so if no argument is given for such a parameter, its default value is used. In [Listing 2.20](#), the default value for the recursion depth is zero.

### [Listing 2.20 Defining Default Values for Parameters](#)

[Click here to view code image](#)

```
fun exploreDirectory(path: String, depth: Int = 0) {
```

The parameter definition syntax is similar to variable declarations but without **val** or **var** keywords. Also, the types cannot be omitted because they form part of the function contract and must be well defined.

This function can now be called in several different ways because an argument for the second parameter is optional, and you can make use of named parameters to assign only selected parameters or to change their order. This is demonstrated in [Listing 2.21](#).

#### **Listing 2.21 Using Named Parameters**

[Click here to view code image](#)

```
val directory = "/home/johndoe/pictures"

exploreDirectory(directory)           // Without
exploreDirectory(directory, 1)        // Recursi
exploreDirectory(directory, depth=2)   // Uses na
exploreDirectory(depth=3, path=directory) // Uses na
```

In these examples, a directory on the file system is explored. The first call to `exploreDirectory` uses the default depth of zero by omitting the optional argument. The second call defines all arguments as usual, whereas the third one uses named parameters to set the depth. This is not required here but may increase readability. In contrast, because the last call uses named parameters to change the order of arguments passed in, all parameters must be explicitly named. Also, for functions with multiple default values, named parameters allow passing in any subset of the optional parameters.

**Note**

Parameters with default values should be placed at the end of the parameter list. That way, all required arguments can be passed in first without any ambiguity about which parameter belongs to which value. If you have multiple optional parameters of the same type, their order decides which value belongs to which.

In any case of ambiguity, named parameters can be used to resolve it.

## Overloading

Functions can be *overloaded* by creating another function with the same name but different parameters. A typical use case for this is when there are alternate ways to represent the data the function requires, as illustrated in Listing 2.22.

**Listing 2.22 An Overloaded Function**

[Click here to view code image](#)

---

```
fun sendEmail(message: String) { ... }

fun sendEmail(message: HTML) { ... }
```

---

This way, the `sendEmail` function can be called either by passing in a string to send a plain text email, or by passing in some `HTML` object to send an email with HTML content.

#### Java Note

A common reason to use overloading in languages without default parameter values (like Java) is to implement optional parameters. This is done by adding overloads that omit an optional parameter and delegate to the full-fledged implementation with a default value passed in. Thanks to default parameter values, there is no need to use function overloading for this in Kotlin, which can drastically reduce the number of code lines.

[Click here to view code image](#)

```
// Java code

Pizza makePizza(List<String> toppings) {

    return new Pizza(toppings);

}

Pizza makePizza() {

    return makePizza((Arrays.asList()));

}
```

Because Kotlin provides default parameter values, there is no need to use function overloading for this:

[Click here to view code image](#)

```
fun makePizza(toppings: List<String> = emptyList()) = Pizza(toppings)
```

This drastic reduction in lines of code improves exponentially with the number of optional parameters.

## Extension Functions

From other languages, you may be familiar with utility classes such as `StringUtils` or `DateUtils` that define a variety of helper functions for strings or dates, respectively. Their intention is to extend the interface (API) of existing classes with additional functions that are useful either in your application context or even in general. In many languages, this is necessary for classes you don't own because you cannot change the actual classes.

In Kotlin, extension functions open up the possibility to add methods and properties directly to existing classes such as `Int` or `Date`, or at least Kotlin makes it look like you could (classes are discussed in detail in [Chapter 4](#)). Consider [Listing 2.23](#) as an example of an extension function.

### Listing 2.23 Creating and Calling Extension Functions

[Click here to view code image](#)

```
fun Date.plusDays(n: Int) = Date(this.time + n * 86400000)

val now = Date()

val tomorrow = now.plusDays(1) // Extension can be called on now
```

To create an extension function, you use the **fun** keyword as usual but prefix the function name with the so-called *receiver class*. The extension can then be called like any method on the receiver class, resulting in cleaner code than when using utility classes. Plus, your IDE can now offer useful autocompletion, unlike when using static methods from utility classes.

### Static Resolving of Extensions

Extensions cannot actually become part of existing classes; Kotlin just allows them to be called using the same syntax. Semantically, however, they are different because they're resolved statically rather than at runtime. Thus, an extension function `print` called on an expression of type `Number` calls `Number::print`, even if the expression evaluates to the subtype `Int` at runtime. Listing 2.24 demonstrates exactly this situation.

### Listing 2.24 Static Resolving of Extension Functions

[Click here to view code image](#)

```
fun Number.print() = println("Number $this") // Extension

fun Int.print() = println("Int $this") // Extension

val n: Number = 42 // Static type is Number
n.print() // Prints "Number 42"
```

Even though `n` is an `Int` at runtime, the extension function being called for `n.print()` is resolved at compile time, thus

calling `Number::print` instead of `Int::print`. So keep in mind there is no polymorphism of this type when using extension functions, and use this feature carefully for functions that callers might expect to behave polymorphically.

**Note**

If the receiver type of an extension function already has a member function with the same signature, the member function takes precedence over the extension function.

## Scope and Importing

Extension functions and properties are typically created on the top level and are therefore visible inside the entire file. Just like any other function or variable, they can also be given a visibility. Visibilities in Kotlin are discussed in detail in [Chapter 4](#). For now, it is just important that the default visibility be **public** so that extensions can be imported into other files as well, which looks like static imports, as shown in [Listing 2.25](#). Assume here that the `plusDays` extension from [Listing 2.23](#) is defined in a file residing in a package named `time`.

### [Listing 2.25 Importing Extension Functions](#)

[Click here to view code image](#)

```
import com.example.time.plusDays
```

Note that the function name directly follows the package name because the extension does not reside inside a class or object. If there are multiple extension functions with the name `plusDays` defined in the package, all of them are imported.

**Tip**

Extension functions offer a convenient way to work around limitations of third-party APIs that you cannot modify directly. For example, you can add methods to interfaces or encapsulate boilerplate around APIs. This is highly useful on Android, and you will see several examples of this in [Chapter 7, Android App Development with Kotlin: Kudoo App](#), and [Chapter 8, Android App Development with Kotlin: Nutriicious](#).

## Infix Functions

Normally, function calls in Kotlin use prefix notation with parentheses around the arguments. For functions with two parameters, however, you may want to put the function name between the arguments, similar to computations like `7 + 2` where the operator also stands between its arguments. Kotlin allows you to define such infix functions and has several predefined ones. You've already seen the infix functions `until`, `downTo`, and `step` that are commonly used in `for` loops. As another example, consider the `to` function that is predefined in Kotlin and shown in Listing 2.26.

**Listing 2.26 The Infix Function to**

[Click here to view code image](#)

```
infix fun <A, B> A.to(that: B) = Pair(this, that) //
```

`A` and `B` are generic parameters, which are discussed in [Chapter 4](#). For now, notice the `infix` modifier for the function. This allows you to call this function as shown in [Listing 2.27](#), with the function name between the arguments.

**Listing 2.27 Calling an Infix Function**

[Click here to view code image](#)

```
val pair = "Kotlin" to "Android"
```

```
val userToScore = mapOf("Peter" to 0.82, "John" to 0.
```

The first line creates a `Pair("Kotlin", "Android")`. The second example shows the most common use case for this infix function: instantiating maps using the helper function `mapOf` that accepts a `vararg` of type `Pair` and using `to` to create the `Pair` objects for the map. Although it's not necessary to use the `to` function instead of the `Pair` constructor, it can improve readability in many cases.

You can also define your own infix functions using the `infix` modifier, but only if the function is either a member function

of a class or an extension function and has only one additional argument (just like the `to` function above). For instance, you could create an infix function that duplicates a string a certain number of times using the standard library function `repeat`, as in Listing 2.28.

**Listing 2.28 Defining Own Infix Functions**

[Click here to view code image](#)

```
infix fun Int.times(str: String) = str.repeat(this)

val message = 3 times "Kotlin " // Results in "Kotlin
```

◀ ▶

## Operator Functions

When looking at the code above, you may be thinking that it would be nicer if you could instead just write `3 * "Kotlin "`—and you can achieve this. The concept behind this is called *operators* or *operator functions*. Operators are symbols with built-in semantics for the compiler such as `+`, `-`, or `+=`; and by writing your own operator functions, you can define your own semantics for these operators. This is done by defining functions with a specific name such as `plus`, `minus`, or `plusAssign` for the three operators listed above.

To achieve the more concise syntax `3 * "Kotlin "`, you can use the operator function `times` that is associated with the `*` operator, as in Listing 2.29.

**Listing 2.29 Defining and Calling Operators**

[Click here to view code image](#)

```
operator fun Int.times(str: String) = str.repeat(this)

val message = 3 * "Kotlin " // Still results in "Kotl
```

◀ ▶

Compared with the infix function, the only difference is the use of the `operator` keyword instead of the `infix` keyword. Operators can increase the expressiveness and conciseness of your code, but they should be used judiciously.

The Kotlin team deliberately allows only a predefined set of operators to be used, in contrast to Scala for instance. The reason behind this is that overusing operators makes your code hard to understand for other developers (and yourself). [Table 2.2](#) gives an overview of the most important operators available in Kotlin.

Table 2.2 Available Operator Functions

Function Name	Operator	Example	Equivalent
<b>Arithmetic Operators</b>			
plus	+	2 + 3	2.plus(3)
<b>Assignment Operators</b>			
minus	-	"Kotlin" - "in"	"Kotlin".minus("in")
times	*	"Coffee" * 2	"Coffee".times(2)
div	/	7.0 / 2.0	(7.0).div(2.0)
rem	%	"android" % 'a'	"android".rem('a')
<b>Assignment Operators</b>			
plusAssign	+=	x += "Island"	x.plusAssign("island")
minusAssign	-=	x -= 1.06	x.minusAssign(1.06)
timesAssign	*=	x *= 3	x.timesAssign(3)
divAssign	/=	x /= 2	x.divAssign(2)
remAssign	%=	x %= 10	x.remAssign(10)

## Miscellaneous Operators

inc	++	grade++	grade.inc()
-----	----	---------	-------------

dec	--	'c'--	'c'.dec()
-----	----	-------	-----------

contains	in	"A" in list	list.contains("A")
----------	----	-------------	--------------------

rangeTo	..	now..the n	now.rangeTo(t hen)
---------	----	------------	--------------------

get	[...]	skipList [4]	skipList.get(4)
-----	-------	--------------	-----------------

set	[...]	array[5] = ...	array.set(5, 42)
-----	-------	----------------	------------------

### Tip

There are more operators available that are not listed here. To get a complete list of all operator functions available for your version of Kotlin, simply use autocompletion in IntelliJ or Android Studio. For instance, typing “operator fun Int .” and invoking autocompletion after the dot brings up all available operators to override.

## NULL SAFETY

In 1965, Tony Hoare designed an object-oriented programming language called ALGOL W.<sup>8</sup> As he admitted, a **null** reference was added simply because it was easy to implement and thus tempting. He now calls it his billion-dollar mistake. This is not to talk down his accomplishments; he has contributed incredibly much to the field of computer science. But **null** references cause a lot of pain, as you know from the dreaded **NullPointerException**.

<sup>8</sup>. <https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare>

Fortunately, Kotlin allows you to control the risk of **null** references and even eliminate them from pure Kotlin code. This is achieved by differentiating between nullable and non-nullable types.

## Nullable Types

Every type in Kotlin is by default non-nullable, meaning that it cannot hold a **null** reference. Thus, it's also safe to access any property or method on it, as you've seen in the previous code examples in this book. To create a *nullable* variable, you have to append a question mark to the variable type as shown in [Listing 2.30](#).

**Listing 2.30 Nullable and Non-Nullable Types**

[Click here to view code image](#)

```
val name: String? = "John Doe" // Nullable variable

val title: String? = null           // Nullable variable

val wrong: String = null           // Causes compiler error
```

The type `String?` denotes a nullable string, in contrast to the normal `String`, which cannot hold **null**. You use the same syntax for all types, for instance `Int?` or `Person?`. A good practice is to avoid them whenever possible. Unfortunately, this is not always possible when interoperating with Java because every value coming from Java may potentially be **null**. Interoperability issues are covered in more detail in [Chapter 5, Interoperability with Java](#).

## Working with Nullables

When you use a nullable variable, then in general you can no longer safely access properties and methods on it. For this reason, Kotlin forbids normal access.

## Safe-Call Operator

[Listing 2.31](#) shows how you can safely access members of nullable types, first using explicit **null** checks and then using the *safe call operator*.

**Listing 2.31 Accessing Members of Nullable Variables**

[Click here to view code image](#)

```
val name: String? = "John Doe"

println(name.length)    // Not safe => causes compiler error

if (name != null) {      // Explicit null check

    println(name.length) // Now safe so compiler allows it

}

println(name?.length)    // Safe call operator

val upper = name?.toUpperCase() // Safe call operator
```

You cannot access `name.length` directly because `name` may be **null**, thus causing a `NullPointerException`. Instead, you can explicitly check for **null**. The compiler will track your condition and allow direct access inside the `if` block because it's safe. This concept is called *smart cast* and is discussed in detail in the next section. Alternately, you can use Kotlin's safe-call operator by appending a question mark to the variable as in `name?.length`. This returns the string length if `name` is not **null**, and otherwise returns **null**. Thus, the type of `name?.length` is `Int?`, and the type of the variable `upper` in the above code is `String?`.

**Note**

You can define extensions on nullable types so that they can be called directly on that type, without safe-call operator or other null handling. In fact, this is how the `toString` method is defined in Kotlin. The extension is defined as `Any?.toString() = ...`.

When defining extensions on nullable types, you must handle the `null` case inside the extension function to make sure it can be called safely.

## Elvis Operator

In order to quickly define a default value for the case in which a variable is `null`, Kotlin offers the *elvis operator*, which looks like a shortened form of the ternary conditional operator from other languages and gets its name from the look of the operator when seen as an emoticon. Listing 2.32 shows an example for this operator.

**Listing 2.32 Elvis Operator**

[Click here to view code image](#)

```
val name: String? = null

val len = name?.length ?: 0 // Assigns length, or e]
```

The elvis operator `?:` can be read as “or else” and is a powerful tool to get rid of nullability in your code quickly. Here, `len` is now an `Int` instead of an `Int?` because the elvis operator returns the value on its right-hand side if its left-hand side evaluates to `null`. So here, `len` will be zero.

Apart from the elvis operator, Kotlin also provides a variety of useful functions to deal with nullable types. For instance, the above result could also be achieved using `name.orEmpty().length`.

**Note**

At this point, you could create utility functions like `orEmpty` yourself. In fact, `orEmpty` is simply an extension function on `String?` that uses the elvis operator.

Navigate to the declaration of such functions in your IDE to see how they are defined; this can help you tremendously in grasping the language.

## Unsafe Call Operator

Lastly, there is also an *unsafe call operator* that allows you to force accessing a property or method on a nullable type without handling **null**. You should use this judiciously, never as a shortcut to handling nullability correctly. However, it has legitimate uses and you can use it when you're certain that an object cannot be **null** at a point in your code. Listing 2.33 shows how to use the unsafe call operator; it is intentionally designed to look like you're shouting at the compiler using a double exclamation mark.

**Listing 2.33** Unsafe Call Operator—Member Access

[Click here to view code image](#)

```
val name: String? = "John Doe" // May also be null
val len = name!!.length // Asserts the compiler that
```

The second line assures the compiler that `name` is not **null** at that point using `name!!` (which succeeds in this case). The syntax `name!!` effectively transforms the nullable type to a non-nullable type so that you can directly access any of its members. Thus, it is often used without any member access following it, as shown in Listing 2.34.

**Listing 2.34** Unsafe Call Operator—Passing Arguments

[Click here to view code image](#)

```
fun greet(name: String) = println("Hi $name!")
greet(name!!) // Better be sure that 'r
greet(name ?: "Anonymous") // Safe alternative using
```

Using the unsafe call operator injudiciously is one of the few ways you can force a `NullPointerException` in pure Kotlin code. That nullability even exists in Kotlin is due to its very pragmatic goal to be 100% interoperable with Java.

Fundamentally, there's no need for `null` references at all, so try to avoid them whenever you can. Even when you do need them, it is good practice to keep nullability local, for instance, by defining useful default values early on using the elvis operator.

## EQUALITY CHECKS

There are two fundamental ways to check equality of objects: *referential equality* and *structural equality*.

Referential equality of two variables means that they point to the same object in memory (they use the same *reference*). In contrast, structural equality means that the *values* of two variables are the same, even if stored at different locations. Note that referential equality implies structural equality. Kotlin provides one operator for each of them: You use `==` to check for referential equality and `==` to check for structural equality, as done in Listing 2.35.

**Listing 2.35 Referential and Structural Equality Checks**

[Click here to view code image](#)

```
val list1 = listOf(1, 2, 3) // List object

val list2 = listOf(1, 2, 3) // Different list object

list1 === list2 // false: the variables reference different objects

list1 == list2 // true: the objects they reference are equal

list1 !== list2 // true: negation of first comparison

list1 != list2 // false: negation of second comparison
```

Structural equality checks use the `equals` method that every object has for comparison. In fact, the default implementation of `a == b` is `a?.equals(b) ?: (b === null)`. So if `a` is not `null`, it must be equal to `b`, and if `a` is `null`, then `b` must also be `null`.

Note that `a == null` is automatically translated to `a === null` so there's no need for you to optimize your code when comparing against `null`.

## Floating Point Equality

Kotlin adheres to the IEEE Standard for Floating Point Arithmetic<sup>9</sup> when comparing floating-point numbers, but only if the types of both compared numbers can be statically evaluated to `Float` or `Double` (or their nullable counterparts). Otherwise, Kotlin falls back to the Java implementations of `equals` and `compare` for `Float` and `Double`, which do not adhere to the standard because `NaN` (Not-a-Number) is considered equal to itself, and `-0.0f` is considered less than `0.0f`. Listing 2.36 makes this corner case more tangible.

9. <https://ieeexplore.ieee.org/document/4610935/>

### Listing 2.36 Equality of Floating Point Values

[Click here to view code image](#)

```
val negativeZero = -0.0f           // Statically inf
val positiveZero = 0.0f            // Statically inf
Float.NaN == Float.NaN           // false (IEEE st
negativeZero == positiveZero     // true (IEEE sta

val nan: Any = Float.NaN          // Explicit type:
val negativeZeroAny: Any = -0.0f // Explicit type:
val positiveZeroAny: Any = 0.0f  // Explicit type:

nan == nan                       // true (not IEEE
negativeZeroAny == positiveZeroAny // false (not IEEE
```

These examples give the same results for `Double` values. Remember that, in general, you shouldn't compare floating-

point values for equality but rather compare with tolerance due to machine precision limitations when storing such values.

## EXCEPTION HANDLING

Exceptions provide a standardized way to report errors in languages like Kotlin, Java, or C# so that all developers follow the same concept of reporting and handling errors. This way, exceptions contribute to developer productivity while also increasing robustness of the code. In this section, you'll learn about the principles of exception handling, how it's done in Kotlin, how checked and unchecked exceptions differ, and why Kotlin only has unchecked exceptions.

### Principles of Exception Handling

Not only do exceptions allow you to denote erroneous or unexpected behavior or results in your code, they also remove the obligation to handle such unexpected behavior right where it happens. Instead, exceptions can bubble up in your code. Through exception bubbling, you're able to handle exceptions at the level where it's appropriate in your code and where you know how to recover from the exception. For instance, a common source of exceptions is input/output (I/O) operations at the lowest levels of an application. However, at that level, you're unlikely to know how to handle errors—whether to retry the operation, notify the user, skip the operation altogether, and so on. More high-level parts of your application, on the other hand, should be able to recover from an exception like this.

## Exception Handling in Kotlin

On the syntax level, exception handling consists of an operation to throw (*create*) exceptions and ways to catch (*handle*) them. The handling of an exception may occur far from where it was thrown. Kotlin uses the **throw** keyword to throw a new exception and **try-catch** blocks to handle them. Listing 2.37 shows an introductory example.

Listing 2.37 Throwing and Catching Exceptions

[Click here to view code image](#)

```
fun reducePressureBy(bar: Int) {  
    // ...  
  
    throw IllegalArgumentException("Invalid pressure reading")  
}  
  
  
try {  
    reducePressureBy(30)  
}  
catch(e: IllegalArgumentException) {  
    // Handle exception here  
}  
catch(e: IllegalStateException) {  
    // You can handle multiple possible exceptions  
}  
finally {  
    // This code will always execute at the end of the block  
}
```

The **throw** keyword followed by the exception class throws a new exception. Note that there's no **new** keyword for object instantiation in Kotlin. The **try** block contains the code that may throw an exception. If an exception occurs, it is handled in the corresponding **catch** block. The **finally** block guarantees that its code is executed at the end of the **try**-

**catch**, irrespective of whether an exception occurred. It is commonly used to close any open connections to resources. Both **catch** and **finally** are optional, but one of them must be present.

Consistent with the rest of the language, **try-catch** blocks are expressions in Kotlin and thus have a value. This value is again defined by the last lines inside the **try**- and **catch** blocks for each case, as demonstrated in Listing 2.38.

**Listing 2.38** Using **try-catch** as an Expression

[Click here to view code image](#)

```
val input: Int? = try {  
    inputString.toInt() // Tries to parse input to an  
} catch(e: NumberFormatException) {  
    null // Returns null if input could  
}
```

Here, if the input string can be parsed to an integer, its value is stored in `input`. Otherwise, `toInt` will cause a number format exception, making the execution jump into the **catch** block and resulting in the value `null` for the **try** expression. If appropriate, you could instead return a default value inside the **catch** block to avoid nullability.

Taking the concept further, even **throw** is an expression in Kotlin. You might wonder what the value of such an expression should be, considering that execution cannot continue to the lines following a **throw** statement. But this is exactly the point because non-termination (and based on this unreachable code) is marked by the special type **Nothing** in Kotlin. If you're familiar with domain theory, this is the bottom element. It's also the bottom of Kotlin's type system, meaning it is the subtype of every other type. It has no possible values, and for **Nothing?** the only valid value is **null**. Because **throw** is an expression, you can use it with

the elvis operator as in Listing 2.39 (remember you can think of the elvis operator as “or else”).

#### **Listing 2.39 Using `throw` as an Expression**

[Click here to view code image](#)

```
val bitmap = card.bitmap ?: throw IllegalArgumentException  
    bitmap.prepareToDraw() // Known to have type Bitmap
```

Notice that the compiler makes use of the information that the `Nothing` type carries—namely, if the code reaches the **throw** expression, the first line of code will not terminate. In other words, if `card.bitmap` is `null`, the first line of code won’t terminate. With this, the compiler can infer that the `bitmap` variable must be non-null in the second line.

Otherwise, execution wouldn’t even reach the second line.

You can also use the `Nothing` type to mark functions that never return to make use of this compiler behavior. A common example is a `fail` function as known from testing frameworks, as illustrated in Listing 2.40.

#### **Listing 2.40 Using `Nothing` for Non-Returning Functions**

[Click here to view code image](#)

```
fun fail(message: String): Nothing { // Nothing type  
    throw IllegalArgumentException(message)  
}  
  
val bitmap = card.bitmap ?: fail("Bitmap required")  
    bitmap.prepareToDraw()
```

## Checked and Unchecked Exceptions

An unchecked exception is an exception that a developer may handle but is not forced by the compiler to handle. In other words, the compiler doesn't *check* if the exception is handled. If an unchecked exception occurs and isn't handled until it bubbles up to the top level (in the stack frame), it will cause the system to crash and will show the error that caused it. In contrast, for a checked exception, the compiler forces developers to handle it. So if a method indicates in its signature that it may throw an exception, you have to handle that exception when calling the method—either using a **catch** block or by explicitly passing along the exception. The problem that checked exceptions as used in Java have shown over the years is that developers are distracted by the obligation to handle exceptions, when really they want to focus on the actual logic. The result is swallowed exceptions and empty **catch** blocks. These are worse than letting the exception bubble up because it just defers the error to far-away parts of the code. This makes finding the root cause of the error exponentially harder. Plus, such **try-catch** blocks add unnecessary boilerplate and complexity to the code.

There has been a lot of discussion about the merits and downsides of checked exceptions. Modern Java-inspired languages like Kotlin, C#,<sup>10</sup> and Ruby<sup>11</sup> decided not to add them and instead use only unchecked exceptions. Thus, all exceptions in Kotlin are unchecked, leaving it to the developer to decide whether to handle them. In Kotlin, this decision is also based on JetBrains' desire for a language for large-scale systems, and although checked exceptions have proven useful in smaller projects, they introduce considerable obstacles for larger projects. The reason is that, in smaller projects, exceptions can be handled rather close to the source of the exception, whereas large systems require developers to annotate all possible exceptions that can occur at each level in between. That is, every function that passes along an exception and doesn't handle it yet must include a **throws** annotation in

its signature. Interoperability issues associated with this difference are discussed in [Chapter 5](#).

10. <https://docs.microsoft.com/en-us/dotnet/csharp/>

11. <https://www.ruby-lang.org/en/>

**Java Note**

Java uses both checked and unchecked exceptions, so you *could* use only unchecked exceptions in your code. However, third-party code using checked exceptions forces you to handle them in your code and add `throws` annotations to each function that passes along the exception.

## SUMMARY

You are now familiar with the basics of Kotlin, including data types, control flow, function declarations, handling nullability, and exceptions. Throughout the chapter, Kotlin's focus on conciseness and safety became apparent with features such as type inference, explicit nullable types, default values, function shorthand syntax, and the fact that almost all language constructs are expressions. In addition to this, you saw how extension functions can be used to enhance the API of classes that you don't own, and you explored why Kotlin uses only unchecked exceptions for exception handling.

In the following two chapters, you will explore the two main programming paradigms on which Kotlin is based—namely, functional programming and object orientation—and how they are incorporated into the language.

# 3

## Functional Programming in Kotlin

*Divide each difficulty into as many parts as is feasible and necessary to resolve it.*

René Descartes

This chapter covers the fundamentals of functional programming and explains how these conventions are incorporated into Kotlin. It starts with an overview of the principles and benefits of functional programming and moves on to specific topics, such as higher-order functions, lambda expressions, and lazy evaluation, which are essential in functional programming.

### PURPOSE OF FUNCTIONAL PROGRAMMING

As the name implies, functional programming emphasizes the use of functions for application development. Similar to object orientation, where the use of classes and objects is emphasized, functional programming can be seen as a programming paradigm. At an abstract level, every program or component can be seen as a function with an input and an output. This model allows for new ways to compose and therefore modularize programs as functions. The main concepts to understand in functional programming are *higher-order functions*, *lambda expressions*, and *lazy evaluation*.

- **Higher-order functions** are functions that take in one or more other functions as input, return a function as their value, or both. They may

also take in other types of arguments. The use cases for this are wide and powerful, and you will see many of them in this chapter.

- **Lambda expressions** are unnamed functions and typically defined on the fly. They allow denoting functions concisely at the use site. Lambda expressions are frequently used in combination with higher-order functions to define the functions passed in as arguments or returned from a higher-order function.
- **Lazy evaluation** is all about evaluating expressions only as far as necessary. For instance, when you have a sequence of a million elements, you might not want Kotlin to evaluate all these elements at runtime unless they are actually used. This is what Kotlin's *sequences* do. Lazy sequences allow you to use infinite data structures and significantly improve performance in certain cases.

These three major concepts of functional programming open up completely new ways to approach and solve problems, to write highly expressive code, and to modularize code.

Apart from being a paradigm, functional programming is also a programming style in which you make heavy use of immutability and avoid functions with side effects.

Immutability increases robustness of the code, especially in multithreaded environments, because it avoids unwanted state changes. Avoiding such side effects of functions improves understandability, testability, and robustness because it minimizes hidden and unexpected state changes. This also helps reason about lazily evaluated expressions because these may otherwise work with a mutated state.

Also, pure functional languages normally do not offer **null**, as opposed to object-oriented languages. This maps well to Kotlin's ideals, which advocate immutability and avoiding the use of **null**, which is typically used in combination with mutable objects. However, as an object-oriented language that must interoperate with Java, Kotlin does still have **null**.

**Java Note**

Kotlin's lazy sequences are conceptually the same as Java 8 Streams,<sup>1</sup> and their usage is very similar as well.

1. <https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html>

## Benefits of Functional Programming

There are many ways in which functional programming can improve your code, and it works well in combination with object orientation.

First, higher-order functions and lambda expressions can greatly improve the conciseness and expressiveness of code. For instance, they often allow replacing imperative-style loops with a single line of code. Also, they prove useful for working with collections such as lists and sets—again stemming from the fact that you can use more concise alternatives to imperative code.

Second, lazy evaluation can significantly enhance performance when working with large collections or when performing expensive computations on the items of a collection. Also, lazy sequences in Kotlin allow for the definition of infinite data types, such as a list of all prime numbers. Internally, this means that any item in the list will be computed on the fly and only once it becomes necessary, similar to invoking a function to compute the first  $n$  primes for some number  $n$ .

Third, functional programming opens up new ways to solve common development tasks. This includes working with collections and implementing design patterns such as the *Strategy pattern*<sup>2</sup> more easily using lambda expressions.

<sup>2</sup>. [https://sourcemaking.com/design\\_patterns/strategy](https://sourcemaking.com/design_patterns/strategy)

You will be able to form your own picture of the benefits of functional programming when going through this chapter and applying the concepts in your code. You should use the presented features with discipline. Not everything has to be solved in a purely functional style, especially because it may be unfamiliar to other developers reading the code. But used appropriately in combination with Kotlin's other language features, it allows you to write cleaner, more readable code and synergizes with other paradigms such as object orientation.

## FUNCTIONS

Naturally, functions are at the heart of functional programming. This section introduces the concepts of a function signature and function types, and it demonstrates that Kotlin supports proper function types.

At its core, a function is a modularized block of code that takes in zero or more input arguments, performs some computations on them, and may return a result value.

Information about parameters and return values is encapsulated in the *function signature*. Listing 3.1 demonstrates a function signature in Kotlin. It defines the function’s name, its inputs, and its outputs (in that order).

**Listing 3.1 Function Signature**

[Click here to view code image](#)

---

```
fun countOccurrences(of: Char, inStr: String): Int /
```

Because functions are first-class members in Kotlin, you can denote the type of a function using proper *function types*, for instance, as the type of a variable holding a function. This is shown in Listing 3.2.

**Listing 3.2 Function Type**

[Click here to view code image](#)

---

```
val count: (Char, String) -> Int = ::countOccurrences
```

The `count` variable holds a function, and its type is `(Char, String) -> Int`, meaning a function with a `Char` as first parameter, a `String` as second parameter, and an `Int` return value. We call this a *first-order function*—in contrast to higher-order functions—because it has no functions as parameters or return types.

In order to refer to a named function, you can use function pointers by prefixing the function name with two colons, as in `::countOccurrences`. As you will see later, you use

lambda expressions instead in many cases because they are more flexible.

**Java Note**

There are no proper function types in Java 8. Instead, there is only a fixed number of predefined function types.<sup>3</sup>

3. <https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html>

## LAMBDA EXPRESSIONS

Lambda expressions allow the definition of unnamed functions at the call site, without the boilerplate of declaring a named function. Oftentimes, we will refer to lambda expressions as *lambdas for short*.

Lambdas are one of the main features of most modern programming languages like Scala, Java 8, or Dart. Since it intends to be a conglomerate of best practices, it is no wonder that Kotlin also has lambda expressions baked into the language. Because Kotlin supported lambdas right from the start and doesn't carry much legacy, it can provide smooth syntactic support for them.

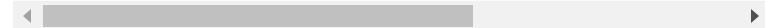
### Using Lambdas in Kotlin

Kotlin has powerful syntactic support for lambdas, allowing a very concise and readable notation. In its full glory, a lambda expression can be written in Kotlin as shown in Listing 3.3.

#### Listing 3.3 Full Lambda Expression

[Click here to view code image](#)

```
{ x: Int, y: Int -> x + y } // Lambdas are denoted v
```



Lambda expressions are wrapped in curly braces, their parameters defined on the left-hand side of the arrow, and the function body on the right-hand side. This lambda function takes in two integers and returns their sum, so its type is

`(Int, Int) -> Int`. Hence, you could assign it to a variable of that type, as done in [Listings 3.4 through 3.6](#).

You can use Kotlin's type inference when working with lambdas, as well as some other syntactic sugar. Let us start with the lambda expression from [Listing 3.3](#), assign it to a variable, and improve the code step by step. [Listing 3.4](#) shows the most explicit way to assign it to a variable, with all types mentioned.

#### **Listing 3.4 Assigning a Lambda Expression**

[Click here to view code image](#)

---

```
val sum: (Int, Int) -> Int = { x: Int, y: Int -> x +
```

First of all, Kotlin can infer from the variable type that `x` and `y` must be of type `Int`, so you don't have to repeat it inside the lambda. This is shown in [Listing 3.5](#).

#### **Listing 3.5 Type Inference for Lambda Arguments**

[Click here to view code image](#)

---

```
val sum: (Int, Int) -> Int = { x, y -> x + y } // Ir
```

Conversely, you can use explicit types inside the lambda and then let Kotlin infer the variable type, as in [Listing 3.6](#).

#### **Listing 3.6 Type Inference for Lambda Variables**

[Click here to view code image](#)

---

```
val sum = { x: Int, y: Int -> x + y } // Infers the
```

Kotlin not only infers the parameter types but also the return type from the lambda's definition.

If a lambda expression has only one argument, there is another way to write lambdas concisely. To see this, consider a lambda that transforms a `String` to uppercase so that the string is the only argument. You can then refer to the string via the implicit parameter name `it` and then skip the parameter list completely, as shown in [Listing 3.7](#).

### Listing 3.7 Implicit Argument Name `it`

[Click here to view code image](#)

```
val toUpper: (String) -> String = { it.toUpperCase()
```

Note that this only works if Kotlin can infer the type of `it`, which is usually the case because you use lambdas primarily as arguments for higher-order functions.

In this particular case, you can use a function reference instead of a lambda because you are just applying `String.toUpperCase` and this is actually a predefined, named function. Thus, Listing 3.8 is equivalent to Listing 3.7.

### Listing 3.8 Function Pointers

[Click here to view code image](#)

```
val toUpper: (String) -> String = String.toUpperCase
```

Lambda expressions are used to avoid the overhead of defining named functions whenever it is not necessary, for instance, when they are only used in one specific place. If there already is a named function that can be used directly instead, there is no need to wrap its call into a lambda expression; you can reference the named function instead.

## HIGHER-ORDER FUNCTIONS

Higher-order functions take another function as input or return a function. Consider the definition of a higher-order function called `twice` shown in Listing 3.9.

### Listing 3.9 Defining a Higher-Order Function

[Click here to view code image](#)

```
fun twice(f: (Int) -> Int): (Int) -> Int = { x -> f(f
```

This function takes in a function `f` that maps an `Int` to an `Int` and also returns a function from `Int` to `Int`. More specifically, it returns the function that applies `f` twice.

Higher-order functions unfold their full potential only in combination with lambda expressions to pass in the arguments. For example, to call the higher-order function `twice`, you must pass in a function for `f`. You can do this using a lambda expression or a function reference. Both options are shown in [Listing 3.10](#).

#### **Listing 3.10 Calling a Higher-Order Function**

[Click here to view code image](#)

```
val plusTwo = twice({ it + 1 }) // Uses lambda
val plusTwo = twice(Int::inc) // Uses function reference
```

When using a lambda, Kotlin infers the type of `it` using the type of `f`, which is `(Int) -> Int`. Kotlin thus infers `it` to be an integer and validates your lambda based on this type information.

Whenever the *last* argument of a higher-order function is a lambda expression, you can move the lambda out of the parentheses, as in [Listing 3.11](#).

#### **Listing 3.11 Lambdas as Last Argument**

[Click here to view code image](#)

```
val plusTwo = twice { it + 1 } // Moves lambda out of parentheses
```

Why would you do this? First, for a higher-order function with a lambda as its *only* argument, it allows skipping the parentheses altogether, as in [Listing 3.11](#). You will see many such functions when discussing collections in the next section. Second, this lets you define higher-order functions that look like language keywords and encapsulate related code into blocks. This will become clearer once we explore the Kotlin standard library at the end of this chapter. Third, this feature is what allows building simple domain-specific languages (DSLs) in Kotlin. This is covered in [Chapter 9](#), Kotlin DSLs.

The Kotlin standard library includes many useful higher-order functions that facilitate certain development tasks. You will get

to know some of them later in this chapter when exploring collections and Kotlin’s scoping functions.

## Inlining Higher-Order Functions

When working with higher-order functions, it is important to consider what happens at runtime. Functions are derived from classes and represented as objects at runtime. This has important performance implications. Consider Listing 3.12’s call to the `twice` function from Listing 3.9.

### Listing 3.12 Call to Higher-Order Function

[Click here to view code image](#)

---

```
val plusTwo = twice { it + 1 } // Creates a function
```

At runtime, the lambda expression becomes an object. More generally, for every lambda that you use, an object must be created at runtime—leading to memory consumption and garbage-collector invocations that could be avoided.

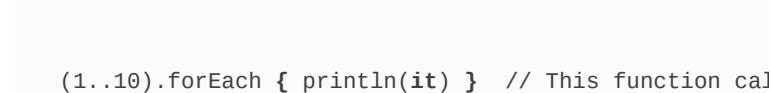
Higher-order functions can be declared as inline functions in order to avoid this problem. For instance, consider the `forEach` function from Kotlin’s standard library, shown in Listing 3.13. Note that `forEach` is a generic function, recognizable by its generic type parameter `<T>`. Generics are discussed in detail in Chapter 4, Object Orientation in Kotlin.

### Listing 3.13 Inlining Higher-Order Functions

[Click here to view code image](#)

---

```
public inline fun <T> Iterable<T>.forEach(action: (T)
```



```
    for (element in this) action(element)
```

```
}
```

```
(1..10).forEach { println(it) } // This function call
```

To declare an inline function, you add the **inline** modifier to the function declaration. This way, lambda expressions passed to the higher-order function are inlined at compile time.

Intuitively, the code will be copy-and-pasted to the call site so that the last line from [Listing 3.13](#) would result in the code from [Listing 3.14](#) getting compiled.

#### **Listing 3.14 Result of Inlined Function Call**

[Click here to view code image](#)

```
for (element in 1..10) println(element) // No lambda
```

Inline functions are used frequently in the Kotlin standard library because there is often no need for function objects at runtime. If you do need a function object at runtime, for instance to pass it to another higher-order function, you cannot use inlining. Also, note that inlining only makes sense for higher-order functions, and only for those that accept another function as an argument because this is where inlining prevents objects at runtime. If you use **inline** where it's useless, the Kotlin compiler will give you a warning.

If you have multiple functions as parameters but only want to inline some of them, you can exclude the others by placing the **noinline** modifier in front of the parameter name.

#### Java Note

Although the Java compiler may decide to inline function calls, there is no way to specify that a certain higher-order function should always inline its arguments.

## Non-Local Returns

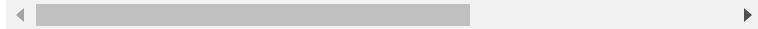
Lambda expressions in Kotlin are not allowed to **return** from their enclosing function. Thus, you cannot use a bare **return** inside a lambda—these can only be used to return from functions declared using **fun**. However, if the lambda's enclosing function is inlined, then it is possible to use **return** because the return is inlined as well.

[Listing 3.15](#) shows a typical example using the higher-order function **forEach**.

**Listing 3.15 Returning from an Enclosing Function**

[Click here to view code image](#)

```
fun contains(range: IntRange, number: Int): Boolean {  
    range.forEach {  
        if (it == number) return true // Can use return  
    }  
    return false  
}
```

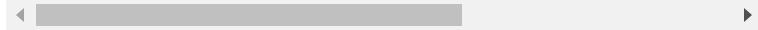


Here, the lambda passed to `forEach` returns from the enclosing `contains` function. This is called a *non-local return* because the return is inside the lambda, but it returns from the enclosing scope. It is allowed because `forEach` is an `inline` function so that the code from [Listing 3.15](#) effectively becomes the code shown in [Listing 3.16](#).

**Listing 3.16 Returning from an Enclosing Function—Inlined Result**

[Click here to view code image](#)

```
fun contains(range: IntRange, number: Int): Boolean {  
    for (element in range) {  
        if (element == number) return true // This return  
    }  
    return false  
}
```



This shows that the return inside the lambda ends up as a regular return inside a loop. Thus, it is possible to allow returns inside the lambda in this case. In general, it is possible to do so if the higher-order function calls the lambda directly inside its function body.

In other cases, where the lambda is called from a different execution context such as another lambda or from inside an

object, this is not possible because it would try to return from that other execution context instead of from the enclosing higher-order function. Thus, if you want to inline such a higher-order function, you must disallow returns from the lambda. This is what the **crossinline** modifier is for.

For instance, to inline the `twice` function from Listing 3.9, you must disallow returns from its lambda named `f` because `twice` calls `f` from inside another lambda. A valid inlined definition of `twice` is shown in Listing 3.17.

**Listing 3.17 Disallowing Non-Local Returns Using `crossinline`**

[Click here to view code image](#)

---

```
inline fun twice(crossinline f: (Int) -> Int): (Int)
```

---

Non-local returns from inside `f` must be disallowed here because they would be located inside their wrapping lambda expression after inlining—and as mentioned above, lambdas cannot normally use `return`. Additionally, the inner `f` would even try to return from inside its wrapping invocation of `f`, which is also not allowed.

## WORKING WITH COLLECTIONS

Kotlin collections are based on the Java Collections API, but there are some important differences in their declaration and also their use. This section covers how to work with collections and then explores higher-order functions defined on collections that greatly facilitate working with them.

## Kotlin Collections API versus Java Collections API

Consistent with Kotlin's focus on immutability, the Kotlin Collections API clearly differentiates between mutable and read-only collections. For each collection type (lists, sets, and maps), there is a read-only and a mutable collection type in Kotlin, namely `List` and `MutableList`, `Set` and `MutableSet`, and `Map` and `MutableMap`. It is thus important to remember that the collection types without "mutable" in their names are read-only in Kotlin and should be the first choice unless you really need a mutable collection.

Once initialized, you cannot add, replace, or remove elements from a read-only collection. However, you can still mutate any mutable objects stored *inside* the collection. So to achieve immutability with collections, you now have three levels to consider.

- Using `val` to hold the collection, meaning that you cannot reassign the variable
- Using an immutable collection (or at least a read-only one)
- Storing immutable objects inside the collection

### Note

Kotlin's collections are not immutable in a strict sense because there are (hacky) ways to mutate them. This is why we refer to them as *read-only* in this book.<sup>4</sup> However, the Kotlin team is working on truly immutable collections as well.<sup>5</sup>

4. <https://github.com/Kotlin/kotlinx.collections.immutable>

Each of the mutable subinterfaces `MutableList`, `MutableSet`, and `MutableMap` add mutating methods such as `add`, `set`, or `put` to the interface definition. Thus, you can add, replace, and remove elements from these mutable collections.

As you will see in the following section, the distinction between read-only and mutable collections is also reflected in the way you instantiate collections in Kotlin.

## Instantiating Collections in Kotlin

The Kotlin collection types are typically instantiated using one of the available helper functions that allow you to pass in an arbitrary number of items (a so-called **vararg**), as in Listing 3.18.

**Listing 3.18 Helper Functions for Creating Collections**

[Click here to view code image](#)

```
// Collections

val languages = setOf("Kotlin", "Java", "C++") // Cr

val votes = listOf(true, false, false, true) // Cr

val countryToCapital = mapOf( // Cr
    "Germany" to "Berlin", "France" to "Paris")

// Arrays

val testData = arrayOf(0, 1, 5, 9, 10) // Cr
```

This way, you can instantiate sets, lists, maps, and other collection types easily. All of the functions mentioned above create read-only collections, apart from the **arrayOf** function that was included because arrays are similar to collections, although they do not fall under the Collections API.

Apart from the mentioned helper functions, there are also ones to create mutable collections and collections of specific subtypes, as portrayed in Table 3.1.

Table 3.1 Helper Functions to Create Collections

Result Type	List	Set	Map
<b>Read-Only Collection</b>	<code>listOf</code>	<code>setOf</code>	<code>mapOf</code>
<b>Mutable Collection</b>	<code>mutableListOf</code>	<code>mutableSetOf</code>	<code>mutableMapOf</code>
<b>Specific Subtype</b>	<code>arrayListOf</code>	<code>hashSetOf</code>	<code>hashMapOf</code>
		<code>linkedSetOf</code>	<code>linkedMapOf</code>
		<code>sortedSetOf</code>	<code>sortedMapOf</code>

For arrays, there are also helper functions for all of Java's primitive types such as `intArrayOf`, `doubleArrayOf`, and `booleanArrayOf`. In the bytecode, these translate to arrays of the corresponding primitive type in Java such as `int[]`, `double[]`, and `boolean[]` instead of `Integer[]`, `Double[]`, and `Boolean[]`, respectively. This is important in performance-critical code to avoid unnecessary object creation, autoboxing, unboxing, and memory consumption.

### Accessing and Editing Collections

The way you access and edit a collection depends on the type of the collection and whether it is mutable or read-only. The method names of Java's collections have mostly been kept the same in Kotlin. But whenever a `get` method is defined, it is possible—and recommended—to use the *indexed access operator*, which is shown in Listing 3.19.

Listing 3.19 Indexed Access Operator for Reading

[Click here to view code image](#)

```
votes[1]      // false
testData[2]   // 5
countryToCapital["Germany"] // "Berlin"
```

There are several things to notice here. First, the example excludes the `languages` set because sets, per definition, have no concept of ordering. So it is not possible to select an `n`-th element from a set. Next, the `Map` type defines a `get` method, which takes in a key, in this case a `String`. Internally, the indexed access operator simply calls the `get` method on the object. You could do this explicitly with the same result, for instance, calling `votes.get(1)`.

For mutable collections that define a `set` method, you can also use the indexed access operator to replace elements. For [Listing 3.20](#), assume you had created mutable collections above so that you can replace their elements.

[Listing 3.20 Indexed Access Operator for Writing](#)

[Click here to view code image](#)

```
// Assumes mutable collections now
votes[1] = true                      // Replaces se
testData[2] = 4                        // Replaces thi
countryToCapital["Germany"] = "Bonn"  // Replaces val
```

This internally calls the `set` method defined for each of these mutable collection types, for instance, `votes.set(1, true)`.

With the possibility to instantiate and manipulate their elements, you can now look at how to work with collections effectively using common higher-order functions.

## Filtering Collections

A common task is to filter a collection of elements based on some condition. In Kotlin, you can use the higher-order function `filter` for this task. It is a higher-order function because the predicate is passed in as a function. Imagine you want to count the number of yes votes as in Listing 3.21.

### Listing 3.21 Filtering a Collection

[Click here to view code image](#)

```
val numberOfYesVotes = votes.filter { it == true }.co
```

Because the lambda is the `filter`’s only argument, you can skip the parentheses altogether. For a collection of elements of type A, the predicate must be a function of type (A)  $\rightarrow$  Boolean. This can also be used for search. For instance, search countries starting with an “f” as done in Listing 3.22.

### Listing 3.22 Searching a Collection

[Click here to view code image](#)

```
val searchResult = countryToCapital.keys.filter { it.
```

Essentially, the `filter` function filters a collection to one of its subsets where only the elements fulfilling the given predicate remain.

## Mapping Collections

The higher-order function `map` is also well known from functional languages. It allows you to easily apply (or “map”) a given function to each element of a collection. Imagine you want to square each element in the array `testData` as in Listing 3.23.

### Listing 3.23 Applying a Function to Each Element Using `map`

[Click here to view code image](#)

```
val squared = testData.map { it * it } // Creates li
```

Again, you can omit the parentheses for the call to the `map` function because the lambda is the only argument. For a given element, the lambda returns the element squared (note that `it` has type `Int` here). Since this operation will be performed on each element, the `squared` variable will be a list with the elements 0, 1, 25, 81, 100. Notice that `map` always returns a list, even if you begin with a different collection type. You can see this looking at its signature given in [Listing 3.24](#).

#### **Listing 3.24 Signature of the `map` Function**

[Click here to view code image](#)

---

```
inline fun <T, R> Iterable<T>.map(transform: (T) -> F
```

The `map` function can also perform a projection (similar to SQL<sup>5</sup>), where a collection of objects is mapped to a collection of only relevant attributes, as shown in [Listing 3.25](#).

5. <http://www.cs.uakron.edu/~echeng/db/sequel.pdf>

#### **Listing 3.25 Projections Using `map`**

[Click here to view code image](#)

---

```
val countries = countryToCapital.map { it.key } // [
```

Here, each element in the collection has the type `Entry<String, String>`, so this is the type of `it` in this example. Each of those entries consists of a key and a value, which is projected to only the key, which contains the country name.

## **Grouping Collections**

You can group a collection's elements by some key using the higher-order function `groupBy`. For instance, you could group people by their first name, users by their status, or animals by their age. [Listing 3.26](#) gives an example that groups words by their length.

#### **Listing 3.26 Grouping Words by Length**

[Click here to view code image](#)

```
val sentence = "This is an example sentence with several words of varying lengths."  
  
val lengthToWords = sentence.split(" ") // ["This", "is", "an", "example", "sentence", "with", "several", "words", "of", "varying", "lengths."]  
  
.groupBy { it.length }  
  
println(lengthToWords) // {4=[This, with], 2=[is, an], 5=[example, sentence], 6=[several, words], 9=[varying, lengths.], 3=[of], 7=[lengths]}
```

First, `split` turns the sentence into a list of its words by splitting the string at every space, and then the `groupBy` function groups the words by their length. In general, `groupBy` transforms any `Iterable` into a map, and the keys for that map are defined by the lambda you pass to `groupBy`. Here, the keys are the word lengths because the lambda uses `it.length` as the key.

## Associating Collections

Another higher-order function that turns an iterable object into a map is `associate`. This one lets you associate two different parts of your data to each other. For instance, you could associate your users' ages to their average shopping cart value or your users' subscription plans to the time they spend on your site. Another use case is simply transforming a list into a map. Listing 3.27 does this and uses incrementing integers as the map's keys.

**Listing 3.27** Turning a List into a Map

[Click here to view code image](#)

```
val words = listOf("filter", "map", "sorted", "grouped", "associate")  
  
var id = 0 // The first word is mapped to 0  
  
val map = words.associate { id++ to it } // Associate each word with its ID  
  
println(map) // {0=filter, 1=map, 2=sorted, 3=grouped, 4=associate}
```

Inside the lambda, you define the pairs that make up the resulting map. Here, each word is mapped to a pair with the next ID as its key and the word as its value.

## Calculating a Minimum, Maximum, and Sum

If you have a list of integers named `numbers`, you can simply use `numbers.min()`, `numbers.max()`, and `numbers.sum()` in Kotlin to calculate their minimum, maximum, and sum, respectively. But what if you have a list of users and want to get the one with minimum age, get the one with maximum score, or calculate the sum of their monthly payments? For these cases, Kotlin has higher-order functions called `minBy`, `maxBy`, and `sumBy`. Listing 3.28 assumes you have users with an `age`, `score`, and `monthlyFee` to implement these use cases.

**Listing 3.28 Minimum, Maximum, and Sum by Some Metric**

[Click here to view code image](#)

```
users.minBy { it.age }          // Returns user object
users.maxBy { it.score }        // Returns user object
users.sumBy { it.monthlyFee }   // Returns sum of all
```

As you can see, these allow for a readable and terse syntax for such use cases. Note that `minBy { it.age }` is not equivalent to `map { it.age }.min()` because the latter doesn't return the whole user but just the minimum age that was found.

## Sorting Collections

Kotlin provides several functions to sort collections, some of which accept a function as a parameter. They are shown in Listing 3.29.

**Listing 3.29 Sorting Collections**

[Click here to view code image](#)

```
languages.sorted()              // ["C++", "Java", "Kotlin"]
languages.sortedDescending()    // ["Kotlin", "Java", "C++"]
testData.sortedBy { it % 3 }     // [0, 9, 1, 10, 5]
```

```
votes.sortedWith(  
    Comparator { o1, _ -> if (o1 == true) -1 else 1 }  
    countryToCapital.toSortedMap() // {France=Paris, Ger
```

All of the methods used above do not change the original object on which they are called. In other words, they do not perform the sort in place but create a new collection and return that instead. For lists, sets, and arrays, these sorting methods all return a list. For arrays, there is also `sortedArray` that returns an array. For maps, as in the last line, there is only `toSortedMap`, which returns a `SortedMap`—a map that is sorted by the natural ordering on its keys by default (but can also accept a `Comparator`).

The default sorting operations in the first two lines use natural ordering, in this case the lexicographical ordering on strings.

For more special use cases, the method `sortedBy` allows the developer to define a custom value by which to sort. [Listing 3.29](#) sorts the elements in `testData` by their remainder when dividing by three. In general, the function you pass in must take in an element of the collection and return some subtype of `Comparable`; here `(Int) -> Int` is used.

Lastly, `sortedWith` accepts a `java.util.Comparator` that defines how two elements ought to be compared. The way this works is that you compare two objects and return a negative number if the first is less than the second and vice versa. Here, affirmative votes are defined to be “less than” negative votes, thus sorting them to the beginning. Notice that, since you do not need the second vote to compute the result, you can use an underscore in place of the lambda argument to signify that the second argument is not used.

For arrays, there are also in-place sorting operations following the same structure of method names but prefixed with `sort` instead of `sorted`, for instance `sortDescending` and `sortBy`.

## Folding Collections

The `fold` function is used to transform a collection into a value by iteratively applying a function to two adjacent elements at a time, starting with a given initial value and the first element in the collection, then progressing through the collection from the “left” (the start).

To make this more tangible, consider the implementations of calculating the sum and product of a collection shown in [Listing 3.30](#).

### **Listing 3.30 Calculating Sum and Product Using `fold`**

[Click here to view code image](#)

```
val testData = arrayOf(0, 1, 5, 9, 10)

val sum = testData.fold(0) { acc, element -> acc + e}

val product = testData.fold(1) { acc, element -> acc
```

In these examples, the type of the accumulator is the same as the element type—both are integers. In general, any type of accumulator can be used. The first line will result in the computation of  $((((0 + 0) + 1) + 5) + 9) + 10$ . The name `fold` comes from the fact that you can imagine this operation to fold the collection element-wise, each time applying the given function to the hitherto accumulated value and the next element.

First, the additional zero start value is folded onto the first collection element, resulting in the computation of  $(0 + 0) = 0$ . This new value is folded onto the second element, thus calculating  $(0 + 1) = 1$ . Next comes  $(1 + 5) = 6$ , and so forth. A common way to visualize this behavior is shown in [Figure 3.1](#), where the colon stands for appending an element, and `[ ]` represents an empty array.

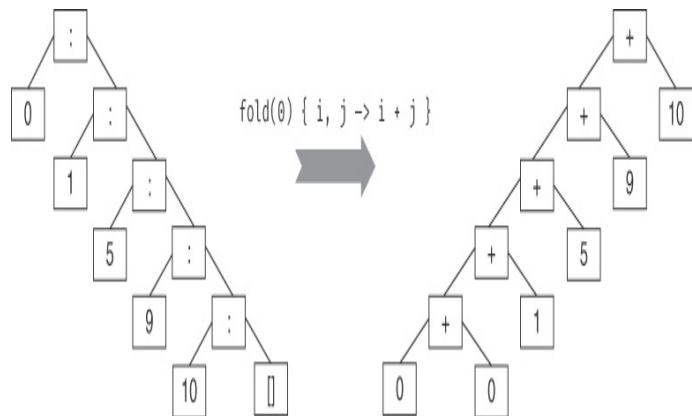


Figure 3.1 Visualization of the `fold` function

The left-hand side of the diagram represents the  `testData` array with elements 0, 1, 5, 9, and 10. The right-hand side shows the computation tree that  `fold` goes through, from bottom to top. Because this operation folds the collection from left to right, it is also known as *left fold*. Kotlin also provides a function  `foldRight`, which works similarly but folds the collection from right to left. For the two examples above, both operations deliver the same result because both addition and multiplication are commutative operations:  $((((0 + 0) + 1) + 5) + 9) + 10$  is equal to  $(0 + (0 + (1 + (5 + (9 + 10))))))$ .

In abstract terms, the signature of the `fold` function on an `Iterable<E>` can be denoted as  $(R, (R, E) \rightarrow R, Iterable<E>) \rightarrow R$  where `E` is the type of element stored in the collection and `R` is the accumulator type of `fold`. The first parameter is a start value, which is combined with the first collection element for the first `fold` step. The second parameter is a function that defines the next value based on the accumulated value so far (of type `R`) and the next collection element of type `E`, returning the next accumulator value. The last parameter is the iterable that `fold` is called on, and ultimately it returns the accumulated value of type `R`.

**Note**

Kotlin also offers the functions `reduce` and `reduceRight`, which are very similar to `fold` and `foldRight` but work without an additional start value; they begin by folding the first (or last) two collection elements onto each other, respectively.

For the examples above, `reduce` is actually preferable because adding an additional zero or multiplying by one are redundant. However, to have a constant offset, for instance, the folding start value can be used.

## Chaining Function Calls

In actual applications that manipulate collections, it is common to see chains of calls to higher-order functions, effectively forming a pipeline of computations. This is possible because these functions again return their modified collection so that the following function can work with it. Consider having a long list of `User` objects stored in a `users` variable, and you want to print the user names of the first ten active users, as shown in Listing 3.31.

**Listing 3.31 Chaining Functions**

[Click here to view code image](#)

```
val activeUserNames = users.filter { it.isActive } /  
    .take(10) //  
    .map { it.username } /
```

The implicit `it` variable and the possibility of chaining the function calls allows you to write extremely concise code. Note that `take` returns the first `n` elements of an `Iterable`. We will discuss this in more detail in the section on lazy sequences.

In the Kotlin standard library, there are more useful higher-order functions defined that facilitate working with collections and can be used in a similar way. In the section about lazy sequences, we will discuss performance concerns for chains of function calls.

# SCOPING FUNCTIONS

This section introduces Kotlin's scoping functions from the file `Standard.kt` in the standard library. It also covers the `use` function defined in `Closeable.kt` because it is similar to the scoping functions. All these functions help avoid common sources of errors, structure your code, and avoid code duplication.

## Using `let`

The `let` function is a useful tool for variable scoping and for working with nullable types. It has three main use cases.

- Scoping variables so that they are only visible inside a lambda
- Executing some code only if a nullable variable is not `null`
- Converting one nullable object into another

First, variable scoping can be used whenever a variable or object should only be used in a small region of the code, for instance, a buffered reader reading from a file, as shown in [Listing 3.32](#).

[Listing 3.32 Scoping with `let`](#)

[Click here to view code image](#)

```
val lines = File("rawdata.csv").bufferedReader().let
    val result = it.readLines()
    it.close()
    result
}
```

// Buffered reader and 'result' variable are not visible outside this block

This way, the buffered reader object and all variables declared inside the `let` block (the lambda expression) are only visible inside that block. This avoids unwanted access to the reader from outside this block. Notice that, similar to `if` and `when` expressions, `let` returns the value defined in the last line of

the lambda. This example returns a list of all lines in a file and must store that into a temporary variable in order to call `close` in between and close the file—we will improve this later with `use`.

Second, `let` is useful to work with nullable types because, when called on a nullable object using the safe call operator, the `let` block will only be executed if that object is not `null`. Otherwise, the `let` block will simply be ignored. Imagine trying to fetch weather data with a network call that may return `null`, as illustrated in Listing 3.33.

**Listing 3.33 Handling Nullables with `let`**

[Click here to view code image](#)

```
val weather: Weather? = fetchWeatherOrNull()

weather?.let {
    updateUi(weather.temperature) // Only updates UI if
} ▶
```

When using `let` with a safe-call operator like this, then inside the `let` block, the Kotlin compiler helps you by automatically casting the `weather` variable to a non-nullable type so that it can be used without further addressing nullability. If the weather data could not be fetched, the lambda passed to `let` is not run. Thus, the UI will not be updated.

Lastly, you can convert one nullable object into another with the same construct as in Listing 3.33 by calling `let` on the original nullable object and defining the desired converted value in the last line inside the `let` block. This last line defines the return value of `let`.

This is the first time you are using higher-order functions and lambda expressions in a way that looks as if `let` was a keyword of the language, stemming from the syntax without parentheses and with curly braces containing a block of code.

It is important to keep in mind that these are regular higher-order functions, and that you can look at their declaration at any time.

## Using apply

The higher-order function `apply` has two primary use cases.

- Encapsulating multiple calls on the same object
- Initializing objects

[Listing 3.34](#) shows how to encapsulate multiple calls on the same object into an own code block. In the following subsections, let's assume `countryToCapital` was a mutable map, not a read-only one, so that you can add new elements to it.

[Listing 3.34 Using apply](#)

[Click here to view code image](#)

```
countryToCapital.apply {  
    putIfAbsent("India", "Delhi") // Accesses 'country'  
    putIfAbsent("France", "Paris")  
}  
  
println(countryToCapital) // {"Germany"="Berlin", "F
```

With `apply`, you do not refer to `it` because the lambda you pass in is executed like an extension function on the object that you call `apply` on, here `countryToCapital`. Thus, you can write the code inside the `apply` block as if it were part of the `MutableMap` class. You could write

`this.putIfAbsent`, but using `this` as a prefix is optional.

Most commonly, `apply` is used for object initialization as in

[Listing 3.35](#).

[Listing 3.35 Initializing Objects with apply](#)

[Click here to view code image](#)

```
val container = Container().apply { // Initializes container

    size = Dimension(1024, 800)

    font = Font.decode("Arial-bold-22")

    isVisible = true

}
```

This approach uses the fact that `apply` first runs any code in its lambda and then returns the object that it was called on. Thus, the `container` variable includes any changes made to the object inside the lambda. This also enables chaining `apply` with other calls on the object, as shown in [Listing 3.36](#).

[Listing 3.36 Return Value of apply](#)

[Click here to view code image](#)

```
countryToCapital.apply { ... } // 'apply' returns modified object

    .filter { ... } // 'filter' works with the modified object

    .map { ... } // 'map' works with the modified object
```

Using this structure, coherent blocks of code that operate on the same object are clearly highlighted in the code through their indentation, plus there is no need to repeat the variable name for each call.

## Using `with`

The `with` function behaves almost like `apply` and has two major use cases.

- To encapsulate multiple calls on the same object
- To limit the scope of a temporary object

In contrast to `apply`, it returns the result of the lambda, not the object on which it was called. This is shown in [Listing 3.37](#).

**Listing 3.37 Using with for Multiple Calls on Same Object**

[Click here to view code image](#)

```
val countryToCapital = mutableMapOf("Germany" to "Ber  
  
val countries = with(countryToCapital) {  
    putIfAbsent("England", "London")  
    putIfAbsent("Spain", "Madrid")  
    keys // Defines return value of the lambda, and th  
}  
  
println(countryToCapital) // {Germany=Berlin, England=London, Spain=Madrid}  
println(countries) // [Germany, England, Spain]
```

This shows that the return value of `with` is the return value of the lambda you pass in—defined by the last line inside the lambda. Here, the expression in the last line is the keys of the map, so the `with` block returns all countries stored as keys in the map.

Similar to `let`, you can limit the scope of an object to only the lambda expression if you create that object ad-hoc when passing it into `with`. This is demonstrated in Listing 3.38.

**Listing 3.38 Using with to Minimize Scope**

[Click here to view code image](#)

```
val essay = with(StringBuilder()) { // String builder  
    appendln("Intro")  
    appendln("Content")  
    appendln("Conclusion")  
    toString()  
}
```

Note that `with` is superior to `apply` in this case because you don't want to get the string builder back as a result but rather the lambda's result. Builders like this are the most common example, where `with` is preferable to `apply`.

## Using run

The `run` function is helpful to:

- Work with nullables as with `let` but then use `this` inside the lambda instead of `it`.
- Allow immediate function application.
- Scope variables into a lambda.
- Turn an explicit parameter into a receiver object.

First, `run` can be seen as a combination of `with` and `let`. Imagine that `countryToCapital` was nullable, and that you want to call multiple methods on it, as shown in Listing 3.39.

**Listing 3.39 Using run with Nullables**

[Click here to view code image](#)

```
val countryToCapital: MutableMap<String, String>?  
    = mutableMapOf("Germany" to "Berlin")  
  
val countries = countryToCapital?.run { // Runs block  
    putIfAbsent("Mexico", "Mexico City")  
    putIfAbsent("Germany", "Berlin")  
    keys  
}  
  
println(countryToCapital) // {Germany=Berlin, Mexico=Mexico City}  
println(countries) // [Germany, Mexico]
```

It fulfills the same purpose as `with` in Listing 3.37 but skips all operations in the lambda if the map is `null`.

Next, you can use `run` for immediate function application, which means plainly running a given lambda. This is shown in Listing 3.40.

#### Listing 3.40 Using `run` for Immediate Function Application

[Click here to view code image](#)

```
run {  
    println("Running lambda")  
  
    val a = 11 * 13  
  
}
```

This example already hints at the next use case, namely minimizing the scope of temporary variables to a lambda expression, as done in Listing 3.41.

#### Listing 3.41 Using `run` to Minimize Variable Scope

[Click here to view code image](#)

```
val success = run {  
  
    val username = getUsername() // Only visible inside run  
  
    val password = getPassword() // Only visible inside run  
  
    validate(username, password)  
  
}
```

Here, `username` and `password` are only accessible where they really have to be. Reducing scopes like this is a good practice to avoid unwanted accesses.

Lastly, you can use `run` to transform an explicit parameter into a receiver object if you prefer working with it that way. This is demonstrated in Listing 3.42, which assumes a `User` class that has a `username`.

#### Listing 3.42 Using `run` to Turn Parameter into Receiver

[Click here to view code image](#)

```
fun renderUsername(user: User) = user.run {      // The code inside the run block runs in the context of the user
    val premium = if (paid) " (Premium)" else ""    // A local variable
    val displayName = "$username$premium"           // Accessed from the run block
    println(displayName)
}
```

Inside the lambda expression, you can access all members of the `User` class directly (or via `this`). This is especially useful if the parameter name is long and you have to refer to it often inside the function.

## Using also

This last higher-order function from `Standard.kt` has two main use cases.

- Performing ancillary operations like validation or logging
- Intercepting function chains

First, consider [Listing 3.43](#) for an example that performs validation using `also`.

[Listing 3.43 Using also for Validation](#)

[Click here to view code image](#)

```
val user = fetchUser().also {
    requireNotNull(it)
    require(it!!.monthlyFee > 0)
}
```

First, the `fetchUser` function is called and may return `null`. After that, the lambda is executed, and only if it runs through is the variable assigned to the user. The usefulness of this function for logging and other operations “on the side” only becomes apparent in a chain of function calls, as in [Listing 3.44](#).

#### Listing 3.44 Using also for Ancillary Operations in a Function Chain

[Click here to view code image](#)

```
users.filter { it.age > 21 }

    .also { println("${it.size} adult users found.")

        .map { it.monthlyFee }
```

Outside of such chains, there's no direct need to use `also` because you could do logging and other side operations on the next line. In call chains, however, `also` lets you intercept the intermediate results without breaking the chain.

### Using use

The `use` function is not part of `Standard.kt` but has a similar structure and benefits. It ensures that the resource it is called on is closed after the given operations are performed. For this reason, it is only defined for subtypes of `Closeable` like `Reader`, `Writer`, or `Socket`.

Using this, you can improve the code from [Listing 3.32](#) as shown in [Listing 3.45](#).

#### Listing 3.45 Handling Closeables with use

[Click here to view code image](#)

```
val lines = File("rawdata.csv").bufferedReader().use
```

Since the buffered reader is now closed automatically at the end of the `use` block, there is no need to store the result in a temporary variable, and you can reduce the lambda to a single line. Apart from ensuring `close` is called, `use` is just like `let`. Internally, it uses a `finally` block to ensure the `Closeable` instance is ultimately closed.

#### Java Note

The `use` function is equivalent to Java's `try-with-resources` (introduced in Java 7).

## Combining Higher-Order Functions

The power of higher-order functions partly stems from the possibility to chain them. You've already seen several examples of this. Here, we want to give two more examples to help you familiarize yourself with the functional style. First, [Listing 3.46](#) combines Kotlin's scoping functions to build a SQL query using some `SqlQuery` class.

**Listing 3.46 Combining Scope Operators**

[Click here to view code image](#)

```
val sql = SqlQuery().apply { // 'apply' initializes
    append("INSERT INTO user (username, age, paid) VALUES")
    bind("johndoe")
    bind(42)
    bind(true)
}.also {
    println("Initialized SQL query: $it") // 'also' intercepts the chain
}.run {
    DbConnection().execute(this) // 'run' applies the query
}
```

As recommended in this section, this code uses `apply` to initialize the `SqlQuery` object, then intercepts the chain using `also` to log the resulting query, and then executes the query using `run`. This way, the scopes of the `SqlQuery` and `DbConnection` objects are restricted so that they're not accessible from outside the function chain.

Beyond this, you can combine the scope operators with the other higher-order functions as well. [Listing 3.47](#) gives an example of how this works with a map (that maps authors to their list of books) to find all authors that have at least one book stored in the map.

#### Listing 3.47 Combining All Higher-Order Functions

[Click here to view code image](#)

```
val authors = authorsToBooks.apply {  
    putIfAbsent("Martin Fowler", listOf("Patterns of Er  
    }.filter {  
        it.value.isNotEmpty()  
    }.also {  
        println("Authors with books: ${it.keys}")  
    }.map {  
        it.key  
    }  
}
```

Here, first `apply` is used to add another author if it doesn't exist already, then it filters out authors whose list of books is empty, prints all found authors using `also`, and finally transforms the map to a list of authors using `map`.

### Lambdas with Receivers

The principle underlying the higher-order functions `apply`, `with`, and `run`—where you refer to the object it was called on via `this` instead of `it`—is called *lambdas with receivers*. Here, *receiver* refers to the object on which the function is called, for instance `authorToBooks` in the case of `apply` in Listing 3.47. The lambda expression that you pass in as an argument is attached to the receiver. This lets you effectively write your lambda expression as if you were writing it inside the receiver type: You can access members of the receiver class directly. In other words, you can write the lambda as if it was an extension function on the receiver. In fact, lambdas with receivers use a syntax similar to extension functions, as shown in Listing 3.48.

#### Listing 3.48 Signatures of `let` and `run`

[Click here to view code image](#)

```
fun <T, R> T.let(block: (T) -> R): R = block(this) /  
fun <T, R> T.run(block: T.() -> R): R = block()    //
```

The signature difference between the two is only in the input of the lambda parameter. Both `let` and `run` are defined as extension functions on a generic type `T`. However, `let` accepts a function (`block`) that has `T` as a *parameter*. It calls that function on the object of type `T` on which `let` was called. Thus, the argument is accessible as `it` from inside the lambda.

For `run`, on the other hand, the passed-in lambda is declared with `T` as its *receiver* using the syntax `T.() -> R`. This effectively makes the lambda an extension function on `T` and you can implement the lambda like an extension function, meaning you can access members of `T` via `this`. This allows the implementation of `run` to call `this.block()` or simply `block()`.

This is one way to differentiate between the five scoping functions. In fact, there are three dimensions you can differentiate.

- Whether the scoping function's parameter uses a lambda with receiver or a “normal” lambda with parameter—and therefore if you use `this` or `it` inside the lambda, respectively
- Whether the scoping function returns the object on which it was called or the lambda's return value
- Whether the scoping function *itself* is an extension function or accepts a parameter (`with` is the only one that accepts an explicit parameter)

Figure 3.2 provides a visual representation of this as a reference to quickly recap the differences between the scope functions.

Lambda's input	Extension function of T?	
Receiver	apply	run
Parameter	also	let
Output	Object it was called on	Result of the lambda

Figure 3.2 Overview of scoping functions

## LAZY SEQUENCES

Having covered higher-order functions and lambda expressions, the last cornerstone of functional programming that is left to discuss is lazy evaluation. More specifically, this section covers *sequences* in Kotlin —a data structure using lazy evaluation.

### Java Note

As mentioned, Kotlin's sequences work the same way as *streams* from Java 8. The reason behind reinventing the wheel here instead of reusing Java's streams was to support them on all platforms, even those that do not support Java 8 (primarily Android).

### Lazy Evaluation

The concept of *lazy evaluation* is all about evaluating expressions only if and when it becomes necessary at runtime. This is in contrast to *eager evaluation*, where every expression is eagerly evaluated even if the result or part of it is never used.

The main benefit of lazy evaluation is that it can improve performance when working with large collections or when performing expensive operations on them when only part of the results are actually required. This performance benefit stems from two things: avoiding unnecessary computations and avoiding the creation of list objects to hold intermediate results.

[Listing 3.49](#) shows a simple example in which `animals` could be a normal collection (list, set, or map) using eager evaluation or a sequence using lazy evaluation. How these would behave differently is explained below, but the way you can work with them using higher-order functions is the same in either case.

**Listing 3.49 Filtering and Mapping a Collection or Sequence**

[Click here to view code image](#)

```
// 'animals' stores the strings "Dog", "Cat", "Chicken", "Cow"  
// 'animals' may be an eager collection or a lazy sequence  
animals.filter { it.startsWith("C") }  
    .map { "$it starts with a 'C'" }  
    .take(1)
```

This code filters the animals to keep only those starting with a “C,” maps these to a different string, and finally takes the first result. Let’s explore how this code behaves in both eager evaluation and lazy evaluation:

- In eager evaluation, if `animals` is a collection or an array, the code would first filter all four elements and then store the intermediate result in a newly created list object. After that, it performs the `map` on each element of this intermediate result and produces another intermediate list. Finally, it takes the first element from the mapped list. In total, two intermediate objects are created, and four filter operations and two map operations are performed—even though ultimately just one element is used.
- In lazy evaluation, if `animals` is a sequence, each element goes through the function chain one by one. Thus, it would first filter out “Dog” because it doesn’t start with a “C” and immediately proceed with the next element. It then moves on to “Cat”, which passes the filter. Then, it maps this element and calls `take(1)`. With this, the query is fulfilled and no more operations are performed—the last two elements are not touched.

Kotlin’s main language feature supporting lazy evaluation like this is sequences.

## Using Lazy Sequences

This subsection covers three ways to get hold of a sequence. First, there is a helper function `sequenceOf`, consistent with the helpers that create collections. It is shown in [Listing 3.50](#).

### **Listing 3.50 Lazy Sequence from Scratch**

[Click here to view code image](#)

```
val sequence = sequenceOf(-5, 0, 5)  
println(sequence.toString()) // -5, 0, 5
```

Second, when you already have a (potentially large) collection, you can transform this collection into a sequence by calling `asSequence`. All the higher-order functions discussed in this chapter are available for sequences as well. For instance, imagine you had a list of all large cities in the world and want to print those cities whose names start with “W,” as shown in [Listing 3.51](#).

### **Listing 3.51 Lazy Sequence from Collection**

[Click here to view code image](#)

```
val output = cities.asSequence() // Transforms eager  
    .filter { it.startsWith("W") }  
    .map { "City: $it" }  
    .joinToString()  
  
println(output) // City: Warsaw, City: Washington, C
```

There are several things to notice in [Listing 3.51](#). First, the call to `asSequence` transforms the normal collection into a lazy sequence to make sure all operations are performed lazily. Second, the rest looks just like it did for normal, eagerly evaluated, collections. Third, you differentiate between *intermediate* and *terminal* operations. Intermediate operations are all operations that again return a sequence, such as

`filter`, `map`, `sort`, and `fold`. In contrast, terminal operations are typically the last operation in a chain and may return anything but a sequence. Here, `joinToString` is called as the terminal operation to retrieve a string. Other common terminal operations include `toList`, `toSet`, `toMutableList`, `toMutableSet`, `first`, `last`, `min`, and `max`. Without a terminal operation, a lazy sequence performs no computations at all.

The third way in which lazy sequences may be used is to create a sequence from the get-go instead of transforming an existing collection into a sequence. For this, the helper function `generateSequence` is used, which takes in a seed element and a function to calculate the next element based on its predecessor, as shown in Listing 3.52.

**Listing 3.52 Lazy Sequences with `generateSequence`**

[Click here to view code image](#)

```
val naturalNumbers = generateSequence(0) { it + 1 }

val integers = generateSequence(0) { if (it > 0) -it
```

The first line uses zero as the seed element to start the sequence. Each next element adds one to its predecessor, resulting in the sequence 0, 1, 2, 3, ... of natural numbers. The second line takes it one step further to define the sequence 0, 1, -1, 2, -2, 3, -3, ... of all integers. This shows that each element in the sequence is *calculated* based on the previous one, and only when necessary.

**Note**

With this, you defined your first *infinite* sequence. This is only possible due to lazy evaluation; there is no way to actually store a data structure of all natural numbers in memory.

## Take and Drop

The functions `take` and `drop` are simple but essential, and it is worth noting how to best use them. As briefly mentioned, `take(n)` returns the first  $n$  elements of a collection or sequence. Its counterpart `drop(n)` returns the tail of the sequence without its first  $n$  elements, as shown in Listing 3.53. Note that both leave the original sequence unchanged.

### **Listing 3.53 Using take and drop**

[Click here to view code image](#)

When chaining function calls, especially when they contain `take` or `drop` calls, order of execution can greatly affect performance. It is a good practice to reduce the size of large collections early on to save unnecessary operations—and this is what `take` and `drop` do. Although this is also a good practice when working with lazy sequences, it is crucial when using eager collections because these will otherwise perform all operations on all elements. As an example, reconsider Listing 3.51 and say you only want to print the first 20 cities that match your selection, as done in Listing 3.54.

### Listing 3.54 Using take (and drop) Early

[Click here to view code image](#)

```
// Not good

cities.filter { it.startsWith("W") }

    .map { "City: $it" } // Calls map before take

    .take(20)           // Takes only the first 20

    .joinToString()
```

```
// Better

cities.filter { it.startsWith("W") }

    .take(20)           // Reduces the size of the list

    .map { "City: $it" } // Calls map at most 20 times

    .joinToString()
```

You cannot place the `take` call before the filter because you want the first 20 cities starting with “W.” But you can place it before the `map` call. For instance, if 2,000 cities make it through the filter, this avoids 1,980 unnecessary applications of the `map` function. Thus, the performance benefits depend on how many operations can be saved and how early in the chain the collection can be shrunk. If you can greatly reduce the size early in the chain, eager collections are likely to perform better than lazy sequences. In other cases, lazy sequences can be the better choice.

## Performance of Lazy Sequences

To illustrate the difference between lazy sequences and eager collections, let us look again at the minimal example in Listing 3.55 that performs a chain of functions on a list of cities.

### **Listing 3.55 Eager Evaluation on Collections**

**Click here to view code image**

If using normal collections, each function in the chain is fully evaluated, the intermediate result is stored in an object, and

only then is the next function executed.

1. First, `filter` iterates over the whole collection, leaving only Washington and Worcester. These are stored in a new list as an intermediate result.
2. Then, `map` is called on this intermediate list of two elements, mapping each to a string with a "City: " prefix. Another list is created to store this next intermediate result.
3. Finally, `take` returns another newly created list as the result.

Thus, this produces the output shown in [Listing 3.56](#) due to eager evaluation.

#### [Listing 3.56 Output for Eager Evaluation](#)

[Click here to view code image](#)

```
filter: Washington
filter: Houston
filter: Seattle
filter: Worcester
filter: San Francisco // Eagerly filtered all elements
map: Washington
map: Worcester
```

With sequences, the processing is very different. Each element goes through the chain one by one. [Listing 3.57](#) shows the same example using sequences.

#### [Listing 3.57 Lazy Evaluation on Sequences](#)

[Click here to view code image](#)

```
val cities = listOf("Washington", "Houston", "Seattle")
cities.asSequence()
    .filter { println("filter: $it"); it.startsWith("W") }
    .map { println("map: $it"); "City: $it" }
    .take(2) // Should still better be called before map
    .toList()
```

---

Here, a call to `asSequence` is included and also a call to `toList` as the terminal operation. Without such a terminal operation, no computation would be performed due to the laziness. At this point, the `take` function comes into play. The code in [Listing 3.57](#) results in the output shown in [Listing 3.58](#).

**Listing 3.58 Output for Lazy Evaluation**

[Click here to view code image](#)

---

```
filter: Washington
map: Washington // Passes element on to next step in
filter: Houston
filter: Seattle
filter: Worcester
map: Worcester // Two elements found, "San Francisco"
```



First, you can see that each element goes through the chain of functions one by one. Thus, there is no need to store intermediate results. Second, it becomes obvious that `map` is only performed on elements passing the filter (this is also the case in eager evaluation). Lastly, due to taking only two elements, the last element (San Francisco) is not processed at all. This is different from eager evaluation and is the second reason why sequences can improve performance.

Now it becomes clear why sequences tend to improve performance for large collections or when performing expensive operations that are partly unnecessary.

- The larger the collection is, the larger are the intermediate list objects created after each step in eager evaluation. Lazy evaluation creates no such intermediate results.
- The more expensive the operations are, the more computation time can be saved by performing them lazily—thus skipping all unnecessary operations.

These are your rules of thumb regarding when you may want to prefer sequences over normal collections. However, Kotlin

collections are implemented very efficiently. So, which works better in a specific use case should be evaluated beforehand.

To get a rough estimate, Kotlin's built-in function `measureTimeMillis` can be used. This is another higher-order function that takes in a block of code as a lambda expression and returns the time in milliseconds that was needed to run that code block.

## SUMMARY

In this chapter, you explored how functional programming emphasizes immutability and extensive use of functions as first-class members of the language. Several examples of predefined and self-written higher-order functions demonstrated the power of this concept for code modularization and writing concise code.

You saw many examples combining these with lambda expressions to define anonymous functions, typically to create the arguments or return values for higher-order functions. One Kotlin-specific feature surrounding lambdas is the implicit `it` variable for lambdas with only one parameter. Another is the possibility to write lambdas that appear as the last argument outside of the function call parentheses. You saw how this can result in functions that look like language keywords, like many from the Kotlin standard library.

Lastly, this chapter introduced the concept of lazy evaluation and Kotlin's lazy sequences as one example of this. You saw how their performance compares to normal collections and learned in which cases you may prefer sequences over collections.

## Object Orientation in Kotlin

*Complexity has nothing to do with intelligence, simplicity does.*

Larry Bossidy

Object orientation (OO) has become a fixture in software development since its incarnation in languages like Simula and Smalltalk in the 1960s and 1970s. This has led to a majority of developers who tackle problems mostly in an object-oriented way, and many modern languages continue to incorporate the concepts of OO, often combined with other paradigms. Kotlin incorporates both OO and functional paradigms. In this chapter, you learn this second pillar of the language. Some of the presented concepts are known from other object-oriented languages. But Kotlin introduces various useful features that languages like C# and Java don't have, enabling Kotlin users to write programs quickly and with little boilerplate code.

## CLASSES AND OBJECT INSTANTIATION

Basic class declarations are introduced using the **class** keyword followed by the name of the class. In Kotlin, there's no **new** keyword for object instantiation, as demonstrated in Listing 4.1.

**Listing 4.1 Class Declaration and Instantiation**

[Click here to view code image](#)

---

```
class Task {
```

```
// Implement class here...  
}  
  
val laundry = Task() // Instantiates object of type
```

With this, you already saw the two most fundamental entities in OO: classes and objects. Classes act as blueprints from which you can instantiate specific objects. However, without any properties that differ between each object, this is hardly useful. So let us introduce some properties in the `Task` class.

## PROPERTIES

To explore how properties work in Kotlin, and how you can best add them to your class, let us start with a non-ideal way to implement a class with a property as it is done in other languages (shown in [Listing 4.2](#)), and then refactor it step by step to idiomatic Kotlin code.

### [Listing 4.2 Adding a Property the Hard Way](#)

[Click here to view code image](#)

```
class Task {  
  
    val title: String // Declares a property  
  
    constructor(title: String) { // Defines a constructor  
        this.title = title // initializes the property  
    }  
  
    val dishes = Task("Wash dishes") // Calls constructor  
    val laundry = Task("Do laundry") // Calls constructor
```

Here, the class contains a member property `title`, which is initialized inside a constructor that accepts a `String`. The constructor is run whenever you instantiate a `Task` object. Notice how often you have to mention the title in this code and manually set it in a constructor. To improve this, you can use a *primary constructor* instead in Kotlin. It directly follows the class name, as in [Listing 4.3](#). Constructors are discussed in detail in the section titled, “[Primary and Secondary Constructors](#).”

**Listing 4.3 Using a Primary Constructor**

[Click here to view code image](#)

```
class Task(title: String) { // Primary constructor

    val title: String = title // Initializes property

}

val laundry = Task("Do laundry") // Calls constructor
```

The parameters of the primary constructor can be used directly inside the class body to instantiate any member properties.

Notice also that the compiler can differentiate between the declared property `title` and the constructor parameter of the same name. This code is already a lot more concise, but the idiomatic way uses a shortcut in Kotlin that allows you to “upgrade” a primary constructor parameter to a property, as in [Listing 4.4](#).

**Listing 4.4 Idiomatic Way to Add a Simple Property**

[Click here to view code image](#)

```
class Task(val title: String) // Primary constructor

val laundry = Task("Do laundry")
```

Prefixing a parameter in a primary constructor with either `val` or `var` implicitly makes it a class property and automatically

handles its initialization with the given argument, here "Do laundry". Again, **val** makes the property read-only and **var** makes it mutable. Notice that curly braces can be omitted if the class body is empty.

## Properties versus Fields

What is interesting here is that we're talking about *properties*, not *fields*<sup>1</sup> as in many other languages (including Java). You can think of properties as fields extended with getters and setters that allow access to the underlying field. In other words, fields hold the actual data, whereas properties form part of the class' interface to the outside, allowing access to some of the class' data via getters and setters. In this way, properties are similar to methods because getters and setters are in fact methods. But properties are purely for data access and don't typically perform any logic besides returning or setting their corresponding data.

<sup>1</sup> Some say "attributes," but this term is vague and not used in the Java documentation.

Fields should almost always be private, unless they are static or constant. In Java, you do this manually by marking your field private and defining separate getter and setter methods. In C#, you can explicitly create properties that usually wrap a private field and define a getter and a setter. Both languages add boilerplate code around this simple accessor logic. Most of the time, the code inside these methods follows the same pattern, which is why your IDE can generate them for you.

The main problem with such boilerplate is not the time it takes to write it—your IDE can generate it in a second. The problem is that it distracts from the actual logic, and it's hard to scan for possible deviations from the standard pattern. Auto-properties in C# help avoid much of this boilerplate, and Kotlin doesn't even let you declare fields yourself. Instead, you declare properties. Accessing a property automatically calls its getter or setter, as shown in Listing 4.5.

**Listing 4.5 Property Access Calls Getter or Setter**

[Click here to view code image](#)

```
class Task(var title: String) // Uses 'var' now to t

val laundry = Task("Do laundry")

laundry.title = "Laundry day" // Calls setter

println(laundry.title) // Calls getter
```

The `title` property is now declared as mutable in order to have a setter. Reassigning the property calls its setter, and retrieving its value calls its getter. So while this kind of code would almost always be bad practice when using fields because it would indicate direct field access, it is perfectly in accordance with the principle of information hiding in Kotlin because there is no direct field access, only getter and setter access.

## Getters and Setters

When you want to change the default getter or setter implementation, you can do so using a specialized syntax to override the implicit `get` and `set` methods, as in

[Listing 4.6](#). In this case, the shorthand property syntax in the primary constructor is no longer possible because `get` and `set` must directly follow the declaration of their respective property.

[Listing 4.6 Custom Getter and Setter](#)

[Click here to view code image](#)

```
class Task(title: String, priority: Int) {

    val title: String = title

    get() = field.toUpperCase() // Defines custom get

    var priority: Int = priority

    get() = field // Same as default impl

    set(value) { // Defines custom setter
```

```
    if (value in 0..100) field = value else throw ...  
}  
}  
  
val laundry = Task("Do laundry", 40)  
  
println(laundry.title)           // Calls getter, pri  
laundry.priority = 150          // Calls setter, thr  
  
println(laundry.priority)       // Calls getter, wol
```

Here, the properties are declared inside the class body again, so that they can be followed by their custom getter and setter implementations. Because `title` is a read-only **val**, it only has a getter and no setter. The getter returns the title in uppercase. For the priority, the getter is just the default implementation, and the setter only accepts values between 0 and 100.

Every property has an implicit *backing field* that stores the actual data. Its name is **field**, and it can only be accessed inside **get** or **set**. In fact, you *have* to use it to access the property's value. If you used `get() = title.toUpperCase()` in Listing 4.6 instead, you'd theoretically end up in an infinite recursion because `title.toUpperCase` would again call the getter.

In practice, Kotlin would instead consider the `title` property to have no backing field. Thus, you could no longer initialize it with the `title` from the constructor (as in the second code line). Properties without backing fields calculate their values on the fly, instead of storing their data in a field. Note that the variable name **field** is fixed, whereas the parameter name `value` for the setter is only a convention.

## Late-Initialized Properties

Trying to follow the good practices of using immutability and avoiding nullability is not always easy. Especially the latter is a shift in mindset when migrating from a language without null-safety. One situation where following these principles can look difficult is when you have read-only and non-null properties. These have to be initialized right inside the constructor, but doing this is often inconvenient or just unnecessary. Common examples where initialization directly at declaration site is unnecessary are test cases where objects are initialized by a test setup method, via dependency injection, or in `onCreate` on Android.

In Kotlin, the solution for such situations can often be found in specific language features. The situation above can be solved using late-initialized properties. These allow you to defer the initialization to a later point, as in [Listing 4.7](#).

### [Listing 4.7 Using Late-Initialized Properties](#)

[Click here to view code image](#)

```
class CarTest {  
  
    lateinit var car: Car // No initialization  
  
    @BeforeEach  
  
    fun setup() {  
  
        car = TestFactory.car() // Re-initializes proper  
    }  
  
    // ...  
}
```

This way, there's no need to make the `car` property nullable, making it much easier to work with. The cost for this is that **lateinit** properties must always be mutable **vars** (this is necessary here anyway to reassign it inside the test setup

method). There are more restrictions for properties with which the **lateinit** keyword can be used.

1. The property must be declared inside a class, and not in its primary constructor. Properties in the primary constructor will be initialized anyway, so **lateinit** would be useless. The **lateinit** keyword is for properties that you cannot initialize in the constructor in a useful way.
2. The property must not have a custom getter or setter. With a custom accessor, the compiler could no longer infer whether an assignment actually initializes the property—imagine an empty setter, for instance.
3. The property must be non-nullable. Otherwise, you could simply initialize with **null** instead. The purpose of **lateinit** is to avoid exactly this.
4. The property must not have a type that is mapped to a primitive Java type at runtime, such as **Int**, **Double**, or **Char**. This is because, under the hood, **lateinit** still has to use **null**, and this is impossible with primitive types.

What if you accidentally access the property before it's initialized? The compiler will spit out an **UninitializedPropertyAccessException** with an appropriate error message that leads you to the source of the exception. To check beforehand if it's initialized, you could use **this::car.isInitialized** from within the **CarTest** class. However, you can only use this from scopes that have access to the property's backing field.

## Delegated Properties

Although most property accessors perform no additional logic other than returning or setting a value, there are several types of more sophisticated accessors that are commonly used—for instance, lazy and observable properties. In Kotlin, property accessors that perform such logic can delegate to a separate implementation that provides the logic. This makes the accessor logic reusable because the delegate encapsulates it.

Kotlin achieves this via *delegated properties*, as shown in [Listing 4.8](#).

### [Listing 4.8 Syntax of Delegated Properties](#)

[Click here to view code image](#)

```
class Cat {  
  
    var name: String by MyDelegate() // Delegates to a  
    // separate implementation
```

```
}
```

Here, instead of giving the `name` property a concrete value, you only indicate that getting and setting its value is handled by an object of the class `MyDelegate`. To specify the delegate, you use the `by` keyword. The delegate class must have a method `getValue` and, for mutable properties, also `setValue`. Listing 4.9 shows an implementation of `MyDelegate` that demonstrates the structure. Although it's not required, I recommend you implement the interface `ReadOnlyProperty` or `ReadWriteProperty` for read-only and mutable properties, respectively. `ReadOnlyProperty` requires you to implement `getValue`, and `ReadWriteProperty` additionally requires a `setValue` method. Interfaces were not discussed yet; this listing is meant as a reference for how to create custom delegates. If you do not yet understand all concepts used in the code, you can come back to it after finishing this chapter.

**Listing 4.9 Implementing a Custom Delegate**

[Click here to view code image](#)

```
import kotlin.properties.ReadWriteProperty

import kotlin.reflect.KProperty

class MyDelegate : ReadWriteProperty<Cat, String> {

    var name: String = "Felix"

    // Delegate must have a getValue method (and setVa]

    override operator fun getValue(thisRef: Cat, prop:
        println("$thisRef requested ${prop.name} from MyD

    return name
}
```

```
}

override operator fun setValue(thisRef: Cat, prop:
    println("$thisRef wants to set ${prop.name} to $value")
    if (value.isNotBlank()) {
        this.name = value
    }
}

val felix = Cat()
println(felix.name) // Prints "Cat@1c655221 requested"
felix.name = "Feli" // Prints "Cat@1c655221 wants to"
println(felix.name) // Prints "Cat@1c655221 requested"
```

The exact signature of the interface `ReadWriteProperty` is `ReadWriteProperty<in R, T>`, where `R` is the class that has the property, and `T` is the type of the property (this is a generic class, which is discussed later in this chapter). Here, the property is for the `Cat` class, and it's a `String` property. Hence, it's a `ReadWriteProperty<Cat, String>`. By inheriting from it, you can generate the method stubs in Android Studio using `Ctrl+I` (also on Mac). Notice that both methods must be declared as operator functions in order to be used as a delegate. In contrast, the `override` modifier can be omitted if you choose not to implement any of the interfaces. With this, reading the `name` property of the `Cat` class calls `MyDelegate::getValue` and setting a new value delegates to `MyDelegate::setValue`.

## Predefined Delegates

As mentioned, the motivation behind delegated properties is to encapsulate common accessor logic in a reusable way. Thus, this is exactly what was done in the Kotlin standard library in the form of several predefined delegates.

### Lazy Properties

The first predefined delegate allows you to easily implement lazy properties. These compute their value only when accessed the first time and then they cache it. In other words, the value is not computed until it's used. This is why lazy properties are particularly useful when computing the value is expensive and there's a chance that it's not used. Listing 4.10 augments the `Cat` class to exemplify this.

#### Listing 4.10 Using Lazy Properties

[Click here to view code image](#)

```
import java.time.LocalDate

import java.time.temporal.ChronoUnit

class Cat(val birthday: LocalDate) {

    val age: Int by lazy { // Lazy property, computed

        println("Computing age...")

        ChronoUnit.YEARS.between(birthday, LocalDate.now()

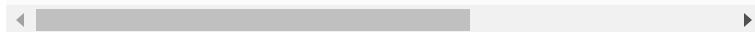
    }

}

val felix = Cat(LocalDate.of(2013, 10, 27))

println("age = ${felix.age}") // Prints "Computing a

println("age = ${felix.age}") // Prints "age = 5"; r
```



Here, the cat's age is lazily computed only when accessed for the first time. The higher-order function `lazy` accepts a lambda expression as its last parameter, thus allowing this syntax. Inside the lambda, you define how the value is computed, and the expression in the last line defines the return value.

The `lazy` function is thread-safe by default. If you don't need thread safety, you can pass in an optional safety mode as the first parameter, for instance, `lazy(LazyThreadSafetyMode.NONE)` for a non-thread-safe implementation.

**Note**

On Android, lazy properties are useful to start up activities more quickly and prevent repetitive code. For example, instead of using `findViewById` explicitly in `onCreate`, you can define your UI properties as lazy:

[Click here to view code image](#)

```
val textView: TextView by lazy { findViewById<TextView>(R.id.title) }
```

This can be further encapsulated into a `bind` extension function:

[Click here to view code image](#)

```
fun <T: View> Activity.bind(resourceId: Int) = lazy { findViewById<T>(resou
```

```
val textView: TextView by bind(R.id.title)
```

You can use lazy properties in the same way for other resources like strings or drawables. Additionally, you can use them to delay creation of heavy objects to the point where they are used. For example, you can use this to avoid performing too many I/O operations on startup and thus prevent the app from not responding or even crashing.

### Observable Properties

Observable properties are another common pattern that can be realized using delegation. In Kotlin, it is predefined as `Delegates.observable`, which accepts an initial value and a change handler function that is executed whenever the property's value changes. In other words, it *observes* the property. Listing 4.11 shows how to use observable properties by further extending the class `Cat`. It also introduces an enumeration class with the possible moods of a cat. Such enumerations are discussed in more detail later in this chapter.

#### **Listing 4.11 Using Observable Properties**

[Click here to view code image](#)

```
import java.time.LocalDate

import kotlin.properties.Delegates

class Cat(private val birthday: LocalDate) {

    // ...

    var mood: Mood by Delegates.observable(Mood.GRUMPY) {
        property, oldValue, newValue -> // Lambda parameter
            println("${property.name} change: $oldValue -> $newValue")
            when (newValue) {
                Mood.HUNGRY -> println("Time to feed the cat..")
                Mood.SLEEPY -> println("Time to rest...")
                Mood.GRUMPY -> println("All as always")
            }
    }
}
```

```
enum class Mood { GRUMPY, HUNGRY, SLEEPY } // Enums

val felix = Cat(LocalDate.of(2013, 11, 27))

felix.mood = Mood.HUNGRY // "mood change: GRUMPY ->
felix.mood = Mood.SLEEPY // "mood change: HUNGRY ->
felix.mood = Mood.GRUMPY // "mood change: ASLEEP ->
```

If you look into the implementation, you'll see that `Delegates.observable` returns a `ReadWriteProperty`. Its first parameter defines the initial value of the property. Here, it is `Mood.GRUMPY`. On each value update, the delegate prints the change. Inside the `when` expression, an action is taken depending on the new value. This is where you'll usually notify any observers that are interested in changes to this property. This is useful to update a UI whenever its underlying data is changed. For instance, delegated properties provide a more natural way to implement view binding.

#### Vetoable Properties

When using observable properties, the handler is executed after the value of the property has already been updated. If you want the same behavior but be able to prevent the new value from being accepted, you can use vetoable properties instead.

For this, you use `Delegates.vetoable`. This works the same way as `Delegates.observable` but you have to return a Boolean value that indicates whether to accept the new value or not. If you return `false`, the property's value will not be updated.

## Delegating to a Map

Lastly, Kotlin allows you to delegate properties to a map, which maps the property names to their respective values. When would you use this? Most commonly when parsing objects from a non-object-oriented representation such as JavaScript Object Notation (JSON). Imagine you're retrieving information about persons in JSON format from an API. Listing 4.12 shows how you can translate this back to an object using a map as a delegate.

**Listing 4.12 Delegating Properties to a Map**

[Click here to view code image](#)

```
class JsonPerson(properties: Map<String, Any>) {

    val name: String by properties // Delegates prop

    val age: Int by properties // Delegates prop

    val mood: Mood by properties // Delegates prop

}

// Let's assume this data comes from JSON; keys in the map are strings
// You may need to preprocess some data (requires Mutation)
jsonData["mood"] = Mood.valueOf(jsonData["mood"] as String)

// Creates an object from the map

val john = JsonPerson(jsonData) // Properties are mapped to the object

println(john.name) // Prints "John Doe"
println(john.age) // Prints 42
println(john.mood) // Prints "GRUMPY"

// Read-only property changes if backing map changes
```

```
jsonData["name"] = "Hacker"  
  
println(john.name) // Prints "Hacker"
```

Using maps is still in line with the premise that delegates must have a `getValue` method. For maps, these accessors are defined as extension functions in the file `MapAccessors.kt`. This way, the property simply delegates its accessors to the map. For instance, `john.name` delegates to `properties["name"]`.

To support mutable properties, you have to use a `MutableMap` as the delegate to have a `setValue` method. However, be aware that even read-only properties that delegate to a mutable map can unexpectedly change their value if the backing map is changed. This is illustrated by the last two lines in [Listing 4.12](#). To prevent this, you could accept a read-only map for the read-only properties and a mutable map for the mutable properties. If changes are not frequent, you could also use only read-only properties and reinstantiate the whole object from the map whenever you want to change a value.

## Using Delegated Implementations

The delegation pattern is not only useful for properties but for method implementations as well. Its idea is that you can implement an interface by delegating to an existing implementation of it instead of implementing it all over again, and you can do so without boilerplate code. As an example, in [Listing 4.13](#), the class `Football` implements a `Kickable` interface by delegating to an existing implementation.

[Listing 4.13 Delegating to an Implementation Using by](#)

[Click here to view code image](#)

```
interface Kickable {  
  
    fun kick()  
  
}
```

```
// Existing implementation of Kickable

class BaseKickHandler : Kickable {

    override fun kick() { println("Got kicked") }

}

// Football implements interface by delegating to existing implementation

class Football(kickHandler: Kickable) : Kickable by kickHandler
```

Here, the `Football` class implements `Kickable` without defining any explicit overrides. Instead, it delegates to an existing implementation using `by`. This makes the compiler generate all required methods and delegate them to the delegate object that is passed into the constructor, here `kickHandler`. [Listing 4.14](#) illustrates the generated code, as well as the code that you'd have to write without native support for delegation.

**Listing 4.14 Delegated Implementation**

[Click here to view code image](#)

---

```
// Implements interface with manual delegation

class Football(val kickHandler: Kickable) : Kickable {

    override fun kick() {

        kickHandler.kick() // Trivial forwarding; necessary for delegation

    }

}
```

This is boilerplate code, and it grows with the number of methods that must be overridden. For instance, if you wanted to have a delegating mutable set that you can safely use for inheritance, you can do so in Kotlin in a single line, as shown in [Listing 4.15](#).

#### Listing 4.15 Delegating Mutable Set

[Click here to view code image](#)

```
open class ForwardingMutableSet<E>(set: MutableSet<E>
```

To achieve this without delegation using `by`, you would have to override 11 methods, all of them just boilerplate. This is another way in which Kotlin allows for more concise and expressive code.

## METHODS

At this point, you know how to add data to your classes and control access to it using getters, setters, and delegated properties. As you know, there are two major components to classes in OO, and data is only one of them. In this section, you'll learn how to add the second component, behavior, to your classes in the form *methods*.

Methods are essentially the same as functions, except that they're nested into a class. Thus, you can apply all you know about functions from the previous chapters here. This includes the syntax; defining parameters and return types; using the shorthand notation, default values, and named parameters; and creating inline, infix, and operator functions. In the case of infix functions, the type of the first parameter is automatically the class containing the function. Listing 4.16 provides an overview.

#### Listing 4.16 Declaring and Calling Methods

[Click here to view code image](#)

```
class Foo {  
  
    fun plainMethod() { ... }  
  
    infix fun with(other: Any) = Pair(this, other)  
  
    inline fun inlined(i: Int, operation: (Int, Int) ->  
        operator fun Int.times(str: String) = str.repeat(t  
    fun withDefaults(n: Int = 1, str: String = "Hello v
```

```
}

val obj = Foo()

obj/plainMethod()

val pair = obj with "Kotlin"

obj/inlined(3, { i, j -> i * j })           // 126

obj/withDefaults(str = "Hello Kotlin") // "Hello Kot

with(obj) { 2 * "hi" }                  // Uses 'with'
```

What differentiates methods from functions is that they live in the realm of a class hierarchy. Their purpose is to manipulate a class' data and provide a well-defined interface to the rest of the system. Being part of a class hierarchy, methods support features like overriding and polymorphism, both of which are cornerstones of OO. *Overriding* means you can redefine the implementations of a parent class in subclasses to adapt their behavior. *Polymorphism* means that if you call a method on a variable, it executes the method definition of the (sub)class that is actually stored in that variable, which may vary from the implementation in the parent type. This behavior is also called *dynamic dispatch*.

