

Kotlin In-Depth

[Vol. I]

*A Comprehensive Guide to
Modern Multi-Paradigm Language*

by
Aleksei Sedunov



FIRST EDITION 2020

Copyright © BPB Publications, India

ISBN: 978-93-89328-585

All Rights Reserved. No part of this publication may be reproduced or distributed in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author's & publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners.

Distributors:

BPB PUBLICATIONS

20, Ansari Road, Darya Ganj

New Delhi-110002

Ph: 23254990/23254991

MICRO MEDIA

Shop No. 5, Mahendra Chambers,

150 DN Rd. Next to Capital Cinema,

V.T. (C.S.T.) Station, MUMBAI-400 001

Ph: 22078296/22078297

DECCAN AGENCIES

4-3-329, Bank Street,

Hyderabad-500195

Ph: 24756967/24756400

BPB BOOK CENTRE

376 Old Lajpat Rai Market,

Delhi-110006

Ph: 23861747

Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002
and Printed by him at Repro India Ltd, Mumbai

Dedicated to

*Tatiana, my guiding light
and the incessant source of inspiration*

About the Author

Aleksei Sedunov has been working as a Java developer since 2008. After joining JetBrains in 2012 he's been actively participating in the Kotlin language development focusing on IDE tooling for the IntelliJ platform. Currently he's working in a DataGrip team, a JetBrains Database IDE, carrying on with using Kotlin as a main development tool.

Acknowledgement

Above all others I would like to give my gratitude to the entire Kotlin team at JetBrains which has created such a beautiful language and continues to relentlessly work on its improvement – especially Andrey Breslav who's been leading the language design from the very first day.

I'm really grateful to everyone at BPB publications for giving me this splendid opportunity for writing this book and lending a tremendous support in improving the text before it gets to the readers.

Last but not least I'd like to thank my beloved family for their support throughout the work on the book.

- Aleksei Sedunov

Preface

Since its first release in 2016 (and even long before that) Kotlin has been gaining in popularity as a powerful and flexible tool in a multitude of development tasks being equally well-equipped for dealing with mobile, desktop and server-side applications finally getting its official acknowledgment from Google in 2017 and 2018 as a primary language for Android development. This popularity is well-justified since language pragmatism, the tendency to choose the best practice among known solutions was one of the guiding principles of its design.

With the book you're holding in your hands I'd like to invite you to the beautiful world of Kotlin programming where you can see its benefits for yourself. After completing this book you'll have all the necessary knowledge to write in Kotlin on your own.

The first volume deals with the fundamentals of Kotlin language such as its basic syntax, procedural, object-oriented and functional programming aspects as well as the Kotlin Standard Library. The material is divided into 9 chapters organized as follows:

Chapter 1 explains key ideas behind the language design, gives an overview of the Kotlin ecosystem and tooling and guides the reader through first steps required to set up a Kotlin project in various environments.

Chapter 2 introduces the reader to the Kotlin syntax, explains how to use variables and describes simple data types such as integers or boolean values as well their built-in operations. On top of that it addresses the basics of more complex data structures such as strings and arrays.

Chapter 3 discusses the syntax of Kotlin functions and explains the uses of various control structures supported by Kotlin such as binary/multiple choice, iteration and error handling. Additionally it addresses the matter of using packages for code structuring.

Chapter 4 introduces the reader to the basic aspects of object-oriented programming in Kotlin. It explains how to create and initialize a class instance and how to control member access, describe the use of object declarations and non-trivial kinds of properties and brings up the concept of type nullability.

Chapter 5 explains the functional aspects of Kotlin and introduces the reader to the idea of higher-order and anonymous functions, addresses the uses of inline functions

and explain how one can add features to existing types using extension functions and properties.

Chapter 6 explains the use of special kinds of classes tailored at specific programming tasks: data classes for simple data holders, enumerations for representing a fixed set of instances and inline classes for creating lightweight wrappers.

Chapter 7 continues the treatment of object-oriented features introduced in chapters 4 and 6 focusing on the idea of a class hierarchy. It explains how to define subclasses, how to use abstract classes and interfaces and how to restricted hierarchies using sealed classes.

Chapter 8 describes a major part of the Kotlin standard library which is concerned with various collection types and their operations as well as utilities simplifying file access and stream-based I/O.

Chapter 9 introduces the idea of generic declarations and explain how to define and use generic classes, functions and properties in Kotlin. It also explains the notion of variance and how it can be used to improve flexibility of your generic code.

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors if any, occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Table of Contents

1. Kotlin: Powerful and Pragmatic

Structure

Objective

What is Kotlin?