## Extension Methods

In Kotlin, you can define extensions inside a class. Let's call these *extension methods*. Recall that, inside an extension function, you can access the properties of the extended class without qualifier (as if you were writing that function inside the extended class). In the case of extension methods, you can additionally access all members of the containing class without qualifier, as all methods can. Thus, you must differentiate between the *dispatch receiver* (the containing class) and the *extension receiver* (the class you extend). If there's a clash because both define a method `foo()`, the extension receiver takes precedence. You can still access the dispatch receiver's `foo()` using a qualified `this@MyClass`. Listing 4.17 demonstrates all this.

### **Listing 4.17 Extension Methods**

**Click here to view code image**

```
class Container {  
  
    fun Int.foo() {  
        // Extends Int  
  
        println(toString())  
        // Calls Int.toString()  
  
        println(this@Container.toString())  
        // Calls Container.toString()  
  
    }  
  
    fun bar() = 17.foo()  
    // Uses explicit receiver  
  
}  
  
Container().bar()  
// Prints '17'
```

Here, the containing class defines an extension method on `Int` called `foo` that first calls `toString` without qualifier—equivalent to calling `this.toString`. Because both `Container` and `Int` have a `toString` method (every class inherits one), the extension receiver (`Int`) takes precedence.

After that, the dispatch receiver's `toString` implementation is called using the qualified `this@Container`.

**Tip**

Extension methods are useful to limit the scope of extensions to a class (and its child classes). As projects grow, extensions declared on top-level become increasingly problematic because they pollute the global namespace with lots of potentially irrelevant functions—autocomplete takes longer and becomes less helpful. Limiting scope is a good practice in general and crucial when working with extensions.

## Nested and Inner Classes

Kotlin not only lets you create nested classes, it even provides a convenient way to let them access properties and methods from their containing class: so-called *inner classes*, as demonstrated in Listing 4.18.

**Listing 4.18 Comparing Nested and Inner Classes**

[Click here to view code image](#)

```
class Company {  
  
    val yearlyRevenue = 10_000_000  
  
    class Nested {  
  
        val company = Company()          // Must create insta  
        val revenue = company.yearlyRevenue  
  
    }  
  
    inner class Inner {              // Inner class due  
        val revenue = yearlyRevenue // Has access to outer  
    }  
}
```

In normal nested classes, you must first get a reference to the containing class to access any of its members. Inner classes allow you to access these directly. Under the hood, the

compiler simply generates the same code as shown in the nested class.

## PRIMARY AND SECONDARY CONSTRUCTORS

Kotlin differentiates between primary and secondary constructors. When you have multiple constructors in other languages, you often think of one of them as being the primary one, but Kotlin makes this explicit and provides a pragmatic way to declare properties directly in the primary constructor.

### Primary Constructors

You already briefly saw a primary constructor when you learned how to add properties to a class. Now, let's dive a little deeper. The primary constructor directly follows the class name (and possible modifiers) as demonstrated in Listing 4.19. Parameters are defined the same way as in normal methods.

But where do you place initialization logic that you'd normally write into the constructor? For this purpose, you use an **init** block in Kotlin, also shown in Listing 4.19. So, the primary constructor is split into two parts: the parameters in the class header, and the constructor logic in the **init** block. Be aware that normal constructor parameters (without **val** or **var**) can be accessed only inside this **init** block and in property initializers but not in class methods because they are not properties of the class.

#### **Listing 4.19 Using a Primary Constructor**

[Click here to view code image](#)

---

```
class Task(_title: String, _priority: Int) { // Defi
    val title = _title.capitalize() // Uses
    var priority: Int
    init {
```

```
    priority = Math.max(_priority, 0)          // Uses
}
}
```



Here, underscores are used to differentiate the constructor parameters from the properties and avoid name clashes. The `_title` parameter is used in an initializer and the `_priority` parameter is used in the `init` block to initialize the corresponding properties.

If the constructor has modifiers, the **constructor** keyword must be used and directly precedes the parameter list as in [Listing 4.20](#).

#### [Listing 4.20 Adding Modifiers to a Primary Constructor](#)

[Click here to view code image](#)

---

```
class Task private constructor(title: String, priorit
```



The explicit **constructor** keyword is required to set the constructor's visibility and to add annotations or other modifiers. A common reason is dependency injection via `@Inject constructor(...)` using a tool like Dagger.

Next, Kotlin provides a concise way to upgrade the parameters of a primary constructor to properties of the class by prefixing them with **val** or **var**, producing read-only and mutable properties, respectively (as shown in [Listing 4.21](#)).

#### [Listing 4.21 Upgrading Primary Constructor Parameters to Properties](#)

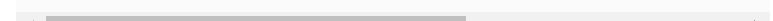
[Click here to view code image](#)

---

```
class Task(val title: String, var priority: Int) {

    init {

        require(priority >= 0)
    }
}
```



---

Notice that this code is not equivalent to [Listing 4.19](#): You can no longer capitalize the title because the `title` property is read-only and thus fixed to whatever value is passed into the constructor. Still, for simple properties, using `val` and `var` inside the primary constructor is the idiomatic—and most concise—way to introduce properties.

## Secondary Constructors

Classes may have any number of secondary constructors, including zero. But how do they differ from primary constructors? Think of the primary constructor as the main interface for object creation. In Kotlin, all secondary constructors must delegate to the primary constructor if one exists. Then, you can be sure that the primary constructor is executed on every object creation.

Secondary constructors provide alternative interfaces for object creation that transform the input data and delegate to the primary constructor, as in [Listing 4.22](#).

**Listing 4.22 Combining Primary and Secondary Constructors**

[Click here to view code image](#)

---

```
class Task(val title: String, var priority: Int) {  
  
    constructor(person: Person) : this("Meet with ${per  
        println("Created task to meet ${person.name}")  
  
    }  
  
}
```

---

Delegation to the primary constructor is achieved by extending the secondary constructor with a colon followed by `this(...)` to call the primary constructor. Secondary constructors can have a constructor body; only the primary constructor uses the `init` block.

In case you don't consider one of the constructors to be the primary one, alternately, you can use only secondary

constructors. However, note that you can't use the shorthand to create class properties that way.

Tip

Do not use secondary constructors to implement optional constructor parameters as in the so-called telescoping (anti-)pattern:

[Click here to view code image](#)

```
class Food(calories: Int, healthy: Boolean) {  
  
    constructor(calories: Int) : this(calories, false)  
  
    constructor(healthy: Boolean) : this(0, healthy)  
  
    constructor() : this(0, false)  
  
}
```

This is obsolete thanks to default parameter values that allow you to concisely implement optional parameters:

[Click here to view code image](#)

```
class Food(calories: Int = 0, healthy: Boolean = false)
```

## INHERITANCE AND OVERRIDING RULES

Inheritance is one of the cornerstones of OO. At its core, OO is about abstraction and programming the differences between classes. This is enabled by inheriting shared logic from a parent class that represents an abstraction of its children. Overriding rules specify how inheritance works, what can be inherited, and which logic can be overridden.

In order to follow along this section, there are three terms that you should know. All of these may apply to classes, properties, and methods.

- Being *abstract* means that the class or member is not fully implemented —the remaining implementation is left for child classes to provide.
- Being *open* means that the class or member is fully implemented and therefore ready to be instantiated (class) or accessed (property or method), but it allows child classes to override the implementation to adjust to their needs.
- Being *closed* means that the class or member is fully implemented and ready to be used, and doesn't allow child classes to override its

implementation. For a class, this means it cannot have subclasses. For members, it means they cannot be overridden in subclasses.

Kotlin follows the *closed-by-default* principle. This means that classes and their members are by default closed unless you explicitly make them abstract or open.

To understand inheritance, this section explores common entities involved in inheritance, namely interfaces, abstract classes, and open classes in Kotlin, and how you can use them for inheritance.

## Interfaces

Interfaces form the highest level of abstraction in your code. They define capabilities that users can rely on without restricting how implementing classes realize that capability. Typically, they only define a few abstract methods to indicate the capabilities, as done in [Listing 4.23](#).

### [Listing 4.23 Defining an Interface](#)

[Click here to view code image](#)

```
interface Searchable {  
  
    fun search() // All implementing classes have this  
  
}
```

In Kotlin, interfaces can have default implementations. So although interface members are typically abstract as in [Listing 4.23](#), they may have a default implementation. [Listing 4.24](#) shows properties and methods both with and without default implementations. Although interfaces can contain properties, they cannot have state. Thus, properties are not allowed to have a backing field. That's why all properties are either abstract or have accessors that use no backing field, such as the getter in [Listing 4.24](#). Because non-abstract properties must not have a backing field, they must be read-only.

### [Listing 4.24 Defining an Interface with Default Implementations](#)

[Click here to view code image](#)

```
interface Archivable {  
  
    var archiveWithTimeStamp: Boolean // Abstract prop  
  
    val maxArchiveSize: Long // Property with  
    get() = -1 // Default implementation  
  
  
    fun archive() // Abstract method  
  
    fun print() { // Open method via interface  
  
        val withOrWithout = if (archiveWithTimeStamp) "with" else "without"  
  
        val max = if (maxArchiveSize == -1L) "∞" else "$max"  
  
        println("Archiving up to $max entries $withOrWithout")  
  
    }  
  
}
```

The **Archivable** interface defines which capabilities any of its implementing classes must at least provide, namely archiving and printing. In an interface, any method without body is implicitly abstract. The same goes for properties without accessors. All other methods and properties are implicitly open, meaning they can be overridden as well. Thus, all members of interfaces can be overridden in implementing classes.

**Java Note**

Java 8 introduced interfaces with default method implementations as well.<sup>2</sup>

<sup>2</sup>. <https://docs.oracle.com/javase/tutorial/java/landI/defaultmethods.html>

A class can implement any number of interfaces using the syntax shown in [Listing 4.25](#).

**Listing 4.25 Inheriting from Interfaces**

[Click here to view code image](#)

```
class Task : Archivable, Serializable { // Implement
```

```
override var archiveWithTimeStamp = true

override fun archive() { ... }

}
```

Here, the class `Task` implements the two interfaces `Archivable` and `Serializable`. By convention, the colon is surrounded by spaces on both sides here, in contrast to when it's used to define variable types or return types. The `override` modifier is required to override a property or method from a superclass. This way, overriding is always explicit and cannot happen accidentally.

You should use default implementations in interfaces judiciously because they blur the line between interfaces and abstract classes.

## Abstract Classes

Abstract classes are one abstraction level below interfaces. Like interfaces, they are used to define abstract methods and properties but typically already contain more concrete implementations and can carry state. Their purpose is to encapsulate the similarities of their child classes. They cannot be instantiated themselves because they don't yet represent a usable entity of the application—some members may still be abstract and therefore unusable. Abstract classes are introduced using the `abstract` modifier as shown in Listing 4.26.

**Listing 4.26 Abstract Classes and Overriding**

[Click here to view code image](#)

---

```
abstract class Issue(var priority: Int) {

    abstract fun complete() // Abstract

    open fun trivial() { priority = 15 } // Open method

    fun escalate() { priority = 100 } // Closed method

}
```

```
class Task(val title: String, priority: Int) : Issue(  
    // ...  
  
    override fun complete() { println("Completed task: ")  
  
    override fun trivial() { priority = 20 }  
  
    // Cannot override 'escalate' because it is closed  
}
```

Here, `Issue` represents entities with a priority that can be completed, such as tasks, meetings, or daily routines. Inheriting from an abstract class works similarly to implementing an interface, but the class name must be followed by parentheses to call its constructor. Here, this is done via `Issue(priority)` in the class header. Any parameters must be passed into the constructor of the parent class, here `priority`.

You can access properties and methods of a parent class or interface simply via their name. If their name clashes with a member of the subclass, you can access the members of the parent class by qualifying them with `super`. For instance, `Task` can access `super.priority` (the property of the `Issue` class) and `super.escalate()`.

Overriding members of parent classes works the same way as overriding members of interfaces. Kotlin's closed-by-default principle comes into play here because, in contrast to interfaces, methods in abstract classes are closed by default. To allow children to override them, you must use the `open` modifier in Kotlin. To make them abstract, you must use `abstract`. Both options are shown in Listing 4.26.

**Tip**

Think of abstract versus open as "must override" versus "can override." A non-abstract class *must* override all abstract members (properties or methods) it inherits, and *is allowed to* override any open members. Closed members cannot be overridden.

Note that you can also override properties in primary constructors using **class Task(..., override val priority: Int)** but there is no need for it here.

## Open Classes

The closed-by-default principle doesn't only apply to properties and methods in Kotlin but also to classes: All classes in Kotlin are closed by default, meaning they cannot be inherited from. This language design choice prompts developers to "explicitly design for inheritance or else prohibit it" (see *Effective Java* by Joshua Bloch<sup>3</sup>). When classes are open by default, they often allow inheritance without being designed for it, which can lead to fragile code.

3. <https://www.pearson.com/us/higher-education/program/Bloch-Effective-Java-3rd-Edition/PGM1763855.html>

### Java Note

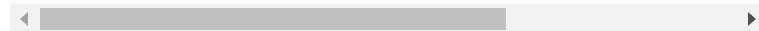
Being closed by default, normal classes in Kotlin correspond to final classes in Java and open classes in Kotlin correspond to normal Java classes.

[Listing 4.27](#) demonstrates how to declare open classes. It also illustrates that the child of an open class is again closed by default, so you'll have to explicitly declare it to be open again to allow further inheritance.

### Listing 4.27 Open Classes and Closed-by-Default Classes

[Click here to view code image](#)

```
class Closed // Closed class:  
  
class ChildOfClosed : Closed() // NOT allowed: c  
  
  
open class Open // Open class: o  
  
class ChildOfOpen : Open() // Allowed  
  
class ChildOfChild : ChildOfOpen() // NOT allowed: c
```



Classes without an explicit superclass inherit from `Any`, the superclass of all non-nullables types in Kotlin. It only defines `hashCode`, `equals`, and `toString`. The reason is that `Any` is platform-agnostic and can thus also be used when targeting Kotlin/JS or Kotlin/Native. Other methods specifically for the JVM are attached via extension functions.

**Java Note**

Kotlin's `Any` type is similar to `Object` in Java except that it is non-nullable and, as mentioned, it only defines three methods as members. `Any` gets mapped to `Object` in the bytecode.

## Overriding Rules

When working with inheritance, there are several rules to consider. Fortunately, they're logical when you think about them.

First, due to default methods in interfaces, you may run into the situation where a class inherits conflicting implementations for a method or property from its superclass and implemented interfaces. In that case, Kotlin forces you to override that member. Inside the override, you can call a specific parent implementation using the syntax  
`super<MyClass>.foo()` or  
`super<MyInterface>.foo()` to differentiate between them.

Second, you're allowed to override read-only properties with mutable properties but not vice versa. That's because a read-only property only implies a getter. So overriding it with a mutable property is possible because an additional setter can be added. However, a mutable property in a parent class comes with a setter, which cannot be undone in a child class.

## TYPE CHECKING AND CASTING

Type checks and casts in Kotlin are conceptually the same as in other statically typed languages. But in Kotlin, you have to think about nullability again in order to safely cast objects. For this section, imagine you're using the *Composite pattern*<sup>4</sup> and you implemented the classes **Component**, **Composite**, and **Leaf**, where **Component** is the abstract parent class of the other two classes.

4. [https://sourcemaking.com/design\\_patterns/composite](https://sourcemaking.com/design_patterns/composite)

### Type Checking

To check whether an object has a specific type, Kotlin provides the **is** operator. Listing 4.28 shows how to use it for type checks.

**Listing 4.28 Type Checks Using is**

[Click here to view code image](#)

```
val item: Component = Leaf()    // 'item' is a Leaf at

if (item is Composite) { ... }    // Checks if 'item' is

if (item !is Composite) { ... }  // Checks if 'item' is
```

The **is** operator is negated as **!is** to check whether an object is *not* of a specific type. Unless the class structure changes, the second type check is equivalent to **item is Leaf** in this example because that is the only other alternative.

## Type Casting

In statically typed languages like Kotlin, type casting allows you to map (“cast”) the type of an object into another one, if the types are compatible. In Kotlin, there are several ways to cast an object because, as always, Kotlin makes you think about nullability and how to handle it. Consequently, you can cast safely using `as?` or you can cast without considering the `null` case using `as`.

Both are shown in [Listing 4.29](#).

**Listing 4.29** Type Casts for Nullable Types

[Click here to view code image](#)

```
val item: Component? = null

val leaf: Leaf      = item as Leaf           //
val leafOrNull: Leaf? = item as Leaf?        //
val leafSafe: Leaf?  = item as? Leaf          //
val leafNonNull: Leaf = (item as? Leaf) ?: Leaf() //
```

This listing shows four ways to cast `item` to a `Leaf` object. The first shows that trying to cast a nullable object to a non-null type unsafely (using `as`) results in a `TypeCastException` if the object is `null`. To avoid this, the second example explicitly casts to the nullable type `Leaf?`, which changes the type of the expression accordingly. The third cast uses the *safe cast operator* `as?` that returns `null` instead of throwing an exception if the cast fails. To avoid nullability right at the declaration site of the variable, the fourth cast uses the elvis operator to provide a default value for the `null` case.

In the above example, casting to a nullable type and using the safe cast operator are equivalent. This is no longer the case when trying to cast to a wrong class, as demonstrated in [Listing 4.30](#).

**Listing 4.30** Type Casts and `ClassCastException`

[Click here to view code image](#)

```
val composite: Component = Composite()

val leafOrNull: Leaf? = composite as Leaf?           // C]

val leafSafe: Leaf?     = composite as? Leaf           // E\
```

Here, the original object is not even nullable, so casting to a `Leaf?` does not make the cast any safer, and the first cast results in a `ClassCastException`. The difference in the safe cast operator `as?` is that it never throws an exception on failure but returns `null` instead.

## Smart Casts

With smart casts, the Kotlin compiler helps you avoid redundant casts by doing them for you. Kotlin does this whenever the compiler can infer stricter type constraints, including smart-casting from nullable to non-nullable types. This is demonstrated in [Listing 4.31](#). For this, assume that `Component` defines the function `component()`, `Composite` has an additional function `composite()`, and `Leaf` has a function `leaf()`.

[Listing 4.31 Smart Casts](#)

[Click here to view code image](#)

```
val comp: Component? = Leaf() // Type is nullable 'C

if (comp != null) { comp.component() }

if (comp is Leaf) { comp.leaf() }

when (comp) {

    is Composite -> comp.composite()

    is Leaf -> comp.leaf()

}
```

```
if (comp is Composite && comp.composite() == 16) {}

if (comp !is Leaf || comp.leaf() == 43) {} // Smart-cast

if (comp !is Composite) return

comp.composite() // Smart-cast to Composite (because
```

The Kotlin compiler can infer stricter type constraints in many situations, helping you focus on your actual logic. This includes casting to non-nullable types, as in the first check above. It also includes casts to subtypes, which is enabled using the **is** operator in **if** and **when** expressions. Casting inside the conditions is possible because Kotlin evaluates non-atomic conditions lazily. This means that in conjunctions (composed with `&&`), the right-hand side will *not* be evaluated if the left-hand side is already **false**. Similar logic applies to disjunctions (composed with `||`). Lastly, the compiler is intelligent enough to infer when a previous check would have led to a **return**, as in the last example. If `comp` were not of type `Composite`, the last line wouldn't be reached, thus it can be smart-cast there as well.

**Note**

Smart casts can only be applied if the variable cannot change between the type check (or null check) and its usage. This is another reason to prefer `val` over `var` and to apply the principles of information hiding in object-oriented code to limit variable manipulations from the outside.

You can recognize smart-casts in IntelliJ and Android Studio 3 by their green highlighting.

## VISIBILITIES

A crucial principle in OO is information hiding, meaning the internal implementation details of a class shouldn't be visible to the outside. Instead, only a well-defined interface is exposed that makes it more predictable in which ways the class is used and mutated. Visibilities are the facilitators of information hiding. They allow you to define what's accessible from where.

## Declarations in Classes or Interfaces

First, let's consider class or interface members (properties and methods). For these, there are four visibility modifiers (three of which work the same way as in Java). In roughly ascending order of restrictiveness, they are

- **public**: The member is accessible wherever its containing class is visible. This is used for the members that form the well-defined interface of the class or interface.
- **internal**: The member is accessible anywhere inside the same module, given that the containing class is visible. This is used for parts of the well-defined interface that are only relevant for clients from the same module.
- **protected**: The member is visible inside the class itself and its child classes. This is useful for members that shouldn't be accessible to the outside but are useful to implement child classes. Note that this only makes sense for abstract and open classes.
- **private**: The member is accessible only inside its containing class. This is used heavily to hide class internals from the outside.

### Java Note

Note that there is no package-private visibility as in Java. The default visibility in Kotlin is **public**, and the closest thing to package-private is **internal**.

The **internal** visibility refers to *modules*. So what's meant by module here? A module is a set of Kotlin files that are compiled together, for instance an Android Studio module, a Maven module, a Gradle source set, or a set of files compiled in one Ant task.

[Listing 4.32](#) demonstrates the effect of all visibility modifiers.

### Listing 4.32 Visibility Modifiers

[Click here to view code image](#)

```
open class Parent {  
  
    val a = "public"  
  
    internal val b = "internal"  
  
    protected val c = "protected"  
  
    private val d = "private"
```

```

inner class Inner {

    val accessible = "$a, $b, $c, $d" // All accessible

}

}

class Child : Parent() {

    val accessible = "$a, $b, $c" // d not accessible

}

class Unrelated {

    val p = Parent()

    val accessible = "${p.a}, ${p.b}" // p.c, p.d not accessible

}

```

Inner classes can access even private members, whereas child classes (from the same module) can access all but private members. Unrelated classes can access public members, plus internal ones if the unrelated class is in the same module.

To add visibility modifiers to constructors, getters, and setters, you must add the corresponding soft keywords (**constructor**, **get**, **set**) explicitly, as shown in [Listing 4.33](#).

#### [Listing 4.33 Setting the Visibility of Constructors, Getters, and Setters](#)

[Click here to view code image](#)

```

open class Cache private constructor() {

    val INSTANCE = Cache()

    protected var size: Long = 4096 // Getter inherits visibility

    private set // Setter can have different visibility

```

```
}
```

To attach a nondefault visibility to the primary constructor, the **constructor** keyword is required. Secondary constructors can simply be prefixed with visibility modifiers as they are. The visibility of getters is always defined by the visibility of the property itself, thus there's no additional line saying **protected get**. By default, the setter inherits the property's visibility as well, but this can be changed by adding the **set** keyword together with the desired visibility. Because there's no custom setter implementation here, writing **private set** suffices. Note that you cannot use primary constructor properties if you want to set an explicit setter visibility.

**Note**

One more thing to consider with visibilities is that inline methods cannot access class properties with a more restrictive visibility. For instance, a public inlined method cannot access an internal property because that property may not be accessible at the inlined position.

## Top-Level Declarations

Kotlin allows top-level (or file-level) declarations of properties, functions, types, and objects. So how can you restrict the visibilities of these?

For top-level declarations, only three of the visibility modifiers are allowed because **protected** doesn't make sense here.

Other than that, the visibilities work conceptually the same way, except there's no containing class that further restricts the visibility.

- **public**: The declaration is accessible everywhere.
- **internal**: The declaration is accessible everywhere inside the same module.
- **private**: The declaration is accessible only within its containing *file*.

# DATA CLASSES

With data classes, Kotlin provides a convenient way to implement classes with the main purpose of holding data. Typical operations on data objects are reading, altering, comparing, and copying the data. All of these are particularly easy with Kotlin's data classes.

## Using Data Classes

Declaring a data class is as simple as prefixing the class declaration with the **data** modifier as in Listing 4.34.

Listing 4.34 Declaring a Data Class

[Click here to view code image](#)

```
data class Contact(val name: String, val phone: Stri
```

This way, the compiler automatically generates several useful methods that can be derived from just the data, namely **hashCode** and **equals**, **toString**, a **copy** method, and **componentN** methods (**component1**, **component2**, ...), one for each property of the data class. Listing 4.35 shows how these can be used.

Listing 4.35 Generated Members of Data Classes

[Click here to view code image](#)

```
val john = Contact("John", "202-555-0123", true)

val john2 = Contact("John", "202-555-0123", true)

val jack = Contact("Jack", "202-555-0789", false)

// toString

println(jack) // Contact(name=Jack, pho
```

```
// equals

println(john == john2) // true

println(john == jack) // false
```

```
// hashCode

val contacts = hashSetOf(john, jack, john2)

println(contacts.size)      // 2 (no duplicates in set)

// componentN

val (name, phone, _) = john      // Uses destructuring

println("$name's number is $phone") // John's number

// copy

val johnsSister = john.copy(name = "Joanne")

println(johnsSister) // Contact(name=Joanne, phone=234-5678)
```

The compiler-generated implementations of `hashCode`, `equals`, and `toString` work as you'd expect. Without the `data` modifier, `toString` would print the memory address, `john` would be unequal `john2`, and `contacts.size` would be 3. Remember that the double-equal operator in Kotlin calls `equals` to check for structural equality.

Up to this point, data classes essentially remove the burden of generating those implementations in your IDE. But that only takes a few seconds, so the main benefit here is not the time you save but *readability* and *consistency*. Can you quickly tell whether that IDE-generated `equals` method contains a manual change? Are `equals` and `hashCode` still up to date, or did someone forget to regenerate them after changing a property? With data classes, you don't have these problems.

Additionally, the `componentN` functions enable so-called destructuring declarations as shown in [Listing 4.35](#). These allow you to easily extract all values stored inside an object of

a data class into separate variables. Note that you can use an underscore to ignore values you don't need.

Lastly, the `copy` method allows you to copy a data object while modifying any of its properties, using named parameters. So you could just as well change the phone number or the favorite indicator without changing any of the other properties.

**Note**

Inside the data class, you can still declare custom implementations for `hashCode`, `equals`, and `toString`. If you do, the compiler won't generate the corresponding method.

However, `componentN` and `copy` methods with signatures that conflict with the generated ones cannot be defined inside the data class. This ensures they work as expected.

When working with data classes, there are several limitations you should be aware of.

- Data classes cannot be inherited from. In other words, they cannot be **open** or **abstract**.
- They cannot be **sealed** or **inner** classes (sealed classes are discussed later).
- The primary constructor needs at least one parameter in order to generate the methods.
- All parameters in the *primary* constructor must be upgraded to properties using **val** or **var**. You may also add secondary constructors to data classes.

## Inheritance with Data Classes

Data classes cannot be inherited from. However, they may have a superclass and implement interfaces. In case you make use of this, there are several things to consider.

- If the parent class contains *open* implementations of `hashCode`, `equals`, or `toString`, the compiler will generate implementations that override those.
- If the parent class contains *final* implementations of `hashCode`, `equals`, or `toString`, the implementations from the parent class are used.
- The parent class may contain methods called `componentN` as long they're *open* and their *return types match* the data class. Here, the compiler will again generate overriding implementations for the data class. Parent implementations that are **final** or don't have a matching signature will cause an error at compile time.

- Similarly, the parent class must not have an implementation of `copy` that matches the signature it would have in the data class.
- The latter two rules make sure that destructuring declarations and copying always work as expected.

Lastly, you cannot inherit between data classes. The reason for this is there's no way to generate the required methods in a consistent and correct way for such hierarchies. I won't show a listing for all this because I'd recommend you avoid using general inheritance with data classes. A legitimate special case, which covers sealed classes, is shown in the next section.

**Tip**

Data classes are useful to define containers for multiple values whenever a function should conceptually return multiple values. If a function for instance returns a name and a password, you can encapsulate that as follows:

[Click here to view code image](#)

```
data class Credentials(val name: String, val password: String)
```

Then you can use `Credentials` as the return type.

Kotlin also has the predefined data classes `Pair` and `Triple` to carry two or three values, respectively. However, those are less expressive than a dedicated data class.

## ENUMERATIONS

For enumerations, Kotlin provides enum classes. Listing 4.36 shows the simplest form of using an enum class. They're useful to model a finite set of distinct objects, such as states of a system, directions, or a set of colors.

[Listing 4.36 Declaring a Simple Enum Class](#)

[Click here to view code image](#)

```
enum class PaymentStatus {
    OPEN, PAID
}
```

The `enum` modifier transforms a class declaration into an enumeration class, similar to `inner` and `data` class modifiers. Enum constants are separated by commas and are

objects at runtime, meaning there can only ever be one instance of each.

Although this already covers the most frequent use cases of enums, you can also attach constructor parameters, properties, and methods. This is demonstrated in [Listing 4.37](#).

**Listing 4.37 Declaring an Enum Class with Members**

[Click here to view code image](#)

```
enum class PaymentStatus(val billable: Boolean) { //  
  
    OPEN(true) {  
        override fun calculate() { ... }  
    },  
    PAID(false) {  
        override fun calculate() { ... }  
    }; // Note the semicolon: it separates the enum from the class body  
  
    fun print() { println("Payment is ${this.name}") }  
    abstract fun calculate()  
}
```

What's important to note here is that the enum instances must be defined first, followed by a semicolon to separate any members that follow. You can add any properties or methods to the class. Abstract methods must be overridden in each enum instance. [Listing 4.38](#) shows how to use your own as well as compiler-generated members of enums.

**Listing 4.38 Working with Enums**

[Click here to view code image](#)

```
val status = PaymentStatus.PAID  
  
status.print() // Prints "Payment is PAID"
```

```
status.calculate()

// Kotlin generates 'name' and 'ordinal' properties
println(status.name)      // PAID
println(status.ordinal)   // 1
println(status.billable)  // false

// Enum instances implement Comparable (order = order
println(PaymentStatus.PAID > PaymentStatus.OPEN)  //

// 'values' gets all possible enum values
var values = PaymentStatus.values()
println(values.joinToString()) // OPEN, PAID

// 'valueOf' retrieves an enum instance by name
println(PaymentStatus.valueOf("OPEN")) // OPEN
```

For each enum instance, Kotlin generates a `name` property that holds the instance name as a string. The `ordinal` property is simply the zero-indexed order in which enum instances are declared inside the enum class. All enum instances implement `Comparable` by comparing the ordinal values.

Custom properties and members can be accessed as usual. Additionally, Kotlin generates the method `valueOf` to get an enum instance by name, and `values` to get all possible instances. There are also generic variants of these two called `enumValueOf` and `enumValues`. If there's no enum instance with the given name, an `IllegalArgumentException` is thrown.

Since enums have a fixed set of possible instances, the compiler can infer if a **when** expression that uses the enum is exhaustive, as shown in Listing 4.39.

Listing 4.39 Exhaustive when Expression with Enum

[Click here to view code image](#)

```
val status = PaymentStatus.OPEN

val message = when(status) {
    PaymentStatus.PAID -> "Thanks for your payment!"
    PaymentStatus.OPEN -> "Please pay your bill so we can
} // No else branch necessary
```

## SEALED CLASSES

Sealed classes are used to build restricted class hierarchies, in the sense that all direct subtypes of the sealed class must be defined inside the same file as the sealed class itself. Listing 4.40 shows a sealed class that represents binary trees. It must be declared in a `.kt` file, not a script.

Listing 4.40 Declaring a Sealed Class with Subtypes

[Click here to view code image](#)

```
sealed class BinaryTree // Sealed class with two direct subtypes

data class Leaf(val value: Int) : BinaryTree()

data class Branch(val left: BinaryTree, val right: BinaryTree)

// Creates a binary tree object

val tree: BinaryTree = Branch(Branch(Leaf(1), Branch(Leaf(2), Leaf(3))), Branch(Leaf(4), Leaf(5)))
```

To declare a sealed class, simply prefix the class declaration with the **sealed** modifier. Remember that the class body

with curly braces is optional if it's empty. The sealed class itself is implicitly abstract (thus cannot be instantiated) and its constructor is private (thus the class cannot be inherited from in another file). This already implies that all direct subtypes must be declared inside the same file. However, subtypes of subtypes may be declared anywhere. You could add abstract or concrete members to the sealed class, just like in other abstract classes. Data classes are useful to define the subtypes of sealed classes, as shown in [Listing 4.40](#).

**Note**

You can think of sealed classes as a generalization of enum classes. They have the additional possibility to instantiate as many objects of each subtype as you want. In contrast, enums always have single instances such as PAID and OPEN.

The benefit of sealed classes is that you have more control over your class hierarchy, and the compiler uses this as well. Because all direct child classes are known at compile time, the compiler can check **when** expressions for exhaustiveness. Hence, you can omit the **else** branch whenever you cover all cases, as with enums in [Listing 4.39](#). This is demonstrated in [Listing 4.41](#) with the example of a sealed class representing expressions.

[Listing 4.41 Exhaustive when Expression with Sealed Class](#)

[Click here to view code image](#)

```
sealed class Expression // Sealed class representing expressions

data class Const (val value: Int) : Expression()

data class Plus (val left: Expression, val right: Expression) : Expression()

data class Minus (val left: Expression, val right: Expression) : Expression()

data class Times (val left: Expression, val right: Expression) : Expression()

data class Divide(val left: Expression, val right: Expression) : Expression()

fun evaluate(expr: Expression): Double = when(expr) {

    is Const -> expr.value.toDouble()
```

```
    is Plus  -> evaluate(expr.left) + evaluate(expr.right)
    is Minus -> evaluate(expr.left) - evaluate(expr.right)
    is Times -> evaluate(expr.left) * evaluate(expr.right)
    is Divide -> evaluate(expr.left) / evaluate(expr.right)
}
} // No else branch required: all possible cases are covered
```

```
val formula: Expression = Times(Plus(Const(2), Const(3)), Times(Const(4), Const(5)))
println(evaluate(formula)) // 42.0
```

Here, the value of the **when** expression is used as the return value of **evaluate**. Since all cases are covered, no **else** branch is required and there's still always a well-defined return value. Note that subtypes of subtypes don't interfere with the exhaustiveness of the type check because all sub-subtypes are subsumed in their parent subtype. Also remember that Kotlin smart-casts **expr** on the right-hand sides of the **when** expression, making either **value** or **left** and **right** accessible.

Regarding control over the hierarchy, sealed classes implement a strict form of the closedness principle. Without this, an interface should be frozen once it's public because there could be any number of child classes in different projects. Changing or removing a method would crash all of them. With sealed classes, new methods can be added easily because there can be no new direct child classes. On the downside, adding subtypes to sealed classes involves effort for adjusting all **when** expressions that use no **else** branch. You have to balance between how likely subtypes are to change versus how likely their operations are to change.

### On Algebraic Data Types

For the mathematically inclined reader, I want to mention that data classes and sealed classes are Kotlin's way to implement *algebraic data types*, more specifically, sum and product types.

These concepts originate from functional programming and type theory. You can think of product types as records or tuples of multiple elements (or a struct in C/C++). These can be implemented using data classes:

[Click here to view code image](#)

```
interface A { val a: Int }

interface B { fun b() }

data class Both(override val a: Int) : A, B { override fun b() { ... } }
```

To instantiate an object of such a product type, you need values for each of its constituent types. Here, B doesn't have properties so only a is required. If there are x possible values for type A and y possible values of type B, their product type has  $x \cdot y$  possible values—thus the name *algebraic*, and in this case *product*.

Sum types, on the other hand, are used to construct types that can contain *either one* of their constituent types. This is what is expressed with sealed classes; a `BinaryTree` is either a `Leaf` or a `Branch`. The more general sum type of two types A and B is often aptly named `Either`:

[Click here to view code image](#)

```
sealed class Either

class MyA(override val a: Int) : Either(), A

class MyB : Either(), B { override fun b() { ... } }
```

Here, any object of type `Either` either conforms to the interface A or B, but not both.

This is why sealed classes and data classes often work well together, producing a sum type consisting of product types, such as the `BinaryTree`.

## OBJECTS AND COMPANIONS

Most of the time, you create objects as instances of classes. In Kotlin, they can be declared directly as well. The first way to do this is to use *object expressions*, which allow you to create an object on the fly and use it as an expression. Listing 4.42 shows the simplest case of an ad-hoc object that holds some data.

**Listing 4.42 Simple Object Expressions**

[Click here to view code image](#)

```
fun areaOfEllipse(vertex: Double, covertex: Double):
```

```
    val ellipse = object { // Ad-hoc object
        val x = vertex
        val y = covertex
    }
    return Math.PI * ellipse.x * ellipse.y
}
```



Kotlin uses the **object** keyword to create objects. For simple objects, such as Listing 4.42, the keyword is directly followed by the object body. However, object expressions are more powerful and allow creating an object of a class while overriding certain members but without defining a whole new subclass. For this, objects can inherit from a class and any number of interfaces, just like classes. Shown in Listing 4.43, listeners are a common use case.

**Java Note**

Object expressions as shown in Listing 4.43 supersede Java's anonymous inner classes.

As a side note, object expressions can access nonfinal variables from their enclosing scope, in contrast to anonymous inner classes in Java.

#### Listing 4.43 Object Expressions with Supertype

[Click here to view code image](#)

```
scrollView.setOnDragListener(object : View.OnDragList
    override fun onDrag(view: View, event: DragEvent)
        Log.d(TAG, "Dragging...")
        return true
    }
}
```



To create an object with supertypes, the **object** keyword is now followed by a colon and its supertypes, just as you do for classes. Note that **OnDragListener** is an interface,

therefore it is not followed by parentheses because there's no constructor to call.

**Tip**

Java interfaces with just a single abstract method (called *SAM interfaces*) such as `OnDragListener` are more easily constructed using a lambda expression:

[Click here to view code image](#)

```
scrollView.setOnDragListener { view, event -> ... }
```

Objects can be considered a generalization of this and can be used even when you want to implement multiple interfaces or methods, not just one.

Object expressions are only usable as types within local and private declarations. If you return an object expression from a public method, its return type is `Any` because callers cannot see the object expression. You can mitigate this problem by using an interface that makes the object's structure available to the outside. [Listing 4.44](#) demonstrates this behavior.

**Listing 4.44 Object Expressions as Return Type**

[Click here to view code image](#)

```
class ReturnsObjectExpressions {  
  
    fun prop() = object {  
        // Returns an object expression  
        // with a single property.  
        //  
        // This is not accessible from outside the class.  
        val prop = "Not accessible"  
    }  
  
    fun propWithInterface() = object : HasProp { // Function type  
        // Returns an object expression  
        // that implements an interface.  
        //  
        // This is accessible from outside the class.  
        override val prop = "Accessible"  
    }  
  
    fun access() {  
        prop().prop // Compile-time error (Type mismatch)  
        propWithInterface().prop // Now possible (HasProp)  
    }  
}
```

```
}
```

```
interface HasProp { // Allows exposing the 'prop' to the outside
```

```
    val prop: String
```

```
}
```

Basically, as long as you're using object expressions locally, Kotlin makes sure you can access the object's members. If you return an object expression from a nonprivate method, make sure to implement an interface that can be used as the return type. Otherwise, the return value is hardly useful.

### Object Declarations as Singletons

Used judiciously, singletons are a useful pattern for software development. If you think about it, singletons are just objects—meaning there's only one instance at runtime. Thus, you use the **object** keyword to create singletons in Kotlin, as in Listing 4.45.

Listing 4.45 Object Declaration for a Registry Singleton

[Click here to view code image](#)

```
import javafx.scene.Scene
```

```
object SceneRegistry { // Declares an object: this is no longer anonymous
```

```
    lateinit private var homeScene: Scene
```

```
    lateinit private var settingsScene: Scene
```

```
    fun buildHomeScene() { ... }
```

```
    fun buildSettingsScene() { ... }
```

```
}
```

Unlike object *expressions*, this object is no longer anonymous because the **object** keyword is followed by the object's

name. This is called an *object declaration*. It's not an expression and cannot be used on the right-hand side of an assignment. Members of an object are accessed by simply qualifying them with the object name as in Listing 4.46.

#### **Listing 4.46 Accessing Object Members**

[Click here to view code image](#)

```
val home = SceneRegistry.buildHomeScene()  
  
val settings = SceneRegistry.buildSettingsScene()
```

Object declarations can use inheritance as usual. Also, they can be nested into (non-inner) classes or other object declarations, but they cannot be local (declared inside a function). In contrast, object expressions can be used everywhere.

**Note**

Object declarations are initialized lazily, when they're first accessed. Hence, if the object is never used at runtime, it will use zero resources. This is a common way to implement singletons manually as well.

In contrast to this, object expressions are evaluated and initialized eagerly right where they're declared. This makes sense because their value is typically used directly.

## **Companion Objects**

Object declarations inside classes can be made *companion objects*. Members of such companion objects are then accessible by only prefixing their containing class, and without creating an object of that class. Listing 4.47 demonstrates this.

#### **Listing 4.47 A Factory as Companion Object**

[Click here to view code image](#)

```
data class Car(val model: String, val maxSpeed: Int)  
  
companion object Factory {  
  
    fun defaultCar() = Car("SuperCar XY", 360)  
  
}
```

```
val car = Car.defaultCar() // Calls companion method
```

Declaring a companion object is as simple as prepending the object declaration with the **companion** modifier. You can give it an explicit name, such as `Factory` in [Listing 4.47](#); otherwise it gets the default name `Companion`. All members of the companion object can be accessed directly on the `Car` class, or alternately by prefixing the companion name explicitly.

You may have wondered about a **static** modifier in Kotlin already—there is none. Instead of **static**, Kotlin uses companion objects and top-level declarations. While companion objects better resemble **static** declarations from other languages, top-level declarations may be the more idiomatic way to migrate certain static functions to Kotlin. Companions are useful for methods closely related to the containing class, such as factory methods. Note that, in contrast to static members in other languages, the companion members are not accessible on an *instance* of the containing class because this would be bad practice anyway.

Also, even though accessing companion members looks like accessing static members in other languages, companion objects really are objects at runtime. Thus, they can inherit a class and implement interfaces, as in [Listing 4.48](#).

#### [Listing 4.48 Companion Object with Factory Interface](#)

[Click here to view code image](#)

```
interface CarFactory {  
    fun defaultCar(): Car  
}  
  
data class Car(val model: String, val maxSpeed: Int)  
companion object Factory : CarFactory { // Companion object for Car
```

```
    override fun defaultCar() = Car("SuperCar XY", 360)  
}  
}  
◀ ▶
```

All supertypes are appended after a colon, and you now use the **override** modifier inside the companion. Regarding interoperability, the way `defaultCar` is declared in [Listing 4.48](#), you can only call this from Java as `Car.Factory.defaultCar()`. You can tell the compiler to compile this to actual static members in the Java bytecode using the `@JvmStatic` annotation, as in `@JvmStatic override fun defaultCar()`. That way, you can call it directly on the class from Java, using `Car.defaultCar()`. Interoperability issues and annotations are covered in detail in [Chapter 5, Interoperability with Java](#).

**Note**

You can define extension functions on companions as on any other objects, which can then be called from Kotlin in the same way as all other companion members:

[Click here to view code image](#)

```
fun Car.Factory.cheapCar() = Car("CheapCar 2000", 110)  
  
val cheap = Car.cheapCar()
```

# GENERICs

Generic programming is a powerful way to avoid code duplication. The idea is to lift type-specific algorithms and data structures to more generic ones. Like so often in software development, this is a process of abstraction. You focus on the fundamental properties required for the algorithm or data structure to work and abstract them from a specific type. When Musser and Stepanov introduced the concept in 1989, they described it as follows: “Generic programming centers around the idea of abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software.”<sup>5</sup>

5. David R. Messer and Alexander A. Stepanov, “Generic programming.” Presented at the First International Joint Conference of International Symposium on Symbolic and Algebraic Computation (ISSAC)-88 and Applied Algebra, Algebraic Algorithms, and Error Correcting Codes (AAECC)-6, Rome, Italy, July 4–8, 1988. *Lecture Notes in Computer Science*, P. Gianni. 358. Springer-Verlag, 1989, pp. 13–25. (<http://stepanovpapers.com/genprog.pdf>)

*Generics* as you know them from most object-oriented languages and as they’re discussed in this section are just one genericity mechanism to achieve this. Implementing algorithms in a parent class is also generic programming per the definition above. However, this section covers the specific genericity mechanism called generics.

## Generic Classes and Functions

As mentioned, the main idea is to generalize data structures and algorithms. For data structures, you use generic classes. These typically implement methods, which are then generic. Additionally, you can declare generic functions outside of generic classes to define generic algorithms. With this, the two main entities you want to generalize are covered. Prominent examples of generic data structures are collection types such as `List<E>` and `Map<K, V>`—where `E`, `K`, and `V` are called *generic type parameters* (or just *type parameters*).

**Note**

Regarding the term *type parameter*, think about what a parameter fundamentally is. As in mathematical functions, parameters introduce degrees of freedom. Imagine for instance a family of functions  $f(x; p) = p * x^2$  with parameter  $p$ . We call  $f$  parameterized, and can instantiate a family of similar but different functions from it by changing  $p$ . All share certain properties such as  $f(-x) = f(x)$ , but not others such as  $f(1) = 1$ .

The same applies to generic types like `Set<E>`, where the type parameter `E` introduces a degree of freedom—the type of the elements. The concrete type is not fixed when declaring the class but only when instantiating it. Still, all sets share certain properties.

## Generic Classes

To declare a generic class in Kotlin, you define all its generic type parameters in angle brackets directly after the class name, as shown in [Listing 4.49](#).

### [Listing 4.49 Declaring and Instantiating a Generic Class](#)

[Click here to view code image](#)

```
class Box<T>(elem: T) { // Generic class: can be a type parameter

    ...
    val element: T = elem

}

val box: Box<Int> = Box(17) // Type Box<Int> can also be a type parameter
println(box.element)      // 17
```

The declaration of the `Box` type uses one type parameter `T`, which is appended to the class name in angle brackets. This parameter can then be used just like any other type inside the class to define property types, return types, and so forth. At runtime, it will be the concrete type used when instantiating a `Box` object, here `Int`. Thus, the constructor accepts an `Int` and `element` has type `Int` as well. If you want more than one type parameter, comma-separate them inside the angle brackets.

**Tip**

You don't have to give the name `T` to the type parameter, but the convention is to use single uppercase letters that carry semantic meaning. Most frequent are the following:

`T` = type  
`E` = element  
`K` = key  
`V` = value  
`R` = return type  
`N` = number

In [Listing 4.49](#), notice how the concrete type of `Box` you want to have is only defined when instantiating a `Box`, not when declaring it. That's when you use your degree of freedom to use a specific `Box` variant. Without this, you'd have to repeat all code inside `Box` for every type you want to support, such as `IntBox` or `PersonBox`. You may think about just creating an `AnyBox`, but that stops being type-safe once you actually want to do something with it. The advantage of generics is that they are type-safe.

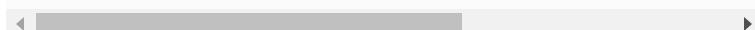
To get you accustomed to generics, [Listing 4.50](#) provides a more complex example that generalizes the `BinaryTree` data structure from the section on sealed classes. With this implementation, you can create binary trees that contain any kind of value in their leaves.

**Listing 4.50 Generalizing the `BinaryTree` Class**

[Click here to view code image](#)

---

```
sealed class BinaryTree<T> // Now generic: can carry  
  
data class Leaf<T>(val value: T) : BinaryTree<T>()  
  
data class Branch<T>(  
  
    val left: BinaryTree<T>,  
  
    val right: BinaryTree<T>  
  
) : BinaryTree<T>()  
  
  
val tree: BinaryTree<Double> = Leaf(3.1415) // Uses
```



### Type Aliases

A useful feature in combination with generic classes, but also function types, are *type aliases*. These allow you to provide a more meaningful name to existing types, for instance:

[Click here to view code image](#)

```
typealias Deck = Stack<Card>      // New name for generics-induced type

typealias Condition<T> = (T) -> Boolean // New name for function type
```

At use-site, Kotlin inlines the aliased type so that you can use the type alias as you would use the underlying type:

[Click here to view code image](#)

```
val is NonNegative: Condition<Int> = { it >= 0 } // Can assign a lambda
```

Note that Kotlin infers the type of the `it` variable based on the definition of the type alias.

## Generic Functions

Generic functions allow genericity on the algorithm side, even outside generic classes. They are defined and used as in [Listing 4.51](#).

### Listing 4.51 Declaring and Using a Generic Function

[Click here to view code image](#)

```
fun <T> myList0f(vararg elements: T) = Arrays.asList()

val list0: List<Int> = myList0f<Int>(1, 2, 3) // Alters the type of the variable

val list1: List<Int> = myList0f(1, 2, 3) // Infers the type of the variable

val list2 = myList0f(1, 2, 3)           // Infers the type of the variable

val list3 = myList0f<Number>(1, 2, 3)    // Infers the type of the variable

// Without parameters

val list4 = myList0f<Int>()           // Infers the type of the variable

val list5: List<Int> = myList0f()       // Infers the type of the variable
```

This function is similar to Kotlin’s built-in `listOf` and helps constructing lists. To make it a generic function, the `fun` keyword is followed by the generic type parameters in angle brackets. These can be used like any other type inside the function body. Like the built-in `listOf` function, you can call it with or without explicit type arguments—the compiler can infer the type based on the arguments. If you want to use a different type (such as `Number`) or don’t pass in any arguments, you have to either declare the variable type explicitly, as in `val list: List<Number>`, or declare the function type parameters explicitly, as in `myListOf<Number>`.

**Note**

We say that generics support *compile-time polymorphism*, specifically *parametric polymorphism*. This means that a generic function behaves uniformly for all concrete type arguments. This is in contrast to runtime polymorphism via inheritance and method overloading.

## Reification

In generic functions, you sometimes want to access the generic type parameter. A common example is to check whether an object is a subtype of the type parameter. However, type information for generic type parameters is not preserved at runtime so that you cannot check subtypes using `is` as you normally would. This is called *type erasure* and is a limitation of the JVM that forces you to use *reflection* in these cases. Reflection means introspecting (and potentially even modifying) your own code at runtime, for instance, examining the type of an object at runtime for which no static type information is available (due to type erasure). Listing 4.52 illustrates an example with a generic function that filters an iterable object by a given type.

**Listing 4.52 A Generic Function that Accesses Its Type Parameter**

[Click here to view code image](#)

---

```
fun <T> Iterable<*>.filterByType(clazz: Class<T>): Li
    @Suppress("UNCHECKED_CAST") // Must suppress unche
```

```
    return this.filter { clazz.isInstance(it) }.map { i

}

val elements = listOf(4, 5.6, "hello", 6, "hi", null,
    println(elements.filterByType(Int::class.javaObjectType))
```

This function filters any iterable object to only the types that are assignable to the given type `T`. But to do so, `isInstance` must use reflection to check whether an element is an instance of `T`. Also, you have to suppress unchecked cast warnings, and calling the function could be easier.

Luckily, there is a “trick” in Kotlin called *reification*. Reification allows you to access generic type information—but only in inline functions. Because for these, the concrete type argument is inlined into the call site, allowing you to make it available at runtime. Listing 4.53 provides a better implementation of the function from Listing 4.52.

#### Listing 4.53 A Generic Function Using Reification

[Click here to view code image](#)

---

```
inline fun <reified T> Iterable<*>.filterByType(): List<T> {
    return this.filter { it is T }.map { it as T? }
}

val elements = listOf(4, 5.6, "hello", 6, "hi", null,
    println(elements.filterByType<Int>()) //
```

This is now an inline function, and the type parameter `T` has the `reified` modifier, which allows you to now use the `is` operator with it. Also, there are no warnings to suppress, and the call site is concise. The way this works is that all

occurrences of `T` in the inline function are replaced with the concrete type argument. Concretely, the function call from [Listing 4.53](#) is effectively transformed to the function call in [Listing 4.54](#).

#### **Listing 4.54 Inlined Code of Generic Function**

[Click here to view code image](#)

```
elements.filter { it is Int }.map { it as Int? } //
```

In short, reification is useful if you need access to a generic type parameter at runtime but is available only for inline functions—you cannot generally prevent type erasure on the JVM.

## **Covariance and Contravariance**

This section first introduces the concepts of types versus classes, and then moves on to variance, more specifically covariance and contravariance.

### **Java Note**

So far, generics in Kotlin work the same way as in Java. But diving deeper into generics and variance now, you'll see several ways in which Kotlin improves on Java.

First, you have to differentiate between classes and types. Consider, for instance, Kotlin's `Int` class. This is also a type. But what about the nullable `Int?` then? This is not a class: Nowhere do you declare a class `Int?`. But it's still a type, the type of nullable integers. Similarly, `List<Int>` is not a class but a type; `List` is a class. The same holds for all generic classes—concrete instantiation induces a new *type*. This is what makes generics so powerful. [Table 4.1](#) gives more examples of classes versus types.

Table 4.1 Classes Versus Types

Entity	Is It a Class?	Is It a Type?
Task	Yes	Yes
Task?	No	Yes
List<Task>	No	Yes
List	Yes	Yes
(Task) -> Unit	No	Yes

**Tip**

Think about it like this: Classes and types are in a subset relation, meaning that the set of all classes is a subset of the set of all types. In other words, every class is also a type, but not every type is also a class.

Now that you know the difference between classes and types, let's explore the concept of variance, starting with covariance.

## Covariance

Intuitively, covariance means that a subtype can be used in place of one of its supertypes. You're used to this from variable assignments, where you can assign an object of a subclass to a variable of a superclass, as in Listing 4.55.

### Listing 4.55 Covariance of Subclasses in Assignments

[Click here to view code image](#)

```
val n: Number = 3 // Int
val todo: Archivable = Task("Write book", 99) // Task
```

Beside assignments, return types also allow the use of variance, as shown in Listing 4.56.

### Listing 4.56 Covariance of Return Types

[Click here to view code image](#)

```
abstract class Human(val birthday: LocalDate) {  
  
    abstract fun age(): Number // Specifies that method  
}  
  
  
class Student(birthday: LocalDate) : Human(birthday)  
  
    // Return types allow variance so that Int can be used  
    // instead of Number  
  
    override fun age(): Int = ChronoUnit.YEARS.between(  
        LocalDate.now(), birthday)  
}
```

So far, this is completely intuitive because type safety is preserved even if you use a subclass in these places.

**Java Note**

In Java, arrays are covariant, meaning `Student[]` is a subtype of `Human[]`, but this is not type-safe. It can cause an `ArrayStoreException` at runtime:

[Click here to view code image](#)

```
Number[] arr = new Integer[3]; // Arrays are covariant  
  
arr[0] = 3.7; // Causes ArrayStoreException
```

In Kotlin, arrays are not covariant and therefore type-safe. Covariance of arrays would allow you to instantiate an array of integers and assign it to an array of numbers. Adding elements to that array would then not be type-safe because you could add any number to that array, which doesn't fit into the integer array that is actually used. This is illustrated in Listing 4.57.

**Listing 4.57 Invariance of Arrays in Kotlin (Type-Safe)**

[Click here to view code image](#)

```
val arr: Array<Number> = arrayOf<Int>(1, 2, 3) // Create array of integers  
  
arr[0] = 3.1415 // Won't compile
```

The same potential problem applies to collection types and generic types in general. That's why, for instance, Java's collections such as `List<E>` and `Set<E>` (which are *mutable* in Java) are not covariant to preserve type safety, as demonstrated in [Listing 4.58](#).

**Listing 4.58 Invariance of Java Collections (Type-Safe)**

[Click here to view code image](#)

```
List<Number> numbers = new ArrayList<Integer>(); //
```



We say that `List<E>` is not *covariant with respect to* its type parameter `E` (in Java). And as you know now, it would be unsafe if it was because it would have the same problem shown above for arrays. So why does [Listing 4.59](#) compile just fine in Kotlin?

**Listing 4.59 Variance of Read-Only Collections in Kotlin (Type-Safe)**

[Click here to view code image](#)

```
val numbers: List<Number> = listOf<Int>(1, 2, 3) //
```

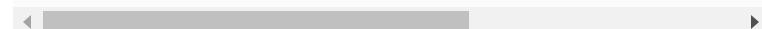


This is in fact type-safe in Kotlin. To see why, remember that Kotlin clearly differentiates between mutable and read-only collections in its standard library (in contrast to Java). Also, recall that the problem with type safety stemmed from adding new elements to an array or list. Generally, covariance of generic types with respect to a type parameter `T` is type-safe as long as objects of type `T` are never *consumed* but only *produced*. Thus, covariance is safe for the read-only list above but no longer safe for mutable collections, which is why those are *not* covariant in Kotlin, as shown in [Listing 4.60](#).

**Listing 4.60 Invariance of Mutable Collections in Kotlin**

[Click here to view code image](#)

```
val numbers: MutableList<Number> = mutableListOf<Int>
```



If this was possible, you could again produce an error at runtime simply by calling `numbers.add(3.7)`, which

would try to add a double into a list of integers. In other words, in `MutableList<T>`, the type parameter `T` is *consumed* by methods such as `add(t: T)` and therefore not only produced. This makes covariance unsafe.

As an intuition, we said that covariance means you can use a subtype in place of a supertype. More formally, it means that subtype relations are preserved when wrapping with a covariant type. Assume you have a covariant type `Covariant<T>`, and a type `Child` with its parent type `Parent`. Covariance implies that `Covariant<Child>` is still a subtype of `Covariant<Parent>`, just like `List<Int>` is a subtype of `List<Number>` because Kotlin's `List<T>` is covariant.

## Contravariance

Contravariance is the counterpart to covariance and intuitively means that, in place of a subtype, a supertype can be used. This may be less intuitive at first glance, but the following subsection demonstrates some intuitive use cases. While you can think of covariant types as *producers* (they cannot consume `T`, only produce `T`), you can think of contravariant types as *consumers*. The following subsection introduces several examples of contravariant types.

Formally, contravariance means that subtype relationships are reversed when wrapping with a contravariant type. If you have a contravariant type `Contravariant<T>`, then `Contravariant<Parent>` becomes a subtype of `Contravariant<Child>`. For example, a `Comparable<Number>` is a `Comparable<Int>`. This makes sense because an object that knows how to compare numbers also knows how to compare integers. So the intuition is not formally correct here, and you are in fact still using subtypes in place of supertypes.

## Declaration-Site Variance

Kotlin introduces declaration-site variance as a language concept so that variance becomes part of a class' contract. This way, the class behaves as a co- or contravariant type at all use sites. For this purpose, Kotlin uses the **in** and **out** modifiers. Listing 4.61 demonstrates how to declare a covariant, and therefore read-only, stack class.

**Listing 4.61 Covariant Stack Class**

[Click here to view code image](#)

```
open class Stack<out E>(vararg elements: E) { // 'out' is the variance annotation

    protected open val elements: List<E> = elements.toList()

    fun peek(): E = elements.last()

    fun size() = elements.size

}

val stack: Stack<Number> = Stack<Int>(1, 2, 3) // A]
```

This defines a read-only stack type. As you know now, read-only access is essential to be able to make this type covariant. Covariance with respect to the type parameter **E** is indicated by **<out E>**, so that **E** can only appear only in *out-position*—and indeed, it only appears as return type in the `peek` function. Note that, in the `elements` type `List<E>`, **E** appears in covariant position as well because `List<E>` is covariant. Using `MutableList<E>` would cause a compile-time error because **E** would appear in invariant position there.

Due to **<out E>**, it's now part of the contract of the `Stack` class that it only produces objects of type **E** but never consumes them. However, you can see that covariance comes at a cost—you cannot define methods in the `Stack` class that add or remove elements. For this, you'll need a mutable subclass, like in the predefined collection classes. Listing 4.62 gives an example.

**Listing 4.62 An Invariant Mutable Stack Class**

[Click here to view code image](#)

```
class MutableStack<E>(vararg elements: E) : Stack<E>{  
    override val elements: MutableList<E> = elements.toMutableList()  
  
    fun push(element: E) = elements.add(element)    /  
    fun pop() = elements.removeAt(elements.size - 1)    /  
}  
  
  
val mutable: MutableStack<Number> = MutableStack<Int>  
◀ ▶
```

Because a mutable stack is no longer purely a producer and uses `E` at in-positions (as parameter of `push` and in `MutableList<E>`), it cannot have the `out` modifier. Hence, `MutableStack<Int>` is not a subtype of `MutableStack<Number>`. Note that the stack implementations here are very simplistic and don't handle exceptions such as empty stacks.

Conversely, you can declare a type as being contravariant by using the `in` modifier, as in Listing 4.63.

**Listing 4.63 A Contravariant `Compare<T>` Type**

[Click here to view code image](#)

```
interface Compare<in T> {          // 'in' indicates contravariance  
    fun compare(a: T, b: T): Int    // T is only used at the boundary  
}  
◀ ▶
```

The `Compare` type is now contravariant with respect to its type parameter. Intuitively, this means you can use a more general comparator to compare more specific objects. Consider Listing 4.64 as an example and recall that `Task` is a subclass of `Issue`.

**Listing 4.64 Using Contravariance of `Compare<T>`**

[Click here to view code image](#)

```
val taskComparator: Compare<Task> = object : Compare<  
    override fun compare(a: Issue, b: Issue) = a.priori  
}
```



This code uses an object expression to initialize an object of type `Compare<Issue>` and assigns it to a variable of type `Compare<Task>`. Contravariance makes sense here because an object that can compare issues in general can also compare tasks. Notice that you can think of `Compare` (or comparators) as consumers because they only consume objects of type `T` to compare them but never produce objects of type `T`. To familiarize yourself with contravariance, [Listing 4.65](#) presents another example.

[Listing 4.65 Another Contravariant Type Repair<T>](#)

[Click here to view code image](#)

```
interface Repair<in T> { // Contravariant  
    fun repair(t: T) // T in in-position  
}  
  
open class Vehicle(var damaged: Boolean)  
class Bike(damaged: Boolean) : Vehicle(damaged)  
  
class AllroundRepair : Repair<Vehicle> { // !  
    override fun repair(vehicle: Vehicle) {  
        vehicle.damaged = false  
    }  
}  
  
val bikeRepair: Repair<Bike> = AllroundRepair() // !
```

Here, `Bike` is a subtype of `Vehicle` and the interface `Repair<in T>` indicates that an object of that type is able to repair `T` objects. Hence, `AllroundRepair` is able to repair vehicles of all kinds, in particular bikes. This is a typical scenario of contravariance, and again it makes sense that you can assign `bikeRepair` to an object that implements `Repair<Vehicle>`; because it can repair any vehicle, it can also repair bikes.

With declaration-site variance, the implementor of a type has to think about whether the type can support variance already when declaring it. This takes the burden from the user of the type who would otherwise have to think about variance at the use site. This results in more readable code. However, not every type can support co- or contravariance, as you've seen with mutable collections. This only works in the special cases where objects of a type *always* act purely as a producer (covariant) or a consumer (contravariant).

If this isn't the case, Kotlin still supports use-site variance as well. This is useful if a type is not strictly a producer or consumer, but you're using it like one in a part of your code. Then, in that part of your code, the type can be co- or contravariant.

**Java Note**

Java supports only use-site variance and no declaration-site variance, but there is a JEP for it.<sup>6</sup>

6. <http://openjdk.java.net/jeps/300>

Table 4.2 summarizes the distinction between covariant, contravariant, and invariant types.

Table 4.2 An Overview of Variance

	Covariance	Contravariance	Invariance
Role	Producer	Consumer	Producer and Consumer
Example	Kotlin's List<T>	Repair<T>	MutableList<T>
Keyword	out	in	-

Finally, Figure 4.1 shows the effect of variance on the subtype relationships in Kotlin's type system. This diagram only shows non-nullable types but the same relationships exist for their corresponding nullable types. Note that **Nothing** really is the absolute subtype of all other types while **Any** is still a subtype of the absolute supertype **Any?** that is not shown here.

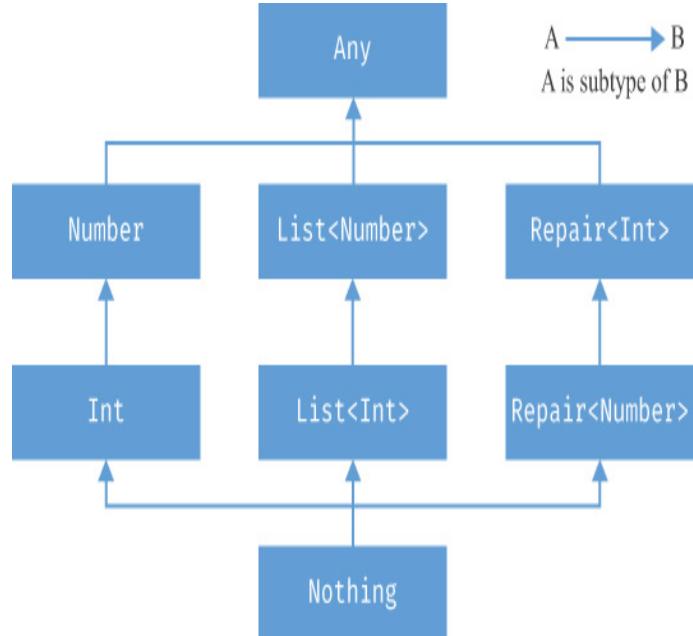


Figure 4.1 Kotlin-type system in the light of variance

## Use-Site Variance

For use-site variance, Kotlin has type projections. A projection is simply a restriction of a type `Type<T>` to a subset of itself. For instance, by projecting `Type<T>` to its covariant subset `Type<out T>`, it becomes restricted to members that don't consume `T`. Effectively, `Type<out T>` is a producer. Conversely, projection to `Type<in T>` restricts the type to those members that don't produce `T`, thus making it a consumer.

### Java Note

Java supports only use-site variance and uses the syntax `List<? extends T>` for covariance and `List<? super T>` for contravariance. These are called wildcard types. You may be familiar with the "PECS" mnemonic that Joshua Bloch introduced to better remember these in Java; it stands for Producer-extends-Consumer-super. In Kotlin, it's a lot easier thanks to the more expressively named modifiers `in` and `out`: Producer-out-Consumer-in.

[Listing 4.66](#) gives an example of use-site covariance in assignments.

### Listing 4.66 Use-Site Covariance in Assignments

[Click here to view code image](#)

```
val normal: MutableList<Number> = mutableListOf<Int>()

val producer: MutableList<out Number> = mutableListOf()

// Can only read from producer, not write to it

producer.add(???)           // Parameter type is Not

val n: Number = producer[0] // Returns Number
```

Although `MutableList` is not covariant by itself, you can state at use-site that you want to use it as a producer using `MutableList<out Number>`. This way, consumer methods are no longer callable because, wherever the type parameter `T` was used as a method parameter, the type becomes `Nothing`. And because there is no valid object of type `Nothing`, it is impossible to call methods such as `add`

or `remove`. For the contravariant case, Listing 4.67 gives an example.

**Listing 4.67 Use-Site Contravariance in Assignments**

[Click here to view code image](#)

```
val normal: MutableList<Int> = mutableListOf<Number>()

val consumer: MutableList<in Int> = mutableListOf<Num

// Can write to consumer normally, but only read values
consumer.add(4)

val value: Any? = consumer[0] // Return type is Any?
```

Here, the mutable list is used as a consumer, a contravariant type, using `<in Int>`. Wherever `T` appears as a parameter in the generic class definition, it becomes `Int`. Thus, `add` accepts values of type `Int`. More important, `T` in out-position becomes `Any?` because whatever value is read from the list, it can be stored in a variable of type `Any?`. Hence, it's not strictly a consumer. But reading data from it is discouraged by the fact that it can only return values with type `Any?`.

Note the contrary nature of how `T` is mapped in the co- and contravariant cases. For covariance, to prevent writing to the list, `T` is transformed to `Nothing`, the absolute bottom of the type hierarchy in Kotlin. For contravariance, it's transformed to `Any?`, the absolute supertype in the type hierarchy.

More commonly, use-site variance is used to increase flexibility of methods. Imagine you want to implement a method to move all elements from one mutable stack onto another mutable stack. Listing 4.68 shows a naïve implementation along with its problems.

**Listing 4.68 Function Without Use-Site Variance**

[Click here to view code image](#)

```
fun <E> transfer(from: MutableStack<E>, to: MutableSt
```

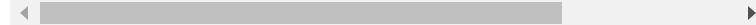
```
while (from.size() > 0) { to.push(from.pop()) }

}

val from = MutableStack<Int>(1, 2, 3)

val to = MutableStack<Number>(9.87, 42, 1.23)

transfer(from, to) // Compile-time error, although i
```



Even though it would be safe to transfer all elements from the stack of integers to the stack of numbers, this is not guaranteed in the contract of the transfer function. Since `MutableStack` is invariant, you might as well be pushing elements onto the `from` stack or reading elements from the `to` stack. Use-site variance solves this problem, as in [Listing 4.69](#).

#### **Listing 4.69 Use-Site Variance Increases Function Flexibility**

[Click here to view code image](#)

```
fun <E> transfer(from: MutableStack<out E>, to: MutableStack<in E>)

    // Same as before

}

// ...

transfer(from, to) // Works now due to variance
```



Note that covariance does not imply immutability in any way; even though you cannot call methods like `add` or `remove` on a `MutableList<out Int>`, you could still call a method such as `clear`, which removes all elements. This is also the case in [Listing 4.69](#). In fact, the `from` stack is emptied as a side effect of the `transfer` function, due to the calls to the `pop` function.

**Tip**

Prefer declaration-site variance to use-site variance when the class allows it, meaning when it's a producer or consumer. This simplifies code at the use site and allows for increased flexibility without additional syntax.

## Bounded Type Parameters

Sometimes, you want to restrict the possible concrete types that can be used for a generic type parameter. For instance, the interface `Repair<in T>` may only make sense for type parameters that are a subclass of `Vehicle`. That way, users of the class cannot instantiate a `Repair<Person>` or a `Repair<Int>`. Listing 4.70 does this by bounding the type parameter.

**Listing 4.70 Bounded Type Parameter**

[Click here to view code image](#)

```
interface Repair<in T : Vehicle> { // Upper bound for T

    fun repair(t: T)

}
```

The only thing that changed here is the upper bound of `T` in `<in T : Vehicle>`. Like a supertype, the upper bound is specified after a colon. If you want to define multiple upper bounds for the same type parameter, you must use a `where` clause as in Listing 4.71. For this example, let's assume there are vehicles that are not repairable so you have a separate interface `Repairable` that is not implemented by `Vehicle`.

**Listing 4.71 Multi-Bounded Type Parameter**

[Click here to view code image](#)

```
interface Repairable { ... }

interface Repair<in T>

    where T : Vehicle, // where clause lists all type

        T : Repairable {

    fun repair(t: T)

}
```

Here, the upper bounds are pulled out of the angle brackets and defined in the **where** clause instead. Note that each upper bound is separated by a comma, and the class body comes after the **where** clause. Also, by convention, the **where** clause is indented by twice the normal indentation and the upper bounds are aligned at the same indentation level.

**Tip**

You can use `where` clauses whenever you define a type parameter, not only when there are multiples for the same type. In particular, you can use it in generic methods to avoid repeating an upper bound for multiple parameters:

[Click here to view code image](#)

```
fun <T> min(a: T, b: T): T where T : Comparable<T> = if (a < b) a else
```

◀ ▶

Note that you can also inline the `where` clause if it's sufficiently short. Otherwise, you should favor making the code less dense and splitting it up into multiple lines.

Upper bounds induce a stricter contract on the objects of type `T`. In other words, you know more about the objects and can therefore do more with them. For example, inside the `repair(t: T)` method in the interface `Repair` from [Listing 4.72](#), you get access to all members of `Vehicle` as well as `Repairable` on `t` because you know that `t: T` fulfills the contracts of both these types.

## Star Projections

Sometimes, you don't need to know anything about the concrete type of a type parameter. Still, you want your code to be type-safe. In such cases, you can use Kotlin's *star projections*. [Listing 4.72](#) shows an example of a function that prints all elements of an array.

### [Listing 4.72 Using Star Projections](#)

[Click here to view code image](#)

```
fun printAll(array: Array<*>) { // Nothing known about array
    array.forEach(::println)
}
```

```
printAll(arrayOf(1, 2, 3))

printAll(arrayOf("a", 2.7, Person("Sandy")))
```

Here, you don't care about the concrete type parameter of `Array`, and star projections let you express that as `Array<*>`. Note that you could also define a generic function `fun <E> printAll(array: Array<E>)`, but there's no need for this function to be generic. You could even call the generic version in the same way because the compiler automatically infers that you're calling `printAll<Int>(...)` or `printAll<Any>(...)`.

When working with star projections, the compiler differentiates between three cases when accessing them. You should understand what it does and how to work with it.

1. For covariant types `Producer<out T>` that you can only read from, `Producer<*>` becomes `Producer<out Any?>` because that's the only type-safe choice if `T` is unknown and unrestricted. However, if `T` has an upper bound `Upper` as in `Producer<out T: Upper>`, it becomes `Producer<out Upper>` because all read elements can be safely cast to `Upper`.
2. For contravariant types, `Consumer<*>` becomes `Consumer<in Nothing>` because there's no way to safely pass an element of type `T` if `T` is unknown. This is easy to see because nothing stops you from actually instantiating a `Consumer<Nothing>`, making it in fact impossible to pass any element of type `T`.
3. Lastly, an invariant type `Inv<T>` combines the two cases above. For read access, `Inv<*>` becomes `Inv<out Any?>` (or `Inv<out Upper>` if `T` has an upper bound). Again, that's the only type-safe choice for an unknown `T`. Conversely, for write access, `Inv<*>` becomes `Inv<in Nothing>` as it's impossible to safely pass an element of an unknown type `T`.

[Listing 4.73](#) exemplifies the first two cases. It defines an interface with one contravariant and one covariant type parameter, and it shows how they behave with star projections.

#### Listing 4.73 Star Projections for Variant Types

[Click here to view code image](#)

```
interface Function<in T, out R> { // Contravariant v

    fun apply(t: T): R
```

```

}

object IssueToInt : Function<Issue, Int> {

    override fun apply(t: Issue) = t.priority

}

// Star projections

val f1: Function<*, Int> = IssueToInt // apply() requires Issue

val f2: Function<Task, *> = IssueToInt // apply() consumes Task

val f3: Function<*, *> = IssueToInt // apply() requires Task

val errand = Task("Save the world", 80)

val urgency: Any? = f2.apply(errand)

```

For the following, note that `apply` is a consumer of `T` and a producer of `R`. Using a star projection on the contravariant type parameter `T` transforms `apply(t: T)` to `apply(t: Nothing)` so that it becomes impossible to call this method. You could still call methods that don't have `T` as a parameter if there were any.

Conversely, projecting the covariant type parameter `R` doesn't prevent you from calling any methods but makes all methods with return type `R` return `Any?`, resulting in `apply(t: T): Any?`. This is why, when calling `f2` in the last line, `urgency` does not have type `Int`. This is intuitive because you can read the declaration `f2: Function<Task, *>` as "a function that takes in a task and returns a value of unknown type."

Lastly, projecting both type parameters declares a function taking in some unknown type and returning an unknown type. This combines the two restrictions above; you can no longer

call methods that consume `T`, and all methods that return `R` can only safely return `Any?`.

An example for the third case, star projection of an invariant type, was already given in [Listing 4.72](#): Inside `printAll`, you cannot call `array.set` because its signature is `array.set(index: Int, value: Nothing)`.

Reading from the array returns values of type `Any?`. Hence, the restrictions at work are similar to `Function<*, *` except that there is only one type parameter.

In summary, when you use star projections, the compiler ensures type safety by projecting types appropriately where necessary.

## SUMMARY

As an object-oriented language, Kotlin lets you compose systems based on classes and objects that encompass data as properties and behavior as methods. Such object-oriented systems should follow the principle of information hiding to keep up maintainability and reduce coupling between classes. Visibilities are the primary way to implement information hiding.

Combining Kotlin's object-oriented concepts with functional programming features offers powerful ways to write more concise code, solve common tasks more easily, and increase reusability. For example, nonfixed parts of the logic in classes may be passed as lambda expressions. Consider the Strategy pattern, a well-known design pattern for object-oriented systems. It can be implemented more easily by using lambdas that define specific strategies.

You now know how to declare and use object-oriented entities in Kotlin. Here's an overview of the main entities along with a differentiation of which is used in which kind of situation.

1. Use normal classes for general entities in your system.
2. Use interfaces to define the most high-level or abstract contracts for your types.
3. Use abstract classes for non-instantiable classes that are not directly entities in the system but encapsulate common logic of multiple subclasses.

- 4.** Use inheritance (judiciously) to avoid code duplication and to ideally implement only the differences between more specialized subclasses.
- 5.** Use data classes for types carrying mainly or only data.
- 6.** Use sealed classes to implement restricted hierarchies (or algebraic types).
- 7.** Use enum classes to define finite sets of distinct values.
- 8.** Use object expressions to create ad-hoc objects or an implementation of one or more interfaces that is used only once.
- 9.** Use object declarations to create singletons or companion objects.
- 10.** Use generic classes if the logic doesn't depend on the specific type of elements, most commonly in data structures.

Apart from these object-oriented entities, Kotlin provides a variety of convenient language features that help solve common development tasks. These features include delegated properties to reuse common accessor logic, extension functions to easily enhance existing APIs, declaration-site variance to avoid duplicate code at the call site, and many more. All these can greatly improve developer productivity and code quality.

# 5

## Interoperability with Java

*Synergy is better than my way or your way. It's our way.*

Stephen Covey

Interoperability has been one of Kotlin's priorities from its inception. In this chapter, you'll learn the ins and outs of mixing Kotlin and Java, how their language concepts map to each other, what happens under the hood in the Kotlin compiler to aid interoperability, and how you can write code that further facilitates it.

### USING JAVA CODE FROM KOTLIN

Java code can be called from Kotlin in the way you would expect most of the time. In this section, we'll explore the details and possible pitfalls, as well as how to write Kotlin-friendly Java code and how to improve your Java code to provide additional information for the Kotlin compiler, especially about nullability.

First, it's important to appreciate the fact that you can easily use the Java standard library and any third-party library in a natural way. This is demonstrated in [Listing 5.1](#).

**Listing 5.1 Using Java Libraries from Kotlin**

[Click here to view code image](#)

---

```
import com.google.common.math.Stats

import java.util.ArrayList
```

```
// Using the standard library

val arrayList = ArrayList<String>() // Uses java.util.ArrayList

arrayList.addAll(arrayOf("Mercury", "Venus", "Jupiter"))

arrayList[2] = arrayList[0] // Indexed access

// Looping as usual, ArrayList provides iterator()

for (item in arrayList) { println(item) }

// Using a third-party library (Google Guava)

val stats = Stats.of(4, 8, 15, 16, 23, 42)

println(stats.sum()) // 108.0
```

Concepts that differ in Java are mapped to Kotlin automatically. For instance, the indexed access operator for collections is translated to `get` and `set` calls under the hood. Also, you can still iterate anything that provides an `iterator` method using a `for` loop. This is the easiest and most prevalent use case: simply using Java from Kotlin seamlessly without even thinking about it. With this in mind, let's now explore special cases to be able to handle mixed-language projects confidently.

**Note**

What's critical to Kotlin is that you can use any existing Java libraries, opening up a plethora of powerful code to reuse. This includes the standard library<sup>1</sup> and any third-party libraries and frameworks, whether it be Spring,<sup>2</sup> JUnit, or the Android Software Development Kit (SDK).

1. <https://spring.io/>
2. <https://junit.org/>

## Calling Getters and Setters

In Kotlin, you don't explicitly call `getSomething` or `setSomething` but instead use the property syntax that calls the getter and setter under the hood. You want to keep it that way when using Java field accessors for consistency and brevity, and it's possible by default.

[Listing 5.2](#) gives an example.

### **Listing 5.2 Calling Java Getters and Setters**

[Click here to view code image](#)

---

```
// Java

public class GettersAndSetters {

    private String readOnly = "Only getter defined";

    private String writeOnly = "Only setter defined";

    private String readWrite = "Both defined";

    public String getReadOnly() { return readOnly; }

    public void setWriteOnly(String writeOnly) { this.v

    public String getReadWrite() { return readWrite; }

    public void setReadWrite(String readWrite) { this.r

}

// Kotlin

private val gs = GettersAndSetters()

println(gs.readOnly)           // Read-only attribute

gs.readWrite = "I have both"  // Read-write attribute

println(gs.readWrite)

gs.setWriteOnly("No getter") // Write-only property
```

---

This automatic conversion to a Kotlin property works as long as a Java class provides a parameterless method starting with “`get`” and optionally a single-parameter method starting with “`set`”. It also works for Boolean expressions where the getter starts with “`is`” instead of “`get`”, but there’s currently no such mechanism if they start with “`has`” or other prefixes.

Write-only properties are currently not supported in Kotlin, which is why `setWriteOnly` cannot be called with property syntax as `writeOnly`.

## Handling Nullability

Regarding nullability, there’s a gap in expressiveness between Kotlin and Java that has to be handled: in Kotlin, every variable is either nullable or not, in Java there’s no nullability information expressed in the language itself because every variable may potentially be `null` (except primitive types). However, handling all data coming from Java as nullable in Kotlin would introduce unnecessary code complexity due to `null` handling. To handle this problem, Kotlin uses so-called *platform types* for data coming from Java.

## Nullables and Mapped Types

As mentioned in [Chapter 2](#), Diving into Kotlin, primitive types are mapped to the corresponding (non-nullable) Kotlin types. For example, `int` is mapped to `kotlin.Int`, `boolean` to `kotlin.Boolean`, and vice versa. This makes sense because primitive types cannot be `null` in Java. However, boxed types must be mapped to nullable types, for instance `java.lang.Integer` to `kotlin.Int?`, `java.lang.Character` to `kotlin.Char?`, and vice versa.

Generally, interoperability with Java introduces the problems of mapping any objects coming from Java to a Kotlin type. Obviously, it’s not possible to map them all to non-nullable

types because that would be unsafe. But because you want to exclude nullability from your code as much as possible, Kotlin doesn't simply make them all nullable either. Kotlin supports nullability annotations from JSR 305,<sup>3</sup> JetBrains, Android, FindBugs,<sup>4</sup> and more that define annotations such as `@Nullable` and `@NotNull`. This additional information allows a better mapping to appropriate Kotlin types.

3. <https://jcp.org/en/jsr/detail?id=305>

4. <http://findbugs.sourceforge.net/>

## Platform Types

Ultimately, the Kotlin team decided to leave the nullability decision to the developer in cases where nullability cannot be inferred. This is represented using *platform types* such as `SomeType!`. Here, `SomeType!` means either `SomeType` or `SomeType?`. These are the cases where you can choose to store an object of that platform in a nullable or non-nullable variable. See [Table 5.1](#) for several examples of mapped types.

Table 5.1 Mapped Types and Nullability

Java Declaration	Kotlin Type	Explanation
<code>String name;</code>	<code>kotlin .String!</code>	May or may not be null
<code>@NotNull String name;</code>	<code>kotlin .String</code>	Supposed to not be null
<code>@Nullable String name;</code>	<code>kotlin .String?</code>	Can be null
<code>@NotNull String name = null;</code>	<code>kotlin .String</code>	Source of null pointer exception even when using Kotlin

Java objects without annotation may or may not be **null**, so Kotlin will map them to the corresponding platform type, allowing you to treat it as either nullable or non-nullable. If annotations are used, Kotlin uses the additional knowledge about nullability. However, objects marked as `@NotNull` in Java that still contain **null** by mistake can cause a null pointer exception in Kotlin.

**Note**

You cannot explicitly create platform types yourself. Types such as `String!` are not valid syntax in Kotlin and are only used by the compiler and IDE in order to communicate types to you.

So platform types denote that the associated type may or may not be nullable, and the developer is able to decide whether the platform type should be handled as nullable or not by using the corresponding type; this is shown in Listing 5.3.

**Listing 5.3 Handling Platform Types in Kotlin**

[Click here to view code image](#)

```
// Java

public static String hello() { return "I could be null" }

// Kotlin

val str = hello() // Inferred type: String
println(str.length) // 15

val nullable: String? = hello() // Explicit type: String?
println(nullable?.length) // 15
```

Without an explicit type, the compiler infers the non-nullable variant of the platform type so that you can use it without handling **null**. This is unsafe if you don't think about whether the method you're calling does actually never return **null**. Hence, I recommend stating the type explicitly when calling a Java method. That way, your intention and the fact that you thought about nullability are explicit.

You'll quickly come across more complex platform types as well. These come from mapped types and generic types that may or may not carry nullability information. Consider the examples in [Table 5.2](#).

Table 5.2 Examples of Platform Types (Combined with Mapped Types)

Java Return Type	Platform Type	Explanation
String	String!	May or may not be null, thus either String or String?
List<String>	(Mutable)List<String!>!	Maybe mutable and maybe nullable list, containing maybe nullable Strings
Map<String>	(Mutable)Map<String!, String!>!	Maybe mutable and maybe nullable map, containing maybe nullable Ints as keys and maybe nullable Strings as values
Object[]	Array<(out) Any!>!	Maybe nullable array, containing maybe nullable Anys or maybe subtypes of it (because arrays are covariant in Java)
long[]	LongArray!	Maybe nullable array of primitive longs (the whole array may be null)
List<? super Integer>	MutableList<in Int!>!	Maybe nullable and mutable list, containing Ints or parent types of it

The first row demonstrates that unannotated types are translated to platform types by default. Next, a `List<T>` coming from Java may be used as a mutable list or not, and may itself again be nullable or not (like any other type). Additionally, elements of the list are also potentially nullable. The same holds for other generic types and their type

arguments, as shown in the third row. There, the generic type arguments `Int` and `String` are mapped to their platform type counterparts. This is necessary because Java allows `null` as both key and value.

Regarding the fourth row, an array coming from Java is also potentially nullable. On top of that, it may contain subtypes of the declared type argument because arrays are covariant in Java so that an `Object[]` may contain subtypes of `Object`. This is why it is translated to a potentially out-projected type. Note also that `java.lang.Object` is translated to `kotlin.Any` as defined per the mapped types. A complete list of mapped types is available in the language documentation.<sup>5</sup> Additionally, arrays of primitive types are mapped to their corresponding specialized Kotlin class, such as `IntArray`, `DoubleArray`, and `CharArray`. Lastly, a contravariant list coming from Java becomes an in-projected type in Kotlin. As you can see, it even becomes a definitely mutable list because it wouldn't be usable otherwise as an in-projected list. However, you shouldn't use a wildcard type as a return type in Java because it makes the return type harder to work with for the caller.

5. <https://kotlinlang.org/docs/reference/java-interop.html#mapped-types>

**Note**

Kotlin not only maps primitive arrays to their corresponding mapped types (`IntArray`, `LongArray`, `CharArray`, and so forth) and vice versa, it also incurs no performance cost compared to Java when working with these.

For instance, reading a value from an `IntArray` does not actually call `get`, and writing a value does not generate a call to `set`. Also, iteration and `in` checks generate no iterator or other objects. The same holds for all mapped types of primitive arrays.

Recall that such arrays can be created in Kotlin directly using dedicated helper methods `IntArrayOf`, `DoubleArrayOf`, `LongArrayOf`, and so on.

## Adding Nullability Annotations

Handling platform types is only required if there's no nullability information the compiler can use. Even though the Java language itself doesn't have a concept for this, you can actually attach nullability info via annotations such `@NotNull` and `@Nullable`. Such annotations are already fairly widespread, and there are various libraries containing such annotations. The Kotlin compiler can currently process the following nullability annotations.

- The Android annotations `@NonNull` and `@Nullable` from the package `android.support.annotations`. These can be used out of the box on Android.
- The JetBrains annotations `@NotNull` and `@Nullable` from `org.jetbrains.annotations` that are used for static analysis in Android Studio and IntelliJ IDEA and can also be used out of the box in both.
- Annotations from the `javax.annotation` package.
- FindBugs annotations from the package `edu.umd.cs.findbugs.annotations`.
- The Lombok annotation `lombok.NonNull`.<sup>6</sup>
- Eclipse's<sup>7</sup> nullability annotations from `org.eclipse.jdt.annotation`.

[6. https://projectlombok.org/features/NonNull](https://projectlombok.org/features/NonNull)

[7. http://www.eclipse.org/](http://www.eclipse.org/)

With these, Kotlin supports the most widely used annotations. Listing 5.4 provides an example for how to use JetBrains' annotations; the others work the same way. If you're using others, such as NetBeans<sup>8</sup> or Spring Framework annotations, you'll have to transition to one of the above to get better type inference in Kotlin.

[8. https://netbeans.org/](https://netbeans.org/)

### Listing 5.4 Using Nullability Annotations

[Click here to view code image](#)

```
// Java

public static @Nullable String nullable() { return null }

public static @NotNull String nonNull() { return "Cool" }
```

```
// Kotlin

val s1 = nullable() // Inferred type: String?

val s2 = nonNull() // Inferred type: String
```

Note that nothing stops you from returning **null** from the `nonNull` method in Java, you'll only receive a warning in AndroidStudio and IntelliJ, depending on which nullability annotations you use. For instance, the JetBrains annotations are used for static analysis in Android Studio and IntelliJ so that such code would yield a warning.

In this way, nullability annotations allow you to get more type information in Kotlin. In other words, they disambiguate the impreciseness of `Type!` to either `Type` or `Type?`. Let's consider the examples in [Table 5.3](#) and contrast them to the above table of platform types.

Table 5.3 Examples of Inferred Types (When Using Nullability Annotations)

Java Return Type	Inferred Kotlin Type	Explanation
<code>@NonNull String</code>	<code>String</code>	Non-nullable String
<code>@Nullable String</code>	<code>String?</code>	Nullable String
<code>List&lt;String&gt;</code>	<code>(Mutable )List&lt;String!&gt;</code>	Non-nullable but maybe mutable list, containing maybe nullable Strings
<code>List&lt;@NonNull String&gt;</code>	<code>(Mutable )List&lt;String&gt;!</code>	Maybe nullable and maybe mutable list, containing non-nullable Strings
<code>List&lt;@Nullable String&gt;</code>	<code>(Mutable )List&lt;String&gt;</code>	Non-nullable but maybe mutable list, containing non-nullable Strings

In short, by resolving the ambiguity to either `@NonNull` or `@Nullable`, the return type of the Java method is mapped accordingly when called from Kotlin. Note that, to use annotations on generic types argument as in `List<@NonNull String>`, you need to use an implementation of JSR 308<sup>9</sup> such as the checker framework.<sup>10</sup> The purpose of JSR 308 is to allow annotating any type in Java, including generic type parameters, and it was incorporated into Java Standard Edition 8.

9. <https://jcp.org/en/jsr/detail?id=308>

10. <https://checkerframework.org/>

10. <https://checkerframework.org/>

## Escaping Clashing Java Identifiers

When mapping between languages, there is always a disparity between the keywords defined in the languages. Kotlin has several keywords that don't exist in Java and are thus valid Java identifiers, such as **val**, **var**, **object**, and **fun**. All these must be escaped using backticks in Kotlin as shown in Listing 5.5. This is done automatically in IntelliJ and Android Studio.

### Listing 5.5 Handling Name Clashes

[Click here to view code image](#)

```
// Java

class KeywordsAsIdentifiers {

    public int val = 100;

    public Object object = new Object();

    public boolean in(List<Integer> list) { return true; }

    public void fun() { System.out.println("This is fun"); }

}

// Kotlin

val kai = KeywordsAsIdentifiers()

kai.`val`

kai.`object`

kai.`in`(listOf(1, 2, 3))

kai.`fun`()
```



Even though it looks inconvenient, it's not something you have to think about explicitly thanks to the IDE. It still does clutter up the code a little but should only appear in rare cases anyway. If you have to escape frequently, you may want to

rethink your naming standards in Java—none of the above identifiers convey much meaning when used as identifiers.

## Calling Variable-Argument Methods

Calling **vararg** methods defined in Java works naturally so that you can pass in an arbitrary number of arguments. However, in contrast to Java, you cannot pass an array to a **vararg** method directly. Instead, you have to use the *spread operator* by prefixing a `*` to the array. [Listing 5.6](#) shows both ways to call a variable-argument method.

[Listing 5.6 Calling a Variable-Argument Method](#)

[Click here to view code image](#)

```
// Java

public static List<String> myListOf(String... strings)

    return Arrays.asList(strings);

}

// Kotlin

val list = myListOf("a", "b", "c")

val values = arrayOf("d", "e", "f")

val list2 = myListOf(*values)
```

## Using Operators

You can call Java methods like operators in Kotlin if they have the right signature. For instance, defining a `plus` or `minus` method in a class allows you to add or subtract objects of it. [Listing 5.7](#) provides an example.

[Listing 5.7 Using Java Methods as Operators](#)

[Click here to view code image](#)

```
// Java

public class Box {
```

```

private final int value;

public Box(int value) { this.value = value; }

public Box plus(Box other) { return new Box(this.value + other.value); }

public Box minus(Box other) { return new Box(this.value - other.value); }

public int getValue() { return value; }

}

// Kotlin

val value1 = Box(19)

val value2 = Box(37)

val value3 = Box(14)

val result = value1 + value2 - value3 // Uses 'plus'

println(result.value) // 42

```

Although you can write Java code that targets the predefined set of operators, you cannot define other methods that allow infix notation in Kotlin.

## Using SAM Types

Interfaces with a single abstract method (“SAM types”) can be called from Kotlin without boilerplate thanks to *SAM conversions*. Listing 5.8 shows how a lambda expression can be used to implement the single abstract method of a SAM interface.

**Listing 5.8 Using SAM Types from Kotlin**

[Click here to view code image](#)

---

```

// Java

interface Producer<T> { // SAM interface (single abstract method)
    T produce();
}

```

```
    T produce();

}

// Kotlin

private val creator = Producer { Box(9000) } // Infer type
```

In the Java code, this defines a SAM type, with `produce` being the single abstract method. To create a `Producer<T>` in Kotlin, you don't need to use an object expression (or an anonymous inner class as in Java) but can instead use a lambda expression that provides the implementation of the single abstract method. In [Listing 5.8](#), you have to add `Producer` in front of the lambda expression, otherwise the type of the right-hand side would be `( ) -> Box` and not `Producer<Box>`. This is not necessary when passing in a function argument, as shown in [Listing 5.9](#), because the compiler can then infer the type.

#### **Listing 5.9 SAM Conversion in Function Arguments**

[Click here to view code image](#)

```
// Kotlin

val thread = Thread { // No need to write Thread(Runnable)
    println("I'm the runnable")
}
```

The `Thread` constructor accepts a `Runnable`, which is instantiated here using SAM conversions. The code is equivalent to `Thread(Runnable { ... })`. But because the constructor argument is of type `Runnable`, you're not required to add an additional type hint in front of the lambda here. Consequently, you can use Kotlin's convention to move lambdas out of the parentheses and then omit the parentheses because the lambda is the only parameter.

This mechanism is extremely useful for many built-in SAM types such as `Comparator`, `Runnable`, and many listener classes.

### Further Interoperability Considerations

You've now seen that you can call Java code from Kotlin in a natural way most of the time. Sometimes, as with SAM conversions, it's even more convenient than from Java itself. The main point to keep in mind is nullability and how it maps to Kotlin. Other considerations to keep in mind can be explained briefly.

- Methods that throw a checked exception in Java can be called from Kotlin without having to handle the exception because Kotlin decided against checked exceptions.
- You can retrieve the Java class of an object as `SomeClass::class.java` or `instance.javaClass`.
- You can use reflection on Java classes from Kotlin and use a reference to the Java class as the entry point. For instance, `Box::class.java.getDeclaredMethods` returns the methods declared in the `Box` class.
- Inheritance works naturally across Kotlin and Java; both support only one superclass but any number of implemented interfaces.

Since Kotlin was designed with interoperability in mind, it works seamlessly with Java most of the time. The following section explores what you should keep in mind when calling Kotlin code from Java.

## USING KOTLIN CODE FROM JAVA

The other way around, Kotlin code can also be called from Java. The best way to understand how to access certain elements, such as extension functions or top-level declarations, is to explore how Kotlin translates to Java bytecode. An additional benefit of this is that you get more insights into the inner workings of Kotlin.

## Accessing Properties

Before diving into the details, remember that when I say “field,” I’m usually referring to Java (unless I’m explicitly referring to Kotlin’s backing fields). Contrarily, when talking about “properties,” I’m referring to Kotlin because they don’t directly exist in Java.

As you know, you don’t need to implement property getters and setters manually in Kotlin. They’re created automatically and can be accessed from Java as you would expect, as shown in [Listing 5.10](#).

### **Listing 5.10 Calling Getters and Setters**

[Click here to view code image](#)

```
// Kotlin

class KotlinClass {

    val fixed: String = "base.KotlinClass"

    var mutable: Boolean = false

}

// Java

KotlinClass kotlinClass = new KotlinClass();

String s = kotlinClass.getFixed();      // Uses getter

kotlinClass.setMutable(true);           // Uses setter

boolean b = kotlinClass.getMutable();  // Uses getter
```

Notice that Boolean getters also use the prefix `get` by default instead of `is` or `has`. However, if the property name itself starts with `is`, the property name is used as the getter name—and not just for Boolean expressions; this is irrespective of the property type. Thus, calling the Boolean property `isMutable` instead would result in the getter of the same name `isMutable`. Again, there’s currently no such mechanism for properties starting with `has` but you can

always define your own JVM name by annotating the getter or setter with `@JvmName`, as in [Listing 5.11](#).

**Listing 5.11 Custom Method Name Using `@JvmName`**

[Click here to view code image](#)

```
// Kotlin

class KotlinClass {

    var mutable: Boolean = false

    @JvmName("isMutable") get           // Specifies the JVM name for the getter
}

// Java

boolean b = kotlinClass.isMutable(); // Now getter is "isMutable"
```

#### Tip

One extremely useful feature in IntelliJ IDEA and Android Studio is the ability to see the compiled bytecode of your Kotlin code and then the decompiled Java code. For this, press **Ctrl+Shift+A** (**Cmd+Shift+A** on Mac) to invoke the action search, then type “Show Kotlin Bytecode” (or “skb”) and press Enter.

Inside the panel that opens, you see the generated Kotlin bytecode and you can click the *Decompile* button to see the decompiled Java code.<sup>11</sup> For the simple class from [Listing 5.10](#) without annotation, you’ll see this result (plus some metadata):

[11.](#) If the *Decompile* button doesn’t appear, make sure the Java Bytecode Decompiler plugin is enabled.

[Click here to view code image](#)

```
public final class KotlinClass {  
  
    @NotNull private final String fixed = "base.KotlinClass";  
  
    private boolean mutable;  
  
  
    @NotNull public final String getFixed() {  
  
        return this.fixed;  
  
    }  
  
    public final boolean getMutable() { return this.mutable; }  
  
    public final void setMutable(boolean var1) { this.mutable = var1; }  
  
}
```

Note that the class is final in Java because it’s non-open, that the nonprimitive String field has a `@NotNull` annotation to carry the nullability information to Java, and that the read-only property `fixed` is final as well. Lastly, note the amount of boilerplate that can be avoided due to Kotlin’s syntax (`val` and `var`), its choice of defaults (closed classes and non-nullables), and compiler-generated code (getters and setters).

## Exposing Properties as Fields

As you’ve learned, properties are compiled to a private field with getter and setter by default (this is also the case for file-level properties). However, you can use `@JvmField` to expose a property directly as a field in Java, meaning the field inherits the visibility of the Kotlin property and no getter or setter is generated. [Listing 5.12](#) demonstrates the difference.

[Listing 5.12 Exposing a Property as a Field using @JvmField](#)

[Click here to view code image](#)

```
// Kotlin

val prop = "Default: private field + getter/setter" /

@JvmField

val exposed = "Exposed as a field in Java" /

// Decompiled Java code (surrounding class omitted)

@NotNull private static final String prop = "Default:

@NotNull public static final String getProp() { return prop; }

@NotNull public static final String exposed = "Exposed as a field in Java";
```

As you can see, the property annotated with **@JvmField** is compiled to a public field that can be accessed directly in Java. This works the exact same way for properties inside a class; by default, they are also accessible via getters and setters but can be exposed as fields.

Note that there are several restrictions for using the **@JvmField** annotation, and it's a good exercise to think about why these restrictions exist.

- The annotated property must have a backing field. Otherwise, there would be no field to expose in the Java bytecode.
- The property cannot be private because that makes the annotation superfluous. For private properties, no getters or setters are generated anyway because there would be no value to them.
- The property cannot have the **const** modifier. Such a property becomes a static final field with the property's visibility anyway, so **@JvmField** would have no effect.
- It cannot have an **open** or **override** modifier. This way, field visibilities and existence of getters and setters is consistent between superclass and subclasses. Otherwise, you could accidentally hide the superclass field in Java with a field that has a more restrictive visibility. This can lead to unexpected behavior and is a bad practice.
- A **lateinit** property is always exposed so that it can be initialized from anywhere it's accessible, without assumptions about how it's

initialized. This is useful when an external framework initializes the property. `@JvmField` would be superfluous here as well.

- It cannot be a delegated property. Delegation only works with getters and setters that can be routed to the delegate's `getValue` and `setValue` methods, respectively.

## Using File-Level Declarations

In Kotlin, you can declare properties and functions on the file level. Java doesn't support this, so these are compiled to a class that contains the properties and functions. Let's say you have a Kotlin file `sampleName.kt` in a package `com.example` as in [Listing 5.13](#).

**Listing 5.13** File-Level Declarations in Kotlin

[Click here to view code image](#)

```
// sampleName.kt

package com.example

class FileLevelClass           // Generated by the compiler

object FileLevelObject         // Generated by the compiler

fun fileLevelFunction() {}     // Goes into the generated class

val fileLevelVariable = "Usable from Java" // Goes into the generated class
```

Note that classes and objects are also file-level declarations but are simply compiled to a corresponding Java class as you would expect (classes generated for objects implement the singleton pattern). The concepts of file-level properties and functions cannot be mapped to Java directly. So for the code in [Listing 5.13](#), the compiler generates not only the classes `com.example.FileLevelClass` and `com.example.FileLevelObject` but also a class `com.example.SampleNameKt` with a static field for the file-level property and a static method for the file-level function. The fields are private with a public static getter (and setter).

You can explore all classes in the decompiled Java code. The static members can be called from Java as usual after importing `SampleNameKt` or statically importing the members.

You can also give a shorter and more meaningful name to the generated class than the one based on the file name. This is done in Kotlin using `@file:JvmName("YOUR_NAME")` to annotate the entire file, as shown in [Listing 5.14](#).

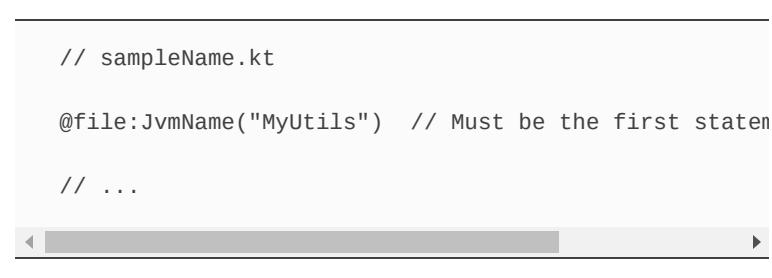
**Listing 5.14 Using `@JvmName` on File Level**

[Click here to view code image](#)

```
// sampleName.kt

@file:JvmName("MyUtils") // Must be the first statement

// ...
```



This way, the name of the generated class is `MyUtils`. You can even compile multiple Kotlin files into a single Java class by adding `@file:JvmMultifileClass` and `@file:JvmName` with the same name given to all of them. Be aware that using this increases the chance of name clashes and should only be used if all incorporated file-level declarations are closely related; this is questionable because they come from separate Kotlin files, so use this judiciously.

There are several other annotations that allow you to adjust some parameters of how your Kotlin code is mapped to Java. All of these consistently use the `@Jvm` prefix, and most will be discussed in this chapter.

## Calling Extensions

Extension functions and properties are typically declared on the file level and can then be called as a method on the generated class like other top-level declarations. In contrast to Kotlin, they cannot be called on the extension receiver type directly because there is no such feature in Java. [Listing 5.15](#) provides a brief example.

**Listing 5.15 Calling Top-Level Extensions**

[Click here to view code image](#)

```
// Kotlin

@file:JvmName("Notifications")

fun Context.toast(message: String) { // Top-level function
    Toast.makeText(this, message, Toast.LENGTH_SHORT)
}

// Within a Java Activity (Android)
Notifications.toast(this, "Quick info...");
```

Here, you define an extension function that facilitates showing toast messages on Android and use `@JvmName` to provide a more descriptive name for the generated class. From Kotlin, you could simply call this extension method as `toast("Quick info...")` from inside an activity because every activity is itself a context, and `Context` is the receiver type for this extension. With a static import, you can achieve a similar syntax in Java but the `Context` must still be passed in as the first argument.

Similarly, you can declare extensions inside a type declaration such as a class or object. These can be called from Java via an instance of that encompassing type. To highlight the difference, Listing 5.16 defines a similar extension but this time inside a `Notifier` class.

#### **Listing 5.16 Calling Class-Local Extensions**

[Click here to view code image](#)

```
// Kotlin

class Notifier {

    fun Context.longToast(message: String) { // Becomes a local extension
        Toast.makeText(this, message, Toast.LENGTH_LONG)
}
```

```
    }

}

// Calling from a Kotlin Android activity

with(Notifier()) { longToast("Important notification.

Notifier().apply { longToast("Important notification.

// Calling from a Java Android Activity

Notifier notifier = new Notifier();

notifier.longToast(this, "Important notification...")
```

To call a class-local extension from outside its containing class, you need an instance of the containing class, irrespective of whether you want to call it from Kotlin or Java. From Java, you can call it as a static method on the instance and must again pass in the context. From Kotlin, you have to get access to the class scope, which you can do via `with` or `apply`.

Recall that, inside the lambda expression, you can then write code as if you were inside the `Notifier` class. In particular, you can call `longToast`.

## Accessing Static Members

Several of Kotlin's language elements compile down to static fields or methods. These can be called directly on their containing class as usual.

## Static Fields

Although there's no `static` keyword in Kotlin, there are several language elements in Kotlin that will generate static fields in the Java bytecode and can thus be called as such from Java. Apart from the examples already seen (top-level declarations and extensions), static fields may be generated from:

- Properties in object declarations
- Properties in companion objects
- Constant properties

All these are shown in [Listing 5.17](#).

**Listing 5.17 Generating Static Fields on the JVM**

[Click here to view code image](#)

```
// Kotlin

const val CONSTANT = 360

object Cache { val obj = "Expensive object here..." }

class Car {

    companion object Factory { val defaultCar = Car() }

}

// Decompiled Java code (simplified)

public final class StaticFieldsKt {

    public static final int CONSTANT = 360;

}

public final class Cache { // Simplified

    private static final String obj = "Expensive object

    public final String getObj() { return obj; }

}

public final class Car { // Simplified

    private static final Car defaultCar = new Car();

    public static final class Factory {
```

```
public final Car getDefaultCar() { return Car.def
}
}
```

Properties from object declarations and companion objects produce private fields with getters and setters. For object declarations, these are inside the Java class that corresponds to the object, such as `Cache`. For companion objects, the field itself lives inside the containing class while the getter and setter stay inside the nested class. Static members are accessed as usual from Java, for instance as `Cache.INSTANCE.getObj()` or `Car.Factory.getDefaultCar()`.

The generated static fields are private by default (except when using `const`) but they can again be exposed using `@JvmField`. Alternately, you could use `lateinit` or `const`; as you learned, these also expose the field. However, it's not their main purpose but rather a side effect. Note that, using `@JvmField`, the static field gets the visibility of the property itself whereas using `lateinit`, it gets the visibility of the setter.

## Static Methods

While top-level functions become static methods in the Java bytecode by default, methods declared in named and companion objects are not static by default. You can change this using the `@JvmStatic` annotation as shown in Listing 5.18.

**Listing 5.18 Using `@JvmStatic` to Generate Static Methods**

[Click here to view code image](#)

```
// Kotlin
object Cache {
    @JvmStatic fun cache(key: String, obj: Any) { ... }
```

```
}

class Car {

    companion object Factory {

        @JvmStatic fun produceCar() { ... }

    }

}

// Inside a Java method

Cache.cache("supercar", new Car());      // Static men

Cache.INSTANCE.cache("car", new Car());  // Bad pract

Car.produceCar();                      // Static

Car.Factory.produceCar();              // Also poss

(new Car()).produceCar();              // Bad pract
```

In named objects (object declarations), using `@JvmStatic` allows you to call the method directly on the class, as you're used to for static methods. Also, it's unfortunately possible to call static methods on instances as well but you should avoid this because it can lead to confusing code and adds no value. Nonstatic methods of named objects can only be called on instances, as usual.

In companion objects, using `@JvmStatic` allows you to call the method directly on the enclosing class; here, `Car`. It can still be called explicitly on the nested companion object as well. Nonstatic methods can only be called as instance methods on the companion object; here, `Car.Factory`. Again, the static method on an instance should be avoided.

You can also use `@JvmStatic` on named and companion object *properties* to make their accessors static. But you

cannot use `@JvmStatic` outside of named and companion objects.

## Generating Method Overloads

Method overloading can often be avoided in Kotlin using default parameter values, saving many lines of code.

When compiling to Java, you can decide whether overloaded methods should be generated to enable optional parameters in Java to some extent, as in [Listing 5.19](#).

**Listing 5.19 Generating Overloads with `@JvmOverloads`**

[Click here to view code image](#)

```
// Kotlin

@JvmOverloads // Triggers generation of overloaded methods

fun <T> Array<T>.join(delimiter: String = ", ",
                        prefix: String = "",
                        suffix: String = ""): String {
    return this.joinToString(delimiter, prefix, suffix)
}

// Java

String[] languages = new String[] {"Kotlin", "Scala", "Java"};


// Without @JvmOverloads: you must pass in all parameters
ArrayUtils.join(languages, ";", "{}", "{}"); // Assumes languages is not null

// With @JvmOverloads: overloaded methods
ArrayUtils.join(languages); // Skip
ArrayUtils.join(languages, ";"); // Skip
ArrayUtils.join(languages, ";", "Array: "); // Skip
```

```
ArrayUtils.join(languages, " ", "[" , "]"); // Pass
```

Using `@JvmOverloads`, the compiler generates one additional overloaded method for each parameter with default value. This results in a series of methods where each has one fewer parameters, the optional one. Naturally, parameters without default value are never omitted. This increases flexibility when calling the method from Java but still doesn't allow as many combinations as Kotlin because the order of parameters is fixed and you cannot use named parameters. For instance, you cannot pass in a suffix without passing in a prefix.

Note that you can also use `@JvmOverloads` on constructors to generate overloads. Also, if *all* parameters of a constructor have a default value, a parameterless constructor is generated anyway, even without the annotation. This is done to support frameworks that rely on parameterless constructors.

## Using Sealed and Data Classes

Both sealed classes and data classes translate to normal classes in Java and can be used as such. Of course, Java does not treat sealed classes specially, so they cannot be used with Java's `switch` as they can with Kotlin's `when`. For instance, you must use `instanceof` checks to determine the specific type of an object of the parent sealed class, as shown in Listing 5.20.

**Listing 5.20** Working with Sealed Classes

[Click here to view code image](#)

```
// Kotlin

sealed class Component

data class Composite(val children: List<Component>) :

data class Leaf(val value: Int): Component()

// Java
```

```
Component comp = new Composite(asList(new Leaf(1), ne

    if (comp instanceof Composite) {           // Cannot use
        out.println("It's a Composite");        // No smart-casts
    } else if (comp instanceof Leaf) {          // No exhaust
        out.println("It's a Leaf");
    }
}
```

The sealed class becomes an abstract class, making it impossible to instantiate an object of it. Its child classes can be used as normal classes but carry no special semantics in Java because there is no concept for sealed classes.

Data classes can be used intuitively from Java, but there are two restrictions to keep in mind. First, Java does not support destructuring declarations so that the `componentN` functions are unnecessary. Second, there are no overloads generated for the `copy` method so that it has no benefit compared to using the constructor. This is because generating all possible overloads would introduce an exponential number of methods (with respect to the number of parameters). Also, generating only some overloads as is the case for `@JvmOverloads`, there is no guarantee that this would generate a useful subset of overloads. Hence, no overloads are generated at all for `copy`. All of this is illustrated in [Listing 5.21](#).

**Listing 5.21 Working with Data Classes**

[Click here to view code image](#)

```
// Kotlin

data class Person(val name: String = "", val alive: E

// Java

Person p1 = new Person("Peter", true);
```

```
Person p2 = new Person("Marie Curie", false);

Person p3 = p2.copy("Marie Curie", false); // No adv
```

```
String name = p1.getName(); // componentN() methods

out.println(p1); // Calls toString()

p2.equals(p3); // true
```

## Visibilities

The available visibilities don't map exactly between Kotlin and Java, plus you have top-level declarations in Kotlin. So let's explore how visibilities are mapped to Java. First, some visibilities can be mapped trivially.

- Private members remain private.
- Protected members remain protected.
- Public language elements remain public, whether top-level or member.

Other visibilities cannot be mapped directly but are rather compiled to the closest match:

- Private top-level declarations also remain private. However, to allow calls to them from the same Kotlin file (which may be a different class in Java), synthetic methods are generated on the JVM. Such methods cannot be called directly but are generated to forward calls that would not be possible otherwise.
- All of Kotlin's **internal** declarations become public because package-private would be too restrictive. Those declared inside a class go through name mangling to avoid accidental calls from Java. For instance, an internal method `C.foo` will appear as `c.foo$production_sources_for_module_yourmodulenam e()` in the bytecode, but you're not able to actually call them as such. You can use `@JvmName` to change the name in the Java bytecode if you want to be able to call internal members from Java.

This explains how each visibility maps to Java for both top-level declarations and members.

## Getting a KClass

`KClass` is Kotlin's representation of classes and provides reflection capabilities. In case you have a Kotlin function accepting a `KClass` as a parameter and need to call it from Java, you can use the predefined class `kotlin.jvm.JvmClassMappingKt` as in Listing 5.22. From Kotlin, you can access both `KClass` and `Class` more easily.

**Listing 5.22 Getting KClass and Class References**

[Click here to view code image](#)

```
// Java

import kotlin.jvm.JvmClassMappingKt;

import kotlin.reflect.KClass;

KClass<A> clazz = JvmClassMappingKt.getKotlinClass(A.

// Kotlin

import kotlin.reflect.KClass

private val kclass: KClass<A> = A::class

private val jclass: Class<A> = A::class.java
```

## Handling Signature Clashes

With Kotlin, you may declare methods that have the same JVM signature. This mostly happens due to type erasure of generics types, meaning that type parameter information is not available at runtime on the JVM. Thus, at runtime, Kotlin (like Java) only knows that `List<A>` and `List<B>` have type `List`. Listing 5.23 demonstrates the situation.

**Listing 5.23 JVM Name Clash**

[Click here to view code image](#)

```
fun List<Customer>.validate() {}      // JVM signature:  
fun List<CreditCard>.validate() {}  // JVM signature:
```

Here, you wouldn't be able to call these methods from Java because there's no way to differentiate between the two at runtime. In other words, they end up with the same bytecode signature. As you may expect, this is easily resolved using `@JvmName`, as in [Listing 5.24](#).

[Listing 5.24 Resolving JVM Name Clashes](#)

[Click here to view code image](#)

```
fun List<Customer>.validate() { ... }  
  
@JvmName("validateCC") // Resolves the name clash  
  
fun List<CreditCard>.validate() { ... }  
  
// Both can be called as validate() from Kotlin (because of the @JvmName annotation)  
  
val customers = listOf(Customer())  
  
val ccs      = listOf(CreditCard())  
  
customers.validate()  
  
ccs.validate()
```

From Java, the methods are available as two static methods, `FileNameKt.validate` and `FileNameKt.validateCC`. From Kotlin, you can access both as `validate` because the compiler dispatches that to the appropriate method internally (at compile time, all necessary type information for this is available).

## Using Inline Functions

You can call inline functions from Java just like any other function, but of course they are not actually inlined—there is no such feature in Java. Listing 5.25 demonstrates inlining *when used from Kotlin*. Be aware that inline functions with reified type parameters are not callable from Java at all because it doesn’t support inlining, and reification without inlining doesn’t work. Thus, you cannot use reified type parameters in methods that should be usable from Java.

**Listing 5.25 Calling Inline Functions (from Kotlin)**

[Click here to view code image](#)

```
// Kotlin

inline fun require(predicate: Boolean, message: () ->
    if (!predicate) println(message())

}

fun main(args: Array<String>) { // Listing uses main
    require(someCondition()) { "someCondition must be true" }
}

// Decompiled Java Code (of main function)

public static final void main(@NotNull String[] args)
    Intrinsics.checkParameterIsNotNull(args, "args");
    boolean predicate$iv = someCondition();
    if (!predicate$iv) {
        String var2 = "someCondition must be true";
        System.out.println(var2);
    }
}
```

As you can see, the **if** statement and its body are inlined into the **main** method and there is no actual call to the **require** method anymore. Without the **inline** keyword, or when calling the method from Java, there would be a call to **require** instead.

## Exception Handling

Because there are no checked exceptions in Kotlin, you can call all Kotlin methods from Java without handling exceptions. This is because exceptions are then not declared in the bytecode (there are no **throws** clauses).

To allow exception handling in Java, you can use the **@Throws** annotation as shown in Listing 5.26.

**Listing 5.26 Generating Throws Clauses**

[Click here to view code image](#)

```
// Kotlin

import java.io.*

@Throws(FileNotFoundException::class) // Generates throws clause
fun readInput() = File("input.csv").readText()

// Java

import java.io.FileNotFoundException;
// ...

try { // Must handle exception
    CsvUtils.readInput(); // Assumes @file:JvmName("CsvUtils")
} catch (FileNotFoundException e) {
    // Handle non-existing file...
}
```

Without the `@Throws` annotation, you could call the `readInput` method from both Kotlin and Java without handling exceptions. With the annotation, you're *free to* handle exceptions when calling it from Kotlin, and you *must* handle all checked exceptions when calling it from Java.

Looking at the decompiled Java code, you can see that all the annotation does is to add a `throws FileNotFoundException` to the method signature, as demonstrated in [Listing 5.27](#).

**Listing 5.27 Difference in Decompiled Java Code**

[Click here to view code image](#)

```
// Without @Throws

public static final String readInput() { ... }

// With @Throws

public static final String readInput() throws FileNotFoundException
```

## Using Variant Types

Regarding variant types, there's a disparity between Kotlin and Java because Java only has use-site variance whereas Kotlin also has declaration-site variance. Thus, Kotlin's declaration-site variance must be mapped to use-site variance. How is this done? Whenever an out-projected type appears as a parameter or variable type, the wildcard type `<? extends T>` is automatically generated.

Conversely, for in-projected types that appear as a parameter, `<? super T>` is generated. This lets you use the type's variance in Java. Consider [Listing 5.28](#) as an example.

**Listing 5.28 Mapping Declaration-Site Variance to Use Site**

[Click here to view code image](#)

```
// Kotlin

class Stack<out E>(vararg items: E) { ... }
```

```
fun consumeStack(stack: Stack<Number>) { ... }

// Java: you can use the covariance of Stack
consumeStack(new Stack<Number>(4, 8, 15, 16, 23, 42))
consumeStack(new Stack<Integer>(4, 8, 15, 16, 23, 42)
```



The Java signature for the `consumeStack` method is **void** `consumeStack(Stack<? extends Number> stack)` so that you can call it with a `Stack<Number>` as well as a `Stack<Integer>`. You cannot see this signature in the decompiled Java code due to type erasure, but looking at the Kotlin bytecode directly, you can find a comment stating `declaration: void consumeStack(Stack<? extends java.lang.Number>)`. So you're still able to use the bytecode tool to see what's happening internally.

The generation of wildcard types only happens when in- or out-projected types are used as parameters. For return types, no such wildcards are generated because this would go against Java coding standards. Still, you can intercept the generation process here as well. To generate wildcards where they normally wouldn't, you can use `@JvmWildcard` as in [Listing 5.29](#). To suppress wildcards, you can use `@JvmSuppressWildcards`, also shown in [Listing 5.29](#).

#### [Listing 5.29 Adjusting Wildcard Generation](#)

[Click here to view code image](#)

```
// Kotlin

fun consumeStack(stack: Stack<@JvmSuppressWildcards Number>) { ... }

// Normally no wildcards are generated for return types
fun produceStack(): Stack<@JvmWildcard Number> {
    return Stack(4, 8, 15, 16, 23, 42)
```

```
}

// Java

consumeStack(new Stack<Number>(4, 8, 15, 16, 23, 42))

consumeStack(new Stack<Integer>(4, 8, 15, 16, 23, 42)

Stack<Number> stack = produceStack();           // Er

Stack<? extends Number> stack = produceStack(); // ]
```

Note that you can also annotate methods or even whole classes with these to control wildcard generation. However, most of the time you'll be able to call Kotlin methods from Java just fine without intercepting the generation process because declaration-site variance is mapped sensibly by default.

**Tip**

`@JvmSuppressWildcards` can be the solution to errors that may be hard to debug. For instance, a Java class that implements an interface and should override one of its methods must match the method's signature exactly. If the interface doesn't include wildcards, you won't be able to override it without using `@JvmSuppressWildcards`.

A similar problem may occur if you're using a framework that forces you to provide an implementation of a specific interface, let's say `Adapter<T>`. Assume you want to implement `Adapter<Pair<String, String>>`. Because `Pair` is covariant, you'll actually provide an implementation for `Adapter<Pair<? extends String, ? extends String>>`. So keep in mind how variant types map to Java and, more important, remember to investigate the byte-code and decompiled Java code.

## The Nothing Type

Lastly, there's no counterpart to Kotlin's `Nothing` type in Java because even `java.lang.Void` accepts `null` as a value. Because it's still the closest representation of `Nothing` that's available in Java, `Nothing` return types and parameters are mapped to `Void`. As shown in [Listing 5.30](#), this doesn't result in the exact same behavior.

[Listing 5.30 Using the Nothing Type](#)

[Click here to view code image](#)

```
// Kotlin

fun fail(message: String): Nothing {           // Indi
    throw AssertionError(message)
}

fun takeNothing(perpetualMotion: Nothing) {} // Impc

// Java

NothingKt.takeNothing(null); // Possible in Java (but
NothingKt.fail("Cannot pass null to non-null variable")
System.out.println("Never reached but Java doesn't kr
```

You can actually call `takeNothing` from Java, an action not possible from Kotlin. However, you'll at least receive a warning because the signature is **void**

`takeNothing(@NotNull Void
perpetualMotion)`. Since **null** is the only valid value for `Void`, that's really the closest representation of `Nothing` currently available in Java. Similarly, the return type of `fail` becomes `Void` so that Java cannot infer unreachable code like Kotlin.

Lastly, using `Nothing` as a generic type argument generates a raw type in Java to at least provoke unchecked call warnings. For instance, `List<Nothing>` becomes a raw `List` in Java.

# BEST PRACTICES FOR INTEROP

To conclude this chapter, I want to summarize the major points by distilling best practices that follow from the concepts and pitfalls discussed. Following these practices helps make interoperability between Kotlin and Java even more seamless.

## Writing Kotlin-Friendly Java Code

Calling Java code from Kotlin works seamlessly in nearly all cases. Still, there are some practices you can follow to avoid special cases of conflict and aid interoperability even further.

- Add appropriate nullability annotations to all nonprimitive return types, parameters, and fields to avoid ambiguous platform types.
- Use the accessor prefixes `get`, `set`, and `is` so that they are automatically accessible as properties from Kotlin.
- Move lambda parameters to the end of signatures to allow for the simplified syntax where you can pull the lambda out of the parentheses.
- Do not define methods that can be used as operators from Kotlin unless they indeed make sense for the corresponding operator.
- Do not use identifiers in Java that are hard keywords in Kotlin<sup>12</sup> to avoid escaping.

12. <https://kotlinlang.org/docs/reference/keyword-reference.html#hard-keywords>

All of these make interoperability even more seamless and can improve the quality of your Java code as well, such as nullability annotations.

## Writing Java-Friendly Kotlin Code

As you've learned, calling Kotlin code from Java entails several things to keep in mind that usually become clear when looking at the bytecode or decompiled Java code.

There are also more practices you can follow to aid interoperability.

- Provide expressive names for all files with top-level declarations to avoid the default `FileNameKt` naming of the class (which also leaks the language). Use `@file:JvmName("ExpressiveName")` for this and consider using `@file:JvmMultifileClass` if other closely related top-level declarations exist in a different Kotlin file.

- If possible, avoid `Unit` return types in lambda parameters of methods that are supposed to be called from Java because this requires an explicit `return Unit.INSTANCE` in Java. Future versions of Kotlin may provide ways to avoid this problem.
- Add a `@Throws` annotation for each checked exception in all methods if you want to be able to handle the exception in Java. This basically lets you set your own balance between checked and unchecked exceptions. For this, also consider possible exceptions thrown by methods called inside that method.
- Do not expose read-only Kotlin collections directly to Java, for instance as return types, because they are mutable there. Instead, make defensive copies before exposing them or expose only an unmodifiable wrapper for the Kotlin collections, such as `Collections.unmodifiableList(kotlinList)`.
- Use `@JvmStatic` on companion object methods and properties to be able to call them directly on the containing class in Java. You can do the same for methods and properties of named objects (object declarations) as well.
- Use `@JvmField` for effectively constant properties in companion objects that are not `const` (because `const` only works for primitive types and `String`). Without `@JvmField`, they are accessible from Java as `MyClass.Companion.getEFFECTIVE_CONSTANT` with an ugly getter name. With it, they are accessible as `MyClass.EFFECTIVE_CONSTANT`, following Java's coding conventions.
- Use `@JvmName` on methods where the idiomatic naming differs between Kotlin and Java. This is mostly the case for extension methods and infix methods due to the different ways in which they are called.
- Use `@JvmOverloads` on any method that has default parameter values and is supposed to be called from Java. Make sure all generated overloads make sense, and reorder the parameters if not. Move parameters with default values to the end.

These best practices help you write idiomatic code in both Kotlin and Java even while mixing the languages. Most of these use Kotlin's JVM annotations to fine-tune the way in which Kotlin compiles to Java bytecode.

## SUMMARY

In mixed-language projects and during migration, you may use Kotlin and Java simultaneously in a project. Fortunately, most language concepts map naturally between the languages. For the remainder, it is important to understand how Kotlin compiles to Java bytecode and how you can write code that aids interoperability. On the Java side, this mainly includes adding nullability information and enabling shorter syntax in Kotlin, for example, by moving lambda parameters to the end. On the Kotlin side, it primarily encompasses using JVM annotations to adjust the compilation process, for instance, to generate static modifiers, descriptive names, or overloaded methods. This way, both languages can be used together without a large amount of overhead on either side.

## Concurrency in Kotlin

*“Juggling is an illusion. . . . In reality, the balls are being independently caught and thrown in rapid succession. . . . It is actually task switching.”*

Gary Keller

Since the emergence of multicore processors, concurrency has become an omnipresent topic. However, concurrency is hard and comes with a list of hard-to-debug problems. This chapter briefly introduces the basic concepts surrounding concurrency, introduces Kotlin’s coroutines as a unique approach to concurrency, and shows how to represent a variety of asynchrony and concurrency patterns with coroutines.

## CONCURRENCY

In today’s world of REST<sup>1</sup> APIs and microservices, concurrency and asynchrony play a central role for developers. On Android, apps may not actually perform many computations but rather spend most of their time waiting—whether it be for weather data fetched from a server, user data retrieved from a database, or an email to arrive.

1. Acronym stands for “representational state transfer.”

For this chapter, I want to provide a working definition for several terms that are often confused with each other: concurrency, parallelism, multitasking, multithreading, and asynchrony. The distinction is not essential to follow the practical part on Kotlin’s coroutines and apply them in your

code. However, a solid foundation helps to put things into context and to discuss the topic.

- First, *concurrency* itself only means that different parts of a program can run out of order without it affecting the end result. This does not imply parallel execution; you may have a concurrent program running on a single-core machine with tasks that overlap in time because processor time is shared between processes (and threads). However, concurrency by this definition is a necessary condition for a correct parallel program—a nonconcurrent program run in parallel produces unpredictable results.
- The terms *parallelism* and *multitasking* are used synonymously in this book and refer to the actual parallel execution of multiple computations at the same time. In contrast to concurrency, this requires a multicore machine (see following note). The fact that parallel execution is also called *concurrent execution* may be the source of some confusion.
- *Multithreading* is one particular way to achieve multitasking (parallelism), namely via threads. A thread is intuitively a sequence of instructions managed by a scheduler. Multiple threads can live inside the same process and share its address space and memory. Because these threads can execute independently on multiple processors or cores, they enable parallelism.
- Lastly, *asynchrony* allows operations outside of the main program flow. Intuitively, this is usually a background task that is issued and executed without forcing the caller to wait for its completion—such as fetching resources from a network. This is in contrast to synchronous behavior, meaning that the caller waits for the completion of the previous operation before performing the next operation. This may make sense if the synchronous task is actually performing computations on the CPU but not if it is waiting for a third party, such as a REST API, to answer.

An important aspect of asynchrony is that you don't want to block the main thread while waiting for a result—in fact, the terms *asynchronous* and *nonblocking* are used synonymously in this book. On Android, blocking the main thread freezes the UI and after a while causes an exception, resulting in a bad user experience. To achieve asynchrony, you would typically use some implementation of *futures* (also called *promises*) such as `CompletableFuture` from Java 8. In this chapter, you will discover coroutines as a more lightweight solution that also allows writing intuitive code.

**Note**

For the sake of completeness, you can differentiate between parallelism on the bit level, instruction level, data level, and functional level. On the bit level, for instance, modern processors can process a word size of 64 bits in one cycle, which is also a form of parallelism. Bitlevel parallelism is transparent for developers and does not depend on concurrency.

In this book, parallelism refers to the functional level, meaning that the same or different computations are executed on the same or different data *simultaneously*. This is the kind of parallelism that requires concurrency to guarantee a correct end result.

To connect these concepts, let us consider how they relate. Assuming that parallelism always refers to functional-level parallelism, it relies on concurrency. Without concurrency, parallelism leads to unpredictable results and randomly occurring bugs (so-called *heisenbugs*). Asynchrony does not necessarily imply parallelism, at least not on your machine (if you take into account the remote machine, you may think of it as parallelism). So while parallelism requires a multicore machine, you may make an asynchronous call on a single-core machine using a single thread.

As real-life examples, you can imagine parallelism as a group of people, each performing their own task with a tennis ball: bouncing it on the floor, throwing it in the air, and so on. Concurrency could be a single person juggling four tennis balls, focusing on one ball at a time (time-sharing). Asynchrony could be a person doing the dishes while (nonblockingly) waiting for his or her laundry to finish.

Here's a brief overview of all these terms as a quick reference:

- **Concurrency:** Ability of units of code to execute out of order (whether parallel or time-sharing) and still get the correct result
- **Parallelism:** Actual simultaneous execution of multiple units of code
- **Multitasking:** Same as parallelism
- **Multithreading:** Parallelism achieved using threads
- **Asynchrony:** Nonblocking operations outside of main thread

With this, let us explore common challenges introduced by concurrency and current state-of-the-art solutions, and then dive into Kotlin's coroutines for parallel programming.

## Challenges

As you can imagine, not all programs are concurrent. More often than not, order of execution does matter. This is especially true if the concurrent units, such as threads, share *mutable state*. For instance, two threads may both read the value of the same variable, modify it, and then write it back—in that case, the second write will override the first.

A useful example that portrays many of the common challenges is that of the *dining philosophers*. Imagine five philosophers sitting around a round table, with five chopsticks distributed between them. Each philosopher has a bowl of rice, and a chopstick to its left and to its right. Philosophers are either in the state “eating” or in the state “thinking.” To eat, they must have both their left and right chopstick. The philosophers are not allowed to communicate. [Figure 6.1](#) illustrates this situation.

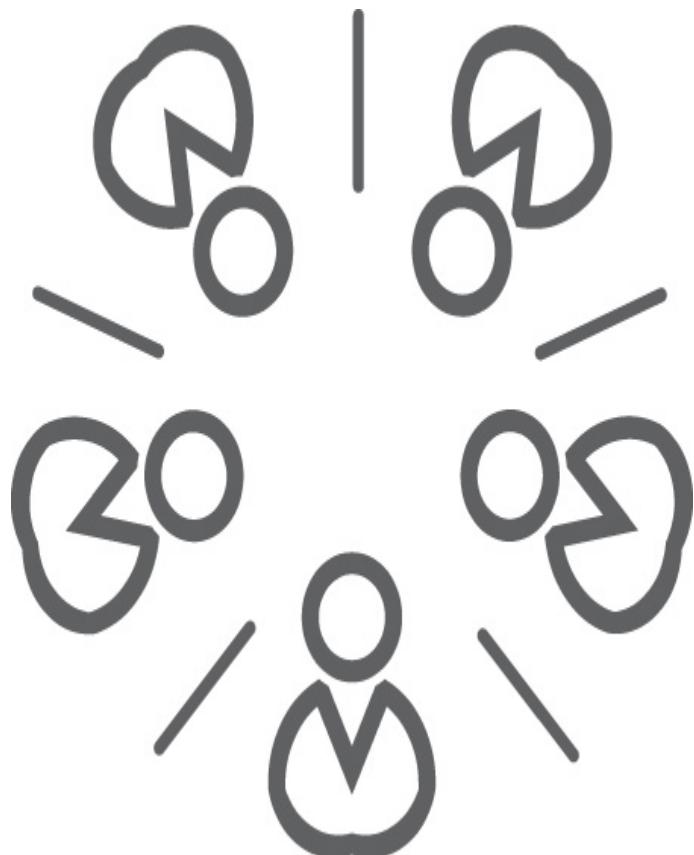


Figure 6.1 Dining philosophers:  $n$  philosophers share  $n$  chopsticks and cannot communicate

In this scenario, the chopsticks are the shared resources; for each chopstick, there are two philosophers competing for it. You can imagine the philosophers as parallel threads. To see how this parallelism can cause issues, let's say each philosopher first takes up the chopstick to his or her left and then the one to his or her right, eats, and then put them back on the table. If all philosophers start this process at the same time, they will all pick up the chopstick to the left and wait for the other one indefinitely.

This type of problem is called a *deadlock* and arises whenever no parallel unit can proceed due to circular dependencies. In the example, each philosopher is waiting for the chopstick to his or her right. For a deadlock to happen, multiple conditions (called Coffman conditions) must come into play, one of which is *mutual exclusion*. However, mutual exclusion is often necessary to prevent other synchronization problems. It means a resource cannot be used by multiple threads at the same time, just like a chopstick cannot be used by multiple philosophers at the same time.

As a next problem, you do not want two adjacent philosophers to pick up the chopstick they compete for at the same time. This would be called *race condition* and arises from threads accessing the same resource simultaneously. It is prevented using mutually exclusive access to the resource (chopstick) so that only one thread (philosopher) can access it at a time, commonly achieved via *locks* (a philosopher grabbing a chopstick locks access to it for others).

More generally, race conditions are bugs that arise because the results of parallel tasks depend on their execution order, thus not supporting concurrency by definition. Locks are one way to achieve concurrency, that is, to make code independent of the order of execution. For performance reasons, you want to wait for locks (wait for access to a chopstick) in a nonblocking way so that CPU time can be used for other computations until the lock is available. Kotlin offers locks in its standard library via extension functions on the class `java.util.concurrent.locks.Lock` and offers

nonblocking lock and mutex implementations in the coroutines library.

Apart from these classical synchronization issues that come along with concurrency and parallelism, there are also timing issues to consider on Android. For instance, an asynchronous call may only finish after the activity that issued it has already been destroyed and recreated. Consequently, a reference to the destroyed activity may be kept, causing a memory leak that prevents the activity from being garbage-collected. Also, trying to access the destroyed activity will crash the app. This particular problem can be solved using lifecycle-aware components introduced as part of Android's Architecture Components in 2017. That way, lifecycle-aware asynchronous calls are cancelled automatically if their caller is destroyed.

Another very different issue is the fact that threads are expensive objects, and so developers learn to create them judiciously, closely watch existing threads, and reuse previously created ones. As you will discover in this chapter, Kotlin's coroutines solve this problem because they are orders of magnitude less expensive to spin up.

### **State-of-the-Art Solutions**

In a world of multicore processors, multitasking is the primary means to improve the performance of parallelizable algorithms. The most common way to do this is multithreading, for example, to run a map task on a large collection in multiple threads that operate on disjoint parts of the collection.

In general, multiple threads are allowed per process, so they share the process' address space and memory (see [Figure 6.2](#) for a hierarchical breakdown). This allows for efficient communication between them but also introduces the synchronization issues discussed above because of the shared state. Also, the fact that threads may interact with each other leads to a huge number of possible execution paths. These are some of the reasons why concurrency is considered to be hard. Debugging becomes more complicated and the program becomes harder to reason about. Other models of concurrent

computation, such as the *actor model*, reduce such problems by removing shared mutable state. As you will learn, Kotlin’s coroutines allow both thread-like behavior and actor-like behavior, among others.

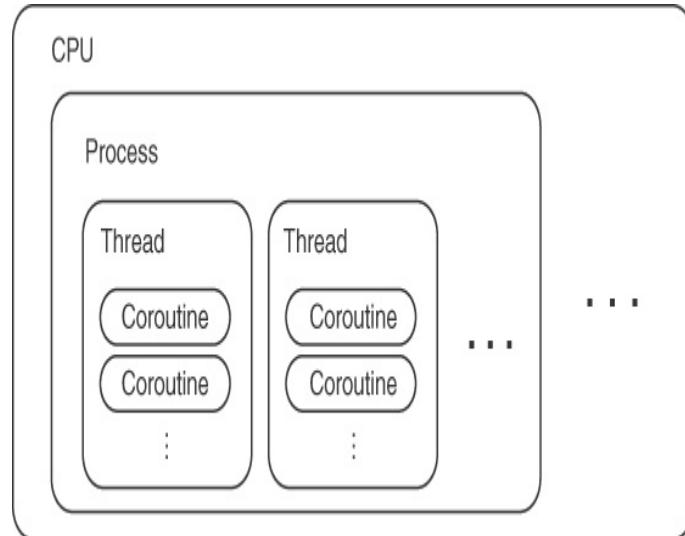


Figure 6.2 A CPU has multiple processes, each of which can contain multiple threads that in turn can each run a large number of coroutines

For asynchronous calls, one common approach is to use *callbacks*. A callback is a function `c` that is passed to another function `f` and is executed once the result of `f` is available (or another event occurs). One example of this is fetching weather data from an API and letting the callback function update the UI with the new weather report. However, you usually want to do another operation after that, and then another, and so forth. The problem with callbacks then becomes that, with each nested operation, the indentation level in your code increases, leading to highly nested and unreadable code (see [Listing 6.1](#)). Additional callbacks for error handling only add to the complexity. You may know this phenomenon under the term *callback hell*. But not only that—imagine doing less trivial control flow with callbacks, such as loops or **try-catch** blocks. The code no longer highlights the logical workflow but how the callbacks are wired together. With Kotlin’s coroutines, you are able to write sequential-style code without any syntactic overhead, as you will see later.

#### [Listing 6.1 Asynchrony with Callbacks](#)

[Click here to view code image](#)

```
// Simplified, no error handling

fetchUser(userId) { user ->

    fetchLocation(user) { location ->

        fetchWeather(location) { weatherData ->

            updateUi(weatherData)

        }

    }

}


```

This style of programming with callbacks is also known as *Continuation Passing Style (CPS)*. Intuitively, a *continuation* is simply a callback. In CPS, continuations are always passed explicitly, and the results of previous computations can become arguments of the next one. One consequence of this is that expressions must be turned inside out because continuation order must reflect evaluation order. For instance, you can't call `fetchWeather(fetchLocation(...))` in [Listing 6.1](#) because what is computed first must appear first.

Interestingly, CPS is considered hard to read and write and therefore not commonly used, but it is standard in callback-based code. CPS is in contrast to *direct style* in which continuations are implicit—the continuation of one line of code is the lines of code following it. This is what you're used to from sequential code. As mentioned, Kotlin preserves the possibility to write code in sequential style even when writing asynchronous code.

Another way to use asynchrony are *futures*, which may appear with different names such as `CompletableFuture`, `Call`, `Deferred`, or `Promise`, but eventually all these represent a unit of asynchronous computation that is supposed to return a value (or an exception) later. Futures prevent the nesting issue of callbacks and typically provide combinators to schedule additional computation to be performed on the result, such as

`thenApply`, `thenAccept`, or `thenCombine`, as shown in Listing 6.2.

**Listing 6.2 Asynchrony with Futures**

[Click here to view code image](#)

```
// Simplified, no error handling

fetchUser(userId)

    .thenCompose { user -> fetchLocation(user) }

    .thenCompose { location -> fetchWeather(location)

        .thenAccept { weatherData -> updateUi(weatherData)
    }
}

```

These combinators allow a fluent programming style and pass along exceptions, but you have to memorize lots of combinators; also, they're not consistent between implementations, adding to the cognitive overhead. So even with the improvements of futures compared to callbacks, Kotlin coroutines still improve readability because code looks natural, without the need for such combinators, as shown later.

Another way to write asynchronous code is to use `async`-`await` language constructs as in C#, TypeScript, or Python. Here, `async` and `await` are language *keywords*. An `async` function is one that performs an asynchronous operation, and `await` is used to explicitly wait for the completion of such an operation, as in Listing 6.3. You could say that `await` transitions back into the synchronous world, where you wait for an operation to finish before proceeding to the next. Kotlin's concept of *suspending functions* generalizes the `async`-`await` pattern without introducing keywords for it. Later in this chapter, the rationale for not simply adopting the well-known `async`-`await` pattern will become clearer.

**Listing 6.3 Asynchrony with `async`-`await` Pattern (C#)**

[Click here to view code image](#)

```
// C#: Notice the return type Task<Location> instead

async Task<Location> FetchLocation(User user) { ... }
```

```
var location = await FetchLocation(user) // Need await
```

Regarding synchronization issues, state-of-the-art solutions include concurrent data structures, locks, and avoiding shared mutable state. First, concurrent data structures allow concurrent access without disrupting the integrity of the stored data. They facilitate parallel programming because they can be used for shared mutable state. Second, locks restrict access to a critical section to one concurrent unit to avoid race conditions. More generally, semaphores restrict access to a given number of concurrent units. Third, avoiding shared mutable state is a good practice and brings forth approaches such as the previously mentioned actor model in which concurrent units communicate exclusively via message passing and thus have no shared state at all. These concepts all apply to concurrency with Kotlin as well and go more into the direction of coding practices than language design.

## KOTLIN COROUTINES

Having discussed the virtues, challenges, and approaches of concurrency, this section introduces Kotlin's prime feature for concurrency and asynchrony: *coroutines*.

Conceptually, you can imagine coroutines as very lightweight threads. However, they are not threads; in fact, *millions* of coroutines may run in a *single* thread. Also, a coroutine is not bound to one specific thread. For example, a coroutine may start on the main thread, suspend, and resume on a background thread later. You can even define *how* coroutines are dispatched, which is useful to distribute operations between the UI thread and background threads on Android and other UI frameworks.

Coroutines are similar to threads in that they have a similar lifecycle: They get created, get started, and can get suspended and resumed. Like threads, you can use coroutines to implement parallelism, you can join coroutines to wait for their completion, and you can cancel coroutines explicitly. However, compared to threads, coroutines take up much fewer

resources so there is no need to keep close track of them as you do with threads; you can usually create a new coroutine for every asynchronous call without worrying about overhead. Another difference is that threads use *preemptive multitasking*, meaning that processor time is shared between threads by a scheduler. You can imagine threads as being selfish; they do not think about other threads and use all the processor time they can get for themselves. Coroutines, on the other hand, use *cooperative multitasking* in which each coroutine itself yields control to other coroutines. There is no separate mediator involved to support fairness.

**Note**

Coroutines were experimental until Kotlin 1.3. At the time of writing, Kotlin 1.3 was not released in its stable version yet. Even though some details of coroutines as presented here may have changed, the concepts discussed remain and all code examples should still work as described here or with small changes.

Any such changes and coroutine updates are covered on the companion website at [kotlinandroidbook.com](http://kotlinandroidbook.com), along with adapted code listings where necessary.

## Setup

Before diving deeper into coroutines and exploring code examples, you should set up coroutines in a Kotlin project to follow along: Add the core dependency to the `dependencies` section (not the one inside of `buildscript`) of your app's `build.gradle` file:

[Click here to view code image](#)

```
implementation "org.jetbrains.kotlinx:kotlinx-corouti
```

For Android, also add the Android-specific dependency:

[Click here to view code image](#)

```
implementation "org.jetbrains.kotlinx:kotlinx-corouti
```

You can replace the version number with the latest one.<sup>2</sup> With this, you should now be able to use coroutines in your project.

If you face issues with your setup, you can find a more detailed setup guide on GitHub.<sup>3</sup>

2. Find the latest version here:

<https://github.com/Kotlin/kotlinx.coroutines/releases>

3. <https://github.com/Kotlin/kotlinx.coroutines/blob/master/README.md#using-in-your-projects>

At the time of writing, coroutines don't play well with `.kts` scripts, so the code listings in this chapter use main functions.

## Concepts

With coroutines, the Kotlin team intended to generalize the concepts of `async-await` and generators (asynchronous iterators). They also wanted coroutines to be independent from any specific implementation of futures, and they wanted to be able to wrap asynchronous APIs such as Java NIO (nonblocking I/O). All of this is possible with coroutines. On top of that, coroutines allow you to write natural function signatures without callbacks or `Future<T>` return values, let you write sequential-style code without callbacks or combinators, and make concurrency explicit at the call site. In fact, it is encouraged to always make concurrency and asynchrony explicit in idiomatic Kotlin code. You will see how this is done in this chapter.

With coroutines, the example portrayed in [Listings 6.1](#) and [6.2](#) can be written as in [Listing 6.4](#). As you can see, coroutines allow you to write asynchronous code in the sequential style that you are used to and that represents the logical workflow of the program. Also, you could add conditions, loops, **try-catch** blocks, and other control flow constructs naturally to the code in [Listing 6.4](#)—which is not trivial with callbacks.

### [Listing 6.4 Asynchrony with Coroutines](#)

[Click here to view code image](#)

---

```
val user = fetchUser(userId) // Asynchronous call ir  
  
val location = fetchLocation(user)  
  
val weatherData = fetchWeather(location)
```

```
updateUi(weatherData)
```

One thing that may be confusing without IDE support is that you would expect code that looks like this to be blocking. For instance, in the first line, you would expect the current thread to be blocked until the user info has been received, preventing the thread from performing other tasks in the meantime.

However, these functions are *suspending functions*, meaning they can suspend execution in a nonblocking way, and Android Studio highlights those in the left margin of the code editor so that you're aware of suspension points and the fact that asynchrony is used.

## Suspending Functions

Following the intuition provided by Roman Elizarov from the Kotlin team, *suspending functions are functions with an additional superpower*: they may suspend their execution in a nonblocking way at well-defined points.

This means that the coroutine that is currently running the suspending function is detached from the thread on which it was operating, and then waits until it is resumed (potentially on another thread). In the meantime, the thread is free to perform other tasks. Listing 6.5 defines `fetchUser` as a suspending function.

### **Listing 6.5 Declaring a Suspending Function**

[Click here to view code image](#)

```
suspend fun fetchUser(): User { ... }
```

The **suspend** modifier transforms a normal function into a suspending function. In principle, nothing more is required. As you can see, the function signature itself is completely natural, without additional parameters or a future wrapping the return value. Compare this to the function signature for callback-based or future-based asynchrony in Listing 6.6. Callbacks introduce an additional parameter, whereas futures wrap the desired return value.

#### Listing 6.6 Function Signatures for Callbacks and Futures

[Click here to view code image](#)

```
// Callback-based

fun fetchUser(callback: (User) -> Unit) { ... }

// Future-based

fun fetchUser(): Future<User> { ... }
```

With coroutines, both the actual function signature and the calling code look natural. At this point, it's important to understand that the **suspend** modifier is redundant unless the function actually contains a *suspension point*. Suspension points are the mentioned “well-defined points” at which a suspending function may suspend. Without a suspension point, the function is still effectively a blocking function because there is no way for it to suspend execution.

So how can you introduce such a suspension point? By calling another suspending function. All suspending functions from the Kotlin standard library work for this because, in the end, all of these end up calling the low-level function `suspendCoroutine`. You can also call `suspendCoroutine` directly, but it is typically used to implement higher-level APIs for libraries. Assuming all fetch functions are declared as suspending, Listing 6.7 denotes the suspension points similar to how Android Studio does.

#### Listing 6.7 Asynchrony with Coroutines

[Click here to view code image](#)

```
suspend fun updateWeather() {

    val user = fetchUser(userId)           // Suspen
    val location = fetchLocation(user)     // Suspen
    val weatherData = fetchWeather(location) // Suspen
    updateUi(weatherData)
}
```

```
}
```

The `updateWeather` function may yield control in a nonblocking way at each of the suspension points—and only at these points. This allows waiting until the result of each call is available without blocking the CPU. The three calls are executed sequentially by default so that, for instance, line three is only executed once the second line completes.

This contrasts to the default behavior of `async` functions in languages like C#, which work asynchronously by default—to make them behave sequentially, developers have to call `await` on them. The Kotlin team decided to make sequential behavior the default for two reasons. First, they expect it to be the most common use case and align with what developers are used to. Second, they encourage developers to always make asynchrony explicit. For example, a better name for a future-based `fetchUser` function would be `fetchUserAsync` or `asyncFetchUser`, so that asynchrony is explicit at the call site (see [Listing 6.8](#)). Because suspending functions are sequential by default, no such prefix or suffix is necessary there.

#### [Listing 6.8 Naming Asynchronous Functions](#)

[Click here to view code image](#)

```
fun fetchUserAsync(): Future<User> { ... }

// Call-site

val user = fetchUserAsync() // Asynchrony is explicit
```

Excited about suspending functions, you may try to call a suspending function from within `main` or from `onCreate` in your Android app but find that this is not possible. If you think about it, this makes sense: Your suspending function has the superpower to suspend execution without blocking the thread, but `main` and `onCreate` are just normal functions and don't

possess that superpower. So naturally, they cannot call a suspending function. Suspending functions can only be called from another suspending function, from a suspending lambda, from a coroutine, or from a lambda that is inlined into a coroutine (because then it is effectively called from the coroutine). The Kotlin compiler will give you a friendly reminder of this when you try to call a suspending function differently, as shown in [Listing 6.9](#). Intuitively, you have to enter the world of suspending functions first. This is done using coroutine builders, which are introduced in the next section.

#### **Listing 6.9 Cannot Call Suspending Function from Nonsuspending Function**

[Click here to view code image](#)

```
fun main(args: Array<String>) {  
  
    // Error: Suspend function must be called from coro  
  
    updateWeather()  
  
}
```

Before exploring different ways to enter the world of suspending functions, I want to give more examples of how suspending functions compose naturally and allow control flow structures. Basically, they abstract away much of the complexity of concurrency and allow you to write code as you're used to. For instance, imagine each user has a *list* of locations for which they want weather info. You can simply call suspending functions from a loop as in [Listing 6.10](#).

#### **Listing 6.10 Suspending Functions and Loops**

[Click here to view code image](#)

```
suspend fun updateWeather(userId: Int) {  
  
    val user = fetchUser(userId)  
  
    val locations = fetchLocations(user)    // Now return  
  
    for (location in locations) {          // Uses loop  
  
        val weatherData = fetchWeather(location)
```

```
        updateUi(weatherData, location)  
    }  
}
```



Similarly, you can use **try-catch** expressions as usual to handle exceptions, as demonstrated in [Listing 6.11](#). This provides a familiar way to handle exceptions on different levels of granularity without the need for additional failure callbacks or failure combinators.

#### [Listing 6.11 Suspending Functions and Exception Handling](#)

[Click here to view code image](#)

---

```
suspend fun updateWeather(userId: Int) {  
  
    try { // Uses try-catch naturally within suspendir  
  
        val user = fetchUser(userId)  
  
        val location = fetchLocation(user)  
  
        val weatherData = fetchWeather(location)  
  
        updateUi(weatherData)  
  
    } catch(e: Exception) {  
  
        // Handle exception  
  
    } finally {  
  
        // Cleanup  
  
    }  
}
```



Due to the fact that suspending functions retain the natural return value, they can also be combined with Kotlin's powerful standard library and higher-order functions. For instance, [Listing 6.10](#) could be rewritten as in [Listing 6.12](#) because `fetchLocations` directly returns a `List<Location>` (the natural return type).

#### [Listing 6.12 Suspending Functions and Higher-Order Functions](#)

[Click here to view code image](#)

```
suspend fun updateWeather(userId: Int) {  
  
    val user = fetchUser(userId)  
  
    fetchLocations(user).forEach {           // Can use a]  
  
        val weatherData = fetchWeather(it) // Can be ca]  
  
        updateUi(weatherData, it)  
  
    }  
  
}
```

In short, you can usually use suspending functions just like you would use any other function, except that you can only call them from a coroutine or another suspending function. In fact, a coroutine requires at least one suspending function in order to start, so every coroutine is supplied with a suspending function as its task upon creation. This is typically in the form of a suspending lambda that is passed into a coroutine builder to start a new coroutine, as discussed in the following section.

## Coroutine Builders

Coroutine builders open the gate to the world of suspending functions. They are the starting point to spin up a new coroutine and to make an asynchronous request. As mentioned, coroutine builders accept a suspending lambda that provides the coroutine's task. There are three essential coroutine builders with different intentions; they can be characterized very briefly as follows.

- `launch` is used for fire-and-forget operations without return value.
- `async` is used for operations that return a result (or an exception).
- `runBlocking` is used to bridge the regular world to the world of suspending functions.

## Bridge to Suspending World with `runBlocking`

The characterization of `runBlocking` may sound confusing at first but it is exactly what solves the problem from [Listing 6.9](#) where bridging between the two worlds is required. [Listing 6.13](#) demonstrates how to use `runBlocking` to be able to call suspending functions from `main`.

### Note

All listings in this book assume that Kotlin 1.3 has been released by the time you're reading this so that coroutines are no longer experimental. If they are still experimental, simply replace the package `kotlinx.coroutines` by `kotlinx.coroutines.experimental` in all imports in this chapter, as well as in [Chapter 7](#), [Android App Development with Kotlin: Kudoo App](#), and [Chapter 8](#), [Android App Development with Kotlin: Nutrilicious](#).

### [Listing 6.13 Using `runBlocking`](#)

[Click here to view code image](#)

```
import kotlinx.coroutines.runBlocking

fun main(args: Array<String>) {
    runBlocking {
        // Can call suspending functions inside runBlocking
        updateWeather()
    }
}
```

As a coroutine builder, `runBlocking` creates and starts a new coroutine. Living up to its name, it blocks the current thread until the block of code passed to it completes. The thread can still be interrupted by other threads, but it cannot do any other work. This is why `runBlocking` should never be called from a coroutine; coroutines should always be nonblocking. In fact, `runBlocking` is meant to be used in `main` functions and in test cases for suspending functions.

Note that `runBlocking`, like all coroutine builders, is not a keyword but just a function, similar to the scoping functions `with`, `apply`, and `run`. The lambda passed into the coroutine builder is a suspending lambda because of the signature of `runBlocking` (see [Listing 6.14](#)).

**Listing 6.14 Signature of `runBlocking` (Simplified)**

[Click here to view code image](#)

```
fun <T> runBlocking(block: suspend () -> T) // Blocks
```

Kotlin allows you to define a lambda parameter as a suspending function by prepending the function type with the **suspend** modifier. This is called a *suspending function type* (also *suspending lambda*), similar to regular function types such as `(String) -> Int`. The modifier on the parameter makes the passed lambda suspending so that other suspending functions can be called inside `runBlocking`. This is the trick that allows you to enter the world of suspending functions. A more idiomatic way to call `runBlocking` is shown in [Listing 6.15](#).

**Listing 6.15 Idiomatic Way to Use `runBlocking`**

[Click here to view code image](#)

```
fun main(args: Array<String>) = runBlocking<Unit> {  
    updateWeather()  
}  
  
// In a test case...  
  
@Test fun testUpdateWeather() = runBlocking { ... updat
```

The idiomatic way uses shorthand function notation to call `runBlocking`. In the case of the main function, you may have to explicitly specify `Unit` as the return type by using `runBlocking<Unit>`. This is *required* if the return value

of the lambda is not `Unit` (as defined by its last line) because the main function must return `Unit`. To avoid confusion, you may want to consistently use `runBlocking<Unit>` because that makes sure the return type is always correct, independent of the lambda's return type. This is only necessary when using it with shorthand notation for a main function. The following listings will assume Kotlin scripts again, so there will be no main functions unless necessary.

**Note**

`runBlocking` should never be used from within a coroutine, only in regular code to use suspending functions.

## Fire-and-Forget Coroutines using `launch`

Intuitively, the `launch` coroutine builder is used for fire-and-forget coroutines that execute independent of the main program. These are similar to threads in that they are typically used to introduce parallelism and uncaught exceptions from the coroutine are printed to `stderr` and crash the application. Listing 6.16 gives an introductory example.

**Listing 6.16** Launching a New Coroutine

[Click here to view code image](#)

```
import kotlinx.coroutines.*

// Launches non-blocking coroutine
launch {
    println("Coroutine started")    // 2nd print
    delay(1000)                   // Calls suspending
    println("Coroutine finished")  // 3rd print
}

println("Script continues")      // 1st print
```

```
Thread.sleep(1500)          // Keep program alive\n\n    println("Script finished") // 4th print
```

This code launches one new coroutine that first delays for a second—in a nonblocking way—and then prints a message. The `delay` function is a suspending function from the standard library and thus represents a suspension point. However, even the print statement before the delay is only printed after the print statement that follows `launch`. This simple example already uncovers Kotlin’s approach to coroutine scheduling: Coroutines are *appended* to the target thread for later execution. This is the same behavior as in JavaScript. In contrast, C# executes each `async` function immediately, until its first suspension point. Following that approach, “`coroutine started`” would be printed before “`Script continues`” in [Listing 6.16](#). This is more efficient, but the JavaScript approach is more consistent and less prone to errors, which is why the Kotlin team chose that option.

You can still use the C# behavior using `launch(CoroutineStart.UNDISPATCHED)`. For the sake of completeness, I want to mention that even with the default `CoroutineStart`, a coroutine may start immediately—namely if the dispatcher doesn’t need to dispatch. Dispatchers are discussed in detail in the following section. Still, the rule of thumb is that Kotlin schedules coroutines for later execution on the target thread.

#### Note

The default “target thread” for coroutine builders is determined by the `DefaultDispatcher` object. At the time of writing, this default dispatcher is `CommonPool`, which is a thread pool using `ForkJoinPool.commonPool` if that is available (since Java 8), otherwise it tries to create a new instance of `ForkJoinPool` (since Java 7) and otherwise it falls back to creating a thread pool with `Executors.newFixedThreadPool`. However, `CommonPool` is planned to be deprecated in the future so might no longer be the default dispatcher when you’re reading this.

In any case, the default dispatcher will still be a thread pool that is intended for *CPU-bound operations*. Thus, its pool size is equal to the number of cores. It should not be used for network calls or I/O operations. It is a good practice to have separate thread pools for such network calls and for I/O operations, and coroutines that access the UI must always use the main thread.

The code in Listing 6.16 can be improved. First, the code currently mixes suspending and regular functions because it uses the nonblocking `delay` as well as the blocking `Thread.sleep`. By using `runBlocking`, this can be made consistent because `Thread.sleep` can be replaced with `delay`. However, yielding control for a fixed time (using `delay`) to wait for a parallel job is not good practice. Instead, you can use the `Job` object that `launch` returns to await completion of the job explicitly.

Listing 6.17 applies both these improvements. One should note that waiting is only required because active coroutines do not prevent the program from exiting. Also, because `join` is a suspending function, `runBlocking` is still required.

**Listing 6.17 Waiting for a Coroutine**

[Click here to view code image](#)

```
import kotlinx.coroutines.*

runBlocking { // runBlocking is required to call suspend functions

    val job = launch { // launch returns a Job object
        println("Coroutine started")
        delay(1000)
        println("Coroutine finished")
    }

    println("Script continues")

    job.join() // Waits for completion of coroutine (runBlocking)
    println("Script finished")
}
```



So far, this exact behavior can be achieved with threads just as well. Kotlin even provides coroutine builder-like syntax for

thread creation with `thread { ... }`. But with coroutines, you can easily start 100,000 jobs to implement the scenario of 100,000 workers working in parallel. This is shown in [Listing 6.18](#). Doing this with threads would likely cause an out-of-memory exception, because each thread takes around 1MB of memory, or it would at least take orders of magnitude longer to finish due to large overhead for thread creation and context switching.

**Listing 6.18 Launching 100,000 Coroutines**

[Click here to view code image](#)

```
import kotlinx.coroutines.*  
  
runBlocking {  
    // Launch 100,000 coroutines  
  
    val jobs = List(100_000) {  
        launch {  
            delay(1000)  
            print("+")  
        }  
    }  
  
    jobs.forEach { it.join() }  
}
```

This prints “+” 100,000 times without an out-of-memory error. The listing uses the `List` constructor that accepts the list size and a lambda that defines how each list item is created. Because `launch` returns a `Job`, the result is a `List<Job>` that can be used to wait for all coroutines to complete. As shown in [Listing 6.19](#), it can also be used to cancel a coroutine. Note that `repeat` is just a utility function to write simple `for` loops more concisely.

**Listing 6.19 Cancelling a Coroutine**

[Click here to view code image](#)

```
import kotlinx.coroutines.*

runBlocking {
    val job = launch {
        repeat(10) {
            delay(300) // Cooperative cancellation
            println("${it + 1} of 10...") // Only 1 of 10,
        }
    }

    delay(1000)
    println("main(): No more time")
    job.cancel() // Can control cancellation on per-coroutine basis
    job.join() // Then wait for it to cancel
    println("main(): Now ready to quit")
}
```

Once `job.cancel()` is called, the coroutine receives a signal to cancel its execution. This signal is represented by a `CancellationException`. This exception does not print to `stderr` because it is an expected way for a coroutine to stop. Similar to cooperative multitasking, coroutines also follow *cooperative cancellation*. This means that coroutines must explicitly support cancellation. You can do this by regularly calling any suspending function from Kotlin's standard library as they are all cancellable, thus making your coroutine cancellable. Another way is to regularly check the `isActive` property available inside the coroutine scope. First, consider the noncooperative implementation in [Listing 6.20](#), which is *not cancellable*.

**Listing 6.20 Noncooperative Coroutine Cannot Be Cancelled**

[Click here to view code image](#)

```
import kotlinx.coroutines.*

runBlocking {
    val job = launch {
        repeat(10) {
            Thread.sleep(300)           // Non-cooperative sleep
            println("${it + 1} of 10...") // All ten iterations
        }
    }

    delay(1000)
    job.cancelAndJoin() // Cancel is ignored, will wait
}

```

This code uses the noncooperative `Thread.sleep` instead of Kotlin's cooperative `delay`. This is why, even though cancellation is issued after 1 second, the coroutine will continue running until the end, keeping the thread busy in the meantime. Therefore, the call to the utility function `cancelAndJoin` has to wait for about 2 seconds. The easiest fix in this case is to use `delay` instead of `Thread.sleep`. That way, the coroutine can be cancelled whenever it reaches `delay`, thus introducing ten cancellation points.

Another way that doesn't require calling a suspending function from the standard library is to check `isActive` manually, as done in Listing 6.21.

**Listing 6.21 Making Coroutine Cooperative Using `isActive`**

[Click here to view code image](#)

```
import kotlinx.coroutines.*
```

```
runBlocking {
```

```
    val job = launch {
```

```
        repeat(10) {
```

```
            if (isActive) { // Cooperative
```

```
                Thread.sleep(300)
```

```
                println("${it + 1} of 10...")
```

```
            }
```

```
        }
```

```
    }
```

```
    delay(1000)
```

```
    job.cancelAndJoin()
```

```
}
```

This introduces ten points at which the coroutine can be cancelled—each time `isActive` is checked. Cancellation causes `isActive` to be `false`, thus making the coroutine jump through the remaining iterations quickly, evaluating the condition to false each time and finishing execution. So to achieve cooperative cancellation, you just have to make sure that you check `isActive` often enough and that `isActive` being `false` causes the coroutine to finish.

## Asynchronous Calls with Return Value Using `async`

The next essential coroutine builder is `async`, which is used for asynchronous calls that return a result or an exception. This is highly useful for REST API requests, fetching entries from a database, reading contents from a file, or any other operation that introduces wait times and fetches some data. If the lambda passed to `async` returns type `T`, then the `async` call returns a `Deferred<T>`. `Deferred` is a lightweight future implementation from the Kotlin coroutines library. To get the actual result, you must call `await` on it.

As in C#, `await` waits for the completion of the `async` task—of course nonblocking—to be able to return the result.

However, unlike C#, Kotlin has no `await` keyword, it's just a function. Listing 6.22 demonstrates how easily `async` allows you to make multiple asynchronous calls simultaneously (if they're independent).

**Listing 6.22 Starting Asynchronous Tasks**

[Click here to view code image](#)

```
import kotlinx.coroutines.*
```

```
fun fetchFirstAsync() = async { // Return type is Deferred<Int>
    delay(1000)
    294 // Return value of lambda, type Int
}
```

```
fun fetchSecondAsync() = async { // Return type is Deferred<Int>
    delay(1000)
    7 // Return value of lambda, type Int
}
```

```
runBlocking {  
    // Asynchronous composition: total runtime ~1s  
  
    val first = fetchFirstAsync()    // Inferred type:  
  
    val second = fetchSecondAsync() // Inferred type:  
  
    val result = first.await() / second.await() // Await both  
    println("Result: $result")           // 42  
  
}
```

The two asynchronous functions both start a new coroutine and simulate network calls that each take 1 second and return a value. Because the lambdas passed to `async` return `Int`, the two functions both return a `Deferred<Int>`. Following naming conventions, the function names carry an “`async`” suffix so that asynchrony is explicit at the call site and a `Deferred` can be expected. Calling `first.await()` suspends execution until the result is available and returns `Int`, the unwrapped value. So when the code reaches the expression `first.await()`, it will wait for about a second before moving on to `second.await()`, which requires no further waiting at that point because that asynchronous job is also finished after 1 second. Therefore, the total execution time of [Listing 6.22](#) is about one second.

With suspending functions in place of `async`, it would take 2 seconds because suspending functions are sequential by default. Similarly, calling `await` on the first `Deferred` before even issuing the second asynchronous call, as in [Listing 6.23](#), also results in a runtime of around 2 seconds. This is because the first `await` call suspends execution for a second, and only after that, the second coroutine is created and can start working.

#### [Listing 6.23 Running Asynchronous Functions Synchronously](#)

[Click here to view code image](#)

```
import kotlinx.coroutines.*  
  
runBlocking {  
    // Effectively synchronous: total runtime ~2s  
  
    val first = fetchFirstAsync().await()      // Inferred  
  
    val second = fetchSecondAsync().await()    // Inferred  
  
}
```

The two asynchronous functions used so far are not idiomatic Kotlin code. Kotlin encourages sequential behavior as the default, and asynchrony as an opt-in option. However, in [Listing 6.23](#), asynchrony is the default and synchrony must be forced by calling `await` directly after each call. [Listing 6.24](#) shows the idiomatic approach.

**Listing 6.24 Idiomatic Asynchrony**

[Click here to view code image](#)

```
import kotlinx.coroutines.*  
  
  
suspend fun fetchFirst(): Int {  
    delay(1000); return 294  
  
}  
  
  
suspend fun fetchSecond(): Int {  
    delay(1000); return 7  
  
}  
  
  
runBlocking {  
  
    val a = async { fetchFirst() }    // Asynchrony is explicit  
  
    val b = async { fetchSecond() }  // Asynchrony is explicit
```

```
    println("Result: ${a.await() / b.await()}"
```

```
}
```

Notice that the two fetch functions are now suspending functions, have the natural return value, and there is no need for an “`async`” suffix. This way, they behave sequentially by default and you can opt in to asynchrony simply by wrapping the call into `async { ... }`. This keeps the asynchrony explicit.

In contrast to `launch`, you should not forget about coroutines started with `async` because they carry a result or an exception. Unless processed, an exception is swallowed silently. [Listing 6.25](#) shows how to handle the failure case. Note that `await` fulfills two functions: It returns the result if the operation completed successfully, and it rethrows the exception if the operation completed exceptionally. Thus, it can be wrapped into a **try-catch** block.

#### [Listing 6.25 Handling Exceptions from Asynchronous Calls](#)

[Click here to view code image](#)

```
import kotlinx.coroutines.*
```

```
runBlocking {
```

```
    val deferred: Deferred<Int> = async { throw Exception("Test") }
```

```
    try {
```

```
        deferred.await() // Tries to get result of async
```

```
    } catch (e: Exception) {
```

```
        // Handle failure case here
```

```
    }
```

```
}
```

You can also check the state of a `Deferred` directly using the properties `isCompleted` and `isCompletedExceptionally`. Be aware that the latter being `false` does not mean the operation completed successfully, it may instead still be active. In addition to these, there is also a `deferred.invokeOnCompletion` callback in which you may use `deferred.getCompletionExceptionOrNull` to get access to the thrown exception. In most cases, the approach in [Listing 6.25](#) should suffice and is more readable.

**Note**

Notice the similar relations between `launch/join` and `async/await`. Both `launch` and `async` are used to start a new coroutine that performs work in parallel. With `launch`, you may wait for completion using `join`, whereas with `async`, you normally use `await` to retrieve the result.

While `launch` returns a `Job`, `async` returns a `Deferred`. However, `Deferred` implements the `Job` interface so you may use `cancel` and `join` on a `Deferred` as well, although this is less common.

## Coroutine Contexts

All coroutine builders accept a `CoroutineContext`. This is an indexed set of elements such as the coroutine name, its dispatcher, and its job details. In this regard, `CoroutineContext` is similar to a set of `ThreadLocals`, except that it's immutable. This is possible because coroutines are so lightweight that you can simply create a new one if the context should be modified. The two most important elements of a coroutine context are the `CoroutineDispatcher` that decides on which thread the coroutines run, and the `Job` that provides details about the execution and can be used to spawn child coroutines. Apart from these two, context also contains a `CoroutineName` and a `CoroutineExceptionHandler`.

### Coroutine Dispatchers

Whenever a coroutine resumes, its dispatcher decides on which thread it resumes. Listing 6.26 shows how to pass a `CoroutineDispatcher` to launch a coroutine on the UI thread on Android.

**Listing 6.26 Launching a Coroutine on the UI Thread (Android)**

[Click here to view code image](#)

```
import kotlinx.coroutines.android.UI

import kotlinx.coroutines.launch

launch(UI) { // The UI context is provided by the co
    updateUi(weatherData)
}
```

The UI dispatcher is provided by the Android-specific `coroutines` package, which is imported with the Gradle dependency `org.jetbrains.kotlinx:kotlinx-coroutines-android`. Such UI dispatchers exist for all major UI frameworks, including Android, Swing, and JavaFX. You can also implement custom ones. The Android UI dispatcher is defined as a top-level public property as in Listing 6.27. It refers to Android's main looper, which lives in the main thread.

**Listing 6.27 Definition of UI Context on Android**

[Click here to view code image](#)

```
val UI = HandlerContext(Handler(Looper.getMainLooper()
```

Before resuming a coroutine, the `CoroutineDispatcher` has a chance to change where it resumes. A coroutine with the default dispatcher always resumes on one of the `CommonPool` threads and therefore jumps between threads with every suspension. All the UI dispatchers make sure that execution *always* resumes on the UI thread. In contrast, a

coroutine with default context uses the `CommonPool` and thus may resume on any of the background threads from this pool. Listing 6.28 shows three equivalent `launch` statements (at the time of writing; the `DefaultDispatcher` is subject to change and may no longer be `CommonPool`).<sup>4</sup>

4. Please visit the companion website, <https://kotlinandroidbook.com/>, for any such updates.

#### **Listing 6.28 Default Coroutine Dispatcher**

[Click here to view code image](#)

```
launch { ... }

launch(DefaultDispatcher) { ... }

launch(CommonPool) { ... }
```

Another possibility is to run a coroutine in a new, dedicated thread. Keep in mind that this introduces the cost of thread creation and therefore removes the advantage of lightness.

Listing 6.29 shows how to do this using `newSingleThreadContext`. This creates a pool of one thread. You can also create a thread pool with multiple threads using `newFixedThreadPoolContext`.

#### **Listing 6.29 Dispatching Onto New Thread**

[Click here to view code image](#)

```
launch(newSingleThreadContext("MyNewThread")) { ... }
```

If you want a coroutine to be completely unconfined from any threads (and the `CommonPool`), you can use the `Unconfined` dispatcher as in Listing 6.30. This starts the coroutine in the current call frame and then resumes it wherever the resuming suspension function runs. Basically, the unconfined dispatcher never intercepts resumption and just lets the suspending function that resumes the coroutine decide which thread to run on. An example that illustrates this behavior is given later in Listing 6.32.

#### **Listing 6.30 Unconfined Dispatcher**

```
launch(Unconfined) { ... }
```

The last dispatcher, or more precisely, `CoroutineContext` that *contains* a dispatcher, is `coroutineContext` (see Listing 6.31). This is a top-level property that's available inside every coroutine and represents its context. By using a coroutine's context to launch another one, that other one becomes a child coroutine. To be precise, the parent-child relationship is not induced by the dispatcher but by the job that is also part of the context. Cancelling a parent job recursively cancels all its child jobs (still relying on cooperative cancellation).

### **Listing 6.31 Creating a Child Coroutine**

[Click here to view code image](#)

```
    launch(coroutineContext) { ... }
```

To get a better feeling for the differences among these dispatchers and what behavior results from them, Listing 6.32 uses all of them and compares which threads they are dispatched to before and after a suspension.

### **Listing 6.32 Comparing Dispatchers**

[Click here to view code image](#)

```
delay(500)

println("Default: In thread ${Thread.currentThread().name}")

}

jobs += launch(newSingleThreadContext("New Thread"))

println("New Thread: In thread ${Thread.currentThread().name}")

delay(500)

println("New Thread: In thread ${Thread.currentThread().name}")

}

jobs += launch(Unconfined) {

println("Unconfined: In thread ${Thread.currentThread().name}")

delay(500)

println("Unconfined: In thread ${Thread.currentThread().name}")

}

jobs += launch(coroutineContext) {

println("CC: In thread ${Thread.currentThread().name}")

delay(500)

println("CC: In thread ${Thread.currentThread().name}")

}

jobs.forEach { it.join() }

}
```

◀ ▶

This code starts four coroutines inside `runBlocking` that cover all the discussed dispatchers. Each coroutine first executes a print statement, then reaches a delay that suspends

it, and next resumes for another print statement. The interesting part here is to see in which thread each coroutine runs before and after the suspension. Table 6.1 gives an overview of the results.

Table 6.1 Results of Dispatching from Listing 6.32

Coroutine Dispatcher	Before Delay	After Delay
Default Dispatcher (CommonPool)	In thread ForkJoinPoo l  .commonPo ol-worker-1	In thread ForkJoinPool .commonPool-worker-1
Single-thread context	In thread New Thread	In thread New Thread
Unconfined	In thread main	In thread kotlinx.coroutines.DefaultExecutor
coroutineContext	In thread main	In thread main

With the default dispatcher, the coroutine is always dispatched to some worker thread from the pool. Running this with Java 8 as the compilation target uses the `ForkJoinPool.commonPool` as expected. With a single-thread context, the coroutine always executes within that particular thread. For `Unconfined` and `coroutinesContext`, it is important to notice that the current thread is the main thread because that is where `runBlocking` executes. Because `Unconfined` never actively dispatches anywhere, the coroutines just start off in the current main thread. The same goes for `coroutineContext` because it inherits the context from

`runBlocking`. However, after the suspension, the unconfined coroutine resumes wherever the suspending function tells it to. Since `delay` uses a so-called `DefaultExecutor` from the `kotlinx.coroutines` package, that is where it resumes. In contrast, `coroutineContext` always resumes in the main thread that it inherited from its parent.

In many cases, you want more fine-grained control of the context. More specifically, you want different parts of a coroutine to run on different threads. A typical scenario is making asynchronous calls to fetch data on a background thread and then displaying it in the UI once it is available. The `updateWeather` function presented above resembles this pattern. A better implementation for it is given in [Listing 6.33](#).

**Listing 6.33 Switching Context Inside Coroutine**

[Click here to view code image](#)

---

```
import kotlinx.coroutines.*  
  
import kotlinx.coroutines.android.UI  
  
  
suspend fun updateWeather(userId: Int) {  
  
    val user = fetchUser(userId)  
  
    val location = fetchLocation(user)  
  
    val weatherData = fetchWeather(location)  
  
  
    withContext(UI) {  
  
        updateUi(weatherData)  
  
    }  
  
}  
  
  
// Call-site  
  
launch { updateWeather() }
```

Running part of a coroutine in a specific context is done using `withContext`. That way, only the code that touches the UI is executed on the UI thread. The suspending function itself can now be launched in a thread pool to perform the asynchronous calls without affecting the UI thread at all until the final result is available. Note that context switching between coroutines is far less expensive than context switching between threads if the coroutines run in the same thread. However, if they are in different threads (like here), switching coroutine contexts requires an expensive thread context switch.

**Tip**

The `withContext` function is crucial on Android and other UI frameworks to easily switch between background threads and the UI thread (all UI updates must be done on the UI thread).

Unless you actually want multiple asynchronous calls running in parallel, `withContext` is often the better alternative to `async-await` when expecting back a return value from a suspending function. For instance, imagine you want to perform a database operation:

[Click here to view code image](#)

```
suspend fun loadUser(id: Int): User = withContext(DB) { ... }
```

This allows using the natural return type and ensures that the database operation always runs in the dedicated DB context. You will see several examples of this in [Chapters 7 and 8](#).

Another convenient library function for asynchronous calls is `withTimeout`, which lets you easily timeout calls after a specified amount of time. [Listing 6.34](#) shows its basic usage. This code reaches the print statement twice before timing out.

**Listing 6.34 Timeout for Asynchronous Calls**

[Click here to view code image](#)

```
import kotlinx.coroutines.*
```

```
runBlocking {  
    withTimeout(1200) {
```

```
repeat(10) {  
  
    delay(500)  
  
    println("${it + 1} of 10")  
  
}  
  
}  
  
}
```

Similar to regular cancellation, a timeout is signaled by a `TimeoutCancellationException` but unlike a `CancellationException`, a `TimeoutCancellationException` gets printed to `stderr` and causes the program to stop if it is not caught. That's because a timeout is not the expected way for the coroutine to stop. Listing 6.35 shows a safer way to call `withTimeout`. A `finally` block is useful to close any connections opened by the coroutine.

#### **Listing 6.35 Handling Timeouts**

[Click here to view code image](#)

---

```
import kotlinx.coroutines.*  
  
  
runBlocking {  
  
    try {  
  
        withTimeout(1200) {  
  
            repeat(10) {  
  
                delay(500)  
  
                println("${it + 1} of 10")  
  
            }  
  
        }  
  
    } catch (e: TimeoutCancellationException) {  
  
        println("Time is up!")  
  
    }  
  
}
```

```
    } finally {

        println("Cleaning up open connections...")

    }

}
```

### A Coroutine's Job

Apart from a `CoroutineDispatcher`, the context contains a `Job` object that is also worth exploring in more detail. It can be used to create parent-child hierarchies of coroutines that facilitate cancellation, and to bind coroutines to the lifecycle of an activity, among other things. As you know, passing `coroutineContext` creates a child coroutine. Because it is passed in by default, using a nested coroutine builder suffices to create a child coroutine. Listing 6.36 shows how this affects cancellation. It starts ten coroutines of which only three will complete before being cancelled. This shows that cancelling the parent coroutine cancels its children, and in fact this goes on recursively.

**Listing 6.36 Child Coroutines and Cancellation**

[Click here to view code image](#)

---

```
import kotlinx.coroutines.*

runBlocking {

    val parent = launch {

        repeat(10) { i ->

            launch { // Implicitly uses coroutineContext,
                delay(300 * (i + 1))

                println("Coroutine ${i + 1} finished") // Or

            }
        }
    }
}
```

```
    }

    delay(1000)

    parent.cancelAndJoin() // Cancels parent coroutine

}
```

Here, cancelling the parent coroutine recursively cancels all its children so that only the first three coroutines get to print something. The `Job` also provides access to important properties of the job execution. The property `isCompleted` is one of these, and similarly there are also `isCancelled` and `isActive`. Useful functions apart from `cancel` and `join` include `cancelChildren` to only cancel all child coroutines, `joinChildren` to wait for the completion of all the coroutine's children, and the `invokeOnCompletion` callback.

In principle, a coroutine may run longer than its associated UI element but still hold references that prevent garbage collection. For instance, an Android activity (a “screen” in an app) may have started an asynchronous call just before the user rotates the app, causing the activity to be destroyed. You can avoid such a memory leak by creating an explicit parent job per activity, which is used as the parent for every coroutine that should be bound to its corresponding activity. This approach works for any elements that have a lifecycle and may get destroyed while a coroutine runs, but we will consider an Android activity in the following. So the goal is to bind a parent job to an activity and cancel any coroutines that were launched and are still running when that activity gets destroyed. One approach to implement this is described in the “Guide to UI Programming with Coroutines.”<sup>5</sup> As a first step, you can model the concept of a class that has a parent job attached to it as an interface, as in Listing 6.37.

5. <https://github.com/Kotlin/kotlinx.coroutines/blob/master/ui/coroutines-guide-ui.md>

**Listing 6.37 Binding Coroutines to Activity #1: The JobHolder Interface**

[Click here to view code image](#)

```
import kotlinx.coroutines.*  
  
interface JobHolder { val job: Job }
```

This interface can then be implemented by any activity that should have coroutines attached to its lifecycle. The key point is that the activity then calls `job.cancel()` in its `onDestroy` method, which will also cancel all child coroutines of the job, as shown in [Listing 6.38](#).

**Listing 6.38 Binding Coroutines to Activity #2: The Activity**

[Click here to view code image](#)

```
import kotlinx.coroutines.Job  
  
import android.support.v7.app.AppCompatActivity  
  
  
class MainActivity : AppCompatActivity(), JobHolder {  
  
    override val job = Job()  
  
    override fun onDestroy() {  
  
        super.onDestroy()  
  
        job.cancel()  
  
    }  
  
}
```

With this, if you launch all coroutines with `launch(job) { ... }` or `async(job) { ... }`, you no longer need to keep track of the coroutines started by the activity. If the activity gets destroyed, the coroutine cancellation policy will automatically cancel all coroutines attached to the activity. The guide goes one step further to make a job available from every view. On Android, every view carries a context that refers to

its activity, so this context can be used to access the parent job if the activity is a `JobHolder`, as in [Listing 6.39](#).

**Listing 6.39 Binding Coroutines to Activity #3: Accessing Job from Views**

[Click here to view code image](#)

```
import android.view.View

import kotlinx.coroutines.NonCancellable

val View.contextJob

    get() = (context as? JobHolder)?.job ?: NonCancellable
```

For every view, this adds an extension property that returns a reference to the parent activity's job if it is a `JobHolder`, or to `NonCancellable` otherwise. Using

`launch(contextJob)`, every view can start a coroutine that is automatically cancelled if the containing activity is destroyed.

The dispatcher and job are the context elements you will likely use most often, but there are more predefined context elements that should be mentioned. Specifically, there is `CoroutineName`, which is useful for debugging, and `CoroutineExceptionHandler`, which is called in case an exception occurs during coroutine execution. You can simply create instances of these and pass them as the context parameter, as shown in [Listing 6.40](#).

**Listing 6.40 CoroutineName and CoroutineExceptionHandler**

[Click here to view code image](#)

```
import kotlinx.coroutines.*

// CoroutineName

val name = CoroutineName("Koroutine")

launch(name) { ... }
```

```
// CoroutineExceptionHandler

val exceptionHandler = CoroutineExceptionHandler { context, exception ->

    println("Crashed with context $context")

    exception.printStackTrace()

}

launch(exceptionHandler) { ... }
```

At this point, the observant reader may be wondering how to combine these context elements. There is only one `CoroutineContext` parameter, so what if you want a coroutine running on the UI thread with a specific parent job, or a coroutine with a name *and* an exception handler, or all of these? Conveniently, the `CoroutineContext` class defines an operator function `plus`, which allows exactly these combinations. [Listing 6.41](#) shows the operator's signature.

#### [Listing 6.41 CoroutineContext Combination Operator](#)

[Click here to view code image](#)

---

```
public operator fun plus(context: CoroutineContext):
```

Because all context elements implement `CoroutineContext`, each of them work with this operator. So you can easily put together the desired context for a coroutine as in [Listing 6.42](#).

#### [Listing 6.42 Combining Contexts](#)

[Click here to view code image](#)

---

```
// CoroutineName + CoroutineExceptionHandler

launch(name + exceptionHandler) { ... }

// Job + CoroutineDispatcher

launch(contextJob + UI) { ... } // Uses 'contextJob' if
```

Note that elements from the context on the right-hand side replace all elements with the same key from the left-hand side. The keys are basically the class names of the elements, meaning you can access them as shown in Listing 6.43.

**Listing 6.43 Accessing Context Elements**

[Click here to view code image](#)

```
import kotlinx.coroutines.*  
  
import kotlin.coroutines.experimental.ContinuationInt  
  
  
launch(name + exceptionHandler) {  
  
    println("Context: ${coroutineContext}")  
  
    println("Job: ${coroutineContext[Job]}")  
  
    println("Dispatcher: ${coroutineContext[Cont...  
  
    println("Name: ${coroutineContext[Coro...  
  
    println("Exception Handler: ${coroutineContext[Coro...  
  
}
```

Recall that each coroutine carries a `coroutineContext` that can be accessed from inside it. Again, this context is an indexed set of elements. Particular elements from it can be accessed using the corresponding key. For instance, to retrieve the `Job` from it, you can use `coroutineContext[Job]`. This works because the `Job` class defines a **companion object** `Key : CoroutineContext.Key<Job>` so that the element access is equivalent to `coroutineContext[Job.Key]`.

## Further Coroutine Parameters

At this point, you have a good understanding of the coroutine context, its elements, and how to use them. But in addition to the `CoroutineContext` parameter that every coroutine builder accepts, `launch` and `async` also accept a `CoroutineStart` parameter. You have briefly encountered this in the context of JavaScript versus C#-style coroutine scheduling.

As discussed, Kotlin typically schedules a coroutine for later execution (JavaScript-style) but you can use `CoroutineStart.UNDISPATCHED` to use the C#-style instead, meaning the coroutine is run immediately until it reaches its first suspension point. Apart from `CoroutineStart.DEFAULT` and `CoroutineStart.UNDISPATCHED`, there are two more interesting ways to schedule your coroutines.

The first one is `CoroutineStart.LAZY`, which starts running the coroutine only when necessary. For `launch`, this means it can be started explicitly using `job.start()` or implicitly once `job.join()` is called. For `async`, there's the additional option to call `await` to trigger execution. Note that `join` causes the coroutine to execute synchronously in [Listing 6.44](#). It's still nonblocking, but be aware that lazy coroutines may take a while to complete.

[Listing 6.44 Starting Coroutines Lazily](#)

[Click here to view code image](#)

```
runBlocking {  
  
    val job = launch(start = CoroutineStart.LAZY) { //  
        println("Lazy coroutine started")  
  
    }  
  
    println("Moving on...")  
  
    delay(1000)  
  
    println("Still no coroutine started")
```

```
    job.join() // Triggers execution of the coroutine

    println("Joined coroutine")

}
```

The last possible value is `CoroutineStart.ATOMIC`, which guarantees that the coroutine cannot be cancelled before it at least starts its execution.

The last parameter for coroutine builders (apart from the lambda block) is a separate parent job. This was introduced in the current version of coroutines and overrides any parent job from the coroutine context. But you can also use the coroutine context to define the parent job as shown in this chapter.

You are free to implement your own coroutine builders and dispatchers, although wrappers are available for the most prominent frameworks and APIs. These should cover most use cases<sup>6</sup>:

6. A complete, up-to-date list is provided at <https://kotlin.github.io/kotlinx.coroutines/>

- `kotlinx-coroutines-core` for basic coroutine support
- `kotlinx-coroutines-android` for Android development
- `kotlinx-coroutines-javafx` for JavaFX
- `kotlinx-coroutines-swing` for Swing
- `kotlinx-coroutines-jdk8` for interoperating with `CompletableFuture`
- `kotlinx-coroutines-nio` for interoperability with Java's nonblocking I/O
- `kotlinx-coroutines-rx2` for RxJava 2.x support

For instance, if you prefer working with Java's `CompletableFuture` and its fluent API, you can still use coroutines. In `kotlinx-coroutines-jdk8`, you find a future coroutine builder, which works similar to `async` (and similar to `CompletableFuture.supplyAsync`) but returns a `CompletableFuture` instead of a `Deferred` (see Listing 6.45).

**Listing 6.45 Combining Coroutines and CompletableFuture**

[Click here to view code image](#)

```
import kotlinx.coroutines.*  
  
import kotlinx.coroutines.future.*  
  
  
fun fetchValueAsync() = future { delay(500); 7 } //  
  
  
runBlocking {  
  
    fetchValueAsync().thenApply { it * 6 }  
  
    .thenAccept { println("Retrieved value: $it") }  
  
    .await()  
  
}  
  
◀ ▶
```

To recap what was covered so far and to give a better intuition and mental model of the world of suspending functions and how it relates to regular code, [Figure 6.3](#) illustrates their relationships. As mentioned, coroutine builders allow you to enter the world of suspending functions, suspending functions keep you there, and calling `SuspendCoroutine` in a suspending function captures its continuation so that you can work with it as with a callback, effectively taking you back to regular code.

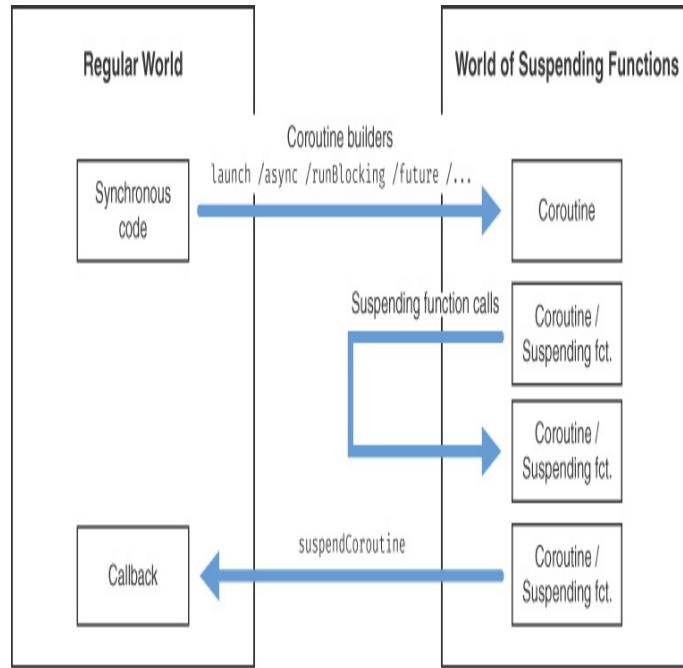


Figure 6.3 Overview of the relationship between regular code and the “world of suspending functions,” and the different ways to switch between them

## Generators

As mentioned at the beginning of this chapter, coroutines are intended to be general enough to also incorporate the concepts of `async-await`, other asynchronous libraries, and generators. At this point, you know how to write `async-await`-style code with Kotlin’s coroutines, and you also know that they can wrap other asynchronous APIs such as Java NIO and other future implementations such as `CompletableFuture`. This section introduces generators and explains how they are implemented using coroutines.

A generator is an iterator that yields values lazily on demand. Although its implementation may look sequential, execution is suspended until the next value is requested. Values are emitted using the `yield` function, similar to a `yield` keyword in languages such as Python or C#. Another, more theoretical way to look at generators is that they are less general coroutines in that coroutines can yield control to any other coroutine, whereas generators always yield control to their

caller. Listing 6.46 implements the Fibonacci sequence with a generator.

**Listing 6.46 Creating a Suspending Sequence (Generator)**

[Click here to view code image](#)

```
import kotlin.coroutines.experimental.buildSequence

val fibonacci = buildSequence {
    yield(1)
    var a = 0
    var b = 1
    while (true) {
        val next = a + b
        yield(next)
        a = b
        b = next
    }
}

fibonacci.take(10).forEach { println(it) } // 1, 1,
```

Similar to coroutine builders, `buildSequence` accepts a suspending lambda but is not a suspending function itself, and it uses coroutines to compute new values. `yield` is a suspending function that suspends execution until the next element is requested so that each call to `next` causes the generator to run until it reaches the next `yield`. Note that you can use control flow structures naturally within generators as you can inside any suspending function. Also note that `buildSequence` is similar to `generateSequence` but uses suspending functions and `yield`, whereas

`generateSequence` calls the supplied function each time to produce the next value and does not use coroutines at all.

**Note**

Generators do not introduce asynchrony, their execution is completely controlled by the caller, and they run synchronously with their caller. This is one example of synchronous suspending functions and demonstrates that suspending computation does not necessarily imply asynchrony. You can think of it as no progress being made inside a generator unless it is invoked by a caller; there is no asynchronous computation taking place. In other words, `yield` is a suspending function but not an asynchronous function.

This also explains why Kotlin uses `suspend` as the keyword and not `async` as in other languages like C#. Remember: In Kotlin, you have a choice to opt in to asynchrony; it is not implied by `suspend` alone.

Aside from `buildSequence`, Kotlin also offers a `buildIterator` function. An iterator is almost the same as a sequence—in fact, sequences in Kotlin just wrap iterators. In contrast to iterators, they can typically be iterated multiple times, and they are stateless. In other words, each iteration over a sequence gives you a fresh iterator. Listing 6.46 showed a sequence that allows traversing its iterator multiple times: You could take the first five elements from `fibonacci` again do to something with them. Other sequence implementations only allow accessing their iterator once (see `ConstrainedOnceSequence`).

One more concept that can be explained well by means of generators is the `@RestrictsSuspension` annotation. In the standard library, it is used on the `SequenceBuilder` class, which is the receiver of the lambda passed to `buildSequence` and `buildIterator` (see Listing 6.47). So inside that lambda you have access to methods from `SequenceBuilder`, which is where `yield` and `yieldAll` come from.

**Listing 6.47 Declaration of buildSequence**

[Click here to view code image](#)

---

```
fun <T> buildSequence(block: suspend SequenceBuilder<
```



Now imagine you are the library developer for `SequenceBuilder`. You must make sure that *all* suspension points are exclusively from yields when using that class. In other words, you must restrict library users so that they cannot add new ways to suspend the coroutine via extension functions. This is what `@RestrictsSuspension` in the declaration of `SequenceBuilder` does (see Listing 6.48).

**Listing 6.48 Declaration of SequenceBuilder with Restricted Suspension**

[Click here to view code image](#)

```
@RestrictsSuspension

public abstract class SequenceBuilder<in T> { ... } //
```

Because of this annotation, suspending extension functions must call either another suspending extension or a suspending member function of `SequenceBuilder`. This way, all suspending extensions end up calling a suspending member function—in this case, `yield` or `yieldAll`.

## Actors and Channels

At the outset of this chapter, I briefly mentioned the actor model for concurrent programming. In this model, an actor represents a unit of concurrent computation that is decoupled from other actors. In particular, actors share no mutable state. Instead, they communicate only via message passing. To receive messages, each actor has a message channel attached to it. Based on incoming messages, the actor decides what to do—spawn more actors, send messages, or manipulate its private state. The key point is that there is no chance for race conditions and thus no need for locks when there is no shared state.

### Note

In the original actor model described by Carl Hewitt, there are no channels.<sup>7</sup> Actors communicate directly, and a channel would be another separate actor because, in the model, everything is an actor. However, programming languages that incorporate actors such as Erlang, Elixir, Go, or Kotlin do not strictly follow this original model.

7. Hewitt talks about his concept here: [https://youtu.be/7erJ1DV\\_Tlo](https://youtu.be/7erJ1DV_Tlo)

More generally, actors can have a many-to-many relationship to channels so that a single actor may read from multiple channels. Likewise, multiple actors may read messages from the same channel to distribute work. Although this is a model of concurrency, actors themselves work sequentially. If an actor receives three messages, it will process them one after the other. The power of the model originates from a system of actors working in parallel. Concurrency is facilitated by communicating only via message passing.

The actor model has affected many programming languages. For example, it is a fundamental concept in Erlang,<sup>8</sup> the Ericsson language, because its originators needed a model for fault-tolerant distributed systems. Consequently, actors are also essential in Elixir,<sup>9</sup> a modern language that builds on Erlang. The Go programming language,<sup>10</sup> developed by Google, is also directly influenced by the actor model. In Go, actors are implemented using “goroutines,” which are closely related to Kotlin’s coroutines.<sup>11</sup> Accordingly, you can implement actors with coroutines in Kotlin. Listing 6.49 presents a simple actor that can receive a string message and print it. The code prints “Hello World!” when run.

8. <https://www.erlang.org/>

9. <https://elixir-lang.org/>

10. <https://golang.org/>

#### **Listing 6.49 A Simple Actor**

[Click here to view code image](#)

---

```
import kotlinx.coroutines.channels.actor
import kotlinx.coroutines.runBlocking

val actor = actor<String> {
    val message = channel.receive()
    println(message)
}
```

```
runBlocking {  
    actor.send("Hello World!") // Sends an element to  
    actor.close() // Closes channel because  
}
```

To create a new actor, Kotlin provides an `actor` function, which is in fact another coroutine builder—remember that actors are coroutines in Kotlin, so creating an actor means creating a new coroutine. The actor tries to receive a message from its `channel`, which is accessible inside the lambda from its receiver type `ActorScope`. From the main thread, a string is sent to the actor's channel and after that, it is closed. There are several things to note here:

- Both `send` and `receive` are suspending functions; `send` suspends execution as long as the channel is full, and `receive` suspends execution as long as the channel is empty.
- For that reason, the main function must be wrapped with `runBlocking` to be allowed to call the suspending `send`.
- Calling `close` does not immediately stop the actor coroutine. Instead, the mental model is that `close` sends a special “close token” to the channel, and the channel is still read first-in-first-out (FIFO) so that all previous messages are processed before the channel is actually closed.
- The actor above terminates after printing the first message, therefore the channel is closed. Trying to send another message to it causes a `ClosedSendChannelException`. Thus, calling `actor.close` is actually redundant in Listing 6.49. This is not typical behavior for an actor; a more real-world way to implement an actor is shown in Listing 6.50, which prints “Hello” on one line and then “World” on the next.

**Listing 6.50 Actor that Keeps Reading from its Channel**

[Click here to view code image](#)

```
import kotlinx.coroutines.channels.actor  
  
import kotlinx.coroutines.runBlocking  
  
// This actor keeps reading from its channel indefinitely  
  
val actor = actor<String> {
```

```
    for (value in channel) { println(value) }

}

runBlocking {
    actor.send("Hello") // Makes actor print out "Hello"
    actor.send("World") // Makes actor print out "World"
    actor.close()
}
```

The crucial part is the **for** loop, which keeps the actor alive until its channel is closed. Now, you can send an arbitrary number of messages to the actor and then finally call **close** when the actor is no longer needed.

The **actor** function returns a **SendChannel** so that you are only allowed to *send* messages to the actor but not receive messages from the channel (only the actor should do that).

Inside the actor lambda, the **channel** has type

**Channel<E>**, which implements both **SendChannel<E>** and **ReceiveChannel<E>** so that you can call both **send** and **receive** inside the lambda. However, when sending to itself, an actor must make sure the channel is not full.

Otherwise, it will suspend indefinitely while waiting for a **receive** and effectively produce a deadlock. So you should check if **channel.isFull** beforehand, and you must also make the channel buffer at least one element—this is easily achieved by setting the capacity as shown later in this section.

**Note**

Kotlin's channels are *fair* in the sense that they follow the FIFO pattern, meaning elements that are sent first are also received first.

As a coroutine builder, the **actor** function is very similar to the ones you have already seen. It accepts a **CoroutineContext**, a **CoroutineStart**, an explicit

parent `Job`, and the suspending lambda that defines its behavior. These work the same way as before. A lazily started actor becomes active once an element is first sent to its channel.

In addition to these parameters, actors have an additional capacity that defines how many elements its channel can buffer. By default, the capacity is zero, meaning that `send` must suspend until a `receive` is issued, and vice versa. This is called a `RendezvousChannel` because the sending and receiving sides must “meet” at one point in time. In [Listing 6.51](#), the `receive` is implicitly called in the `for` loop and suspends until a new element is sent to the channel.

Another useful type of channel is a conflated channel that only keeps the last element sent to it. This is often the desired behavior, for instance, to update a UI only with the most recent data.

**Listing 6.51 Actor with Conflated Channel**

[Click here to view code image](#)

```
import kotlinx.coroutines.channels.*  
  
import kotlinx.coroutines.*  
  
  
runBlocking {  
  
    val actor = actor<String>(capacity = Channel.CONFLATED)  
  
    for (value in channel) { println(value) }  
  
}  
  
  
actor.send("Hello") // Will be overwritten by following  
actor.send("World") // Overwrites Hello if it was  
delay(500)  
actor.close()  
}
```

This example sets the capacity parameter to `Channel.CONFLATED` so that `send` no longer suspends until a `receive` is issued. Instead, the new value just overrides the old one so that only the most recent value is retrieved with `receive`. Note that, without the delay, the code will likely not print anything because then the close token is the last one when the actor gets to receive a message for the first time.

Apart from these, you can use `Channel.UNLIMITED` to create a channel with unlimited capacity (using a `LinkedListChannel`) or use `actor(capacity = n)` to use a channel with a specific buffer size `n` (using `ArrayChannel`). As you can imagine, the former uses a linked list to buffer elements, whereas the latter uses an array. With unlimited capacity, the sender never suspends.

**Tip**

I recommend reading the source code documentation in the Kotlin standard library. It provides detailed info about most entities. `Ctrl+Q` (`Ctrl+J` on Mac) to show quick docs and `Ctrl+B` (`Cmd+B` on Mac) to jump to declaration are your best friends for this in Android Studio and IntelliJ.

Jumping to declaration has the added benefit that you can explore the actual implementation. For instance, you'll notice that all coroutine builders are implemented in a very similar way.

Next, you can create actor coroutines that use multiple channels and receive their next message from whichever channel first has an element. Listing 6.52 implements an actor that reads numbers from two channels simultaneously using a `select` expression (as in Go). It explicitly creates the two channels, therefore there's no need for the `actor` coroutine builder. Any coroutine that selects from these two channels qualifies as an actor here.

Because `receive` only allows reading from one channel at a time, this code uses the `onReceive` clause. There are other such clauses available for use inside a `select` such as `onSend`, `onAwait`, `onLock`, `onTimeout`, and `onReceiveOrNull`. Internally, all these implement a

`SelectClause` interface, and they can be combined arbitrarily inside a `select` to do something with the value that is first available—whether it's a deferred result, a timeout, a lock acquired, or a new message on a channel. In other words, whatever event occurs first will be “selected.”

**Listing 6.52 Actor with Multiple Channels**

[Click here to view code image](#)

```
import kotlinx.coroutines.channels.*  
  
import kotlinx.coroutines.*  
  
import kotlinx.coroutines.selects.select  
  
  
runBlocking {  
  
    val channel1 = Channel<Int>()  
  
    val channel2 = Channel<Int>()  
  
  
    launch { // No need for the actor coroutine builder  
  
        while (true) {  
  
            select<Unit> { // Provide any number of alternatives  
  
                channel1.onReceive { println("From channel 1: $it") }  
  
                channel2.onReceive { println("From channel 2: $it") }  
  
            }  
  
        }  
  
        channel1.send(17) // Sends 17 to channel 1, thus triggering the first branch  
  
        channel2.send(42) // Sends 42 to channel 2, causing the second branch to trigger  
  
        channel1.close(); channel2.close()  
  
    }  
  
    A horizontal scroll bar with a grey track and a white slider, indicating the code block is scrollable.  
◀ ▶
```

**Note**

The `select` function is biased to the first clause, meaning that if multiple clauses are selectable at the same time, the first one will be selected. You can use this intentionally, for instance, to send messages to a primary channel while it's not full and to a secondary channel otherwise.

In case you explicitly don't want this behavior, you can use `selectUnbiased` instead. This randomly shuffles the clauses each time before selecting, making the selection fair.

Similarly, you can make multiple actors send to the same channel, as in [Listing 6.53](#). Here, the used channel is simply a local variable so that it is accessible by all actors. Three producers are launched, all of which try sending a message to the channel repeatedly and suspend until there is a receiver (this is a *rendezvous channel*). On the consumer side, `consumeEach` can be used as an alternative to an explicit `for` loop. Multiple coroutines that *read* from the same channel can be implemented the same way. You just have to provide access to the same channel object.

**Listing 6.53 Multiple Actors on Same Channels**

[Click here to view code image](#)

```
import kotlinx.coroutines.channels.*  
  
import kotlinx.coroutines.*  
  
  
runBlocking {  
  
    val channel = Channel<String>()  
  
    repeat(3) { n ->  
  
        launch {  
  
            while (true) {  
  
                channel.send("Message from actor $n")  
  
            }  
  
        }  
  
        channel.take(10).consumeEach { println(it) }  
    }  
}
```

```
    channel.close()  
}
```

In the previous examples, there are several coroutines that don't behave like actors in the classical sense because they are not associated with an incoming channel but rather with an outgoing channel. We shall call these *producers*. For instance, Listing 6.53 creates three producers. Intuitively, producers are the counterparts to actors because they have an *outgoing* channel attached.

Kotlin provides a more convenient way to create such producers, shown in Listing 6.54. Here, the `produce` coroutine builder creates a producer. Accordingly, it returns a `ReceiveChannel` because you're only supposed to receive the producer's messages from the outside, whereas sending is left to the producer.

**Listing 6.54 Creating a Producer**

[Click here to view code image](#)

```
import kotlinx.coroutines.*  
  
import kotlinx.coroutines.channels.*  
  
  
runBlocking {  
  
    val producer = produce {  
  
        var next = 1  
  
        while (true) {  
  
            send(next)  
  
            next *= 2  
  
        }  
  
    }  
  
    producer.take(10).consumeEach { println(it) }  
  
}
```



Note that this producer emits a potentially infinite stream of values. It is only decelerated by the fact that it uses a `RendezvousChannel` so that `send` suspends until there's a receiver on the other end. Without the `take(10)` restriction, the program would keep printing powers of two until the integer overflows.

To use more interesting message types than just `Int` or `String`, you can either use any other predefined type or define a custom message type. Sealed classes are a good fit for this use case to create a restricted class hierarchy of possible message types, as done in Listing 6.55.

**Listing 6.55 Actor with Custom Message Type**

[Click here to view code image](#)

---

```
sealed class Transaction // Uses a .kt file to be at
    data class Deposit(val amount: Int) : Transaction()
    data class Withdrawal(val amount: Int) : Transaction()

    fun newAccount(startBalance: Int) = actor<Transaction> {
        var balance = startBalance
        for (tx in channel) {
            when (tx) {
                is Deposit    -> { balance += tx.amount; print]
                is Withdrawal -> { balance -= tx.amount; print]
            }
        }
    }

    fun main(args: Array<String>) = runBlocking<Unit> {
        val bankAccount = newAccount(1000)
```

```
bankAccount.send(Deposit(500))

bankAccount.send(Withdrawal(1700))

bankAccount.send(Deposit(4400))

}
```

Here, the actor acts as a bank account that receives transactions as messages. A transaction can either be a deposit or a withdrawal, as defined by the sealed class. The account can buffer up to ten transactions; further transactions are suspended until the account has a chance to catch up.

**Note**

Although coroutines and the actor model may seem like modern concepts, they are not. The term *coroutine* goes back all the way to 1963 and was first used by Melvin E. Conway to refer to a generalization of a subroutine that can yield control and resume later.<sup>12</sup> Similarly, as mentioned, the actor model was introduced by Carl Hewitt back in 1973.<sup>13</sup>

<sup>11.</sup> In fact, you can easily simulate them using `fun go(block: suspend () -> Unit) = CommonPool.runParallel(block)`

<sup>12.</sup> <http://www.melconway.com/Home/pdf/compiler.pdf>

<sup>13.</sup> [https://www.researchgate.net/scientific-contributions/7400545\\_Carl\\_Hewitt](https://www.researchgate.net/scientific-contributions/7400545_Carl_Hewitt)

Remember that Kotlin does not claim to be innovative but rather an amalgamation of wellknown and proven concepts that are suitable to develop large-scale software. What is unique about coroutines, however, is how they allow writing asynchronous code in sequential style.

## Concurrency Styles

In order to recap the differences between coroutine-based, callback-based, and future-based concurrency, it is a useful exercise to explore how to transform between them. This way, it becomes clear that all these are ultimately just fancy callbacks, and if you understand just one of these approaches and the transformation between them, you understand them all.

[Listing 6.56](#) demonstrates this transformation via three function signatures that represent the same purpose—asynchronously fetching an image.

### **Listing 6.56 Transformation between Concurrency Styles**

[Click here to view code image](#)

```
// Callbacks

fun fetchImageAsync(callback: (Image) -> Unit) { ... }

// Futures

fun fetchImageAsync(): Future<Image> { ... }

// Coroutines

suspend fun fetchImage(): Image { ... }
```

In the case of futures, the callback is defined separately, typically via combinators like `thenCompose` or `thenAccept`, but it does still use callbacks. These define what happens after the value becomes available. In the case of coroutines, Kotlin hides the callback (more precisely, the continuation) but the compiler passes an additional `Continuation` parameter to every suspending function. This continuation represents which operations are left to do after suspension—in other words, it’s a callback. Intuitively, at every suspension point, the corresponding continuation contains the code that follows the suspension point in the function—because that is what’s left to do after resuming at that suspension point. As an example, refer back to [Listing 6.7](#), in which the suspension points are highlighted.

## Coroutines in Practice

To facilitate getting started with coroutines in your project and to overcome the likely roadblocks, this section discusses practical issues like debugging, interoperability with Java, and interoperability with asynchrony libraries.

## Debugging Coroutines

Concurrency is hard, and debugging concurrent code can be excruciating. To ease the pain, this section discusses practices for debugging coroutines. However, also keep in mind that concurrency becomes a lot more predictable when avoiding shared mutable state and using immutable data structures. Consider these practices if you find yourself debugging your concurrent code a lot due to synchronization issues.

One well-known way to monitor program execution is logging. In concurrent code, it's a good idea to add the name of the current thread to the log so that each logged operation can be assigned to its corresponding thread, as is done in [Listing 6.57](#). However, there may be thousands of coroutines running on the same thread so that context is not sufficient. Luckily, if you use the compiler flag -

`Dkotlinx.coroutines.debug`, Kotlin automatically appends the current coroutine's ID to the thread name. That way, a log item produced in [Listing 6.57](#) would start with `[main @coroutine#1]` instead of just `[main]`.

### [Listing 6.57 Logging Current Thread \(and Coroutine\)](#)

[Click here to view code image](#)

---

```
fun log(message: String) = println("[${Thread.current
```



The name suffix always has the form “`@name#id`” and uses the `CoroutineName` and `CoroutineId` from the coroutine context. You can set a custom `CoroutineName`, but `CoroutineId` is a private class used by Kotlin. The default name is simply “`coroutine`” and the ID is incremented by one for each newly created coroutine. To pass the debug flag automatically when running your application in IntelliJ, you can create a run configuration (Ctrl+Shift+A / Cmd+Shift+A, and type in “edit config”) with - `Dkotlinx.coroutines.debug` flag added as a VM option.

In Android Studio, you cannot create such a run configuration. Instead, you can create an `Application` class that sets up the system property as in Listing 6.58. This class is automatically instantiated before any activity and thus suitable to set up such configuration. It is generally used to contain the global state of an Android app. To wire it into your project, you must specify it as the `android:name` inside the `<application>` in your manifest file.

**Listing 6.58 Enabling Coroutine Debugging on Android**

[Click here to view code image](#)

```
class App : Application() {  
  
    override fun onCreate() {  
        super.onCreate()  
        System.setProperty("kotlinx.coroutines.debug",  
            if (BuildConfig.DEBUG) "on" else "off")  
    }  
}  
  
// In AndroidManifest.xml  
  
<application  
    android:name=".App" ...>  
    </application>
```

Besides this particular feature, you are free to use debugging facilities as usual in coroutine code: breakpoints, variable watchers, thread overview, frames, and more. When placing breakpoints in IntelliJ or Android Studio, you can decide whether that breakpoint should stop just the thread that reaches it or all threads. This is important to keep in mind if a breakpoint doesn't behave as you would expect.

## Wrapping Asynchronous APIs

Imagine you want to work with a future implementation for which there are no wrappers available yet—and you want to use them idiomatically in Kotlin. For this, you can write your own coroutine builder and custom await function. This way, you basically map from futures to the world of suspending functions. To dissect the structure of a coroutine builder, consider the implementation of the `future` coroutine builder that wraps Java 8's `CompletableFuture`. Listing 6.59 shows a simplified version of its implementation.

**Listing 6.59 Implementation of `future` Coroutine Builder (Simplified)**

[Click here to view code image](#)

```
public fun <T> future(...): CompletableFuture<T> {  
    // Set up coroutine context... (omitted)  
  
    val future = CompletableFutureCoroutine<T>(newConte  
        job.cancelFutureOnCompletion(future)  
  
        future.whenComplete { _, exception -> job.cancel(e)  
            start(lambda, receiver=future, completion=future)  
  
            return future  
        }  
  
    private class CompletableFutureCoroutine<T>(...) : ... {  
        // ...  
  
        override fun resume(value: T) { complete(value) }  
  
        override fun resumeWithException(exception: Throwabl  
            completeExceptionally(exception)  
        }  
    }  
}
```

---

The `future` coroutine builder first sets up the coroutine context for the new coroutine. Then it creates a new coroutine of the specialized type `CompletableFutureCoroutine`. It also tells the coroutine to cancel the future on completion—this forwards cancellation of the coroutine to the underlying future. After that, it sets things up the other way around so that the coroutine job is cancelled when the future completes. With this, the future is ready to go so the function starts the coroutine by running the given lambda with the newly created future as its *receiver type* and as its *completion*. For this, it invokes the given `CoroutineStart` object with the block of code, the receiver, and the completion.

Concerning the receiver, you may be wondering how the receiver type of the lambda can be changed even though you implement the lambda with `CoroutineScope` as its receiver. This works because

`CompletableFutureCoroutine` also implements `CoroutineScope` so that this remains a valid receiver.

Regarding the `completion`: This is the continuation that is invoked when the coroutine completes. Here it becomes very apparent that both futures and continuations are essentially just callbacks, and therefore map well onto each other.

Specifically, `complete` maps to `resume` and `completeExceptionally` maps to `resumeWithException`. All these methods define what's supposed to happen when the asynchronous call finishes—in other words, when the supplied lambda terminates.

In this case, what happens is that either `CompletableFutureCoroutine.complete` is invoked with the result value or `completeExceptionally` is invoked if the lambda produced an exception. This populates the future with the computed result (or exception) and it makes that result available to the outside. The coroutine itself is the `CompletableFuture` that is returned. For example, a user may hold a reference to the returned future, which is internally

a `CompletableFutureCoroutine`, but they only see it as a `CompletableFuture` due to the return type. So when the user calls `join` on the future, it waits until either `complete` or `completeExceptionally` is called—which is the same as waiting for the coroutine to finish.

Adding your custom `await` function for the future is simpler because no new coroutine has to be created. The main task is to map the functions appropriately, similar to the way it is done in the `CompletableFutureCoroutine`. Listing 6.60 shows the basic structure of a custom `await` function. The key point here is `suspendCancelledCoroutine`, which is a low-level primitive to suspend execution. This captures the current state in a `Continuation` object and passes that continuation to the lambda block. This is what allows the code to resume at a later point—all local state is captured in the continuation. As always, the continuation also defines what happens after the suspension. This is expressed in the lambda; accordingly, its type is `(Continuation<T> -> Unit)`.

In Listing 6.60, the continuation waits for the future to complete and resumes the coroutine normally or with exception, depending on whether an exception occurred. If you look at the actual implementation of Kotlin’s `CompletableFuture.await`, you will see that it performs additional optimizations. But this listing presents the basic structure to be followed by any custom `await` function. The key takeaway is that every future typically implements a normal and a failure case, which are mapped to `resume` and `resumeWithException`, respectively.

#### **Listing 6.60 Custom Await Function (Simplified)**

[Click here to view code image](#)

---

```
import kotlinx.coroutines.future.future
import kotlinx.coroutines.*
import java.util.concurrent.CompletableFuture
```

```

suspend fun <T> CompletableFuture<T>.myAwait(): T {
    return suspendCancellableCoroutine { continuation ->
        whenComplete { value, exception ->
            if (exception == null) continuation.resume(value)
            else continuation.resumeWithException(exception)
        }
    }
}

runBlocking {
    val completable = future { delay(1000); 42 }

    println(completable.myAwait()) // 42
}

val fail = future<Int> { throw Exception("Could not")
    println(fail.myAwait()) // Throws exception
}

```

As an example, you can use the shown approach in order to work with Retrofit's `Call<T>` in an idiomatic way, meaning you could create a `call { ... }` coroutine builder with a custom await function defined as an extension on `Call<T>`. However, even for futures without an official wrapper, you can often find wrappers implemented by the community.

**Note**

Coroutines themselves only use heap resources and no native resources. State is captured in continuation objects, which are automatically garbage-collected when there are no more references to them. This facilitates resource management with coroutines. You only have to make sure to close resources that you explicitly opened with a coroutine (or in the case that the coroutine gets cancelled).

## Interoperability

If you try calling a suspending function from Java, it turns out to not be so easy. Why? Remember that the compiler adds an additional `Continuation` parameter to every suspending function and that it provides this automatically in Kotlin code. However, in Java, you would have to pass a continuation yourself, which is hardly realistic and should never be done because it is highly prone to errors and produces unreadable code. Instead, add an asynchronous implementation for every suspending function that should be callable from Java, as done in [Listing 6.61](#).

### **Listing 6.61 Wrapping Suspending Function for Java Interop**

[Click here to view code image](#)

```
import kotlinx.coroutines.future.future

suspend fun fetchScore(): Int { ... } // This can hard

fun fetchScoreAsync() = future { fetchScore() } // 1

// Java

CompletableFuture<Integer> future = fetchScoreAsync()

int asyncResult = future.join();
```

Remember that the `future` coroutine builder is from the `kotlinx-coroutines-jdk8` library. Because it returns a `CompletableFuture`, the result can be processed in Java as usual. `async` is not suitable for this because it returns a `Deferred` with its suspending `await` function that again requires a `Continuation`. This is why coroutines in general are hardly usable from Java—coroutines become powerful only in conjunction with suspending functions, and those can barely be called from Java. This goes back to the fact that

suspending functions really are a Kotlin concept and there is no equivalent to map them to in Java.

## Under the Hood

I briefly mentioned that suspending functions are a pure compiler feature. So, from the compiler's view, what is the difference between a normal function and a function with **suspend** modifier? How is it treated differently to realize this world of suspending functions you've explored in this chapter? It is easiest to grasp the internal mechanisms by looking at a concrete example. So, the following discussion is based on the suspending function in Listing 6.62.

**Listing 6.62** Suspending Function to be Examined

[Click here to view code image](#)

```
suspend fun suspending(): Int {  
  
    delay(1000)      // Suspension point #1  
  
    val a = 17  
  
    otherSuspend()  // Suspension point #2  
  
    println(a)  
  
  
    return 0  
}
```

As you know, the compiler implicitly adds a continuation parameter to every suspending function. Thus, the compiled signature in the Java bytecode looks as in Listing 6.63. Note that this is a transformation to CPS because the continuation is now passed explicitly (imagine this transformation applied to all suspending functions).

**Listing 6.63** Step 1: Adding Continuation Parameter

[Click here to view code image](#)

```
suspend fun suspending(c: Continuation<Int>): ... { ... }
```

Note that the original return type is now captured as the generic type parameter of the continuation, here

`Continuation<Int>`. This indicates the “return type” of the continuation. More precisely, it becomes the type accepted by its `resume` function, which in turn passes along its value to the last suspension point. The function name `resume` indicates the intuition: It is the value that the program *resumes* with after the continuation finishes. You can think of this behavior as a `return` value of the continuation if this intuition helps you.

This transformation enables the next step: removing the `suspend` modifier as in [Listing 6.64](#). Suspension is handled *inside* the method by compiler-generated code as discussed in Step #4 ([Listing 6.66](#)), and resumption is explicit with the passed continuation.

#### [Listing 6.64 Step 2: Removing suspend Modifier](#)

[Click here to view code image](#)

```
fun suspending(c: Continuation<Int>): ... { ... }
```

Next, the return type changes. Irrespective of the original return type, the compiler changes the return type to `Any?` as in [Listing 6.65](#). This return type actually represents the union of the original return type and the special `COROUTINE_SUSPENDED` type (but Kotlin has no proper union types<sup>14</sup>). Using the original return type allows the suspending function to not suspend and instead to return a result directly (for example, if an asynchronous result is already available). In contrast, `COROUTINE_SUSPENDED` indicates that the function did indeed suspend execution.

<sup>14</sup>. Interesting discussion about union types: <https://discuss.kotlinlang.org/t/union-types/77>

#### [Listing 6.65 Step 3: Changing Return Type to Any?](#)

[Click here to view code image](#)

```
fun suspending(c: Continuation<Int>): Any? { ... }
```

---

At this point, you may be wondering what the compiler passes in as a continuation. Basically, the continuation represents the rest of the execution. For instance, Listing 6.62 produces three continuations.

- The initial continuation represents the entire function with its five lines of code.
- The continuation for `delay` (at suspension point #1) contains the four subsequent lines because that represents the rest of the computation.
- At suspension point #2, the continuation contains only the two lines of code that remain after it. Additionally, it captures the current value of `a` so that, upon resumption, the value is still available. This is why you can also think of continuations as objects that capture the state of a coroutine at a suspension point.

In general, you get one continuation for every suspension point, plus one for the **suspend** function itself (thus three in this case). Note that this example demonstrates that, in direct-style sequential code, continuations are implicitly defined by the sequence of code lines—without explicitly passing around continuations.

The question becomes how the compiler provides the correct continuations for all three suspending calls. One solution would be to transform the code into a callback-based style similar to Listing 6.1 at the beginning of this chapter. However, this is not how it is done. Instead, the compiler generates a state machine for each suspending function. In this state machine, each suspension point is a state, and state transition is achieved by executing the code until the next suspension point. This state machine allows the compiler to pass the right continuation to every suspending function. To do so, it creates a coroutine object that implements `Continuation` and stores both the program context (like local variables) and the current state of the state machine. This is then used as the continuation for every suspending call. One advantage of this is that the state machine object can be reused for all suspending calls in the function. So it requires only one state machine object to be instantiated and is therefore efficient. Listing 6.66 shows this approach in a slightly simplified version; the actual implementation also handles cancellation and other cases.

**Listing 6.66 Step 4: State Machine (Simplified)**

[Click here to view code image](#)

```
fun suspending(c: Continuation<Int>): Any? {  
  
    val stateMachine = object : CoroutineImpl(...) // In  
  
        when (stateMachine.state) { // Three cases for  
            is 0 -> {  
                stateMachine.state = 1 // Updates current  
                delay(1000, stateMachine) // Passes state machine  
            }  
            is 1 -> {  
                val a = 17  
  
                stateMachine.my$a = a // Captures variable  
                stateMachine.state = 2 // Updates current  
                otherSuspend(stateMachine) // Passes state machine  
            }  
            is 2 -> {  
                val a = stateMachine.my$a // Recovers value  
  
                println(a)  
  
                return 0  
            }  
        }  
    }  
}
```

At first, the `stateMachine` object represents the initial continuation—the one that contains the whole function body. Thus, it starts off in state zero and executes until the first

suspension point. Before it reaches that suspension point, it updates the current state to reflect the progress and to resume with the correct continuation. So after `delay`, the continuation resumes in state one, thus proceeding to set the value of `a`. Right after every such assignment of a variable that is required again later, the value is captured in the state machine. Similarly, before the variable `a` is used in state two, its value is restored from the continuation.

With this, you now have a firm understanding of the inner workings of coroutines, in addition to the practical knowledge of how to use them. Although it is not required to use coroutines in practice, a good conceptual model of the foundations is useful to understand how things actually work and therefore how to resolve issues that come up during development.

## SUMMARY

In this chapter, you dove deep into Kotlin’s coroutines, the principles on which they are based, their use cases, and how they are implemented. I consider this to be a more advanced topic and reckon that it is important to recap the main points. I would also like to highlight that using coroutines is ultimately very convenient compared with other asynchrony approaches. The main takeaways that you should remember are the following.

- Concurrency is not the same as parallelism.
- Coroutines are conceptually very lightweight threads—but they’re not really threads; you may have millions of coroutines per thread.
- Kotlin coroutines are generic enough to cover the use cases of `async-await` and `futures`, generators, parallelism, lazy sequences, actors, Go-like channels and `select`, and the producer-consumer pattern—all in a natural way.
- With coroutines, you can write asynchronous code in familiar sequential style including all control flow constructs, and with natural function signatures.
- Coroutines are powered by suspending functions. These are functions with the additional superpower to suspend execution at well-defined points (called suspension points).
- Coroutine builders are how you enter the world of suspending functions. They create a new coroutine that executes a given suspending lambda.

- `runBlocking` is used for main and test functions to bridge the gap between normal and suspending worlds.
- `launch` is used for “fire-and-forget” tasks without return value.
- `async` is used for asynchronous tasks with a return value to await.
- `withContext` allows switching between coroutine contexts and can be a good alternative to `async` for calls that return a result.
- `future` is an interoperability coroutine builder for Java 8’s `CompletableFuture`.
- `actor` is used to create a coroutine with an attached incoming channel.
- `produce` is used to create a coroutine with an attached outgoing channel.
- `buildSequence` and `buildIterator` are used to create generators.
- There are libraries for Android and other UI frameworks to facilitate concurrency.
- Continuations are essentially callbacks but can also store the state of a coroutine at a suspension point.
- Coroutines are only attached to a specific thread during execution. While suspended, they are independent of any threads and may use a different thread when resumed.

Note that Kotlin’s focus on immutability is especially crucial for concurrent code. Synchronization problems arise from shared *mutable* state, not shared *immutable* state. Similarly, *private* mutable state is unproblematic. Since coroutines themselves execute sequentially, you can use mutable data structures within a coroutine without evoking synchronization issues. Only when *sharing* state between coroutines must you use immutable or concurrent data structures to avoid synchronization problems.

Coroutines are a powerful tool. Investing the time to understand how they work pays off in reduced development time and more stable asynchronous code. Even without understanding all the internal workings of coroutines, they are convenient to use in practice and allow for more natural code.

This chapter introduced some of the inner workings as well because it helps to take part in community discussions, understanding the further development, and potential issues in

your code. Covering the topic in full would require a whole book, but this chapter gives you the tools you need to understand and use coroutines effectively. In the next two chapters, you will put coroutines into practice while developing two Android apps.

||  
Kotlin on Android

## Android App Development with Kotlin: Kudoo App

*Whether you think you can, or you think you can't—you're right.*

Henry Ford

This chapter first explains how to set up Kotlin projects for Android and proceeds to implement a simple to-do list app called Kudoo. This app uses Android Architecture Components, a `RecyclerView`, and other fundamental Android components so that you can learn how to use these with Kotlin.

### SETTING UP KOTLIN FOR ANDROID

This section guides you through the required setup for your first Kotlin Android app. You'll learn what changes are made to the Gradle build configuration files and how you can use Android Studio effectively to set up Android projects using Kotlin.

#### Using Kotlin in Android Studio

Kotlin comes preinstalled since Android Studio version 3, in accordance with Google's statement to officially support Kotlin as a language on Android. For you, this means there's no need to install the Kotlin plugin separately anymore.

## Auto-Generated Gradle Configuration

Before diving into the Gradle configuration, notice that your Android project has two different Gradle build scripts, both called `build.gradle`. To follow along in this chapter and [Chapter 8, Android App Development with Kotlin: Nutrilicious](#), you must be able to differentiate between the two scripts.

- First, there is the project's `build.gradle` file. This one resides in the project's root directory. In Android Studio, you can find it in the Android project view that is located on the left-hand side of the IDE by default.
- More important, the module's `build.gradle` file is the one you will be changing frequently in the following two chapters. It's located inside the `app` directory under the project root. In Android Studio's project view, which shows all Gradle scripts in one place, you can find it under the module's `build.gradle` file, as shown in [Figure 7.1](#).

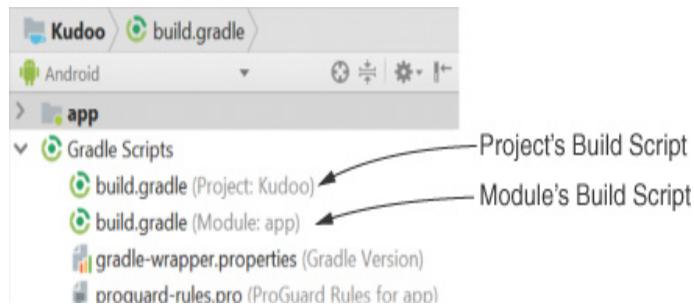


Figure 7.1 Gradle build scripts

When creating a new project in Android Studio 3, Kotlin is automatically configured and activated. For existing Java projects, you can invoke Android Studio's action *Configure Kotlin in Project* in order to perform the necessary changes.

First, it adds the `kotlin-gradle-plugin` dependency to your project's Gradle build file. This plugin is responsible for compiling your Kotlin sources. With this, the project's build file has a structure as in [Listing 7.1](#).

### **Listing 7.1 Project's build.gradle File**

[Click here to view code image](#)

```
buildscript {  
  
    ext.kotlin_version = "1.2.60" // Adjust to your cl  
  
    // ...  
  
    dependencies {
```

```
// ...  
  
    classpath "org.jetbrains.kotlin:kotlin-gradle-plu  
  
    }  
  
}  
  
// ...  
  
◀ ▶
```

Second, it includes the Kotlin standard library in your module's Gradle build file to make it available inside that module. Third, it applies the required plugin `kotlin-android` to target Android during build.

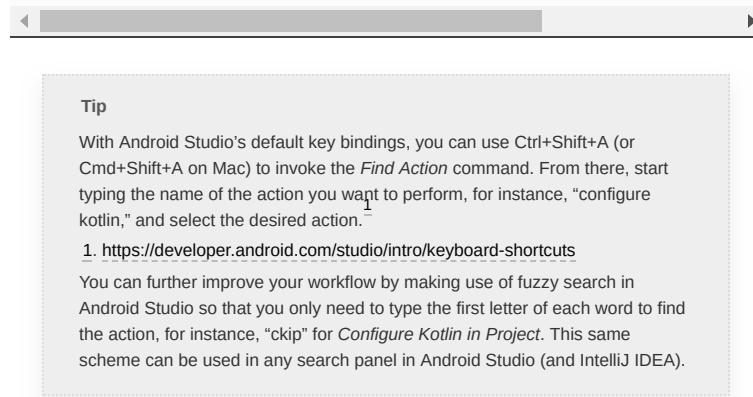
In the current version of Android Studio 3, the plugin `kotlin-android-extensions` is also included by default, even though it is not *required* to build a Kotlin project targeting Android. However, the Kotlin Android Extensions are extremely useful, so I recommend you use them, and they will be used in both apps presented in this book. The module's resulting Gradle build file should look similar to [Listing 7.2](#).

#### **Listing 7.2** Module's `build.gradle` File

[Click here to view code image](#)

---

```
apply plugin "com.android.application"  
  
apply plugin "kotlin-android"  
  
apply plugin "kotlin-android-extensions" // Recommen  
  
...  
  
dependencies {  
  
    ...  
  
    implementation "org.jetbrains.kotlin:kotlin-stdli  
  
    }  
  
}
```



## Adapting Your Gradle Configuration

If you want to separate your Kotlin files into a directory, such as `src/main/kotlin`, you'll need to add the code from [Listing 7.3](#) to your module's `build.gradle` file to make this directory known to Android Studio for indexing. Alternately, you can place your Kotlin files alongside your Java files under `src/main/java`, which is what we do in this book.

### **Listing 7.3 Add Separate Kotlin Sources Directory to Gradle (Optional)**

[Click here to view code image](#)

```
android {  
    ...  
  
    sourceSets {  
        main.java.srcDirs += "src/main/kotlin"  
    }  
}
```

In case you don't need to target JDK 6 with your app but rather JDK 7 or 8, you can use the corresponding specialized dependency shown in [Listing 7.4](#) instead of the one in [Listing 7.2](#). These add additional extensions for APIs introduced in JDK 7 or 8. The targeted JDK version can be changed under *Kotlin Compiler* in the Android Studio settings.

#### Listing 7.4 Dependencies for Targeting JDK 7 or 8

[Click here to view code image](#)

```
implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:1.3.72"
implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk8:1.3.72"
```

Similarly, if you want to use Kotlin's reflection or testing APIs, you must add the appropriate dependencies from [Listing 7.5](#).

#### Listing 7.5 Dependencies for Kotlin Reflection and Test Libraries

[Click here to view code image](#)

```
implementation "org.jetbrains.kotlin:kotlin-reflect:1.3.72"
testImplementation "org.jetbrains.kotlin:kotlin-test:1.3.72"
testImplementation "org.jetbrains.kotlin:kotlin-test-junit:1.3.72"
```

Other dependencies can be added in the same way. But this covers all essential Kotlin-related dependencies you may want to use for Android.

## Using Annotation Processors

If you use libraries that contain annotation processors, like Dagger for dependency injection, the Kotlin plugin must be configured to make them work with Kotlin classes. As shown in [Listing 7.6](#), you need to replace the `android-apt` plugin for annotation processing with Kotlin's `kapt` plugin.

#### Listing 7.6 Configuring kapt for Annotation Processing (Example: Dagger 2)

[Click here to view code image](#)

```
apply plugin: 'kotlin-kapt' // Enables kapt for annotation processing
// ...
dependencies {
    implementation 'com.google.dagger:dagger:2.17'
```

```
    kapt 'com.google.dagger:dagger-compiler:2.17'    //
```

```
}
```

The first step is to enable the plugin `kotlin-kapt`, which replaces `android-apt` for annotation processing—it will handle annotations on Java files as well, so any dependencies on `apt` can be removed. After that, dependencies can be added to the `kapt` configuration, such as the `dagger-compiler` in [Listing 7.6](#), while normal dependencies are still added as usual.

To process annotations in `test` and `androidTest` sources, you can use the more specialized `kaptTest` and `kaptAndroidTest` configurations to add the required dependencies. Any dependencies from the `kapt` configuration are inherited to these two, however, so there is no need to include them twice—dependencies added using `kapt` are available in both test and production code.

### Converting Java Code to Kotlin

If you’re not starting with a fresh Kotlin project but want to mix Kotlin into an existing Java project, you can do so by first invoking the *Configure Kotlin in Project* action as above. After that, you can make use of the Java-to-Kotlin converter to transpile an existing Java file to a Kotlin file. To do this, you can click *Convert Java File to Kotlin File* under the *Code* menu. Alternately, you can use `Ctrl+Shift+A` (or `Cmd+Shift+A` on Mac) and type “`cjtk`” (for *Convert Java to Kotlin*) to fuzzy-search for that same action and invoke it.

Be aware that you may want to keep a copy of the original Java file, especially if you’re doing this with a production app. Ideally, I recommend you start introducing Kotlin into an existing app by converting non-business-critical parts of your app, or better yet, start by migrating a pet project. [Chapter 10](#), Migrating to Kotlin, covers processes and best practices to migrate Java apps to Kotlin.

**Note**

The quality of autoconverted Kotlin code may not be production-ready. For instance, it won't make use of Kotlin's powerful language features where appropriate. So, although the converter is good for getting used to the basic syntax and features of Kotlin, it will not open up all the possibilities that the language offers you.

In short, Android Studio 3 has strong support for Kotlin and avoids the need for any manual configuration for basic Kotlin projects. You can use Kotlin out of the box and integrate it into existing Java apps. In all this, Android Studio supports your workflow with actions you can easily invoke that, for instance, adjust the Gradle build or convert Java files to Kotlin.

## APP #1: KUDO, A TO-DO LIST APP

It's time to write your first app with Kotlin: Kudoo. This simple to-do list app lets users create to-do items and check them off when they're completed. Although this app is basic, it already introduces several fundamental Android components and Kotlin language features. Most important are the following:

- **Android Architecture Components** to split the app into loosely coupled layers
- **Recycler Views** to improve the list view performance
- **Kotlin Android Extensions** to omit `findViewById`
- **Intents** to switch between activities and share data
- **Coroutines** for database operations

Figure 7.2 shows the finished app.

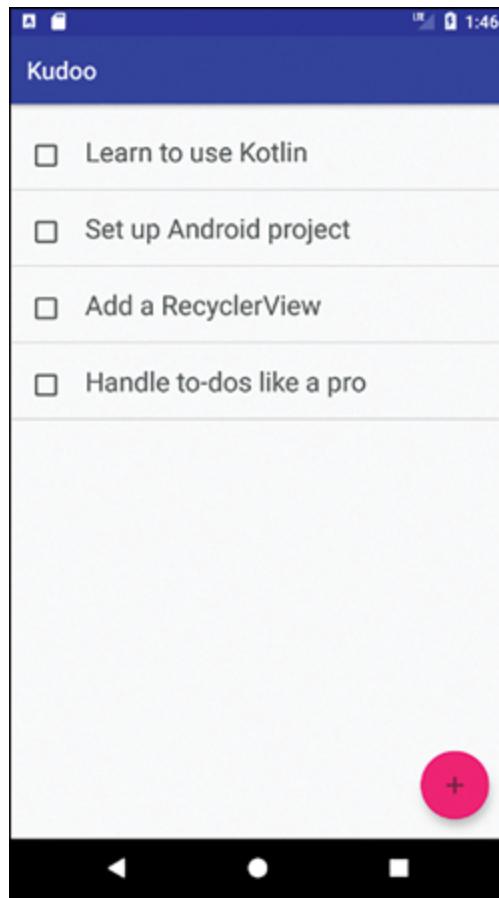


Figure 7.2 The finished to-do app you will create in this chapter

Let's start by creating a new project with a basic activity.

### Creating the Project

Under the *File* menu, select *New* and *New Project*. Name the app Kudoo, as shown in Figure 7.3, and use any unique company domain; `example.com` will do if you don't have one as you won't be publishing this app to the Play Store. Finally, select *Include Kotlin support*.

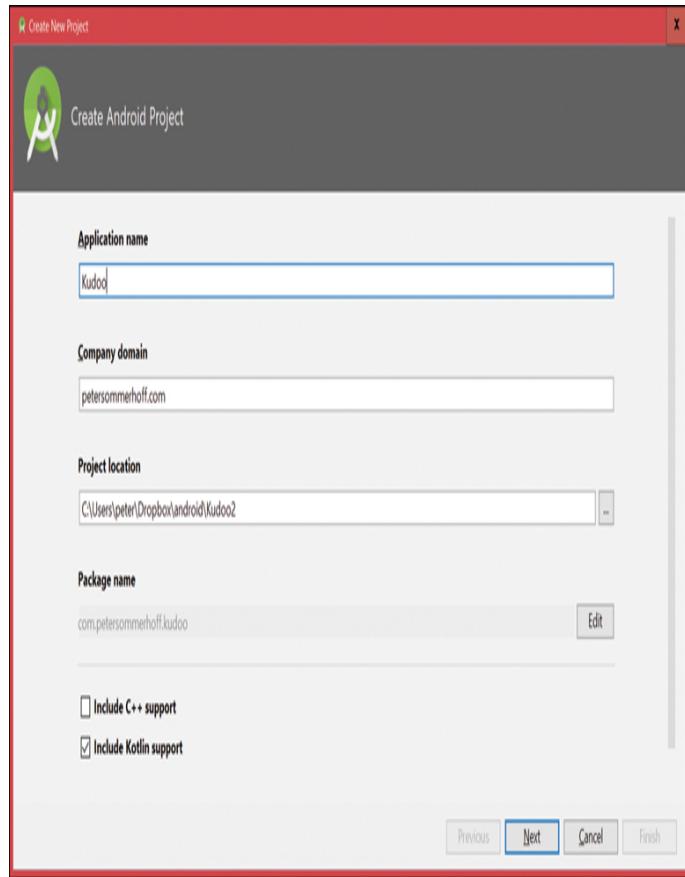


Figure 7.3 Create Android Project screen

**Note**

In case the following screenshots do not exactly match the most current version of Android Studio that you're using, please follow the up-to-date instructions from the [Android Developers website](#) to create a project with a *Basic Activity* and Kotlin support.

2. <https://developer.android.com/studio/projects/create-project>

After clicking *Next*, you can keep *Phone & Tablet* selected with the default minimum SDK (API 19 at the time of writing). Click *Next* again and select *Basic Activity* (not *Empty Activity*) as the template. Click *Next*, and keep the defaults in order to generate a `MainActivity.kt` file along with its layout file (see Figure 7.4). Click *Finish*, and Android Studio will set up and build your project.

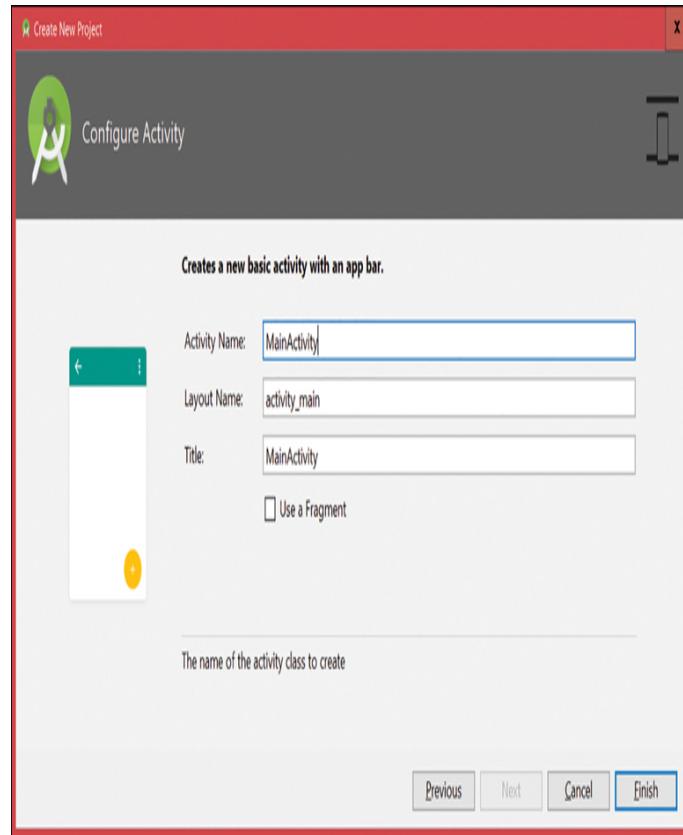


Figure 7.4 Create a `MainActivity` with its layout

Once it's done, make sure that the module's `build.gradle` file is set up correctly. You can easily navigate to it using `Ctrl+Shift+N` (`Cmd+Shift+O` on Mac) and typing in `"build.gradle"`, then selecting the one from the `app` module, or by using the Android project view as was shown in [Figure 7.1](#). You should see the following two lines in there:

[Click here to view code image](#)

```
apply plugin: 'kotlin-android'  
apply plugin: 'kotlin-android-extensions'
```

You now have a barebones app that should run fine when you launch it (using the green arrow button or `Shift+F10/Ctrl+R` on Mac). This is a good time to try running it to make sure everything's working as expected and to set up an Android Virtual Device (AVD)<sup>3</sup> if you don't have one yet.

3. Set up a virtual device: <https://developer.android.com/studio/run/managing-avds>

To finish up the project template, you can remove the menu because you won't need that for this simple app. To do that, remove the methods `onCreateOptionsMenu` and `onOptionsItemSelected` from `MainActivity.kt` and remove the resources folder `res/menu`. With this, you're all set to start writing the app.

**Note**

If you want to use Git while working on this app, you can visit [gitignore.io](https://gitignore.io) to generate the content for the `.gitignore` file. I recommend using one configured for Android, Android Studio, Kotlin, and Java.<sup>4</sup>

4. <https://www.gitignore.io/api/java%2Ckotlin%2Candroid%2Candroidstudio>

Android Studio creates a `.gitignore` file automatically, you can use `Ctrl+Shift+N` (`Cmd+ Shift+O` on Mac) and type in `".gitignore"` to find it (it doesn't show up in the *Android* project view), then choose the one in the root directory, the one with the `(Kudo)` suffix. Replace all its contents with the ones from [gitignore.io](https://gitignore.io). To run `git init`, the integrated Terminal comes in handy; you can open that using `Alt+F12` from inside Android Studio.

The code for the Kudo app is on GitHub<sup>5</sup> and the repository has a directory for each working state of the app, corresponding to the results after each section in this book. So if you get an error, you can always compare to the directory corresponding to the section you are reading. Definitely do write the apps yourself while reading this chapter and [Chapter 8](#); I guarantee you will learn a lot more this way.

5. <https://github.com/petersommerhoff/kudo-app>

## Adding the RecyclerView

The app's central element will be the list of to-do items, implemented using a `RecyclerView`. A `RecyclerView` is a list view that avoids lag even for very long lists by reusing the view objects required to display the list. A good way to start out with this app is to implement the basic `RecyclerView` and feed it some sample data to show. Here's an overview of the involved steps to achieve this (which are covered in this section).

1. First, adjust and create all the required layouts in XML.
2. Create a simple to-do item class that represents the data shown in each list item.
3. Implement the `RecyclerView` adapter that feeds the recycler view with data.

4. Set up the `RecyclerView` in the `MainActivity`.

## Setting up the Recycler View Layouts

Because you chose *Basic Activity* during project setup, there is an `activity_main.xml` layout file inside the `res/layout` directory. This layout includes the `content_main.xml` layout. Inside the `content_main.xml`, replace the `TextView` with a `RecyclerView` so that the layout file looks as it does in [Listing 7.7](#).

**Listing 7.7** RecyclerView Layout

[Click here to view code image](#)

```
<android.support.constraint.ConstraintLayout  
    app:layout_behavior="@string/appbar_scrolling_view_behavior"  
  
    <android.support.v7.widget.RecyclerView  
        android:id="@+id/recyclerViewTodos"  
        android:scrollbars="vertical"  
        android:layout_width="match_parent"  
        android:layout_height="match_parent" />  
    </android.support.constraint.ConstraintLayout>
```

With this setup, the `RecyclerView` takes up all the height and width of its container so that it effectively fills the whole screen. Note that the `ConstraintLayout` that was generated for you enables scrolling if the list overflows the screen via its `layout_behavior` attribute.

**Note**

If the RecyclerView cannot be found, make sure you have the Design Support Library in your module's dependencies:

[Click here to view code image](#)

```
implementation 'com.android.support:design:27.1.1'
```

Next, let's create the layout that each list item in the **RecyclerView** will use. This comprises a **TextView** to show the to-do list title with a **CheckBox** beside it to check it off. Under **res/layout**, add a new layout resource file called **todo\_item.xml**. A simple **LinearLayout** as in [Listing 7.8](#) will do; add it to the newly created layout file.

[Listing 7.8 RecyclerView Item Layout](#)

[Click here to view code image](#)

```
<?xml version="1.0" encoding="utf-8"?>

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

    <CheckBox
        android:id="@+id/cbTodoDone"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="@dimen/margin_medium" />

    <TextView
        android:id="@+id/tvTodoTitle"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

```

```
        android:layout_height="wrap_content"  
  
        android:padding="@dimen/padding_large"  
  
        android:textSize="@dimen/font_large" />  
  
  
    </LinearLayout>
```

To add the missing dimension resources (the ones starting with `@dimen`), press Alt+Enter with the cursor inside them and click the suggested action *Create dimen value resource*. The `margin_medium` and `padding_large` should be `16dp`, `font_large` should be `22sp`. Alternately, you can add these resources to `res/values/dimens.xml`<sup>6</sup> by hand.

6. [https://github.com/petersommerhoff/kudoo-app/blob/master/01\\_AddingRecyclerView/app/src/main/res/values/dimens.xml](https://github.com/petersommerhoff/kudoo-app/blob/master/01_AddingRecyclerView/app/src/main/res/values/dimens.xml)

#### Note

Android uses several types of resources. Dimensions are one of them; others include strings, layouts, and drawables. All reside in the `res` directory, and simple values like dimensions and strings are inside the `res/values` subdirectory. Looking at the files inside `res/values`, you will see they all share the same basic structure. You can add resources into these files manually as well, instead of using Android Studio's actions to create them for you.

All resources are accessed programmatically via the generated `R` class, such as `R.string.enter_todo` or `R.layout.activity_main`.

That's all the required layout for now, so you can now dive into the actual Kotlin code, beginning with the model.

## Model

Models represent the entities used in your app. The only model this app needs is one to represent a to-do item, and Kotlin's data classes greatly simplify its declaration, as Listing 7.9 shows. Place this class into a new `model` package directly under the `kudoo` package.

[Listing 7.9 TodoItem Data Class](#)

[Click here to view code image](#)

```
data class TodoItem(val title: String)
```

---

This simple to-do item model that carries only a title is all you need as models for this app.

### The RecyclerView Adapter

The main work when using `RecyclerView` is implementing the adapter. The adapter provides the data for the `RecyclerView` (the list items) and handles how the item views are reused. This is how the `RecyclerView` improves performance: It reuses existing views from a so-called *view holder* to avoid creating a new object and more importantly to avoid inflating the layouts for these views. In contrast, a normal `ListView` would create and inflate dedicated views for every list item, without any reuse.

The first step is to add a new package `view` with subpackage `main` (you can do this in one step in Android Studio by typing in `view.main` as the package name in the package creation dialog) and to add a new Kotlin class `RecyclerListAdapter` inside it, as in [Listing 7.10](#). In total, the package should now be `com.example.kudoo.view.main`, assuming you used `example.com` as your company domain during project creation.

#### [Listing 7.10 RecyclerListAdapter Signature](#)

[Click here to view code image](#)

---

```
import android.support.v7.widget.RecyclerView

import com.example.kudoo.model.TodoItem

class RecyclerListAdapter(
    private val items: MutableList<TodoItem>
) : RecyclerView.Adapter<RecyclerListAdapter.ViewHolder> {
    // ...
}
```



Because the `RecyclerView` will show to-do items and users may add or remove items on the list, the adapter carries a `MutableList<TodoItem>`, which represents the data that will be shown in the `RecyclerView`.

This class extends Android's `RecyclerView.Adapter` and thus has to override three methods:

1. `onCreateViewHolder`: This method creates a `ViewHolder` object, meaning an object that is used to hold all the views for a single list item. In this case, one `TextView` and one `CheckBox`. As mentioned, the point is that the recycler view then reuses these view objects to avoid unnecessary object creations and expensive layout inflations.
2. `onBindViewHolder`: This method binds a given `TodoItem` to such a `ViewHolder`, meaning that it populates the views with the data from the `TodoItem`. Here, it shows the to-do title in the text view.
3. `getItemCount`: This must return the number of items to be shown in the list.

It's common practice to add the custom `ViewHolder` class that implements `RecyclerView.ViewHolder` as a nested class into the adapter. This view holder class holds all views for a single list item (the text view and the check box) and knows how to bind a `TodoItem` to them. Listing 7.11 shows the `ViewHolder` for this app.

**Listing 7.11 Custom ViewHolder**

[Click here to view code image](#)

---

```
import android.support.v7.widget.RecyclerView

import android.view.View

import android.widget.*

import com.example.kudoo.R

import com.example.kudoo.model.TodoItem


class RecyclerListAdapter(...) : ... {

    // ...

    class ViewHolder(listItemView: View) : RecyclerView
```

```
// ViewHolder stores all views it needs (only call  
  
    val tvTodoTitle: TextView = listItemView.findViewById(R.id.todo_title)  
  
    val cbTodoDone: CheckBox = listItemView.findViewById(R.id.todo_done)  
  
    fun bindItem(todoItem: TodoItem) {    // Binds a todo item to the list item view  
        tvTodoTitle.text = todoItem.title    // Populates the title  
        cbTodoDone.isChecked = false    // To-do items are not checked by default  
    }  
}  
}
```

As you can see, the `ViewHolder` caches all its views and only calls `findViewById` once when initialized. This is part of how a `RecyclerView` improves performance compared to the old `ListView`: it reuses existing view objects and populates them with the desired data in `bindItem`, without doing any expensive operations.

You must use `findViewById` here because the `ViewHolder` is not a `LayoutContainer`. If you want, you can change this using the *Kotlin Android Extensions* by enabling experimental features (where *experimental* does not mean *unstable*). To do so, simply add the code from [Listing 7.12](#) to the module's `build.gradle` file and sync the project. You can place it below the `dependencies { ... }` section.

#### **Listing 7.12 Enabling Experimental Kotlin Android Extensions**

[Click here to view code image](#)

```
androidExtensions {  
    experimental = true  
}
```

With this, you can now get rid of `findViewById` in the `ViewHolder` by implementing the `LayoutContainer` interface that's now available, as shown in [Listing 7.13](#). All that's required to implement this interface is to override the `containerView` property, which is what was called `listItemView` in [Listing 7.11](#). Thus, you can override it directly in the constructor parameter. Then, you can access the UI elements directly by the ID you gave them in the XML layout file, here `tvTodoTitle` and `cbTodoDone`. Being able to access UI elements like this, without explicitly calling `findViewById`, is one of the popular benefits of Kotlin on Android and is enabled by the Kotlin Android Extensions.

**Listing 7.13** ViewHolder Without `findViewById`

[Click here to view code image](#)

```
// ... (imports from before here)

import kotlinx.android.extensions.LayoutContainer

import kotlinx.android.synthetic.main.main.todo_item.* //



class RecyclerListAdapter(...) : ... {

    // ...

    class ViewHolder(

        override val containerView: View // Overrides p
    ) : RecyclerView.ViewHolder(containerView), LayoutC

        fun bindItem(todoItem: TodoItem) {

            tvTodoTitle.text = todoItem.title // Still ca
            cbTodoDone.isChecked = false
        }
    }
}
```

---

Note that the `LayoutContainer` caches all views as well, although it is not directly apparent. You can see that this is the case by looking at the decompiled Java code—remember, you can use Ctrl+Shift+A (Cmd+Shift+A), then type “Show Kotlin Bytecode” or simply “skb” and then click *Decompile*.

Alternately, you can use the *Tools* menu; under *Kotlin* there’s the option *Show Kotlin Bytecode*.

With this, all that’s left to do is to override the three methods mentioned above in the adapter. First, `onCreateViewHolder` must create a `ViewHolder` object and should use the list item layout (`todo_item.xml`) you’ve created to inflate the view’s layout. This is commonly done using a `LayoutInflater`, as shown in Listing 7.14.

**Listing 7.14** `RecyclerListAdapter.onCreateViewHolder()`

[Click here to view code image](#)

---

```
// ... (imports from before)

import android.view.LayoutInflater

import android.view.ViewGroup

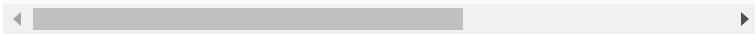
class RecyclerListAdapter(...) : ... {

    // ...

    override fun onCreateViewHolder(parent: ViewGroup,
        val itemView: View = LayoutInflater.from(parent.c
            .inflate(R.layout.todo_item, parent, false)

        return ViewHolder(itemView)

    }
}
```



This first creates a new view (a list item) by inflating the list item layout. It does so without attaching the view to any parent yet because the third argument of `inflate` is set to `false`.

Then it passes that to a new `ViewHolder` that manages this view from then on and will let the recycler view reuse it later.

Next up is `onBindViewHolder`, which should bind a given `TodoItem` to a `ViewHolder`. The logic for this is already implemented in `ViewHolder.bindItem` so that you can delegate to that method, as done in [Listing 7.15](#). Place this into the `RecyclerListAdapter` class, like `onCreateViewHolder`.

[Listing 7.15 RecyclerListAdapter.onBindViewHolder\(\)](#)

[Click here to view code image](#)

```
override fun onBindViewHolder(holder: ViewHolder, pos
    holder.bindItem(items[position]) // Populates the ] }


```

Lastly, `getItemCount` is the easiest to implement since the `RecyclerView` should render as many items as the list of to-do items given to its adapter. [Listing 7.16](#) implements this method. This also goes into the `RecyclerListAdapter` class.

[Listing 7.16 RecyclerListAdapter.getItemCount\(\)](#)

[Click here to view code image](#)

```
override fun getItemCount() = items.size
```

Putting everything together, the `RecyclerListAdapter` looks as it does in [Listing 7.17](#).

[Listing 7.17 Complete RecyclerListAdapter](#)

[Click here to view code image](#)

```
import android.support.v7.widget.RecyclerView
import android.view.*
import com.example.kudoo.R
import com.example.kudoo.model.TodoItem
```

```
import kotlinx.android.extensions.LayoutContainer

import kotlinx.android.synthetic.main.todo_item.*


class RecyclerListAdapter(
    private val items: MutableList<TodoItem>
) : RecyclerView.Adapter<RecyclerListAdapter.ViewHolder> {

    override fun onCreateViewHolder(parent: ViewGroup,
        val itemView: View = LayoutInflater.from(parent.context)
            .inflate(R.layout.todo_item, parent, false)
        return ViewHolder(itemView)
    }

    override fun getItemCount() = items.size

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        holder.bindItem(items[position])
    }

    class ViewHolder(
        override val containerView: View
    ) : RecyclerView.ViewHolder(containerView), LayoutContainer {

        fun bindItem(todoItem: TodoItem) {
            tvTodoTitle.text = todoItem.title
            cbTodoDone.isChecked = false
        }
    }
}
```

```
}
```

```
}
```

## The MainActivity

With the `RecyclerView` ready to go, now you only need to set it up in the `MainActivity` and populate it with some sample data. Thanks to Kotlin Android Extensions, the `RecyclerView` can be accessed directly by its layout ID, `recyclerViewTodos`, so that you can again avoid `findViewById`. Listing 7.18 shows the setup logic.

**Listing 7.18 Setting Up the RecyclerView with the Adapter**

[Click here to view code image](#)

```
// ... (imports from before)

import android.support.v7.widget.*

import com.example.kudoo.model.TodoItem

import com.example.kudoo.view.main.RecyclerListAdapter

import kotlinx.android.synthetic.main.activity_main.*

import kotlinx.android.synthetic.main.content_main.*


class MainActivity : AppCompatActivity() {

    // ...

    private fun setUpRecyclerView() = with(recyclerViewTodos) {
        adapter = RecyclerListAdapter(sampleData()) // F
        layoutManager = LinearLayoutManager(this@MainActivity)
        itemAnimator = DefaultItemAnimator() // C
        addItemDecoration(
            DividerItemDecoration(this@MainActivity, DividerItemDecoration.VERTICAL))
    }
}
```

```
private fun sampleData() = mutableListOf(  
    TodoItem("Implement RecyclerView"),  
    TodoItem("Store to-dos in database"),  
    TodoItem("Delete to-dos on click")  
)  
}
```

In `setUpRecyclerView`, the adapter is assigned to an instance of your `RecyclerListAdapter`, with sample data passed in as the list of to-do items that should be displayed. Next, a simple `LinearLayoutManager` is used to lay out the items as a vertical list. As an optional bonus, an item animator is added to improve the delete animation and a divider item decorator adds a separator line between each item. Note how this code makes use of the `with` function and shorthand function syntax.

**Note**

Beware not to import `R.id.recyclerViewTodos` instead of the synthetic property even when it cannot be found. If Android Studio marks these references in red, rebuild the project using `Ctrl+F9` (`Cmd+F9` on Mac) or by running the app.

In autocompletions, Android Studio marks the correct imports with an *(Android Extensions)* suffix. They're from packages like `kotlinx.android.synthetic.main.content_main.*`.

Now, you just need to call `setUpRecyclerView` in `onCreate` and you should see the three sample to-do items displayed in your app. This is shown in [Listing 7.19](#).

**Listing 7.19 Adjusting `onCreate`**

[Click here to view code image](#)

---

```
override fun onCreate(savedInstanceState: Bundle?) {  
    // ...  
    setUpRecyclerView()
```

```
}
```

The most central component is now ready, but the data is hard-coded in the `MainActivity`. In the next step, you'll use Room to retrieve sample data from a SQLite database instead.

## Adding a Room Database

With this section, you'll start integrating Android Architecture Components into this app. Introduced at Google I/O 2017, this library of architecture components has rapidly gained widespread use. Room is a component that handles database access and greatly simplifies the use of SQLite. The `ViewModel` and `LiveData` architecture components will be integrated afterward.

To be able to use Room, you must add the dependencies from [Listing 7.20](#) to your module's `build.gradle` file (you can also extract dependency versions into the project's `build.gradle` file if you prefer).

### **Listing 7.20 Gradle Dependencies for Room**

[Click here to view code image](#)

```
dependencies {  
    // ...  
  
    def room_version = "1.1.1" // Use latest version  
    implementation "android.arch.persistence.room:run  
    kapt "android.arch.persistence.room:compiler:$roc  
}  
▶
```

Remember to use `kapt` instead of `annotationProcessor` when using Kotlin, and add the corresponding plugin at the top of the `build.gradle` file, as in [Listing 7.21](#).

### **Listing 7.21 Enabling the Kotlin Annotation Processor Plugin**

[Click here to view code image](#)

```
apply plugin: 'kotlin-android-extensions' // Should  
apply plugin: 'kotlin-kapt' // Added recently
```

Room makes it easy to store models to a database, but first you'll have to tell Room which models to store. For this app, only the `TodoItem` class should be mapped to the database. To let Room know, you must annotate it with `@Entity`. In addition, a `@PrimaryKey` is required to uniquely identify each to-do item in the database. Listing 7.22 shows the adjusted model.

**Listing 7.22** `TodoItem` as an Entity

[Click here to view code image](#)

```
import android.arch.persistence.room.Entity  
import android.arch.persistence.room.PrimaryKey  
  
@Entity(tableName = "todos") // Indicates that this is a database table  
data class TodoItem(val title: String) {  
    @PrimaryKey(autoGenerate = true) // Unique primary key  
    var id: Long = 0 // 0 is considered not set  
}
```

Inside `@Entity`, you can specify a custom name for the associated database table; here, it will be `todos`. An additional `id` serves as the primary key. Room will autogenerated these IDs for you when setting `autoGenerate = true`; it does so by simply incrementing it by one for each new record. Note that the `id` is initialized to zero because Room will consider zero as being not set, thus allowing it to set it to the autogenerated value.

With only these few lines of code, Room has all the information it needs to map `TodoItem` objects to a database

table. What's next is to access that table using a *data access object* (DAO)—this is your access point for all database operations and will be generated by Room as well. All you have to do is define an interface with the operations and queries you want to use. This is shown in [Listing 7.23](#). You can place this `TodoItemDao` class in a new `db` package, directly under the `kudoo` package.

**Listing 7.23** `TodoItemDao` for Database Access

[Click here to view code image](#)

```
import android.arch.persistence.room.*  
  
import android.arch.persistence.room.OnConflictStrategy  
  
import com.example.kudoo.model.TodoItem  
  
  
@Dao  
  
interface TodoItemDao {  
  
    @Query("SELECT * FROM todos")  
    fun loadAllTodos(): List<TodoItem> // Allows retrieval  
  
    @Insert(onConflict = IGNORE) // Does nothing if exists  
    fun insertTodo(todo: TodoItem) // Allows insertion  
  
    @Delete  
    fun deleteTodo(todo: TodoItem) // Allows deletion  
}
```



By annotating the interface with `@Dao`, you let Room know to generate its implementation. Inside the DAO, you can use `@Query`, `@Insert`, `@Update`, and `@Delete`. The latter three need no further setup. For `@Insert` and `@Update`, you

may set a strategy for the on-conflict case, which defines how Room behaves if an element with the same ID already exists. `@Query` allows you to implement arbitrary queries on your database. The only query you need here is one to load all to-do items. Room validates your queries at compile-time and Android Studio analyzes them instantly as well, giving you a very fast feedback loop.

You're almost done implementing the database. The last step is to implement a `RoomDatabase`. This is done using an abstract class that extends `RoomDatabase` and is annotated with `@Database`. Also, it should provide an instance of itself to the outside. In [Listing 7.24](#), `AppDatabase` fills that role. This also belongs into the `db` package.

[Listing 7.24](#) `AppDatabase`

[Click here to view code image](#)

```
import android.arch.persistence.room.*  
  
import android.content.Context // Needs access to Ar  
  
import com.example.kudoo.model.TodoItem  
  
  
@Database(entities = [TodoItem::class], version = 1)  
  
abstract class AppDatabase : RoomDatabase() {  
  
    companion object {  
  
        private var INSTANCE: AppDatabase? = null  
  
  
        fun getDatabase(ctx: Context): AppDatabase {  
  
            if (INSTANCE == null) {  
  
                INSTANCE = Room.databaseBuilder(ctx, AppData  
                    .build()  
  
            }  
        }  
    }  
}
```

```
        return INSTANCE!!  
    }  
}
```

```
    abstract fun todoItemDao(): TodoItemDao // Trigger  
}
```

The `@Database` annotation requires all entities the database should contain as well as a version number. Whenever the schema changes, you must increase this version number.

Recall that companion object members work like static members in languages like Java. The `AppDatabase` caches an instance of itself in a private `INSTANCE` property that is initialized lazily when first accessed. This initialization uses Room's database builder to build an implementation of the abstract `AppDatabase`. Lastly, add abstract methods for any DAOs you want to expose for accessing this database—here, you only need the `TodoItemDao`.

With this, the database will be set up correctly and is in principle ready to be used in the `MainActivity`. However, there's no sample data in it. The clean way to prepopulate a Room database is to add a callback when instantiating it. Room's callbacks allow you to override an `onCreate` method, which is exactly what you need to add sample data when the database is created. Listing 7.25 shows the adjusted code for the companion object. You will only need this code temporarily until users can create their own to-do items.

#### **Listing 7.25 Populating the AppDatabase with Sample Data**

[Click here to view code image](#)

```
// ... (imports from before)  
  
import android.arch.persistence.db.SupportSQLiteData  
  
import kotlinx.coroutines.experimental.*
```

```
val DB = newSingleThreadContext("DB") // CoroutineContext

@Database(entities = [TodoItem::class], version = 1)
abstract class AppDatabase : RoomDatabase() {

    companion object {

        private var INSTANCE: AppDatabase? = null

        fun getDatabase(ctx: Context): AppDatabase {
            if (INSTANCE == null) {
                INSTANCE = Room.databaseBuilder(ctx, AppDatabase::class.java, "app-db")
                    .addCallback(prepopulateCallback(ctx))
                    .build()
            }
            return INSTANCE!!
        }

        private fun prepopulateCallback(ctx: Context): Callback {
            return object : Callback() {
                override fun onCreate(db: SupportSQLiteDatabase) {
                    super.onCreate(db)
                    populateWithSampleData(ctx)
                }
            }
        }
    }
}
```

```
private fun populateWithSampleData(ctx: Context)

    launch(DB) { // DB operations must be done on
        with(getDatabase(ctx).todoItemDao()) { // Use the
            insertTodo(TodoItem("Create entity"))
            insertTodo(TodoItem("Add a DAO for data acc
            insertTodo(TodoItem("Inherit from RoomData
        }
    }
}

abstract fun todoItemDao(): TodoItemDao
}
```

Here, you extend Room's `Callback` to override its `onCreate` method and insert the sample data. Inserting data is a database operation and therefore must be performed on a background thread. As you can see, the code uses `launch { ... }` to perform the database operations in the background. It uses a dedicated single-thread context for database operations because `CommonPool` is intended for CPU-bound operations. To make this code work, you need to include the coroutine dependencies in your module's `build.gradle` file, as in [Listing 7.26](#).

#### Listing 7.26 Gradle Dependencies for Kotlin Coroutines

[Click here to view code image](#)

---

```
def coroutines_version = "0.24.0" // Use latest vers
implementation "org.jetbrains.kotlinx:kotlinx-corouti
```

```
implementation "org.jetbrains.kotlinx:kotlinx-corouti
```

```
◀ ━━━━━━ ▶
```

With this, the database is finished and will even be populated with sample data when first created. All that's left to do is use it in `MainActivity`, as shown in Listing 7.27.

**Listing 7.27 Using the Database from MainActivity**

[Click here to view code image](#)

```
// ... (imports from before)

import kotlinx.coroutines.experimental.android.UI

import kotlinx.coroutines.experimental.*

import com.example.kudoo.db.*

class MainActivity : AppCompatActivity() {

    private lateinit var db: AppDatabase // Stores an

    override fun onCreate(savedInstanceState: Bundle?) {
        // ...
        db = AppDatabase.getDatabase(applicationContext)
        setUpRecyclerView() // Sets up recycler view *after* onCreate
        // ...
    }

    private fun setUpRecyclerView() = with(recyclerView)
        .launch {
            val todos = sampleData().toMutableList()
            withContext(UI) { adapter = RecyclerViewAdapter(todos) }
        }
}
```

```
layoutManager = LinearLayoutManager(this@MainActivity)

itemAnimator = DefaultItemAnimator()

addItemDecoration(
    DividerItemDecoration(this@MainActivity, DividerItemDecoration.VERTICAL)
)

private suspend fun sampleData() =
    withContext(DB) { db.todoItemDao().loadAllTodos() }
```

For now, the `MainActivity` holds a reference to the `AppDatabase` to access the database via the DAO. This reference can only be initialized inside `onCreate` once the application context is available, so it uses a late-initialized property.

As a database operation, the actual call to `loadAllTodos` must be performed in the background. For this, it uses `withContext(DB) { ... }` to run it on the dedicated database dispatcher and retrieve a result. Due to the `withContext` call, `sampleData` must be a **suspend** function, so its call is wrapped inside a `launch { ... }` in `setUpRecyclerView`.

That's it! You've now set up a simple Room database with Kotlin, written sample data into the database, and retrieved it back to show in the UI. You can run this app now to see the sample data from the database.

#### Troubleshooting

If something went wrong trying to prepopulate the database the first time, you can delete it in order to trigger `onCreate` again. To do so, use Android Studio's *Device File Explorer* to remove the directory `data/data/com.example.kudoo/databases`.

Also, at the time of writing, *Apply Changes* can cause issues in combination with coroutines. If you get an error mentioning "CoroutineImpl.label is inaccessible," try re-running the app normally without *Apply Changes*.

As the next step, you will introduce a `ViewModel` to avoid the direct dependency on `AppDatabase` in `MainActivity`.

## Using a `ViewModel`

A view model is an Android Architecture Component that holds the data for an associated activity. There are several benefits to this approach:

- Activities only need to know of their view model(s) to get all the data they need, unaware of whether that data comes from a cache, a database, a network call, or another data source. In other words, it decouples the activity from the data source.
- Android's `ViewModel` is lifecycle-aware, meaning that it automatically preserves data across configuration changes such as screen rotations. This way, data doesn't need to be reloaded after each configuration change.
- Activities should not perform asynchronous calls themselves because these may potentially take a long time and the activity has to manage them to avoid memory leaks. Separating this concern into its own class is therefore cleaner and avoids huge activity classes that try to perform all app logic themselves.

Let us reap these benefits by using Android's `ViewModel` class in the Kudoo app. The first step is to add the required dependencies shown in [Listing 7.28](#). These already include `LiveData` as well, which you'll incorporate in the next step.

[Listing 7.28 Gradle Dependencies for `ViewModel` \(and `LiveData`\)](#)

[Click here to view code image](#)

```
dependencies {  
    // ...  
  
    def lifecycle_version = "1.1.1"  // Replace with latest version  
    implementation "android.arch.lifecycle:extensions:$lifecycle_version"  
    kapt "android.arch.lifecycle:compiler:$lifecycle_version"  
}
```

Next, add a new package `viewmodel` (under `kudoo`) and add a new `TodoViewModel` class, which will be the view

model for `MainActivity`. A view model should extend either the `ViewModel` or the `AndroidViewModel` class—the latter is required if the `ViewModel` requires an application context. Thus, the `TodoViewModel` will extend `AndroidViewModel` to be able to construct an `AppDatabase` because the database needs the application context. So you get the class header shown in [Listing 7.29](#).

**Listing 7.29 Class Header of TodoViewModel**

[Click here to view code image](#)

```
import android.app.Application

import android.arch.lifecycle.AndroidViewModel

class TodoViewModel(app: Application) : AndroidViewM
```

Every subclass of `AndroidViewModel` must accept an `Application` object in its constructor and pass it along to its parent. This is how the view model becomes aware of the application context. This view model wraps the database and provides a clean API for the `MainActivity` to use. [Listing 7.30](#) introduces the required members.

**Listing 7.30 Complete TodoViewModel**

[Click here to view code image](#)

```
// ... (imports from before)

import com.example.kudoo.db.*

import com.example.kudoo.model.TodoItem

import kotlinx.coroutines.experimental.*

class TodoViewModel(app: Application) : AndroidViewM

private val dao by lazy { AppDatabase.getDatabase(c

suspend fun getTodos(): MutableList<TodoItem> = wit
```

```
        dao.loadAllTodos().toMutableList()

    }

    fun add(todo: TodoItem) = launch(DB) { dao.insertTodo(todo) }

    fun delete(todo: TodoItem) = launch(DB) { dao.deleteTodo(todo) }

}
```



This view model lazily requests a database instance when it's accessed for the first time. It provides a suspending function to retrieve all to-do items from the database. Additionally, it exposes methods to add and delete to-do items, which launch the corresponding database operations in the background.

With this, you can now replace the `AppDatabase` in `MainActivity` with the new `TodoViewModel`. To this end, remove the `AppDatabase` property and add the `TodoViewModel` instead, as shown in [Listing 7.31](#).

**Listing 7.31 Integrating the TodoViewModel into MainActivity**

[Click here to view code image](#)

```
class MainActivity : AppCompatActivity() {

    private lateinit var viewModel: TodoViewModel // ...

    override fun onCreate(savedInstanceState: Bundle?) {
        // ...
        viewModel = getViewModel(TodoViewModel::class) // ...
        setUpRecyclerView()
    }

    private fun setUpRecyclerView() = with(recyclerView) {
        launch(UI) { adapter = RecyclerListAdapter(viewModel) }
    }
}
```

```
// ...  
}  
}  
◀ ▶
```

The view model is again late-initialized because it requires the activity to be attached to the application—and this is the case in `onCreate`. You can remove the `sampleData` method and simply pass in `viewModel.getTodos()` to the `RecyclerView`'s adapter. Because this performs a database operation, it is wrapped inside `launch`. In this regard, the `MainActivity` is not yet completely independent of the implementation details of the `ViewModel`—but we'll fix this using `LiveData` in the next step.

For now, you have to add the `getViewModel` extension function to make this code compile. Remember that extensions are your best friend on Android to work around API boilerplate. Here, you use it to retrieve view models more easily. Create a new package `view.common` and add a new file `ViewExtensions.kt` to it. Inside this file, you can define the extension shown in Listing 7.32 as a file-level function.

#### **Listing 7.32 Extension to Retrieve View Models**

[Click here to view code image](#)

---

```
import android.arch.lifecycle.*  
  
import android.support.v4.app.FragmentActivity  
  
import kotlin.reflect.KClass  
  
  
fun <T : ViewModel> FragmentActivity.getViewModel(mod  
    ViewModelProviders.of(this).get(modelClass.java)  
◀ ▶
```

It's an extension on the `FragmentActivity` class that simply accepts a `KClass<T>` (a Kotlin class) where `T` must

be a `ViewModel`. In this way, it provides a more natural API to retrieve your view models. You must import it into your `MainActivity` to resolve the remaining error.

**Tip**

Kotlin's extension functions are extraordinarily useful to avoid repeating boilerplate around Android APIs. For a collection of extensions that help you write concise and expressive code, and that is maintained by Google themselves, check out `Android KTX`.

7. <https://developer.android.com/kotlin/ktx>

At the time of writing, `Android KTX` is still in alpha and APIs are prone to change. So to make sure you can follow along these apps seamlessly, they are not used here. When you're reading this, `Android KTX` may be stable, and if so, I recommend exploring the extensions.

You have now integrated Android's `ViewModel` into your app, preserving your data across configuration changes and helping to separate concerns between activities and data-handling code—the activity should only be responsible for *showing* the data and providing notifications of user actions. However, so far the to-do items retrieved from the view model don't automatically reflect changes to the data. You can verify this by calling `viewModel.add(...)` after some delay—the new item will not yet show up in the UI. To handle this idiomatically with Android Architecture Components, let's integrate `LiveData` next.

## Integrating `LiveData`

`LiveData` is a lifecycle-aware data holder. App components like activities and fragments can observe a `LiveData` object to automatically reflect data changes in the UI. Because it's lifecycle aware, `LiveData` makes sure to notify only active observers. For instance, it doesn't update activities that are currently in the background or that have been destroyed by Android to recover memory. Like `ViewModel`, this has several benefits.

- Activities don't have to handle lifecycles, they can simply observe a `LiveData`, which makes sure not to send data to inactive consumers (which would crash the app).
- Data is automatically kept up to date whenever the activity is active. For instance, after configuration change, the activity will immediately

receive the latest data.

- `LiveData` makes all its observers perform cleanups when their associated lifecycle is destroyed so that memory leaks are prevented.

Integrating `LiveData` into the app is fairly simple because it works well with Room and `ViewModel` out of the box. As a first step, the DAO should return a `LiveData` instead of just a `List<TodoItem>` so that it can be observed for data changes. Fortunately, Room can do this for you automatically; just wrap the return value into `LiveData` as in [Listing 7.33](#) and Room will perform the required transformation.

#### [Listing 7.33 Returning LiveData from the DAO](#)

[Click here to view code image](#)

```
// ... (imports from before)

import android.arch.lifecycle.LiveData

@Dao

interface TodoItemDao {

    // ...

    @Query("SELECT * FROM todos")

    fun loadAllTodos(): LiveData<List<TodoItem>> // Wrapping the return value in LiveData

}
```



Next, you'll have to adjust `TodoViewModel.getTodos` accordingly, as done in [Listing 7.34](#).

#### [Listing 7.34 Returning LiveData from the ViewModel](#)

[Click here to view code image](#)

```
// ... (imports from before)

import android.arch.lifecycle.LiveData

class TodoViewModel(app: Application) : AndroidViewModel {

    // ...
}
```

```
// Now uses a LiveData of a read-only list

suspend fun getTodos(): LiveData<List<TodoItem>> =  
    dao.loadAllTodos()  
}  
// ...  
}
```

Now you're ready to observe the `LiveData` in `MainActivity`, as shown in Listing 7.35.

**Listing 7.35 Observing the `LiveData` from `MainActivity`**

[Click here to view code image](#)

```
// ... (imports from before)

import android.arch.lifecycle.LiveData
import kotlinx.coroutines.experimental.android.UI

class MainActivity : AppCompatActivity() {

    // ...

    private fun setUpRecyclerView() { // No longer use
        with(recyclerViewTodos) {
            adapter = RecyclerListAdapter(mutableListOf())
        }
    }

    launch(UI) { // Uses UI thread to access recycle
        val todosLiveData = viewModel.getTodos() // Re
        todosLiveData.observe(this@MainActivity, Observ
            // Observes changes in the LiveData
            todos?.let {
```

```
    val adapter = (recyclerViewTodos.adapter as
        RecyclerListAdapter).apply {
            adapter.setItems(it) // Updates list items
        }
    }
}
```

Here, the `RecyclerView` adapter is first initialized with an empty list. Note that you no longer need `launch` to assign the adapter. To add data to the list, the view model's `LiveData` is observed for changes—on any change, the adapter will show the new list of to-do items. In effect, when a new record is inserted into or removed from the database, Room automatically reflects this in the `LiveData`, which in turn notifies its observers (the `MainActivity`) of the data change, finally causing the `RecyclerView` to update.

To make this code compile, you must add the `setItems` method to the `RecyclerListAdapter` class, as shown in [Listing 7.36](#).

[Listing 7.36 Adding `setItems` to the `RecyclerListAdapter`](#)

[Click here to view code image](#)

---

```
class RecyclerListAdapter(
    private val items: MutableList<TodoItem>
) : RecyclerView.Adapter<RecyclerListAdapter.ViewHolder> {
    // ...
    fun setItems(items: List<TodoItem>) {
        this.items.clear()
        this.items.addAll(items)
        notifyDataSetChanged() // Must notify recycler view
    }
}
```

```
}
```

```
}
```

After updating the list of items the `RecyclerView` should display, remember to call `notifyDataSetChanged` to trigger redrawing the view. For large lists, you would want to use `DiffUtil`<sup>8</sup> to improve performance, but it's not necessary for this simple to-do list.

8. <https://developer.android.com/reference/android/support/v7/util/DiffUtil>

This is all that's required to react to data changes using `LiveData` and immediately update your UI to show the latest data. You can verify this works by calling `viewModel.add(...)` in your `MainActivity` to add a new item (you may want to use `delay` as well to see it appear). Or you can wait until you complete the next step that allows users to add new to-do items.

### Adding New To-Do Items

All infrastructure for data persistence and data presentation in the UI is set up now. So what's left is to allow users to change this data. In this section, you will implement a second activity that allows users to add new to-do items, and after this you will also allow them to check off to-do items to delete them.

To add the new activity, create a new package `view.add` and add a new activity to it by right-clicking the package. Choose *New*, then *Activity*, and then *Empty Activity*. Name it `AddTodoActivity` and let Android Studio generate the layout file `activity_add_todo.xml`. Let's first set up the layout. A simple `LinearLayout` as in [Listing 7.37](#) will do for this activity.

[Listing 7.37 Layout for AddTodoActivity](#)

[Click here to view code image](#)

---

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".view.add.AddTodoActivity">

    <EditText
        android:id="@+id/etNewTodo"
        android:hint="@string/enter_new_todo"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="@dimen/margin_medium"
        android:textAppearance="@android:style/TextAppearanceLarge"
        tools:text="@string/enter_new_todo"
        android:inputType="text" />

    <Button
        android:id="@+id/btnAddTodo"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/add_to_do"
        android:textAppearance="@android:style/TextAppearanceLarge"
        android:layout_gravity="center_horizontal" />

</LinearLayout>
```

Add the missing string resources using Android Studio’s suggested actions or by editing `res/values/strings.xml`. I used “Add to-do” and “Enter new todo...” as the values. Before implementing the logic inside this new activity, let’s adjust the floating action button in `MainActivity`—first its layout and then its click handler.

Instead of showing an email icon, the floating action button should have a simple plus icon. To this end, navigate to `res/drawable`, right-click, choose *New*, and then *Image Asset*. Fill in the required information:

- Icon Type: Action Bar and Tab Icons
- Name: `ic_add`
- Asset type: Clip Art
- Click on the *Clip Art* button, search for “add,” and select the simple plus icon
- Theme: HOLO\_LIGHT

Click *Next* and then *Finish*. Now you can use this image asset in `activity_main.xml` by replacing the existing `app:srcCompat` attribute, as shown in [Listing 7.38](#).

[Listing 7.38 Layout for the FloatingActionButton](#)

[Click here to view code image](#)

```
<android.support.design.widget.CoordinatorLayout ...>

<!-- ... -->

<android.support.design.widget.FloatingActionButton
    app:srcCompat="@drawable/ic_add" />

</android.support.design.widget.CoordinatorLayout>
```

With the looks in place, it’s time to adjust the floating action button’s behavior. To this end, go into `MainActivity`, remove the existing default click listener in `onCreate`, and introduce a new setup function, as shown in [Listing 7.39](#).

[Listing 7.39 Setting Up the FloatingActionButton](#)

[Click here to view code image](#)

```
// ... (imports from before)

import android.content.Intent

import com.example.kudoo.view.add.AddTodoActivity

class MainActivity : AppCompatActivity() {

    // ...

    private fun setUpFloatingActionButton() {

        fab.setOnClickListener {

            val intent = Intent(this, AddTodoActivity::class.java)

            startActivity(intent) // Switches to AddTodoActivity
        }
    }
}
```

This helper method enables switching to the new activity when clicking the floating action button. It sets up an intent to switch to the new `AddTodoActivity`, where users can then add a new to-do. Now you can call this new setup method in `onCreate`, as shown in Listing 7.40.

**Listing 7.40 Adjusting `onCreate()`**

[Click here to view code image](#)

```
class MainActivity : AppCompatActivity() {

    // ...

    override fun onCreate(savedInstanceState: Bundle?) {

        // ...

        setUpRecyclerView()

        setUpFloatingActionButton()
    }
}
```

```
}
```

With this, the `MainActivity` is all set. So now it's time to make the new `AddTodoActivity` play its part. It has to use the text the user enters in order to store a new to-do into the database. [Listing 7.41](#) provides all code required for this activity.

**Listing 7.41** Implementing the `AddTodoActivity`

[Click here to view code image](#)

```
import android.os.Bundle

import android.support.v7.app.AppCompatActivity

import com.example.kudoo.R

import com.example.kudoo.db.DB

import com.example.kudoo.model.TodoItem

import com.example.kudoo.view.common.getViewModel

import com.example.kudoo.viewmodel.TodoViewModel

import kotlinx.android.synthetic.main.activity_add_todo.*

import kotlinx.coroutines.experimental.launch

class AddTodoActivity : AppCompatActivity() {

    private lateinit var viewModel: TodoViewModel // L

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_add_todo)

        viewModel = getViewModel(TodoViewModel::class)
    }
}
```

```
        setUpListeners()

    }

private fun setUpListeners() { // Adds new to-do item
    btnAddTodo.setOnClickListener {
        val newTodo = etNewTodo.text.toString()
        launch(DB) { viewModel.add(TodoItem(newTodo)) }
        finish() // Switches back to MainActivity
    }
}
```

The click listener for the *Add to-do* button first reads out the user's text from the `EditText` and then starts a new coroutine that stores the new to-do item to the database. Then the activity finishes, causing the current activity to fade out so that the user gets back to the `MainActivity`, where the to-do item automatically shows up thanks to `LiveData`.

**Tip**

When you test your app in the emulator, you may want to enable keyboard input to type a lot faster in the emulator. If it's not activated, open the AVD Manager in Android Studio, click on the pen icon for the virtual device you're using, click *Show Advanced Settings*, then scroll down to the bottom and check *Enable Keyboard Input*.

This concludes the requirement to let users add their own to-do items. You can now run your app, click on the plus to switch activities, enter your to-do item, and see it pop up in the `RecyclerView` automatically. This is the power of Room working together with a `LiveData` bound to a `RecyclerView`.

As a final touch, you may want to allow users to navigate up from the `AddTodoActivity` to the `MainActivity` without entering a to-do item, and you can do so by making it

a child activity of `MainActivity` in your `AndroidManifest.xml` file. Listing 7.42 shows how to modify the `activity` tag under `application` to achieve this.

**Listing 7.42 Enabling Navigating Up from AddTodoActivity to MainActivity**

[Click here to view code image](#)

```
<activity android:name=".view.add.AddTodoActivity"
          android:parentActivityName=".MainActivity">

<meta-data android:name="android.support.PARENT_ACTIVITY"
          android:value="com.example.kudoo.MainActivity">
</activity>
```

With this, you'll see an arrow at the top of `AddTodoActivity` that allows users to go back without entering a to-do item.

At this point, you may have lots of to-do items created in your app that you no longer want. So the next and final step is to allow users to check off their to-do items, removing them from the database and therefore from the list.

### Enabling Checking Off To-Do Items

In this section, you'll learn how to handle clicks on `RecyclerView` items to let users delete completed to-do items by checking them off in the `RecyclerView`. First, the adapter must be extended to receive a click handler that it can assign to the checkbox when binding to a view holder. Listing 7.43 shows the required changes.

**Listing 7.43 Assigning Event Handlers in the RecyclerView Adapter**

[Click here to view code image](#)

```
class RecyclerListAdapter(
    private val items: MutableList<TodoItem>,
```

```
    private val onItemCheckboxClicked: (TodoItem) -> Unit
) : RecyclerView.Adapter<RecyclerListAdapter.ViewHolder> {
    // ...
}

inner class ViewHolder(...) : ... { // Note that this

    fun bindItem(todoItem: TodoItem) {
        // ...
        cbTodoDone.setOnCheckedChangeListener { _, _ ->
            onItemCheckboxClicked(todoItem)
        }
    }
}

}
```

The adapter now accepts a click handler in its constructor, which must be a function that accepts the selected `TodoItem`. This function is used in `bindItem` to assign a change listener to the checkbox that is associated with the given to-do item. To easily access the `onItemCheckboxClicked` property from the outer scope, make the `ViewHolder` an inner class.

Now, the `MainActivity` can pass in the desired event handler as in [Listing 7.44](#).

#### **Listing 7.44 Assigning Event Handlers in the RecyclerView Adapter**

[Click here to view code image](#)

---

```
// ... (imports from before)

import kotlinx.coroutines.experimental.android.UI

class MainActivity : AppCompatActivity() {
```

```
// ...  
  
private fun setUpRecyclerView() {  
  
    with(recyclerViewTodos) {  
  
        adapter = RecyclerListAdapter(mutableListOf(),  
  
        // ...  
  
    }  
  
    // ...  
  
}  
  
  
private fun onRecyclerViewItemClick(): (TodoItem) -> Unit {  
  
    launch(DB) { viewModel.delete(todo) }  
  
}  
  
}  
  
◀ ▶
```

The creation of the click handler is encapsulated into its own method so that every method has a single responsibility. The click handler starts a coroutine to delete the `TodoItem` from the database. So in this app, to-do items are immediately deleted once they are checked off.

This is all that's required to make this use case work. You can now click the checkbox next to any to-do item to delete that item from the database and, therefore, from the `RecyclerView`.

## SUMMARY

The app you created in this chapter covered many fundamental components and concepts from both Kotlin and Android.

- First, you learned how Android Architecture Components (Room, `ViewModel`, and `LiveData`) can facilitate setting up a database and handling lifecycles on Android.

- Next, you used the Kotlin Android Extensions to make your `ViewHolder` a `LayoutContainer` and to avoid calling `findViewById` explicitly.
- You also saw how to implement a `RecyclerView` in Kotlin, and how to attach click handlers to its list items.
- Lastly, throughout the app, you made use of Kotlin's language features like data classes, companion objects, and top-level declarations to solve tasks in a more concise and idiomatic way.

With this, you are now able to implement basic apps for Android using Kotlin following state-of-the-art tools and coding practices.

## Android App Development with Kotlin: Nutrilicious

*The food you eat can be either the safest and most powerful form of medicine or the slowest form of poison.*

Ann Wigmore

In this chapter, you'll implement the “Nutrilicious” app: a more fleshed-out app that allows users to explore food and nutrition data from the U.S. Department of Agriculture to make healthier diet decisions. [Figure 8.1](#) shows the finished app as you will create it in this chapter.



Figure 8.1 The final app allows users to search foods, choose favorites, and explore data

While developing this sample app, you will recap the fundamental concepts by getting more practice. But you will also dive deeper and implement network access, map JSON data to domain classes, map these domain classes to a Room database, and introduce a repository as the single source of truth for data used in your app.

## SETTING UP THE PROJECT

Every app is born by setting up a new Android project. The setup works the same way as before except that this time you choose a *Bottom Navigation Activity* as your *MainActivity*.

**Note**

In case Android Studio's project wizard changed, you can get the exact project template used in this chapter from the app's GitHub repository<sup>1</sup> (just like the code for every step).

**1. <https://github.com/petersommerhoff/nutrilicious-app>**

To adapt the template for this app, start by adjusting the bottom navigation. In `res/menu/navigation.xml`, remove the last item titled *Notifications*—this app only needs two menu items. Next, change the ID and title of the menu item titled *Dashboard* to represent a menu item showing the user's favorite foods. You can use Shift+F6 to rename the ID, or right-click, select “Refactor,” and then click “Rename....” Listing 8.1 shows the resulting code for the bottom menu in `res/menu/navigation.xml`.

**Listing 8.1 Bottom Navigation Menu****[Click here to view code image](#)**

```
<?xml version="1.0" encoding="utf-8"?>

<menu xmlns:android="http://schemas.android.com/apk/r

    <item android:id="@+id/navigation_home"

        android:icon="@drawable/ic_home_black_24dp"

        android:title="@string/title_home" />

    <item android:id="@+id/navigation_my_foods"

        android:icon="@drawable/ic_dashboard_black_24dp"

        android:title="@string/title_my_foods" />

</menu>
```

The `res/values/strings.xml` file must contain the used string resource, as in Listing 8.2.

**Listing 8.2 String Resources for Bottom Navigation Menu****[Click here to view code image](#)**

```
<string name="title_home">Home</string>

<string name="title_my_foods">My Foods</string>
```

Next, remove unnecessary code from `MainActivity`, namely the `when` case for the *Notifications* menu item in the navigation listener. Also, adjust the `when` case for the *Dashboard* item to your new *My Foods* item. You may also want to rename the listener to a more concise name. Your resulting code should be similar to [Listing 8.3](#).

**Listing 8.3** `MainActivity` Setup for the Bottom Navigation Menu

[Click here to view code image](#)

```
class MainActivity : AppCompatActivity() {  
  
    private val navListener = BottomNavigationView.OnNavigationItemRese  
    when(it.itemId) {  
        R.id.navigation_home -> { // Defines action for home  
            return@OnNavigationItemSelectedListener true  
        }  
        R.id.navigation_my_foods -> { // Defines action for my foods  
            return@OnNavigationItemSelectedListener true  
        }  
    }  
    false  
}  
// ...  
}
```

Clicking the menu items doesn't do much yet but it will change which item is shown as being active because you're returning `true` from the listener—you may set the text of the `TextView` according to the selected item at this point, but that `TextView` will be replaced in the next step. Running the

app should present you with a simple text view and the bottom navigation with two items that indicate correctly which one was selected. This is the basic template that you can build on for the remainder of this chapter.

## ADDING A RECYCLERVIEW TO THE HOME SCREEN

As in many apps, especially ones that present data, the centerpiece of this app is a `RecyclerView`. It will show all foods that were found for a user query. The setup follows the same steps as always.

- Define the layout for the activity containing the `RecyclerView`.
- Define the layout for each list item in the `RecyclerView`.
- Implement the adapter that provides data for the `RecyclerView`.
- Set up the `RecyclerView` in the activity that shows it.

### Layout for MainActivity

The `MainActivity` layout will consist of the `RecyclerView` covering the screen and, of course, the bottom navigation. To this end, replace the `TextView` in `activity_main.xml` with the recycler view, as shown in [Listing 8.4](#). Also, add scrolling behavior to the `ConstraintLayout` so that the list of foods is scrollable.

**Listing 8.4** MainActivity Layout

[Click here to view code image](#)

```
<?xml version="1.0" encoding="utf-8"?>

<android.support.constraint.ConstraintLayout

    app:layout_behavior="@string/appbar_scrolling_view_behavior"

        <android.support.v7.widget.RecyclerView

            android:id="@+id/rvFoods"

            android:layout_width="match_parent"
```

```
        android:layout_height="0dp"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintBottom_toTopOf="@+id/navigation"
        app:layout_constraintBottom_toBottomOf="parent"
        android:background="?android:attr/windowBackground"
        app:menu="@menu/navigation" />

    </android.support.constraint.ConstraintLayout>
```

Here, a layout behavior was added to the `ConstraintLayout` so that the view can be scrolled when the `RecyclerView` overflows the screen, and the recycler view was added. Also, the `BottomNavigationView` layout was simplified.

### Layout for RecyclerView Items

In `res/layout`, add a new layout resource file titled `rv_item.xml` that represents a list item. It shows a food name with a short description and an image view containing a star that will be used to indicate favorite foods. Listing 8.5 implements this layout using a `ConstraintLayout` that aligns the text views below each other, with the image view to their right.

**Listing 8.5 Layout for RecyclerView Items**

[Click here to view code image](#)

```
<?xml version="1.0" encoding="utf-8"?>

<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"

    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="@dimen/medium_padding">

    <TextView
        android:id="@+id/tvFoodName"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="@dimen/medium_font_size"
        app:layout_constraintRight_toLeftOf="@+id/ivStart"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:text="Gingerbread" />

    <TextView
        android:id="@+id/tvFoodType"
        tools:text="Sweets and Candy"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintTop_toBottomOf="@+id/tvFoodName"
        app:layout_constraintStart_toStartOf="@+id/tvFoodName"
        app:layout_constraintRight_toLeftOf="@+id/ivStart" />

```

```
        android:textColor="@color/lightGrey"  
        android:textSize="@dimen/small_font_size" />  
  
<ImageView  
        android:id="@+id/ivStar"  
        android:layout_width="32dp"  
        android:layout_height="32dp"  
        android:contentDescription="@string/content_c  
        app:layout_constraintBottom_toBottomOf="parer  
        app:layout_constraintEnd_toEndOf="parent"  
        app:layout_constraintTop_toTopOf="parent" />  
  
</android.support.constraint.ConstraintLayout>
```

Note that you can use `tools:text` to show a given text in Android Studio's Design View and make the layout come to life. All attributes from the `tools` namespace are used only for tooling, typically for Android Studio's Design View; they don't affect the app at runtime. This layout again requires several new resources, and they are given in [Listing 8.6](#).

#### [Listing 8.6 Resources for RecyclerView Item Layout](#)

[Click here to view code image](#)

---

```
// In res/values/strings.xml  
  
<string name="content_description_star">favorite</strin  
  
// In res/values/dimens.xml  
  
<dimen name="tiny_padding">4dp</dimen>  
  <dimen name="medium_padding">8dp</dimen>
```

```
<dimen name="medium_font_size">16sp</dimen>  
  
<dimen name="small_font_size">13sp</dimen>  
  
// In res/values/colors.xml  
  
<color name="lightGrey">#888888</color>
```

With this, all layouts are ready. So now it's time to write some Kotlin code!

### Implementing the Food Model

To implement the adapter, it is useful to have a model that encapsulates the data shown in each list item. This app shows foods, so you need a **Food** class. Thanks to data classes, this is easy to do. Create a new package **model** and add a data class **Food** to it, as in [Listing 8.7](#). This one line of code is all you need for your models, for now.

[Listing 8.7 Food Data Class](#)

[Click here to view code image](#)

```
data class Food(val name: String, val type: String, \
```

### Implementing the RecyclerView Adapter

As always, implementing the recycler view adapter requires overriding the class `RecyclerView.Adapter<YourViewHolder>` and overriding the three methods `onCreateViewHolder`, `onBindViewHolder`, and `getCount`. Remember, you can make the `ViewHolder` a `LayoutContainer` to use the Kotlin Android Extensions there as well. For this purpose, enable experimental extensions at the very bottom of your module's `build.gradle` file, as shown in [Listing 8.8](#).

[Listing 8.8 Enabling the Experimental Kotlin Android Extensions](#)

[Click here to view code image](#)

```
    androidExtensions {  
        experimental = true  
    }
```

With this setup, you can follow the same structure as in Kudoo to implement the adapter. Create a new package `view.main` and add a new class `SearchListAdapter` to it. This will be the adapter for the list. Also, move the `MainActivity` into this new package because it's intended to contain everything related to the `MainActivity`. [Listing 8.9](#) shows the code for the adapter, which is very similar to the one in the Kudoo app. Try to implement it yourself first to see if you stumble upon any roadblocks.

**Listing 8.9** RecyclerView Adapter

[Click here to view code image](#)

```
import android.support.v7.widget.RecyclerView  
  
import android.view.*  
  
import com.example.nutrilicious.R  
  
import com.example.nutrilicious.model.Food  
  
import kotlinx.android.extensions.LayoutContainer  
  
import kotlinx.android.synthetic.main.rv_item.* // ]  
  
  
class SearchListAdapter(  
  
    private var items: List<Food> // Uses a read-on]  
  
) : RecyclerView.Adapter<ViewHolder>() {  
  
  
    override fun onCreateViewHolder(parent: ViewGroup,  
  
        val view = LayoutInflater.from(parent.context) /  
  
        .inflate(R.layout.rv_item, parent, false)
```

```
        return ViewHolder(view) // Creates view holder t
    }

    override fun getItemCount(): Int = items.size

    override fun onBindViewHolder(holder: ViewHolder, p
        holder.bindTo(items[position]) // Binds data to
    }

    // In this app, we'll usually replace all items so
    fun setItems(newItems: List<Food>) {
        this.items = newItems // Replaces whole list
        notifyDataSetChanged() // Notifies recycler view
    }

    inner class ViewHolder(
        override val containerView: View
    ) : RecyclerView.ViewHolder(containerView), LayoutC

    fun bindTo(food: Food) { // Populates text views
        tvFoodName.text = food.name
        tvFoodType.text = food.type

        val image = if (food.isFavorite) {
            android.R.drawable.btn_star_big_on
        } else {
            android.R.drawable.btn_star_big_off
        }
    }
}
```

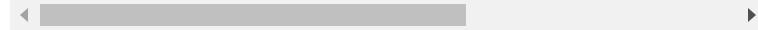
```
        }

        ivStar.setImageResource(image)

    }

}


```



In addition to the basic overrides that every adapter needs and the custom `ViewHolder` class, this implementation already provides a `setItems` method to update the list of items shown in the `RecyclerView`—this method will be used later. Also, the `ViewHolder` already displays the correct `ImageView`, depending on whether the food is a favorite or not. As you can see, these adapters always follow the same basic structure.

### Adding the `RecyclerView` to `MainActivity`

Setting up the `RecyclerView` in your activities also works mostly the same way every time. For now, the adapter is populated with hard-coded sample data again to see if the layout and adapter work as expected. [Listing 8.10](#) encapsulates the `RecyclerView` setup into a method.

[Listing 8.10 Setting Up the RecyclerView in MainActivity](#)

[Click here to view code image](#)

---

```
import android.support.v7.widget.*

class MainActivity : AppCompatActivity() {

    // ...

    private fun setUpSearchRecyclerView() = with(rvFood)

        adapter = SearchListAdapter(sampleData())

        layoutManager = LinearLayoutManager(this@MainActivity)

        addItemDecoration(DividerItemDecoration(
```

```
    this@MainActivity, LinearLayoutManager.VERTICAL  
    ))  
  
    setHasFixedSize(true)  
}  
}
```



As sample data, you can use a list of sample foods as in [Listing 8.11](#). Note that this sample data is used only temporarily. Generally, you should not use hard-coded strings in your Android app and use string resources instead. Here, we skip this step for the sake of brevity.

**Listing 8.11 Hard-Coding the Sample Data**

[Click here to view code image](#)

---

```
import com.example.nutrilicious.model.Food  
  
  
class MainActivity : AppCompatActivity() {  
  
    // ...  
  
    private fun sampleData() = listOf( // Only temporary  
        Food("Gingerbread", "Candy and Sweets", false),  
        Food("Nougat", "Candy and Sweets", true),  
        Food("Apple", "Fruits and Vegetables", false),  
        Food("Banana", "Fruits and Vegetables", true)  
    )  
}
```



Finally, call the setup method in `onCreate`, as in [Listing 8.12](#).

**Listing 8.12 Calling the Setup Method in `onCreate`**

[Click here to view code image](#)

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
    setContentView(R.layout.activity_main)  
    setUpSearchRecyclerView()  
    navigation.setOnNavigationItemSelected(navl  
}  
◀ ▶
```

With this, the **MainActivity** should present the sample foods just like it will in the final app. It should also indicate two of the items as favorites with an active star, as shown in Figure 8.2.

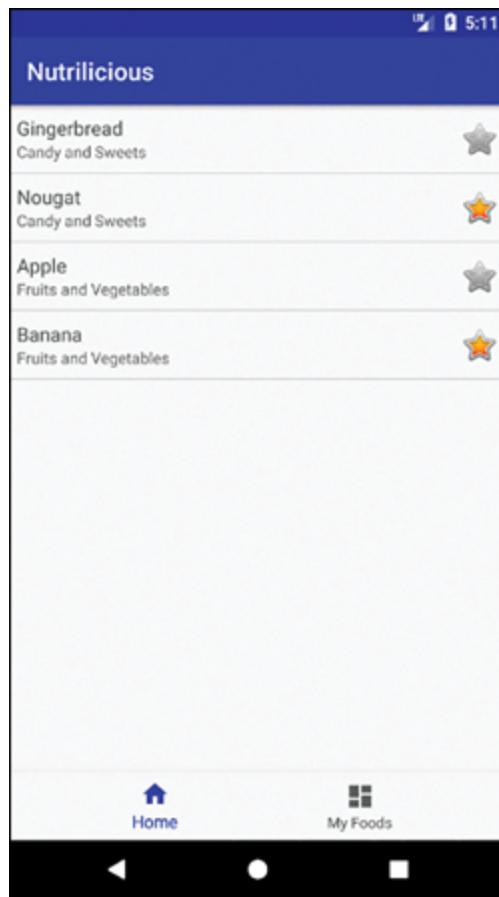


Figure 8.2 Nutrilicious app with a working RecyclerView and hard-coded sample data

Because it always follows a similar process, implementing a **RecyclerView** is straightforward once you have done it a

few times. Even if you were not familiar with it before, I hope these two examples took the magic out of the implementation of RecyclerViews.

## FETCHING DATA FROM THE USDA NUTRITION API

The next major step towards the final app is to actually fetch food data. The U.S. Department of Agriculture (USDA) provides an open API<sup>2</sup> to access its extensive database of food and nutrition details. There are two endpoints that you will use in this app.

### 2. <https://ndb.nal.usda.gov/ndb/doc/index>

- The *Search API* to search for foods that match the user's input
  - Documentation: <https://ndb.nal.usda.gov/ndb/doc/apilist/API-SEARCH.md>
  - Endpoint: <https://api.nal.usda.gov/ndb/search/>
- The *Details API* to retrieve the nutrient amounts for foods
  - Documentation: <https://ndb.nal.usda.gov/ndb/doc/apilist/API-FOOD-REPORTV2.md>
  - Endpoint: <https://api.nal.usda.gov/ndb/V2/reports/>

To use this API, you need to get a free API key from <https://ndb.nal.usda.gov/ndb/doc/index> by clicking on the *Sign up now* link in the middle of the page and entering your information.

For this app, you will use OkHttp<sup>3</sup> to access the network, Retrofit<sup>4</sup> to access the API endpoints, and Moshi<sup>5</sup> to map the JSON data to Kotlin objects. As always, the first step is to add the corresponding dependencies to your module's `build.gradle` file as in [Listing 8.13](#).

### 3. <https://github.com/square/okhttp>

### 4. <https://github.com/square/retrofit>

### 5. <https://github.com/square/moshi>

#### [Listing 8.13 Gradle Dependencies for Network and API Access](#)

[Click here to view code image](#)

---

```
dependencies {
```

```
    // ...
```

```
def retrofit_version = "2.4.0"

implementation "com.squareup.retrofit2:retrofit:$
implementation "com.squareup.retrofit2:converter-


def okhttp_version = "3.6.0"

implementation "com.squareup.okhttp3:logging-inter
implementation "com.squareup.okhttp3:okhttp:$okht
}


```

## Using Retrofit

With the dependencies in place, the next step is to initialize Retrofit to make API calls. As a first step, add a new package `data.network` that contains all network-related code. Inside this package, add a new file `HttpClient.kt` that will use OkHttp and Retrofit to set up a `Retrofit` object that acts as the HTTP client for this app.

At the top of the file, add the constants that you will need inside this file, as in [Listing 8.14](#). For this app, you need the API key that you received from the USDA website and the base URL.

**Listing 8.14** Constants Used for Retrofit

[Click here to view code image](#)

```
import com.example.nutrilicious.BuildConfig

private const val API_KEY = BuildConfig.API_KEY

private const val BASE_URL = "https://api.nal.usda.g
```

As you can see, the API key will come from Gradle's `BuildConfig`. To set this up, you first have to add the key

to your personal Gradle properties, located in the `.gradle` folder in your user directory. The typical locations are

- On Windows: `C:\Users\<USERNAME>\.gradle\gradle.properties`
- On Mac: `/Users/<USERNAME>/.gradle/gradle.properties`
- On Linux: `/home/<USERNAME>/.gradle/gradle.properties`

You may have to create the `gradle.properties` file (if it doesn't exist yet), then add the key to it as shown in [Listing 8.15](#).

**Listing 8.15 Adding API Keys to Your Gradle Properties**

[Click here to view code image](#)

```
Nutrilicious_UsdaApiKey = "<YOUR_API_KEY_HERE>"
```

You can name the property as you like. Here, the project name is used as a prefix so that properties are grouped by project. The next step is to add this property to the project's build configuration. To this end, go to your module's `build.gradle` file and under `buildTypes`, and add a new `debug` build type that makes the key accessible in the project in debug builds. This is shown in [Listing 8.16](#).

**Listing 8.16 Adding a Build Config Field**

[Click here to view code image](#)

```
buildTypes {  
    debug {  
        buildConfigField 'String', "API_KEY", Nutrilicious_UsdaApiKey  
    }  
    release { ... }  
}
```

This makes the API key accessible as `BuildConfig.API_KEY` from anywhere in the project in

debug builds (running the app normally from Android Studio always triggers a debug build). You can add the same build config field to the `release` build type if you want to use the same API key in release builds. After Gradle finishes syncing the project, the API key should be available and allow the assignment from [Listing 8.14](#) that accesses `BuildConfig.API_KEY`.

The next step in `HttpClient.kt` is to start building the Retrofit object using its builder. From there, you'll go down the rabbit hole to construct all the required objects it uses, until you're finally building the complete Retrofit instance with all its dependencies. [Listing 8.17](#) starts off with the function to construct the actual object. Like all other functions in `HttpClient.kt`, it's declared on the file level.

**Listing 8.17 Building the Retrofit Object**

[Click here to view code image](#)

```
import retrofit2.Retrofit

import retrofit2.converter.moshi.MoshiConverterFactory

// ...

private fun buildClient(): Retrofit = Retrofit.Builder()

    .baseUrl(BASE_URL)

    .client(buildHttpClient())

    .addConverterFactory(MoshiConverterFactory.create())

    .build()
```

The base URL is already declared and the `MoshiConverterFactory` comes from the dependency to `retrofit2:converter-moshi`. But building the HTTP client is still to be done using OkHttp. This way, Retrofit relies on OkHttp for the actual HTTP request, and OkHttp allows adding interceptors to perform logging, add the API key to the query, and more. [Listing 8.18](#) sets up an OkHttp client that does exactly that.

**Listing 8.18 Building the HTTP Client**

[Click here to view code image](#)

```
import okhttp3.OkHttpClient

import java.util.concurrent.TimeUnit

// ...

private fun buildHttpClient(): OkHttpClient = OkHttpClient()

    .connectTimeout(30, TimeUnit.SECONDS)

    .readTimeout(30, TimeUnit.SECONDS)

    .addInterceptor(loggingInterceptor()) // Logs API requests

    .addInterceptor(apiKeyInterceptor()) // Adds API key to requests

    .build()
```

As you can see, `OkHttp` makes it easy to set timeouts and attach interceptors. In this app, you'll use a logging interceptor that logs any request results to the Logcat and an interceptor that injects the API key into the request URL.

Creating the interceptors is the last step to finally build the entire Retrofit object. Listing 8.19 implements the logging interceptor.

**Listing 8.19 Building the HTTP Client**

[Click here to view code image](#)

```
import okhttp3.logging.HttpLoggingInterceptor

// ...

private fun loggingInterceptor() = HttpLoggingInterceptor()

    level = if (BuildConfig.DEBUG) {

        HttpLoggingInterceptor.Level.BODY // Only does this if DEBUG is true

    } else {

        HttpLoggingInterceptor.Level.NONE // Otherwise reduces log output

    }
```

```
}
```

The `HTTPLoggingInterceptor` from `OkHttp` already implements the basic logic. All that's left to do is to set the appropriate logging level for development and production. This is done using the predefined `BuildConfig.DEBUG` flag so that logging is only performed during development. Note that the `apply` function allows declaring this function as a single expression.

The next interceptor adds the API key as a query parameter to the URL. [Listing 8.20](#) encapsulates the setup of this kind of interceptor into a separate function.

**Listing 8.20 Encapsulating Interceptor Creation**

[Click here to view code image](#)

```
import okhttp3.Interceptor

// ...

private fun injectQueryParams(
    vararg params: Pair<String, String>
): Interceptor = Interceptor { chain ->

    val originalRequest = chain.request()

    val urlWithParams = originalRequest.url().newBuilder()

        .apply { params.forEach { addQueryParameter(it.key, it.value) } }

        .build()

    val newRequest = originalRequest.newBuilder().url(urlWithParams)

    chain.proceed(newRequest)

}
```

Because `Interceptor` is a SAM interface coming from Java, you can use Kotlin's SAM conversions to create an interceptor with lambda syntax. This implicitly overrides `Interceptor.intercept` to intercept the request chain and add query parameters to it. The details of this method are OkHttp-specific, but notice how `apply` can be extremely useful in combination with builder-style methods.

Using this helper function, setting up the remaining interceptor is a matter of passing in the query parameter for the API key, as shown in [Listing 8.21](#).

**Listing 8.21 Creating the Interceptor that Adds the API Key to the Query**

[Click here to view code image](#)

```
private fun apiKeyInterceptor() = injectQueryParams(  
    "api_key" to API_KEY  
)
```

Due to the use of `Pair` in the helper function, the definition of query parameters becomes clean and readable using Kotlin's `to` function. It would be trivial to add or remove additional query parameters or to create another interceptor without duplicating code.

You're now out of the rabbit hole and the `buildClient` function is able to create a Retrofit object. The next step is to call it to create an object that is used to access the *Search API*. With Retrofit, you first need an interface that defines any requests to make to an endpoint based on the base URL. [Listing 8.22](#) defines the interface needed to access the *Search API*. Place this into a new file `UsdaApi` in the `data.network` package.

**Listing 8.22 Defining the Retrofit Interface to Access the Search API**

[Click here to view code image](#)

```
import okhttp3.ResponseBody  
  
import retrofit2.Call
```

```
import retrofit2.http.*

interface UsdaApi {

    @GET("search?format=json") // Is appended to the base URL
    fun getFoods(
        @Query("q") searchTerm: String, // Only non-null arguments are appended
        @Query("sort") sortBy: Char = 'r', // Sorts by
        @Query("ds") dataSource: String = "Standard Reference",
        @Query("offset") offset: Int = 0
    ): Call<ResponseBody> // Allows type inference
}
```

The `@GET` annotation indicates that this performs a GET request and its argument will be appended to the base URL, resulting in the endpoint for the *Search API*. The `@Query` annotation indicates that the argument is appended to the URL as the value for the corresponding query parameter, such as `q` or `sort`. Thus, the final request URL is built based on the `BASE_URL`, the `@GET` suffix, the query parameter from `@Query`, and any query parameters from interceptors. A resulting request URL has the following form:

[Click here to view code image](#)

```
https://api.nal.usda.gov/ndb/search?format=json&q=raw
&ds=Standard%20Reference&offset=0.
```

The `Call<T>` in the return value is Retrofit's implementation of a *future*. Like every future, it wraps the result of the asynchronous call, which is performed off the main thread. Here, you parse the result into an OkHttp `ResponseBody` to fetch the raw JSON data and see if the request itself works—

mapping the JSON result to domain classes is done in the next section.

To finish the infrastructure for API requests, you just need to build the Retrofit object and then use it to create an implementation of this interface. Listing 8.23 demonstrates how to do this in the `HttpClient.kt` file.

**Listing 8.23 Building the *Search API Object***

[Click here to view code image](#)

```
private val usdaClient by lazy { buildClient() }

val usdaApi: UsdaApi by lazy { usdaClient.create(Usda
```

Note that the `usdaApi` object is the only declaration in `HttpClient.kt` that is exposed to the outside—all other declarations are private and define the internal details of how this object is created. Because this object is expensive to create, its initialization is deferred using `lazy`. This also makes sure it is instantiated only once and then cached.

## Performing API Requests

You can use the `usdaApi` object to perform the API request. To test your setup, you can perform a temporary test request in `MainActivity`. But first, you'll need to add the internet permission to your app and include Kotlin's coroutines to perform the network request in a background thread. First, in your `AndroidManifest.xml`, add the permission as in Listing 8.24.

**Listing 8.24 Enabling Internet Access**

[Click here to view code image](#)

```
<manifest ...>

    <uses-permission android:name="android.permission.INTERNET" />

    <application ...>...</application>

</manifest>
```

To perform asynchronous network requests, add the coroutine dependencies in your module's Gradle build file as in [Listing 8.25](#).

**Listing 8.25 Gradle Dependencies for Kotlin Coroutines**

[Click here to view code image](#)

```
def coroutines_version = "0.24.0" // Latest version

implementation "org.jetbrains.kotlinx:kotlinx-corouti
implementation "org.jetbrains.kotlinx:kotlinx-corouti
```

As a final preparation step, create a dedicated coroutine dispatcher for network calls. For this, add a new file `NetworkDispatcher` into the `data.network` package. Inside it, you can declare a thread pool using two threads for network access, as in [Listing 8.26](#).

**Listing 8.26 Coroutine Dispatcher for Network Calls**

[Click here to view code image](#)

```
import kotlinx.coroutines.newFixedThreadPoolContext

val NETWORK = newFixedThreadPoolContext(2, "NETWORK")
```

With this, you can now perform a test request in `MainActivity.onCreate` to check your setup. Thanks to the logging interceptor, you can observe the request and its results in Android Studio's Logcat. [Listing 8.27](#) demonstrates how to use the `usdaApi`. This use is just temporary and of course not encouraged in production. Later, you will again incorporate view models for asynchronous requests.

**Listing 8.27 Performing an API Request**

[Click here to view code image](#)

```
import com.example.nutrilicious.data.network. *
import com.example.nutrilicious.model.Food
import kotlinx.android.synthetic.main.activity_main.*
```

```
import kotlinx.coroutines.launch

// ...

class MainActivity : AppCompatActivity() { // ...

    override fun onCreate(savedInstanceState: Bundle?) {

        // ...

        launch(NETWORK) {

            usdaApi.getFoods("raw").execute() // Logs result
        }
    }
}
```

This calls the `getFoods` method from the `UsdaApi` interface and executes the asynchronous `Call<T>` in the background to perform the request. When you run the app, you should now see logging entries from `OkHttp` and see the retrieved JSON data. You can open Logcat using Alt+6 or from the bottom toolbar and search for “OkHttp” to filter out the other entries.

**Note**

There's an issue in Android Studio at the time of writing where *Apply Changes* does not play well with coroutines. If you get an error “`CoroutineImpl.label` is inaccessible...,” try re-running the app normally *without using Apply Changes* (using Shift+F10 or Ctrl+R on Mac).

You're now able to perform network requests using Retrofit to fetch data from the USDA in JSON format. Of course, users do not want to see raw JSON data. So in the following section, you will map the data to domain classes.

## MAPPING JSON DATA TO DOMAIN CLASSES

At this point, you have your raw JSON data and your model class `Food`. So let's map the JSON data to the `Food` class in order to work with it in the app. The typical approach using Moshi is to first parse the JSON data into *data transfer objects* (DTOs) and then map those to your model classes.

Moshi officially supports Kotlin, meaning it knows how to handle things like primary constructors and properties when mapping to classes. For Moshi itself, add the dependencies from [Listing 8.28](#) to the module's build script.

**Listing 8.28 Gradle Dependencies for Moshi**

[Click here to view code image](#)

```
def moshi_version = "1.6.0"

implementation "com.squareup.moshi:moshi:$moshi_version"
kapt "com.squareup.moshi:moshi-kotlin-codegen:$moshi_version"
```

To map JSON data to your DTOs, Moshi can either use reflection or code generation. In this app, you use code generation—thus the `moshi-kotlin-codegen` dependency. Using reflection, you would transitively depend on `kotlin-reflect`, which adds over 2MB and around 16K methods to your Android PackAge (APK)<sup>6</sup> (without ProGuard) so I'd recommend avoiding it.

6. Android PackAge is the distribution format for Android apps.

The code generation approach relies on annotations (like Room). So you need to apply the `kotlin-kapt` plugin for annotation processing at the top of your `build.gradle` file, as shown in [Listing 8.29](#).

**Listing 8.29 Enabling the Kotlin Annotation Processor**

[Click here to view code image](#)

```
apply plugin: 'kotlin-kapt'
```

## Mapping JSON to DTOs

To let Moshi map the JSON data to a DTO class, the DTO must replicate the structure of the JSON data. More specifically, they must use the same property names to indicate that a property should be populated with the data from the corresponding JSON field.

### Tip

To write down your DTOs, I'd recommend creating a `.json` file in your project that contains sample data from the API. You can use it to explore the structure and write your DTOs. In Android Studio, you can create a *Sample Data Directory* and place it there.

7

7. [https://api.nal.usda.gov/ndb/search/?format=json&ds=Standard%20Reference&q=raw&sort=r&max=10&api\\_key=DEMO\\_KEY](https://api.nal.usda.gov/ndb/search/?format=json&ds=Standard%20Reference&q=raw&sort=r&max=10&api_key=DEMO_KEY)

Once you have the file, you can use *Window, Editor Tabs*, then *Split Vertically* from the menu to open another editor. This way, you can open the `.json` file in one of them and write down your DTO alongside it. This makes it easier to map the structure correctly.

For the Search API, the returned JSON data has the format shown in Listing 8.30.

### **Listing 8.30 *Search API JSON Format***

[Click here to view code image](#)

```
        "name": "Coriander (cilantro) leaves, raw",
        "ndbno": "11165",
        "ds": "SR",
        "manu": "none"

    },
    // More items here...
]
}

}
```

The actual data you need is nested into the object's `list` property, which again wraps it into an `item` property. Thus, you need to create wrapper DTOs that, intuitively speaking, navigate down that hierarchy. To do so, add a new package `data.network.dto` and add a new file `SearchDtos.kt` to it. Listing 8.31 shows the definition of the wrapper types to navigate down the `list` and `item` properties.

**Listing 8.31** Wrapper DTOs for the *Search API*

[Click here to view code image](#)

---

```
import com.squareup.moshi.JsonClass

@JsonClass(generateAdapter = true)

class ListWrapper<T> {

    var list: T? = null // Navigates down the 'list' adapter
}

@JsonClass(generateAdapter = true)

class ItemWrapper<T> {

    var item: T? = null // Navigates down the 'item' adapter
}
```

```
}
```

```
typealias SearchWrapper<T> = ListWrapper<ItemWrapper<
```

Moshi will map the JSON `list` property to the corresponding field in the `ListWrapper` class, and because you will use `ListWrapper<ItemWrapper<T>>`, the JSON `item` property will be mapped to the property in `ItemWrapper`. The `typealias` allows using a more concise syntax when using the DTOs. Because the two wrappers are never used separately, you only need the `SearchWrapper<T>` outside of this file. The `@JsonClass` annotations tell Moshi to include these classes in the JSON mapping process.

Now that the wrappers navigate to the actual data, you can add the DTO that contains this data, the `FoodDto`. [Listing 8.32](#) shows its declaration, which also belongs to `SearchDtos.kt`.

#### **Listing 8.32 Food DTO for the *Search API***

[Click here to view code image](#)

```
@JsonClass(generateAdapter = true)

class FoodDto { // Uses lateinit for properties that

    lateinit var ndbno: String

    lateinit var name: String

    lateinit var group: String

}
```

There are more properties available in the JSON data, but for this app you only need the nutrition database number (NDBNO) that uniquely identifies a food, its name, and its group (its category). Using late-initialized properties, you can avoid creating nullables here. This is a great use case for `lateinit`, namely when there is a library or tool that is responsible for initialization. Note that the NDBNO is

declared as a `String` because it may be left-padded with zeros, and while the API finds a food for the NDBNO "09070", it will not find one for "9070". The NDBNO will be used later to retrieve details about a specific food.

Now that you have mapped the JSON data to DTOs, you can tell Retrofit to return a DTO from its asynchronous call. It then uses Moshi to perform the mapping. So in `UsdaApi.kt`, update the return type as shown in [Listing 8.33](#).

**Listing 8.33** Returning the DTO from Retrofit Calls

[Click here to view code image](#)

```
import com.example.nutrilicious.data.network.dto.*

fun getFoods(@Query("q") searchTerm: String, ...): Call<
```

## Mapping DTOs to Models

Mapping the DTOs to your domain classes is straightforward and can be done by a secondary constructor in the models. [Listing 8.34](#) shows how to do it for the `Food` class.

Here, the model was extended with an `id` and a secondary constructor that performs the mapping from a DTO. In this example, the mapping is just a matter of renaming properties. This is one of the typical tasks when mapping to your models because DTOs typically use the property names given in the JSON data.

**Listing 8.34** Secondary Constructor to Map DTO to Model

[Click here to view code image](#)

```
data class Food(

    val id: String, // New property

    val name: String,

    val type: String,

    var isFavorite: Boolean = false
```

```
    ) {  
  
    constructor(dto: FoodDto) : this(dto.ndbno, dto.name)  
    }  
  
    
```

For simple mappings like this, you don't necessarily have to separate your DTO and model. Instead, you could use `@SerializedName` to specify the JSON names and then choose different property names. Here, we use separate DTOs consistently to illustrate the process. In general, more complex transformations of the data may be required. In these cases, you should separate your DTOs from the models.

You are now all set to map the raw JSON data to the classes you actually want to use in your app, so you can call it again in the `MainActivity` to sanity-check the mapping. Because the data is eventually represented as a `List<Food>`, you can easily populate the `RecyclerView` with it to display the API data in the app. In Listing 8.35, `MainActivity.onCreate` is adjusted accordingly. Again, this is just temporary use. You should not perform asynchronous requests from `onCreate` like this in production because of potential memory leaks.

#### **Listing 8.35 Displaying Mapped Data in RecyclerView**

[Click here to view code image](#)

---

```
import kotlinx.coroutines.android.UI  
  
import kotlinx.coroutines.withContext  
  
// ...  
  
class MainActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?)  
  
    // ...  
  
    launch(NETWORK) {  
  
        val dtos = usdaApi.getFoods("raw").execute()?.  
        val foods = dtos?.  
        val foodList = foods?.  
        val adapter = FoodAdapter(foodList)  
        val recyclerView = findViewById<RecyclerView>(R.id.recycler_view)  
        val layoutManager = LinearLayoutManager(this)  
        recyclerView.layoutManager = layoutManager  
        recyclerView.adapter = adapter  
    }  
}
```

This executes the request and accesses the data stored inside wrappers. After that, it maps the list of DTOs to a list of models by calling the `Food` constructor with each DTO. In the UI thread, this food list can then be shown in the `RecyclerView` by passing it to the adapter.

At this point, you can remove the `sampleData` and initialize the adapter with an empty list instead. It will then be populated when the request data is ready. Listing 8.36 shows the necessary adjustment in `setUpRecyclerView`.

### **Listing 8.36 Removing the Sample Data**

**Click here to view code image**

```
    adapter = SearchListAdapter(emptyList())
```

When you run the app now, it should fetch JSON data from the USDA API, map it to your domain models, and display it in the `RecyclerView`. With this, the barebones app functionality is already in place. However, performing the asynchronous request from `onCreate` is not safe because it is not lifecycle aware and the call remains active even if the activity is destroyed, potentially causing memory leaks. So before you extend the existing functionality, let's refactor the architecture to avoid asynchronous calls directly from `MainActivity`—you already know how this is done.

## INTRODUCING A VIEWMODEL FOR SEARCH

The `MainActivity` should get its data from a view model. This automatically allows lifecycle-aware asynchronous requests across configuration changes, in addition to cleaning up the architecture. Let's include the dependencies for *all* required Android Architecture Components already as they will be used later. Listing 8.37 shows the dependencies.

**Listing 8.37 Gradle Dependencies for Architecture Components**

[Click here to view code image](#)

```
def room_version = "1.1.0"

implementation "android.arch.persistence.room:runtime:$room_version"
kapt "android.arch.persistence.room:compiler:$room_version"

def lifecycle_version = "1.1.1"

implementation "android.arch.lifecycle:extensions:$lifecycle_version"
```

Now add a new package `viewmodel` and add a new file `SearchViewModel.kt`. This view model can extend the `ViewModel` class instead of `AndroidViewModel` because it does not require the application context. It provides a clean interface for all calls to the *Search API*. To do so, it uses a helper function to perform to actual call, as shown in Listing 8.38.

This executes the API request, reads the response, and navigates down through the `list` and `item` properties of the `SearchWrapper`. The elvis operator handles not only the case that `doRequest` or any property returns `null` but also the case that an exception occurs. In both cases, the method returns an empty list.

**Listing 8.38 Step 1: Implementing the SearchViewModel**

[Click here to view code image](#)

```
import android.arch.lifecycle.ViewModel

import com.example.nutrilicious.data.network.dto.*

import retrofit2.Call

class SearchViewModel : ViewModel() {

    private fun doRequest(req: Call<SearchWrapper<List<

        req.execute().body()?.list?.item ?: emptyList()

    }
}
```

The view model uses this helper to implement a suspending function that performs the asynchronous call using `withContext` (that can return back a result) as in [Listing 8.39](#).

#### **Listing 8.39 Step 2: Implementing the SearchViewModel**

[Click here to view code image](#)

```
import com.example.nutrilicious.data.network.*

import com.example.nutrilicious.model.Food

import kotlinx.coroutines.withContext

class SearchViewModel : ViewModel() {

    suspend fun getFoodsFor(searchTerm: String): List<Food> {
        val request: Call<SearchWrapper<List<FoodDto>>> =
            val foodDtos: List<FoodDto> = withContext(NETWORK)
                return foodDtos.map(:Food)
    }
    // ...
}
```

```
}
```

This performs the asynchronous call using the network dispatcher and maps all retrieved DTOs to `Food` objects. With the view model in place, you can now reference it in `MainActivity` as usual, using a late-initialized property that is initialized in `onCreate`, as done in Listing 8.40.

**Listing 8.40 Using the `SearchViewModel` in `MainActivity`**

[Click here to view code image](#)

```
import com.example.nutrilicious.view.common.getViewModel
```

---

```
class MainActivity : AppCompatActivity() {
```

```
    private lateinit var searchViewModel: SearchViewModel
```

```
    // ...
```

```
    override fun onCreate(savedInstanceState: Bundle?)
```

```
        // ...
```

```
        navigation.setOnNavigationItemSelected(navigationItem: NavigationItem?) {
```

```
            searchViewModel = getViewModel(SearchViewModel::class.java)
```

```
            // ...
```

```
        }
```

```
    // ...
```

```
}
```

The `getViewModel` function is defined in a `ViewExtensions.kt` file as in the Kudoo app. You can simply copy and paste the `view.common` package to this project (as well as future projects). Listing 8.41 recaps this useful extension.

**Listing 8.41 `getViewModel` Extension Function**

[Click here to view code image](#)

```
import android.arch.lifecycle.*  
  
import android.support.v4.app.FragmentActivity  
  
import kotlin.reflect.KClass  
  
  
fun <T : ViewModel> FragmentActivity.getViewModel(modelClass: KClass<T>): T {  
    return ViewModelProviders.of(this).get(modelClass.java)  
}
```

Finally, remove the asynchronous call in `onCreate` and therefore the dependency to `usdaApi` from your `MainActivity` and use the view model instead, as in [Listing 8.42](#).

**Listing 8.42 Using the SearchViewModel in MainActivity**

[Click here to view code image](#)

```
override fun onCreate(savedInstanceState: Bundle?) {  
    // ...  
  
    searchViewModel = getViewModel(SearchViewModel::class.java)  
  
    launch(NETWORK) { // Uses network dispatcher for requests  
        val foods = searchViewModel.getFoodsFor("raw")  
  
        withContext(UI) { // Populates recycler view with foods  
            (rvFoods.adapter as SearchListAdapter).setItems(foods)  
        }  
    }  
}
```

This is all that is required to set up this view model in your project. Now the `MainActivity` only has a reference to its view model that provides it with all the data it needs, and in a lifecycle-aware way. The app should still show the results fetched for “raw” when launched.

## LETTING USERS SEARCH FOODS

As a next step, the app should display what the user searches using a search field at the top. In other words, it should make an API request whenever the user issues a search. First, you should encapsulate the logic for a request into a method in `MainActivity`, as in [Listing 8.43](#).

**Listing 8.43** Encapsulating the Logic For Requests

[Click here to view code image](#)

```
private fun updateListFor(searchTerm: String) {  
    launch(NETWORK) {  
        val foods = searchViewModel.getFoodsFor(searchTer  
  
        withContext(UI) {  
            (rvFoods.adapter as SearchListAdapter).setItems  
        }  
    }  
}
```

This method still contains the `launch` call to perform the API request on a background thread so `updateListFor` is not a suspending function and can therefore be called from outside a coroutine. Next, remove the test request in `onCreate`.

Instead, you’ll implement a proper Android Search Interface<sup>8</sup> that lets users search the foods in which they are interested.

8. <https://developer.android.com/training/search/setup>

## Implementing a Search Interface

The first step is to add a menu resource representing the search bar that will be displayed at the top of the screen.

For this, add a new menu resource

`res/menu/search_menu.xml`, and set it up as in

[Listing 8.44](#) with just a single menu item of type `SearchView`.

[Listing 8.44 Menu Resource for Search Interface](#)

[Click here to view code image](#)

```
<?xml version="1.0" encoding="utf-8"?>

<menu xmlns:app="http://schemas.android.com/apk/res-@
      xmlns:android="http://schemas.android.com/apk/res-@

        <item android:id="@+id/search"
              android:title="@string/search_title"
              android:icon="@android:drawable/ic_menu_search"
              app:showAsAction="always"
              app:actionViewClass="android.widget.SearchView" />

    </menu>
```

This uses a new string resource, shown in [Listing 8.45](#).

[Listing 8.45 String Resource for Search Menu](#)

[Click here to view code image](#)

```
<string name="search_title">Search food...</string>
```

Next, you need a so-called *searchable configuration* that defines the behavior of the search view. This is defined in a new resource file `res/xml/searchable.xml`, as in

[Listing 8.46](#).

[Listing 8.46 Searchable Configuration](#)

[Click here to view code image](#)

```
<?xml version="1.0" encoding="utf-8"?>

<searchable xmlns:android="http://schemas.android.com

    android:label="@string/app_name"

    android:hint="@string/search_title" />
```

The label must be the same as the label of the **application** tag from your `AndroidManifest.xml` file, thus using `@string/app_name`. Also, the hint helps users know what to enter into the search field. Here, the search title from before is reused as a hint.

Next, the Android Manifest must be extended by three things: first, metadata that tells it where to find the search interface; second, which activity should handle the search intents; and third, the main activity must use launch mode `singleTop` to be able to handle the search intents itself. All changes must be made to the **activity** tag for the main activity in the `AndroidManifest.xml`, as shown in [Listing 8.47](#).

**Listing 8.47** Setting Up Search in the Android Manifest

[Click here to view code image](#)

```
<activity

    android:launchMode="singleTop"                   //>

    android:name=".view.main.MainActivity"

    android:label="@string/app_name">

    <meta-data android:name="android.app.searchable" />

        android:resource="@xml/searchable" />

    <intent-filter>

        <action android:name="android.intent.action.SEARC

    </intent-filter>

    ...
```

```
</activity>
```

The `singleTop` launch mode tells Android to route any intents to `MainActivity` to the existing instance of the activity. Without it, Android would create a new instance of `MainActivity` for each intent, causing its state to be lost. Thus, to handle its own searches itself, the activity must use `singleTop` as launch mode.

To inflate a menu into an activity, you must override the `onCreateOptionsMenu` method. So in `MainActivity`, override this method as shown in [Listing 8.48](#) to inflate the search menu.

**Listing 8.48 Inflating and Setting Up the Search Menu**

[Click here to view code image](#)

```
import android.app.SearchManager

import android.widget.SearchView

import android.content.Context

import android.view.Menu

// ...

class MainActivity : AppCompatActivity() {

    // ...

    override fun onCreateOptionsMenu(menu: Menu): Boolean {
        menuInflater.inflate(R.menu.search_menu, menu)

        // Associates searchable configuration with the SearchView
        val searchManager = getSystemService(Context.SEARCH_SERVICE) as SearchManager
        (menu.findItem(R.id.search).actionView as SearchView).apply {
            setSearchableInfo(searchManager.getSearchableInfo(this))
        }
    }
}
```

```
    return true
```

```
}
```

```
}
```

Finally, to handle search intents (when a user initiates a search), override the `onNewIntent` method and filter for the `ACTION_SEARCH` intent, as demonstrated in [Listing 8.49](#).

**Listing 8.49 Handling Search Intents**

[Click here to view code image](#)

```
import android.content.Intent

// ...

class MainActivity : AppCompatActivity() {

    // ...

    override fun onNewIntent(intent: Intent) {

        if (intent.action == Intent.ACTION_SEARCH) { // 

            val query = intent.getStringExtra(SearchManager
                .query)

            updateListFor(query)

        }
    }
}
```

When you run the app now, you should be able to search for any food you want and get the relevant results displayed in the `RecyclerView`. If nothing is shown, make sure to enter a query that returns a result, such as “raw”—the app does not handle empty responses yet.

## INTRODUCING FRAGMENTS I: THE SEARCH FRAGMENT

At this point, the *Home Screen* is almost finished, except for the listeners for the `RecyclerView`. These will be added later and will also be used by the *My Foods Screen*. In order to prevent the `MainActivity` from becoming a god activity that tries to handle everything itself, you will now modularize the `MainActivity` into fragments.

Fragments encapsulate a part of the UI to make it reusable. Typically, an activity is made up of multiple fragments. In this app, you will create one for each item in the bottom navigation menu, so a `SearchFragment` and a

`FavoritesFragment`. In this section, you'll modularize your existing code into a `SearchFragment` and use it in the `MainActivity`, thus making the latter a lot smaller. In the next section, it will then be easy to incorporate the `FavoritesFragment`.

As always, let's start by creating the necessary layout files. In `res/layout`, add a new file `fragment_search.xml` that contains a `SwipeRefreshLayout` with the `RecyclerView` from `activity_main.xml` as its only child—you can cut and paste the recycler view layout from there. Remove any “constraint” attributes that refer to the `ConstraintLayout` used in `activity_main.xml` and give it a `layout_weight` instead. Listing 8.50 shows the layout.

**Listing 8.50 Layout for the SearchFragment**

[Click here to view code image](#)

```
<?xml version="1.0" encoding="utf-8"?>

<android.support.v4.widget.SwipeRefreshLayout

    xmlns:android="http://schemas.android.com/apk/res

        android:id="@+id/swipeRefresh"

        android:layout_width="match_parent"
```

```
    android:layout_height="match_parent">

    <android.support.v7.widget.RecyclerView
        android:id="@+id/rvFoods"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"/>

</android.support.v4.widget.SwipeRefreshLayout>
```

Having cut this element from `activity_main.xml`, the next step is to add a placeholder for the fragment in its place. The placeholder defines where fragments will be attached to the activity. Listing 8.51 uses an empty `FrameLayout` for this.

**Listing 8.51 Layout for the MainActivity**

[Click here to view code image](#)

```
<android.support.constraint.ConstraintLayout ...>

    <!-- Placeholder for fragments -->

    <FrameLayout
        android:id="@+id/mainView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical" />

    <android.support.design.widget.BottomNavigationView
        android:id="@+id/bottomNav"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom"
        android:background="#3399CC"
        android:menu="@menu/bottom_nav_menu" />

</android.support.constraint.ConstraintLayout>
```

This wraps up the layout side, so now you can start implementing the search fragment. In `view/main`, add a new file `SearchFragment.kt` (*not* using Android Studio's wizards to create a fragment). This fragment class must extend `android.support.v4.app.Fragment`.

Fragments have slightly different lifecycle methods you can override to initialize the UI, populate dependencies, and perform other setup logic. The search fragment overrides three lifecycle methods, which are called in the given order:

- `onAttach`: called when the fragment first gets attached to its context (the activity)
- `onCreateView`: called after `onAttach` (and `onCreate`) to initialize the UI
- `onViewCreated`: called directly after `onCreateView` returns

The search fragment uses `onAttach` to get a reference to its `SearchViewModel`, `onCreateView` is used to inflate the layout, and in `onViewCreated`, all views are ready to be initialized. Listing 8.52 shows the `SearchFragment` class this far.

#### **Listing 8.52 Overriding Lifecycle Methods for the Search Fragment**

[Click here to view code image](#)

---

```
import android.content.Context

import android.os.Bundle

import android.support.v4.app.Fragment

import android.view.*

import com.example.nutrilicious.R

import com.example.nutrilicious.view.common.getViewModel

import com.example.nutrilicious.viewmodel.SearchViewModel

class SearchFragment : Fragment() {
```

```
private lateinit var searchViewModel: SearchViewModel

override fun onAttach(context: Context?) {
    super.onAttach(context)
    searchViewModel = getViewModel(SearchViewModel::class.java)
}

override fun onCreateView(inflater: LayoutInflater,
    savedInstanceState: Bundle?): View? {
    return inflater.inflate(R.layout.fragment_search,
}

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
    setUpSearchRecyclerView() // Will come from MainActivity
    setUpSwipeRefresh() // Implemented later
}

}
```

Even though the lifecycle methods are slightly different, the concepts and structure are the same as in activities—such as using a late-initialized property for the view model. Recall that you can use **Ctrl+O** (also on Mac) to override members.

The `setUpSearchRecyclerView` method referenced in `onViewCreated` already exists but not yet in the fragment. You can now move it from `MainActivity` to the `SearchFragment`. Also, move `updateListFor` to the search fragment and make it public. After this, `MainActivity` should have no more private members other than the navigation listener (it no longer needs a reference to

the view model either). In `MainActivity`, you can remove all references to the removed members (`setUpSearchRecyclerView`, `updateListFor`, and `searchViewModel`).

This code uses a different `getViewModel` extension, which is defined on the `Fragment` class, as shown in [Listing 8.53](#).

**Listing 8.53** `getViewModel` Extension for `Fragment`

[Click here to view code image](#)

```
fun <T : ViewModel> Fragment.getViewModel(modelClass:  
    return ViewModelProviders.of(this).get(modelClass.j  
}  
◀ ▶
```

Because you are now using fragments, you should always consider the possibility that the fragment is not attached to its activity so that views are inaccessible. In Kotlin, this can be handled concisely using the safe call operator on the UI elements. [Listing 8.54](#) adjusts `updateListFor` accordingly.

**Listing 8.54** Accessing UI Elements Safely

[Click here to view code image](#)

```
fun updateListFor(searchTerm: String) { // Is now pu  
    launch(NETWORK) { // ...  
        withContext(UI) {  
            (rvFoods?.adapter as? SearchListAdapter)?.setIt  
        }  
    }  
}
```

Next, adjust `setUpSearchRecyclerView` by replacing references to `this@MainActivity` with the fragment's `context`, as in [Listing 8.55](#).

**Listing 8.55** Adjusting the RecyclerView Setup

[Click here to view code image](#)

```
private fun setUpSearchRecyclerView() = with(rvFoods) {
    adapter = SearchListAdapter(emptyList())
    layoutManager = LinearLayoutManager(context)
    addItemDecoration(DividerItemDecoration(
        context, LinearLayoutManager.VERTICAL
    ))
    setHasFixedSize(true)
}
```

There is one new layout component to set up in this fragment, the `SwipeRefreshLayout`. It requires an action that reissues the last search when swiping down to refresh the data. Listing 8.56 adds a simple setup method for it, and a `lastSearch` property to remember the last search.

[Listing 8.56 Setting Up the SwipeRefreshLayout](#)

[Click here to view code image](#)

```
class SearchFragment : Fragment() {
    private var lastSearch = ""

    // ...

    private fun setUpSwipeRefresh() {
        swipeRefresh.setOnRefreshListener {
            updateListFor(lastSearch) // Re-issues last search
        }
    }

    fun updateListFor(searchTerm: String) {
        lastSearch = searchTerm // Remembers last search
    }
}
```

```
// ...  
}  
}  

```

At this point, there should be no more errors in `onViewCreated` because all methods exist. This concludes the `SearchFragment` class; it was mostly a matter of moving parts of the `MainActivity` to this new fragment to split responsibilities.

Naturally, the `MainActivity` now needs adjustments to use this fragment. First, remove any references to the removed methods and to the `SearchViewModel`—the activity itself no longer needs to access a view model or retrieve data. Instead, add a property of type `SearchFragment` to hold the search fragment, as in [Listing 8.57](#).

[Listing 8.57 Adding a Property for the Fragment](#)

[Click here to view code image](#)

---

```
class MainActivity : AppCompatActivity() {  
  
    private lateinit var searchFragment: SearchFragment  
  
    // ...  
}
```



Adding fragments to an activity is done using *fragment transactions*. To get started, let's add an extension function that encapsulates the boilerplate necessary to add a fragment to an activity. [Listing 8.58](#) shows this new extension from `ViewExtensions.kt`.

[Listing 8.58 Extension Function to Include Fragments](#)

[Click here to view code image](#)

---

```
import android.support.v7.app.AppCompatActivity  
  
// ...  
  
fun AppCompatActivity.replaceFragment(viewGroupId: Int
```

```
supportFragmentManager.beginTransaction()  
    .replace(viewGroupId, fragment) // Replaces gi  
    .commit()  
}
```



Any fragment transaction is initiated via the activity's `supportFragmentManager`. The given `viewGroupId` refers to the placeholder view that shall be replaced by the fragment. With this, you could now add this new fragment to the activity in `onCreate`, as shown in [Listing 8.59](#).

#### **Listing 8.59 Including a Fragment into the UI**

[Click here to view code image](#)

---

```
import com.example.nutrilicious.view.common.replaceFr  
  
class MainActivity : AppCompatActivity() {  
    // ...  
    override fun onCreate(savedInstanceState: Bundle?)  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
        searchFragment = SearchFragment()  
        replaceFragment(R.id.mainView, searchFragment) //  
        navigation.setOnNavigationItemSelected(na  
    }  
}
```



However, this creates a completely new `SearchFragment` every time `onCreate` is called; for instance, when the user rotates the screen or switches to another app and back. This causes any state in the fragment to be cleared after such

actions, which evokes a feeling of discontinuity for the user. Instead, you want to keep the search fragment—and its state—alive as long as Android does not garbage-collect it. Storing a fragment is also done using fragment transactions. Listing 8.60 shows a new extension to `ViewExtensions.kt` that adds a fragment to an activity’s state.

**Listing 8.60 Extension to Store Fragment in Activity’s State**

[Click here to view code image](#)

```
import android.support.annotation.IdRes

// ...

fun AppCompatActivity.addFragmentToState(
    @IdRes containerViewId: Int,
    fragment: Fragment,
    tag: String
) {
    supportFragmentManager.beginTransaction()
        .add(containerViewId, fragment, tag) // Stores the fragment
        .commit()
}
```

Now, instead of plainly creating a new fragment each time in `onCreate`, you can implement a helper method that tries to recover an existing fragment first and only creates one if required. Listing 8.61 shows the necessary code.

**Listing 8.61 Restoring an Existing Fragment**

[Click here to view code image](#)

```
import com.example.nutrilicious.view.common.*

// ...

class MainActivity : AppCompatActivity() {
    // ...
}
```

```
private fun recoverOrBuildSearchFragment() {  
  
    val fragment = supportFragmentManager // Tries to  
        .findFragmentByTag(SEARCH_FRAGMENT_TAG) as? SearchFragment  
  
    if (fragment == null) setUpSearchFragment() else  
  
}  
  
  
private fun setUpSearchFragment() { // Sets up search fragment  
  
    searchFragment = SearchFragment()  
  
    addFragmentToState(R.id.mainView, searchFragment,  
  
}  
  
}
```

The `recoverOrBuildSearchFragment` method first tries to read an existing fragment from the activity's state and otherwise falls back to creating a new one. When creating a new one, the fragment is automatically added to the activity's state for next time. The tag is a unique identifier for the fragment that you can declare above the `MainActivity` as in [Listing 8.62](#).

**Listing 8.62 Adding a Fragment Tag**

[Click here to view code image](#)

```
private const val SEARCH_FRAGMENT_TAG = "SEARCH_FRAGMENT_TAG"  
  
  
class MainActivity : AppCompatActivity() { ... }
```

Next, call the new helper method in `onCreate` to recover the fragment instead of creating a new instance, as shown in [Listing 8.63](#).

**Listing 8.63 Adjusting `onCreate`**

[Click here to view code image](#)

```
override fun onCreate(savedInstanceState: Bundle?) {  
    // ...  
  
    recoverOrBuildSearchFragment() // Replaces SearchF  
  
    replaceFragment(R.id.mainView, searchFragment)  
}
```

You can now delegate to the fragment to handle search intents because this is no longer the activity's responsibility. [Listing 8.64](#) adjusts the intent handler accordingly.

#### **Listing 8.64 Delegating Searches to the Search Fragment**

[Click here to view code image](#)

```
override fun onNewIntent(intent: Intent) {  
  
    if (intent.action == Intent.ACTION_SEARCH) {  
  
        val query = intent.getStringExtra(SearchManager.C  
  
        searchFragment.updateListFor(query) // Uses the  
    }  
  
}
```

The final enhancement in this section is to indicate progress properly using the swipe refresh layout. Currently, the swipe refresh indicator doesn't show up when issuing a search via the menu and will not disappear once it has been triggered by swiping down. You can fix both issues easily inside the `updateListFor` method by setting the swipe refresh state appropriately, as shown in [Listing 8.65](#).

#### **Listing 8.65 Handling Swipe Refresh**

[Click here to view code image](#)

```
private fun updateListFor(searchTerm: String) {  
  
    lastSearch = searchTerm  
  
    swipeRefresh?.isRefreshing = true // Indicates tha
```

```
launch(NETWORK) {  
  
    val foods = searchViewModel.getFoodsFor(searchTerm)  
  
    withContext(UI) {  
  
        (rvFoods?.adapter as? SearchListAdapter)?.setIt  
  
        swipeRefresh?.isRefreshing = false // Indicate  
  
    }  
  
}  
  
}
```

Note that this also uses the safe call operator because the fragment may already be detached from its activity by the time the network request returns, making `swipeRefresh` inaccessible.

Now everything should work as before (plus the swipe refresh functionality). Users do not notice if the app uses fragments, but it does improve the internal structure. In particular, it prevents god activities by separating concerns on a more fine-grained level.

## INTRODUCING FRAGMENTS II: THE FAVORITES FRAGMENT

Now that the app uses fragments, it is time to add the favorites fragment and make the bottom navigation work by showing the appropriate fragment. As always, the first step is to create the required layouts. For this, add a new file `fragmentFavorites.xml` in `res/layout` that defines the layout for the new fragment. This fragment shows the list of user-selected favorite foods. Listing 8.66 shows the corresponding layout.

**Listing 8.66 Layout for the Favorites Fragment**

[Click here to view code image](#)

```
<?xml version="1.0" encoding="utf-8"?>

<android.support.constraint.ConstraintLayout

    xmlns:android="http://schemas.android.com/apk/res-
    android"
    xmlns:app="http://schemas.android.com/apk/res-aut-
    o"
    app:layout_behavior="@string/appbar_scrolling_view-
    behavior"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView

        android:id="@+id/tvHeadline"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        android:padding="@dimen/medium_padding"
        android:text="@string/favorites"
        android:textSize="@dimen/huge_font_size" />

    <android.support.v7.widget.RecyclerView

        android:id="@+id/rvFavorites"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        app:layout_constraintTop_toBottomOf="@+id/tvH-
        istory" />

</android.support.constraint.ConstraintLayout>
```

This simple layout consists of a headline and a recycler view to show all the user's favorite foods. Although this uses the more modern `ConstraintLayout`, you could easily implement this as a vertical `LinearLayout` as well. To finish the layout, you must again add the missing resources as shown in [Listing 8.67](#).

**Listing 8.67 Resources for the Layout**

[Click here to view code image](#)

```
// In res/values/dimens.xml

<dimen name="huge_font_size">22sp</dimen>

// In res/values/strings.xml

<string name="favorites">Favorite Foods</string>
```

That's all the layout needed for this fragment. Now, create a new file `FavoritesFragment.kt` in `view.main` and add the necessary overrides to inflate and initialize the layout components. As shown in [Listing 8.68](#), this follows the same structure as in the search fragment. For now, the fragment uses hard-coded sample data because users cannot select favorites yet.

**Listing 8.68 Implementing the Favorites Fragment**

[Click here to view code image](#)

```
import android.os.Bundle

import android.support.v4.app.Fragment

import android.support.v7.widget.*

import android.view.*

import com.example.nutrilicious.R

import com.example.nutrilicious.model.Food

import kotlinx.android.synthetic.main.fragment_favori
```

```
class FavoritesFragment : Fragment() {  
  
    override fun onCreateView(inflater: LayoutInflater,  
                             container: ViewGroup?,  
                             savedInstanceState: Bundle?)  
    {  
        return inflater.inflate(R.layout.fragment_favorite,  
                             container, false)  
    }  
  
    override fun onViewCreated(view: View, savedInstanceState: Bundle?)  
    {  
        super.onViewCreated(view, savedInstanceState)  
        setUpRecyclerView()  
    }  
  
    private fun setUpRecyclerView() = with(rvFavorites)  
    {  
        adapter = SearchListAdapter(sampleData())  
        layoutManager = LinearLayoutManager(context)  
        addItemDecoration(DividerItemDecoration(  
            context, LinearLayoutManager.VERTICAL  
        ))  
        setHasFixedSize(true)  
    }  
  
    // Temporary! Should use string resources instead of hard-coded strings  
    private fun sampleData(): List<Food> = listOf(  
        Food("00001", "Marshmallow", "Candy and Sweets", true),  
        Food("00002", "Nougat", "Candy and Sweets", true),  
        Food("00003", "Oreo", "Candy and Sweets", true)  
    )
```

```
    )  
}
```

Again, the layout is inflated in `onCreateView` and then all views are initialized in `onViewCreated` because that is when they are ready to be manipulated. This initialization works just fine. However, note how the initialization for the `RecyclerView` is now duplicated—it works just the same way in the search fragment. This is because both use the same adapter and should look exactly the same. The only difference between the two is which items are shown. To avoid code duplication, you should move the logic to a place where both fragments can access it. Because this particular logic is related to the `MainActivity`, let's place it there, as shown in

[Listing 8.69.](#)

[Listing 8.69 Moving Common Logic into MainActivity](#)

[Click here to view code image](#)

---

```
import android.support.v7.widget.*  
  
import com.example.nutrilicious.model.Food  
  
// ...  
  
class MainActivity : AppCompatActivity() {  
  
    // ...  
  
    companion object {  
  
        fun setUpRecyclerView(rv: RecyclerView, list: List<Food>)  
        with(rv) {  
  
            adapter = SearchListAdapter(list)  
  
            layoutManager = LinearLayoutManager(context)  
  
            addItemDecoration(DividerItemDecoration(  
  
                context, LinearLayoutManager.VERTICAL  
  
            ))  
        }  
    }  
}
```

```
        setHasFixedSize(true)  
    }  
}  
}
```

Placing this method in a companion object allows you to call it more conveniently, directly on the activity class. By delegating to this method from both fragments, your code becomes DRY again (“Don’t Repeat Yourself”). Listing 8.70 demonstrates the changes.

### Listing 8.70 Removing Duplicated Code from the Fragments

[Click here to view code image](#)

```
// In FavoritesFragment.kt

private fun setUpRecyclerView() {
    MainActivity.setUpRecyclerView(rvFavorites, sampleFavorites)
}

// In SearchFragment.kt

private fun setUpSearchRecyclerView() {
    MainActivity.setUpRecyclerView(rvFoods)
}
```

This way, the logic to set up the recycler view is encapsulated in one place and the fragments delegate to it. You can now use the fragments to make the bottom navigation menu work. Thanks to the extension that wraps fragment transactions, this is easy now. [Listing 8.71](#) adjusts the navigation listener in `MainActivity` accordingly.

### Listing 8.71 Implementing the Bottom Navigation Menu

[Click here to view code image](#)

```
private val handler = BottomNavigationView.OnNavigationItemSelectedListener { item ->
    when (item.itemId) {
        R.id.navigation_home -> {
            replaceFragment(R.id.mainView, searchFragment)
            return@OnNavigationItemSelectedListener true
        }
        R.id.navigation_my_foods -> {
            replaceFragment(R.id.mainView, favoritesFragment)
            return@OnNavigationItemSelectedListener true
        }
    }
    false
}
```

When clicking to the *Home Screen*, the activity switches to the existing search fragment that it keeps in the `searchFragment` property. For the favorites fragment, this is not necessary. Its state will be based on which foods are stored in the database as favorites. Thus, a new fragment is created whenever the user navigates to the *My Foods Screen*.

Currently, the search fragment will be empty after switching to the favorites fragment and back. So as a final step, the search fragment should remember the most recent search results and use them to populate the list when returning to the search fragment. Listing 8.72 shows the required changes in `SearchFragment`.

#### **Listing 8.72 Retaining the Last Search Results**

[Click here to view code image](#)

```
import com.example.nutrilicious.model.Food
```

```
class SearchFragment : Fragment() {  
    // ...  
  
    private var lastResults = emptyList<Food>()  
  
    override fun onViewCreated(view: View, savedInstanceState: Bundle) {  
        // ...  
  
        (rvFoods?.adapter as? SearchListAdapter)?.setItemViewType(0)  
    }  
  
    // ...  
  
    fun updateListFor(searchTerm: String) {  
        // ...  
  
        launch(NETWORK) {  
            val foods = searchViewModel.getFoodsFor(searchTerm)  
  
            lastResults = foods // Remembers last search results  
            withContext(Dispatchers.Main) {  
                rvFoods?.adapter?.submitList(foods)  
            }  
        }  
    }  
}
```

## STORE USER'S FAVORITE FOODS IN A ROOM DATABASE

The next step is to store a user's favorite foods in a database and show those in the favorites fragment. This gives you another chance to familiarize yourself with Room. Setting it up requires the same three steps as in [Chapter 7, Android App Development with Kotlin: Kudoo App](#):

- Define the entities that should be stored in the database.
- Create the DAOs that offer all desired operations to access the database.

- Implement an abstract subclass of `RoomDatabase` that provides the DAOs.

Because the dependencies for all Android Architecture Components are already included, you can start creating the entities right away. For this app, that's the `Food` class. [Listing 8.73](#) shows the required changes.

#### **Listing 8.73 Making Food an Entity**

[Click here to view code image](#)

```
import android.arch.persistence.room.*  
  
// ...  
  
@Entity(tableName = "favorites") // Signals Room to  
  
data class Food(  
  
    @PrimaryKey val id: String, // Unique identifier  
  
    val name: String,  
  
    val type: String,  
  
    var isFavorite: Boolean = false  
  
) { ... }
```

Only two annotations are necessary to turn the class into an entity that Room can work with. The table name “favorites” better highlights the purpose than the default name “food.” The existing `NDBNO` is used as the primary key.

Next step is to add the DAO to perform queries on the favorites table. To this end, add a new package `data.db` and add a `FavoritesDao` interface to it, as implemented in [Listing 8.74](#).

#### **Listing 8.74 Adding the DAO to Access Favorites**

[Click here to view code image](#)

```
import android.arch.lifecycle.LiveData  
  
import android.arch.persistence.room.*  
  
import android.arch.persistence.room.OnConflictStrategy
```

```
import com.example.nutrilicious.model.Food

@Dao

interface FavoritesDao {

    @Query("SELECT * FROM favorites")

    fun loadAll(): LiveData<List<Food>> // Note LiveData<List<Food>>

    @Query("SELECT id FROM favorites")

    fun loadAllIds(): List<String>

    @Insert(onConflict = IGNORE) // Do nothing if food already exists

    fun insert(food: Food)

    @Delete

    fun delete(food: Food)

}
```

The DAO offers all database-related capabilities the Nutrilicious app needs: fetching all favorites (to display them in the fragment), fetching only the IDs of all favorite foods (to highlight them with a star), and adding and deleting favorites (for when the user clicks on a star). Room makes it straightforward to implement the corresponding queries using basic SQL queries. Also note that `fetchAll` makes use of LiveData so that updates to favorites can be reflected immediately in the favorites fragment.

The third and last step is to add an abstract subclass of `RoomDatabase`, which here is named `AppDatabase`. This always follows the structure so Listing 8.75 simply shows its definition.

**Listing 8.75 Adding the AppDatabase**

[Click here to view code image](#)

```
import android.arch.persistence.room.*  
  
import android.content.Context  
  
import com.example.nutrilicious.model.Food  
  
  
@Database(entities = [Food::class], version = 1)  
  
abstract class AppDatabase : RoomDatabase() {  
  
  
    companion object {  
  
        private var INSTANCE: AppDatabase? = null  
  
  
        fun getInstance(ctx: Context): AppDatabase {  
  
            if (INSTANCE == null) { INSTANCE = buildDatabase() }  
  
            return INSTANCE!!  
  
        }  
  
  
        private fun buildDatabase(ctx: Context) = Room  
            .databaseBuilder(ctx, AppDatabase::class.java,  
                "FoodDatabase")  
            .build()  
  
    }  
  
  
    abstract fun favoritesDao(): FavoritesDao // Provides  
    }  
    < >
```

You can use this as a template. All that needs adjustment for other databases is which entities to include in the `@Database` annotation, and which DAOs to expose. Here, the only DAO is

exposed via the `favoritesDao` method so that Room generates an implementation for it.

In order to decouple the fragments and activities that use data from the database, you should add a `FavoritesViewModel` to the `viewmodel` package.

[Listing 8.76](#) shows this view model.

#### **Listing 8.76 The View Model to Access Favorite Foods**

[Click here to view code image](#)

```
import android.app.Application
import android.arch.lifecycle.*
import com.example.nutrilicious.data.db.*
import com.example.nutrilicious.model.Food
import kotlinx.coroutines.*

class FavoritesViewModel(app: Application) : AndroidViewModel(app) {
    private val dao by lazy { AppDatabase.getInstance(context).foodDao() }

    suspend fun getFavorites(): LiveData<List<Food>> = withContext(Dispatchers.IO) {
        dao.loadAll()
    }

    suspend fun getAllIds(): List<String> = withContext(Dispatchers.IO) {
        fun add(favorite: Food) = launch(Dispatchers.IO) { dao.insert(favorite) }
        fun delete(favorite: Food) = launch(Dispatchers.IO) { dao.delete(favorite) }
        dao.getAllIds()
    }
}
```

The view model wraps the DAO's methods and provides a clean interface for them. Retrieving all the user's favorite foods returns a `LiveData` object so that you can observe

changes. Both methods with return value use `withContext` in order to use the natural return type instead of a `Deferred`, whereas adding and deleting elements is taken care of in dedicated “fire-and-forget” coroutines. Note that this view model is a subclass of `AndroidViewModel` because it needs a reference to the application context to retrieve the database object.

This implementation uses a dedicated dispatcher for all database-related actions. It is declared in `DatabaseDispatcher.kt` inside the `data.db` package, as in [Listing 8.77](#).

#### **Listing 8.77 Coroutine Dispatcher for Database Operations**

[Click here to view code image](#)

```
import kotlinx.coroutines.newSingleThreadContext

val DB = newSingleThreadContext("DB") // Single dedi
```

This concludes the database setup. The next step is to make use of it when the user clicks on a star to mark or unmark it as a favorite. In other words, the `ImageView` needs a click handler. This click handler should be passed into the adapter, and it should know which `Food` object and which list position a click refers to. [Listing 8.78](#) adjusts the `SearchListAdapter` accordingly.

#### **Listing 8.78 Adjusting the Adapter to Handle Star Icon Clicks**

[Click here to view code image](#)

```
class SearchListAdapter( // ...

private val onStarClick: (Food, Int) -> Unit
) : ... {

// ...

inner class ViewHolder(...) : ... {

fun bindTo(food: Food) {
```

```
// ...  
  
        ivStar.setOnClickListener { onStarClick(food, true)  
    }  
}  
}  
  
}
```

Only two additional lines of code are necessary to pass in and assign the click handler. You now have to pass in a handler when creating the adapter in `MainActivity`. That handler should toggle a food as being a favorite. Since this will be possible from both fragments, the toggling logic should be placed in `MainActivity` (or in a separate file) to avoid code duplication.

First, adjust the `RecyclerView` setup as shown in [Listing 8.79](#) to construct the adapter. Also, to avoid a large companion object with lots of effectively static methods, I'd suggest removing the companion object at this point. You will adjust their calls in the fragments later.

#### **Listing 8.79 Creating the Adapter with Click Listener**

[Click here to view code image](#)

---

```
class MainActivity : AppCompatActivity() {  
  
    // ...  
  
    fun setUpRecyclerView(rv: RecyclerView, list: List<  
        with(rv) {  
            adapter = setUpSearchListAdapter(rv, list)  
  
            // ...  
        }  
    }  
  
    private fun setUpSearchListAdapter(rv: RecyclerView,  
        SearchListAdapter(items,  
        )
```

```
        onStarClick = { food, layoutPosition -> //  
            toggleFavorite(food)  
            rv.adapter.notifyItemChanged(layoutPosition)  
        })  
    }  
}
```



The adapter creation is now encapsulated into its own method because it has become a little more complex. The action for `onStarClick` toggles the food that corresponds to the click as a favorite and notifies the adapter of the change to that particular item. This causes that single item to be redrawn so that the star icon updates correctly—thus the need for the layout position in the click handler.

Now it's time to implement the method that toggles a favorite. For this, the `MainActivity` needs a reference to the `FavoritesViewModel` in order to add or delete favorites. [Listing 8.80](#) presents the required changes.

#### [Listing 8.80 Toggling Favorites on Star Icon Click](#)

[Click here to view code image](#)

---

```
import com.example.nutrilicious.viewmodel.FavoritesVi  
  
class MainActivity : AppCompatActivity() {  
    // ...  
    private lateinit var favoritesViewModel: FavoritesV  
  
    override fun onCreate(savedInstanceState: Bundle?)  
    // ...  
    favoritesViewModel = getViewModel(FavoritesViewM  
}  
// ...
```

```
private fun toggleFavorite(food: Food) {

    val wasFavoriteBefore = food.isFavorite

    food.isFavorite = food.isFavorite.not() // Adjusts the database

    if (wasFavoriteBefore) {

        favoritesViewModel.delete(food)

        toast("Removed ${food.name} from your favorites")

    } else {

        favoritesViewModel.add(food)

        toast("Added ${food.name} as a new favorite of yours")

    }
}
```

Toggling a food as favorite changes the corresponding property in the `Food` object and also triggers an insert or delete in the database. This automatically makes them appear or disappear in the favorites fragment due to the use of `LiveData`. Toast messages give the user feedback on his or her action. Note that they may appear before the database operation is actually completed (because `add` and `delete` use `launch` internally), but this will be unnoticeable for users. As you may expect, creating toasts this way is enabled by an extension function in `ViewExtensions.kt`, the one shown in [Listing 8.81](#).

#### **Listing 8.81 Extension Function to Create Toasts**

[Click here to view code image](#)

---

```
import android.widget.Toast

// ...

fun AppCompatActivity.toast(msg: String) {
```

```
        Toast.makeText(this, msg, Toast.LENGTH_SHORT).show()  
    }  
}
```

This concludes the changes to the activity. What is left are the changes to the two fragments: The favorites fragment should display all favorites, and the search fragment should indicate which of the found foods are already favorites.

First, let's adjust the `FavoritesFragment`. It should now use the `FavoritesViewModel` to retrieve all favorites and observe changes to the `LiveData` to update the `RecyclerView`. So at this point, the sample data can be removed as well. Listing 8.82 shows the changes.

**Listing 8.82 Favorites Fragment with View Model**

[Click here to view code image](#)

---

```
import android.content.Context  
  
import com.example.nutrilicious.view.common.getViewModel  
  
import com.example.nutrilicious.viewmodel.FavoritesVi  
  
import kotlinx.coroutines.android.UI  
  
import kotlinx.coroutines.launch  
  
import android.arch.lifecycle.Observer  
  
// ...  
  
class FavoritesFragment : Fragment() {  
  
    private lateinit var favoritesViewModel: FavoritesV  
  
    override fun onAttach(context: Context?) {  
  
        super.onAttach(context)  
  
        favoritesViewModel = getViewModel(FavoritesViewMo  
  
    }  
}
```

```
// ...  
  
    override fun onViewCreated(view: View, savedInstanceState: Bundle)  
        super.onViewCreated(view, savedInstanceState)  
  
        val favoritesViewModel = ViewModelProvider.get(FavoritesViewModel::class)  
        val favorites = favoritesViewModel.getFavorites()  
  
        favorites.observe(this@FavoritesFragment, Observer {  
            foods?.let {  
                launch(UI) { (rvFavorites.adapter as SearchableAdapter)?.submitList(it)  
                }  
            }  
        })  
    }  
  
  
private fun observeFavorites() = launch { // Update the UI when  
    val favorites = favoritesViewModel.getFavorites()  
    favorites.observe(this@FavoritesFragment, Observer {  
        foods?.let {  
            launch(UI) { (rvFavorites.adapter as SearchableAdapter)?.submitList(it)  
            }  
        }  
    })  
}  
  
  
private fun setUpRecyclerView() {  
    (activity as MainActivity)?.setUpRecyclerView(rvFavorites)  
}  
}
```

This fragment gets a new property to hold the LiveData of foods. It can be initialized already in `onAttach` because it is independent of any views. The `observeFavorites` method reflects changes on the LiveData in the `RecyclerView`. The `setUpRecyclerView` method is now called by accessing the fragment's `activity` property to get a reference to the `MainActivity` because it's no longer a companion object method, and it passes in an empty list to start with.

The favorites fragment now displays the correct list of favorite foods and the user is able to choose which foods those are.

However, there's one more step: indicating favorites in the search fragment. It gets its data from the API, so you have to use the NDBNO to connect it to the corresponding entry in the database, as shown in Listing 8.83.

**Listing 8.83 Search Fragment**

[Click here to view code image](#)

```
import com.example.nutrilicious.viewmodel.FavoritesVi

// ...

class SearchFragment : Fragment() {

    // ...

    private lateinit var favoritesViewModel: Favorites\

    override fun onAttach(context: Context?) {

        // ...

        favoritesViewModel = getViewModel(FavoritesViewMo

    }

    // ...

    private fun updateListFor(searchTerm: String) {

        lastSearch = searchTerm

        swipeRefresh?.isRefreshing = true

        launch {

            val favoritesIds: List<String> = favoritesViewM

            val foods: List<Food> = searchViewModel.getFood

            .onEach { if (favoritesIds.contains(it.id))

                lastResults = foods
            }
        }
    }
}
```

```
    withContext(UI) { ... }

}

}

private fun setUpSearchRecyclerView() {
    (activity as? MainActivity)?.setUpRecyclerView(r
}

}
```

The view model takes care of retrieving the IDs of all favorites, allowing you to augment the data retrieved from the API with the information regarding whether it is a favorite. Like `forEach`, `onEach` performs the given action on each item. But in contrast to `forEach`, it returns the resulting collection, here the list of foods with their adjusted `isFavorite` property. Note that the setup method for the recycler view is adjusted to call the member method correctly.

With this, users are able to choose their favorite foods, view them in the *My Foods Screen*, see which found foods already are favorites, and add or remove favorites by clicking on the star icon. This is a working app but not yet particularly helpful, except to discover foods users were not previously aware existed. In the following section, you dive deeper into the USDA API to retrieve and show detailed nutritional information and help users make healthier diet choices.

# FETCHING DETAILED NUTRITION DATA FROM THE USDA FOOD REPORTS API

Accessing another endpoint of the USDA API is fairly easy at this point because you can build upon the existing code. To fetch nutrition details from the USDA Food Reports API,<sup>9</sup> you just have to add the corresponding GET request to the `UsdaApi` interface, as shown in [Listing 8.84](#).

9. <https://ndb.nal.usda.gov/ndb/doc/apilist/API-FOOD-REPORTV2.md>

## Listing 8.84 Adding a New Endpoint to the API Interface

[Click here to view code image](#)

```
@GET("V2/reports?format=json")  
  
fun getDetails(  
    @Query("ndbno") id: String, // Only no  
    @Query("type") detailsType: Char = 'b' // b = bas  
): Call<DetailsWrapper<DetailsDto>>
```

With this, the `UsdaApi` interface now allows two different GET requests. This second one appends “V2/reports” to the base URL and accepts the NDBNO of the requested food. You could again retrieve the raw JSON data first. But this time, let’s map the data to DTOs directly. The structure is similar to the Search API, with a `DetailsWrapper` enclosing the actual `DetailsDto` to navigate down the JSON structure. These wrappers and DTOs will be created next. Setting the result type to “b” tells the API to return basic info, which is already more than detailed enough for this app.

I’d recommend adding another JSON file with the result format<sup>10</sup> to the project (for instance, in the `sampledata` directory) to easily explore it. You can call it `detailsFormat.json` and use the *Reformat Code* action to fix line breaks and indentation if necessary. [Listing 8.85](#) portrays a relevant chunk of the JSON data.

10. [https://api.nal.usda.gov/ndb/V2/reports?ndbno=09070&type=b&format=json&api\\_key=DEMO\\_KEY](https://api.nal.usda.gov/ndb/V2/reports?ndbno=09070&type=b&format=json&api_key=DEMO_KEY)

**Listing 8.85 JSON Format for Nutrition Details**

[Click here to view code image](#)

```
{  
  "foods": [  
    {  
      "food": {  
        "sr": "Legacy",  
        "type": "b",  
        "desc": {  
          "ndbno": "09070",  
          "name": "Cherries, sweet, raw",  
          "ds": "Standard Reference",  
          "manu": "",  
          "ru": "g"  
        },  
        "nutrients": [  
          {  
            "nutrient_id": "255",  
            "name": "Water",  
            "derivation": "NONE",  
            "group": "Proximates",  
            "unit": "g",  
            "value": "82.25",  
            "measures": [ ... ]  
          }, ...  
        ]  
      }  
    }  
  ]  
}
```

```
    }
}

]
}
```

This time, the actual data you need is wrapped into the properties `foods` and `food` (just like `list` and `item` before). The list of nutrients is long and contains everything from water to macros to vitamins, minerals, and fats. All values refer to 100 grams of the food. For instance, you can see from [Listing 8.85](#) that 100g of cherries contain 82.25g of water. In this app, you will not use the alternative measures that come with the result (such as oz, cups, or pieces).

Let's map this data to DTOs. In `data.network.dto`, add a new file `DetailsDtos.kt` into which you can place your DTOs. [Listing 8.86](#) provides the DTO wrappers that navigate to the interesting data.

#### **Listing 8.86 DTO Wrappers**

[Click here to view code image](#)

---

```
import com.squareup.moshi.JsonClass

@JsonClass(generateAdapter = true)

class FoodsWrapper<T> {

    var foods: List<T> = listOf() // Navigates down the
}

@JsonClass(generateAdapter = true)

class FoodWrapper<T> {

    var food: T? = null // Navigates down the
}
```

```
typealias DetailsWrapper<T> = FoodsWrapper<FoodWrapper<T>>
```

When writing these DTOs, be careful about which JSON property holds a list (indicated by square brackets) and which holds an object (indicated by curly braces), and map them accordingly. In this example, `foods` is a list and `food` is an object; the Food Reports API allows fetching details for multiple foods in one request, thus the *list* of foods.

Next, Listing 8.87 provides the DTO declarations with property names matching those in the JSON data, so that Moshi knows how to map them.

#### **Listing 8.87 Details DTOs**

[Click here to view code image](#)

```
@JsonClass(generateAdapter = true)

class DetailsDto(val desc: DescriptionDto, val nutrients: List[NutrientDto])

init {
    nutrients.forEach { it.detailsId = desc.ndbno }
}

@ JsonClass(generateAdapter = true)

class DescriptionDto { // Property names must match JSON
    lateinit var ndbno: String
    lateinit var name: String
}

@ JsonClass(generateAdapter = true)

class NutrientDto { // Property names must match JSON
    var nutrient_id: Int? = null // Cannot use lateinit
    var name: String = ""
    var percent_dv: Double? = null
    var amount: Double? = null
    var unit: String? = null
}
```

```
    var detailsId: String? = null // Only field not co
    lateinit var name: String
    lateinit var unit: String
    var value: Float = 0f
    lateinit var group: String
}
```

The response contains a description with the NDBNO and name of the requested food, followed by the list of nutrient details. The `NutrientDto` contains one property that is not populated from the JSON data, namely the `detailsId`. This is used to reference which food (identified by the NDBNO) the nutrient details belong to. It is initialized in the `DetailsDto` when merging the description and nutrient data. This works similar to a foreign key in SQL.

You are now all set to retrieve and map detailed nutrition data from the USDA API. Finally, you can map this to domain classes that are decoupled from the JSON property names and formats. So in `model`, add a new file `FoodDetails.kt` that will contain data classes for the food details. Listing 8.88 presents the required data classes.

#### Listing 8.88 Domain Classes for Food Details

[Click here to view code image](#)

```
import com.example.nutrilicious.data.network.dto.*

data class FoodDetails(
    val id: String,
    val name: String,
    val nutrients: List<Nutrient>
) {
    constructor(dto: DetailsDto) : this(
        id = dto.id,
        name = dto.name,
        nutrients = dto.nutrients
    )
}
```

```
        dto.desc.ndbno,  
  
        dto.desc.name,  
  
        dto.nutrients.map(::Nutrient)  
    )  
}  
  
  
data class Nutrient(  
  
    val id: Int,  
  
    val detailsId: String,  
  
    val name: String,  
  
    val amountPer100g: Float,  
  
    val unit: String,  
  
    val type: NutrientType  
) {  
  
    constructor(dto: NutrientDto) : this(  
  
        dto.nutrient_id!!,  
  
        dto.detailsId!!,  
  
        dto.name,  
  
        dto.value,  
  
        dto.unit,  
  
        NutrientType.valueOf(dto.group.toUpperCase())  
    )  
}  
  
  
enum class NutrientType {  
  
    PROXIMATES, MINERALS, VITAMINS, LIPIDS, OTHER  
}
```

There are three domain classes. The entry point is the `FoodDetails` class that contains a list of nutrients, each of which has a specific nutrient type. Like last time, secondary constructors map the DTOs to these domain classes. In case a late-initialized property was not populated by Moshi or a nullable property remains `null`, the code crashes immediately when trying to map at runtime (and you would be able to identify the causing property easily). As for the nutrient type, there are only five possible values returned by the API, so you can map this to an enum.

This concludes accessing the *Food Reports API*. You can sanity-check it in `MainActivity.onCreate` with a temporary test call as shown in Listing 8.89.

**Listing 8.89 Making a Test Request**

[Click here to view code image](#)

```
import com.example.nutrilicious.data.network.*  
  
import kotlinx.coroutines.launch  
  
  
class MainActivity : AppCompatActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?)  
  
        // ...  
  
        launch(NETWORK) { usdaApi.getDetails("09070").ex  
  
    }  
  
}
```

When running the app, you should see a response with the details in JSON format in the Logcat. Now you are ready to implement a detail activity that presents the user with more actionable information about each food in the app.

## INTEGRATING THE DETAILS ACTIVITY

In this section, you will create a second activity to display the nutritional information about a selected food. To this end, create a new package `view.details`, and inside it generate a new *Empty Activity* along with its layout file using Android Studio's wizard (using right-click, *New*, *Activity*, and then *Empty Activity*). Name it `DetailsActivity` and use the default name `activity_details.xml` for the layout.

As always, let's deal first with the layout. This layout file is quite long because it has four sections for macronutrients (“proximates” in the API response), vitamins, minerals, and lipids (fats). Each section will have a headline and a text view that is populated with data programmatically, followed by a horizontal divider. This divider is a custom view defined in `res/layout/vertical_divider.xml` as shown in [Listing 8.90](#).

### **Listing 8.90 Horizontal Divider Layout**

[Click here to view code image](#)

```
<?xml version="1.0" encoding="utf-8"?>

<View xmlns:android="http://schemas.android.com/apk/r
    android:layout_width="match_parent"
    android:layout_height="@dimen/divider_height"
    android:minHeight="@dimen/divider_minheight"
    android:layout_marginTop="@dimen/divider_margin"
    android:layout_marginBottom="@dimen/divider_margi
    android:background="?android:attr/listDivider" />
```

This renders as a horizontal line that is 2dp thick but at least 1px to avoid being invisible. For the dimensions, add the missing resources from [Listing 8.91](#) to `res/values/dimens.xml`.

#### Listing 8.91 Divider Dimensions

[Click here to view code image](#)

```
<dimen name="divider_height">2dp</dimen>  
<dimen name="divider_minheight">1px</dimen>  
<dimen name="divider_margin">5dp</dimen>
```



With this, you are ready to implement the details activity layout, as shown in Listing 8.92. This code belongs in `res/layout/activity_details.xml`.

#### Listing 8.92 Details Activity Layout

[Click here to view code image](#)

```
<?xml version="1.0" encoding="utf-8"?>  
  
<ScrollView xmlns:android="http://schemas.android.com  
    xmlns:tools="http://schemas.android.com/tools"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
  
    <LinearLayout  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content"  
        android:orientation="vertical"  
        android:padding="@dimen/medium_padding"  
        tools:context=".view.detail.FoodDetailsActivi  
  
        <TextView  
            android:id="@+id/tvFoodName"  
            android:layout_width="match_parent"  
            android:layout_height="wrap_content"
```

```
    android:layout_gravity="center"
    android:gravity="center"
    android:padding="@dimen/medium_padding"
    android:textSize="@dimen/huge_font_size"
```

```
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/proximates"
    android:textColor="@android:color/darker_gray"
    android:textSize="@dimen/medium_font_size"
```

```
<TextView
    android:id="@+id/tvProximates"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:lineSpacingMultiplier="1.1" />
```

```
<include layout="@layout/vertical_divider" />
```

```
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/vitamins"
    android:textColor="@android:color/darker_gray"
    android:textSize="@dimen/medium_font_size"
```

```
<TextView  
    android:id="@+id/tvVitamins"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content" />
```

```
<include layout="@layout/vertical_divider" />
```

```
<TextView  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:text="@string/minerals"  
    android:textColor="@android:color/darker_gray"  
    android:textSize="@dimen/medium_font_size" />
```

```
<TextView  
    android:id="@+id/tvMinerals"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content" />
```

```
<include layout="@layout/vertical_divider" />
```

```
<TextView  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:text="@string/lipids"  
    android:textColor="@android:color/darker_gray"  
    android:textSize="@dimen/medium_font_size" />
```

```
<TextView  
    android:id="@+id/tvLipids"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content" />  
  
</LinearLayout>  
  
</ScrollView>
```

It is a simple `ScrollView` containing a vertical `LinearLayout` with all the sections stacked up. There is nothing particular to note about this layout; however, to make it work, you have to add the missing string resources for the headline, shown in [Listing 8.93](#).

#### **Listing 8.93 String Resources For Headlines**

[Click here to view code image](#)

---

```
<string name="proximates">Proximates</string>  
  
<string name="minerals">Minerals</string>  
  
<string name="vitamins">Vitamins</string>  
  
<string name="lipids">Lipids</string>
```

At this point, the new activity is already usable, although it will only show static texts and no nutrition data yet. To incorporate it into the app, it should be shown when the user clicks on a food item in a `RecyclerView`. Handling these clicks requires a small change in the adapter, as shown in [Listing 8.94](#).

#### **Listing 8.94 Adding an Item Click Listener to the Adapter**

[Click here to view code image](#)

```
class SearchListAdapter(...,  
  
    private val onItemClickListener: (Food) -> Unit,  
  
    private val onStarClick: (Food, Int) -> Unit  
) : RecyclerView.Adapter<ViewHolder>() {  
  
    // ...  
  
  
    inner class ViewHolder(...) : ... {  
  
  
        fun bindTo(food: Food) {  
  
            // ...  
  
            containerView.setOnClickListener { onItemClickListener(  
                }  
  
            }  
  
        }  
  
    }  
  
    
```

The `containerView` property corresponds to the whole list item, so it should be the target for this click handler. The handler accepts the food that corresponds to the clicked item in order to pass on the NDBNO to the `DetailsActivity` and show the correct data. This behavior is defined in `MainActivity`, as shown in Listing 8.95.

**Listing 8.95 Defining the Click Handler for List Items**

[Click here to view code image](#)

```
import com.example.nutrilicious.view.details.Details/  
  
// ...  
  
class MainActivity : AppCompatActivity() {  
  
    private fun setUpSearchListAdapter(rv: RecyclerView  
  
        SearchListAdapter(items,  
  
        onItemClickListener = { startDetailsActivity(it) },  
  
    )
```

```
    onStarClick = { ... }

}

private fun startDetailsActivity(food: Food) {
    val intent = Intent(this, DetailsActivity::class.java)
        putExtra(FOOD_ID_EXTRA, food.id) // Stores the food id
    }
    startActivity(intent) // Switches to DetailsActivity
}
}
```

Note the use of `apply` to initialize the `Intent` and add an extra that carries the NDBNO of the food to display. The `FOOD_ID_EXTRA` identifier is declared as a file-level property in `DetailsActivity.kt` (see [Listing 8.96](#)).

**Listing 8.96 Identifier for the Intent Extra**

[Click here to view code image](#)

---

```
const val FOOD_ID_EXTRA = "NDBNO"

class DetailsActivity : AppCompatActivity() { ... }
```

You can now click on any list item in the app to be taken to the details activity. However, it shows only static headlines at this stage. So the next step is to actually retrieve the desired data. Again, the activity should get its data from a view model. To this end, add a new file `DetailsViewModel.kt` to the `viewmodel` package, as in [Listing 8.97](#).

**Listing 8.97 View Model for Details**

[Click here to view code image](#)

```
import android.arch.lifecycle.ViewModel

import com.example.nutrilicious.data.network./*

import com.example.nutrilicious.data.network.dto./*

import com.example.nutrilicious.model.FoodDetails

import kotlinx.coroutines.withContext

import retrofit2.Call

class DetailsViewModel : ViewModel() {

    suspend fun getDetails(foodId: String): FoodDetails {
        val request: Call<DetailsWrapper<DetailsDto>> = usdaApi.getFood(foodId)
        val detailsDto: DetailsDto = withContext(NETWORK) {
            request.execute().body()?.foods?.get(0)?.food ?:
        } ?: return null

        return FoodDetails(detailsDto)
    }
}
```

This view model defines only one method, which retrieves the details for a given food and provides a cleaner interface for the network call by wrapping the `usdaApi` object. Executing the Retrofit call works as in `SearchViewModel`, this time accessing only the first item of the `foods` list (there is only one because you pass a single NDBNO to the API) and its `food` property. If this ends up being `null` or throwing an exception, the `withContext` block passes this on. Consequently, the whole method returns `null` because of the

elvis operator after `withContext`. In the success case, the DTO is mapped to a `FoodDetails` object that is returned.

Next, remove the test call in `MainActivity.onCreate`. The API is now used by the view model as it should be. In the `DetailsActivity`, add a property for the view model and initialize it in `onCreate` as shown in [Listing 8.98](#).

**Listing 8.98 Adding the View Model to the DetailsActivity**

[Click here to view code image](#)

```
import android.support.v7.app.AppCompatActivity

import android.os.Bundle

import com.example.nutrilicious.R

import com.example.nutrilicious.view.common.getViewModel

import com.example.nutrilicious.viewmodel.DetailsViewModel

class DetailsActivity : AppCompatActivity() {

    private lateinit var detailsViewModel: DetailsViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        // ...
        detailsViewModel = getViewModel(DetailsViewModel::class.java)
    }
}
```

Finally, you can read the desired NDBNO from the intent extra (which was attached by the item click handler), fetch the data for it, and show it to the user. [Listing 8.99](#) reads the NDBNO and performs the request as the first step.

**Listing 8.99 Using the View Model in the Details Activity**

[Click here to view code image](#)

```
import kotlinx.coroutines.android.UI

import kotlinx.coroutines.*

// ...

class DetailsActivity : AppCompatActivity() {

    // ...

    override fun onCreate(savedInstanceState: Bundle?) {
        // ...

        val foodId = intent.getStringExtra(FOOD_ID_EXTRA)

        updateUiWith(foodId)

    }

    private fun updateUiWith(foodId: String) {
        if (foodId.isBlank()) return

        launch {
            val details = detailsViewModel.getDetails(foodId)
            withContext(UI) { bindUi(details) }
        }
    }
}
```

After reading the intent extra, a helper function handles the network request and updates the UI. Here, `launch` does not need an explicit dispatcher because `getDetails` dispatches to the network context internally. Note that, as a suspending function, `getDetails` behaves synchronously by default so that the `details` variable is ready to be used in the line after its declaration. Listing 8.100 shows the code to bind the data to the views.

**Listing 8.100 Displaying the Data**

[Click here to view code image](#)

```
import com.example.nutrilicious.model.*

import kotlinx.android.synthetic.main.activity_detail

// ...

class DetailsActivity : AppCompatActivity() {

    // ...

    private fun bindUi(details: FoodDetails?) {

        if (details != null) {

            tvFoodName.text = "${details.name} (100g)"

            tvProximates.text = makeSection(details, NutrientType.PROXIMATES)

            tvMinerals.text = makeSection(details, NutrientType.MINERALS)

            tvVitamins.text = makeSection(details, NutrientType.VITAMINS)

            tvLipids.text = makeSection(details, NutrientType.LIPIDS)

        } else {

            tvFoodName.text = getString(R.string.no_data)

        }

    }

    private fun makeSection(details: FoodDetails, forType: NutrientType): String {

        details.nutrients.filter { it.type == forType }

            .joinToString(separator = "\n", transform = {

                private fun renderNutrient(nutrient: Nutrient): String {

                    val displayName = name.substringBefore(",") // =

                    "$displayName: $amountPer100g$unit"

                }

            })

    }

}
```

```
}
```



Displaying the data is just a matter of reading out each nutrient and showing it in the corresponding text view. To avoid code duplication, `makeSection` filters to the nutrients of a specific type and shows each in a new line, with its name and amount in 100g of the food. Displaying each single nutrient is handled by `renderNutrient`. If you want to, you can extend the app to include the nutrient type “other” as well.

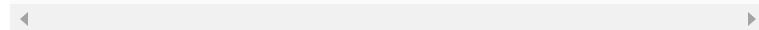
Add the missing string resource in `res/values/strings.xml` to display “No data available” in case something goes wrong and the API call returns `null` (see Listing 8.101).

#### **Listing 8.101 String Resource Indicating No Data**

[Click here to view code image](#)

---

```
<string name="no_data">No data available</string>
```



As a final step, you can allow users to navigate up by making the `DetailsActivity` a child of the `MainActivity`.

Listing 8.102 adjusts the `AndroidManifest.xml` accordingly.

#### **Listing 8.102 Providing Up Navigation**

[Click here to view code image](#)

---

```
<application ...>

    ...

    <activity android:name=".view.details.DetailsActi
        android:parentActivityName=".view.main.MainActivity"

        <meta-data
            android:name="android.support.PARENT_ACTIVITY"
            android:value=".view.main.MainActivity" />

    </activity>
```

```
</application>
```

With this, the Nutrilicious app is already a lot more actionable for users. They can explore foods and see how much of each nutrient it contains, for instance, to see if their favorite foods have a good nutrient profile.

## STORING FOOD DETAILS IN THE DATABASE

In the following, you'll cache food details in the database when first retrieved. This avoids unnecessary network requests when the user frequently accesses the same foods, such as his or her favorites. To arbitrate between the database and the API as alternative data sources for food details, a repository will be introduced, as described in "Guide to App Architecture" on the Android Developers website.<sup>11</sup>

11. <https://developer.android.com/topic/libraries/architecture/guide.html>

Because Room is already included in the project, you can start creating the entities immediately. In this case, this means turning the `FoodDetails` class into an entity. In [Listing 8.103](#), the required annotations are added to the data classes in `FoodDetails.kt`.

**Listing 8.103 Defining the Entities**

[Click here to view code image](#)

```
import android.arch.persistence.room.*  
  
// ...  
  
@Entity(tableName = "details")  
@TypeConverters(NutrientListConverter::class) // Is  
  
data class FoodDetails(  
  
    @PrimaryKey val id: String,  
  
    // ...
```

```
    ) { constructor(dto: DetailsDto) : this(...) }
```

```
    @TypeConverters(NutrientTypeConverter::class)
```

```
    data class Nutrient(...) { ... }
```

Because the nutrients don't need to be queried on their own, they can be inlined into the details table using type converters to keep the schema simple. Thus, only the `FoodDetails` class gets an `@Entity` annotation and a primary key. To inline the `List<Nutrient>` it carries, it uses a `NutrientListConverter` that defines how to map the list to a string and back. Similarly, the `Nutrient` class uses a type converter to store its `NutrientType` as a string and restore it back.

These two converters must be implemented manually. For this, create a file `Converters.kt` under `data.db`. First, the `NutrientListConverter` uses Moshi to encode the nutrient list as a JSON string and decode it when reading from the database. This is done in [Listing 8.104](#).

**Listing 8.104 Type Converter for `List<Nutrient>`**

[Click here to view code image](#)

```
import android.arch.persistence.room.TypeConverter

import com.example.nutrilicious.model.*

import com.squareup.moshi.*
```

```
class NutrientListConverter {

    private val moshi = Moshi.Builder().build()

    private val nutrientList = Types.newParameterizedType(
        List::class.java, Nutrient::class.java
    )

    private val adapter = moshi.adapter<List<Nutrient>>
```

```
    @TypeConverter

    fun toString(nutrient: List<Nutrient>): String = ac

    @TypeConverter fun toListOfNutrient(json: String):
        = adapter.fromJson(json) ?: emptyList()
    }
```

The main work is to set up a Moshi adapter that knows how to transform a `List<Nutrient>` into a string. Setting it up is a little verbose and could be encapsulated, but this is only used once in this app so we'll go with it. To tell Room which methods it should use to map types to and from the database, you must annotate them with `@TypeConverter`. This way, if Room encounters a `List<Nutrient>`, it knows it can use `NutrientListConverter.toString`.

The second converter is simpler because it only needs the Kotlin-generated enum members, as shown in [Listing 8.105](#). This is also placed into `Converters.kt`.

**Listing 8.105 Type Converter for NutrientType**

[Click here to view code image](#)

---

```
class NutrientTypeConverter {

    @TypeConverter
    fun toString(nutrientType: NutrientType) = nutrient

    @TypeConverter
    fun toNutrientType(name: String) = NutrientType.va]
}
```

This enables Room to use these two type converters. Note that, in `FoodDetails.kt`, the classes use the

`@TypeConverters` (with an “s” at the end) to denote which converters to use, whereas the converters themselves use `@TypeConverter` to tell Room that it’s a converter method.

The next step is to add the DAO. So in `data.db`, add a new `DetailsDao.kt` as in [Listing 8.106](#).

**Listing 8.106 DAO for Food Details**

[Click here to view code image](#)

```
import android.arch.persistence.room.*  
  
import com.example.nutrilicious.model.FoodDetails  
  
  
@Dao  
  
interface DetailsDao {  
  
    @Query("SELECT * FROM details WHERE id = :ndbno")  
    fun loadById(ndbno: String): FoodDetails?  
  
  
    @Insert(onConflict = OnConflictStrategy.REPLACE)  
    fun insert(food: FoodDetails)  
}
```

Regarding this details entity, the app must be able to insert new data (to cache it in the database) and retrieve the data for a specific food to display it in the `DetailsActivity`.

The last step is different from before. There is already an `AppDatabase`, so you can extend that with the new entity (because both entities should be stored in the same database). [Listing 8.107](#) includes the new entity and exposes the `DetailsDao`.

**Listing 8.107 Extending the AppDatabase**

[Click here to view code image](#)

```
import com.example.nutrilicious.model.FoodDetails

// ...

@Database(entities = [Food::class, FoodDetails::class]
version = 2)

abstract class AppDatabase : RoomDatabase() {

    // ...

    abstract fun favoritesDao(): FavoritesDao

    abstract fun detailsDao(): DetailsDao // Now expos

}
```

The first two changes are inside the `@Database` annotation. First, the `FoodDetails` class is added to the array of entities to include in the database. Second, the version is increased by one because the schema has changed (there is a new entity). Lastly, there is a new method to get an implementation of the new DAO.

If you run the app now, it will clash with the existing database on your device. During development, you can use destructive migration so that Room simply deletes the old database. This will cause any stored favorites to disappear, but nothing more. Listing 8.108 adjusts `buildDatabase` to enable destructive migration in debug builds (during development).

**Listing 8.108 Enabling Destructive Database Migration in Development**

[Click here to view code image](#)

```
import com.example.nutrilicious.BuildConfig

// ...

private fun buildDatabase(ctx: Context) = Room

    .databaseBuilder(ctx, AppDatabase::class.java, "A

    .apply { if (BuildConfig.DEBUG) fallbackToDestruct

    .build()
```

The `BuildConfig.DEBUG` property is true when working in Android Studio or exporting an unsigned APK to test on your device but not when you finally publish a signed APK. Alternately, you could remove the old database manually from your AVD by using the *Device File Explorer* to remove the directory `data/data/<YOUR_PACKAGE_NAME>/databases`.

The database is now extended to store food details in addition to favorite foods. For food details, there are now two competing data sources: the USDA API and the database. If available, the database should be used because this avoids network requests, improves performance, and works offline. To enforce this behavior, you introduce a *repository* that all consumers should use to access details data—it is the single source of truth for this data.

For the repository, add a new class `DetailsRepository` to the `data` package. This repository offers a clean interface for all operations related to `FoodDetails` and uses both the network and the database. [Listing 8.109](#) shows its implementation.

**Listing 8.109 Details Repository as Single Source of Truth**

[Click here to view code image](#)

---

```
import android.content.Context

import com.example.nutrilicious.data.db.*
import com.example.nutrilicious.data.network.*
import com.example.nutrilicious.data.network.dto.*
import com.example.nutrilicious.model.FoodDetails

import kotlinx.coroutines.*
import retrofit2.Call

class DetailsRepository(ctx: Context) {
```

```
private val detailsDao by lazy { AppDatabase.getIns

fun add(details: FoodDetails) = launch(DB) { detail

suspend fun getDetails(id: String): FoodDetails? {

    return withContext(DB) { detailsDao.loadById(id)

    ?: withContext(NETWORK) { fetchDetailsFromApi

        .also { if (it != null) this.add(it) } // }

}

private suspend fun fetchDetailsFromApi(id: String)

    val request: Call<DetailsWrapper<DetailsDto>> = l

    val detailsDto: DetailsDto = withContext(NETWORK)

        request.execute().body()?.foods?.get(0)?.food

    } ?: return null

}

return FoodDetails(detailsDto)

}

}


```

First, the repository has a reference to the `DetailsDao` to access the database. This is used to add new entries to the database via `this.add`. It is also used in `getDetails` to probe for data. If none is available in the database, the method falls back to a network call using `fetchDetailsFromApi` and caches the retrieved data in the database for subsequent calls. Note that, in a more fleshed-out implementation, you may want to invalidate database entries after a certain time to allow for updates of the data.

The code for the network call is taken from `DetailsViewModel` that now uses the repository as its single source of truth for data, as shown in Listing 8.110.

#### Listing 8.110 Using the Repository in the View Model

[Click here to view code image](#)

```
import com.example.nutrilicious.data.DetailsRepository
// ...
class DetailsViewModel(app: Application) : AndroidViewModel {
    private val repo = DetailsRepository(app)
    suspend fun getDetails(foodId: String): FoodDetails
}
```

As you can see, the view model now simply delegates its task to the repository. Because the repository requires a context, the view model now extends `AndroidViewModel`.

Your app should now cache the details for foods when you first access them. In the emulator, the network call may take a few seconds to complete so that you should notice subsequent clicks on the same food to show the data faster. Accordingly, there should be no log entries from network calls in your Logcat when clicking the same food a second time.

## ADDING RDIs FOR ACTIONABLE DATA

In this section, you will further improve this app by making the data more actionable for users. More specifically, the app should show how much of the *recommended daily intake* (RDI) each of the nutrient contents represents. For instance, users should see instantly that 100g of raw spinach have 20.85% of a person's daily need of iron.

RDI information will be stored statically in the app in a simple map. To this end, create a new file `RDIs.kt` in the `model`

package. In order to store the RDIs, Listing 8.111 introduces classes to properly represent the data.

### **Listing 8.111 Classes to Represent RDIs**

[Click here to view code image](#)

```
data class Amount(val value: Double, val unit: WeightUnit)

enum class WeightUnit {
    GRAMS, MILLIGRAMS, MICROGRAMS, KCAL, IU
}
```

To keep things simple, the app uses roughly averaged RDIs for adult females and males. In reality, the RDI depends on age, gender, lifestyle, and other factors. Also, a more accurate representation would use minimum and maximum targets for each nutrient. For this sample app, we shall be content with a rough indicator to compare foods.

With the two domain classes above, storing the RDIs is now a matter of writing them down, as shown in Listing 8.112. As all other code, you can find this on GitHub<sup>12</sup> to copy and paste.

12. [https://github.com/petersommerhoff/nutrilicious-app/blob/master/12\\_AddingRdisForActionableData/app/src/main/java/com/petersommerhoff/nutrilicious/model/RDI.kt](https://github.com/petersommerhoff/nutrilicious-app/blob/master/12_AddingRdisForActionableData/app/src/main/java/com/petersommerhoff/nutrilicious/model/RDI.kt)

### **Listing 8.112 Storing RDIs Statically**

[Click here to view code image](#)

```
import com.example.nutrilicious.model.WeightUnit.*\n\ninternal val RDI = mapOf(\n    255 to Amount(3000.0, GRAMS),           // water\n    208 to Amount(2000.0, KCAL),           // energy\n    203 to Amount(50.0, GRAMS),            // protein\n    204 to Amount(78.0, GRAMS),            // total fat\n    205 to Amount(275.0, GRAMS),           // carbohydrates\n)
```

```
291 to Amount(28.0, GRAMS),           // fiber
269 to Amount(50.0, GRAMS),           // sugars
301 to Amount(1300.0, MILLIGRAMS),   // calcium
303 to Amount(13.0, MILLIGRAMS),     // iron
304 to Amount(350.0, MILLIGRAMS),    // magnesium
305 to Amount(700.0, MILLIGRAMS),    // phosphorus
306 to Amount(4700.0, MILLIGRAMS),   // potassium
307 to Amount(1500.0, MILLIGRAMS),   // sodium
309 to Amount(10.0, MILLIGRAMS),     // zinc
401 to Amount(85.0, MILLIGRAMS),     // vitamin c
404 to Amount(1200.0, MICROGRAMS),   // vitamin b1
405 to Amount(1200.0, MICROGRAMS),   // vitamin b2
406 to Amount(15.0, MILLIGRAMS),     // vitamin b3
415 to Amount(1300.0, MICROGRAMS),   // vitamin b6
435 to Amount(400.0, MICROGRAMS),    // folate
418 to Amount(3.0, MICROGRAMS),      // vitamin b1
320 to Amount(800.0, MICROGRAMS),    // vitamin a
323 to Amount(15.0, MILLIGRAMS),     // vitamin e
328 to Amount(15.0, MICROGRAMS),    // vitamin d
430 to Amount(105.0, MICROGRAMS),   // vitamin k
606 to Amount(20.0, GRAMS),         // saturated
605 to Amount(0.0, GRAMS),          // transfats
601 to Amount(300.0, MILLIGRAMS)    // cholesterol
)
```

The new data class `Amount` is useful to improve the `Nutrient` class as well. Listing 8.113 combines its

`amountPer100g` and `unit` properties into a single new property with type `Amount` and adjusts the DTO mapping accordingly.

**Listing 8.113 Using the Amount Class in Nutrient**

[Click here to view code image](#)

```
@TypeConverters(NutrientTypeConverter::class)

data class Nutrient(
    // ...,
    val amountPer100g: Amount, // Combines amount ar
    // ...
) {

    constructor(dto: NutrientDto) : this(
        // ...,
        Amount(dto.value.toDouble(), WeightUnit.fromString(
            // ...
        )),
        // ...
    )
}
```

Note that the type converter still works as before and that Room is still able to map this class to the database, even without a schema change.

To make `WeightUnit.fromString` work, the `WeightUnit` enum should know which string each instance corresponds to, such as “g” mapping to `WeightUnit.GRAMS`. Note that this is different from Kotlin’s default string representation of each enum instance using `valueOf`. Thus, Listing 8.114 adds the `toString` method. Additionally, the `toString` method is overridden to display the weight unit correctly in the UI later.

**Listing 8.114 Mapping WeightUnit to String And Vice Versa**

[Click here to view code image](#)

```
enum class WeightUnit {  
    GRAMS, MILLIGRAMS, MICROGRAMS, KCAL, IU; // Mind t  
  
    companion object {  
        fun fromString(unit: String) = when(unit) { // 1  
            "g" -> WeightUnit.GRAMS  
            "mg" -> WeightUnit.MILLIGRAMS  
            "\u00b5g" -> WeightUnit.MICROGRAMS  
            "kcal" -> WeightUnit.KCAL  
            "IU" -> WeightUnit.IU  
            else -> throw IllegalArgumentException("Unknown weight unit $unit")  
        }  
    }  
  
    override fun toString(): String = when(this) { // 2  
        WeightUnit.GRAMS -> "g"  
        WeightUnit.MILLIGRAMS -> "mg"  
        WeightUnit.MICROGRAMS -> "\u00b5g"  
        WeightUnit.KCAL -> "kcal"  
        WeightUnit.IU -> "IU"  
        else -> super.toString()  
    }  
}
```

Note that the **when** expression is exhaustive when mapping enum instances to a string but not the other way around. The logic itself is simply a matter of exploring which units the API returns and mapping them accordingly.

All required RDI data is now in place. You can use this information to enhance the `DetailsActivity`. The methods `bindUi` and `makeSection` can stay as they are, only `renderNutrient` must be adjusted because of the additional information you want to show for each single nutrient. [Listing 8.115](#) shows the extended method.

**Listing 8.115** Displaying a Nutrient

[Click here to view code image](#)

```
private fun renderNutrient(nutrient: Nutrient): String {
    val name = name.substringBefore(",")
    val amount = amountPer100g.value.render()
    val unit = amountPer100g.unit
    val percent = getPercentOfRdi(nutrient).render() /
    val rdiNote = if (percent.isNotEmpty()) "($percent%
    "$name: $amount$unit $rdiNote"
}
```

```
private fun Double.render() = if (this >= 0.0) "%.2f"
```



The amount and unit are now extracted from the `Amount` object. Additionally, the RDI percentage is calculated and displayed if it is greater than or equal to zero—in other words, if no error occurred. The `render` function is a simple extension on `Double` that displays it with two decimal places.

Next, you must implement `getPercentOfRdi`, which calculates how much of the RDI the given nutrient’s amount represents. [Listing 8.116](#) shows its implementation.

**Listing 8.116** Calculating the Percentage of RDI

[Click here to view code image](#)

```
private fun getPercentOfRdi(nutrient: Nutrient): Double {
    val nutrientAmount: Double = nutrient.amountPer100g.value
    val rdiAmount: Double = nutrient.amountPer100g.rdiAmount
    val rdiPercentage: Double = (nutrientAmount / rdiAmount) * 100
    return rdiPercentage
}
```

```
    val rdi: Double = RDI[nutrient.id]?.normalized() ?:  
  
    return nutrientAmount / rdi * 100  
}
```



The percentage of the RDI is a simple division of how much of the nutrient the food contains and what the RDI is. In case no RDI is found, the method returns a negative result due to the default value of `-1.0`. If this happens, no percentage is displayed by the `render` method.

The nutrient values may be given in different units, such as grams and micrograms. Thus, the `Amount` class gets a method to normalize the value, as shown in [Listing 8.117](#). This method should be called before performing any calculations based on this class.

#### **Listing 8.117 Calculating the Percentage of RDI**

[Click here to view code image](#)

---

```
data class Amount(val value: Double, val unit: Weight)  
  
fun normalized() = when(unit) { // Normalizes mil  
    GRAMS, KCAL, IU -> value  
    MILLIGRAMS -> value / 1000.0  
    MICROGRAMS -> value / 1_000_000.0  
}  
}
```



Kilocalories and international units (IU) require no normalization. As for the weights, this method normalized them to a value in grams, but you could choose any of the three weight units as the normalized form.

Users can now immediately see how much of the RDI a food provides and thus choose foods that satisfy their nutritional needs. This is all the functionality this sample app should have. If you want to extend it further, you can turn it into a nutrition-tracking app in which users can enter how much of each food they ate and see what percentage of each RDI they have reached for that day.

## IMPROVING THE USER EXPERIENCE

Although this concludes the app functionality, there are a few finishing touches you can make to improve this app by providing better feedback to the user. Subtleties like this go a long way when it comes to user experience.

### Indicating Empty Search Results

In the search fragment, users are already shown a progress indicator, which is good. But if no foods are found for a search term, users are not notified and simply presented with an empty screen. Luckily, it is easy to show feedback in a Snackbar if no foods were found. Listing 8.118 adjusts `updateListFor` to show this feedback.

**Listing 8.118 Showing a Snackbar to Indicate Empty Search**

[Click here to view code image](#)

```
private fun updateListFor(searchTerm: String) = launch {
    // ...
    withContext(UI) {
        (rvFoods?.adapter as? SearchListAdapter)?.setItem
        swipeRefresh?.isRefreshing = false
        if (foods.isEmpty() && isAdded) {
            snackbar("No foods found")
        }
    }
}
```

```
}
```

```
}
```

If a search returns an empty list and the fragment is still attached to its activity, this shows a snack bar to tell the user that no foods were found for the given search term. The `snackbar` method is an extension defined in `ViewExtensions.kt` (see [Listing 8.119](#)).

**Listing 8.119 Extension Function to Show Snackbar**

[Click here to view code image](#)

```
import android.support.design.widget.Snackbar

import android.view.View

fun Fragment.snackbar(
    msg: String, view: View = activity!!.findViewById(
        R.id.fragment_container
    )
) {
    Snackbar.make(view, msg, Snackbar.LENGTH_SHORT).show()
}
```

This method relies on the fragment being attached to its activity because it uses an unsafe call in its default value. Thus, you must always check `isAdded` before calling this method, or pass in a view manually. After importing this function into the search fragment, you can run the app.

### Indicate Progress in the Details Activity

Similarly, a progress indicator should be shown in the `DetailsActivity` to let the user know the data is being fetched. This activity has no `SwipeRefreshLayout`, so the first step is to add a `ProgressBar` to `activity_details.xml` as in [Listing 8.120](#). Also, the linear layout is given an ID in order to show and hide it later when appropriate.

**Listing 8.120 Adding a ProgressBar to the Layout**

[Click here to view code image](#)

```
<ScrollView ...>

    <ProgressBar
        android:id="@+id/progress"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:visibility="gone"
        style="?android:attr/progressBarStyle" />

    <LinearLayout android:id="@+id/content" ...>
        <!-- sections as before -->
    </LinearLayout>

</ScrollView>
```

The progress bar's default visibility is GONE, meaning that it is not only hidden but does not affect the layout at all. The logic for showing and hiding it is similar as in the search fragment.

[Listing 8.121](#) introduces a new helper method to `DetailsActivity` and adjusts `updateUiWith` to show and hide the progress bar.

[Listing 8.121 Showing and Hiding the Progress Bar](#)

[Click here to view code image](#)

```
import android.view.View

// ...

class DetailsActivity : AppCompatActivity() {

    // ...
```

```
private fun updateUiWith(foodId: String) {  
    if (foodId.isBlank()) return  
    setLoading(true) // Indicates that app is loading  
    launch {  
        val details = detailsViewModel.getDetails(foodId)  
        withContext(Dispatchers.Main) {  
            setLoading(false) // Indicates that app finished loading  
            bindUi(details)  
        }  
    }  
}  
  
private fun setLoading(isLoading: Boolean) {  
    if (isLoading) {  
        content.visibility = View.GONE  
        progress.visibility = View.VISIBLE  
    } else {  
        progress.visibility = View.GONE  
        content.visibility = View.VISIBLE  
    }  
}  
}
```

The helper method `setLoading` toggles between showing the progress bar and hiding the content, and vice versa. With this, the progress bar is shown until the data is received.

Tweaks like these go a long way to improve the app's usability. Therefore, even though not strictly Kotlin related,

they were included here to indicate aspects that improve user experience.

## SUMMARY

Having finished this chapter and [Chapter 7](#), you now have two Kotlin apps under your belt, along with best practices for Android and Kotlin development and a collection of useful extension functions. You are able to implement recycler views, use fragments, and create domain classes, DTOs, and DAOs concisely, and you have familiarized yourself with essential tools such as Retrofit, Moshi and the Android Architecture Components. You have also written idiomatic code using scope operators, delegated properties, immutability where possible, null handling, and other language features. You can build on all of this in your future apps.

## Kotlin DSLs

Jos  
Men build too many walls and not enough bridges.  
eph

Fort Newton

Domain-specific languages (DSLs) regularly come up in discussions related to modern languages like Kotlin and Scala because these allow creating simple DSLs quickly within the language. Such internal DSLs can greatly improve code readability, development efficiency, and changeability. This chapter introduces DSLs and how to create them in Kotlin. It also explores the two most popular Kotlin DSLs on Android, namely a Kotlin DSL for Gradle build configurations and one for creating Android layouts.

## INTRODUCING DSLS

DSLs are by no means new; research goes back decades<sup>1</sup> and is only increasing in recent times, from language modeling in general<sup>2</sup> to DSLs in particular.<sup>3</sup> Today, DSLs are pervasive in software development. This section explains the term and discusses how DSLs can help improve your code.

1. <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380070406>

2. <http://mdebook.irisa.fr/>

3. <http://www.se-rwth.de/publications/MontiCore-a-Framework-for-Compositional-Development-of-Domain-Specific-Languages.pdf>

## What Is a DSL?

As the name implies, a domain-specific language is a language focused on, and often restricted to, a certain application domain. You are likely familiar with many DSLs already, such as SQL, LaTeX,<sup>4</sup> or regular expressions. These are in contrast to general-purpose languages like Kotlin, Java, C++,<sup>5</sup> Python,<sup>6</sup> and others.

4. <https://www.latex-project.org/>

5. <http://www.stroustrup.com/C++.html>

6. <https://www.python.org/>

Focusing on a particular domain allows a more focused and clean syntax and hence better solutions. This is a general trade-off between generic approaches and specific approaches, and it is the reason why languages like COBOL<sup>7</sup> and Fortran<sup>8</sup> were initially created with a specific domain in mind: business processing and numerical computation, respectively.

7.

[http://archive.computerhistory.org/resources/text/Knuth\\_Don\\_X4100/PDF\\_index/k-8-pdf/k-8-u2776-Honeywell-mag-History-Cobol.pdf](http://archive.computerhistory.org/resources/text/Knuth_Don_X4100/PDF_index/k-8-pdf/k-8-u2776-Honeywell-mag-History-Cobol.pdf)

8. <http://www.ecma-international.org/publications/files/ECMA-ST-WITHDRAWN/ECMA-9,%201st%20Edition,%20April%201965.pdf>

However, these are different from DSLs written in Kotlin because they are stand-alone languages. When you create a DSL in Kotlin, it is called an *embedded DSL*—it is embedded into a general-purpose language. This has several consequences that come into play when we discuss benefits and potential drawbacks of these DSLs.

## Benefits and Drawbacks

As with any approach or tool, there are both pros and cons to consider when developing or using DSLs. Van Deursen, Klint, and Visser<sup>9</sup> provide a good overview of both in their literature survey. Here, we view them under the light of embedded DSLs written in Kotlin.

9. <http://sadiyahameed.pbworks.com/f/deursen+-+DSL+annotated.pdf>

## Benefits

The main benefits mentioned by van Deursen et al. are the following:

- **Solutions can be expressed on the abstraction level of the domain,** enabling domain experts to understand, validate, optimize, and often even develop their own solutions in the DSL. This also applies to Kotlin DSLs but (in any case) requires an adequately designed DSL.
- **DSL code is concise, expressive, and reusable.** This certainly applies to Kotlin DSLs just like to other idiomatic Kotlin code.
- **DSLs represent domain knowledge** and thus facilitate documentation and reuse of this knowledge. Again, this applies to any adequately designed DSL.

In addition to these, there are several advantages specific to a Kotlin DSL:

- **It does not add a new technology to your stack** because it is simply Kotlin code.
- **As an embedded DSL, it allows you to use any of Kotlin's language features** such as storing recurring values in variables or using loops inside your DSL.
- **Because it is pure Kotlin code, your DSL is statically typed**, and the excellent tool support automatically works for your DSL, including autocomplete, jumping to declarations, showing docs, and code highlighting.

In summary, DSLs are particularly useful where they can embody domain knowledge and reduce complexity by offering a clean, readable API for developers and domain experts.

## Potential Drawbacks

In their survey, van Deursen et al. also mention several drawbacks of DSLs that we will again examine in the light of Kotlin DSLs. Potential drawbacks of DSLs in general are as follows:

- **DSLs can be hard to design, implement, and maintain.** This partially applies to Kotlin DSLs as well, especially the design process for a good DSL must involve domain experts.<sup>10</sup> However, as you will see in the examples in this chapter, developers themselves are the domain experts for the DSLs discussed in this chapter (such as the Kotlin Gradle DSL).
10. <http://www.se-rwth.de/publications/Design-Guidelines-for-Domain-Specific-Languages.pdf>
- **Educating DSL users can be costly.** In the case of Kotlin DSLs, users are either Kotlin developers who already know the language or domain experts who may need educating but for whom the DSL greatly facilitates understanding compared to non-DSL code.
  - **Finding the proper scope for a DSL can be difficult.** This is independent of whether you use a Kotlin DSL or not. Depending on your domain, it may be hard to choose the proper scope. One way that Kotlin may help is that implementing a prototypical DSL is fast and thus allows

evaluating a DSL with a certain scope and making adjustments to its scope as needed.

- **It can be difficult to balance between domain-specific and general-purpose constructs.** This issue is mostly alleviated in Kotlin DSLs (and any embedded DSL) because you can focus on the domain-specific aspects and still tap into all of Kotlin's general-purpose constructs. In effect, finding the balance is shifted from DSL design to DSL usage because users may use general-purpose language features to any extent they choose (which can hamper understandability if overused).
- **DSLs may be less efficient than hand-coded software.** Fortunately, this does not apply to Kotlin DSLs because they typically compile to the more verbose code you would write without the DSL. Because Kotlin DSLs make heavy use of higher-order functions, using inline functions is essential to avoid any overhead.

Another drawback of embedded DSLs in particular is their limited expressiveness. Being embedded into a general-purpose language, they are naturally constrained in their syntax and in what they can express. For Kotlin DSLs in particular, this chapter focuses on DSLs that allow constructing objects more easily.

All in all, most of the common drawbacks of DSLs do not apply to Kotlin DSLs or are at least alleviated. Still, DSLs are not a golden hammer. Not every object creation or configuration must be wrapped into a DSL. But if it can be used to encapsulate boilerplate, reduce complexity, or help domain experts contribute, a DSL can greatly benefit your project. Embedded DSLs, such as in Kotlin, remove most of the effort involved in creating DSLs and thus greatly reduce the barrier to introducing DSLs to your workflow (but they are also limited in their capabilities).

## CREATING A DSL IN KOTLIN

In this section, you will write your own DSL in Kotlin from scratch to understand the underlying structure. Kotlin DSLs are primarily based on higher-order functions, lambdas with receivers, default values, and extension functions. The resulting DSL should allow you to create user objects with the syntax shown in [Listing 9.1](#).

[Listing 9.1 User DSL](#)

[Click here to view code image](#)

```
user {  
    username = "johndoe"  
    birthday = 1 January 1984  
    address {  
        street = "Main Street"  
        number = 42  
        postCode = "12345"  
        city = "New York"  
    }  
}
```

As you can see, this syntax to create objects almost resembles JSON. It hides Kotlin as the underlying language (because it avoids using general-purpose language constructs) and hence does not require users to know Kotlin. Nontechnical team members would be able to understand, validate, and change the user.

### DSL to Build Complex Objects

To explore how such a DSL works, you will build one from the top down. The entry point to the DSL is the `user` function. You know this trick by now. It is simply a higher-order function that accepts a lambda as its last parameter, thus allowing the syntax from [Listing 9.1](#). The `user` function can be declared as shown in [Listing 9.2](#).

#### [Listing 9.2 DSL Entry Point](#)

[Click here to view code image](#)

```
fun user(init: User.() -> Unit): User {  
    val user = User()  
    user.init()  
    return user
```

```
}
```

The function accepts a lambda with receiver so that the lambda effectively becomes an extension of the `User` class. This allows accessing properties such as `username` and `birthday` directly from within the lambda. You may notice that you can write this function a lot better using Kotlin's `apply` function, as shown in [Listing 9.3](#).

#### **Listing 9.3 DSL Entry Point—Improved**

[Click here to view code image](#)

```
fun user(init: User.() -> Unit) = User().apply(init)
```

The `User` class is a simple data class, and for this first version of the DSL, it partly uses nullable fields to keep things simple. [Listing 9.4](#) shows its declaration.

#### **Listing 9.4 User Data Class**

[Click here to view code image](#)

```
import java.time.LocalDate

data class User(
    var username: String = "",
    var birthday: LocalDate? = null,
    var address: Address? = null
)
```

The code so far accounts for the creation of users with `username` and `birthday` in DSL syntax, but there's currently no way to nest an `address` object as in the initial example. Adding the `address` is simply a matter of repeating the process, this time with a member function of `User` as implemented in [Listing 9.5](#).

#### **Listing 9.5 Adding an Address to a User**

[Click here to view code image](#)

```
data class User(...) {  
  
    fun address(init: Address.() -> Unit) {  
  
        address = Address().apply(init)  
  
    }  
  
}  
  
  
data class Address(  
  
    var street: String = "",  
  
    var number: Int = -1,  
  
    var postCode: String = "",  
  
    var city: String = ""  
  
)
```

The `address` function follows the same concept as the `user` function, but additionally assigns the created address to the user's `address` property. With these few lines of code, you are now able to create a user in DSL syntax as in [Listing 9.1](#) (except for the date).

However, there are several downsides to the current implementation.

- It relies on angelic developers who carefully initialize every field. But it is easy to forget to initialize one or assign an invalid value.
- Working with the resulting `User` object is inconvenient due to its nullable properties.
- The DSL allows undesired nesting of its constructs.

To illustrate this last issue, [Listing 9.6](#) shows an undesirable but currently valid use of the DSL.

#### **Listing 9.6 Shortcomings of Current DSL**

[Click here to view code image](#)

```
user {  
    address {  
        username = "this-should-not-work"  
    }  
    user {  
        address {  
            birthday = LocalDate.of(1984, Month.JANUARY,  
        }  
    }  
}  
}
```



This construct mitigates the DSL advantages of readability and reducing complexity. Properties of the outer scope are visible so that `username` can be set inside the `address` block, and both can be nested arbitrarily. This issue is addressed later in this chapter.

### Immutability through Builders

The first aspect to be improved are the nullable and mutable properties. The desired DSL will effectively be a *type-safe builder*.<sup>11</sup> Accordingly, builders<sup>12</sup> are used in the underlying implementation to accumulate object data, validate the data, and ultimately build the user object. First, write down the desired data classes, as shown in

#### [Listing 9.7.](#)

11. <https://kotlinlang.org/docs/reference/type-safe-builders.html>

12. [https://sourcemaking.com/design\\_patterns/builder](https://sourcemaking.com/design_patterns/builder)

#### **Listing 9.7 Immutable Data Classes**

[Click here to view code image](#)

```
import java.time.LocalDate  
  
  
data class User(val username: String, val birthday: L
```

```
data class Address(
```

```
    val street: String,
```

```
    val number: Int,
```

```
    val postCode: String,
```

```
    val city: String
```

```
)
```

This way, because both `LocalDate` and `Address` are immutable, the `User` class is immutable, too. Objects of these two classes are now constructed by builders so the next step is to add those, starting with the `UserBuilder` shown in

#### [Listing 9.8.](#)

##### **Listing 9.8 User Builder**

[Click here to view code image](#)

---

```
class UserBuilder {
```

```
    var username = ""                      // Gets assigned
```

```
    var birthday: LocalDate? = null        // Gets assigned
```

```
    private var address: Address? = null    // Is built later
```

```
    fun address(init: AddressBuilder.() -> Unit) { //
```

```
        address = AddressBuilder().apply(init).build()
```

```
  
  


```
    fun build(): User { // Validates data and builds user
```


```
        val theBirthday = birthday
```


```
        val theAddress = address
```


```
        if (username.isBlank() || theBirthday == null ||
```


```

```
        throw IllegalStateException("Please set username")  
  
    }  
  
    return User(username, theBirthday, theAddress)  
}
```

The builder differentiates between properties that should be assigned directly (and are thus public) and properties that are created using a nested DSL syntax (the address) that are thus private. The `address` function is moved from the `User` class to the `UserBuilder`, and now accepts a lambda that has `AddressBuilder` as its receiver. Also, you can perform arbitrary validation before finally building the object in `build`. You could use this to implement optional properties as well. Next, you need a builder for address objects, as shown in [Listing 9.9](#).

**Listing 9.9 Address Builder**

[Click here to view code image](#)

```
class AddressBuilder {  
  
    var street = ""  
  
    var number = -1  
  
    var postCode = ""  
  
    var city = ""  
  
  
    fun build(): Address {  
  
        if (notReady())  
  
            throw IllegalStateException("Please set street,  
  
                return Address(street, number, postCode, city)  
    }
```

```
private fun notReady()  
  
    = arrayOf(street, postCode, city).any { it.isB]  
  
}
```

In this case, there is no more nested construct under address, so the builder only has public properties for all required data and a `build` method along with simple validation, but no more nested builders.

Lastly, the entry point to the DSL must be adapted as in [Listing 9.10](#) to use the builder.

#### **Listing 9.10 Adjusted Entry Point to the DSL**

[Click here to view code image](#)

```
fun user(init: UserBuilder.() -> Unit) = UserBuilder(
```

The lambda's receiver is now `UserBuilder`. Accordingly, the `init` function is applied to the `UserBuilder`, and calling `build` is required. So inside `init`, you can initialize the public properties directly or call a DSL function to build a more complex object, such as an address.

### **Nesting Deeper**

The current DSL allows adding an arbitrary number of `address` blocks, but each one would override the address from the previous. So a user can currently only have a single address, but multiple may be desired. There are different ways to design this part of the DSL.

- Check in your DSL if the `address` function was already called and disallow another call so that users can only have a single address and the DSL allows only one `address` block.
- Allow multiple calls to the `address` function and add each new address to a list.
- Implement a dedicated `addresses` block that contains all addresses.

For now, let's assume a user can indeed have multiple addresses but there is no dedicated block to hold them (the

second possibility). One way to achieve this easily is to give the data class a list of addresses, and then in `address`, add the built object to that list as done in [Listing 9.11](#).

**Listing 9.11 Allowing Multiple Addresses**

[Click here to view code image](#)

```
data class User(..., val addresses: List<Address>)

class UserBuilder {

    // ...

    private val addresses: MutableList<Address> = mutableListOf()

    fun address(init: AddressBuilder.() -> Unit) {
        addresses.add(AddressBuilder().apply(init).build())
    }

    fun build(): User { ... }

}
```

This now lets you add multiple `address` blocks, and each one adds another address to the user object. Next, a dedicated `addresses` block should encompass all addresses, yielding the syntax shown in [Listing 9.12](#).

**Listing 9.12 Syntax of Dedicated Addresses Block**

[Click here to view code image](#)

```
user {

    username = "johndoe"

    birthday = LocalDate.of(1984, Month.JANUARY, 1)

    addresses { // New dedicated addresses block

        address { // All address blocks must be placed here
    }
}
```

```
    street = "Main Street"

    number = 42

    postCode = "12345"

    city = "New York"

}

address {

    street = "Plain Street"

    number = 1

    postCode = "54321"

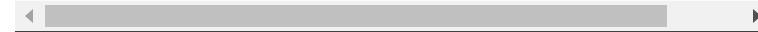
    city = "York"

}

}

}


```



Here, addresses can only be created inside the `addresses` block. To implement such an additional nesting, you need a helper class, an object of which is built by the `addresses` block. Listing 9.13 adds such a class that simply represents a list of addresses.

#### Listing 9.13 Implementing Dedicated Addresses Block

[Click here to view code image](#)

---

```
class UserBuilder {

    // ...

    private val addresses: MutableList<Address> = mutableListOf()

    inner class Addresses : ArrayList<Address>() {

        fun address(init: AddressBuilder.() -> Unit) {
            add(AddressBuilder().apply(init).build())
        }
    }
}
```

```
}

fun addresses(init: Addresses.() -> Unit) { // 'Ac
    addresses.addAll(Addresses().apply(init))
}

fun build(): User {
    val theBirthday = birthday
    if (username.isBlank() || theBirthday == null ||
        theBirthday.time == 0L) {
        return User(username, theBirthday, addresses)
    }
}
```

This is all you need to enable the syntax shown in [Listing 9.12](#). In general, to allow arbitrarily deep nesting, you must introduce the appropriate helper classes and methods, like `Addresses` and the `addresses` function in this case. Once you are familiar with creating DSLs like this, you could even generate (most of) the underlying code because the structure always follows the same pattern. In fact, JetBrains does this with a React DSL used internally.<sup>13</sup>

<sup>13</sup>. Source: “Create Your Own DSL in Kotlin” by Victor Kropp (<https://youtu.be/tZIRovCbYM8>)

## Introducing `@DslMarker`

This small DSL is now mostly finished, but the problems of arbitrary nesting and accessing properties of an outer scope remain (see [Listing 9.6](#)). To help alleviate the problem of accessing the outer scope, Kotlin 1.1 introduced the `@DslMarker` annotation. It's a meta-annotation that can be used only on other annotations, such as `@UserDsl` shown in [Listing 9.14](#).

**Listing 9.14 User DSL Annotation**

```
@DslMarker  
  
annotation class UserDsl
```

Now, whenever there are two implicit receivers of a lambda (for instance, when nesting deeper than one level in your DSL), and both are annotated with `@UserDsl`, only the innermost receiver is accessible. So if you annotate all your DSL classes with it, you can no longer access `User` properties inside the `address` block (or `addresses` block). It also prevents nesting an `address` block into another `address` block because the `address` function belongs to the outer receiver (`UserBuilder`) and not the innermost one.

What is not prevented this way is calling the `user` function again from somewhere within the DSL because it is just a top-level function and thus accessible everywhere. However, this is a mistake developers are unlikely to make accidentally. If you want to prevent it, add a deprecated member method to `UserBuilder` that overshadows the top-level function, as shown in [Listing 9.15](#).

**Listing 9.15 Preventing Nested Calls to Entry Point (in `UserBuilder`)**

[Click here to view code image](#)

```
@Deprecated("Out of scope", ReplaceWith(""), Deprecat  
  
fun user(init: UserBuilder.() -> Unit): Nothing = err
```

With this, your IDE immediately complains if you try to call `user` from within another call to `user`. It will not compile, either.

**Note**

You can follow this same procedure to implement a type-safe builder DSL even if you don't own the classes (or if they are implemented in Java) by using extension functions.

Only in the case of annotations, it is a little more tricky because you cannot annotate a thirdparty class. Instead, you can annotate the lambda receiver:

[Click here to view code image](#)

```
fun user(init: (@UserDsl UserBuilder).() -> Unit)  
    = UserBuilder().apply(init).build()
```

To enable annotating types as done above, add the corresponding annotation target (also, retaining the annotation info after compilation is not necessary):

[Click here to view code image](#)

```
@DslMarker  
@Target(AnnotationTarget.CLASS, AnnotationTarget.TYPE) // Can be used on classes  
@Retention(AnnotationRetention.SOURCE)
```

annotation class UserDsl

## Leveraging Language Features

As embedded DSLs, your Kotlin DSLs can make use of Kotlin's stack of language features. For example, you could use variables within your DSL naturally and without any extra effort (see [Listing 9.16](#)).

### **Listing 9.16 Using Variables in Your DSL**

[Click here to view code image](#)

---

```
user {  
    // ...  
  
    val usercity = "New York"  
  
    addresses {  
        address {  
            // ...  
  
            city = usercity
```

```
    }

    address {

        // ...

        city = usercity

    }

}
```

You could also use conditionals, loops, and other constructs. This automatically allows DSL users to avoid code duplication within the DSL but requires programming knowledge. Other powerful features—such as extensions, infix functions, and operators—can be used to make the code read even more naturally. As an example, Listing 9.17 enables a more natural way to write dates with an infix extension.

**Listing 9.17 Improving DSL Readability**

[Click here to view code image](#)

---

```
infix fun Int.January(year: Int) = LocalDate.of(year,
```

```
user {

    username = "johndoe"

    birthday = 1 January 1984

    // ...

}
```

In reality, you will need 12 extensions to cover all the months, but it can be worth it if your DSL users have to denote many dates because it prevents potential mistakes from zero-indexing versus one-indexing the days and months.

With the addition of this extension function, you can now write code as shown in the initial [Listing 9.1](#), or with a dedicated `addresses` block if you prefer that convention.

But there is still room for improvement because the DSL currently comes with overhead for lambda objects created. You can resolve this issue by inlining the higher-order functions (see [Listing 9.18](#)).

#### **Listing 9.18 Improving DSL Performance**

[Click here to view code image](#)

```
@UserDsl

class UserBuilder {

    // ...

    val addresses: MutableList<Address> = mutableListOf

    inner class Addresses : ArrayList<Address>() {

        inline fun address(init: AddressBuilder.() -> Unit) {
            add(AddressBuilder().apply(init).build())
        }

        inline fun addresses(init: Addresses.() -> Unit) {
            init()
        }
    }

    inline fun user(init: UserBuilder.() -> Unit) =
        UserBuilder().apply(init).build()
}
```



Note that, to inline the `addresses` function, the `addresses` property it uses must become public as well. Otherwise, it may not be accessible in the place where it's inlined. The other two higher-order functions can be inlined without further ripple effects. Now, using the DSL has no

performance overhead compared with using the builders directly.

## DSL FOR ANDROID LAYOUTS WITH ANKO

A fantastic use case for such a DSL are layouts. Here, the underlying object created by the type-safe builder DSL is a root view that encompasses a layout, such as a linear layout or a constraint layout. Creating a layout programmatically like this has several advantages over XML layouts.

- DSL layouts provide type safety and null safety.
- Building the layout is more efficient compared to XML because it costs less CPU time and battery life.
- DSL layouts are more reusable; with XML, you would usually at least have to adjust the element IDs.

In this section, you learn how to programmatically create layouts the hard way first and then familiarize yourself with *Anko Layouts*, part of JetBrains' utility library for Android called *Anko*.<sup>14</sup> To do so, this section covers how to rewrite the layout for the **AddTodoActivity** from the Kudoo app using Anko.

14. <https://github.com/Kotlin/anko>

### Creating Layouts Programmatically

Before introducing Anko, it is worth mentioning that you could create layouts programmatically without any library.

[Listing 9.19](#) shows the code required to implement the **AddTodoActivity** layout programmatically without Anko.

#### Listing 9.19 Creating a Layout Programmatically the Hard Way

[Click here to view code image](#)

---

```
class AddTodoActivity : AppCompatActivity() {  
  
    // ...  
  
    override fun onCreate(savedInstanceState: Bundle?)  
  
        super.onCreate(savedInstanceState)
```

```
        setContentView(createView()) // No inflating of
        viewModel = getViewModel(TodoViewModel::class)
    }

private fun createView(): View {
    val linearLayout = LinearLayout(this).apply { //
        orientation = LinearLayout.VERTICAL
    }

    val etNewTodo = EditText(this).apply { // Sets up
        hint = getString(R.string.enter_new_todo)
        textAppearance = android.R.style.TextAppearance
        layoutParams = ViewGroup.LayoutParams(
            ViewGroup.LayoutParams.MATCH_PARENT,
            ViewGroup.LayoutParams.WRAP_CONTENT
        )
    }

    val btnAddTodo = Button(this).apply { // Sets up
        text = getString(R.string.add_to_do)
        textAppearance = android.R.style.TextAppearance
        layoutParams = LinearLayout.LayoutParams(
            ViewGroup.LayoutParams.WRAP_CONTENT,
            ViewGroup.LayoutParams.WRAP_CONTENT
        ).apply { gravity = Gravity.CENTER_HORIZONTAL }
        setOnClickListener {
            val newTodo = etNewTodo.text.toString()
            launch(DB) { viewModel.add(TodoItem(newTodo)) }
            finish()
        }
    }
}
```

```
    }

}

return linearLayout.apply { // Adds views to the
    addView(etNewTodo)

    addView(btnAddTodo)

}

}
```

You can see that, even though Kotlin's `apply` function helps simplify the code quite a bit, creating a layout like this is quite verbose. There's no support around setting layout parameters, defining listeners, or using string resources to set texts.

Luckily, you can do better using Anko.

## Anko Dependencies

The first way to include Anko in your Gradle project is to use a metadependency that incorporates all of Anko's features. Apart from Anko Layouts, this includes Anko Commons, Anko SQLite, and more. [Listing 9.20](#) shows the corresponding Gradle dependency.

### Listing 9.20 Anko Metadependency

[Click here to view code image](#)

```
def anko_version = "0.10.5"  
implementation "org.jetbrains.anko:anko:$anko_version"
```

You likely do not need all of Anko's features, so there are smaller dependencies available. For the Anko Layout discussed in this section, all you need is given in Listing 9.21.

### Listing 9.21 Dependencies for Anko Layouts

[Click here to view code image](#)

```
implementation "org.jetbrains.anko:anko-sdk25:$anko_\nimplementation "org.jetbrains.anko:anko-sdk25-corouti
```

Anko has many fine-grained dependencies for Android support libraries and coroutines, all of which are listed on GitHub.<sup>15</sup>

[15. https://github.com/Kotlin/anko#gradle-based-project](https://github.com/Kotlin/anko#gradle-based-project)

### Creating Layouts with Anko

Anko uses the same idea you used in the previous section to make layout creation a lot easier. For instance, creating a vertical linear layout that contains a button with a listener is achieved with a few lines of code, as shown in Listing 9.22.

**Listing 9.22 Simple Anko Layout**

[Click here to view code image](#)

```
verticalLayout {\n    button {\n        text = "Receive reward"\n        onClick { toast("So rewarding!") }\n    }\n}
```

The `verticalLayout` function is a utility to create a `LinearLayout` with vertical orientation. By nesting another view inside the lambda, it is automatically added to the containing view. Here, the button becomes part of the linear layout. Setting the text and adding a click listener is as simple as assigning the property and using `onClick`, respectively. The `toast` function is also one of Anko's many utilities, and it works just like the one you wrote yourself.

## Adding Layout Parameters

Properties like width, height, margin, and padding are set via layout parameters, for short `lparams` in Anko. These are chained to the declaration of a view to define how it should be laid out inside its parent. As an example, Listing 9.23 makes the button span its parent's width and adds margins on all sides to it.

**Listing 9.23 Adding Layout Parameters with Anko**

[Click here to view code image](#)

```
verticalLayout {  
    button { ... }.lparams(width = matchParent) {  
        margin = dip(5)  
    }  
}
```

Here, the `lparams` function is chained to the `button` call. There are many overloads of `lparams`, one of which allows setting layout width and height directly as arguments (inside parentheses). Both are set to `wrapContent` by default so, in contrast to XML, you can skip those if the default works for you; Anko provides a `matchParent` property for the other cases.

Inside the `lparams` lambda, you can set margin and padding. As in XML, there are convenient properties for `margin` (all sides), `verticalMargin`, and `horizontalMargin` (the same goes for `padding`). The `dip` function is used for dp values (density-independent pixels), and there is also an `sp` function for text sizes (scale-independent pixels).

**Tip**

With the Anko Support plugin for Android Studio, you can preview your Anko layouts in Android Studio's Design View—if you modularize them into an Anko component:

[Click here to view code image](#)

```
class ExampleComponent : AnkoComponent<MainActivity> {

    override fun createView(ui: AnkoContext<MainActivity>): View = with(ui) {

        verticalLayout {

            button { ... }.lparams(width = matchParent) { ... }

        }
    }
}
```

Unfortunately, the preview requires a rebuild, so the feedback cycle is significantly slower than with XML layouts. Also, it currently seems to work with activities only, not fragments. To get the benefits of both XML and Anko, you can create your layout in XML initially and then migrate it to Anko. The process is straightforward, and Anko even comes with an XML-to-Anko converter that you can find in the menu under *Code* and then *Convert to Anko Layouts DSL* while inside an XML layout.

## Migrating Kudoo's `AddTodoActivity` to Anko Layouts

Let's put these concepts into practice by rewriting part of the layout for the Kudoo app. The simple examples shown so far already cover most of what you need to create a proper layout with Anko. Following the same structure, you can build up the basic layout as it is used in the `AddTodoActivity`, as shown in [Listing 9.24](#).

**Listing 9.24** Activity Layout with Anko

[Click here to view code image](#)

```
class AddTodoActivity : AppCompatActivity() {

    // ...

    override fun onCreate(savedInstanceState: Bundle?) {

        super.onCreate(savedInstanceState)

        setContentView(createView()) // Still no inflation

        viewModel = getViewModel(TodoViewModel::class)
    }
}
```

```
    }

    private fun createView() = verticalLayout { // Set

        val etNewTodo = editText { // Sets up EditText as

            hintResource = R.string.enter_new_todo

            textAppearance = android.R.style.TextAppearance

        }.lparams(width = matchParent, height = wrapContent

            margin = dip(16)

        }

        button(R.string.add_to_do) { // Sets up Button as

            textAppearance = android.R.style.TextAppearance

        }.lparams(width = wrapContent, height = wrapContent

            gravity = Gravity.CENTER_HORIZONTAL

        ).setOnClickListener { // Could also use click

            val newTodo = etNewTodo.text.toString()

            launch(DB) { viewModel.add(TodoItem(newTodo)) }

            finish()

        }

    }
}
```

Here, the layout makes use of variables and assignments naturally within the DSL. Again, this is a benefit of an embedded DSL. Second, it uses one of the helper properties provided by Anko to assign resources directly, namely `hintResource`. This way, you can avoid calls to methods like `getString` to read the value of an Android resource.

Note that the views no longer require IDs for this layout; views that are required outside of the DSL code can be made accessible by assigning them to variables from an outer scope.

## Modularizing the Anko Layout

The code from Listing 9.24 creates a view (a `LinearLayout`) that you can directly use inside the activity's `onCreate` method instead of a layout inflator. However, a better approach is to use `AnkoComponent` to modularize the code, as shown in Listing 9.25.

**Listing 9.25 Modularized Activity Layout Using an Anko Component**

[Click here to view code image](#)

```
class AddTodoActivity : AppCompatActivity() {  
    // ...  
  
    override fun onCreate(savedInstanceState: Bundle?)  
        super.onCreate(savedInstanceState)  
        setContentView(AddTodoActivityUi().createView(AnkoContext.of(this)))  
        viewModel = getViewModel(TodoViewModel::class)  
    }  
  
private inner class AddTodoActivityUi : AnkoComponent{  
    override fun createView(ui: AnkoContext<AddTodoActivity>): View {  
        verticalLayout {  
            val etNewTodo = editText {  
                hintResource = R.string.enter_new_todo  
                textAppearance = android.R.style.TextAppearance  
            }.lparams(width = matchParent, height = wrapContent)  
            margin = dip(16)  
        }  
    }  
}
```

```
    }

    button(R.string.add_to_do) {

        textAppearance = android.R.style.TextAppearance

    }.LayoutParams(width = wrapContent, height = wrapContent)

        gravity = Gravity.CENTER_HORIZONTAL

    }.setOnClickListener {

        val newTodo = etNewTodo.text.toString()

        launch(DB) { viewModel.add(TodoItem(newTodo))

            finish()

        }

    }

}

}

}
```

## Adding Custom Views

Anko comes with builder functions for all of Android's views, but what if you have a custom view? How can you incorporate that into the layout DSL? Fortunately, Anko is extensible in this regard so that you can extend it with your custom view via extension functions, and the syntax looks familiar. Assume you have a custom frame layout like the one in [Listing 9.26](#) that ensures that it always has the same width as height.

### Listing 9.26 Custom Frame Layout

[Click here to view code image](#)

```
import android.content.Context  
  
import android.util.AttributeSet
```

```
import android.widget.FrameLayout

class SquareFrameLayout(
    context: Context,
    attributes: AttributeSet? = null,
    defStyleAttr: Int = 0
) : FrameLayout(context, attributes, defStyleAttr) {

    override fun onMeasure(widthMeasureSpec: Int, heightMeasureSpec: Int) {
        super.onMeasure(widthMeasureSpec, heightMeasureSpec)
    }
}
```

You can incorporate this into the Anko Layout DSL by adding an extension function on Android's `ViewManager` that handles its creation, as shown in [Listing 9.27](#).

#### **Listing 9.27 Integrating a Custom Layout into Anko**

[Click here to view code image](#)

```
import android.view.ViewManager
import org.jetbrains.anko.custom.ankoView

inline fun ViewManager.squareFrameLayout(init: SquareFrameLayout.() -> Unit) = ankoView({ SquareFrameLayout(it) }, theme = 0, init)
```

As you can see, the function signature closely resembles the ones you used to build the User DSL before; the lambda parameter becomes an extension of your `SquareFrameLayout`. The `ankoView` function is used to create the view, it can optionally apply a theme, and it handles further initialization based on the lambda expression that is

passed in. Its implementation is not much different from the builder methods in the User DSL. Its first parameter represents a factory so that you can tell it how to construct the initial object before applying `init`, here just using `SquareFrameLayout(it)`. You could also add a theme parameter to your extension function and pass it along to `ankoView` to allow users to set a theme. With this, you can use `squareFrameLayout` from within the Anko DSL to build an object of this custom view.

### Anko Layouts versus XML Layouts

Anko Layouts have several benefits over XML layouts as listed at the beginning of this section—most notably, type safety and improved performance while saving battery. However, there are also drawbacks when compared with XML. In parts, they have already become apparent throughout this section but I will list them again here.

- XML layouts provide faster preview in Android Studio's design view, hence speeding up the feedback cycle, which is crucial while working on a layout.
- Autocompletion works faster in XML because the search space is a lot smaller.
- Layouts are automatically separated from business logic. With Anko, you are responsible for keeping these concerns separated.

In the end, which one is the better choice for your project depends on which of these points you prioritize. In any case, I recommend starting out with an XML layout until you are satisfied with it. After that, you can evaluate the possibility of migrating it to Anko.

# DSL FOR GRADLE BUILD SCRIPTS

In 2016, Gradle announced a DSL based on Kotlin as an alternative to Groovy to write build scripts, and so the *Gradle Kotlin DSL*<sup>16</sup> was born. The main reason for this decision was Kotlin's static typing that enables better tool support in Gradle, from code completion and navigation to the ability to use all of Kotlin's language features,<sup>17</sup> thus making it easier to write build scripts from scratch.

16. <https://github.com/gradle/kotlin-dsl>

17. <https://blog.gradle.org/kotlin-meets-gradle>

The Gradle Kotlin DSL is a fair alternative to the usual Groovy build scripts and has its advantages and drawbacks. It is certainly not a tool you have to use today (at the time of writing); it still has its weaknesses and not the best documentation. But it is worth exploring, especially in the context of Kotlin DSLs. So in this section, you'll rewrite your Nutrilicious build scripts using the Gradle Kotlin DSL.

## Migrating Nutrilicious to Gradle Kotlin DSL

Based on the existing build scripts, you will migrate to the Gradle Kotlin DSL step by step in this section. This will uncover many similarities and some differences between the Kotlin DSL and the Groovy DSL.

### Note

At the time of writing, Android Studio may not immediately recognize the Gradle Kotlin DSL. In that case, try *Refresh All Gradle Projects* in the Gradle view, and if that does not help try restarting Android Studio.

## Migrating Gradle Settings

Start migrating the simplest Gradle file, the `settings.gradle`. For the Nutrilicious app, its definition in Groovy is just the one line shown in Listing 9.28.

**Listing 9.28** `settings.gradle` (Groovy)

```
include ":app"
```

---

As opposed to Groovy, Kotlin does not allow skipping the parentheses of such method calls, thus its equivalent in the Kotlin DSL uses parentheses, as shown in [Listing 9.29](#).

**Listing 9.29** `settings.gradle.kts` (Kotlin)

```
include(":app")
```

You have to rename the file to `settings.gradle.kts` to indicate that you are using a Kotlin script. Nothing else is required, so the project should still build successfully.

## Migrating the Root Build Script

Although not as complex as the app module's build script, the root build script introduces several new concepts of the Gradle Kotlin DSL.

### Build Script Block

The `buildscript` section defines an extra for the Kotlin version along with repositories and dependencies. Its Groovy code is shown again for reference in [Listing 9.30](#).

**Listing 9.30** `buildscript` Block (Groovy)

[Click here to view code image](#)

```
buildscript {  
  
    ext.kotlin_version = '1.2.50' // Extra that stores  
    repositories {  
  
        google()  
  
        jcenter()  
  
    }  
  
    dependencies {  
  
        classpath 'com.android.tools.build:gradle:3.1.3'  
  
        classpath "org.jetbrains.kotlin:kotlin-gradle-plu  
  
    }  
}
```

```
}
```

This block looks similar using the Gradle Kotlin DSL shown in [Listing 9.31](#).

**Listing 9.31 buildscript Block (Kotlin)**

[Click here to view code image](#)

```
buildscript {  
  
    extra["kotlin_version"] = "1.2.50"  
  
    repositories {  
  
        jcenter()  
  
        google()  
  
    }  
  
    dependencies {  
  
        classpath("com.android.tools.build:gradle:3.1.3")  
  
        classpath("org.jetbrains.kotlin:kotlin-gradle-plu  
  
    }  
  
}
```

The only notable difference is that extras are defined using `extra["key"] = value`, and accordingly, they must also be accessed via the extra. Also, you cannot omit the parentheses when calling the `classpath` function.

**All Projects Block**

The `allprojects` block is in fact the exact same in both DSLs (see [Listing 9.32](#)).

**Listing 9.32 allprojects Block (Groovy and Kotlin)**

```
allprojects {  
  
    repositories {  
  
        jcenter()  
    }  
}
```

```
        google()

    }

}
```

#### Delete Task

The syntax to create tasks is slightly different. In Groovy, the `clean` task is defined as shown in [Listing 9.33](#).

#### [Listing 9.33 Delete Task \(Groovy\)](#)

[Click here to view code image](#)

```
task clean(type: Delete) {

    delete rootProject.buildDir

}
```

Kotlin uses a higher-order function to create tasks. It accepts the task type as a generic parameter, the task name as a string, and a lambda defining the task (see [Listing 9.34](#)).

#### [Listing 9.34 Delete Task \(Kotlin\)](#)

[Click here to view code image](#)

```
task<Delete>("clean") {

    delete(rootProject.buildDir)

}
```

Other than that, the only difference is again the syntax for method calls. This is all that's required to migrate the root `build.gradle` file to the Gradle Kotlin DSL. To make it work, rename the file to `build.gradle.kts`. Android Studio should recognize it as a Gradle build script.

### Migrating the Module Build Script

The module's build script is longer than the other scripts, but the migration is mostly straightforward.

## Plugins

The syntax to apply plugins is quite different. Instead of writing `apply plugin: 'my-plugin'` on the top level for each one, Kotlin introduces a `plugins` block, as shown in Listing 9.35.

**Listing 9.35 Applying Plugins (Kotlin)**

[Click here to view code image](#)

```
plugins {  
    id("com.android.application")  
    id("kotlin-android")  
    id("kotlin-android-extensions")  
    id("kotlin-kapt")  
}
```

Using `id`, you can use the same string as in Groovy to identify the plugins. Alternately, you could use `kotlin`, which prepends `"org.jetbrains.kotlin."` to the given plugin. For instance, you could use `kotlin("android")` instead of `id("kotlin-android")`. A full list of plugins under `org.jetbrains.kotlin` is available via the Gradle plugin search.<sup>18</sup> Personally, I prefer the consistent look of using only `id`.

<sup>18</sup> <https://plugins.gradle.org/search?term=org.jetbrains.kotlin>

## Android

Next comes the `android` block. As shown in Listing 9.36, its definition in Kotlin is similar to the Groovy way.

**Listing 9.36 Android Setup (Kotlin)**

[Click here to view code image](#)

```
android {  
    compileSdkVersion(27)  
    defaultConfig {
```

```
applicationId = "com.example.nutrilicious"

minSdkVersion(19)

targetSdkVersion(27)

versionCode = 1

versionName = "1.0"

testInstrumentationRunner = "android.support.test

}

buildTypes {

    getByName("release") {

        isMinifyEnabled = false

        proguardFiles("proguard-rules.pro")

    }

}

}
```

It is not always obvious which properties can be set using property access and which require a method call. Fortunately, autocompletion helps with this. In the case of SDK versions, the Kotlin DSL requires method calls because the property type is `ApiVersion` under the hood and the methods are helpers to create these objects from the given integer.

Note that existing build types are accessed using `getByName`. Creating a new build type works the same way using `create("buildtype") { ... }`. Product flavors are accessed and created with these same two methods—but in a `productFlavors` block—as you can find out using autocomplete thanks to static typing.

## Dependencies

Adding dependencies in the Gradle Kotlin DSL is straightforward: Here, we omit dependencies that don't show a new concept and focus on a few different ones, shown in Listing 9.37.

**Listing 9.37 Adding Dependencies (Kotlin)**

[Click here to view code image](#)

```
dependencies {  
  
    val kotlin_version: String by rootProject.extra //  
  
    implementation("org.jetbrains.kotlin:kotlin-stdlib-  
    // ...  
  
    val moshi_version = "1.6.0"  
  
    implementation("com.squareup.moshi:moshi:$moshi_ver-  
    kapt("com.squareup.moshi:moshi-kotlin-codegen:$moshi_ver-  
    // ...  
  
    testImplementation("junit:junit:4.12")  
  
    androidTestImplementation("com.android.support.test-  
    androidTestImplementation("com.android.support.test-  
    }  
◀ ▶
```

Extras defined in the root build script are accessed by using the `rootProject.extra` map as a delegate (remember you can use maps as delegates). Other than that, you simply use `val` instead of `def` and add parentheses to method calls. The method names themselves are the same as in Groovy—for example, `implementation` and `kapt`—so they are easily discoverable.

### Experimental Features

The last block in the script enables experimental Kotlin Android Extensions. Unfortunately, at the time of writing, enabling experimental Android extensions does not work in the Kotlin DSL as expected. But the DSL allows you to inject Groovy closures at any point, meaning you can always fall back to Groovy to circumvent such issues, as done in Listing 9.38.

**Listing 9.38 Enabling Experimental Android Extensions (Kotlin)**

[Click here to view code image](#)

```
androidExtensions {  
    configure(delegateClosureOf<AndroidExtensionsExtens  
        isExperimental = true  
    })  
}
```

The given lambda is transformed to a Groovy `Closure<AndroidExtensionsExtension>`, where the `AndroidExtensionsExtension` is also the receiver of the `androidExtensions` lambda. So the closure is simply passed along. By the time you read this, the issue may have been resolved so I recommend trying it without the `configure` call first.

## Using buildSrc in Gradle

One way to modularize your Gradle builds is to make use of Gradle's `buildSrc` directory. If you add it under the project root, Gradle will compile it and add all its declarations to the classpath of your build scripts. The `buildSrc` directory is placed in the project's root directory (next to the `app` directory). Its structure is the same as for any module, including a Gradle build file of its own and the directory `buildSrc/src/main/java`. Inside this `java` directory, add a new file `GradleConfig.kt`. Any declarations in this file will be available in your build script so you can extract all your versions and dependencies into this file, as shown in Listing 9.39.

**Listing 9.39 Gradle Config in buildSrc**

[Click here to view code image](#)

```
private const val kotlinVersion = "1.2.50"

private const val androidGradleVersion = "3.1.3"

private const val supportVersion = "27.1.1"

private const val constraintLayoutVersion = "1.1.0"

// All versions as in build.gradle.kts...

object BuildPlugins {

    val androidGradle = "com.android.tools.build:gradle"

    val kotlinGradlePlugin = "org.jetbrains.kotlin:kotlin-gradle-plugin"

}

object Android {

    val buildToolsVersion = "27.0.3"

    val minSdkVersion = 19
```

```
val targetSdkVersion = 27

val compileSdkVersion = 27

val applicationId = "com.example.nutrilicious"

val versionCode = 1

val versionName = "1.0"

}

object Libs {

    val kotlin_std = "org.jetbrains.kotlin:kotlin-stdlib

    val appcompat = "com.android.support:appcompat-v7:$

    val design = "com.android.support:design:$supportVe

    // All dependencies as in build.gradle.kts...

}
```

This file now encapsulates all the concrete versions and dependencies of the app. To scope the properties, you can place them in corresponding objects.

Next, to enable Kotlin in this module, add the `build.gradle.kts` script from [Listing 9.40](#) directly into the `buildSrc` directory.

[Listing 9.40 buildSrc/build.gradle.kts](#)

[Click here to view code image](#)

```
plugins {

    `kotlin-dsl` // Uses ticks: ``

}
```

All declarations from the `GradleConfig` file can now be used in the build script. For the sake of brevity, [Listing 9.41](#)

only shows snippets, but the full source is available in the GitHub repository for this book.<sup>19</sup>

[19. https://github.com/petersommerhoff/kotlin-for-android-app-development](https://github.com/petersommerhoff/kotlin-for-android-app-development)

#### Listing 9.41 Using the Gradle Config

[Click here to view code image](#)

```
android {  
    // ...  
    targetSdkVersion(Android.targetSdkVersion) // Uses  
    versionCode = Android.versionCode  
    // ...  
}  
  
dependencies {  
    // ...  
    implementation(Libs.moshi) // Uses  
    kapt(Libs.moshi_codegen)  
    // ...  
}
```

The `BuildPlugins` object can be used in the same way in the root build script. Note that you don't need to import anything to use the declarations from `buildSrc` in your build scripts.

#### Benefits and Drawbacks

The benefits of the Gradle Kotlin DSL were already mentioned and are based on tool support through static typing. Android Studio can autocomplete available functions for further nesting, and which other properties and methods can be called. This is especially helpful when writing or extending a build script.

On the other hand, at the time of writing, Android Studio may not correctly discover the DSL methods when migrating to the Gradle Kotlin DSL. As mentioned, a restart usually solves this problem. Unfortunately, at the time of writing, Android Studio also does not seem to pick up on the `settings.gradle.kts` file correctly and does not show it under Gradle Scripts.

When using Gradle's `buildSrc`, Android Studio currently needs a rebuild to reflect updates to the files and provide the correct autocompletions. Also, Android Studio will no longer provide hints to indicate that newer versions of dependencies are available (but there is a Gradle plugin to do this<sup>20</sup>). If you currently use the root build script to store version and dependency info, following this approach can significantly clean up your root build script.

20. <https://github.com/ben-manes/gradle-versions-plugin>

Most of these issues refer to Android Studio and should be resolved in future releases. In summary, it is convenient to use Kotlin as a single language (that you know well) for both logic and build scripts. But considering the current tool limitations, sticking to Groovy is a reasonable choice.

## SUMMARY

You can now create your own simple Kotlin DSLs from scratch by combining higher-order functions, extensions, infix functions, and other language features. You have also seen how this concept is applied to Android layouts with Anko and Gradle build scripts with the Gradle Kotlin DSL. These two are currently the most prevalent Kotlin DSLs for Android development. Both have their advantages and drawbacks you have to weigh before deciding on which approach to use. In any case, Kotlin DSLs are a powerful tool to add to your toolbox to create even cleaner APIs to build complex objects or configurations.

## Migrating to Kotlin

*A language that doesn't affect the way you think about programming is not worth knowing.*

Alan J. Perlis

Migrating to a new programming language can be daunting. This chapter gives practices that have helped other companies successfully migrate. But because a migration is not just a technical matter, this chapter also covers the nontechnical aspects involved, such as communication, getting buy-in, and risks to consider.

### ON SOFTWARE MIGRATIONS

Changing any tool, technology, or technique involved in the software development process is not just a technical decision because it also affects business concerns, such as deployment cycles, user satisfaction, and estimated project effort. For these reasons, a migration requires buy-in from the whole team as well as the responsible manager(s).

Also, as a technical person, it is important to keep in mind that, from a business perspective, many more aspects come together in a project that affect its chance to succeed. Therefore, when pitching the adoption of a new tool like a programming language to your manager, you should have a realistic view of what impact this tool can have, taking into consideration all that makes up the project.

## Risks and Benefits

Every migration or tool change has inherent risks. This means that there is always a certain barrier to introducing change, and a certain “activation energy” necessary to trigger change. Risks exist on the technical and the business level. On the technical level, risks include problems integrating with the toolchain or the existing technology stack, unexpected roadblocks, leaving behind experience with the previous tool or technology, and transferring knowledge. On a business level, risks include negative effects on productivity, team and user satisfaction, and keeping deadlines even during the transition phase.

Each of these risks can also be turned into a benefit if the effects of the migration are predominantly positive. Namely, increasing satisfaction and motivation, fewer roadblocks and smoother development, better interoperability with the current technology stack, and so forth. As a rule of thumb, migrating to a new tool or technology will likely slow you down in the short term, but this initial slowdown should repay in the medium or long term. Of course, this can make it hard to struggle through the migration and justify its necessity—which is why buy-in is crucial before attempting to migrate to any new technology or tool.

**Note**

Think about each of these general points in terms of migrating from Java to Kotlin to contemplate benefits and risks it may have in your opinion.

Generally, a migration must have a defined purpose—and adopting a new technology because it is becoming popular is not a purpose. This is not to imply that adopting Kotlin doesn’t have a purpose—it can have many benefits as you will see. Good questions to ask before any migration include the following.

- Is the tool, technology, or technique mature and proven to be effective in practice?
- Are other companies successful with it?
- Does it have a supportive community?

- Do you have experts in the team who can facilitate the adoption?
- Is the team interested in using (and learning) the new tool or technology?
- Does it integrate well with the current technology stack and tooling?

In the case of Kotlin, it has been proven mature enough to be incorporated into the technology stack; many companies have done so successfully. These also contribute to Kotlin's very active community. The other questions depend on your company and team, but can also be affected by you—you can be the expert facilitating the change, and convince your team of the benefits Kotlin can have to spark interest in the language and the motivation to learn it.

In terms of tooling for Kotlin, there are certainly obstacles to overcome. For instance, compiler plugins and static analysis tools are not as abundant for this young language as they are for Java. Additionally, while Gradle is pushing forward with its excellent Kotlin support, other build tools do not offer special Kotlin support and may not play well with language features that don't exist in Java. The same can be said for several libraries such as Gson (a JSON mapper), which cannot handle concepts like primary constructors. While Gson can be replaced by Moshi when using Kotlin, not all libraries may have such a direct counterpart. Thus, the best way to evaluate Kotlin is to test it with *your* tech and tool stack to explore possible issues.

## LEADING THE CHANGE

Bringing about a tool change is a task that requires leadership, communication, and ultimately convincing. Especially when it comes to programming languages, developers tend to have strong opinions and preferences because it's one of the most immediate tools they constantly work with—just like the IDE and the version control system. Thus, adopting a new language may be one of the hardest changes if your team doesn't like the new language—or one of the easiest if you can get them excited about the new language. This section provides guidance and actionable tips to lead the change.

## Getting Buy-In

Adopting a new programming language requires buy-in from everyone involved—but how can you get that buy-in? This depends heavily on whom you want to convince (a technical versus business person) and at what kind of company you work (a startup versus a large corporation). Ultimately, with buy-in, developers are more likely to push through the obstacles that *will* occur during migration.

Here, I'll loosely differentiate between “technical people” and “business people” to give examples for arguments addressing both perspectives. Mostly, it is about framing the same underlying benefit in the correct context. Arguments you can use to entice the use of Kotlin include the following.

- **Fewer null-pointer exceptions:** Even after full migration to Kotlin, unfortunately you cannot guarantee complete null safety because you're still interoperating with Java in the Android SDK and likely in several dependencies. Still, Kotlin helps reduce null-pointer exceptions which, from a business perspective, means fewer app crashes, more ads shown, better user retention, and so forth.
- **Smaller pull request and code reviews:** With fewer lines of (readable) code, developers can review code faster. Also, you need less time to think about null cases in reviews if you use mostly non-nullable types in Kotlin and can focus on more interesting aspects instead. From a business perspective, faster reviews mean increased productivity and higher code quality.
- **Map language features to issues:** Be perceptive about common issues in your code base and link them to Kotlin features that can help mitigate such issues so that you have concrete examples for how Kotlin would improve the code base. Christina Lee used this approach, among others, to introduce Kotlin at Pinterest in 2016.<sup>1</sup>

1. <https://www.youtube.com/watch?v=mDpnc45WwII>

- **Excited and motivated developers:** Refer to surveys<sup>2,3</sup> that indicate that Kotlin users love the language<sup>4,5</sup> and therefore tend to be more motivated and more productive.
  - 2. <https://pusher.com/state-of-kotlin>
  - 3. <https://www.jetbrains.com/research/devecosystem-2018/kotlin/>
  - 4. <https://insights.stackoverflow.com/survey/2018/#technology-most-loved-dreaded-and-wanted-languages>
  - 5. <https://zeroturnaround.com/rebellabs/developer-productivity-report-2017-why-do-you-use-java-tools-you-use/>
- **Removing third-party dependencies:** If migrating to Kotlin fully,<sup>6</sup> libraries like Retrolambda,<sup>7</sup> Butter Knife,<sup>7</sup> Lombok, and AutoValue<sup>8</sup> are

no longer necessary in the long term, also reducing the method count and APK size.

6. <https://github.com/luontola/retrolambda>
7. <http://jakewharton.github.io/butterknife/>
8. <https://github.com/google/auto/tree/master/value>

- **Officially supported:** Last but not least, don't forget to mention Kotlin is an official language on Android backed by Google. This and the fact that JetBrains has a large team working on the language means that it's not going away anytime soon. For the business, this means it is a stable tool that is unlikely to become legacy in a few years.

These are by no means all the benefits. You can pour out all that you've learned in this book to give an accurate picture of Kotlin's advantages and potential drawbacks, then team up with others excited about Kotlin to lead the change. With enough buy-in from your team members, you can pitch the idea to everyone involved, for instance, by composing a brief document that pitches the language—like Jake Wharton did to lead the adoption at Square.<sup>9</sup>

9. <https://docs.google.com/document/d/1ReS3ep-hjxWA8kZi0YqDbEhCqTc29hG8P44aA9W0DM8/>

Although you want to focus on the benefits here, you must manage expectations. Obviously, you should paint an accurate picture of the language and communicate clearly that adoption comes with a learning curve that will initially slow down the team but should be amortized later. Also, the decision to migrate must be evaluated thoroughly; keep in mind that not migrating may be the better choice for you or your company.

## Sharing Knowledge

Sharing knowledge is particularly essential before and when starting the migration in order to inform people about the technology and what it can do. This should be planned for ahead of time as it will take time and effort across the team. Approaches to share knowledge successfully include:

- **Pair programming:** This allows instant feedback between colleagues and sparks discussions about best practices, idiomatic code, and so forth. Consider pairing a more experienced Kotlin developer with a language learner to speed up knowledge transfer.
- **Organize talks, presentations, and workshops:** These can be internal, in user groups, or one of the big Kotlin conferences like KotlinConf.<sup>10</sup>

10. <https://kotlinconf.com/>

- **User groups:** If you do not have a user group in your area yet, consider founding one to bounce ideas off likeminded developers, learn from them, and let them learn from you. If there is one already, it's an ideal way to expose yourself and your team to the language and community.
- **Promote collective discussions:** Especially while people familiarize themselves with Kotlin, it's important to address their doubts, ideas, questions, and opinions. Discussions with the whole team allow carving out agreed-upon practices, conventions, and action plans for the migration.
- **Document knowledge:** An internal wiki or another easily accessible resource is a great place to document best practices, benefits, risks, and all aspects affecting the migration.
- **Tap into the community:** The best way to succeed with migration is to learn from people who have done it. Luckily, the Kotlin community is extremely active and supportive, so remember to use it. Personally, I consider the Slack channel to be the primary place to tap into the community.<sup>11</sup>

11. <https://kotlinlang.org/community/>

## PARTIAL OR FULL MIGRATION

Let's assume you have successfully pitched Kotlin adoption, or maybe you just decided to migrate a pet project of yours. You need a migration plan. This section discusses advantages and disadvantages of the two main types of migration—partial migration versus full migration. As you will see, these have quite different consequences.

### Partial Migration

Partial migration means that you mix Kotlin and Java in your project. Even so, you get several benefits:

- **Reduced lines of code:** This affects overall code base size, code reviews, and pull requests.
- **The feeling of having Kotlin deployed:** This shows feasibility and builds trust that Kotlin can be used in production and deployed to clients without problems.
- **Experience gain:** Any exposure to Kotlin, especially writing code yourself, increases knowledge and experience. The more you migrate, the more proficient you become.

These are on top of all of Kotlin's benefits that you already know of. Unfortunately, partial migration and a polyglot code

base with mixed languages comes at a cost. The most important drawbacks to keep in mind include the following:

- **Harder to maintain:** Constantly switching between two languages in your code base means a lot of context switching and thus cognitive overhead. Also, in the case of Kotlin and Java, you will likely find yourself writing constructs of one language in the other from time to time because they look so similar.
- **Harder to hire people:** There are far fewer Kotlin developers than Java developers so, from a business perspective, it is important to be aware that it is harder to find proficient developers, and you may have increased onboarding time on the project.
- **Increased build times:** Mixing Kotlin and Java in your project will increase compilation and build times. Assuming you're using Gradle, incremental build times don't increase as much as clean build times, and Kotlin build times have also improved significantly. The first converted file has most impact, and each following Kotlin file does not significantly affect build time anymore. [Figure 10.1](#) demonstrates why this effect occurs. Basically, introducing just one Kotlin file requires the Kotlin compiler to compile all Java files that it depends on, which takes a noticeable amount of time.

Migrating module by module reduces this increase in build time and reduces the number of integration points and thus context switches. As mentioned, build tools other than Gradle may show more significant increases in build time because Gradle specifically works on its Kotlin integration.

- **Interoperability issues:** Although Kotlin interoperates well with Java, there are still several things to consider—which is why this book has a whole chapter dedicated to interoperability. If at least your own code is entirely in Kotlin, many interoperability issues disappear.
- **Hard to reverse:** Migrating and converting files from Java to Kotlin is easy but can be hard to reverse; in fact, it would require *rewriting* the file from scratch if too many changes have been made so that version control becomes useless.

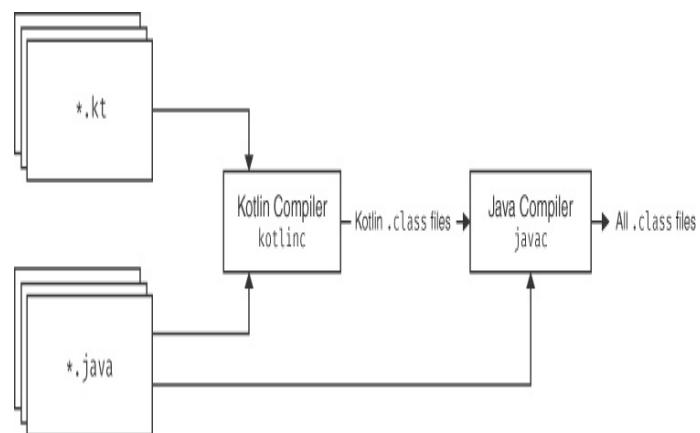


Figure 10.1 Compilation process of mixed-language project. Kotlin compiler uses both Java and Kotlin source files to link them properly but only emits class files for Kotlin source files

In short, a polyglot code base introduces several drawbacks that may outweigh the benefits of introducing Kotlin in your project. Therefore, it is important to realistically evaluate the pros and cons before doing a partial migration (which may be the only feasible option of migration). In large projects, even if your goal is to migrate entirely, you may have to live with these issues for a considerable amount of time. In smaller projects, where it can be done in a reasonable amount of time, I would recommend focusing on migrating the entire code base quickly (or not migrating at all).

## Full Migration

All of the benefits of partial migration above also apply to full migration, as well as some of the drawbacks.

- **It is still harder to hire people** but at least developers must be proficient only in Kotlin.
- **The migration is hard to reverse** once you've introduced enough changes in Kotlin so that the last Java version in version control is useless.
- **Interoperability issues are not entirely mitigated** because at least the generated R file and `BuildConfig` continue to use Java, as well as likely some of the libraries you use.

However, dedicating to a full migration brings great benefits.

*On top of all the benefits listed for partial migration, these include:*

- **Better build times** because all your own modules use exclusively Kotlin, taking full benefit of the reduction in build time this creates.
- **A system that is easier to maintain** because single-language production and test code requires fewer context switches, introduces fewer interoperability issues, and allows faster onboarding.

In summary, aim for a full migration *if* you do decide to adopt Kotlin at your company. For small projects, this can be done in a relatively short time. In larger projects, it may well be that a full migration is not feasible due to third-party dependencies, internal restrictions, or simply the effort involved. In these cases, you can still follow a migration plan as outlined below.

In any case, you should have an agreed-upon migration plan that everyone involved follows, and you should introduce Kotlin ideally module by module and for each module package

by package to minimize build time, the number of integration points, and context switches.

## WHERE TO START

If you decide to adopt Kotlin, whether partially or fully, the next question becomes: “Where do I start?” This section covers three ways to start integrating Kotlin into your code base, along with the respective advantages and disadvantages.

### Test Code

The first possibility is to start writing or migrating *test cases* in Kotlin. At the time of writing, this is proposed on the Android Developers website<sup>12</sup> and many companies have successfully used this approach. Test code is rather simple, so this is an easy place to start trying out Kotlin. It’s also relatively easy to migrate back to Java.

12. <https://developer.android.com/kotlin/get-started#kotlin>

However, there are many arguments against this approach. There are two basic scenarios: You are adding new test cases or migrating existing ones. In the first scenario, you’re adding new tests in Kotlin for existing functionality—so you’re testing after the fact and not in a test-first manner. Also, this only works if you had gaps in your tests in the first place. In the other scenario, you’re migrating existing Java test cases to Kotlin. This incurs several risks.

- **Bugs that you unknowingly introduce into the test code can lead to production bugs** in the corresponding functionality—and introducing such bugs can happen especially while you don’t have much experience with Kotlin yet. For instance, consider the subtle difference given in [Listing 10.1](#) that may cause a bug without your knowing.
- **You don’t refactor test cases as often as your production code.** Thus, an intricate bug in the test code is likely to remain for a long time.

[Listing 10.1 Subtle Difference](#)

[Click here to view code image](#)

---

```
val person: Person? = getPersonOrNull()
```

```
if (person != null) {  
    person.getSpouseOrNull() // Let's say t  
}  
else {  
    println("No person found (if/else)") // Not printed  
}  
  
person?.let {  
    person.getSpouseOrNull() // Let's say t  
} ?: println("No person found (let)") // Printed (because it's not null)
```

- **Test cases are not tested themselves, so you are basically operating in the dark.** Most test code is rather simple, but if you *do* make a mistake, you are less likely to notice.
- **In test code, you cannot usually make full use of Kotlin's powerful language features** that provide actual benefits. In fact, if you only rewrite your JUnit<sup>13</sup> test cases with Kotlin using JUnit, you won't gain much at all—you would mostly be writing Java-like code in Kotlin, not improving your code quality much at all.

13. <https://junit.org/>

All in all, although test code has become a popular starting point to introduce Kotlin, I personally consider the following two approaches preferable.

## Production Code

I'll assume here that your production code is thoroughly tested so that changes can be made with confidence, and if a bug is introduced during migration, one of the test cases should fail. This greatly supports migration because you know if you did something wrong. Other benefits include the following.

- **You're implementing actual product features**, which is not only a lot more motivating but also gives you the confidence to know that you can deploy Kotlin to your users.
- **You can make full use of Kotlin's language features**, even those that are rarely used in test code, such as sealed classes, coroutines, and delegated properties. You can compare your implementation directly with the previous one in Java to see the benefits.

- **You can start with very simple features** or functionality so that you are unlikely to make mistakes, then work your way up to migrating the more complex parts of your app.
- **Your production code is refactored regularly** so that even if you did introduce a bug or wrote unclean or unidiomatic code, it is more likely to be fixed soon after.

In summary, don't assume that migrating test code is the safest way to start migrating. If you have a strong test suite, making changes to your production code is a lot safer because you get direct feedback in case you introduce a bug.

If you do not yet have a (good) test suite, then adding new tests in Kotlin first is a reasonable alternative. Adding them after the fact is still better than not testing at all, and you can combine this with migrating the corresponding production code.

## Pet Projects

Pet projects are probably the best way to gain experience with Kotlin once you're familiar with the language—which you certainly are after working through this book. You should work on pet projects by yourself to further familiarize yourself with the language before pitching it to your company. If other team members are interested in evaluating Kotlin, it is the perfect chance to work on a pet project together. If you're considering adopting Kotlin at your company, work on the pet project with the team that would be affected by the adoption—ideally also using the same technology stack to encounter possible problems ahead of time. For instance, Kotlin does not play well with Lombok. So if you have a large project using Lombok where you cannot easily migrate all Lombok uses, you'll have to think about how to deal with this incompatibility beforehand.

On the downside, this approach costs effort and time without direct progress on company products. But there are many benefits to pet projects that can make them well worth the investment.

- **Pet projects provide a safe environment** to compare different solutions and conventions, such as finding an adequate balance between conciseness and expressiveness, how much to use functional

programming concepts, or what can be solved easily using the standard library.

- **Each team member has the chance to evaluate Kotlin's drawbacks and benefits** for himself or herself and discuss them with teammates.
- **You can use pair programming in pet projects** to accelerate knowledge transfer from people already familiar with Kotlin and to spark discussions.
- **You can collect data** (ideally with the same technology stack) to discover potential problems ahead of time. To measure build times, you can use the Gradle Profiler.<sup>14</sup>

14. <https://github.com/gradle/gradle-profiler>

- **You will come across issues early on**—and before using Kotlin in production. Whether this is not being able to run individual test cases with Spek<sup>15</sup> (a testing framework for Kotlin), Mockito<sup>16</sup> not always playing smoothly with Kotlin (MockK<sup>17</sup> is a good alternative), or the tendency of tooling to be behind a bit.

15. <https://spekframework.org/>

16. <https://site.mockito.org/>

17. <http://mockk.io/>

Pet projects are incredibly effective to evaluate Kotlin before even considering a migration. It not only gives you the opportunity to research build tool integrations, third-party libraries, and other integration points; it also allows you to start developing internal libraries that encapsulate common use cases in well-defined APIs and would be useful in future projects. What's more, it gives you a chance to evaluate testing best practices and test infrastructure.

Generally, I'd recommend starting off with pet projects in the team and, if Kotlin should be adopted, start with simple and well-tested functionality in a non-business-critical app.

## Make a Plan

The previous sections already outlined general practices that can all be part of a migration plan. Here, we summarize and extend upon them again as an overview.

- **Start with simple and thoroughly tested functionality** where you are unlikely to introduce bugs without noticing.
- **Migrate module by module**, and within that, package by package to improve build times and reduce integration points.
- **Plan when to migrate test code** and evaluate testing frameworks and infrastructure for Kotlin in pet projects.

- **Isolate Kotlin's API** for higher-level Java consumers to avoid interoperability issues from using Kotlin's standard library or own APIs from Java.
- **Write all new features in Kotlin** and enforce this in pull requests.
- **Consider migrating every file that you touch** to fix a bug or to refactor it.
- **Block dedicated time to focus on migration** in larger projects, for instance, in every sprint if you are using Scrum.

These general rules help guide the process. SoundCloud<sup>18</sup> and Udacity,<sup>19</sup> for example, both followed the last three points when adopting Kotlin.<sup>20,21</sup> Agree on a clear set of rules with your team, and work out a concrete migration plan that follows the above ideas.

18. <https://soundcloud.com/>

19. <https://udacity.com/>

20. <https://fernandocejas.com/2017/10/20/smooth-your-migration-to-kotlin/>

21. <https://engineering.udacity.com/adopting-kotlin-c12f10fd85d1>

## TOOL SUPPORT

The Java-to-Kotlin converter is a useful tool to speed up migration. This section covers how to use it, what to do after using it, what to take heed of, and general tips to facilitate migration.

### Java-to-Kotlin Converter

The converter is bundled into the Kotlin plugin so it's accessible in Android Studio and IntelliJ by default. It is useful not only to make quick progress when integrating Kotlin but also to learn the ropes for beginners by comparing the generated code to the original Java code.

You can trigger the converter in different ways. First, you can invoke it under *Code* and then *Convert Java File to Kotlin File* in Android Studio's menu to convert the current file. Second, whenever you paste code from a Java file into a Kotlin file, Android Studio will automatically prompt you to convert the code. Third, although this action is currently named *Convert Java File to Kotlin File*, it can convert whole packages, modules, or even projects. So you can right-click on any

directory in the project view and trigger the action from there to recursively convert all its Java files.

**Note**

Don't autoconvert large parts of your code base without a plan and the time to go through and refactor all converted files. Even then, I'd still recommend doing the conversion file by file to migrate a package or module to have better control over the process.

## Adjusting Autoconverted Code

Irrespective of how you decide to use the converter, you *will* have to adjust most converted code to follow best practices, to use idiomatic patterns, and to improve readability. After all, there is only so much an automated tool can do. Here, we provide a checklist of common changes you'll have to make.

- **Avoid the unsafe call operator** (`! !`) wherever possible—and do not hesitate to restructure the code to avoid nullability in the first place.
- **Move helper methods to file level** where appropriate.
- **Avoid overuse of companion objects** for all that was static in Java; consider using top-level declarations instead, and consider using **const** on top-level variables.
- **Join property declaration and initialization**—Android Studio will suggest this as well.
- **Decide whether to keep @Throws annotations** for methods. Recall that this is useful if they are also called from Java or for documentation purposes.
- **Use function shorthand syntax** where it is possible and improves readability.

More high-level questions to ask include:

- **Which methods would better be extension functions?** Especially when converting a utility class, you will likely want to turn its helper methods into extension functions. But other methods may also benefit from the transformation.
- **Can I use delegated properties?** For instance, if you're building a complex object that is only used under certain conditions, put it into a lazy property.
- **Can more declarations be immutable?** Rethink any use of mutable data and **var** to abide by Kotlin's mentality. This of course should be done in Java code as well.
- **Can I use read-only collections?** Although they're not strictly immutable, prefer Kotlin's read-only collections to mutable ones. As

Java only has mutable collections in its standard library, the converter will keep them and use types like `ArrayList`.

- **Could an infix function or operator improve readability?** You should use these judiciously (especially operators), but if they do fit a use case, they can increase readability at the call site.
- **Would part of the system benefit from a custom DSL?** For instance, if there is a complex class (whether your own or third-party) of which you frequently build objects, a type-safe builder DSL may be a good idea.
- **Would my asynchronous code benefit from coroutines?** For example, if you're using many `AsyncTasks`, using coroutines can greatly reduce complexity.

Not all these changes are trivial; some can require substantial refactoring. But all are important considerations to make on the way to a high-quality code base—after all, this is why you would want to migrate to Kotlin in the first place. My hope is that this checklist helps guide you to a code base that all developers agree was worth the work for the migration.

**Note**

Converting any file will delete the original `.java` file and add a new `.kt` file. Thus, version control history for the Java file quickly becomes useless when modifying the Kotlin code.

## SUMMARY

This chapter covered the technical and nontechnical aspects of migrating to Kotlin (or a new tool in general), from implications on build time and code base quality, to getting buy-in and pitching adoption at your company. This summary recaps the primary steps involved, roughly in a chronological order.

- Get a good understanding of Kotlin—which you have done with this book.
- Implement a pet project in Kotlin by yourself—you have already created two with guidance in this book, so try one without step-by-step guidance. Also, if you have an existing Java pet project, try migrating it to Kotlin.
- Watch for common issues in your company's code base and map them to Kotlin features that would help solve those issues.
- Talk to your colleagues about Kotlin and let them know what it can and cannot do. Paint an accurate picture about its benefits and drawbacks. Be open to discussions and critique to establish a culture of learning and to get buy-in from other team members.

- Pitch an evaluation of Kotlin at your company, for instance, with a document highlighting features, benefits, and compatibility with the company's technology stack.
- If your company wants to adopt Kotlin, decide in advance whether to aim for a full migration and agree on a migration plan.
- Work on a pet project with your team, ideally evaluating the same technology stack as the product you want to eventually migrate.
- Migrate a simple feature that is well tested to Kotlin and celebrate the fact that you can deploy Kotlin to production.
- Use the converter, but expect to adjust the autoconverted code.
- Don't stop at 90% if you aimed for a full migration, even if the last packages and modules are harder to migrate, require bigger restructuring, and you could be working on new features instead. Remember the benefits of full migration.

This is the bird's-eye view of the main steps involved in the adoption of Kotlin. Keep in mind that not every developer will be eager to switch to a new programming language, that it introduces risks on the technical and business level, and that it may in fact not be the best choice for your company. However, Kotlin *can* substantially improve developer experience, productivity, code quality, and eventually product quality. The recommendations in this chapter aim to help you evaluate which are true in your case.

# A

## Further Resources

### OFFICIAL RESOURCES

#### **Kotlin Reference:** <https://kotlinlang.org/docs/reference/>

This is the primary resource for information right from the source. It is well written and briefly covers all aspects of the language.

#### **Kotlin in Action:** <https://www.manning.com/books/kotlin-in-action>

This book written by Dmitry Jemerov and Svetlana Isakova from JetBrains is a great resource on the Kotlin language straight from two members of the Kotlin team.

#### **Talking Kotlin:** <http://talkingkotlin.com/>

In this podcast, Hadi Hariri from JetBrains talks to Kotlin developers about anything Kotlin—for instance DSLs, libraries, Groovy, or Kotlin/Native—for roughly 40 minutes per episode.

#### **Talks from KotlinConf 2017:** <https://bit.ly/2zSB2fn>

The first-ever Kotlin conference was packed with high-quality talks for all levels of experience. Topics include coroutines, building React apps, interop, data science, and much more. By the time you read this, there will also be videos from KotlinConf 2018 available on YouTube (provided above as a shortened URL).

### COMMUNITY

#### **Kotlin Slack Channel:** <http://slack.kotlinlang.org/>

This official Slack channel is packed with experienced Kotlin developers eager to share their knowledge and answer any questions, whether you're a beginner or advanced developer.

### **Kotlin Weekly Newsletter:** <http://www.kotlinweekly.net/>

This weekly newsletter is useful to stay current with the most popular articles written about Kotlin and to keep exploring new topics and best practices surrounding Kotlin.

### **All Community Resources:** <http://kotlinlang.org/community/>

The official Kotlin website maintains an overview of all community resources where you can stay updated on the platform you prefer.

## **FUNCTIONAL PROGRAMMING**

### **Kotlin Arrow:** <https://arrow-kt.io/>

The Arrow library is packed with functional types and abstractions—such as monads, monoids, and options—to build pure functional apps with Kotlin.

## **KOTLIN DSLS**

### **Anko:** <https://github.com/Kotlin/anko>

The Anko library for Android contains, among other things, a DSL to create layouts programmatically and replace XML layouts (see [Chapter 8](#), [Android App Development with Kotlin: Nutrilicious](#)).

### **Kotlin Gradle DSL:** <https://github.com/gradle/kotlin-dsl>

The Kotlin Gradle DSL allows you to write your build scripts with Kotlin instead of Groovy, enabling autocomplete, code navigation, and other tool support (see [Chapter 8](#)).

### **Kotlin HTML DSL:** <https://github.com/Kotlin/kotlinx.html>

This DSL allows you to write HTML code in a type-safe way in Kotlin. It's one of the most used DSL in the Kotlin ecosystem, particularly in combination with Kotlin/JS.

## **MIGRATING TO KOTLIN**

### **Christina Lee's talk on migration at Pinterest:**

<https://youtu.be/mDpnc45WwI>

In her talk, Christina Lee goes over the challenges she faced when pitching and finally introducing Kotlin at Pinterest.

**Jake Wharton's document to pitch at Square Inc:**

<https://docs.google.com/document/d/1ReS3ephjxWA8kZi0YqDbEhCqTt29hG8P44aA9W0DM8/edit?usp=sharing>

In this document, Jake Wharton summarizes Kotlin's benefits and argues why it is worth adopting for the development of Android apps.

**Adoption at Udacity:**

<https://engineering.udacity.com/adopting-kotlin-c12f10fd85d1>

In this article, Nate Ebel goes through his team's experience while migrating to Kotlin, including the challenges, tips they discovered, and what rules they followed during adoption.

## TESTING

**Spek:** <https://spekframework.org/>

Spek is a specification framework developed by JetBrains that allows you to write tests as client-readable specifications and avoid misunderstandings when implementing test cases.

**KotlinTest:** <https://github.com/kotlintest/kotlintest>

KotlinTest is a testing framework for Kotlin, and another alternative to Spek and JUnit that supports several testing styles.

**MockK:** <https://mockk.io/>

MockK is an extensive mocking library specifically for Kotlin. It allows you to mock final classes, provides a DSL for mocking, lets you mock coroutines, and much more.

**Note**

These are only a tiny slice of libraries, frameworks, and resources available for Kotlin. A good way to discover more is <https://kotlin.link/>, which curates a comprehensive list of everything related to Kotlin.

## Glossary

**Accessor:** Getter or setter; allows access to a property.

**Annotation:** Metadata attached to a certain piece of the code, such as a class or function.

**Annotation processor:** Tool that processes metadata from annotations, often at compile time.

**API (Application Programming Interface):** A set of well-defined interfaces for application development, like reusable building blocks.

**ART (Android Runtime):** Android runtime environment that translates bytecode into native instructions executed by the Android device.

**Block of code:** See “Code block.”

**Blocking (operation):** Operation that blocks execution of a thread, such as requesting a lock.

**Boilerplate:** Typically repeated code that is necessary to perform a certain task but does not add to the code’s actual purpose and may hamper understandability.

**Call site:** A place in the code where a function is called. See also “Declaration site,” “Use site.”

**Callback:** Function that is passed to another entity to be called at the appropriate time.

**Changeability (of code):** Ease with which developers can make changes to the code. See also “Maintainability.”

**Class:** A type with certain data and capabilities; acts as a blueprint to create objects from.

**Code block:** Arbitrary number of code lines encompassed within curly braces (in Kotlin).

**Collection:** Data structure that holds zero or more elements, such as a list or a set.

**Compile-time error:** Error that is found and indicated by the compiler, thus can be fixed before causing a crash at runtime.

See also “[Runtime error](#).”

**Composite pattern:** Design pattern that allows creating nested object hierarchies. See also “[Strategy pattern](#).”

**Composition:** Containment of one entity in another. Often used to delegate certain tasks to the contained entity. See also “[Delegation](#).”

**Conciseness (of code):** The level of textual brevity of the code. See also “[Syntax](#).”

**Constructor:** Special function that allows creating an object from a class. See also “[Class](#).”

**D8:** Android dexer that enables faster compilation than its predecessor DX.

**Declaration:** Introduces a new identifier and what it represents, for instance, a function or a variable of a certain type. See also “[Initialization](#).”

**Declaration site:** Place in the code where an identifier is declared. See also “[Use site](#),” “[Call site](#).”

**Delegation:** Referring to a different entity or implementation to implement functionality.

**Dispatch receiver:** Class containing the declaration of an extension function.

**Event handler:** Callback that is executed in case a given event occurs, such as a button click. See also “[Callback](#).”

**Expression:** Piece of code that has a value, such as calling a function or accessing a variable in Kotlin.

**Expressiveness (of code):** The ability of code to express the programmer’s intentions and the code’s functionality.

**Extension receiver:** The class extended by the extension function.

**File-level declaration:** Declaration directly on the file level, not within another declaration. Kotlin allows not only classes and objects on the file level but also variables and functions.

**First-class citizen:** Language construct that allows common operations such as assigning it to a variable, passing it in as an

argument, or returning it from a function.

**Function:** Piece of code that may accept inputs (as parameters), performs the operations defined in its function body, and may produce an output (as return value).

**Generator:** Code construct that emits a sequence of values, yielding them one at a time.

**I/O:** Input and output, for instance, from a database, the network, or files.

**Immutability:** See “[Mutability](#).”

**Inheritance:** The ability to inherit data and capabilities from parent classes to child classes.

**Initialization:** Assigns an initial value to a variable, oftentimes combined with its declaration. See also “[Declaration](#).”

**JVM (Java Virtual Machine):** Runtime environment for Java bytecode, whether compiled from Kotlin, Java, or another JVM language. It abstracts from the operating systems (OS) so that programs can run on any OS.

**Keyword:** Reserved word that carries a special meaning, such as `while`, `try`, or `null`. Soft keywords can be used as identifiers, hard keywords cannot. See also “[Modifier](#).”

**Maintainability (of code):** Ease with which developers can change, improve, and extend the code.

**Memory leak:** When memory cannot release data even though it is no longer needed by the program, for instance, because there are still unused references to the data.

**Method:** Function that is inside a class or object (not on the file level). See also “[Function](#),” “[File-level declaration](#).”

**Modifier:** Reserved word with special meaning in a declaration, such as `open`, `private`, or `suspend`. Can be used as identifier in other contexts. See also “[Declaration](#).”

**Mutability:** The ability of an identifier or data to change after initialization.

**Mutual exclusion:** Restricts access to a critical code section to a single concurrent unit (typically a thread, or a coroutine in Kotlin). See also “[Shared \(mutable\) state](#).”

**Nullability:** The ability of a variable or data to be assigned `null` as a value.

**Operator:** Shorthand syntax for function calls in Kotlin, such as `+`, `-`, `in`, or `%`.

**Readability (of code):** Ease with which developers can read and understand the code.

**Receiver class (of an extension function):** See “[Extension receiver](#).”

**Refactoring:** Improving the internal structure of code without changing its external behavior.

**Reliability (of code):** The level to which code behaves as expected and without failures.

**Runtime error:** Error that occurs only at runtime and can cause the program or app to crash. See also “[Compile-time error](#).”

**Scope:** A variable’s scope is the part of the code in which the variable is visible (accessible).

**Semantics:** Meaning of the syntactical constructs of a language (such as `val`, indicating a variable declaration that is not reassignable). See also “[Syntax](#).”

**Shared (mutable) state:** A program state shared by multiple concurrent units (threads or coroutines). Can lead to synchronization problems if it is mutable. See also “[Mutual exclusion](#).”

**Singleton:** A design pattern that allows at most (or exactly) one instance of a class at runtime.

**Smart cast:** Automatic type cast by the Kotlin compiler when type constraints and execution context permit. See also “[Type cast](#).”

**Statement:** Piece of code that defines actions to be done but does not have a value.

**Static typing:** Expression types are well defined before compile time, only based on the source code, to support type safety.

**Strategy pattern:** Design pattern in which the to-be-used algorithm is chosen at runtime. See also “[Composite pattern](#).”

**Syntax:** Textual rules that make up a valid program. See also “[Semantics](#).”

**Top-level declaration:** See “[File-level declaration](#).”

**Type:** The capabilities of a variable or data in general. Includes classes, concrete instantiations of generic classes, and functional types. See also “[Class](#).”

**Type cast:** Transforming a variable or data of a certain type into another type.

**Use site:** A place in the code where an identifier is used. See also “[Declaration site](#),” “[Call site](#).”

# Index

## SYMBOLS

---

- & (ampersand), 15**
- > (arrow), 16**
- \* (asterisk)**
  - multiplication operator (\*), 28
  - timesAssign operator (\*=), 28
- ` (backtick), 130**
- [ ] (brackets), 28**
- : (colon), 39**
- = (equal sign), 22**
- ! (exclamation mark)**
  - not (!) operator, 15
  - unsafe call operator (!!), 31, 355
- / (forward slash)**
  - divAssign operator (/=), 28
  - division operator (/), 28
- (minus sign)**
  - dec operator (--), 28
  - minusAssign operator (-=), 28
  - subtraction operator (-), 28
- % (percent sign)**
  - rem operator (%), 28
  - remAssign operator (%=), 28
- . (period), 28**
- | (pipe symbol), 15**
- +** (plus sign)
  - inc operator (++) , 28
  - plusAssign operator (+=), 28
- ? (question mark)**
  - elvis operator (?:), 30–31

safe call operator (?), 29–30

## A

---

**abstract classes**, 87–88

**abstract keyword**, 87

**accessing collections**, 47

**actionable data (Nutrilicious app)**, 307–311

**activities, binding coroutines to**, 177–179

**actor function**, 187–188

**actor model for concurrent programming**, 155, 186–192

- conflated channels, 188–189
- custom message types, 191–192
- history of, 192
- multiple actors on same channel, 190–191
- multiple channels, 189–190
- producers, 191
- ReceiveChannel, 188
- rendezvous channels, 190
- SendChannel, 187–188
- simple example, 186–187
- unlimited channels, 189

**adapters, RecyclerView**

- Kudoo app, 215–219
- Nutrilicious app, 246–248

**addition (+) operator**, 28

**address function**, 322, 323

**AddressBuilder class**, 322

**addresses, adding to users**

- multiple addresses, 323–325
- single addresses, 319–320

**addresses block**, 323–325

**AddTodoActivity**, 233–237

- creating programmatically, 328–329
- migrating to Anko layouts, 332

**algebraic data types**, [100](#)  
**ALGOL W**, [29](#)  
**aliases, type**, [107](#)  
**allprojects block (Gradle Kotlin DSL)**, [337](#)  
**also function**, [58–59](#)  
**Amount class**, [307–311](#)  
**ampersand (&)**, [15](#)  
**and (&&) operator**, [15](#)  
**Android**, [125](#)  
**android block (Gradle Kotlin DSL)**, [338–339](#)  
**Android KTX**, [7](#), [230](#)  
**Android layouts**  
    Anko  
        Anko dependencies, [329–330](#)  
        custom views, [334–335](#)  
        layout parameters, [330–331](#)  
        migrating Kudoo’s AddTodoActivity to, [332](#)  
        modularizing, [333](#)  
        previewing, [331](#)  
        simple layout example, [330](#)  
        XML layouts versus, [335](#)  
    creating programmatically, [328–329](#)  
    overview of, [328](#)  
**Android PacKage (APK)**, [258](#)  
**Android Runtime (ART)**, [5–6](#)  
**Android SDK (software development kit)**, [124](#)  
**Android Search Interface**, [265](#)  
**Android Studio**  
    autocomplete, [28](#)  
    compiled bytecode, viewing, [134–135](#)  
    decompiled Java code, viewing, [134–135](#)  
    Device File Explorer, [227](#)  
    overview of, [7](#), [8](#), [205](#)  
    Sample Data Directory, [258](#)

**Android Virtual Device (AVD),** [212](#)

**android-apt plugin,** [208–209](#)

**AndroidExtensionsExtension,** [340](#)

**Anko library,** [6](#)

- dependencies, [329–330](#)
- layouts
  - custom views, [334–335](#)
  - migrating Kudoo’s AddTodoActivity to, [332](#)
  - modularizing, [333](#)
  - parameters, [330–331](#)
  - previewing, [331](#)
  - simple example, [330](#)
  - XML layouts versus, [335](#)

**ankoView function,** [334–335](#)

**annotation processors,** [208–209](#)

**annotations**

- annotation processors, [208–209](#)
- @Database,** [223–224](#)
- @DslMarker,** [325–326](#)
- @Entity,** [222](#)
- @GET,** [255](#)
- @JvmField,** [135–136, 150](#)
- @JvmName,** [134, 137, 150](#)
- @JvmOverloads,** [141, 150](#)
- @JvmStatic,** [140, 150](#)
- @JvmSuppressWildcards,** [147–148](#)
- nullability, [128–129](#)
- @Query,** [255](#)
- @RestrictsSuspension,** [185](#)
- @SerializedName,** [261](#)
- @Throws,** [145–146, 149](#)
- @TypeConverter,** [303](#)
- @TypeConverters,** [303](#)
- @UserDsl,** [325](#)

**Any type, 88**

**APIs (application programming interfaces)**

Android KTX, 7

asynchronous APIs, wrapping, 195–197

Collections. *See collections*

Search API, 255–256, 258–260

USDA Nutrition API, fetching data from

API requests, performing, 256–257

Gradle dependencies for, 250–251

Retrofit interface, 251–256

**AppDatabase class, 223–225, 282**

**apply function, 54–55, 329**

**apps**

Kudoo

AddTodoActivity, 233–237

to-do items, adding, 233–237

to-do items, checking off, 237–239

event handlers, 237–239

finished app, 210

migrating to Anko layouts, 332

project setup, 210–212

RecyclerView, 212–221

Room database, 221–233

Nutrilicious

data mapping, 257–262

DetailsActivity, 293–301

empty search results, indicating, 311–312

Favorites fragment, 276–280

finished app, 241

food details, storing in database, 302–306

migrating to Gradle Kotlin DSL, 336–343

progress indicator, 312–313

project setup, 242–243

RDIIs (recommended daily intake), adding, 307–311

RecyclerView, 243–250, 261–262  
Room database, 280–288  
search interface, 265–268  
SearchFragment, 268–275  
SearchViewModel class, 262–264  
USDA Food Reports API, fetching data from, 288–293  
USDA Nutrition API, fetching data from, 250–257

**Archivable interface, 86**

**arguments, 22**

**arithmetic operators, table of, 28**

**Array type, 127**

**arrayListOf, 46**

**arrays**

- creating, 128
- invariance of, 111

**arrow (->), 16**

**ART (Android Runtime), 5–6**

**as operator, 90**

**assigning lambda expressions, 40**

**assignment operators, 28**

**associate function, 49**

**associating collections, 49**

**asterisk (\*)**

- multiplication operator (\*), 28
- timesAssign operator (\*=), 28

**async keyword, 157, 169–172**

**asynchrony**

- with async-await pattern, 157
- asynchronous APIs, wrapping, 195–197
- with callbacks, 155–156
- with coroutines, 158–159, 160
- with futures, 156–157

overview of, 152–153, 155–157

**autoconverted code, adjusting, 355–356**

**auto-generated Gradle configuration, 205–207**  
**AutoValue, 348**  
**AVD (Android Virtual Device), 212**  
**await keyword, 157**

## B

---

**backing fields, 72**  
**backtick (`), 130**  
**binding coroutines, 177–179**  
**bindUi function, 309**  
**Bloch, Joshua, 88**  
**blocks, 336**  
**Boolean data type, 13**  
**Bossidy, Larry, 69**  
**bounded type parameters, 118–119**  
**Buck, 7**  
**build scripts**  
    Gradle Kotlin DSL, migration to, 335–336  
        benefits of, 342  
        buildSrc directory, 340–342  
        drawbacks of, 342–343  
        Gradle settings, 336  
        module build script, 338–340  
        root build script, 336–338  
    overview of, 205–207  
**builders, coroutine**  
    actor, 186–191  
    async, 169–172  
    contexts, 172  
        accessing elements of, 181  
        coroutine dispatchers, 172–177  
        jobs, 177–181  
        future, 182  
        immutability through, 320–322

launch, 164–169  
overview of, 163  
parameters of, 181–183  
runblocking, 163–164  
withContext, 176

**build.gradle file**, 205–207

**buildIterator function**, 185

**BuildPlugins object**, 342

**buildscript block (Gradle Kotlin DSL)**, 336–337

**buildSequence function**, 184

**buildSrc directory**, 340–342

**business, Kotlin for**  
advantages of, 7–8  
companies incorporating Kotlin, 8  
platform compatibility, 9

**Butter Knife**, 348

**buy-in for migration, gaining**, 347–348

**Byte data type**, 13

## C

---

**C language**, 9

**C++**316

**callback hell**, 155–156

**callbacks**, 155–156

**calling**  
functions, 22  
chaining calls, 52–53  
extension functions, 25  
higher-order functions, 42  
infix functions, 27  
lambdas as arguments to, 42

**getters/setters**, 124–125

**Java code from Kotlin**  
considerations for, 132–133

getters, calling, 124–125  
identifiers, escaping, 130  
Kotlin-friendly Java code, writing, 149–150  
nullability, 125–129  
operators, 131  
overview of, 123–124  
SAM (single abstract method) types, 131–132  
setters, calling, 124–125  
vararg methods, calling, 130–131  
Kotlin code from Java, 133. *See also* [visibilities](#)  
data classes, 142–143  
extensions, 137–138  
file-level declarations, 136–137  
Java-friendly Kotlin code, writing, 149–150  
method overloading, 141  
properties, calling, 133–135  
properties, exposing as fields, 135–136  
sealed classes, 142–143  
static fields, 138–139  
static methods, 140  
methods, 80–81  
operators, 27

**casting types**

- ClassCastException, 90
- nullable types, 90
- smart casts, 30, 90–91

**catch keyword, 33–35**

**catching exceptions, 33–35**

**Ceylon, 4**

**chaining function calls, 52–53**

**channels**

- conflated, 188–189
- multiple, 189–190
- multiple actors on same channel, 190–191

overview of, 186–192  
producers, 191  
ReceiveChannel, 188  
rendezvous, 190  
SendChannel, 187–188  
simple example, 186–187  
unlimited, 189

**Char data type, 13**

**checked exceptions, 35–36**

**child coroutines, 177–178**

**class keyword, 69**

**ClassCastException, 90**

**classes**

- abstract, 87–88
- AddressBuilder, 322
- Amount, 307–311
- AppDatabase, 223–225, 282
- constructors
  - primary, 70, 82–83
  - secondary, 84
- data classes
  - calling from Java, 142–143
  - declaring, 94–95
  - inheritance, 96
- declaring, 69
  - top-level declarations, 93–94
  - visibility modifiers, 92–93
- delegated implementations, 79–80
- DetailsDto, 291
- DetailsRepository, 305–306
- DetailsViewModel, 298–300, 306
- domain classes, mapping data to
  - DTO-to-model mapping, 260–262
  - JSON-to-DTO mapping, 258–260

overview of, 257–258  
enum, 96–98  
FavoritesViewModel, 282–283, 286–287  
Food, 246, 280–281  
FoodDetails, 291–292, 302  
FragmentActivity, 230  
generic, 106–107  
immutable, 320–321  
inheritance  
    overriding rules for, 89  
    overview of, 84–85  
inner, 82  
KClass, 143–144  
methods  
    calling, 80–81  
    declaring, 80–81  
    dynamic dispatch, 80–81  
    extension, 81  
    getters, 71  
    overriding, 80–81  
    setters, 71  
nested, 82  
Nutrient, 308  
NutrientListConverter, 302–303  
NutrientTypeConverter, 303  
open, 88–89  
polymorphism, 80–81  
properties  
    adding, 70–71  
    backing fields, 72  
    delegated, 74–79  
    fields compared to, 71  
    late-initialized, 72–73  
    lazy, 75–76

observable, 76–77

sealed

- calling from Java, 142–143
- declaring, 98–100

SearchListAdapter, 246–248, 283–284

SearchViewModel, 262–264

TodoItem, 222

TodoItemDao, 222–223

types versus, 110

User

- adding addresses to, 319–320, 323–325
- declaring, 319

UserBuilder, 321

ViewHolder, 216–217

ViewModel

- Kudoo app, 227–230
- Nutrilicious app, 262–264

**class-local extensions, calling**, 138

**Clojure**, 4

**closed-by-default principle**, 85, 88

**ClosedSendChannelException**, 187

**COBOL**, 316

**code base, integrating Kotlin into**, 351

- pet projects, 353–354
- production code, 352–353
- test code, 351–352

**collections**

- accessing, 47
- associating, 49
- editing, 47
- filtering, 47–48
- folding, 51–52
- grouping, 49
- helper functions for, 46

- instantiating, 46
- Kotlin versus Java, 45–46
- mapping, 48
- minimum, maximum, and sum calculations, 49–50
- sorting, 50–51
- variance of, 111–112

**Collections API. *See collections***

- colon (:)**, 39
- combining higher-order functions**, 59–60
- companion keyword**, 103–105
- companion objects**, 103–105
- compiled bytecode, viewing**, 134–135
- compile-time polymorphism**, 108
- CompletableFuture**, 152, 182–183
- complex objects, building with DSLs (domain-specific languages)**, 318–320

**concise code**, 4

**concurrency. *See also coroutines***

- actor model for, 186–192
- conflated channels, 188–189
- custom message types, 191–192
- history of, 192
- multiple actors on same channel, 190–191
- multiple channels, 189–190
- producers, 191
- ReceiveChannel, 188
- rendezvous channels, 190
- SendChannel, 187–188
- simple example, 186–187
- unlimited channels, 189
- challenges of, 153–155
- generators, 184–185
- overview of, 151–153
- solutions to, 155–157

transformation between styles of, 193

**concurrent execution, 152–153**

**conditional expressions**

- if keyword, 15–19
- when keyword, 15–19

**configuration. *See also Kudoo app; Nutrilicious app***

- AVD (Android Virtual Device), 212
- Grade
  - annotation processors, 208–209
  - auto-generated configuration, 205–207
  - dependencies, adding, 207–208

**conflated channels, 188–189**

**const keyword, 135**

**constructor keyword, 83**

**constructors**

- primary, 70, 82–83
- secondary, 84

**contains operator, 28**

**contexts, coroutine, 172**

- accessing elements of, 181
- coroutine dispatchers, 172–177
- jobs, 177–181

**Continuation Passing Style (CPS), 156**

**continuations, 156**

**contravariance, 112**

**conversions, SAM (single abstract method), 131–132**

**Convert Java File to Kotlin File command (Code menu), 209, 355**

**converter, Java-to-Kotlin, 8, 355**

**Conway, Melvin E.**192

**cooperative multitasking, 157–158**

**coroutine builders**

- actor, 186–191
- async, 169–172

contexts, 172–181  
future, 182  
immutability through, 320–322  
launch, 164–169  
overview of, 163  
parameters of, 181–183  
runblocking, 163–164  
withContext, 176

### **coroutine dispatchers (Nutrilicious app), 283**

#### **CoroutineContext, 172**

CoroutineDispatcher, 172–177  
Job object, 177–181

#### **CoroutineDispatcher, 172–177**

#### **CoroutineExceptionHandler, 180**

#### **coroutines**

asynchrony with, 158–159, 160, 195–197  
binding to activities, 177–179  
builders  
    actor, 186–191  
    async, 169–172  
    contexts, 172–181  
    future, 182  
    immutability through, 320–322  
    launch, 164–169  
    overview of, 163  
    parameters of, 181–183  
    runblocking, 163–164  
    withContext, 176  
child, 177–178  
contexts, 172  
    accessing elements of, 181  
    coroutine dispatchers, 172–177  
    jobs, 177–181  
debugging, 193–195

generator implementation with, 184–185  
Gradle dependencies for, 256  
history of, 192  
interoperability with Java, 197–198  
lazy start, 182  
overview of, 157–158  
parameters of, 181–183  
setup for, 158  
suspending functions, 159–163  
wrappers for, 182

**CoroutineStart, 181–182**

**covariance, 110–112**

**Covey, Stephen, 123**

**CPS (Continuation Passing Style), 156**

**CPU-bound operations, 165**

**crossinline keyword, 44**

**custom message types, actors with, 191–192**

**custom views, Anko layouts, 334–335**

## D

---

**Dalvik virtual machine, 5–6**

**DAOs (data access objects)**

- FavoritesDao interface, 281
- TodoItemDao class, 222–223

**data classes**

- calling from Java, 142–143
- declaring, 94–95
- inheritance, 96

**data keyword, 94–95**

**data mapping (Nutrilicious app)**

- DTOs to models, 260–262
- JSON to DTOs, 258–260
- overview of, 257–258

**data transfer objects. *See* [DTOs \(data transfer objects\)](#)**

**data types.** *See also* [nullability](#)

- algebraic, [100](#)
- aliases, [107](#)
- Any, [88](#)
- casting
  - ClassCastException, [90](#)
  - nullable types, [90](#)
  - smart casts, [30, 90–91](#)
- checking, [89](#)
- classes versus, [110](#)
- function types, [39](#)
- generic type parameters, [105–106](#)
- mapped, [125](#)
- mapping, [14](#)
- nullable
  - casting, [90](#)
  - elvis operator (?:), [30–31](#)
  - overview of, [29](#)
  - safe call operator (?), [29–30](#)
  - unsafe call operator (!!), [31, 355](#)
- parameters, [40–41, 105–106](#)
- platform types, [126–128](#)
- table of, [13–14](#)
- type inference, [14–15](#)
- variance
  - bounded type parameters, [118–119](#)
  - contravariance, [112](#)
  - covariance, [110–112](#)
  - declaration-site, [113–116](#)
  - star projections, [119–121](#)
  - use-site, [116–118](#)

**@Database annotation, 223–224**

**databases, 6**

Kudoo Room database

AppDatabase class, 223–225  
Gradle dependencies for, 221–222, 225–226  
LiveData class, 230–233  
MainActivity, 226–227  
TodoItem class, 222  
TodoItemDao class, 222–223  
ViewModel class, 227–230  
Nutrilicious Room database  
    AppDatabase class, 282  
    coroutine dispatchers, 283  
    FavoritesDao interface, 281  
    FavoritesViewModel class, 282–283, 286–287  
    Food class, 280–281  
    SearchFragment, 287–288  
    SearchListAdapter class, 283–284  
    storing food details in, 302–306  
    toggleFavorite function, 285  
**deadlock**, 153  
**debugging coroutines**, 193–195  
**dec operator (--)**, 28  
**declarations**  
    classes, 69  
        data, 94–95  
        enum, 96–97  
        generic, 106–107  
        sealed, 98–100  
        top-level declarations, 93–94  
        User, 319  
        visibility modifiers, 92–93  
    file-level, 136–137  
    functions, 21  
        extension, 25  
        generic, 107–108  
        infix, 27

main, 22–23  
suspending, 159

generators, 184

getters/setters, 71

methods, 80–81

objects, 103–105

operators, 27

variables, 12–13

- mutable, 12
- read-only, 12

**declaration-site variance, 113–116, 146–148**

**decompiled Java code, viewing, 134–135**

**deep nesting, 323–325**

**DefaultDispatcher object, 165**

**delegated implementations, 79–80**

**delegated properties**

- implementation of, 74–75
- lazy properties, 75–76
- maps, 78–79
- observable properties, 76–77
- syntax of, 74

**dependencies**

- adding, 207–208
- Anko, 329–330

**Descartes, René, 37**

**design patterns, Strategy, 39**

**DetailsActivity**

- click handler, 297
- data display, 300–301
- DetailsViewModel class, 298–300
- FOOD\_ID\_EXTRA identifier, 298
- item click listener, 296–297
- layout, 293–296
- navigation, 301

string resources for headline, 296

**DetailsDto** class, 291

**DetailsRepository** class, 305–306

**DetailsViewModel** class, 298–300, 306

development of Kotlin, 3–4

**Device File Explorer**, 227

**DiffUtil**, 233

dimension resources, 214

dining philosophers analogy, 153–155

direct style, 156

directories

- buildSrc, 340–342
- Sample Data Directory, 258

**dispatch receivers**, 81

dispatchers, coroutine, 172–177, 283

division (/) operator, 28

to-do list app. *See Kudoo app*

domain classes, mapping data to

- DTO-to-model mapping, 260–262
- JSON-to-DTO mapping, 258–260
- overview of, 257–258

domain-specific languages. *See DSLs (domain-specific languages)*

**Double** data type, 13

**doubleArrayOf** method, 128

do-while loops, 19

**drop** function, 64–65

**@DslMarker** annotation, 325–326

**DSLs (domain-specific languages)**

- benefits of, 316–317
- creating
  - complex objects, building, 318–320
  - deep nesting, 323–325
- @DslMarker** annotation, 325–326

immutability through builders, 320–322  
simple objects, building, 318  
definition of, 315–316  
drawbacks of, 317–318  
embedded, 316  
Gradle Kotlin DSL, migration to, 335–336  
    benefits of, 342  
    buildSrc directory, 340–342  
    drawbacks of, 342–343  
    Gradle settings, 336  
    module build script, 338–340  
    root build script, 336–338  
    history of, 315  
    language features, 326–328  
    layouts, 328  
        creating programmatically, 328–329  
        creating with Anko, 329–335  
    performance of, 327–328  
    readability of, 323–327  
**DTOs (data transfer objects)**  
    DetailsDto, 291  
    mapping JSON data to, 258–260  
    mapping to models, 260–262  
    wrappers for, 290  
**dynamic dispatch, 80–81**

## E

---

**eager evaluation, 62, 65**  
**Eclipse, 7**  
**editing collections, 47**  
***Effective Java (Bloch), 88***  
**Elixir, 186**  
**else keyword, 15**  
**else-if keyword, 15**

**elvis operator (?:),** [30–31](#)

**embedded DSLs (domain-specific languages),** [316](#)

**empty search results, indicating,** [311–312](#)

**@Entity annotation,** [222](#)

**enum classes,** [96–98](#), [309](#)

**enumerations,** [96–98](#)

**equal sign (=)**

- assignment operators, [28](#)
- referential equality operator (==), [32](#)
- single-expression functions, [22](#)
- structural equality operator (==), [32](#)

**equality**

- checking, [32](#)
- floating point, [32–33](#)

**Erlang,** [186](#)

**escaping Java identifiers,** [130](#)

**evaluation, eager,** [62](#), [65](#)

**evaluation, lazy**

- concept of, [62](#)
- definition of, [38](#)
- lazy sequences, [63–66](#)
  - creating, [63–64](#)
  - drop function, [64–65](#)
  - performance of, [65–66](#)
  - take function, [64–65](#)

**event handlers (Kudoo app),** [237–239](#)

**Evernote,** [8](#)

**exception handling**

- checked exceptions, [35–36](#)
- ClassCastException, [90](#)
- ClosedSendChannelException, [187](#)
- CoroutineExceptionHandler, [180](#)
- Java interoperability, [145–146](#)
- NullPointerException, [4](#), [8](#), [29](#)

principles of, 33–35  
suspending functions, 162  
TimeoutCancellationException, 177  
unchecked exceptions, 35–36

**exclamation mark (!)**  
not (!) operator, 15  
unsafe call operator (!!), 31, 355

**execution, concurrent**, 152–153

**experimental extensions, enabling in Gradle Kotlin DSL**, 340

**expressions**  
conditional  
if keyword, 15–19  
when keyword, 15–19  
eager evaluation, 62, 65  
lambda  
as arguments to higher-order functions, 42  
assigning, 40  
defining, 40  
definition of, 37  
implicit arguments, 41  
with receivers, 60–61  
type inference, 40–41  
lazy evaluation  
concept of, 62  
definition of, 38  
lazy sequences, 63–66  
object, 101–102

**extension functions**  
Android KTX, 230  
creating, 25  
importing, 26  
scope, 26  
static resolving of, 25–26

**extension methods, 81**  
**extension receivers, 81**  
**extensions**  
    AndroidExtensionsExtension, 340  
    calling from Java, 137–138  
    experimental, enabling in Gradle Kotlin DSL, 340

## F

---

**Favorites fragment, 276–280**  
**FavoritesDao interface, 281**  
**FavoritesViewModel class, 282–283, 286–287**  
**fetchDetailsFromApi function, 306**  
**fetching data**  
    from USDA Food Reports API, 288–293  
    from USDA Nutrition API  
        API requests, performing, 256–257  
        Gradle dependencies for, 250–251  
        Retrofit interface, 251–256  
**fib function, 21–22**  
**fields, 71**  
    exposing properties as, 135–136  
    static, calling from Java, 138–139  
**file-level declarations, 136–137**  
**filter function, 47–48**  
**filtering collections, 47–48**  
**Find Action command, 207**  
**FindBugs, 125**  
**findViewById function, 217–218**  
**first-order functions, 39**  
**Float data type, 13**  
**Floating Point Arithmetic, 32**  
**floating point equality, 32–33**  
**flow control**  
    conditional expressions

if keyword, 15–19  
when keyword, 15–19

loops  
do-while, 19  
for, 20–21  
while, 19

**fold function, 51–52**

**folding collections, 51–52**

**food and nutrition app. *See Nutrilicious app***

**Food class, 246, 280–281**

**food details, storing in database, 302–306**

**FOOD\_ID\_EXTRA identifier, 298**

**FoodDetails class, 291–292, 302**

**for loops, 20–21**

**Ford, Henry, 205**

**Fortran, 316**

**forward slash (/)**  
divAssign operator (/=), 28  
division operator (/), 28

**FragmentActivity class, 230**

**fragments**  
Favorites, 276–280  
FragmentActivity class, 230  
SearchFragment  
adding to UI, 273  
database connection, 287–288  
fragment transactions, 272  
getViewModel extension, 270–271  
layout for, 268–269  
MainActivity, 269  
methods, 269–270  
onCreate method, 274  
properties, 272  
recoverOrBuildSearchFragment method, 274

restoring, 274  
search intents, 275  
setUpSearchRecyclerView function, 271  
storing in activity’s state, 273  
swipe refresh, 275  
SwipeRefreshLayout, 271–272  
updateListFor function, 271  
transactions, 272

**frameworks for testing, 353**

**fromString method, 309**

**full migration, 350–351**

**fun keyword, 21, 25, 130**

**functional programming.** *See also functions*

- benefits of, 38–39
- collections
  - accessing, 47
  - associating, 49
  - editing, 47
  - filtering, 47–48
  - folding, 51–52
  - grouping, 49
  - helper functions for, 46
  - instantiating, 46
  - Kotlin versus Java, 45–46
  - mapping, 48
  - minimum, maximum, and sum calculations, 49–50
  - sorting, 50–51
  - variance of, 111–112
- lambda expressions
  - assigning, 40
  - defining, 40
  - definition of, 37
  - implicit arguments, 41
  - type inference, 40–41

lazy evaluation, 38

purpose of, 37–38

**functions.** *See also* **lambda expressions; methods;**

### **operators**

actor, 187–188

address, 322, 323

also, 58–59

ankoView, 334–335

apply, 54–55, 329

arguments, 22

arrayListOf, 46

associate, 49

bindUi, 309

buildIterator, 185

buildSequence, 184

callbacks, 155–156

calling, 22, 52–53

chaining calls to, 52–53

declaring, 21

drop, 64–65

extension

    creating, 25

    importing, 26

    scope, 26

    static resolving of, 25–26

fetchDetailsFromApi, 306

fib, 21–22

filter, 47–48

findViewById, 217–218

first-order, 39

fold, 51–52

function types, 39

generic, 107–108

getDetails, 300–301, 306

getPercentOfRdi, 310  
getViewModel, 264  
groupBy, 49  
hashMapOf, 46  
hashSetOf, 46  
higher-order, 37. *See also* specific function names  
    calling, 42  
    combining, 59–60  
    defining, 41  
    inlining, 42–43  
    lambdas as arguments to, 42  
    non-local returns, 44–45  
    suspending functions and, 162–163  
infix, 26–27, 323–327  
inline, 145  
lazy evaluation  
    concept of, 62  
    lazy sequences, 63–66  
let, 53–54  
linkedMapOf, 46  
linkedSetOf, 46  
listOf, 46  
lparams, 330–331  
main, 22–23  
makeSection, 309  
map, 48  
mapOf, 46  
maxBy, 49–50  
minBy, 49–50  
mutableListOf, 46  
mutableMapOf, 46  
mutableSetOf, 46  
overloading, 24–25  
parameters, 23–24

plus, 180  
pointers to, 39  
printAll, 119–121  
receive, 187  
reduce, 52  
reduceRight, 52  
renderNutrient, 301, 310  
run, 56–58  
send, 187  
sendEmail, 24  
setLoading, 313  
setOf, 46  
setUpSearchRecyclerView, 271  
signatures, 21–22, 39, 144–145, 159  
single-expression, 22  
sorted, 50  
sortedArray, 50  
sortedBy, 50  
sortedDescending, 50  
sortedMapOf, 46  
sortedSetOf, 46  
sortedWith, 50  
sumBy, 49–50  
suspendCoroutine, 183  
suspending, 157, 159–163, 164, 198–200  
take, 64–65  
toast, 330  
toggleFavorite, 285  
toSortedMap, 50  
updateListFor, 271, 275, 311  
updateUiWith, 313  
use, 59  
user, 318  
verticalLayout, 330

with, 55–56  
withContext, 176  
withTimeout, 176  
yield, 184  
**future coroutine builder, 182**  
**futures, 152, 156, 255**

## G

---

**generators, 184–185**  
**generics**  
    benefits of, 105  
    classes, 106–107  
    functions, 107–108  
    reification, 108–109  
    type parameters, 105–106  
**@GET annotation, 255**  
**get method, 71**  
**get operator, 28**  
**getDetails function, 300–301, 306**  
**getFoods method, 257**  
**getItemCount method, 216, 218–219, 246**  
**getPercentOfRdi function, 310**  
**getters**  
    calling, 124–125  
    declaring, 71  
**getViewModel extension, 270–271**  
**getViewModel function, 264**  
**Git, 212**  
**GitHub, 158, 212, 242, 307**  
**.gitignore file, 212**  
**Go, 186**  
**Google, support for Kotlin, 5**  
**Gradle dependencies**  
    Kudoo app, 221–222, 225–226

Nutrilicious app  
    architectural components, 262  
    Kotlin coroutines, 256  
    Moshi, 258  
    network and API access, 250–251

**Gradle Kotlin DSL, migration to, 7, 335–336**  
    benefits of, 342  
    buildSrc directory, 340–342  
    configuration  
        annotation processors, 208–209  
        auto-generated, 205–207  
        dependencies, adding, 207–208  
    drawbacks of, 342–343  
    Gradle settings, 336  
    Kudoo app, 221–222, 225–226  
    module build script  
        android block, 338–339  
        dependencies, 339–340  
        experimental features, 340  
        plugins block, 336–338  
    root build script  
        allprojects block, 337  
        buildscript block, 336–337  
        delete task, 337–338  
    support for, 346

**Gradle Profiler, 353**

**groupBy function, 49**

**grouping collections, 49**

**Gson, 346**

## H

---

**hashMapOf function, 46**

**hashSetOf function, 46**

**helper functions, for collections, 46**

**Hewitt, Carl**, 186, 192

**higher-order functions.** *See also specific function names*

calling, 42

combining, 59–60

defining, 41

inlining, 42–43

lambdas as arguments to, 42

non-local returns, 44–45

overview of, 37

suspending functions and, 162–163

**Hoare, Tony**, 29

**HTTPLoggingInterceptor**, 253

---

## I

---

**identifiers, escaping**, 130

**if keyword**, 15–19

**immutability**

data classes, 320–321

through builders, 320–322

variables, 13

**implementations, delegated**, 79–80

**implicit arguments, lambda expressions**, 41

**importing extension functions**, 26

**in operator**, 28

**inc operator (++)**, 28

**indexed access operator**, 47

**inference, type**, 14–15

**infix functions**, 26–27, 323–327

**infix keyword**, 26–27

**inheritance**

abstract classes, 87–88

closed-by-default principle, 85, 88

data classes, 96

interfaces, 85–86

open classes, 88–89  
overriding rules for, 89  
overview of, 84–85

**initializing objects**, 55

**inline functions**

- calling from Java, 145
- higher-order functions, 42–43

**inline keyword**, 42–43, 145

**inner classes**, 82

**instantiating collections**, 46

**Int data type**, 13

**intArrayOf method**, 128

**integrating Kotlin into code base**, 351

- pet projects, 353–354
- production code, 352–353
- test code, 351–352

**IntelliJ IDEA**, 7, 8

- autocompletion, 28
- compiled bytecode, viewing, 134–135
- decompiled Java code, viewing, 134–135

**Interceptor interface**, 254

**interfaces**

- Archivable, 86
- declaring, 92–93
- FavoritesDao, 281
- inheritance, 85–86
- Interceptor, 254

**internal keyword**, 92, 94

**Internet access, enabling**, 256

**interoperability with Java**

- best practices for, 149–150
- Collections API, 45–46
- CompletableFuture, 182–183
- coroutines, 197–198

Java code, calling from Kotlin  
considerations for, 132–133  
getters, 124–125  
identifiers, escaping, 130  
Kotlin-friendly Java code, writing, 149–150  
nullability, 125–129  
operators, 131  
overview of, 4, 123–124  
SAM (single abstract method) types, 131–132  
vararg methods, 130–131  
Java-to-Kotlin converter, 209  
Kotlin code, calling from Java, 133  
data classes, 142–143  
extensions, 137–138  
file-level declarations, 136–137  
Java-friendly Kotlin code, writing, 149–150  
method overloading, 141  
properties, calling, 133–135  
properties, exposing as fields, 135–136  
sealed classes, 142–143  
static fields, 138–139  
static methods, 140  
variance, 115, 116  
visibilities, mapping to Java  
exception handling, 145–146  
inline functions, 145  
KClass, 143–144  
Nothing type, 148  
overview of, 143  
signature clashes, 144–145  
variant types, 146–148  
**interpolation, string**, 20  
**iOS**, 3  
**is operator**, 89

**it keyword**, [61](#)  
**item layout**, Nutrilicious app, [244–246](#)  
**iterator method**, [124](#)

## J

---

**Java code, calling from Kotlin**  
considerations for, [132–133](#)  
getters, [124–125](#)  
identifiers, escaping, [130](#)  
nullability  
    annotations for, [128–129](#)  
    nullables and mapped types, [125](#)  
    overview of, [125](#)  
    platform types, [126–128](#)  
operators, [131](#)  
overview of, [4](#), [123–124](#)  
SAM (single abstract method) types, [131–132](#)  
vararg methods, [130–131](#)

**Java interoperability**  
best practices for, [149–150](#)  
Collections API, [45–46](#)  
CompletableFuture, [182–183](#)  
coroutines, [197–198](#)  
Java code, calling from Kotlin  
    considerations for, [132–133](#)  
    getters, calling, [124–125](#)  
    identifiers, escaping, [130](#)  
    Kotlin-friendly Java code, writing, [149–150](#)  
    nullability, [125–129](#)  
    operators, [131](#)  
    overview of, [123–124](#)  
    SAM (single abstract method) types, [131–132](#)  
    setters, calling, [124–125](#)  
    vararg methods, calling, [130–131](#)

Java-to-Kotlin converter, 8, 209, 355  
Kotlin code, calling from Java, 133  
    data classes, 142–143  
    extensions, 137–138  
    file-level declarations, 136–137  
    Java-friendly Kotlin code, writing, 149–150  
    method overloading, 141  
    properties, calling, 133–135  
    properties, exposing as fields, 135–136  
    sealed classes, 142–143  
    static fields, 138–139  
    static methods, 140  
Kotlin’s advantages over Java, 6, 9  
overview of, 4  
streams, 62  
variance, 115, 116  
visibilities, mapping to Java  
    exception handling, 145–146  
    inline functions, 145  
    KClass, 143–144  
    Nothing type, 148  
    overview of, 143  
    signature clashes, 144–145  
    variant types, 146–148  
**Java on Android, 5–6**  
**Java Streams, 6**  
**JavaFX, 6, 182**  
**JavaScript, 3**  
    Kotlin/JS, 9  
    Kotlin’s advantages over, 9  
**Java-to-Kotlin converter, 8, 209, 355**  
**JetBrains, 3–4, 7, 125**  
**Job object, 177–181**  
**jobs, coroutine, 177–181**

**JSON data, mapping to DTOs (data transfer objects),** 258–  
260

**JSR 305, 125**

**JUnit, 124, 352**

**JVM (Java virtual machine),** 3, 5, 9

**@JvmField annotation, 135–136, 150**

**@JvmName annotation, 134, 137, 150**

**@JvmOverloads annotation, 141, 150**

**@JvmStatic annotation, 140, 150**

**@JvmSuppressWildcards annotation, 147–148**

## K

---

**kapt plugin, 208–209**

**kaptAndroidTest, 209**

**kaptTest, 209**

**KClass, 143–144**

**Keller, Gary, 151**

**keywords. *See also* coroutine builders; functions; methods**

abstract, 87

async, 157, 169–172

await, 157

catch, 33–35

class, 69

companion, 103–105

const, 135

constructor, 83

crossinline, 44

data, 94–95

do-while, 19

else, 15

else-if, 15

for, 20–21

fun, 21, 25, 130

if, 15–19

infix, 26–27  
inline, 42–43, 145  
internal, 92, 94  
it, 61  
lateinit, 73  
launch, 164–169  
object, 101–102, 130  
open, 87  
override, 86, 104–105  
private, 92, 94  
protected, 92  
public, 92, 94  
reified, 109  
runblocking, 163–164  
sealed, 98–100  
static, 104  
super, 87  
suspend, 159–163, 198–200  
this, 61  
throw, 33–35  
try, 33–35  
val, 12, 71, 83, 130  
var, 12–13, 71, 83, 130  
when, 15–19, 99–100  
where, 118–119  
while, 19

**knowledge sharing, 348–349**

**Kotlin/JS, 9**

**Kotlin/JVM, 9**

**Kotlin/Native, 9**

**.kts file extension, 11–12**

**Kudoo app**

AddTodoActivity, 233–237  
creating programmatically, 328–329

migrating to Anko layouts, 332  
to-do items, adding, 233–237  
to-do items, checking off, 237–239  
event handlers, 237–239  
finished app, 210  
project setup, 210–212  
RecyclerView  
    adapter, 215–219  
    layouts, 213–214  
    MainActivity, 220–221  
    model, 215  
    overview of, 212–213  
Room database  
    AppDatabase class, 223–225  
    Gradle dependencies for, 221–222, 225–226  
    LiveData class, 230–233  
    MainActivity, 226–227  
    TodoItem class, 222  
    TodoItemDao class, 222–223  
    ViewModel class, 227–230

## L

---

**lambda expressions, 5–6**  
as arguments to higher-order functions, 42  
assigning, 40  
defining, 40  
definition of, 37  
implicit arguments, 41  
with receivers, 60–61  
suspending, 164  
type inference, 40–41  
**lateinit keyword, 73**  
**lateinit property, 136, 139**  
**late-initialized properties, 72–73**

**LaTeX, 316**

**launch coroutine builder, 164–169**

**layouts**

Anko

Anko dependencies, 329–330

custom views, 334–335

layout parameters, 330–331

migrating Kudoo’s AddTodoActivity to, 332

modularizing, 333

previewing, 331

simple layout example, 330

XML layouts versus, 335

creating programmatically, 328–329

Nutrilicious app

    DetailsActivity, 293–296

    Favorites fragment, 276

    item layout, 244–246

    MainActivity, 243–244

    SwipeRefreshLayout, 271–272

    overview of, 328

    RecyclerView, 213–214

    SearchFragment, 268–269

    XML, 335

**lazy evaluation**

    concept of, 62

    definition of, 38

    lazy sequences

        creating, 63–64

        drop function, 64–65

        overview of, 63–66

        performance of, 65–66

        take function, 64–65

**lazy properties, 75–76**

**lazy sequences**

creating, 63–64  
drop function, 64–65  
overview of, 63–66  
performance of, 65–66  
take function, 64–65

### **leading migration to Kotlin**

buy-in, gaining, 347–348  
knowledge sharing, 348–349

### **Lee, Christina, 347**

### **let function, 53–54**

### **libraries. *See also* interoperability with Java**

Anko. *See* Anko library  
AutoValue, 348  
Buck, 7  
Butter Knife, 348  
Gradle, 7  
Gson, 346  
Lombok, 348  
Mockito, 7  
Moshi, 346  
Retrofit, 7  
Retrolambda, 348

### **linkedMapOf function, 46**

### **linkedSetOf function, 46**

### **List type, 127**

### **listOf function, 46**

### **LiveData class, 230–233**

### **locks, 154**

### **Lombok, 348**

### **Long data type, 13**

### **long[] type, 127**

### **LongArray! type, 127**

### **longArrayOf method, 128**

### **loops**

do-while, 19  
for, 20–21  
suspending functions, 161  
while, 19

## **lparams function, 330–331**

## **Lyft, 8**

---

# M

---

**main function, 22–23**

## **MainActivity**

Kudoo app, 220–221, 226–227  
Nutrilicious app  
    RecyclerView, 243–244, 248–250  
    SearchFragment, 269

## **makeSection function, 309**

## **map function, 48**

## **Map type, 127**

## **mapOf function, 46**

## **mapped types, 125**

## **mapping data**

    collections, 48  
    data types, 14  
    Nutrilicious app  
        DTOs to models, 260–262  
        JSON to DTOs, 258–260  
        overview of, 257–258

## **visibilities**

    exception handling, 145–146  
    inline functions, 145  
    KClass, 143–144  
    Nothing type, 148  
    overview of, 143  
    signature clashes, 144–145  
    variant types, 146–148

**maps, properties delegated to**, 78–79  
**maxBy function**, 49–50  
**maximum, calculating**, 49–50  
**methods. *See also* functions**  
    calling, 80–81  
    constructors  
        primary, 70, 82–83  
        secondary, 84  
    declaring, 80–81  
    doubleArrayOf, 128  
    extension, 81  
    fromString, 309  
    get, 71  
    getFoods, 257  
    getItemCount, 216, 218–219, 246  
    getters, 71, 124–125  
    intArrayOf, 128  
    iterator, 124  
    Java  
        calling as operators, 131  
        vararg methods, calling, 130–131  
    longArrayOf, 128  
    observeFavorites, 287  
    onAttach, 269  
    onBindViewHolder, 216, 218, 246  
    onCreate, 274  
    onCreateOptionsMenu, 267  
    onCreateView, 269  
    onCreateViewHolder, 216, 218, 246  
    onNewIntent, 267  
    onViewCreated, 269  
    overloading, 141  
    overriding, 80–81  
    recoverOrBuildSearchFragment, 274

set, 71  
setItems, 232–233, 248  
setters, 71, 124–125  
setUpRecyclerView, 287  
setUpSearchRecyclerView, 270  
snackbar, 312  
static, calling from Java, 140  
toString, 309

### **migration**

to Anko layouts, 332  
to Gradle Kotlin DSL, 335–336  
    benefits of, 342  
    buildSrc directory, 340–342  
    drawbacks of, 342–343  
    Gradle settings, 336  
    module build script, 338–340  
    root build script, 336–338  
to Kotlin, 7–8  
    autoconverted code, adjusting, 355–356  
    full migration, 350–351  
    Java-to-Kotlin converter, 355  
    leadership for, 346–349  
    partial migration, 349–350  
    pet projects, 353–354  
    planning, 354  
    production code, 352–353  
    risks and benefits of, 345–346  
    test code, 351–352

### **minBy function, 49–50**

### **minimums, calculating, 49–50**

### **minus sign (-)**

dec operator (--), 28  
minusAssign operator (-=), 28  
subtraction operator (-), 28

**MochK**, 353

**Mockito**, 7, 353

**models**

- mapping DTOs to, 260–262
- RecyclerView, 215

**modifiers. *See keywords***

**modularizing Anko layouts**, 333

**module build script, migrating to Gradle Kotlin DSL**

- android block, 338–339
- dependencies, 339–340
- experimental features, 340
- plugins block, 336–338

**Moshi**, 251, 346

- Gradle dependencies for, 258
- MoshiConverterFactory, 253

**multiplication (\*) operator**, 28

**multitasking**

- cooperative, 157–158
- definition of, 152–153
- preemptive, 157–158

**multithreading**, 152–153

**mutable states**, 153

**mutable variables**, 13

**(Mutable)List type**, 127

**MutableList type**, 127

**mutableListOf function**, 46

**(Mutable)Map type**, 127

**mutableMapOf function**, 46

**mutableSetOf function**, 46

**mutual exclusion**, 153

---

## N

---

**named parameters**, 23–24

**navigation, Nutrilicious app**, 242–243, 301

**NDBNO (nutrition database number),** [260](#)  
**nested classes,** [82](#)  
**nesting,** [323–325](#)  
**NetBeans,** [7](#)  
**Netflix,** [8](#)  
**NetworkDispatcher,** [256](#)  
**New Project command,** [210](#)  
**Newton, Joseph Fort,** [315](#)  
**nonblocking,** [152](#)  
**non-local returns,** [44–45](#)  
**not (!) operator,** [15](#)  
**Nothing type,** [148](#)  
**@NotNull,** [126](#)  
**nullability**  
    annotations for, [128–129](#)  
    mapped types and, [125](#)  
    null safety, [29](#)  
    nullable types  
        casting, [90](#)  
        elvis operator (?:), [30–31](#)  
        overview of, [29](#)  
        safe call operator (?), [29–30](#)  
        unsafe call operator (!!), [31, 355](#)  
    overview of, [125](#)  
    platform types, [126–128](#)  
**@Nullable,** [126](#)  
**NullPointerException,** [4, 8, 29](#)  
**Nutrient class,** [308](#)  
**NutrientListConverter,** [302–303](#)  
**NutrientTypeConverter class,** [303](#)  
**Nutrilicious app**  
    data mapping  
        DTOs to models, [260–262](#)  
        JSON to DTOs, [258–260](#)

overview of, 257–258

## DetailsActivity

click handler, 297

data display, 300–301

DetailsViewModel class, 298–300

FOOD\_ID\_EXTRA identifier, 298

item click listener, 296–297

layout, 293–296

navigation, 301

string resources for headline, 296

empty search results, indicating, 311–312

Favorites fragment, 276–280

finished app, 241

food details, storing in database, 302–306

migrating to Gradle Kotlin DSL, 335–336

benefits of, 342

buildSrc directory, 340–342

drawbacks of, 342–343

Gradle settings, 336

module build script, 338–340

root build script, 336–338

progress indicator, 312–313

project setup, 242–243

RDI (recommended daily intake), adding, 307–311

## RecyclerView

adapter, 246–248

displaying mapped data in, 261–262

Food class, 246

item layout, 244–246

MainActivity layout, 243–244

MainActivity setup, 248–250

overview of, 242–243

## Room database

AppDatabase class, 282

coroutine dispatchers, 283  
FavoritesDao interface, 281  
FavoritesViewModel class, 282–283, 286–287  
Food class, 280–281  
SearchFragment, 287–288  
SearchListAdapter class, 283–284  
toggleFavorite function, 285  
search interface  
    implementation of, 265–268  
    overview of, 265  
SearchFragment  
    adding to UI, 273  
    fragment transactions, 272  
    getViewModel extension, 270–271  
    layout for, 268–269  
    MainActivity, 269  
    methods, 269–270  
    onCreate method, 274  
    properties, 272  
    recoverOrBuildSearchFragment method, 274  
    restoring, 274  
    search intents, 275  
    setUpSearchRecyclerView function, 271  
    storing in activity’s state, 273  
    swipe refresh, 275  
    SwipeRefreshLayout, 271–272  
    updateListFor function, 271  
SearchViewModel class, 262–264  
USDA Food Reports API, fetching data from, 288–293  
USDA Nutrition API, fetching data from  
    API requests, performing, 256–257  
    Gradle dependencies for, 250–251  
    Retrofit interface, 251–256  
**nutrition database number (NDBNO), 260**

# O

---

**object expressions, 101–102**

**object keyword, 101–102, 130**

**object orientation. *See also* classes; objects**

generics

    benefits of, 105

    classes, 106–107

    functions, 107–108

    reification, 108–109

    type parameters, 105–106

inheritance

    abstract classes, 87–88

    closed-by-default principle, 85, 88

    data classes, 96

    interfaces, 85–86

    open classes, 88–89

    overriding rules for, 89

    overview of, 84–85

    overview of, 69

polymorphism

    compile-time, 108

    parametric, 108

type casting

    ClassCastException, 90

    nullable types, 90

    smart casts, 90–91

type checking, 89

type parameters, 119–121

variance

    bounded type parameters, 118–119

    contravariance, 112

    covariance, 110–112

    declaration-site, 113–116

    Java interoperability, 115, 116

star projections, 119–121  
use-site, 116–118  
visibilities  
    class declarations, 92–93  
    interface declarations, 92–93  
    top-level declarations, 93–94

**Object[] type, 127**

**objects**  
    BuildPlugins, 342  
    companion, 103–105  
    creating, 101–102  
    declaring, 103  
    DefaultDispatcher, 165  
    DTOs (data transfer objects)  
        DetailsDto, 291  
        mapping JSON data to, 258–260  
        mapping to models, 260–262  
        wrappers for, 290  
    equality  
        checking, 32  
        floating point, 32–33  
        initializing, 55  
        instantiating, 69  
    Job, 177–181  
    object expressions, 101–102  
    usdaApi, 256–257

**observable properties, 76–77**

**observeFavorites method, 287**

**OkHttp, 251, 253–255**

**onAttach method, 269**

**onBindViewHolder method, 216, 218, 246**

**onCreate method, 274**

**onCreateOptionsMenu method, 267**

**onCreateView method, 269**

**onCreateViewHolder method**, 216, 218, 246

**online resources**, 7

**onNewIntent method**, 267

**onViewCreated method**, 269

**OO. See object orientation**

**open classes**, 88–89

**open keyword**, 87

**operators**

as, 90

elvis (?,:), 30–31

indexed access, 47

is, 89

Java methods as, 131

safe call (?), 29–30

table of, 28

ternary conditional, 18

unsafe call (!!), 31, 355

**or (||) operator**, 15

**Oracle, Google’s lawsuit with**, 5

**overloading**

functions, 24–25

methods, 141

**override keyword**, 86, 104–105

**overriding**

inheritance and, 89

methods, 80–81

## P

---

**pair programming**, 348

**parallelism**, 152–153

**parameters, function**, 23–24

**parametric polymorphism**, 108

**partial migration**, 349–350

**percent sign (%)**

rem operator (%), 28  
remAssign operator (%=), 28  
**performance, DSLs (domain-specific languages)**, 327–328  
**period (.)**, 28  
**Perlis, Alan J.**345  
**pet projects, integrating Kotlin into**, 353–354  
**Pinterest**, 8  
**pipe symbol (|) operator**, 15  
**plans, migration**, 354  
**platform types**, 126–128  
**plugins block (Gradle Kotlin DSL)**, 336–338  
**plus function**, 180  
**plus sign (+)**, 28  
    addition operator (+), 28  
    inc operator (++), 28  
    plusAssign operator (+=), 28  
**pointers to functions**, 39  
**polymorphism**, 80–81  
    compile-time, 108  
    parametric, 108  
**populating databases**, 224–225  
**pragmatic programming**, 4  
**preemptive multitasking**, 157–158  
**previewing Anko layouts**, 331  
**primary constructors**, 70, 82–83  
**printAll function**, 119–121  
**private keyword**, 92, 94  
**produce coroutine builder**, 191  
**producers**, creating, 191  
**productFlavors block (Gradle Kotlin DSL)**, 339  
**production code, integrating Kotlin into**, 352–353  
**progress indicator (Nutrilicious app)**, 312–313  
**projects**  
    creating

Kudoo app, 210–212  
Nutrilicious app, 242–243  
integrating Kotlin into, 353–354  
**promises, 152**  
**properties**  
adding to classes, 70–71  
backing fields, 72  
calling from Java, 133–135  
delegated  
implementation of, 74–75  
lazy properties, 75–76  
maps, 78–79  
observable properties, 76–77  
syntax of, 74  
exposing as fields, 135–136  
fields compared to, 71  
late-initialized, 72–73  
lazy, 75–76  
observable, 76–77  
SearchFragment, 272  
**protected keyword, 92**  
**public keyword, 92, 94**  
**public scope, 26**  
**Python, 316**

---

## Q

---

**@Query annotation, 255**  
**question mark (?)**  
elvis operator (?:), 30–31  
safe call operator (?), 29–30

---

## R

---

**race conditions, 154**  
**ranges in for loops, 20–21**

**rangeTo operator (.), 28**  
**read-eval-print loop (REPL), 11**  
**read-only variables, 13**  
**receive function, 187**  
**ReceiveChannel, 188**  
**receivers, lambda expressions with, 60–61**  
**recoverOrBuildSearchFragment method, 274**  
**RecyclerListAdapter, 215–219**  
**RecyclerView**

- Kudoo app
  - adapter, 215–219
  - layouts, 213–214
  - MainActivity, 220–221
  - model, 215
  - overview of, 212–213
- Nutrilicious app
  - adapter, 246–248
  - displaying mapped data in, 261–262
  - Food class, 246
  - item layout, 244–246
  - MainActivity layout, 243–244
  - MainActivity setup, 248–250
  - overview of, 242–243

- Reddit Kotlin community, 7**
- reduce function, 52**
- reduceRight function, 52**
- referential equality operator (==), 32**
- Reformat Code command (Code menu), 258**
- Refresh All Gradle Projects command, 336**
- reification, 108–109**
- reified keyword, 109**
- rem (%) operator, 28**
- renderNutrient function, 301, 310**
- rendezvous channels, 190**

**REPL (read-eval-print-loop), 11**

**requests (API), performing, 256–257**

**resolving extensions, 25–26**

**REST, 151**

**restoring fragments, 274**

**@RestrictsSuspension annotation, 185**

**Retrofit, 7, 251–256**

**Retrolambda, 348**

**returns, non-local, 44–45**

**Room database**

- Kudoo app
  - AppDatabase class, 223–225
  - Gradle dependencies for, 221–222, 225–226
  - LiveData class, 230–233
  - MainActivity, 226–227
  - TodoItem class, 222
  - TodoItemDao class, 222–223
  - ViewModel class, 227–230
- Nutrilicious app
  - AppDatabase class, 282
  - coroutine dispatchers, 283
  - FavoritesDao interface, 281
  - FavoritesViewModel class, 282–283, 286–287
  - Food class, 280–281
  - SearchFragment, 287–288
  - SearchListAdapter class, 283–284
  - storing food details in, 302–306
  - toggleFavorite function, 285

**root build scripts, migrating to Gradle Kotlin DSL**

- allprojects block, 337
- buildscript block, 336–337
- delete task, 337–338

**run function, 56–58**

**runblocking coroutine builder, 163–164**

## S

---

- safe call operator (?), 29–30
- safety, 4
- SAM (single abstract method) conversions, 131–132
- Sample Data Directory, 258
- Scala, 4
  - scope of extension functions, 26
  - scoping functions
    - also, 58–59
    - apply, 54–55
    - let, 53–54
    - run, 56–58
    - use, 59
    - with, 55–56
  - Scrum, 354
  - sealed classes
    - calling from Java, 142–143
    - declaring, 98–100
  - sealed keyword, 98–100
  - Search API, 255–256, 258–260
  - search interface, Nutrilicious app
    - implementation of, 265–268
    - overview of, 265
  - searchable configuration, 266
  - SearchFragment
    - adding to UI, 273
    - database connection, 287–288
    - fragment transactions, 272
    - getViewModel extension, 270–271
    - layout for, 268–269
    - MainActivity, 269
    - methods, 269–270

onCreate method, 274  
properties, 272  
recoverOrBuildSearchFragment method, 274  
restoring, 274  
search intents, 275  
setUpSearchRecyclerView function, 271  
storing in activity’s state, 273  
swipe refresh, 275  
SwipeRefreshLayout, 271–272  
updateListFor function, 271  
**SearchListAdapter class**, 246–248, 283–284  
**SearchViewModel class**, 262–264  
**secondary constructors**, 84  
**send function**, 187  
SendChannel, 187–188  
**sendEmail function**, 24  
**Sequences**, 6  
**sequences, lazy**, 63–66  
    creating, 63–64  
    drop function, 64–65  
    performance of, 65–66  
    take function, 64–65  
**@SerializedName annotation**, 261  
**set method**, 71  
**set operator**, 28  
**setItems method**, 232–233, 248  
**setLoading function**, 313  
**setOf function**, 46  
**setters**  
    calling, 124–125  
    declaring, 71  
**settings (Gradle), migrating**, 336  
**setUpRecyclerView method**, 287  
**setUpSearchRecyclerView function**, 271

**setUpSearchRecyclerView** method, 270  
sharing knowledge, 348–349  
Short data type, 13  
short-circuiting, 15  
Show Kotlin Bytecode command, 217–218  
signatures, function, 21–22, 39, 144–145, 159  
Simula, 69  
single abstract method (SAM) types, 131–132  
single-expression functions, 22  
singletons, declaring, 103  
singleTop launch mode, 266  
Slack, 8  
Slack channel, 7  
Smalltalk, 69  
smart casts, 30, 90–91  
snackbar method, 312  
sorted function, 50  
sortedArray function, 50  
sortedBy function, 50  
sortedDescending function, 50  
sortedMapOf function, 46  
sortedSetOf function, 46  
sortedWith function, 50  
sorting collections, 50–51  
SoundCloud, 354  
Spek, 353  
Split Vertically command, 258  
Spring, 124  
SQL, 316  
Square, 348  
star projections, 119–121  
statements. *See* keywords  
static fields, calling from Java, 138–139  
static keyword, 104

**static methods, calling from Java**, 140  
**static resolving of extensions**, 25–26  
**Strategy pattern**, 39  
**streams**, 6, 62. *See also* sequences, lazy  
**String data type**, 13, 127  
**string interpolation**, 20  
**String! type**, 127  
**structural equality operator (==)**, 32  
**subtraction (-) operator**, 28  
**sumBy function**, 49–50  
**sums, calculating**, 49–50  
**super keyword**, 87  
**support and community**, 7  
**suspend keyword**, 159–163, 198–200  
**suspendCoroutine function**, 183  
**suspending functions**, 157, 159–163, 164, 198–200  
**suspending lambda**, 164  
**suspension points**, 160  
**Swing**, 182  
**swipe refresh**, 275  
**SwipeRefreshLayout**, 271–272

## T

---

**take function**, 64–65  
**ternary conditional operator**, 18  
**test code, integrating Kotlin into**, 351–352  
**testing frameworks**, 353  
**this keyword**, 61  
**threads**, 152, 157–158. *See also* coroutines  
**throw keyword**, 33  
**throwing exceptions**, 33–35  
**@Throws annotation**, 145–146, 149  
**TimeoutCancellationException**, 177  
**timeouts, handling**, 177

**toast function**, 330

**TodoItem class**, 222

**TodoItemDao class**, 222–223

**toggleFavorite function**, 285

**top-level declarations**, 93–94

**top-level extensions**, calling, 137

**toSortedMap function**, 50

**toString method**, 309

**transactions, fragment**, 272

**Trello**, 8

**try keyword**, 33–35

**try-catch blocks**, 33–35

**Twain, Mark**, 3

**type converters (Nutrilicious app)**, 302–303

**type inference**, 14–15, 40–41

**@TypeConverter annotation**, 303

**@TypeConverters annotation**, 303

**types**. *See also* **nullability**

- algebraic, 100
- aliases, 107
- Any, 88
- casting
  - ClassCastException, 90
  - nullable types, 90
  - smart casts, 30, 90–91
- checking, 89
- classes versus, 110
- function types, 39
- generic type parameters, 105–106
- mapped, 125
- mapping, 14
- Nothing, 148
- nullable
  - casting, 90

elvis operator (?:), 30–31  
overview of, 29  
safe call operator (?), 29–30  
unsafe call operator (!!), 31, 355  
parameters, 40–41, 105–106  
platform types, 126–128  
table of, 13–14  
type inference, 14–15  
variance  
    bounded type parameters, 118–119  
    contravariance, 112  
    covariance, 110–112  
    declaration-site, 113–116  
    star projections, 119–121  
    use-site, 116–118  
**type-safe builders**, 320. *See also* **DSLs (domain-specific languages)**

## U

---

**Uber**, 8  
**Udacity**, 8, 354  
**unchecked exceptions**, 35–36  
**unlimited channels**, 189  
**unsafe call operator** (!!), 31, 355  
**updateListFor function**, 271, 275, 311  
**updateUiWith function**, 313  
**USDA Food Reports API**, fetching data from, 288–293  
**USDA Nutrition API**, fetching data from  
    API requests, performing, 256–257  
    Gradle dependencies for, 250–251  
    Retrofit interface, 251–256  
**usdaApi object**, 256–257  
**use function**, 59  
**User class**

adding addresses to  
multiple addresses, 323–325  
single addresses, 319–320  
declaring, 319

**user function, 318**

**user groups, 348**

**user objects, building with DSLs**  
complex objects, 318–320  
simple objects, 318

**UserBuilder class, 321**

**@UserDsl annotation, 325**

**use-site variance, 116–118, 146–148**

## V

---

**val keyword, 12, 71, 83, 130**

**validation, 58**

**values**  
floating point, 32–33  
for function parameters, 23–24

**var keyword, 12–13, 71, 83, 130**

**vararg methods, calling, 130–131**

**variable-argument methods, calling, 130–131**

**variables**  
declaring, 12–13  
in DSLs (domain-specific languages), 326  
immutable, 13  
mutable, 13  
read-only, 13

**variance**  
bounded type parameters, 118–119  
contravariance, 112  
covariance, 110–112  
declaration-site, 113–116  
Java interoperability, 115, 116

star projections, 119–121  
use-site, 116–118, 146–148

### **variant types, 146–148**

**verticalLayout function, 330**

**ViewHolder class, 216–217**

### **ViewModel class**

Kudoo app, 227–230  
Nutrilicious app, 262–264

### **views**

Anko layouts, 334–335

Kudoo RecyclerView

adapter, 215–219

layouts, 213–214

MainActivity, 220–221

model, 215

overview of, 212–213

Nutrilicious RecyclerView

adapter, 246–248

displaying mapped data in, 261–262

Food class, 246

item layout, 244–246

MainActivity layout, 243–244

MainActivity setup, 248–250

overview of, 242–243

### **visibilities**

class declarations, 92–93

interface declarations, 92–93

mapping to Java

exception handling, 145–146

inline functions, 145

KClass, 143–144

Nothing type, 148

overview of, 143

signature clashes, 144–145

variant types, [146–148](#)  
top-level declarations, [93–94](#)

## W

---

**WeightUnit enum, [309](#)**  
**Wharton, Jake, [348](#)**  
**when keyword, [15–19](#), [99–100](#)**  
**where keyword, [118–119](#)**  
**while loops, [19](#)**  
**Wigmore, Ann, [241](#)**  
**wildcard generation, [146–148](#)**  
**with function, [55–56](#)**  
**withContext function, [176](#)**  
**withTimeout function, [176](#)**  
**WordPress, [8](#)**  
**wrappers**  
    for asynchronous APIs, [195–197](#)  
    for DTOs (data transfer objects), [290](#)

## X-Y-Z

---

**XML layouts, [335](#)**  
**yield function, [184](#)**  
**Ziglar, Zig, [11](#)**

## Credits

Cover: Kronstadt at Sunrise, A.F. Smith/Shutterstock.

Foreword: “The limits of ... of my world,” Ludwig Wittgenstein.

Chapter 1: “The secret to getting ahead is getting started.” Mark Twain.

Chapter 2: “When you catch ... when passion is born.” Zig Ziglar.

Chapter 3: “Divide each difficulty ... necessary to resolve it,” René Descartes (1596–1650). Discourse on the Method of Rightly Conducting the Reason and Seeking Truth in the Sciences.

Chapter 4: “Complexity has nothing to do with intelligence, simplicity does,” Larry Bossidy.

Chapter 4: “Generic programming centers ... of useful software,” David R. Messer and Alexander A. Stepanov, “Generic programming.” Presented at the First International Joint Conference of International Symposium on Symbolic and Algebraic Computation (ISSAC)-88 and Applied Algebra, Algebraic Algorithms, and Error Correcting Codes (AAECC)-6, Rome Italy, July 4-8, 1988. Lecture Notes in Computer Science, P. Gianni. 358. Springer-Verlag, 1989, pp. 13–25. (<http://stepanovpapers.com/genprog.pdf>)

Chapter 5: “Synergy is better than my way or your way. It’s our way.” Stephen Covey.

Chapter 6: “Juggling is an illusion ... actually task switching,” Gary Keller.

Chapter 7: “Whether you think you can, or you think you can’t —you’re right,” Henry Ford.

Figure 7.1: Screenshot of Android Studio © Google LLC.

Figure 7.2: Screenshot of Kotlin Android © Google LLC.

Figure 7.3: Screenshot of Android Studio © Google LLC.

Figure 7.4: Screenshot of Android Studio © Google LLC.

Chapter 8: “The food you ... slowest form of poison,” Ann Wigmore.

Figure 8.1: Screenshot of Kotlin Android © Google LLC.

Figure 8.2: Screenshot of Kotlin Android © Google LLC.

Chapter 9: “Men build too many walls and not enough bridges,” Joseph Fort Newton.

Chapter 10: “A language that ... not worth knowing,” Alan J. Perlis.



## VIDEO TRAINING FOR THE **IT PROFESSIONAL**



### LEARN QUICKLY

Learn a new technology in just hours. Video training can teach more in less time, and material is generally easier to absorb and remember.



### WATCH AND LEARN

Instructors demonstrate concepts so you see technology in action.



### TEST YOURSELF



Our Complete Video Courses offer self-assessment quizzes throughout.



### CONVENIENT

Most videos are streaming with an option to download lessons for offline viewing.

Learn more, browse our store, and watch free, sample lessons at  
[informit.com/video](http://informit.com/video)

---

**Save 50%\*** off the list price of video courses with discount code **VIDBOB**



\*Discount code VIDBOB confers a 50% discount off the list price of eligible titles purchased on Informit.com. Eligible titles include most full-course video titles, Book + eBook bundles, book/eBook + video bundles, individual video lessons, Rough Cuts, Safari Books Online, non-discountable titles, titles on promotion with our retail partners, and any title featured as eBook Deal of the Day or Video Deal of the Week is not eligible for discount. Discount may not be combined with any other offer and is not redeemable for cash. Offer subject to change.



Photo by iusek/gettyimages

## Register Your Product at [informit.com/register](http://informit.com/register)

Access additional benefits and **save 35%** on your next purchase

- Automatically receive a coupon for 35% off your next purchase, valid for 30 days. Look for your code in your InformIT cart or the Manage Codes section of your account page.
- Download available product updates.
- Access bonus material if available.\*
- Check the box to hear from us and receive exclusive offers on new editions and related products.

\*Registration benefits vary by product. Benefits will be listed on your account page under Registered Products

## InformIT.com—The Trusted Technology Learning Source

InformIT is the online home of information technology brands at Pearson, the world's foremost education company. At InformIT.com, you can:

- Shop our books, eBooks, software, and video training
- Take advantage of our special offers and promotions ([informit.com/promotions](http://informit.com/promotions))
- Sign up for special offers and content newsletter ([informit.com/newsletters](http://informit.com/newsletters))
- Access thousands of free chapters and video lessons

Connect with InformIT—Visit [informit.com/community](http://informit.com/community)



Addison-Wesley • Adobe Press • Cisco Press • Microsoft Press • Pearson IT Certification • Prentice Hall • Que • Sams • Peachpit Press



## Code Snippets

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the eBook in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a “Click here to view code image” link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

---

```
var mercury: String = "Mercury" // Declares a mutable String variable
mercury = "Venus"           // Can be reassigned because mutable
```

---

---

```
val mercury: String = "Mercury" // Declares a read-only variable
mercury = "Venus"           // Compile-time error: "val cannot be reassigned"
```

---

---

```
val mercury = "Mercury"      // Inferred type: String
val maxSurfaceTempInK = 700   // Inferred type: Int
val radiusInKm = 2439.7      // Inferred type: Double
```

---

---

```
if (mercury == "Mercury") {
    println("Universe is intact.")
} else if (mercury == "mercury") {
    println("Still all right.")
} else {
    println("Universe out of order.")
}
```

---

---

```
when (mercury) {
    "Mercury" -> println("Universe is intact.")
    "mercury" -> println("Still all right.")
    else -> println("Universe out of order.")
}
```

---

---

```
when(maxSurfaceTempInK) {
    700          -> println("This is Mercury's maximum surface temperature") // 1
    0, 1, 2      -> println("It's as cold as it gets")                      // 2
    in 300..699   -> println("This temperature is also possible on Mercury") // 3
    !in 0..300    -> println("This is pretty hot")                         // 4
    earthSurfaceTemp() -> println("This is earth's average surface temperature") // 5
    is Int        -> println("Given variable is of type Int")             // 6
    else          -> {
        // You can also use blocks of code on the right-hand-side, like here
        println("Default case")
    }
}
```

---

---

```
when {
    age < 18 && hasAccess -> println("False positive")
    age > 21 && !hasAccess -> println("False negative")
    else -> println("All working as expected")
}
```

---

---

```
val status = if (mercury == "Mercury") { // 'if' is an expression (it has a value)
    "Universe is intact" // Expression value in case 'if' block is run
} else {
    "Universe out of order" // Expression value in case 'else' block is run
}
```

---

---

```
val status = if (mercury == "Mercury") "Intact" else "Out of order"
```

---

---

```
val temperatureDescription = when(maxSurfaceTempInK) {
    700 -> "This is Mercury's maximum surface temperature"
    // ...
    else -> {
        // More code...
        "Default case" // Expression value if 'else' case is reached
    }
}
```

---

---

```
val number = 42
var approxSqrt = 1.0
var error = 1.0

while (error > 0.0001) { // Repeats code block until error is below threshold
    approxSqrt = 0.5 * (approxSqrt + number / approxSqrt)
    error = Math.abs((number - approxSqrt*approxSqrt) / (2*approxSqrt))
}
```

---

---

```
do {
    val command = readLine()    // Reads user command from console
    // Handle command...
} while (command != ":quit") // Repeats code block until user enters ":quit"
```

---

---

```
for (i in 1..100) println(i)      // Iterates over a range from 1 to 100
```

```
for (i in 1 until 100) println(i) // Iterates over a range from 1 to 99
```

```
for (planet in planets)          // Iterates over a collection
    println(planet)
```

```
for (character in "Mercury") {    // Iterates over a String character by character
    println("$character, ")
}
```

---

---

```
for (i in 100 downTo 1) println(i)      // 100, 99, 98, ..., 3, 2, 1
for (i in 1..10 step 2) println(i)      // 1, 3, 5, 7, 9
for (i in 100 downTo 1 step 5) println(i) // 100, 95, 90, ..., 15, 10, 5
```

---

---

```
fun fib(n: Int): Long {
    return if (n < 2) {
        1
    } else {
        fib(n-1) + fib(n-2) // Calls 'fib' recursively
    }
}
```

---

---

```
val fib = fib(7) // Assigns value 21L ('L' indicates Long)
```

---

---

```
fun fib(n: Int): Long {  
    return if (n < 2) 1 else fib(n-1) + fib(n-2) // Uses 'if' as a ternary operator  
}
```

---

---

```
fun fib(n: Int): Long = if (n < 2) 1 else fib(n-1) + fib(n-2)
```

---

---

```
fun main(args: Array<String>) { // A main function must have this signature
    println("Hello World!")
}
```

---

---

```
fun exploreDirectory(path: String, depth: Int = 0) { ... } // Default depth is zero
```

---

---

```
val directory = "/home/johndoe/pictures"

exploreDirectory(directory)           // Without optional argument, depth is 0
exploreDirectory(directory, 1)        // Recursion depth set to 1
exploreDirectory(directory, depth=2)  // Uses named parameter to set depth
exploreDirectory(depth=3, path=directory) // Uses named parameters to change order
```

---

---

```
fun sendEmail(message: String) { ... }
```

```
fun sendEmail(message: HTML) { ... }
```

---

```
// Java code
Pizza makePizza(List<String> toppings) {
    return new Pizza(toppings);
}
Pizza makePizza() {
    return makePizza(Arrays.asList());
}
```

```
fun makePizza(toppings: List<String> = emptyList()) = Pizza(toppings)
```

---

```
fun Date.plusDays(n: Int) = Date(this.time + n * 86400000) // Adds n days to date
```

---

```
val now = Date()
```

```
val tomorrow = now.plusDays(1) // Extension can be called directly on a Date object
```

---



---

```
import com.example.time.plusDays
```

---

---

```
infix fun <A, B> A.to(that: B) = Pair(this, that) // Declared in standard library
```

---

---

```
val pair = "Kotlin" to "Android"           // Pair("Kotlin", "Android")
val userToScore = mapOf("Peter" to 0.82, "John" to 0.97) // Creates a map
```

---

---

```
infix fun Int.times(str: String) = str.repeat(this)
val message = 3 times "Kotlin" // Results in "Kotlin Kotlin Kotlin"
```

---

---

```
operator fun Int.times(str: String) = str.repeat(this)
val message = 3 * "Kotlin" // Still results in "Kotlin Kotlin Kotlin"
```

---

---

```
val name: String? = "John Doe" // Nullable variable of type 'String?'
```

```
val title: String? = null // Nullable variables may hold 'null'
```

---

```
val wrong: String = null // Causes compiler error because not nullable
```

---

---

```
val name: String? = "John Doe"
println(name.length) // Not safe => causes compile-time error

if (name != null) { // Explicit null check
    println(name.length) // Now safe so compiler allows accessing length
}

println(name?.length) // Safe call operator '?'
val upper = name?.toUpperCase() // Safe call operator
```

---

---

```
val name: String? = null
val len = name?.length ?: 0 // Assigns length, or else zero as the default value
```

---

---

```
val name: String? = "John Doe" // May also be null
val len = name!!.length // Asserts the compiler that name is not null; unsafe access
```

---

---

```
fun greet(name: String) = println("Hi $name!")

greet(name!!)           // Better be sure that 'name' cannot be null here
greet(name ?: "Anonymous") // Safe alternative using elvis operator
```

---

---

```
val list1 = listOf(1, 2, 3) // List object
val list2 = listOf(1, 2, 3) // Different list object but with same contents
```

```
list1 === list2 // false: the variables reference different objects in memory
list1 == list2 // true: the objects they reference are equal
list1 != list2 // true: negation of first comparison
list1 != list2 // false: negation of second comparison
```

---

---

```
val negativeZero = -0.0f           // Statically inferred type: Float
val positiveZero = 0.0f             // Statically inferred type: Float
Float.NaN == Float.NaN            // false (IEEE standard)
negativeZero == positiveZero      // true (IEEE standard)
val nan: Any = Float.NaN          // Explicit type: Any => not Float or Double
val negativeZeroAny: Any = -0.0f   // Explicit type: Any
val positiveZeroAny: Any = 0.0f    // Explicit type: Any

nan == nan                        // true (not IEEE standard)
negativeZeroAny == positiveZeroAny // false (not IEEE standard)
```

---

---

```
fun reducePressureBy(bar: Int) {
    // ...
    throw IllegalArgumentException("Invalid pressure reduction by $bar")
}

try {
    reducePressureBy(30)
} catch(e: IllegalArgumentException) {
    // Handle exception here
} catch(e: IllegalStateException) {
    // You can handle multiple possible exceptions
} finally {
    // This code will always execute at the end of the block
}
```

---

---

```
val input: Int? = try {
    inputString.toInt() // Tries to parse input to an integer
} catch(e: NumberFormatException) {
    null // Returns null if input could not be parsed to integer
}
```

---

---

```
val bitmap = card.bitmap ?: throw IllegalArgumentException("Bitmap required")
bitmap.prepareToDraw() // Known to have type Bitmap here (non-nullable)
```

---

---

```
fun fail(message: String): Nothing { // Nothing type => function never terminates
    throw IllegalArgumentException(message)
}

val bitmap = card.bitmap ?: fail("Bitmap required") // Inferred type: Bitmap
bitmap.prepareToDraw()
```

---

---

```
fun countOccurrences(of: Char, inStr: String): Int // Inputs: Char, String; Out: Int
```

---

---

```
val count: (Char, String) -> Int = ::countOccurrences // Function ref. matches type
```

---

---

```
{ x: Int, y: Int -> x + y } // Lambdas are denoted with bold braces in this chapter
```

---

---

---

**val** sum: (Int, Int) -> Int = { x: Int, y: Int -> x + y }

---

---

---

---

```
val sum: (Int, Int) -> Int = { x, y -> x + y } // Infers the lambda parameter types
```

---

---

```
val sum = { x: Int, y: Int -> x + y } // Infers the variable type
```

---

---

```
val toUpper: (String) -> String = { it.toUpperCase() } // Uses the implicit 'it'
```

---

---

```
val toUpper: (String) -> String = String::toUpperCase // Uses function reference
```

---

---

```
fun twice(f: (Int) -> Int): (Int) -> Int = { x -> f(f(x)) } // Applies 'f' twice
```

---

---

```
val plusTwo = twice({ it + 1 }) // Uses lambda
val plusTwo = twice(Int::inc)   // Uses function reference
```

---

---

```
val plusTwo = twice { it + 1 } // Moves lambda out of parentheses and omits them
```

---

---

```
val plusTwo = twice { it + 1 } // Creates a function object
```

---

---

```
public inline fun <T> Iterable<T>.forEach(action: (T) -> Unit): Unit { // Uses inline
    for (element in this) action(element)
}
```

---

```
(1..10).forEach { println(it) } // This function call is inlined at compile-time
```

---

---

```
for (element in 1..10) println(element) // No lambda, thus no object at runtime
```

---

---

```
fun contains(range: IntRange, number: Int): Boolean {
    range.forEach {
        if (it == number) return true // Can use return because lambda will be inlined
    }
    return false
}
```

---

---

```
fun contains(range: IntRange, number: Int): Boolean {
    for (element in range) {
        if (element == number) return true // This return is equivalent to the one above
    }
    return false
}
```

---

---

```
inline fun twice(crossinline f: (Int) -> Int): (Int) -> Int = { x -> f(f(x)) }
```

---

---

```
// Collections

val languages = setOf("Kotlin", "Java", "C++") // Creates a read-only set
val votes = listOf(true, false, false, true) // Creates a read-only list
val countryToCapital = mapOf(                // Creates a read-only map
    "Germany" to "Berlin", "France" to "Paris")
```

---

```
// Arrays

val testData = arrayOf(0, 1, 5, 9, 10) // Creates an array (mutable)
```

---

---

```
votes[1]    // false
 testData[2] // 5
countryToCapital["Germany"] // "Berlin"
```

---

---

```
// Assumes mutable collections now
votes[1] = true           // Replaces second element with true
testData[2] = 4            // Replaces third element with 4
countryToCapital["Germany"] = "Bonn" // Replaces value for key "Germany"
```

---

---

```
val numberOfYesVotes = votes.filter { it == true }.count() // Counts positive votes
```

---

---

```
val searchResult = countryToCapital.keys.filter { it.toUpperCase().startsWith('F') }
```

---

---

```
val squared = testData.map { it * it } // Creates list with each element squared
```

---

---

```
inline fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R> // Returns List
```

---

---

```
val countries = countryToCapital.map { it.key } // ["Germany", "France"]
```

---

---

```
val sentence = "This is an example sentence with several words in it"  
val lengthToWords = sentence.split(" ") // ["This", "is", "an", .., "in", "it"]  
    .groupBy { it.length }  
  
println(lengthToWords) // {4=[This, with], 2=[is, an, in, it], .. }
```

---

---

```
val words = listOf("filter", "map", "sorted", "groupBy", "associate")
var id = 0 // The first map key
val map = words.associate { id++ to it } // Associates incrementing keys to elements
println(map) // {0=filter, 1=map, 2=sorted, 3=groupBy, 4=associate}
```

---

---

```
users.minBy { it.age }          // Returns user object of youngest user
users.maxBy { it.score }         // Returns user object of user with maximum score
users.sumBy { it.monthlyFee }    // Returns sum of all users' monthly fees
```

---

---

```
languages.sorted()           // ["C++", "Java", "Kotlin"]
languages.sortedDescending() // ["Kotlin", "Java", "C++"]
testData.sortedBy { it % 3 }  // [0, 9, 1, 10, 5]
votes.sortedWith(
    Comparator { o1, _ -> if (o1 == true) -1 else 1 }) // [true, true, false, false]
countryToCapital.toSortedMap() // {France=Paris, Germany=Berlin}
```

---

---

```
val testData = arrayOf(0, 1, 5, 9, 10)
val sum = testData.fold(0) { acc, element -> acc + element } // 25
val product = testData.fold(1) { acc, element -> acc * element } // 0
```

---

---

```
val activeUserNames = users.filter { it.isActive } // Filter active users
    .take(10)           // Take first ten users
    .map { it.username } // Extract their usernames
```

---

---

```
val lines = File("rawdata.csv").bufferedReader().let { // Scopes the buffered reader
    val result = it.readLines() // Accesses reader as 'it'
    it.close()
    result
}
// Buffered reader and 'result' variable are not visible here
```

---

---

```
val weather: Weather? = fetchWeatherOrNull()

weather?.let {
    updateUi(weather.temperature) // Only updates UI if 'weather' is not null
}
```

---

---

```
countryToCapital.apply {
    putIfAbsent("India", "Delhi") // Accesses 'countryToCapital' methods without prefix
    putIfAbsent("France", "Paris")
}

println(countryToCapital) // {"Germany"="Berlin", "France"="Paris", "India"="Delhi"}
```

---

---

```
val container = Container().apply { // Initializes object inside 'apply'  
    size = Dimension(1024, 800)  
    font = Font.decode("Arial-bold-22")  
    isVisible = true  
}
```

---

---

```
countryToCapital.apply { ... } // 'apply' returns modified 'countryToCapital'  
    .filter { ... } // 'filter' works with result of 'apply'  
    .map { ... } // 'map' works with result of 'filter'
```

---

---

```
val countryToCapital = mutableMapOf("Germany" to "Berlin")

val countries = with(countryToCapital) {
    putIfAbsent("England", "London")
    putIfAbsent("Spain", "Madrid")
    keys // Defines return value of the lambda, and therefore of 'with'
}

println(countryToCapital) // {Germany=Berlin, England=London, Spain=Madrid}
println(countries)      // [Germany, England, Spain]
```

---

---

```
val essay = with(StringBuilder()) { // String builder only accessible inside lambda
    appendln("Intro")
    appendln("Content")
    appendln("Conclusion")
    toString()
}
```

---

---

```
val countryToCapital: MutableMap<String, String>?  
    = mutableMapOf("Germany" to "Berlin")  
  
val countries = countryToCapital?.run { // Runs block only if not null  
    putIfAbsent("Mexico", "Mexico City")  
    putIfAbsent("Germany", "Berlin")  
    keys  
}  
  
println(countryToCapital) // {Germany=Berlin, Mexico=Mexico City}  
println(countries) // [Germany, Mexico]
```

---

---

```
run {
  println("Running lambda")
  val a = 11 * 13
}
```

---

---

```
val success = run {
    val username = getUsername() // Only visible inside this lambda
    val password = getPassword() // Only visible inside this lambda
    validate(username, password)
}
```

---

---

```
fun renderUsername(user: User) = user.run {    // Turns parameter into receiver
    val premium = if (paid) " (Premium)" else "" // Accesses user.paid
    val displayName = "$username$premium"        // Accesses user.username
    println(displayName)
}
```

---

---

```
val user = fetchUser().also {  
    requireNotNull(it)  
    require(it!!.monthlyFee > 0)  
}
```

---

---

```
users.filter { it.age > 21 }
    .also { println("${it.size} adult users found.") } // Intercepts chain
    .map { it.monthlyFee }
```

---

---

```
val lines = File("rawdata.csv").bufferedReader().use { it.readLines() }
```

---

---

```
val sql = SqlQuery().apply { // 'apply' initializes object
    append("INSERT INTO user (username, age, paid) VALUES (?, ?, ?)")
    bind("johndoe")
    bind(42)
    bind(true)
}.also {
    println("Initialized SQL query: $it") // 'also' intercepts computation chain
}.run {
    DbConnection().execute(this) // 'run' applies given operations
}
```

---

---

```
val authors = authorsToBooks.apply {
    putIfAbsent("Martin Fowler", listOf("Patterns of Enterprise Application Arch"))
}.filter {
    it.value.isNotEmpty()
}.also {
    println("Authors with books: ${it.keys}")
}.map {
    it.key
}
```

---

---

```
fun <T, R> T.let(block: (T) -> R): R = block(this) // Lambda with parameter (T) -> R
fun <T, R> T.run(block: T.() -> R): R = block() // Lambda with receiver T.() -> R
```

---

---

```
// 'animals' stores the strings "Dog", "Cat", "Chicken", "Frog"
// 'animals' may be an eager collection or a lazy sequence but usage is the same:
animals.filter { it.startsWith("C") }
    .map { "$it starts with a 'C'" }
    .take(1)
```

---

---

```
val sequence = sequenceOf(-5, 0, 5)
println(sequence.joinToString()) // -5, 0, 5
```

---

---

```
val output = cities.asSequence() // Transforms eager list into a lazy sequence
    .filter { it.startsWith("W") }
    .map { "City: $it" }
    .joinToString()
```

---

```
println(output) // City: Warsaw, City: Washington, City: ..
```

---

---

```
val naturalNumbers = generateSequence(0) { it + 1 } // Next element = previous + 1
val integers = generateSequence(0) { if (it > 0) -it else -it + 1 }
```

---

---

```
val cities = listOf("Washington", "Houston", "Seattle", "Worcester", "San Francisco")
val firstTwo = cities.take(2) // Washington, Houston
val rest = cities.drop(2)    // Seattle, Worcester, San Francisco
firstTwo + rest == cities  // true
```

---

---

```
// Not good
cities.filter { it.startsWith("W") }
    .map { "City: $it" } // Calls map before take (could be a million map calls)
    .take(20)           // Takes only the first 20 results
    .joinToString()

// Better
cities.filter { it.startsWith("W") }
    .take(20)           // Reduces the size of the collection earlier
    .map { "City: $it" } // Calls map at most 20 times
    .joinToString()
```

---

---

```
val cities = listOf("Washington", "Houston", "Seattle", "Worcester", "San Francisco")

cities.filter { println("filter: $it"); it.startsWith("W") } // Washington, Worcester
    .map { println("map: $it"); "City: $it" } // City: Washington, City: Worcester
    .take(2) // Should better be called before 'map'
```

---

---

```
filter: Washington
filter: Houston
filter: Seattle
filter: Worcester
filter: San Francisco // Eagerly filtered all elements
map: Washington
map: Worcester
```

---

---

```
val cities = listOf("Washington", "Houston", "Seattle", "Worcester", "San Francisco")

cities.asSequence()
    .filter { println("filter: $it"); it.startsWith("W") }
    .map { println("map: $it"); "City: $it" }
    .take(2) // Should still better be called before map
    .toList()
```

---

---

```
filter: Washington
map: Washington // Passes element on to next step immediately
filter: Houston
filter: Seattle
filter: Worcester
map: Worcester // Two elements found, "San Francisco" not processed anymore (lazy)
```

---

---

```
class Task {  
    // Implement class here...  
}  
  
val laundry = Task() // Instantiates object of type Task
```

---

---

```
class Task {  
    val title: String           // Declares a property  
  
    constructor(title: String) { // Defines a constructor with one parameter  
        this.title = title      // Initializes the property  
    }  
}  
  
val dishes = Task("Wash dishes") // Calls constructor to create Task object  
val laundry = Task("Do laundry") // Calls constructor with a different title
```

---

---

```
class Task(title: String) { // Primary constructor with one parameter
    val title: String = title // Initializes property using constructor argument
}
```

---

```
val laundry = Task("Do laundry") // Calls constructor as before
```

---

---

```
class Task(val title: String) // Primary constructor declares property directly
```

---

```
val laundry = Task("Do laundry")
```

---

---

```
class Task(var title: String) // Uses 'var' now to be able to call the setter
```

```
val laundry = Task("Do laundry")
laundry.title = "Laundry day" // Calls setter
println(laundry.title) // Calls getter
```

---

---

```
class Task(title: String, priority: Int) {  
    val title: String = title  
    get() = field.toUpperCase() // Defines custom getter  
  
    var priority: Int = priority  
    get() = field // Same as default implementation, no need to define  
    set(value) { // Defines custom setter  
        if (value in 0..100) field = value else throw IllegalArgumentException("")  
    }  
}  
  
val laundry = Task("Do laundry", 40)  
println(laundry.title) // Calls getter, prints "DO LAUNDRY"  
laundry.priority = 150 // Calls setter, throws IllegalArgumentException  
println(laundry.priority) // Calls getter, would print 40
```

---

---

```
class CarTest {  
    lateinit var car: Car    // No initialization required here; must use 'var'  
  
    @BeforeEach  
    fun setup() {  
        car = TestFactory.car() // Re-initializes property before each test case  
    }  
    // ...  
}
```

---

---

```
class Cat {  
    var name: String by MyDelegate() // Delegates to object of class MyDelegate  
}
```

---

---

```
import kotlin.properties.ReadWriteProperty
import kotlin.reflect.KProperty

class MyDelegate : ReadWriteProperty<Cat, String> { // Implements ReadWriteProperty
    var name: String = "Felix"

    // Delegate must have a getValue method (and setValue for mutable properties)
    override operator fun getValue(thisRef: Cat, prop: KProperty<*>): String {
        println("$thisRef requested ${prop.name} from MyDelegate")
        return name
    }

    override operator fun setValue(thisRef: Cat, prop: KProperty<*>, value: String) {
        println("$thisRef wants to set ${prop.name} to $value via MyDelegate")
        if (value.isNotBlank()) {
            this.name = value
        }
    }
}

val felix = Cat()
println(felix.name) // Prints "Cat@1c655221 requested name from MyDelegate\n Felix"
felix.name = "Feli" // Prints "Cat@1c655221 wants to set name to Feli via MyDelegate"
println(felix.name) // Prints "Cat@1c655221 requested name from MyDelegate\n Feli"
```

---

---

```
import java.time.LocalDate
import java.time.temporal.ChronoUnit

class Cat(val birthday: LocalDate) {
    val age: Int by lazy { // Lazy property, computed on demand only if necessary
        println("Computing age...")
        ChronoUnit.YEARS.between(birthday, LocalDate.now()).toInt() // Computes age
    }
}

val felix = Cat(LocalDate.of(2013, 10, 27))
println("age = ${felix.age}") // Prints "Computing age...\\n age = 5" (run in 2018)
println("age = ${felix.age}") // Prints "age = 5"; returns cached value
```

---

```
val textView: TextView by lazy { findViewById<TextView>(R.id.title) }
```

```
fun <T: View> Activity.bind(resourceId: Int) = lazy { findViewById<T>(resourceId) }
val textView: TextView by bind(R.id.title)
```

---

```
import java.time.LocalDate
import kotlin.properties.Delegates

class Cat(private val birthday: LocalDate) {
    // ...
    var mood: Mood by Delegates.observable(Mood.GRUMPY) { // Observable property
        property, oldValue, newValue -> // Lambda parameters with old and new value
            println("${property.name} change: $oldValue -> $newValue")
            when (newValue) {
                Mood.HUNGRY -> println("Time to feed the cat...")
                Mood.SLEEPY -> println("Time to rest...")
                Mood.GRUMPY -> println("All as always")
            }
    }
}

enum class Mood { GRUMPY, HUNGRY, SLEEPY } // Enums are explained later

val felix = Cat(LocalDate.of(2013, 11, 27))
felix.mood = Mood.HUNGRY // "mood change: GRUMPY -> HUNGRY\n Time to feed the cat..."
felix.mood = Mood.SLEEPY // "mood change: HUNGRY -> ASLEEP\n Time to rest..."
felix.mood = Mood.GRUMPY // "mood change: ASLEEP -> GRUMPY\n All as always"
```

---

---

```
class JsonPerson(properties: Map<String, Any>) {  
    val name: String by properties // Delegates property to the map  
    val age: Int by properties // Delegates property to the map  
    val mood: Mood by properties // Delegates property to the map  
}  
  
// Let's assume this data comes from JSON; keys in the map must match property names  
val jsonData = mutableMapOf("name" to "John Doe", "age" to 42, "mood" to "GRUMPY")  
  
// You may need to preprocess some data (requires MutableMap)  
jsonData["mood"] = Mood.valueOf(jsonData["mood"] as String) // 'valueOf' is built-in  
  
// Creates an object from the map  
val john = JsonPerson(jsonData) // Properties are matched to the keys in the map  
println(john.name) // Prints "John Doe"  
println(john.age) // Prints 42  
println(john.mood) // Prints "GRUMPY"  
  
// Read-only property changes if backing map changes  
jsonData["name"] = "Hacker"  
println(john.name) // Prints "Hacker"
```

---

---

```
interface Kickable {
    fun kick()
}

// Existing implementation of Kickable
class BaseKickHandler : Kickable {
    override fun kick() { println("Got kicked") }
}

// Football implements interface by delegating to existing impl. (zero boilerplate)
class Football(kickHandler: Kickable) : Kickable by kickHandler
```

---

---

```
// Implements interface with manual delegation
class Football(val kickHandler: Kickable) : Kickable {
    override fun kick() {
        kickHandler.kick() // Trivial forwarding; necessary for every interface method
    }
}
```

---

---

```
open class ForwardingMutableSet<E>(set: MutableSet<E>) : MutableSet<E> by set
```

---

---

```
class Foo {  
    fun plainMethod() { ... }  
    infix fun with(other: Any) = Pair(this, other)  
    inline fun inlined(i: Int, operation: (Int, Int) -> Int) = operation(i, 42)  
    operator fun Int.times(str: String) = str.repeat(this)  
    fun withDefaults(n: Int = 1, str: String = "Hello World") = n * str  
}  
  
val obj = Foo()  
obj.plainMethod()  
val pair = obj with "Kotlin"  
obj.inlined(3, { i, j -> i * j })      // 126  
obj.withDefaults(str = "Hello Kotlin") // "Hello Kotlin"  
with(obj) { 2 * "hi" }                  // Uses 'with' to access extension
```

---

---

```
class Container {  
    fun Int.foo() {  
        // Extends Int with a 'foo' method  
        println(toString())  
        // Calls Int.toString (like this.toString())  
        println(this@Container.toString()) // Calls Container.toString  
    }  
  
    fun bar() = 17.foo() // Uses extension method  
}  
  
Container().bar() // Prints "17\n Container@33833882"
```

---

---

```
class Company {  
    val yearlyRevenue = 10_000_000  
  
    class Nested {  
        val company = Company()      // Must create instance of outer class manually  
        val revenue = company.yearlyRevenue  
    }  
  
    inner class Inner {          // Inner class due to 'inner' modifier  
        val revenue = yearlyRevenue // Has access to outer class members automatically  
    }  
}
```

---

---

```
class Task(_title: String, _priority: Int) { // Defines regular parameters
    val title = _title.capitalize()           // Uses parameter in initializer
    var priority: Int

    init {
        priority = Math.max(_priority, 0)    // Uses parameter in init block
    }
}
```

---

---

```
class Task private constructor(title: String, priority: Int) { ... }
```

---

---

```
class Task(val title: String, var priority: Int) { // Parameters are now properties
    init {
        require(priority >= 0) // Uses property in init block
    }
}
```

---

---

```
class Task(val title: String, var priority: Int) {           // Primary
    constructor(person: Person) : this("Meet with ${person.name}", 50) { // Secondary
        println("Created task to meet ${person.name}")
    }
}
```

---

```
class Food(calories: Int, healthy: Boolean) {  
    constructor(calories: Int) : this(calories, false)  
    constructor(healthy: Boolean) : this(0, healthy)  
    constructor() : this(0, false)  
}
```

```
class Food(calories: Int = 0, healthy: Boolean = false)
```

---

```
interface Searchable {  
    fun search() // All implementing classes have this capability  
}
```

---

---

```
interface Archivable {
    var archiveWithTimeStamp: Boolean // Abstract property
    val maxArchiveSize: Long // Property with default impl., thus must be val
        get() = -1 // Default implementation returns -1

    fun archive() // Abstract method
    fun print() { // Open method with default implementation
        val withOrWithout = if (archiveWithTimeStamp) "with" else "without"
        val max = if (maxArchiveSize == -1L) "∞" else "$maxArchiveSize"
        println("Archiving up to $max entries $withOrWithout time stamp")
    }
}
```

---

---

```
class Task : Archivable, Serializable { // Implements multiple interfaces
    override var archiveWithTimeStamp = true
    override fun archive() { ... }
}
```

---

---

```
abstract class Issue(var priority: Int) {  
    abstract fun complete()           // Abstract method  
    open fun trivial() { priority = 15 } // Open method  
    fun escalate() { priority = 100 }   // Closed method  
}  
  
class Task(val title: String, priority: Int) : Issue(priority), Archivable {  
    // ...  
    override fun complete() { println("Completed task: $title") } // Required override  
    override fun trivial() { priority = 20 }                      // Optional override  
    // Cannot override 'escalate' because it is closed  
}
```

---

---

```
class Closed           // Closed class: cannot inherit from this
class ChildOfClosed : Closed() // NOT allowed: compile-time error
```

---

```
open class Open          // Open class: can inherit
class ChildOfOpen : Open() // Allowed
class ChildOfChild : ChildOfOpen() // NOT allowed: compile-time error
```

---

---

```
val item: Component = Leaf() // 'item' is a Leaf at runtime

if (item is Composite) { ... } // Checks if 'item' is of type Composite at runtime
if (item !is Composite) { ... } // Checks if 'item' is not a Composite at runtime
```

---

---

```
val item: Component? = null

val leaf: Leaf      = item as Leaf           // TypeCastException
val leafOrNull: Leaf? = item as Leaf?        // Evaluates to null
val leafSafe: Leaf? = item as? Leaf          // Evaluates to null
val leafNonNull: Leaf = (item as? Leaf) ?: Leaf() // Alternative using elvis op.
```

---

---

```
val composite: Component = Composite()

val leafOrNull: Leaf? = composite as Leaf?      // ClassCastException
val leafSafe: Leaf? = composite as? Leaf        // Evaluates to null
```

---

---

```
val comp: Component? = Leaf() // Type is nullable 'Component'

if (comp != null) { comp.component() } // Smart-cast to Component
if (comp is Leaf) { comp.leaf() } // Smart-cast to Leaf
when (comp) {
    is Composite -> comp.composite() // Smart-cast to Component
    is Leaf -> comp.leaf() // Smart-cast to Leaf
}
if (comp is Composite && comp.composite() == 16) {} // Smart-cast to Composite
if (comp !is Leaf || comp.leaf() == 43) {} // Smart-cast to Leaf inside condition
if (comp !is Composite) return
comp.composite() // Smart-cast to Composite (because of return above)
```

---

---

```
open class Parent {  
    val a = "public"  
    internal val b = "internal"  
    protected val c = "protected"  
    private val d = "private"  
  
    inner class Inner {  
        val accessible = "$a, $b, $c, $d" // All accessible  
    }  
}  
  
class Child : Parent() {  
    val accessible = "$a, $b, $c" // d not accessible because private  
}  
  
class Unrelated {  
    val p = Parent()  
    val accessible = "${p.a}, ${p.b}" // p.c, p.d not accessible  
}
```

---

---

```
open class Cache private constructor() {
    val INSTANCE = Cache()

    protected var size: Long = 4096 // Getter inherits visibility from property
    private set // Setter can have a different visibility
}
```

---

---

```
data class Contact(val name: String, val phone: String, var favorite: Boolean)
```

---

---

```
val john = Contact("John", "202-555-0123", true)
val john2 = Contact("John", "202-555-0123", true)
val jack = Contact("Jack", "202-555-0789", false)

// toString
println(jack)           // Contact(name=Jack, phone=202-555-0789, favorite=false)

// equals
println(john == john2)   // true
println(john == jack)    // false

// hashCode
val contacts = hashSetOf(john, jack, john2)
println(contacts.size)   // 2 (no duplicates in sets, uses hashCode)

// componentN
val (name, phone, _) = john      // Uses destructuring declaration
println("$name's number is $phone") // John's number is 202-555-0123

// copy
val johnsSister = john.copy(name = "Joanne")
println(johnsSister) // Contact(name=Joanne, phone=202-555-0123, favorite=true)
```

---

```
data class Credentials(val name: String, val password: String)
```

---

```
enum class PaymentStatus {  
    OPEN, PAID  
}
```

---

---

```
enum class PaymentStatus(val billable: Boolean) { // Enum with a property

    OPEN(true) {
        override fun calculate() { ... }
    },
    PAID(false) {
        override fun calculate() { ... }
    }; // Note the semicolon: it separates the enum instances from the members

    fun print() { println("Payment is ${this.name}") } // Concrete method
    abstract fun calculate() // Abstract method
}
```

---

---

```
val status = PaymentStatus.PAID
status.print() // Prints "Payment is PAID"
status.calculate()

// Kotlin generates 'name' and 'ordinal' properties
println(status.name) // PAID
println(status.ordinal) // 1
println(status.billable) // false

// Enum instances implement Comparable (order = order of declaration)
println(PaymentStatus.PAID > PaymentStatus.OPEN) // true

// 'values' gets all possible enum values
var values = PaymentStatus.values()
println(values.joinToString()) // OPEN, PAID

// 'valueOf' retrieves an enum instance by name
println(PaymentStatus.valueOf("OPEN")) // OPEN
```

---

---

```
val status = PaymentStatus.OPEN

val message = when(status) {
    PaymentStatus.PAID -> "Thanks for your payment!"
    PaymentStatus.OPEN -> "Please pay your bill so we can buy coffee."
} // No else branch necessary
```

---

---

```
sealed class BinaryTree // Sealed class with two direct subclasses (and no others)
data class Leaf(val value: Int) : BinaryTree()
data class Branch(val left: BinaryTree, val right: BinaryTree) : BinaryTree()

// Creates a binary tree object
val tree: BinaryTree = Branch(Branch(Leaf(1), Branch(Leaf(2), Leaf(3))), Leaf(4))
```

---

---

```
sealed class Expression // Sealed class representing possible arithmetic expressions
data class Const (val value: Int) : Expression()
data class Plus (val left: Expression, val right: Expression) : Expression()
data class Minus (val left: Expression, val right: Expression) : Expression()
data class Times (val left: Expression, val right: Expression) : Expression()
data class Divide(val left: Expression, val right: Expression) : Expression()

fun evaluate(expr: Expression): Double = when(expr) {
    is Const -> expr.value.toDouble()
    is Plus -> evaluate(expr.left) + evaluate(expr.right)
    is Minus -> evaluate(expr.left) - evaluate(expr.right)
    is Times -> evaluate(expr.left) * evaluate(expr.right)
    is Divide -> evaluate(expr.left) / evaluate(expr.right)
} // No else branch required: all possible cases are handled

val formula: Expression = Times(Plus(Const(2), Const(4)), Minus(Const(8), Const(1)))
println(evaluate(formula)) // 42.0
```

---

```
interface A { val a: Int }
interface B { fun b() }
data class Both(override val a: Int) : A, B { override fun b() { ... } }
```

```
sealed class Either
class MyA(override val a: Int) : Either(), A
class MyB : Either(), B { override fun b() { ... } }
```

---

```
fun areaOfEllipse(vertex: Double, covertex: Double): Double {
    val ellipse = object { // Ad-hoc object
        val x = vertex
        val y = covertex
    }
    return Math.PI * ellipse.x * ellipse.y
}
```

---

---

```
scrollView.setOnDragListener(object : View.OnDragListener {  
    override fun onDrag(view: View, event: DragEvent): Boolean {  
        Log.d(TAG, "Dragging...")  
        return true  
    }  
})
```

---

```
scrollView.setOnDragListener { view, event -> ... }
```

---

```
class ReturnsObjectExpressions {  
    fun prop() = object {  
        // Returns an object; infers Any type  
        val prop = "Not accessible"  
    }  
  
    fun propWithInterface() = object : HasProp { // Returns an object; infers HasProp  
        override val prop = "Accessible"  
    }  
  
    fun access() {  
        prop().prop // Compile-time error (Any does not have prop)  
        propWithInterface().prop // Now possible (HasProp has prop)  
    }  
}  
  
interface HasProp { // Allows exposing the 'prop' to the outside  
    val prop: String  
}
```

---

---

```
import javafx.scene.Scene

object SceneRegistry { // Declares an object: this is effectively a Singleton
    lateinit private var homeScene: Scene
    lateinit private var settingsScene: Scene
    fun buildHomeScene() { ... }
    fun buildSettingsScene() { ... }
}
```

---

---

```
val home = SceneRegistry.buildHomeScene()
val settings = SceneRegistry.buildSettingsScene()
```

---

---

```
data class Car(val model: String, val maxSpeed: Int) {  
    companion object Factory {  
        fun defaultCar() = Car("SuperCar XY", 360)  
    }  
}  
  
val car = Car.defaultCar() // Calls companion method: same as Car.Factory.defaultCar()
```

---

---

```
interface CarFactory {
    fun defaultCar(): Car
}

data class Car(val model: String, val maxSpeed: Int) {
    companion object Factory : CarFactory { // Companions can implement interfaces
        override fun defaultCar() = Car("SuperCar XY", 360)
    }
}
```

---

```
fun Car.Factory.cheapCar() = Car("CheapCar 2000", 110)
val cheap = Car.cheapCar()
```

---

```
class Box<T>(elem: T) { // Generic class: can be a box of integers, strings,  
    "  
    val element: T = elem  
}  
  
val box: Box<Int> = Box(17) // Type Box<Int> can also be inferred  
println(box.element)      // 17
```

---

---

```
sealed class BinaryTree<T> // Now generic: can carry values of any type
data class Leaf<T>(val value: T) : BinaryTree<T>()
data class Branch<T>(
    val left: BinaryTree<T>,
    val right: BinaryTree<T>
) : BinaryTree<T>()

val tree: BinaryTree<Double> = Leaf(3.1415) // Uses a binary tree with Double values
```

---

```
typealias Deck = Stack<Card>           // New name for generics-induced type
typealias Condition<T> = (T) -> Boolean // New name for function type
```

---

```
val isNonNegative: Condition<Int> = { it >= 0 } // Can assign a lambda
```

---

```
fun <T> myListOf(vararg elements: T) = Arrays.asList(*elements) // * is spread operator

val list0: List<Int> = myListOf<Int>(1, 2, 3) // All types explicit, no type inference
val list1: List<Int> = myListOf(1, 2, 3) // Infers myListOf<Int>
val list2 = myListOf(1, 2, 3)           // Infers MutableList<Int> and myListOf<Int>
val list3 = myListOf<Number>(1, 2, 3)    // Infers MutableList<Number>

// Without parameters
val list4 = myListOf<Int>()           // Infers MutableList<Int>
val list5: List<Int> = myListOf()       // Infers myListOf<Int>
```

---

---

```
fun <T> Iterable<*>.filterByType(clazz: Class<T>): List<T?> {
    @Suppress("UNCHECKED_CAST") // Must suppress unchecked cast
    return this.filter { clazz.isInstance(it) }.map { it as T? } // Uses reflection
}

val elements = listOf(4, 5.6, "hello", 6, "hi", null, 1)
println(elements.filterByType(Int::class.javaObjectType)) // 4, 6, 1
```

---

---

```
inline fun <reified T> Iterable<*>.filterByType(): List<T?> { // Inline + reified
    return this.filter { it is T }.map { it as T? } // No reflection; can use 'is'
}

val elements = listOf(4, 5.6, "hello", 6, "hi", null, 1)
println(elements.filterByType<Int>())           // 4, 6, 1
```

---

---

```
elements.filter { it is Int }.map { it as Int? } // Inlined function and inserted T
```

---

---

```
val n: Number = 3 // Integer is a subclass of Number
val todo: Archivable = Task("Write book", 99) // Task is a subclass of Archivable
```

---

---

```
abstract class Human(val birthday: LocalDate) {  
    abstract fun age(): Number // Specifies that method returns a Number  
}  
  
class Student(birthday: LocalDate) : Human(birthday) {  
    // Return types allow variance so that Int can be used in place of Number  
    override fun age(): Int = ChronoUnit.YEARS.between(birthday, LocalDate.now()).toInt()  
}
```

---

```
Number[] arr = new Integer[3]; // Arrays are covariant
arr[0] = 3.7; // Causes ArrayStoreException
```

---

```
val arr: Array<Number> = arrayOf<Int>(1, 2, 3) // Compile-time error
arr[0] = 3.1415 // Would be unsafe (but cannot happen)
```

---

---

```
List<Number> numbers = new ArrayList<Integer>(); // Compile-time error
```

---

---

```
val numbers: MutableList<Number> = mutableListOf<Int>(1, 2, 3) // Compile-time error
```

---

---

```
val numbers: MutableList<Number> = mutableListOf<Int>(1, 2, 3) // Compile-time error
```

---

---

```
open class Stack<out E>(vararg elements: E) { // 'out' indicates covariance
    protected open val elements: List<E> = elements.toList() // E used in out-position
    fun peek(): E = elements.last() // E used in out-position
    fun size() = elements.size
}
```

---

```
val stack: Stack<Number> = Stack<Int>(1, 2, 3) // Allowed because covariant
```

---

---

```
class MutableStack<E>(vararg elements: E) : Stack<E>() { // Mutable: must be invariant
    override val elements: MutableList<E> = elements.toMutableList()
    fun push(element: E) = elements.add(element)      // E used in in-position
    fun pop() = elements.removeAt(elements.size - 1) // E used in out-position
}
```

---

```
val mutable: MutableStack<Number> = MutableStack<Int>(1, 2, 3) // Compile-time error
```

---

---

```
interface Compare<in T> {      // 'in' indicates contravariance
    fun compare(a: T, b: T): Int // T is only used at in-position
}
```

---

---

```
val taskComparator: Compare<Task> = object : Compare<Issue> { // Uses contravariance
    override fun compare(a: Issue, b: Issue) = a.priority - b.priority
}
```

---

---

```
interface Repair<in T> { // Contravariant
    fun repair(t: T)      // T in in-position
}

open class Vehicle(var damaged: Boolean)
class Bike(damaged: Boolean) : Vehicle(damaged)

class AllroundRepair : Repair<Vehicle> {      // Knows how to repair any vehicle
    override fun repair(vehicle: Vehicle) {
        vehicle.damaged = false
    }
}

val bikeRepair: Repair<Bike> = AllroundRepair() // Uses contravariance
```

---

---

```
val normal: MutableList<Number> = mutableListOf<Int>(1, 2, 3) // Compile-time error
val producer: MutableList<out Number> = mutableListOf<Int>(1, 2, 3) // Now covariant

// Can only read from producer, not write to it
producer.add(???) // Parameter type is Nothing, so impossible to call 'add'
val n: Number = producer[0] // Returns Number
```

---

---

```
val normal: MutableList<Int> = mutableListOf<Number>(1.7, 2f, 3) // Compile-time error
val consumer: MutableList<in Int> = mutableListOf<Number>(1.7, 2f, 3) // Contravariant

// Can write to consumer normally, but only read values as Any?
consumer.add(4)
val value: Any? = consumer[0] // Return type is Any? (only type-safe choice)
```

---

---

```
fun <E> transfer(from: MutableStack<E>, to: MutableStack<E>) {
    while (from.size() > 0) { to.push(from.pop()) }
}

val from = MutableStack<Int>(1, 2, 3)
val to = MutableStack<Number>(9.87, 42, 1.23)
transfer(from, to) // Compile-time error, although it would be safe
```

---

---

```
fun <E> transfer(from: MutableStack<out E>, to: MutableStack<in E>) {
    // Same as before
}
// ...
transfer(from, to) // Works now due to variance
```

---

---

```
interface Repair<in T : Vehicle> { // Upper bound for T is Vehicle
    fun repair(t: T)
}
```

---

---

```
interface Repairable { ... }

interface Repair<in T>
  where T : Vehicle, // where clause lists all type bounds
    T : Repairable {
  fun repair(t: T)
}
```

---

```
fun <T> min(a: T, b: T): T where T : Comparable<T> = if (a < b) a else b
```

---

```
fun printAll(array: Array<*>) { // Nothing known about type of array, but type-safe
    array.forEach(::println)
}

printAll(arrayOf(1, 2, 3))
printAll(arrayOf("a", 2.7, Person("Sandy"))) // Can pass in arrays of arbitrary type
```

---

---

```
interface Function<in T, out R> { // Contravariant w.r.t. T and covariant w.r.t. R
    fun apply(t: T): R
}

object IssueToInt : Function<Issue, Int> {
    override fun apply(t: Issue) = t.priority
}

// Star projections

val f1: Function<*, Int> = IssueToInt // apply() not callable but would return Int
val f2: Function<Task, *> = IssueToInt // apply() callable with Task, returns Any?
val f3: Function<*, *> = IssueToInt // apply() not callable and would return Any?

val errand = Task("Save the world", 80)
val urgency: Any? = f2.apply(errand)
```

---

---

```
import com.google.common.math.Stats
import java.util.ArrayList

// Using the standard library
val arrayList = ArrayList<String>() // Uses java.util.ArrayList
arrayList.addAll(arrayOf("Mercury", "Venus", "Jupiter"))
arrayList[2] = arrayList[0] // Indexed access operator calls 'get' and 'set'

// Looping as usual, ArrayList provides iterator()
for (item in arrayList) { println(item) }

// Using a third-party library (Google Guava)
val stats = Stats.of(4, 8, 15, 16, 23, 42)
println(stats.sum()) // 108.0
```

---

---

```
// Java

public class GettersAndSetters {
    private String readOnly = "Only getter defined";
    private String writeOnly = "Only setter defined";
    private String readWrite = "Both defined";

    public String getReadOnly() { return readOnly; }
    public void setWriteOnly(String writeOnly) { this.writeOnly = writeOnly; }
    public String getReadWrite() { return readWrite; }
    public void setReadWrite(String readWrite) { this.readWrite = readWrite; }
}

// Kotlin

private val gs = GettersAndSetters()
println(gs.readOnly)          // Read-only attribute acts like a val property

gs.readWrite = "I have both"  // Read-write attribute acts like a var property
println(gs.readWrite)

gs.setWriteOnly("No getter") // Write-only properties not supported in Kotlin
```

---

---

```
// Java
public static String hello() { return "I could be null"; }

// Kotlin
val str = hello()           // Inferred type: String (by default)
println(str.length)          // 15

val nullable: String? = hello() // Explicit type: String?
println(nullable?.length)     // 15
```

---

---

```
// Java
public static @Nullable String nullable() { return null; }
public static @NotNull String nonNull() { return "Could be null, but with warning"; }

// Kotlin
val s1 = nullable() // Inferred type: String?
val s2 = nonNull() // Inferred type: String
```

---

---

```
// Java
class KeywordsAsIdentifiers {
    public int val = 100;
    public Object object = new Object();
    public boolean in(List<Integer> list) { return true; }
    public void fun() { System.out.println("This is fun."); }
}
```

```
// Kotlin
val kai = KeywordsAsIdentifiers()
kai.`val`
kai.`object`
kai.`in`(listOf(1, 2, 3))
kai.`fun`()
```

---

---

```
// Java
public static List<String> myListOf(String... strings) { // Vararg method from Java
    return Arrays.asList(strings); // Passes in vararg unchanged
}

// Kotlin
val list = myListOf("a", "b", "c") // Can pass in any number of args
val values = arrayOf("d", "e", "f")
val list2 = myListOf(*values) // Spread operator required
```

---

---

```
// Java
public class Box {
    private final int value;

    public Box(int value) { this.value = value; }
    public Box plus(Box other) { return new Box(this.value + other.value); }
    public Box minus(Box other) { return new Box(this.value - other.value); }
    public int getValue() { return value; }
}

// Kotlin
val value1 = Box(19)
val value2 = Box(37)
val value3 = Box(14)

val result = value1 + value2 - value3 // Uses 'plus' and 'minus' as operators
println(result.value) // 42
```

---

---

```
// Java
interface Producer<T> { // SAM interface (single abstract method)
    T produce();
}

// Kotlin
private val creator = Producer { Box(9000) } // Inferred type: Producer<Box>
```

---

---

```
// Kotlin
val thread = Thread { // No need to write Thread(Runnable { ... })
    println("I'm the runnable")
}
```

---

---

```
// Kotlin
class KotlinClass {
    val fixed: String = "base.KotlinClass"
    var mutable: Boolean = false
}

// Java
KotlinClass kotlinClass = new KotlinClass();
String s = kotlinClass.getFixed();      // Uses getter of 'val'
kotlinClass.setMutable(true);          // Uses setter of 'var'
boolean b = kotlinClass.getMutable();  // Uses getter of 'var'
```

---

---

```
// Kotlin
class KotlinClass {
    var mutable: Boolean = false
        @JvmName("isMutable") get // Specifies custom getter name for Java bytecode
    }

// Java
boolean b = kotlinClass.isMutable(); // Now getter is accessible as 'isMutable'
```

---

```
public final class KotlinClass {  
    @NotNull private final String fixed = "base.KotlinClass";  
    private boolean mutable;  
  
    @NotNull public final String getFixed() {  
        return this.fixed;  
    }  
    public final boolean getMutable() { return this.mutable; }  
    public final void setMutable(boolean var1) { this.mutable = var1; }  
}
```

---

```
// Kotlin
val prop = "Default: private field + getter/setter" // Here no setter because read-only

@JvmField
val exposed = "Exposed as a field in Java" // No getter or setter generated

// Decompiled Java code (surrounding class omitted)
@NotNull private static final String prop = "Default: private field + getter/setter";
@NotNull public static final String getProp() { return prop; }

@NotNull public static final String exposed = "Exposed as a field in Java";
```

---

---

```
// sampleName.kt

package com.example

class FileLevelClass           // Generates class FileLevelClass
object FileLevelObject        // Generates FileLevelObject as Singleton
fun fileLevelFunction() {}    // Goes into generated class SampleNameKt
val fileLevelVariable = "Usable from Java" // Goes into generated class SampleNameKt
```

---

---

```
// sampleName.kt
@file:JvmName("MyUtils") // Must be the first statement in the file
// ...
```

---

---

```
// Kotlin
@file:JvmName("Notifications")

fun Context.toast(message: String) { // Top-level function => becomes static method
    Toast.makeText(this, message, Toast.LENGTH_SHORT).show()
}

// Within a Java Activity (Android)
Notifications.toast(this, "Quick info...");
```

---

---

```
// Kotlin
class Notifier {
    fun Context.longToast(message: String) { // Becomes member method
        Toast.makeText(this, message, Toast.LENGTH_LONG).show()
    }
}

// Calling from a Kotlin Android activity
with(Notifier()) { longToast("Important notification...") }
Notifier().apply { longToast("Important notification...") }

// Calling from a Java Android Activity
Notifier notifier = new Notifier();
notifier.longToast(this, "Important notification..."); // 'this' is the Context arg
```

---

---

```
// Kotlin
const val CONSTANT = 360

object Cache { val obj = "Expensive object here..." }
```

```
class Car {
    companion object Factory { val defaultCar = Car() }
}
```

```
// Decompiled Java code (simplified)
public final class StaticFieldsKt {
    public static final int CONSTANT = 360;
}
```

---

```
public final class Cache { // Simplified
    private static final String obj = "Expensive object here..."; // private static field
    public final String getObj() { return obj; } // with getter
}
```

```
public final class Car { // Simplified
    private static final Car defaultCar = new Car(); // Static field
    public static final class Factory {
        public final Car getDefaultCar() { return Car.defaultCar; } // Nonstatic method
    }
}
```

---

---

```
// Kotlin

object Cache {
    @JvmStatic fun cache(key: String, obj: Any) { ... } // Becomes a static member
}

class Car {
    companion object Factory {
        @JvmStatic fun produceCar() { ... } // Now becomes static as well
    }
}

// Inside a Java method
Cache.cache("supercar", new Car()); // Static member is callable directly on class
Cache.INSTANCE.cache("car", new Car()); // Bad practice

Car.produceCar(); // Static
Car.Factory.produceCar(); // Also possible
(new Car()).produceCar(); // Bad practice
```

---

---

```
// Kotlin
@JvmOverloads // Triggers generation of overloaded methods in Java bytecode
fun <T> Array<T>.join(delimiter: String = ", ",
                         prefix: String = "",
                         suffix: String = ""): String {
    return this.joinToString(delimiter, prefix, suffix)
}

// Java
String[] languages = new String[] {"Kotlin", "Scala", "Java", "Groovy"};

// Without @JvmOverloads: you must pass in all parameters
ArrayUtils.join(languages, ";", "{}"); // Assumes @file:JvmName("ArrayUtils")

// With @JvmOverloads: overloaded methods
ArrayUtils.join(languages);           // Skips all optional parameters
ArrayUtils.join(languages, ";");      // Skips prefix and suffix
ArrayUtils.join(languages, ";", "Array: "); // Skips suffix
ArrayUtils.join(languages, ";", "[", "]"); // Passes in all possible arguments
```

---

---

```
// Kotlin
sealed class Component
data class Composite(val children: List<Component>) : Component()
data class Leaf(val value: Int): Component()

// Java
Component comp = new Composite(asList(new Leaf(1), new Composite(...)));

if (comp instanceof Composite) { // Cannot use 'switch', must use 'if'
    out.println("It's a Composite"); // No smart-casts
} else if (comp instanceof Leaf) { // No exhaustiveness inferred
    out.println("It's a Leaf");
}
```

---

---

```
// Kotlin
data class Person(val name: String = "", val alive: Boolean = true)

// Java
Person p1 = new Person("Peter", true);
Person p2 = new Person("Marie Curie", false);
Person p3 = p2.copy("Marie Curie", false); // No advantage over constructor

String name = p1.getName(); // componentN() methods superfluous
out.println(p1);           // Calls toString()
p2.equals(p3);            // true
```

---

---

```
// Java

import kotlin.jvm.JvmClassMappingKt;

import kotlin.reflect.KClass;

KClass<A> clazz = JvmClassMappingKt.getKotlinClass(A.class);

// Kotlin

import kotlin.reflect.KClass

private val kclass: KClass<A> = A::class
private val jclass: Class<A> = A::class.java
```

---

---

```
fun List<Customer>.validate() {} // JVM signature: validate(java.util.List)
fun List<CreditCard>.validate() {} // JVM signature: validate(java.util.List)
```

---

---

```
fun List<Customer>.validate() { ... }

@JvmName("validateCC") // Resolves the name clash
fun List<CreditCard>.validate() { ... }

// Both can be called as validate() from Kotlin (because dispatched at compile-time)
val customers = listOf(Customer())
val ccs      = listOf(CreditCard())
customers.validate()
ccs.validate()
```

---

---

```
// Kotlin

inline fun require(predicate: Boolean, message: () -> String) {
    if (!predicate) println(message())
}

fun main(args: Array<String>) { // Listing uses main function to show decompiled code
    require(someCondition()) { "someCondition must be true" }
}

// Decompiled Java Code (of main function)
public static final void main(@NotNull String[] args) {
    Intrinsics.checkNotNull(args, "args"); // Always generated by Kotlin
    boolean predicate$iv = someCondition();
    if (!predicate$iv) { // Inlined function call
        String var2 = "someCondition must be true";
        System.out.println(var2);
    }
}
```

---

---

```
// Kotlin
import java.io.*

@Throws(FileNotFoundException::class) // Generates throws clause in bytecode
fun readInput() = File("input.csv").readText()

// Java
import java.io.FileNotFoundException;
// ...
try {                      // Must handle exception
    CsvUtils.readInput();   // Assumes @file:JvmName("CsvUtils")
} catch (FileNotFoundException e) {
    // Handle non-existing file...
}
```

---

---

```
// Without @Throws
public static final String readInput() { ... }

// With @Throws
public static final String readInput() throws FileNotFoundException { ... }
```

---

---

```
// Kotlin
class Stack<out E>(vararg items: E) { ... }
fun consumeStack(stack: Stack<Number>) { ... }

// Java: you can use the covariance of Stack
consumeStack(new Stack<Number>(4, 8, 15, 16, 23, 42));
consumeStack(new Stack<Integer>(4, 8, 15, 16, 23, 42));
```

---

---

```
// Kotlin
fun consumeStack(stack: Stack<@JvmSuppressWildcards Number>) { ... } // No more wildcards

// Normally no wildcards are generated for return types, unless you use @JvmWildcard
fun produceStack(): Stack<@JvmWildcard Number> {
    return Stack(4, 8, 15, 16, 23, 42)
}

// Java
consumeStack(new Stack<Number>(4, 8, 15, 16, 23, 42)); // Matches exactly
consumeStack(new Stack<Integer>(4, 8, 15, 16, 23, 42)); // Error: No longer allowed

Stack<Number> stack = produceStack(); // Error: No longer allowed
Stack<? extends Number> stack = produceStack(); // Inconvenient, thus bad practice
```

---

---

```
// Kotlin

fun fail(message: String): Nothing {           // Indicates non-terminating function
    throw AssertionError(message)
}

fun takeNothing(perpetualMotion: Nothing) {} // Impossible to call from Kotlin

// Java

NothingKt.takeNothing(null); // Possible in Java (but with warning due to @NotNull)
NothingKt.fail("Cannot pass null to non-null variable"); // Cannot terminate
System.out.println("Never reached but Java doesn't know"); // Dead code
```

---

---

```
// Simplified, no error handling
fetchUser(userId) { user ->
    fetchLocation(user) { location ->
        fetchWeather(location) { weatherData ->
            updateUi(weatherData)
        }
    }
}
```

---

---

```
// Simplified, no error handling
fetchUser(userId)
    .thenCompose { user -> fetchLocation(user) }
    .thenCompose { location -> fetchWeather(location) }
    .thenAccept { weatherData -> updateUi(weatherData) }
```

---

---

```
// C#: Notice the return type Task<Location> instead of Location
async Task<Location> FetchLocation(User user) { ... }

var location = await FetchLocation(user) // Need await keyword to fetch location
```

---

```
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:0.25.3"
```

```
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:0.25.3"
```

---

```
val user = fetchUser(userId) // Asynchronous call in sequential style
val location = fetchLocation(user)
val weatherData = fetchWeather(location)
updateUi(weatherData)
```

---

---

```
suspend fun fetchUser(): User { ... }
```

---

---

```
// Callback-based
fun fetchUser(callback: (User) -> Unit) { ... }
```

---

```
// Future-based
fun fetchUser(): Future<User> { ... }
```

---

---

```
suspend fun updateWeather() {  
    val user = fetchUser(userId)          // Suspension point #1  
    val location = fetchLocation(user)    // Suspension point #2  
    val weatherData = fetchWeather(location) // Suspension point #3  
    updateUi(weatherData)  
}
```

---

---

```
fun fetchUserAsync(): Future<User> { ... }

// Call-site
val user = fetchUserAsync() // Asynchrony is explicit, so you expect a Future<User>
```

---

---

```
fun main(args: Array<String>) {
    // Error: Suspend function must be called from coroutine or other suspend function
    updateWeather()
}
```

---

---

```
suspend fun updateWeather(userId: Int) {
    val user = fetchUser(userId)
    val locations = fetchLocations(user) // Now returns a list of locations
    for (location in locations) {      // Uses loop naturally in suspending function
        val weatherData = fetchWeather(location)
        updateUi(weatherData, location)
    }
}
```

---

---

```
suspend fun updateWeather(userId: Int) {
    try { // Uses try-catch naturally within suspending function
        val user = fetchUser(userId)
        val location = fetchLocation(user)
        val weatherData = fetchWeather(location)
        updateUi(weatherData)
    } catch(e: Exception) {
        // Handle exception
    } finally {
        // Cleanup
    }
}
```

---

---

```
suspend fun updateWeather(userId: Int) {
    val user = fetchUser(userId)
    fetchLocations(user).forEach {           // Can use all higher-order fcts. as usual
        val weatherData = fetchWeather(it) // Can be called inside higher-order function
        updateUi(weatherData, it)
    }
}
```

---

---

```
import kotlinx.coroutines.runBlocking

fun main(args: Array<String>) {
    runBlocking {
        // Can call suspending functions inside runBlocking
        updateWeather()
    }
}
```

---

---

```
fun <T> runBlocking(block: suspend () -> T) // Blocks thread until coroutine finishes
```

---

---

```
fun main(args: Array<String>) = runBlocking<Unit> { // Allows suspend calls in main
    updateWeather()
}

// In a test case...
@Test fun testUpdateWeather() = runBlocking { ... updateWeather() ... }
```

---

---

```
import kotlinx.coroutines.*

// Launches non-blocking coroutine
launch {
    println("Coroutine started") // 2nd print
    delay(1000)                // Calls suspending function from within coroutine
    println("Coroutine finished") // 3rd print
}

println("Script continues") // 1st print
Thread.sleep(1500)          // Keep program alive
println("Script finished") // 4th print
```

---

---

```
import kotlinx.coroutines.*

runBlocking { // runBlocking is required to call suspending 'join'
    val job = launch { // launch returns a Job object
        println("Coroutine started")
        delay(1000)
        println("Coroutine finished")
    }

    println("Script continues")
    job.join() // Waits for completion of coroutine (non-blocking)
    println("Script finished")
}
```

---

---

```
import kotlinx.coroutines.*

runBlocking {
    // Launch 100,000 coroutines
    val jobs = List(100_000) {
        launch {
            delay(1000)
            print("+")
        }
    }
    jobs.forEach { it.join() }
}
```

---

---

```
import kotlinx.coroutines.*

runBlocking {
    val job = launch {
        repeat(10) {
            delay(300)           // Cooperative delay from stdlib
            println("${it + 1} of 10...") // Only 1 of 10, 2 of 10, 3 of 10 are printed
        }
    }

    delay(1000)
    println("main(): No more time")
    job.cancel() // Can control cancellation on per-coroutine level
    job.join()   // Then wait for it to cancel
    println("main(): Now ready to quit")
}
```

---

---

```
import kotlinx.coroutines.*

runBlocking {
    val job = launch {
        repeat(10) {
            Thread.sleep(300)           // Non-cooperative
            println("${it + 1} of 10...")
        }
    }

    delay(1000)
    job.cancelAndJoin() // Cancel is ignored, will wait for another 2 seconds
}
```

---

---

```
import kotlinx.coroutines.*

runBlocking {
    val job = launch {
        repeat(10) {
            if (isActive) { // Cooperative
                Thread.sleep(300)
                println("${it + 1} of 10...")
            }
        }
    }

    delay(1000)
    job.cancelAndJoin()
}
```

---

---

```
import kotlinx.coroutines.*

fun fetchFirstAsync() = async { // Return type is Deferred<Int>
    delay(1000)
    294 // Return value of lambda, type Int
}

fun fetchSecondAsync() = async { // Return type is Deferred<Int>
    delay(1000)
    7 // Return value of lambda, type Int
}

runBlocking {
    // Asynchronous composition: total runtime ~1s
    val first = fetchFirstAsync() // Inferred type: Deferred<Int>
    val second = fetchSecondAsync() // Inferred type: Deferred<Int>
    val result = first.await() / second.await() // Awaits completion; now type is Int
    println("Result: $result") // 42
}
```

---

---

```
import kotlinx.coroutines.*

runBlocking {
    // Effectively synchronous: total runtime ~2s
    val first = fetchFirstAsync().await()    // Inferred type: Int (runtime ~1s)
    val second = fetchSecondAsync().await()  // Inferred type: Int (runtime ~1s)
}
```

---

---

```
import kotlinx.coroutines.*

suspend fun fetchFirst(): Int {
    delay(1000); return 294
}

suspend fun fetchSecond(): Int {
    delay(1000); return 7
}

runBlocking {
    val a = async { fetchFirst() } // Asynchrony is explicit
    val b = async { fetchSecond() } // Asynchrony is explicit
    println("Result: ${a.await() / b.await()}")
}
```

---

---

```
import kotlinx.coroutines.*

runBlocking {
    val deferred: Deferred<Int> = async { throw Exception("Failed...") }

    try {
        deferred.await() // Tries to get result of asynchronous call, throws exception
    } catch (e: Exception) {
        // Handle failure case here
    }
}
```

---

---

```
import kotlinx.coroutines.android.UI
import kotlinx.coroutines.launch

launch(UI) { // The UI context is provided by the coroutines-android dependency
    updateUi(weatherData)
}
```

---

---

```
val UI = HandlerContext(Handler(Looper.getMainLooper()), "UI")
```

---

---

```
launch { ... }
launch(DefaultDispatcher) { ... }
launch(CommonPool) { ... }
```

---

---

```
launch(newSingleThreadContext("MyNewThread")) { ... } // Runs coroutine in new thread
```

---

---

```
launch(coroutineContext) { ... }
```

---

---

```
import kotlinx.coroutines.*
import kotlin.coroutines.experimental.coroutineContext

runBlocking {
    val jobs = mutableListOf<Job>()

    // DefaultDispatcher (CommonPool at the time of writing)
    jobs += launch {
        println("Default: In thread ${Thread.currentThread().name} before delay")
        delay(500)
        println("Default: In thread ${Thread.currentThread().name} after delay")
    }

    jobs += launch(newSingleThreadContext("New Thread")) {
        println("New Thread: In thread ${Thread.currentThread().name} before delay")
        delay(500)
        println("New Thread: In thread ${Thread.currentThread().name} after delay")
    }
}
```

---

```
jobs += launch(Unconfined) {
    println("Unconfined: In thread ${Thread.currentThread().name} before delay")
    delay(500)
    println("Unconfined: In thread ${Thread.currentThread().name} after delay")
}

jobs += launch(coroutineContext) {
    println("cC: In thread ${Thread.currentThread().name} before delay")
    delay(500)
    println("cC: In thread ${Thread.currentThread().name} after delay")
}

jobs.forEach { it.join() }
```

---

---

```
import kotlinx.coroutines.*
import kotlinx.coroutines.android.UI

suspend fun updateWeather(userId: Int) {
    val user = fetchUser(userId)
    val location = fetchLocation(user)
    val weatherData = fetchWeather(location)

    withContext(UI) {
        updateUi(weatherData)
    }
}

// Call-site
launch { updateWeather() }
```

---

```
suspend fun loadUser(id: Int): User = withContext(DB) { ... }
```

---

```
import kotlinx.coroutines.*

runBlocking {
    withTimeout(1200) {
        repeat(10) {
            delay(500)
            println("${it + 1} of 10")
        }
    }
}
```

---

---

```
import kotlinx.coroutines.*

runBlocking {
    try {
        withTimeout(1200) {
            repeat(10) {
                delay(500)
                println("${it + 1} of 10")
            }
        }
    } catch (e: TimeoutCancellationException) {
        println("Time is up!")
    } finally {
        println("Cleaning up open connections...")
    }
}
```

---

---

```
import kotlinx.coroutines.*

runBlocking {
    val parent = launch {
        repeat(10) { i ->
            launch { // Implicitly uses coroutineContext, thus parent context
                delay(300 * (i + 1))
                println("Coroutine ${i + 1} finished") // Only 1, 2, and 3 get to print
            }
        }
    }

    delay(1000)
    parent.cancelAndJoin() // Cancels parent coroutine along with all its children
}
```

---

---

```
import kotlinx.coroutines.*
```

---

```
interface JobHolder { val job: Job }
```

---

---

```
import kotlinx.coroutines.Job
import android.support.v7.app.AppCompatActivity

class MainActivity : AppCompatActivity(), JobHolder {
    override val job = Job()
    override fun onDestroy() {
        super.onDestroy()
        job.cancel()
    }
}
```

---

---

```
import android.view.View
import kotlinx.coroutines.NonCancellable

val View.contextJob
    get() = (context as? JobHolder)?.job ?: NonCancellable
```

---

---

```
import kotlinx.coroutines.*

// CoroutineName
val name = CoroutineName("Koroutine")
launch(name) { ... }

// CoroutineExceptionHandler
val exceptionHandler = CoroutineExceptionHandler { context, exception ->
    println("Crashed with context $context")
    exception.printStackTrace()
}
launch(exceptionHandler) { ... }
```

---

---

```
public operator fun plus(context: CoroutineContext): CoroutineContext
```

---

---

```
// CoroutineName + CoroutineExceptionHandler
launch(name + exceptionHandler) { ... }

// Job + CoroutineDispatcher
launch(contextJob + UI) { ... } // Uses 'contextJob' from Listing 6.39 (Android)
```

---

---

```
import kotlinx.coroutines.*
import kotlin.coroutines.experimental.ContinuationInterceptor

launch(name + exceptionHandler) {
    println("Context: ${coroutineContext}")
    println("Job: ${coroutineContext[Job]}")
    println("Dispatcher: ${coroutineContext[ContinuationInterceptor]}")
    println("Name: ${coroutineContext[CoroutineName]}")
    println("Exception Handler: ${coroutineContext[CoroutineExceptionHandler]}")
}
```

---

---

```
runBlocking {  
    val job = launch(start = CoroutineStart.LAZY) { // Runs only when triggered  
        println("Lazy coroutine started")  
    }  
    println("Moving on...")  
    delay(1000)  
    println("Still no coroutine started")  
    job.join() // Triggers execution of the coroutine using join()  
    println("Joined coroutine")  
}
```

---

---

```
import kotlinx.coroutines.*  
import kotlinx.coroutines.future.*  
  
fun fetchValueAsync() = future { delay(500); 7 } // Uses 'future' coroutine builder  
  
runBlocking {  
    fetchValueAsync().thenApply { it * 6 }  
    .thenAccept { println("Retrieved value: $it") }  
    .await()  
}
```

---

---

```
import kotlin.coroutines.experimental.buildSequence
```

```
val fibonacci = buildSequence {  
    yield(1)  
    var a = 0  
    var b = 1  
    while (true) {  
        val next = a + b  
        yield(next)  
        a = b  
        b = next  
    }  
}
```

---

```
fibonacci.take(10).forEach { println(it) } // 1, 1, 2, 3, 5, 8, ..
```

---

```
fun <T> buildSequence(block: suspend SequenceBuilder<T>.() -> Unit): Sequence<T>
```

---

---

```
@RestrictsSuspension
public abstract class SequenceBuilder<in T> { ... } // Declaration from stdlib
```

---

---

```
import kotlinx.coroutines.channels.actor
import kotlinx.coroutines.runBlocking

val actor = actor<String> {
    val message = channel.receive()
    println(message)
}

runBlocking {
    actor.send("Hello World!") // Sends an element to the actor's channel
    actor.close()           // Closes channel because actor is no longer needed
}
```

---

---

```
import kotlinx.coroutines.channels.actor
import kotlinx.coroutines.runBlocking

// This actor keeps reading from its channel indefinitely (until it's closed)
val actor = actor<String> {
    for (value in channel) { println(value) }
}

runBlocking {
    actor.send("Hello") // Makes actor print out "Hello"
    actor.send("World") // Makes actor print out "World"
    actor.close()
}
```

---

---

```
import kotlinx.coroutines.channels.*
import kotlinx.coroutines.*

runBlocking {
    val actor = actor<String>(capacity = Channel.CONFLATED) { // Conflated channel
        for (value in channel) { println(value) }
    }

    actor.send("Hello") // Will be overwritten by following element
    actor.send("World") // Overwrites Hello if it was not yet consumed by the actor
    delay(500)
    actor.close()
}
```

---

---

```
import kotlinx.coroutines.channels.*
import kotlinx.coroutines.*
import kotlinx.coroutines.selects.select

runBlocking {
    val channel1 = Channel<Int>()
    val channel2 = Channel<Int>()

    launch { // No need for the actor coroutine builder
        while (true) {
            select<Unit> { // Provide any number of alternative data sources in here
                channel1.onReceive { println("From channel 1: $it") }
                channel2.onReceive { println("From channel 2: $it") }
            }
        }
    }

    channel1.send(17) // Sends 17 to channel 1, thus this source is selected first
    channel2.send(42) // Sends 42 to channel 2, causing channel 2 to be selected next
    channel1.close(); channel2.close()
}
```

---

---

```
import kotlinx.coroutines.channels.*
import kotlinx.coroutines.*

runBlocking {
    val channel = Channel<String>()
    repeat(3) { n ->
        launch {
            while (true) {
                channel.send("Message from actor $n")
            }
        }
    }
    channel.take(10).consumeEach { println(it) }
    channel.close()
}
```

---

---

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

runBlocking {
    val producer = produce {
        var next = 1
        while (true) {
            send(next)
            next *= 2
        }
    }
    producer.take(10).consumeEach { println(it) }
}
```

---

---

```
sealed class Transaction // Uses a .kt file to be able to declare sealed class
data class Deposit(val amount: Int) : Transaction()
data class Withdrawal(val amount: Int) : Transaction()

fun newAccount(startBalance: Int) = actor<Transaction>(capacity = 10) {
    var balance = startBalance
    for (tx in channel) {
        when (tx) {
            is Deposit    -> { balance += tx.amount; println("New balance: $balance") }
            is Withdrawal -> { balance -= tx.amount; println("New balance: $balance") }
        }
    }
}

fun main(args: Array<String>) = runBlocking<Unit> { // Not a script (see first line)
    val bankAccount = newAccount(1000)
    bankAccount.send(Deposit(500))
    bankAccount.send(Withdrawal(1700))
    bankAccount.send(Deposit(4400))
}
```

---

---

```
// Callbacks
fun fetchImageAsync(callback: (Image) -> Unit) { ... }
```

```
// Futures
fun fetchImageAsync(): Future<Image> { ... }
```

```
// Coroutines
suspend fun fetchImage(): Image { ... }
```

---

---

```
fun log(message: String) = println("[${Thread.currentThread().name}] $message")
```

---

---

```
class App : Application {

    override fun onCreate() {
        super.onCreate()
        System.setProperty("kotlinx.coroutines.debug",
            if (BuildConfig.DEBUG) "on" else "off")
    }
}

// In AndroidManifest.xml
<application
    android:name=".App" ...>
</application>
```

---

---

```
public fun <T> future(...): CompletableFuture<T> {
    // Set up coroutine context... (omitted)
    val future = CompletableFutureCoroutine<T>(newContext + job)
    job.cancelFutureOnCompletion(future)
    future.whenComplete { _, exception -> job.cancel(exception) }
    start(lambda, receiver=future, completion=future)

    return future
}

private class CompletableFutureCoroutine<T>(...): ... {
    // ...
    override fun resume(value: T) { complete(value) }
    override fun resumeWithException(exception: Throwable) {
        completeExceptionally(exception)
    }
}
```

---

---

```
import kotlinx.coroutines.future.future
import kotlinx.coroutines.*
import java.util.concurrent.CompletableFuture

suspend fun <T> CompletableFuture<T>.myAwait(): T {
    return suspendCancellableCoroutine { continuation ->
        whenComplete { value, exception ->
            if (exception == null) continuation.resume(value)
            else continuation.resumeWithException(exception)
        }
    }
}

runBlocking {
    val completable = future { delay(1000); 42 }
    println(completable.myAwait()) // 42

    val fail = future<Int> { throw Exception("Could not fetch value") }
    println(fail.myAwait()) // Throws exception
}
```

---

---

```
import kotlinx.coroutines.future

suspend fun fetchScore(): Int { ... } // This can hardly be called directly from Java
fun fetchScoreAsync() = future { fetchScore() } // This is to be used from Java

// Java
CompletableFuture<Integer> future = fetchScoreAsync();
int asyncResult = future.join();
```

---

---

```
suspend fun suspending(): Int {  
    delay(1000)    // Suspension point #1  
    val a = 17  
    otherSuspend() // Suspension point #2  
    println(a)  
  
    return 0  
}
```

---

---

```
suspend fun suspending(c: Continuation<Int>): ... { ... }
```

---

---

```
fun suspending(c: Continuation<Int>): ... { ... }
```

---

---

```
fun suspending(c: Continuation<Int>): Any? { ... }
```

---

---

```
fun suspending(c: Continuation<Int>): Any? {

    val stateMachine = object : CoroutineImpl(...) // Implements Continuation

    when (stateMachine.state) { // Three cases for three suspension points (states)
        is 0 -> {
            stateMachine.state = 1 // Updates current state
            delay(1000, stateMachine) // Passes state machine as continuation
        }
        is 1 -> {
            val a = 17
            stateMachine.my$a = a // Captures variable value
            stateMachine.state = 2 // Updates current state
            otherSuspend(stateMachine) // Passes state machine as continuation
        }
        is 2 -> {
            val a = stateMachine.my$a // Recovers value from context
            println(a)
            return 0
        }
    }
}
```

---

---

```
buildscript {  
    ext.kotlin_version = "1.2.60" // Adjust to your current version  
    // ...  
    dependencies {  
        // ...  
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"  
    }  
}  
// ...
```

---

---

```
apply plugin "com.android.application"
apply plugin "kotlin-android"
apply plugin "kotlin-android-extensions" // Recommended

...
dependencies {
    ...
    implementation "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
}
}
```

---

---

```
    android {  
        ...  
  
        sourceSets {  
            main.java.srcDirs += "src/main/kotlin"  
        }  
    }  
}
```

---

---

```
implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version" // JDK 7
implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk8:$kotlin_version" // JDK 8
```

---

---

```
implementation "org.jetbrains.kotlin:kotlin-reflect:$kotlin_version"
testImplementation "org.jetbrains.kotlin:kotlin-test:$kotlin_version"
testImplementation "org.jetbrains.kotlin:kotlin-test-junit:$kotlin_version"
```

---

---

```
apply plugin: 'kotlin-kapt' // Enables kapt for annotation processing
// ...

dependencies {
    implementation 'com.google.dagger:dagger:2.17'
    kapt 'com.google.dagger:dagger-compiler:2.17' // Uses kapt
}
```

---

```
apply plugin: 'kotlin-android'  
apply plugin: 'kotlin-android-extensions'
```

---

```
<android.support.constraint.ConstraintLayout
    app:layout_behavior="@string/appbar_scrolling_view_behavior" ...>

    <android.support.v7.widget.RecyclerView
        android:id="@+id/recyclerViewTodos"
        android:scrollbars="vertical"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</android.support.constraint.ConstraintLayout>
```

---

implementation 'com.android.support:design:27.1.1'

---

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

    <CheckBox
        android:id="@+id/cbTodoDone"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="@dimen/margin_medium" />

    <TextView
        android:id="@+id/tvTodoTitle"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:padding="@dimen/padding_large"
        android:textSize="@dimen/font_large" />

</LinearLayout>
```

---

---

```
data class TodoItem(val title: String)
```

---

---

```
import android.support.v7.widget.RecyclerView
import com.example.kudoo.model.TodoItem

class RecyclerListAdapter(
    private val items: MutableList<TodoItem>
) : RecyclerView.Adapter<RecyclerListAdapter.ViewHolder>() { // ViewHolder impl. next
    // ...
}
```

---

---

```
import android.support.v7.widget.RecyclerView
import android.view.View
import android.widget.*
import com.example.kudoo.R
import com.example.kudoo.model.TodoItem

class RecyclerListAdapter(...) : ... {
    // ...
    class ViewHolder(listItemView: View) : RecyclerView.ViewHolder(listItemView) {
        // ViewHolder stores all views it needs (only calls 'findViewById' once)
        val tvTodoTitle: TextView = listItemView.findViewById(R.id.tvTodoTitle)

        fun bindItem(todoItem: TodoItem) { // Binds a to-do item to the views
            tvTodoTitle.text = todoItem.title // Populates the text view with the to-do
            cbTodoDone.isChecked = false // To-do items are always 'not done' (or deleted)
        }
    }
}
```

---

---

```
androidExtensions {  
    experimental = true  
}
```

---

---

```
// ... (imports from before here)

import kotlinx.android.extensions.LayoutContainer
import kotlinx.android.synthetic.main.todo_item.* // Note synthetic import

class RecyclerListAdapter(...) : ... {
    ...
    class ViewHolder(
        ...
        override val containerView: View // Overrides property from LayoutContainer
    ) : RecyclerView.ViewHolder(containerView), LayoutContainer {

        ...
        fun bindItem(todoItem: TodoItem) {
            tvTodoTitle.text = todoItem.title // Still calls findViewById only once
            cbTodoDone.isChecked = false
        }
    }
}
```

---

---

```
// ... (imports from before)

import android.view.LayoutInflater
import android.view.ViewGroup

class RecyclerListAdapter(..) : .. {
    // ...
    override fun onCreateViewHolder(parent: ViewGroup, layoutId: Int): ViewHolder {
        val itemView: View = LayoutInflater.from(parent.context)
            .inflate(R.layout.todo_item, parent, false) // Creates a list item view
        return ViewHolder(itemView) // Creates a view holder for it
    }
}
```

---

---

```
override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    holder.bindItem(items[position]) // Populates the list item with to-do data
}
```

---

---

```
override fun getItemCount() = items.size
```

---

---

```
import android.support.v7.widget.RecyclerView
import android.view.*
import com.example.kudoo.R
import com.example.kudoo.model.TodoItem
import kotlinx.android.extensions.LayoutContainer
import kotlinx.android.synthetic.main.todo_item.*

class RecyclerListAdapter(
    private val items: MutableList<TodoItem>
) : RecyclerView.Adapter<RecyclerListAdapter.ViewHolder>() {

    override fun onCreateViewHolder(parent: ViewGroup, layoutId: Int): ViewHolder {
        val itemView: View = LayoutInflater.from(parent.context)
            .inflate(R.layout.todo_item, parent, false)
        return ViewHolder(itemView)
    }
}
```

---

```
override fun getItemCount() = items.size

override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    holder.bindItem(items[position])
}

class ViewHolder(
    override val containerView: View
) : RecyclerView.ViewHolder(containerView), LayoutContainer {

    fun bindItem(todoItem: TodoItem) {
        tvTodoTitle.text = todoItem.title
        cbTodoDone.isChecked = false
    }
}
```

---

---

```
// ... (imports from before)

import android.support.v7.widget.*

import com.example.kudoo.model.TodoItem

import com.example.kudoo.view.main.RecyclerListAdapter

import kotlinx.android.synthetic.main.activity_main.* // From Kotlin Android Ext.

import kotlinx.android.synthetic.main.content_main.* // From Kotlin Android Ext.

class MainActivity : AppCompatActivity() {

    // ...

    private fun setUpRecyclerView() = with(recyclerViewTodos) {
        adapter = RecyclerListAdapter(sampleData()) // Populates adapter/list with data
        layoutManager = LinearLayoutManager(this@MainActivity) // Uses linear layout
        itemAnimator = DefaultItemAnimator() // Optional layout niceties
        addItemDecoration(
            DividerItemDecoration(this@MainActivity, DividerItemDecoration.VERTICAL)
        )
    }

    private fun sampleData() = mutableListOf(
        TodoItem("Implement RecyclerView"),
        TodoItem("Store to-dos in database"),
        TodoItem("Delete to-dos on click")
    )
}
```

---

---

```
override fun onCreate(savedInstanceState: Bundle?) {  
    // ...  
    setUpRecyclerView()  
}
```

---

---

```
dependencies {
    // ...
    def room_version = "1.1.1" // Use latest version 1.x if you want
    implementation "android.arch.persistence.room:runtime:$room_version"
    kapt "android.arch.persistence.room:compiler:$room_version"
}
```

---

---

```
apply plugin: 'kotlin-android-extensions' // Should already exist
apply plugin: 'kotlin-kapt'           // Added now for annotation processing
```

---

---

```
import android.arch.persistence.room.Entity
import android.arch.persistence.room.PrimaryKey

@Entity(tableName = "todos")      // Indicates that this is a database entity
data class TodoItem(val title: String) {
    @PrimaryKey(autoGenerate = true) // Unique primary key must identify an object
    var id: Long = 0              // 0 is considered 'not set' by Room
}
```

---

---

```
import android.arch.persistence.room.*
import android.arch.persistence.room.OnConflictStrategy.IGNORE
import com.example.kudoo.model.TodoItem

@Dao
interface TodoItemDao {
    @Query("SELECT * FROM todos")
    fun loadAllTodos(): List<TodoItem> // Allows retrieving all to-do items

    @Insert(onConflict = IGNORE) // Does nothing if entry with ID already exists
    fun insertTodo(todo: TodoItem) // Allows inserting a new to-do item

    @Delete
    fun deleteTodo(todo: TodoItem) // Allows deleting an existing to-do item
}
```

---

---

```
import android.arch.persistence.room.*
import android.content.Context // Needs access to Android context to build DB object
import com.example.kudoo.model.TodoItem

@Database(entities = [TodoItem::class], version = 1) // TodoItem is only DB entity
abstract class AppDatabase : RoomDatabase() {

    companion object {
        private var INSTANCE: AppDatabase? = null

        fun getDatabase(ctx: Context): AppDatabase { // Builds and caches DB object
            if (INSTANCE == null) {
                INSTANCE = Room.databaseBuilder(ctx, AppDatabase::class.java, "AppDatabase")
                    .build()
            }
            return INSTANCE!!
        }
    }

    abstract fun todoItemDao(): TodoItemDao // Triggers Room to provide an impl.
}
```

---

---

```
// ... (imports from before)

import android.arch.persistence.db.SupportSQLiteDatabase
import kotlinx.coroutines.experimental.*

val DB = newSingleThreadContext("DB") // CoroutineContext for DB operations

@Database(entities = [TodoItem::class], version = 1)
abstract class AppDatabase : RoomDatabase() {

    companion object {
        private var INSTANCE: AppDatabase? = null

        fun getDatabase(ctx: Context): AppDatabase {
            if (INSTANCE == null) {
                INSTANCE = Room.databaseBuilder(ctx, AppDatabase::class.java, "AppDatabase")
                    .addCallback(prepopulateCallback(ctx)) // Adds callback to database
                    .build()
            }
            return INSTANCE!!
        }
    }
}
```

---

```
private fun prepopulateCallback(ctx: Context): Callback {
    return object : Callback() {
        override fun onCreate(db: SupportSQLiteDatabase) { // Uses onCreate callback
            super.onCreate(db)
            populateWithSampleData(ctx)
        }
    }
}

private fun populateWithSampleData(ctx: Context) { // Adds sample data to DB
    launch(DB) { // DB operations must be done on a background thread
        with(getDatabase(ctx).todoItemDao()) { // Uses DAO to insert items into DB
            insertTodo(TodoItem("Create entity"))
            insertTodo(TodoItem("Add a DAO for data access"))
            insertTodo(TodoItem("Inherit from RoomDatabase"))
        }
    }
}

abstract fun todoItemDao(): TodoItemDao
}
```

---

---

```
def coroutines_version = "0.24.0" // Use latest version if you want
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:$coroutines_version"
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:$coroutines_version"
```

---

---

```
// ... (imports from before)

import kotlinx.coroutines.experimental.android.UI
import kotlinx.coroutines.experimental.*
import com.example.kudoo.db.*

class MainActivity : AppCompatActivity() {

    private lateinit var db: AppDatabase // Stores an AppDatabase object

    override fun onCreate(savedInstanceState: Bundle?) {
        // ...
        db = AppDatabase.getDatabase(applicationContext)
        setUpRecyclerView() // Sets up recycler view *after* db reference is initialized
        // ...
    }
}
```

---

```
private fun setUpRecyclerView() = with(recyclerViewTodos) {
    launch {
        val todos = sampleData().toMutableList()
        withContext(UI) { adapter = RecyclerListAdapter(todos) } // Uses UI context
    }
    layoutManager = LinearLayoutManager(this@MainActivity)
    itemAnimator = DefaultItemAnimator()
    addItemDecoration(
        DividerItemDecoration(this@MainActivity, DividerItemDecoration.VERTICAL))
}


---


```

```
private suspend fun sampleData() =
    withContext(DB) { db.todoItemDao().loadAllTodos() } // Uses DB context
}
```

---

```
dependencies {
    // ...
    def lifecycle_version = "1.1.1" // Replace with latest version if you want
    implementation "android.arch.lifecycle:extensions:$lifecycle_version"
    kapt "android.arch.lifecycle:compiler:$lifecycle_version"
}
```

---

---

```
import android.app.Application
import android.arch.lifecycle.AndroidViewModel

class TodoViewModel(app: Application) : AndroidViewModel(app) { ... }
```

---

---

```
// ... (imports from before)

import com.example.kudoo.db.*
import com.example.kudoo.model.TodoItem
import kotlinx.coroutines.experimental.*

class TodoViewModel(app: Application) : AndroidViewModel(app) {
    private val dao by lazy { AppDatabase.getDatabase(getApplicationContext()).todoItemDao() }

    suspend fun getTodos(): MutableList<TodoItem> = withContext(DB) {
        dao.loadAllTodos().toMutableList()
    }

    fun add(todo: TodoItem) = launch(DB) { dao.insertTodo(todo) }

    fun delete(todo: TodoItem) = launch(DB) { dao.deleteTodo(todo) }
}
```

---

---

```
class MainActivity : AppCompatActivity() {

    private lateinit var viewModel: TodoViewModel // Now references view model, not DB

    override fun onCreate(savedInstanceState: Bundle?) {
        // ...
        viewModel = getViewModel(TodoViewModel::class) // 'getViewModel' is impl. next
        setUpRecyclerView()
    }

    private fun setUpRecyclerView() = with(recyclerViewTodos) {
        launch(UI) { adapter = RecyclerListAdapter(viewModel.getTodos()) }
        // ...
    }
}
```

---

---

```
import android.arch.lifecycle.*  
import android.support.v4.app.FragmentActivity  
import kotlin.reflect.KClass  
  
fun <T : ViewModel> FragmentActivity.getViewModel(modelClass: KClass<T>): T =  
    ViewModelProviders.of(this).get(modelClass.java)
```

---

---

```
// ... (imports from before)
import android.arch.lifecycle.LiveData

@Dao
interface TodoItemDao {
    // ...
    @Query("SELECT * FROM todos")
    fun loadAllTodos(): LiveData<List<TodoItem>> // Wraps return type in LiveData now
}
```

---

---

```
// ... (imports from before)
import android.arch.lifecycle.LiveData

class TodoViewModel(app: Application) : AndroidViewModel(app) {
    // Now uses a LiveData of a read-only list
    suspend fun getTodos(): LiveData<List<TodoItem>> = withContext(DB) {
        dao.loadAllTodos()
    }
    // ...
}
```

---

---

```
// ... (imports from before)

import android.arch.lifecycle.LiveData
import kotlinx.coroutines.experimental.android.UI

class MainActivity : AppCompatActivity() {
    // ...

    private fun setUpRecyclerView() { // No longer uses shorthand notation
        with(recyclerViewTodos) {
            adapter = RecyclerListAdapter(mutableListOf()) // Initializes with empty list
            // ...
        }

        launch(UI) { // Uses UI thread to access recycler view adapter
            val todosLiveData = viewModel.getTodos() // Runs in DB context
            todosLiveData.observe(this@MainActivity, Observer { todos ->
                // Observes changes in the LiveData
                todos?.let {
                    val adapter = (recyclerViewTodos.adapter as RecyclerListAdapter)
                    adapter.setItems(it) // Updates list items when data changes
                }
            })
        }
    }
}
```

---

---

```
class RecyclerListAdapter(
    private val items: MutableList<TodoItem>
) : RecyclerView.Adapter<RecyclerListAdapter.ViewHolder>() {
    // ...
    fun setItems(items: List<TodoItem>) {
        this.items.clear()
        this.items.addAll(items)
        notifyDataSetChanged() // Must notify recycler view of changes to the data
    }
}
```

---

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".view.add.AddTodoActivity">

    <EditText
        android:id="@+id/etNewTodo"
        android:hint="@string/enter_new_todo"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"

```

```
    android:layout_margin="@dimen/margin_medium"
    android:textAppearance="@android:style/TextAppearance.Medium"
    tools:text="@string/enter_new_todo"
    android:inputType="text" />

<Button
    android:id="@+id/btnAddTodo"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/add_to_do"
    android:textAppearance="@android:style/TextAppearance"
    android:layout_gravity="center_horizontal" />

</LinearLayout>
```

---

---

```
<android.support.design.widget.CoordinatorLayout ...>
    <!-- ... -->
    <android.support.design.widget.FloatingActionButton ...
        app:srcCompat="@drawable/ic_add" />
</android.support.design.widget.CoordinatorLayout>
```

---

---

```
// ... (imports from before)

import android.content.Intent
import com.example.kudoo.view.add.AddTodoActivity

class MainActivity : AppCompatActivity() {
    // ...
    private fun setUpFloatingActionButton() {
        fab.setOnClickListener {
            val intent = Intent(this, AddTodoActivity::class.java)
            startActivity(intent) // Switches to AddTodoActivity
        }
    }
}
```

---

---

```
class MainActivity : AppCompatActivity() {  
    // ...  
    override fun onCreate(savedInstanceState: Bundle?) {  
        // ...  
        setUpRecyclerView()  
        setUpFloatingActionButton()  
    }  
}
```

---

---

```
import android.os.Bundle
import android.support.v7.app.AppCompatActivity
import com.example.kudoo.R
import com.example.kudoo.db.DB
import com.example.kudoo.model.TodoItem
import com.example.kudoo.view.common.getViewModel
import com.example.kudoo.viewmodel.TodoViewModel
import kotlinx.android.synthetic.main.activity_add_todo.*
import kotlinx.coroutines.experimental.launch

class AddTodoActivity : AppCompatActivity() {

    private lateinit var viewModel: TodoViewModel // Uses the view model as well

    override fun onCreate(savedInstanceState: Bundle?) {
```

---

```
super.onCreate(savedInstanceState)
setContentView(R.layout.activity_add_todo)

viewModel = getViewModel(TodoViewModel::class)
setUpListeners()
}

private fun setUpListeners() { // Adds new to-do item to DB when clicking button
    btnAddTodo.setOnClickListener {
        val newTodo = etNewTodo.text.toString()
        launch(DB) { viewModel.add(TodoItem(newTodo)) } // Initiates DB transaction
        finish() // Switches back to MainActivity
    }
}
-----
```

---

```
<activity android:name=".view.add.AddTodoActivity"
    android:parentActivityName=".MainActivity">

    <meta-data android:name="android.support.PARENT_ACTIVITY"
        android:value="com.example.kudoo.MainActivity" />
</activity>
```

---

---

```
class RecyclerListAdapter(
    private val items: MutableList<TodoItem>,
    private val onItemCheckboxClicked: (TodoItem) -> Unit
) : RecyclerView.Adapter<RecyclerListAdapter.ViewHolder>() {
    // ...

    inner class ViewHolder(...) : ... { // Note that this is now an 'inner' class

        fun bindItem(todoItem: TodoItem) {
            // ...
            cbTodoDone.setOnCheckedChangeListener { _, _ -> // Adds listener to check box
                onItemCheckboxClicked(todoItem)
            }
        }
    }
}
```

---

---

```
// ... (imports from before)

import kotlinx.coroutines.experimental.android.UI

class MainActivity : AppCompatActivity() {

    // ...

    private fun setUpRecyclerView() {
        with(recyclerViewTodos) {
            adapter = RecyclerListAdapter(mutableListOf(), onRecyclerItemClick())
            // ...
        }
        // ...
    }

    private fun onRecyclerItemClick(): (TodoItem) -> Unit = { todo ->
        launch(DB) { viewModel.delete(todo) }
    }
}
```

---

---

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/navigation_home"
          android:icon="@drawable/ic_home_black_24dp"
          android:title="@string/title_home" />
    <item android:id="@+id/navigation_my_foods"
          android:icon="@drawable/ic_dashboard_black_24dp"
          android:title="@string/title_my_foods" />
</menu>
```

---

---

```
<string name="title_home">Home</string>
<string name="title_my_foods">My Foods</string>
```

---

---

```
class MainActivity : AppCompatActivity() {

    private val navListener = BottomNavigationView.OnNavigationItemSelectedListener {
        when(it.itemId) {
            R.id.navigation_home -> { // Defines action for when 'Home' is clicked
                return@OnNavigationItemSelectedListener true
            }
            R.id.navigation_my_foods -> { // Defines action for when 'My Foods' is clicked
                return@OnNavigationItemSelectedListener true
            }
        }
        false
    }
    // ...
}
```

---

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    app:layout_behavior="@string/appbar_scrolling_view_behavior" ...>

    <android.support.v7.widget.RecyclerView
        android:id="@+id/rvFoods"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintBottom_toTopOf="@+id/navigation" />

    <android.support.design.widget.BottomNavigationView
        android:id="@+id/navigation"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        app:layout_constraintBottom_toBottomOf="parent"
        android:background="?android:attr/windowBackground"
        app:menu="@menu/navigation" />

</android.support.constraint.ConstraintLayout>
```

---

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="@dimen/medium_padding">

    <TextView
        android:id="@+id/tvFoodName"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="@dimen/medium_font_size"
        app:layout_constraintRight_toLeftOf="@+id/ivStar"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:text="Gingerbread" />
```

---

```
<TextView
    android:id="@+id/tvFoodType"
    tools:text="Sweets and Candy"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintTop_toBottomOf="@+id/tvFoodName"
    app:layout_constraintStart_toStartOf="@+id/tvFoodName"
    app:layout_constraintRight_toLeftOf="@+id/ivStar"
    android:textColor="@color/lightGrey"
    android:textSize="@dimen/small_font_size" />

<ImageView
    android:id="@+id/ivStar"
    android:layout_width="32dp"
    android:layout_height="32dp"
    android:contentDescription="@string/content_description_star"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

</android.support.constraint.ConstraintLayout>
```

---

---

```
// In res/values/strings.xml
<string name="content_description_star">favorite</string>

// In res/values/dimens.xml
<dimen name="tiny_padding">4dp</dimen>
<dimen name="medium_padding">8dp</dimen>
<dimen name="medium_font_size">16sp</dimen>
<dimen name="small_font_size">13sp</dimen>

// In res/values/colors.xml
<color name="lightGrey">#888888</color>
```

---

---

```
data class Food(val name: String, val type: String, var isFavorite: Boolean = false)
```

---

---

```
androidExtensions {  
    experimental = true  
}
```

---

---

```
import android.support.v7.widget.RecyclerView
import android.view.*
import com.example.nutrilicious.R
import com.example.nutrilicious.model.Food
import kotlinx.android.extensions.LayoutContainer
import kotlinx.android.synthetic.main.rv_item.* // Imports synthetic properties

class SearchListAdapter(
    private var items: List<Food> // Uses a read-only list of items to display
) : RecyclerView.Adapter<ViewHolder>() {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        val view = LayoutInflater.from(parent.context) // Inflates layout to create view
            .inflate(R.layout.rv_item, parent, false)
        return ViewHolder(view) // Creates view holder that manages the list item view
    }
}
```

---

```
override fun getItemCount(): Int = items.size

override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    holder.bindTo(items[position]) // Binds data to a list item
}

// In this app, we'll usually replace all items so DiffUtil has little use
fun setItems(newItems: List<Food>) {
    this.items = newItems // Replaces whole list
    notifyDataSetChanged() // Notifies recycler view of data changes to re-render
}

inner class ViewHolder(
    override val containerView: View
) : RecyclerView.ViewHolder(containerView), LayoutContainer {

    fun bindTo(food: Food) { // Populates text views and star image to show a food
        tvFoodName.text = food.name
        tvFoodType.text = food.type
    }
}
```

```
    val image = if (food.isFavorite) {
        android.R.drawable.btn_star_big_on
    } else {
        android.R.drawable.btn_star_big_off
    }
    ivStar.setImageResource(image)
}
}
```

---

---

```
import android.support.v7.widget.*

class MainActivity : AppCompatActivity() {
    // ...
    private fun setUpSearchRecyclerView() = with(rvFoods) {
        adapter = SearchListAdapter(sampleData())
        layoutManager = LinearLayoutManager(this@MainActivity)
        addItemDecoration(DividerItemDecoration(
            this@MainActivity, LinearLayoutManager.VERTICAL
        ))
        setHasFixedSize(true)
    }
}
```

---

---

```
import com.example.nutrilicious.model.Food

class MainActivity : AppCompatActivity() {
    // ...
    private fun sampleData() = listOf( // Only temporary sample data, thus hard-coded
        Food("Gingerbread", "Candy and Sweets", false),
        Food("Nougat", "Candy and Sweets", true),
        Food("Apple", "Fruits and Vegetables", false),
        Food("Banana", "Fruits and Vegetables", true)
    )
}
```

---

---

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    setUpSearchRecyclerView()
    navigation.setOnNavigationItemSelectedListener(navListener)
}
```

---

---

```
dependencies {
    // ...
    def retrofit_version = "2.4.0"
    implementation "com.squareup.retrofit2:retrofit:$retrofit_version"
    implementation "com.squareup.retrofit2:converter-moshi:$retrofit_version"

    def okhttp_version = "3.6.0"
    implementation "com.squareup.okhttp3:logging-interceptor:$okhttp_version"
    implementation "com.squareup.okhttp3:okhttp:$okhttp_version"
}
```

---

---

```
import com.example.nutrilicious.BuildConfig

private const val API_KEY = BuildConfig.API_KEY
private const val BASE_URL = "https://api.nal.usda.gov/ndb/"
```

---

---

Nutrilicious\_UsdaApiKey = "<YOUR\_API\_KEY\_HERE>"

---

---

```
buildTypes {
    debug {
        buildConfigField 'String', "API_KEY", Nutrilicious_UsdaApiKey // From properties
    }
    release { ... }
}
```

---

---

```
import retrofit2.Retrofit
import retrofit2.converter.moshi.MoshiConverterFactory
// ...
private fun buildClient(): Retrofit = Retrofit.Builder() // Builds Retrofit object
    .baseUrl(BASE_URL)
    .client(buildHttpClient())
    .addConverterFactory(MoshiConverterFactory.create()) // Uses Moshi for JSON
    .build()
```

---

---

```
import okhttp3.OkHttpClient
import java.util.concurrent.TimeUnit
// ...
private fun buildHttpClient(): OkHttpClient = OkHttpClient.Builder()
    .connectTimeout(30, TimeUnit.SECONDS)
    .readTimeout(30, TimeUnit.SECONDS)
    .addInterceptor(loggingInterceptor()) // Logs API responses to Logcat
    .addInterceptor(apiKeyInterceptor()) // Adds API key to request URLs
    .build()
```

---

---

```
import okhttp3.logging.HttpLoggingInterceptor
// ...
private fun loggingInterceptor() = HttpLoggingInterceptor().apply {
    level = if (BuildConfig.DEBUG) {
        HttpLoggingInterceptor.Level.BODY // Only does logging in debug mode
    } else {
        HttpLoggingInterceptor.Level.NONE // Otherwise no logging
    }
}
```

---

---

```
import okhttp3.Interceptor
// ...
private fun injectqueryParams(
    vararg params: Pair<String, String>
): Interceptor = Interceptor { chain ->

    val originalRequest = chain.request()
    val urlWithParams = originalRequest.url().newBuilder()
        .apply { params.forEach { addQueryParameter(it.first, it.second) } }
        .build()
    val newRequest = originalRequest.newBuilder().url(urlWithParams).build()

    chain.proceed(newRequest)
}
```

---

---

```
private fun apiKeyInterceptor() = injectqueryParams(  
    "api_key" to API_KEY  
)
```

---

---

```
import okhttp3.ResponseBody
import retrofit2.Call
import retrofit2.http.*

interface UsdaApi {

    @GET("search?format=json")           // Is appended to the base URL
    fun getFoods(
        @Query("q") searchTerm: String,    // Only non-optional parameter
        @Query("sort") sortBy: Char = 'r', // Sorts by relevance by default
        @Query("ds") dataSource: String = "Standard Reference",
        @Query("offset") offset: Int = 0
    ): Call<ResponseBody>           // Allows to retrieve raw JSON for now
}
```

---

[https://api.nal.usda.gov/ndb/search?format=json&q=raw&api\\_key=<YOUR\\_API\\_KEY>&sort=r&ds=Standard%20Reference&offset=0](https://api.nal.usda.gov/ndb/search?format=json&q=raw&api_key=<YOUR_API_KEY>&sort=r&ds=Standard%20Reference&offset=0).

---

---

```
private val usdaClient by lazy { buildClient() }

val usdaApi: UsdaApi by lazy { usdaClient.create(UsdaApi::class.java) } // Public
```

---

---

```
<manifest ...>
  <uses-permission android:name="android.permission.INTERNET" />
  <application ...>...</application>
</manifest>
```

---

---

```
def coroutines_version = "0.24.0" // Latest version may differ slightly in use
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:$coroutines_version"
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:$coroutines_version"
```

---

---

```
import kotlinx.coroutines.newFixedThreadPoolContext

val NETWORK = newFixedThreadPoolContext(2, "NETWORK") // Dedicated network context
```

---

---

```
import com.example.nutrilicious.data.network.*
import com.example.nutrilicious.model.Food
import kotlinx.android.synthetic.main.activity_main.*
import kotlinx.coroutines.launch
// ...
class MainActivity : AppCompatActivity() { // ...
    override fun onCreate(savedInstanceState: Bundle?) {
        // ...
        launch(NETWORK) {
            usdaApi.getFoods("raw").execute() // Logs results to Logcat due to interceptor
        }
    }
}
```

---

---

```
def moshi_version = "1.6.0"
implementation "com.squareup.moshi:moshi:$moshi_version"
kapt "com.squareup.moshi:moshi-kotlin-codegen:$moshi_version"
```

---

---

---

```
apply plugin: 'kotlin-kapt'
```

---

---

---

```
{  
  "list": {  
    "q": "raw",  
    "ds": "Standard Reference",  
    "sort": "I",  
    ...  
    "item": [  
      {  
        "offset": 0,  
        "group": "Vegetables and Vegetable Products", // Type of the food  
        "name": "Coriander (cilantro) leaves, raw", // Food title  
        "ndbno": "11165", // Unique identifier  
        "ds": "SR",  
        "manu": "none"  
      },  
      // More items here...  
    ]  
  }  
}
```

---

---

```
import com.squareup.moshi.JsonClass

@JsonClass(generateAdapter = true)
class ListWrapper<T> {
    var list: T? = null // Navigates down the 'list' object
}

@JsonClass(generateAdapter = true)
class ItemWrapper<T> {
    var item: T? = null // Navigates down the 'item' array
}

typealias SearchWrapper<T> = ListWrapper<ItemWrapper<T>>
```

---

---

```
@JsonClass(generateAdapter = true)
class FoodDto { // Uses lateinit for properties that must be populated by Moshi
    lateinit var ndbno: String
    lateinit var name: String
    lateinit var group: String
}
```

---

---

```
import com.example.nutrilicious.data.network.dto.*

fun getFoods(@Query("q") searchTerm: String, ...): Call<SearchWrapper<List<FoodDto>>>
```

---

---

```
data class Food(  
    val id: String, // New property  
    val name: String,  
    val type: String,  
    var isFavorite: Boolean = false  
) {  
    constructor(dto: FoodDto) : this(dto.ndbno, dto.name, dto.group) // Maps from DTO  
}
```

---

---

```
import kotlinx.coroutines.android.UI
import kotlinx.coroutines.withContext
// ...
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        // ...
        launch(NETWORK) {
            val dtos = usdaApi.getFoods("raw").execute()?.body()?.list?.item!!
            val foods: List<Food> = dtos.map(::Food) // Maps all DTOs to Food models

            withContext(UI) { // Must use main thread to access UI elements
                (rvFoods.adapter as SearchListAdapter).setItems(foods)
            }
        }
    }
}
```

---

---

```
adapter = SearchListAdapter(emptyList())
```

---

---

```
def room_version = "1.1.0"
implementation "android.arch.persistence.room:runtime:$room_version"
kapt "android.arch.persistence.room:compiler:$room_version"

def lifecycle_version = "1.1.1"
implementation "android.arch.lifecycle:extensions:$lifecycle_version"
```

---

---

```
import android.arch.lifecycle.ViewModel
import com.example.nutrilicious.data.network.dto.*
import retrofit2.Call

class SearchViewModel : ViewModel() {

    private fun doRequest(req: Call<SearchWrapper<List<FoodDto>>>): List<FoodDto> =
        req.execute().body()?.list?.item ?: emptyList()
}
```

---

---

```
import com.example.nutrilicious.data.network.*
import com.example.nutrilicious.model.Food
import kotlinx.coroutines.withContext

class SearchViewModel : ViewModel() {

    suspend fun getFoodsFor(searchTerm: String): List<Food> { // Fetches foods from API
        val request: Call<SearchWrapper<List<FoodDto>>> = usdaApi.getFoods(searchTerm)
        val foodDtos: List<FoodDto> = withContext(NETWORK) { doRequest(request) }
        return foodDtos.map(:Food)
    }
    // ...
}
```

---

---

```
import com.example.nutrilicious.view.common.getViewModel // Created next

class MainActivity : AppCompatActivity() {

    private lateinit var searchViewModel: SearchViewModel
    // ...
    override fun onCreate(savedInstanceState: Bundle?) {
        // ...
        navigation.setOnNavigationItemSelectedListener(navListener)
        searchViewModel = getViewModel(SearchViewModel::class)
        // ...
    }
    // ...
}
```

---

---

```
import android.arch.lifecycle.*  
import android.support.v4.app.FragmentActivity  
import kotlin.reflect.KClass  
  
fun <T : ViewModel> FragmentActivity.getViewModel(modelClass: KClass<T>): T {  
    return ViewModelProviders.of(this).get(modelClass.java)  
}
```

---

---

```
override fun onCreate(savedInstanceState: Bundle?) {
    // ...
    searchViewModel = getViewModel(SearchViewModel::class)

    launch(NETWORK) { // Uses network dispatcher for network call
        val foods = searchViewModel.getFoodsFor("raw")

        withContext(UI) { // Populates recycler view with fetched foods (on main thread)
            (rvFoods.adapter as SearchListAdapter).setItems(foods)
        }
    }
}
```

---

---

```
private fun updateListFor(searchTerm: String) {
    launch(NETWORK) {
        val foods = searchViewModel.getFoodsFor(searchTerm)

        withContext(UI) {
            (rvFoods.adapter as SearchListAdapter).setItems(foods)
        }
    }
}
```

---

---

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:app="http://schemas.android.com/apk/res-auto"
      xmlns:android="http://schemas.android.com/apk/res/android">

    <item android:id="@+id/search"
          android:title="@string/search_title"
          android:icon="@android:drawable/ic_menu_search"
          app:showAsAction="always"
          app:actionViewClass="android.widget.SearchView" />

</menu>
```

---

---

```
<string name="search_title">Search food...</string>
```

---

---

```
<?xml version="1.0" encoding="utf-8"?>
<searchable xmlns:android="http://schemas.android.com/apk/res/android"
    android:label="@string/app_name"
    android:hint="@string/search_title" />
```

---

---

```
<activity
    android:launchMode="singleTop"          // Reuses existing instance
    android:name=".view.main.MainActivity"
    android:label="@string/app_name">
    <meta-data android:name="android.app.searchable" // Where to find searchable conf.
        android:resource="@xml/searchable" />
    <intent-filter>
        <action android:name="android.intent.action.SEARCH" /> // Handles search intents
    </intent-filter>
    ...
</activity>
```

---

---

```
import android.app.SearchManager
import android.widget.SearchView
import android.content.Context
import android.view.Menu
// ...
class MainActivity : AppCompatActivity() {
    // ...
    override fun onCreateOptionsMenu(menu: Menu): Boolean {
        menuInflater.inflate(R.menu.search_menu, menu)

        // Associates searchable configuration with the SearchView
        val searchManager = getSystemService(Context.SEARCH_SERVICE) as SearchManager
        (menu.findItem(R.id.search).actionView as SearchView).apply {
            setSearchableInfo(searchManager.getSearchableInfo(componentName))
        }
        return true
    }
}
```

---

---

```
import android.content.Intent
// ...
class MainActivity : AppCompatActivity() {
    // ...
    override fun onNewIntent(intent: Intent) {
        if (intent.action == Intent.ACTION_SEARCH) { // Filters for search intents
            val query = intent.getStringExtra(SearchManager.QUERY)
            updateListFor(query)
        }
    }
}
```

---

---

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.widget.SwipeRefreshLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/swipeRefresh"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <android.support.v7.widget.RecyclerView
        android:id="@+id/rvFoods"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"/>

</android.support.v4.widget.SwipeRefreshLayout>
```

---

---

```
<android.support.constraint.ConstraintLayout ...>

    <!-- Placeholder for fragments -->
    <FrameLayout
        android:id="@+id/mainView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical" />

    <android.support.design.widget.BottomNavigationView ... />

</android.support.constraint.ConstraintLayout>
```

---

---

```
import android.content.Context
import android.os.Bundle
import android.support.v4.app.Fragment
import android.view.*
import com.example.nutrilicious.R
import com.example.nutrilicious.view.common.getViewModel
import com.example.nutrilicious.viewmodel.SearchViewModel

class SearchFragment : Fragment() {

    private lateinit var searchViewModel: SearchViewModel

    override fun onAttach(context: Context?) {
        super.onAttach(context)
        searchViewModel = getViewModel(SearchViewModel::class)
    }

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?): View? {
        return inflater.inflate(R.layout.fragment_search, container, false)
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        setUpSearchRecyclerView() // Will come from MainActivity
        setUpSwipeRefresh() // Implemented later
    }
}
```

---

---

```
fun <T : ViewModel> Fragment.getViewModel(modelClass: KClass<T>): T {  
    return ViewModelProviders.of(this).get(modelClass.java)  
}
```

---

---

```
fun updateListFor(searchTerm: String) { // Is now public
    launch(NETWORK) { // ...
        withContext(UI) {
            (rvFoods?.adapter as? SearchListAdapter)?.setItems(foods) // Uses safe ops.
        }
    }
}
```

---

---

```
private fun setUpSearchRecyclerView() = with(rvFoods) {
    adapter = SearchListAdapter(emptyList())
    layoutManager = LinearLayoutManager(context)
    addItemDecoration(DividerItemDecoration(
        context, LinearLayoutManager.VERTICAL
    ))
    setHasFixedSize(true)
}
```

---

---

```
class SearchFragment : Fragment() {

    private var lastSearch = ""

    // ...

    private fun setUpSwipeRefresh() {
        swipeRefresh.setOnRefreshListener {
            updateListFor(lastSearch) // Re-issues last search on swipe refresh
        }
    }

    fun updateListFor(searchTerm: String) {
        lastSearch = searchTerm // Remembers last search term
        // ...
    }
}
```

---

---

```
class MainActivity : AppCompatActivity() {  
    private lateinit var searchFragment: SearchFragment  
    // ...  
}
```

---

---

```
import android.support.v7.app.AppCompatActivity
// ...
fun AppCompatActivity.replaceFragment(viewGroupId: Int, fragment: Fragment) {
    supportFragmentManager.beginTransaction()
        .replace(viewGroupId, fragment) // Replaces given view group with fragment
        .commit()
}
```

---

---

```
import com.example.nutrilicious.view.common.replaceFragment

class MainActivity : AppCompatActivity() {
    // ...
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        searchFragment = SearchFragment()
        replaceFragment(R.id.mainView, searchFragment) // Replaces the placeholder
        navigation.setOnNavigationItemSelectedListener(navListener)
    }
}
```

---

---

```
import android.support.annotation.IdRes
// ...
fun AppCompatActivity.addFragmentToState(
    @IdRes containerViewId: Int,
    fragment: Fragment,
    tag: String
) {
    supportFragmentManager.beginTransaction()
        .add(containerViewId, fragment, tag) // Stores fragment with given tag
        .commit()
}
```

---

---

```
import com.example.nutrilicious.view.common.*
// ...
class MainActivity : AppCompatActivity() {
    // ...
    private fun recoverOrBuildSearchFragment() {
        val fragment = supportFragmentManager // Tries to load fragment from state
            .findFragmentByTag(SEARCH_FRAGMENT_TAG) as? SearchFragment
        if (fragment == null) setUpSearchFragment() else searchFragment = fragment
    }

    private fun setUpSearchFragment() { // Sets up search fragment and stores to state
        searchFragment = SearchFragment()
        addFragmentToState(R.id.mainView, searchFragment, SEARCH_FRAGMENT_TAG)
    }
}
```

---

---

```
private const val SEARCH_FRAGMENT_TAG = "SEARCH_FRAGMENT"
```

---

```
class MainActivity : AppCompatActivity() { ... }
```

---

---

```
override fun onCreate(savedInstanceState: Bundle?) {
    // ...
    recoverOrBuildSearchFragment() // Replaces SearchFragment() constructor call
    replaceFragment(R.id.mainView, searchFragment)
}
```

---

---

```
override fun onNewIntent(intent: Intent) {
    if (intent.action == Intent.ACTION_SEARCH) {
        val query = intent.getStringExtra(SearchManager.QUERY)
        searchFragment.updateListFor(query) // Uses the search fragment
    }
}
```

---

---

```
private fun updateListFor(searchTerm: String) {
    lastSearch = searchTerm
    swipeRefresh?.isRefreshing = true // Indicates that app is loading

    launch(NETWORK) {
        val foods = searchViewModel.getFoodsFor(searchTerm)
        withContext(UI) {
            (rvFoods?.adapter as? SearchListAdapter)?.setItems(foods)
            swipeRefresh?.isRefreshing = false // Indicates that app finished loading
        }
    }
}
```

---

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/tvHeadline"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

```
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    android:padding="@dimen/medium_padding"
    android:text="@string/favorites"
    android:textSize="@dimen/huge_font_size" />

<android.support.v7.widget.RecyclerView
    android:id="@+id/rvFavorites"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    app:layout_constraintTop_toBottomOf="@+id/tvHeadline" />

</android.support.constraint.ConstraintLayout>
```

---

---

```
// In res/values/dimens.xml
<dimen name="huge_font_size">22sp</dimen>

// In res/values/strings.xml
<string name="favorites">Favorite Foods</string>
```

---

---

```
import android.os.Bundle
import android.support.v4.app.Fragment
import android.support.v7.widget.*
import android.view.*
import com.example.nutrilicious.R
import com.example.nutrilicious.model.Food
import kotlinx.android.synthetic.main.fragmentFavorites.*

class FavoritesFragment : Fragment() {

    override fun onCreateView(inflater: LayoutInflater,
                           container: ViewGroup?,
                           savedInstanceState: Bundle?): View? {
        return inflater.inflate(R.layout.fragmentFavorites, container, false)
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        setUpRecyclerView()
    }
}
```

---

```
}

private fun setUpRecyclerView() = with(rvFavorites) {
    adapter = SearchListAdapter(sampleData())
    layoutManager = LinearLayoutManager(context)
    addItemDecoration(DividerItemDecoration(
        context, LinearLayoutManager.VERTICAL
    ))
    setHasFixedSize(true)
}
```

```
// Temporary! Should use string resources instead of hard-coded strings in general
private fun sampleData(): List<Food> = listOf(
    Food("00001", "Marshmallow", "Candy and Sweets", true),
    Food("00002", "Nougat", "Candy and Sweets", true),
    Food("00003", "Oreo", "Candy and Sweets", true)
)
```

---

---

```
import android.support.v7.widget.*
import com.example.nutrilicious.model.Food
// ...
class MainActivity : AppCompatActivity() {
    // ...
    companion object {
        fun setUpRecyclerView(rv: RecyclerView, list: List<Food> = emptyList()) {
            with(rv) {
                adapter = SearchListAdapter(list)
                layoutManager = LinearLayoutManager(context)
                addItemDecoration(DividerItemDecoration(
                    context, LinearLayoutManager.VERTICAL
                ))
                setHasFixedSize(true)
            }
        }
    }
}
```

---

---

```
// In FavoritesFragment.kt
private fun setUpRecyclerView() {
    MainActivity.setUpRecyclerView(rvFavorites, sampleData())
}
```

---

```
// In SearchFragment.kt
private fun setUpSearchRecyclerView() {
    MainActivity.setUpRecyclerView(rvFoods)
}
```

---

---

```
private val handler = BottomNavigationView.OnNavigationItemSelectedListener {  
    when (it.itemId) {  
        R.id.navigation_home -> {  
            replaceFragment(R.id.mainView, searchFragment) // Uses existing search fragment  
            return@OnNavigationItemSelected true  
        }  
        R.id.navigation_my_foods -> {  
            replaceFragment(R.id.mainView, FavoritesFragment()) // Creates new fragment  
            return@OnNavigationItemSelected true  
        }  
    }  
    false  
}
```

---

---

```
import com.example.nutrilicious.model.Food

class SearchFragment : Fragment() {
    // ...
    private var lastResults = emptyList<Food>()

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        // ...
        (rvFoods?.adapter as? SearchListAdapter)?.setItems(lastResults) // Recovers state
    }
    // ...
    fun updateListFor(searchTerm: String) {
        // ...
        launch(NETWORK) {
            val foods = searchViewModel.getFoodsFor(searchTerm)
            lastResults = foods // Remembers last search results
            withContext(UI) { ... }
        }
    }
}
```

---

---

```
import android.arch.persistence.room.*
// ...
@Entity(tableName = "favorites") // Signals Room to map this entity to the DB
data class Food(
    @PrimaryKey val id: String, // Unique identifier for a food (the NDBNO)
    val name: String,
    val type: String,
    var isFavorite: Boolean = false
) { ... }
```

---

---

```
import android.arch.lifecycle.LiveData
import android.arch.persistence.room.*
import android.arch.persistence.room.OnConflictStrategy.IGNORE
import com.example.nutrilicious.model.Food

@Dao
interface FavoritesDao {

    @Query("SELECT * FROM favorites")
    fun loadAll(): LiveData<List<Food>> // Note LiveData return type

    @Query("SELECT id FROM favorites")
    fun loadAllIds(): List<String>

    @Insert(onConflict = IGNORE) // Do nothing if food with same NDBNO already exists
    fun insert(food: Food)

    @Delete
    fun delete(food: Food)
}
```

---

---

```
import android.arch.persistence.room.*
import android.content.Context
import com.example.nutrilicious.model.Food

@Database(entities = [Food::class], version = 1)
abstract class AppDatabase : RoomDatabase() {

    companion object {
        private var INSTANCE: AppDatabase? = null

        fun getInstance(ctx: Context): AppDatabase {
            if (INSTANCE == null) { INSTANCE = buildDatabase(ctx) }
            return INSTANCE!!
        }

        private fun buildDatabase(ctx: Context) = Room
            .databaseBuilder(ctx, AppDatabase::class.java, "AppDatabase")
            .build()
    }

    abstract fun favoritesDao(): FavoritesDao // Provides access to the DAO
}
```

---

---

```
import android.app.Application
import android.arch.lifecycle.*
import com.example.nutrilicious.data.db.*
import com.example.nutrilicious.model.Food
import kotlinx.coroutines.*

class FavoritesViewModel(app: Application) : AndroidViewModel(app) {

    private val dao by lazy { AppDatabase.getInstance(getApplicationContext()).favoritesDao()}

    suspend fun getFavorites(): LiveData<List<Food>> = withContext(DB) {
        dao.loadAll()
    }

    suspend fun getAllIds(): List<String> = withContext(DB) { dao.loadAllIds() }

    fun add(favorite: Food) = launch(DB) { dao.insert(favorite) }

    fun delete(favorite: Food) = launch(DB) { dao.delete(favorite) }

}
```

---

---

```
import kotlinx.coroutines.newSingleThreadContext

val DB = newSingleThreadContext("DB") // Single dedicated thread for DB operations
```

---

---

```
class SearchListAdapter( // ...
    private val onStarClick: (Food, Int) -> Unit
) : ... {
    // ...
    inner class ViewHolder(...) : ... {
        fun bindTo(food: Food) {
            // ...
            ivStar.setOnClickListener { onStarClick(food, this.layoutPosition) }
        }
    }
}
```

---

---

```
class MainActivity : AppCompatActivity() {  
    // ...  
    fun setUpRecyclerView(rv: RecyclerView, list: List<Food> = emptyList()) {  
        with(rv) {  
            adapter = setUpSearchListAdapter(rv, list)  
            // ...  
        }  
    }  
  
    private fun setUpSearchListAdapter(rv: RecyclerView, items: List<Food>) =  
        SearchListAdapter(items,  
            onStarClick = { food, layoutPosition -> // Toggles favorite on click  
                toggleFavorite(food)  
                rv.adapter.notifyItemChanged(layoutPosition)  
            })  
}
```

---

---

```
import com.example.nutrilicious.viewmodel.FavoritesViewModel

class MainActivity : AppCompatActivity() {
    // ...
    private lateinit var favoritesViewModel: FavoritesViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        // ...
        favoritesViewModel = getViewModel(FavoritesViewModel::class)
    }
    // ...
    private fun toggleFavorite(food: Food) {
```

---

```
val wasFavoriteBefore = food.isFavorite
food.isFavorite = food.isFavorite.not() // Adjusts Food object's favorite status

if (wasFavoriteBefore) {
    favoritesViewModel.delete(food)
    toast("Removed ${food.name} from your favorites.")
} else {
    favoritesViewModel.add(food)
    toast("Added ${food.name} as a new favorite of yours!")
}
}
```

---

---

```
import android.widget.Toast
// ...
fun AppCompatActivity.toast(msg: String) {
    Toast.makeText(this, msg, Toast.LENGTH_SHORT).show()
}
```

---

---

```
import android.content.Context
import com.example.nutrilicious.view.common.getViewModel
import com.example.nutrilicious.viewmodel.FavoritesViewModel
import kotlinx.coroutines.android.UI
import kotlinx.coroutines.launch
import android.arch.lifecycle.Observer
// ...
class FavoritesFragment : Fragment() {

    private lateinit var favoritesViewModel: FavoritesViewModel

    override fun onAttach(context: Context?) {
        super.onAttach(context)
        favoritesViewModel = getViewModel(FavoritesViewModel::class)
    }
    // ...
}
```

---

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    // ...
    observeFavorites()
}

private fun observeFavorites() = launch { // Updates list when favorites change
    val favorites = favoritesViewModel.getFavorites()
    favorites.observe(this@FavoritesFragment, Observer { foods ->
        foods?.let {
            launch(UI) { (rvFavorites.adapter as? SearchListAdapter)?.setItems(foods) }
        }
    })
}

private fun setUpRecyclerView() {
    (activity as? MainActivity)?.setUpRecyclerView(rvFavorites, emptyList())
}
```

---

---

```
import com.example.nutrilicious.viewmodel.FavoritesViewModel
// ...
class SearchFragment : Fragment() {
    // ...
    private lateinit var favoritesViewModel: FavoritesViewModel

    override fun onAttach(context: Context?) {
        // ...
        favoritesViewModel = getViewModel(FavoritesViewModel::class)
    }
    // ...
    private fun updateListFor(searchTerm: String) {
        lastSearch = searchTerm
        swipeRefresh?.isRefreshing = true

        launch {
            val favoritesIds: List<String> = favoritesViewModel.getAllIds()
            val foods: List<Food> = searchViewModel.getFoodsFor(searchTerm)
                .onEach { if (favoritesIds.contains(it.id)) it.isFavorite = true }
            lastResults = foods

            withContext(UI) { ... }
        }
    }

    private fun setUpSearchRecyclerView() {
        (activity as? MainActivity)?.setUpRecyclerView(rvFoods)
    }
}
```

---

---

```
@GET("V2/reports?format=json")
fun getDetails(
    @Query("ndbno") id: String,           // Only non-optional parameter is food ID
    @Query("type") detailsType: Char = 'b' // b = basic, f = full, s = stats
): Call<DetailsWrapper<DetailsDto>>
```

---

```
{  
  "foods": [  
    {  
      "food": {  
        "sr": "Legacy",  
        "type": "b",  
        "desc": {  
          "ndbno": "09070",  
          "name": "Cherries, sweet, raw",  
          "ds": "Standard Reference",  
          "manu": "",  
          "ru": "g"  
        },  
        "nutrients": [  
          // Nutrient data for food above  
        ]  
      }  
    ]  
  ]  
}
```

```
{  
    "nutrient_id": "255", // Unique nutrient ID  
    "name": "Water", // Nutrient name  
    "derivation": "NONE",  
    "group": "Proximates", // Nutrient category  
    "unit": "g", // Unit used by 'value' below  
    "value": "82.25", // Amount of this nutrient per 100g of food  
    "measures": [ ... ]  
}, ...  
]  
}  
}  
]  
}  
-----
```

---

```
import com.squareup.moshi.JsonClass

@JsonClass(generateAdapter = true)
class FoodsWrapper<T> {
    var foods: List<T> = listOf() // Navigates down the 'foods' list from JSON
}

@JsonClass(generateAdapter = true)
class FoodWrapper<T> {
    var food: T? = null // Navigates down the 'food' object
}

typealias DetailsWrapper<T> = FoodsWrapper<FoodWrapper<T>>
```

---

---

```
@JsonClass(generateAdapter = true)
class DetailsDto(val desc: DescriptionDto, val nutrients: List<NutrientDto>) {
    init {
        nutrients.forEach { it.detailsId = desc.ndbno } // Connects the two DTOs below
    }
}
```

```
@JsonClass(generateAdapter = true)
class DescriptionDto { // Property names must match JSON names
    lateinit var ndbno: String
    lateinit var name: String
}
```

```
@JsonClass(generateAdapter = true)
class NutrientDto { // Property names must match JSON names
    var nutrient_id: Int? = null // Cannot use lateinit with Int
    var detailsId: String? = null // Only field not coming from JSON
    lateinit var name: String
    lateinit var unit: String
    var value: Float = 0f
    lateinit var group: String
}
```

---

---

```
import com.example.nutrilicious.data.network.dto.*
```

```
data class FoodDetails(  
    val id: String,  
    val name: String,  
    val nutrients: List<Nutrient>  
) {  
    constructor(dto: DetailsDto) : this(  
        dto.desc.ndbno,  
        dto.desc.name,  
        dto.nutrients.map(:Nutrient)  
    )  
}
```

```
data class Nutrient(  
    val id: Int,  
    val detailsId: String,  
-----
```

```
    val name: String,  
    val amountPer100g: Float,  
    val unit: String,  
    val type: NutrientType  
)  
{  
    constructor(dto: NutrientDto) : this(  
        dto.nutrient_id!!,  
        dto.detailsId!!,  
        dto.name,  
        dto.value,  
        dto.unit,  
        NutrientType.valueOf(dto.group.toUpperCase())  
    )  
}  
  
enum class NutrientType {  
    PROXIMATES, MINERALS, VITAMINS, LIPIDS, OTHER  
}
```

---

---

```
import com.example.nutrilicious.data.network.*
import kotlinx.coroutines.launch

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        // ...
        launch(NETWORK) { usdaApi.getDetails("09070").execute() } // Temporary
    }
}
```

---

---

```
<?xml version="1.0" encoding="utf-8"?>
<View xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="@dimen/divider_height"
    android:minHeight="@dimen/divider_minheight"
    android:layout_marginTop="@dimen/divider_margin"
    android:layout_marginBottom="@dimen/divider_margin"
    android:background="?android:attr/listDivider" />
```

---

---

```
<dimen name="divider_height">2dp</dimen>
<dimen name="divider_minheight">1px</dimen>
<dimen name="divider_margin">5dp</dimen>
```

---

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:padding="@dimen/medium_padding"
        tools:context=".view.detail.FoodDetailsActivity">

        <TextView
            android:id="@+id/tvFoodName"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_gravity="center"
            android:gravity="center"
            android:padding="@dimen/medium_padding"
            android:textSize="@dimen/huge_font_size" />
    
```

```
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/proximates"
    android:textColor="@android:color/darker_gray"
    android:textSize="@dimen/medium_font_size" />
```

```
<TextView
    android:id="@+id/tvProximates"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:lineSpacingMultiplier="1.1" />
```

```
<include layout="@layout/horizontal_divider" />
```

```
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/vitamins"
    android:textColor="@android:color/darker_gray"
    android:textSize="@dimen/medium_font_size" />
```

---

```
<TextView
    android:id="@+id/tvVitamins"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />

<include layout="@layout/vertical_divider" />

<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/minerals"
    android:textColor="@android:color/darker_gray"
    android:textSize="@dimen/medium_font_size" />

<TextView
    android:id="@+id/tvMinerals"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />

<include layout="@layout/vertical_divider" />
```

---

```
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/lipids"
    android:textColor="@android:color/darker_gray"
    android:textSize="@dimen/medium_font_size" />

<TextView
    android:id="@+id/tvLipids"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />

</LinearLayout>

</ScrollView>
```

---

---

```
<string name="proximates">Proximates</string>
<string name="minerals">Minerals</string>
<string name="vitamins">Vitamins</string>
<string name="lipids">Lipids</string>
```

---

---

```
class SearchListAdapter(...,  
    private val onItemClick: (Food) -> Unit,  
    private val onStarClick: (Food, Int) -> Unit  
) : RecyclerView.Adapter<ViewHolder>() {  
    // ...  
  
    inner class ViewHolder(...) : ... {  
  
        fun bindTo(food: Food) {  
            // ...  
            containerView.setOnClickListener { onItemClick(food) }  
        }  
    }  
}
```

---

---

```
import com.example.nutrilicious.view.details.DetailsActivity
// ...
class MainActivity : AppCompatActivity() {
    private fun setUpSearchListAdapter(rv: RecyclerView, items: List<Food>) =
        SearchListAdapter(items,
            onItemClick = { startDetailsActivity(it) },
            onStarClick = { ... }
    )

    private fun startDetailsActivity(food: Food) {
        val intent = Intent(this, DetailsActivity::class.java).apply {
            putExtra(FOOD_ID_EXTRA, food.id) // Stores the desired food's ID in the Intent
        }
        startActivity(intent) // Switches to DetailsActivity
    }
}
```

---

---

```
const val FOOD_ID_EXTRA = "NDBNO"
```

---

```
class DetailsActivity : AppCompatActivity() { ... }
```

---

---

```
import android.arch.lifecycle.ViewModel
import com.example.nutrilicious.data.network.*
import com.example.nutrilicious.data.network.dto.*
import com.example.nutrilicious.model.FoodDetails
import kotlinx.coroutines.withContext
import retrofit2.Call

class DetailsViewModel : ViewModel() {

    suspend fun getDetails(foodId: String): FoodDetails? {
        val request: Call<DetailsWrapper<DetailsDto>> = usdaApi.getDetails(foodId)

        val detailsDto: DetailsDto = withContext(NETWORK) {
            request.execute().body()?.foods?.get(0)?.food // Runs request and extracts data
        } ?: return null

        return FoodDetails(detailsDto)
    }
}
```

---

---

```
import android.support.v7.app.AppCompatActivity
import android.os.Bundle
import com.example.nutrilicious.R
import com.example.nutrilicious.view.common.getViewModel
import com.example.nutrilicious.viewmodel.DetailsViewModel

class DetailsActivity : AppCompatActivity() {

    private lateinit var detailsViewModel: DetailsViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        // ...
        detailsViewModel = getViewModel(DetailsViewModel::class)
    }
}
```

---

---

```
import kotlinx.coroutines.android.UI
import kotlinx.coroutines.*
// ...
class DetailsActivity : AppCompatActivity() {
    // ...
    override fun onCreate(savedInstanceState: Bundle?) {
        // ...
        val foodId = intent.getStringExtra(FOOD_ID_EXTRA) // Reads out desired food's ID
        updateUiWith(foodId)
    }

    private fun updateUiWith(foodId: String) {
        if (foodId.isBlank()) return

        launch {
            val details = detailsViewModel.getDetails(foodId) // Retrieves details
            withContext(UI) { bindUi(details) } // Populates UI
        }
    }
}
```

---

---

```
import com.example.nutrilicious.model.*
import kotlinx.android.synthetic.main.activity_details.*
// ...
class DetailsActivity : AppCompatActivity() {
    // ...
    private fun bindUi(details: FoodDetails?) {
        if (details != null) {
            tvFoodName.text = "${details.name} (100g)"
            tvProximates.text = makeSection(details, NutrientType.PROXIMATES)
            tvMinerals.text = makeSection(details, NutrientType.MINERALS)
            tvVitamins.text = makeSection(details, NutrientType.VITAMINS)
            tvLipids.text = makeSection(details, NutrientType.LIPIDS)
        } else {
            // ...
        }
    }
}
```

---

```
        tvFoodName.text = getString(R.string.no_data)
    }
}

private fun makeSection(details: FoodDetails, forType: NutrientType) =
    details.nutrients.filter { it.type == forType }
        .joinToString(separator = "\n", transform = ::renderNutrient)

private fun renderNutrient(nutrient: Nutrient): String = with(nutrient) {
    val displayName = name.substringBefore(",") // = whole name if it has no comma
    "$displayName: $amountPer100g$unit"
}
-----
```

---

```
<string name="no_data">No data available</string>
```

---

---

```
<application ...>  
    ...  
    <activity android:name=".view.details.DetailsActivity"  
        android:parentActivityName=".view.main.MainActivity">  
        <meta-data  
            android:name="android.support.PARENT_ACTIVITY"  
            android:value=".view.main.MainActivity" />  
    </activity>  
</application>
```

---

---

```
import android.arch.persistence.room.*
// ...
@Entity(tableName = "details")
@TypeConverters(NutrientListConverter::class) // Is implemented next
data class FoodDetails(
    @PrimaryKey val id: String,
    // ...
) { constructor(dto: DetailsDto) : this(..) }

@TypeConverters(NutrientTypeConverter::class)
data class Nutrient(..) { .. }
```

---

---

```
import android.arch.persistence.room.TypeConverter
import com.example.nutrilicious.model./*
import com.squareup.moshi./*
class NutrientListConverter {

    private val moshi = Moshi.Builder().build()

    private val nutrientList = Types.newParameterizedType( // Represents List<Nutrient>
        List::class.java, Nutrient::class.java
    )

    private val adapter = moshi.adapter<List<Nutrient>>(nutrientList) // Builds adapter

    @TypeConverter
    fun toString(nutrient: List<Nutrient>): String = adapter.toJson(nutrient)

    @TypeConverter fun toListOfNutrient(json: String): List<Nutrient>
        = adapter.fromJson(json) ?: emptyList()
}
```

---

---

```
class NutrientTypeConverter {  
  
    @TypeConverter  
    fun toString(nutrientType: NutrientType) = nutrientType.name // Type -> String  
  
    @TypeConverter  
    fun toNutrientType(name: String) = NutrientType.valueOf(name) // String -> Type  
}
```

---

---

```
import android.arch.persistence.room.*
import com.example.nutrilicious.model.FoodDetails

@Dao
interface DetailsDao {

    @Query("SELECT * FROM details WHERE id = :ndbno")
    fun loadById(ndbno: String): FoodDetails?

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun insert(food: FoodDetails)
}
```

---

---

```
import com.example.nutrilicious.model.FoodDetails
// ...
@Database(entities = [Food::class, FoodDetails::class], version = 2) // Version 2
abstract class AppDatabase : RoomDatabase() {
    // ...
    abstract fun favoritesDao(): FavoritesDao
    abstract fun detailsDao(): DetailsDao // Now exposes a DetailsDao as well
}
```

---

---

```
import com.example.nutrilicious.BuildConfig
// ...
private fun buildDatabase(ctx: Context) = Room
    .databaseBuilder(ctx, AppDatabase::class.java, "AppDatabase")
    .apply { if (BuildConfig.DEBUG) fallbackToDestructiveMigration() }
    .build()
```

---

---

```
import android.content.Context
import com.example.nutrilicious.data.db.*
import com.example.nutrilicious.data.network.*
import com.example.nutrilicious.data.network.dto.*
import com.example.nutrilicious.model.FoodDetails
import kotlinx.coroutines.*
import retrofit2.Call

class DetailsRepository(ctx: Context) {

    private val detailsDao by lazy { AppDatabase.getInstance(ctx).detailsDao() }

    fun add(details: FoodDetails) = launch(DB) { detailsDao.insert(details) }
```

---

```
suspend fun getDetails(id: String): FoodDetails? {
    return withContext(DB) { detailsDao.loadById(id) } // Prefers database
    ?: withContext(NETWORK) { fetchDetailsFromApi(id) } // Falls back to network
    .also { if (it != null) this.add(it) } // Adds newly fetched foods to DB
}

private suspend fun fetchDetailsFromApi(id: String): FoodDetails? {
    val request: Call<DetailsWrapper<DetailsDto>> = usdaApi.getDetails(id)
    val detailsDto: DetailsDto = withContext(NETWORK) {
        request.execute().body()?.foods?.get(0)?.food // Same as before
    } ?: return null

    return FoodDetails(detailsDto)
}
}
```

---

---

```
import com.example.nutrilicious.data.DetailsRepository
// ...
class DetailsViewModel(app: Application) : AndroidViewModel(app) {
    private val repo = DetailsRepository(app)
    suspend fun getDetails(foodId: String): FoodDetails? = repo.getDetails(foodId)
}
```

---

---

```
data class Amount(val value: Double, val unit: WeightUnit)
```

```
enum class WeightUnit {  
    GRAMS, MILLIGRAMS, MICROGRAMS, KCAL, IU  
}
```

---

---

```
import com.example.nutrilicious.model.WeightUnit.*

internal val RDI = mapOf(
    255 to Amount(3000.0, GRAMS),      // water
    208 to Amount(2000.0, KCAL),      // energy
    203 to Amount(50.0, GRAMS),      // protein
    204 to Amount(78.0, GRAMS),      // total fat (lipids)
    205 to Amount(275.0, GRAMS),      // carbohydrates
    291 to Amount(28.0, GRAMS),      // fiber
    269 to Amount(50.0, GRAMS),      // sugars
    301 to Amount(1300.0, MILLIGRAMS), // calcium
    303 to Amount(13.0, MILLIGRAMS), // iron
    304 to Amount(350.0, MILLIGRAMS), // magnesium
    305 to Amount(700.0, MILLIGRAMS), // phosphorus
    306 to Amount(4700.0, MILLIGRAMS), // potassium
```

---

```
307 to Amount(1500.0, MILLIGRAMS), // sodium
309 to Amount(10.0, MILLIGRAMS), // zinc
401 to Amount(85.0, MILLIGRAMS), // vitamin c
404 to Amount(1200.0, MICROGRAMS), // vitamin b1 (thiamin)
405 to Amount(1200.0, MICROGRAMS), // vitamin b2 (riboflavin)
406 to Amount(15.0, MILLIGRAMS), // vitamin b3 (niacin)
415 to Amount(1300.0, MICROGRAMS), // vitamin b6 (pyridoxine)
435 to Amount(400.0, MICROGRAMS), // folate
418 to Amount(3.0, MICROGRAMS), // vitamin b12 (cobalamine)
320 to Amount(800.0, MICROGRAMS), // vitamin a
323 to Amount(15.0, MILLIGRAMS), // vitamin e (tocopherol)
328 to Amount(15.0, MICROGRAMS), // vitamin d (d2 + d3)
438 to Amount(105.0, MICROGRAMS), // vitamin k
606 to Amount(20.0, GRAMS), // saturated fats
605 to Amount(0.0, GRAMS), // transfats
601 to Amount(300.0, MILLIGRAMS) // cholesterol
)
```

---

---

```
@TypeConverters(NutrientTypeConverter::class)
data class Nutrient(
    // ...
    val amountPer100g: Amount, // Combines amount and unit into single property
    // ...
) {
    constructor(dto: NutrientDto) : this(
        // ...
        Amount(dto.value.toDouble(), WeightUnit.fromString(dto.unit)),
        // ...
    )
}
```

---

---

```
enum class WeightUnit {
    GRAMS, MILLIGRAMS, MICROGRAMS, KCAL, IU; // Mind the semicolon

    companion object {
        fun fromString(unit: String) = when(unit) { // Transforms string to weight unit
            "g" -> WeightUnit.GRAMS
            "mg" -> WeightUnit.MILLIGRAMS
            "\u00b5g" -> WeightUnit.MICROGRAMS
            "kcal" -> WeightUnit.KCAL
            "IU" -> WeightUnit.IU
            else -> throw IllegalArgumentException("Unknown weight unit: $unit")
        }
    }

    override fun toString(): String = when(this) { // Transforms weight unit to string
        WeightUnit.GRAMS -> "g"
        WeightUnit.MILLIGRAMS -> "mg"
        WeightUnit.MICROGRAMS -> "\u00b5g"
        WeightUnit.KCAL -> "kcal"
        WeightUnit.IU -> "IU"
    }
}
```

---

---

```
private fun renderNutrient(nutrient: Nutrient): String = with(nutrient) {  
    val name = name.substringBefore(",")  
    val amount = amountPer100g.value.render()  
    val unit = amountPer100g.unit  
    val percent = getPercentOfRdi(nutrient).render() // Is implemented next  
    val rdiNote = if (percent.isNotEmpty()) "($percent% of RDI)" else ""  
    "$name: $amount$unit $rdiNote"  
}
```

---

```
private fun Double.render() = if (this >= 0.0) "%.2f".format(this) else ""
```

---

---

```
private fun getPercentOfRdi(nutrient: Nutrient): Double {
    val nutrientAmount: Double = nutrient.amountPer100g.normalized() // Impl. next
    val rdi: Double = RDI[nutrient.id]?.normalized() ?: return -1.0

    return nutrientAmount / rdi * 100
}
```

---

---

```
data class Amount(val value: Double, val unit: WeightUnit) {  
  
    fun normalized() = when(unit) { // Normalizes milligrams and micrograms to grams  
        GRAMS, KCAL, IU -> value  
        MILLIGRAMS -> value / 1000.0  
        MICROGRAMS -> value / 1_000_000.0  
    }  
}
```

---

---

```
private fun updateListFor(searchTerm: String) = launch {
    // ...
    withContext(Dispatchers.Main) {
        (rvFoods?.adapter as? SearchListAdapter)?.setItems(foods)
        swipeRefresh?.isRefreshing = false
    }
    if (foods.isEmpty() && isAdded) {
        snackbar("No foods found")
    }
}
```

---

---

```
import android.support.design.widget.Snackbar
import android.view.View

fun Fragment.snackbar(
    msg: String, view: View = activity!!.findViewById<View>(android.R.id.content)) {
    Snackbar.make(view, msg, Snackbar.LENGTH_SHORT).show()
}
```

---

---

```
<ScrollView ...>

    <ProgressBar
        android:id="@+id/progress"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:visibility="gone"
        style="?android:attr/progressBarStyle" />
```

```
<LinearLayout android:id="@+id/content" ...>
    <!-- sections as before -->
</LinearLayout>
```

---

```
</ScrollView>
```

---

```
import android.view.View
// ...
class DetailsActivity : AppCompatActivity() {
    // ...
    private fun updateUiWith(foodId: String) {
        if (foodId.isBlank()) return
        setLoading(true) // Indicates that app is loading
        launch {
            val details = detailsViewModel.getDetails(foodId)
            withContext(UI) {
                setLoading(false) // Indicates that app finished loading
                bindUi(details)
            }
        }
    }

    private fun setLoading(isLoading: Boolean) {
        if (isLoading) {
            content.visibility = View.GONE
            progress.visibility = View.VISIBLE
        } else {
            progress.visibility = View.GONE
            content.visibility = View.VISIBLE
        }
    }
}
```

---

---

```
user {
  username = "johndoe"
  birthday = 1 January 1984
  address {
    street = "Main Street"
    number = 42
    postCode = "12345"
    city = "New York"
  }
}
```

---

---

```
fun user(init: User.() -> Unit): User {
    val user = User()
    user.init()
    return user
}
```

---

---

```
fun user(init: User.() -> Unit) = User().apply(init)
```

---

---

```
import java.time.LocalDate

data class User(
    var username: String = "",
    var birthday: LocalDate? = null,
    var address: Address? = null
)
```

---

---

```
data class User(...) {  
    fun address(init: Address.() -> Unit) {  
        address = Address().apply(init)  
    }  
}
```

```
data class Address(  
    var street: String = "",  
    var number: Int = -1,  
    var postCode: String = "",  
    var city: String = ""  
)
```

---

---

```
user {
  address {
    username = "this-should-not-work"
    user {
      address {
        birthday = LocalDate.of(1984, Month.JANUARY, 1)
      }
    }
  }
}
```

---

---

```
import java.time.LocalDate

data class User(val username: String, val birthday: LocalDate, val address: Address)

data class Address(
    val street: String,
    val number: Int,
    val postCode: String,
    val city: String
)
```

---

---

```
class UserBuilder {

    var username = "" // Gets assigned directly in DSL => public
    var birthday: LocalDate? = null // Gets assigned directly in DSL => public
    private var address: Address? = null // Is built via builder => private

    fun address(init: AddressBuilder.() -> Unit) { // Nested function to build address
        address = AddressBuilder().apply(init).build()
    }

    fun build(): User { // Validates data and builds user object
        val theBirthday = birthday
        val theAddress = address
        if (username.isBlank() || theBirthday == null || theAddress == null)
            throw IllegalStateException("Please set username, birthday, and address.")

        return User(username, theBirthday, theAddress)
    }
}
```

---

---

```
class AddressBuilder {

    var street = ""
    var number = -1
    var postCode = ""
    var city = ""

    fun build(): Address {
        if (notReady())
            throw IllegalStateException("Please set street, number, postCode, and city.")

        return Address(street, number, postCode, city)
    }

    private fun notReady()
        = arrayOf(street, postCode, city).any { it.isBlank() } || number <= 0
}
```

---

---

```
fun user(init: UserBuilder.() -> Unit) = UserBuilder().apply(init).build()
```

---

---

```
data class User(..., val addresses: List<Address>)

class UserBuilder {
    // ...
    private val addresses: MutableList<Address> = mutableListOf()

    fun address(init: AddressBuilder.() -> Unit) {
        addresses.add(AddressBuilder().apply(init).build())
    }

    fun build(): User { ... }
}
```

---

---

```
user {
    username = "johndoe"
    birthday = LocalDate.of(1984, Month.JANUARY, 1)
    addresses { // New dedicated addresses block
        address { // All address blocks must be placed here
            street = "Main Street"
            number = 42
            postCode = "12345"
            city = "New York"
        }
        address {
            street = "Plain Street"
            number = 1
            postCode = "54321"
            city = "York"
        }
    }
}
```

---

---

```
class UserBuilder {  
    // ...  
    private val addresses: MutableList<Address> = mutableListOf()  
  
    inner class Addresses : ArrayList<Address>() {  
        fun address(init: AddressBuilder.() -> Unit) {  
            add(AddressBuilder().apply(init).build())  
        }  
    }  
  
    fun addresses(init: Addresses.() -> Unit) { // 'Addresses' is the receiver now  
        addresses.addAll(Addresses().apply(init))  
    }  
  
    fun build(): User {  
        val theBirthday = birthday  
        if (username.isBlank() || theBirthday == null || addresses.isEmpty()) throw ...  
  
        return User(username, theBirthday, addresses)  
    }  
}
```

---

---

```
@Deprecated("Out of scope", ReplaceWith(""), DeprecationLevel.ERROR)
fun user(init: UserBuilder.() -> Unit): Nothing = error("Cannot access user() here.")
```

---

```
fun user(init: (@UserDsl UserBuilder).() -> Unit)
    = UserBuilder().apply(init).build()
```

```
@DslMarker
@Target(AnnotationTarget.CLASS, AnnotationTarget.TYPE) // Can be used on types
@Retention(AnnotationRetention.SOURCE)
annotation class UserDsl
```

---

```
user {
  // ...
  val usercity = "New York"
  addresses {
    address {
      // ...
      city = usercity
    }
    address {
      // ...
      city = usercity
    }
  }
}
```

---

---

```
infix fun Int.January(year: Int) = LocalDate.of(year, Month.JANUARY, this)
```

---

```
user {  
    username = "johndoe"  
    birthday = 1 January 1984  
    // ...  
}
```

---

---

```
@UserDsl
class UserBuilder {
    // ...
    val addresses: MutableList<Address> = mutableListOf()

    inner class Addresses : ArrayList<Address>() {
        inline fun address(init: AddressBuilder.() -> Unit) { ... } // Now inlined
    }

    inline fun addresses(init: Addresses.() -> Unit) { ... } // Now inlined
    // ...
}

inline fun user(init: UserBuilder.() -> Unit) = // Now inlined
    UserBuilder().apply(init).build()
```

---

---

```
class AddTodoActivity : AppCompatActivity() {
    // ...
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(createView()) // No inflating of an XML layout
        viewModel = getViewModel(TodoViewModel::class)
    }

    private fun createView(): View {
        val linearLayout = LinearLayout(this).apply { // Sets up the linear layout
            orientation = LinearLayout.VERTICAL
        }
        val etNewTodo = EditText(this).apply { // Sets up the EditText
            hint = getString(R.string.enter_new_todo)
            textAppearance = android.R.style.TextAppearance_Medium
            layoutParams = ViewGroup.LayoutParams(
                ViewGroup.LayoutParams.MATCH_PARENT,
                ViewGroup.LayoutParams.WRAP_CONTENT
            )
        }
        // ...
    }
}
```

---

```
    }

    val btnAddTodo = Button(this).apply { // Sets up the Button
        text = getString(R.string.add_to_do)
        textAppearance = android.R.style.TextAppearance
        layoutParams = LinearLayout.LayoutParams(
            ViewGroup.LayoutParams.WRAP_CONTENT,
            ViewGroup.LayoutParams.WRAP_CONTENT
        ).apply { gravity = Gravity.CENTER_HORIZONTAL }
        setOnClickListener {
            val newTodo = etNewTodo.text.toString()
            launch(DB) { viewModel.add(TodoItem(newTodo)) }
            finish()
        }
    }

    return linearLayout.apply { // Adds views to the linear layout and returns it
       .addView(etNewTodo)
       .addView(btnAddTodo)
    }
}
```

---

---

```
def anko_version = "0.10.5"
implementation "org.jetbrains.anko:anko:$anko_version" // Includes all of Anko
```

---

---

```
implementation "org.jetbrains.anko:anko-sdk25:$anko_version"
implementation "org.jetbrains.anko:anko-sdk25-coroutines:$anko_version"
```

---

---

```
verticalLayout {  
    button {  
        text = "Receive reward"  
        onClick { toast("So rewarding!") }  
    }  
}
```

---

---

```
verticalLayout {  
    button { ... }.LayoutParams(width = matchParent) {  
        margin = dip(5)  
    }  
}
```

---

```
class ExampleComponent : AnkoComponent<MainActivity> {  
    override fun createView(ui: AnkoContext<MainActivity>): View = with(ui) {  
        verticalLayout {  
            button { ... }.lparams(width = matchParent) { ... }  
        }  
    }  
}
```

---

```
class AddTodoActivity : AppCompatActivity() {
    // ...
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(createView()) // Still no inflating of an XML layout
        viewModel = getViewModel(TodoViewModel::class)
    }

    private fun createView() = verticalLayout { // Sets up vertical linear layout

        val etNewTodo = editText { // Sets up EditText and adds it to the linear layout
            hintResource = R.string.enter_new_todo
            textAppearance = android.R.style.TextAppearance_Medium
        }.lparams(width = matchParent, height = wrapContent) {
            margin = dip(16)
        }
    }
}
```

---

```
    }

    button(R.string.add_to_do) { // Sets up Button and adds it to the linear layout
        textAppearance = android.R.style.TextAppearance
    }.LayoutParams(width = wrapContent, height = wrapContent) {
        gravity = Gravity.CENTER_HORIZONTAL
    }.setOnClickListener { // Could also use onClick inside button {...} instead
        val newTodo = etNewTodo.text.toString()
        launch(DB) { viewModel.add(TodoItem(newTodo)) }
        finish()
    }
}
}
```

---

---

```
class AddTodoActivity : AppCompatActivity() {  
    // ...  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(AddTodoActivityUi().createView(AnkoContext.create(ctx, this)))  
        viewModel = getViewModel(TodoViewModel::class)  
    }  
  
    private inner class AddTodoActivityUi : AnkoComponent<AddTodoActivity> {  
  
        override fun createView(ui: AnkoContext<AddTodoActivity>): View = with(ui) {  
  
            verticalLayout {  
                val etNewTodo = editText {  
                    hintResource = R.string.enter_new_todo  
                    textAppearance = android.R.style.TextAppearance_Medium  
                }.lparams(width = matchParent, height = wrapContent) {  
                    margin = dip(16)  
                }  
            }  
        }  
    }  
}
```

---

```
        }
```

```
        button(R.string.add_to_do) {
            textAppearance = android.R.style.TextAppearance
        }.LayoutParams(width = wrapContent, height = wrapContent) {
            gravity = Gravity.CENTER_HORIZONTAL
        }.setOnClickListener {
            val newTodo = etNewTodo.text.toString()
            launch(DB) { viewModel.add(TodoItem(newTodo)) }
            finish()
        }
    }
}
```

---

---

```
import android.content.Context
import android.util.AttributeSet
import android.widget.FrameLayout

class SquareFrameLayout(
    context: Context,
    attributes: AttributeSet? = null,
    defStyleAttr: Int = 0
) : FrameLayout(context, attributes, defStyleAttr) {

    override fun onMeasure(widthMeasureSpec: Int, heightMeasureSpec: Int) {
        super.onMeasure(widthMeasureSpec, widthMeasureSpec) // Equal width and height
    }
}
```

---

---

```
import android.view.ViewManager
import org.jetbrains.anko.custom.ankoView

inline fun ViewManager.squareFrameLayout(init: SquareFrameLayout.() -> Unit) =
    ankoView({ SquareFrameLayout(it) }, theme = 0, init = init)
```

---

---

```
buildscript {
    ext.kotlin_version = '1.2.50' // Extra that stores Kotlin version
    repositories {
        google()
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.1.3'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
}
```

---

---

```
buildscript {
    extra["kotlin_version"] = "1.2.50"
    repositories {
        jcenter()
        google()
    }
    dependencies {
        classpath("com.android.tools.build:gradle:3.1.3")
        classpath("org.jetbrains.kotlin:kotlin-gradle-plugin:${extra["kotlin_version"]}")
    }
}
```

---

---

```
task clean(type: Delete) {
    delete rootProject.buildDir
}
```

---

---

```
task<Delete>("clean") {
    delete(rootProject.buildDir)
}
```

---

---

```
plugins {
    id("com.android.application")
    id("kotlin-android")
    id("kotlin-android-extensions")
    id("kotlin-kapt")
}
```

---

---

```
    android {
        compileSdkVersion(27)
        defaultConfig {
            applicationId = "com.example.nutrilicious"
            minSdkVersion(19)
            targetSdkVersion(27)
            versionCode = 1
            versionName = "1.0"
            testInstrumentationRunner = "android.support.test.runner.AndroidJUnitRunner"
        }
        buildTypes {
            getByName("release") {
                isMinifyEnabled = false
                proguardFiles("proguard-rules.pro")
            }
        }
    }
```

---

---

```
dependencies {

    val kotlin_version: String by rootProject.extra // Uses extra from root script
    implementation("org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version")
    // ...

    val moshi_version = "1.6.0"
    implementation("com.squareup.moshi:moshi:$moshi_version")
    kapt("com.squareup.moshi:moshi-kotlin-codegen:$moshi_version")
    // ...

    testImplementation("junit:junit:4.12")
    androidTestImplementation("com.android.support.test:runner:1.0.2")
    androidTestImplementation("com.android.support.test.espresso:espresso-core:3.0.2")
}
```

---

---

```
androidExtensions {  
    configure(delegateClosureOf<AndroidExtensionsExtension> { // Injects Groovy code  
        isExperimental = true  
    })  
}
```

---

---

```
private const val kotlinVersion = "1.2.50"
private const val androidGradleVersion = "3.1.3"

private const val supportVersion = "27.1.1"
private const val constraintLayoutVersion = "1.1.0"
// All versions as in build.gradle.kts...

object BuildPlugins {
    val androidGradle = "com.android.tools.build:gradle:$androidGradleVersion"
    val kotlinGradlePlugin = "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlinVersion"
}

object Android {
    val buildToolsVersion = "27.0.3"
}
```

---

```
val minSdkVersion = 19
val targetSdkVersion = 27
val compileSdkVersion = 27
val applicationId = "com.example.nutrilicious"
val versionCode = 1
val versionName = "1.0"
}

object Libs {
    val kotlin_std = "org.jetbrains.kotlin:kotlin-stdlib:$kotlinVersion"
    val appcompat = "com.android.support:appcompat-v7:$supportVersion"
    val design = "com.android.support:design:$supportVersion"
    // All dependencies as in build.gradle.kts...
}
```

---

---

```
plugins {
    `kotlin-dsl` // Uses ticks: ``
}
```

---

---

```
    android {  
        // ...  
        targetSdkVersion(Android.targetSdkVersion) // Uses the 'Android' object  
        versionCode = Android.versionCode  
        // ...  
    }  
  
    dependencies {  
        // ...  
        implementation(Libs.moshi) // Uses the 'Libs' object  
        kapt(Libs.moshi_codegen)  
        // ...  
    }  
-----
```

---

```
val person: Person? = getPersonOrNull()

if (person != null) {
    person.getSpouseOrNull()           // Let's say this returns null
} else {
    println("No person found (if/else)") // Not printed
}

person?.let {
    person.getSpouseOrNull()           // Let's say this returns null
} ?: println("No person found (let)") // Printed (because left-hand side is null!)
```

---