Safe

Multiparadigm

Concise and expressive

Interoperable

Multiplatform

Kotlin Ecosystem

Coroutines

Testing

Android Development

Web Development

Desktop applications

Getting started with Kotlin

Setting up an IntelliJ project

Using REPL

Kotlin Playground

Setting up an Eclipse project

Conclusion

2. Language Fundamentals

Structure

Objective

Basic syntax

Comments

Defining a variable

Identifiers

Mutable variables

Expressions and operators

Basic types

Integer types
Floating-Point types
Arithmetic operations
Bitwise operations
Char type
Numeric conversions
Boolean type and logical operations
Comparison and equality

Strings

String templates
Basic String operations

Arrays

Constructing an array
Using arrays

Conclusion

3. Defining Functions

Structure

Objective

Functions

The Anatomy of a Kotlin Function
Positional versus named arguments
Overloading and default values
Varargs
Function scope and visibility

Packages and imports

The packages and directory structure
Using import directives

Conditionals

Making decisions with if statements
Ranges, progressions, and in operation
When statements and multiple choice

Loops

The while/do-while loop
Iterables and for loop
Changing loop control-flow: break and continue

- Nested loops and labels*
- Tail-recursive functions*
- Exception handling
 - Throwing an exception*
 - Handling errors with try statements*
- Conclusion
- Questions

4. Working with Classes and Objects

- Structure
- Objective
- Defining a class
 - A class anatomy*
 - Constructors*
 - Member visibility*
 - Nested classes*
 - Local classes*
- Nullability
 - Nullable types*
 - Nullability and smart casts*
 - Not-null assertion operator*
 - Safe call operator*
 - Elvis operator*
- Properties: beyond simple variables
 - Top-level properties*
 - Late initialization*
 - Using custom accessors*
 - Lazy properties and delegates*
- Objects
 - Object declarations*
 - Companion objects*
 - Object expressions*
- Conclusion
- Questions

5. Leveraging Advanced Functions and Functional Programming

Structure

Objective

Functional programming in Kotlin

Higher-order functions

Functional types

Lambdas and anonymous functions

Callable references

Inline functions and properties

Non-local control flow

Extensions

Extension functions

Extension properties

Companion extensions

Lambdas and functional types with receiver

Callable references with receiver

Scope functions

run / with functions

run without context

apply / also functions

Extensions as class members

Conclusion

Questions

6. Using Special-Case Classes

Structure

Objective

Enum classes

Exhaustive when expressions

Declaring enums with custom members

Using common members of enum classes

Data classes

Data classes and their operations

Destructuring declarations

Inline classes

Defining an inline class

Unsigned integers

Conclusion

Questions

7. Exploring Collections and I/O

Structure

Objective

Collections

Collection types

Iterables

Collections, Lists, and Sets

Sequences

Maps

Comparables and Comparators

Creating a Collection

Basic operations

Accessing collection elements

Collective conditions

Aggregation

Filtering

Transformation

Extracting subcollections

Ordering

Files and I/O streams

Stream utilities

Creating streams

URL utilities

Accessing file content

File system utilities

Conclusion

Questions

8. Understanding Class Hierarchies

Structure

Objective

Inheritance

Declaring a subclass

- Subclass initialization*
- Type checking and casts*
- Common methods*
- Abstract classes and interfaces
 - Abstract classes and members*
 - Interfaces*
 - Sealed classes*
 - Delegation*
- Conclusion
- Questions

9. Generics

- Structure
- Objective
- Type parameters
 - Generic declarations*
 - Bounds and constraints*
 - Type erasure and reification*
- Variance
 - Variance: distinguishing producers and consumers*
 - Variance at the declaration site*
 - Use-site variance with projections*
 - Star projections*
- Type aliases
- Conclusion
- Questions

CHAPTER 1

Kotlin: Powerful and Pragmatic

This chapter is meant to explain the major features that make Kotlin an excellent and efficient language for modern application development and why you would want to learn it. We'll learn the basic ideas which stand behind the Kotlin design, get an overview of Kotlin libraries and frameworks for different application areas, such as Android applications, concurrency, testing and web development. In conclusion, we'll guide you through the steps required to set up a Kotlin project in two popular development environments, IntelliJ IDEA and Eclipse, and introduce you to the interactive Kotlin shell.

Structure

- What is Kotlin?
- Major components of Kotlin ecosystem
- Setting up a Kotlin project in IntelliJ and Eclipse

Objective

At the end of the chapter you'll get an understanding of the basic principles and ecosystem of Kotlin as well as how a simple Kotlin program looks like and how to set up a project in common IDEs.

What is Kotlin?

Kotlin is a multiplatform and multiparadigm programming language emphasizing safety, conciseness and interoperability. Conceived in late 2010, it was first released in February, 2016, and has since steadily become increasingly popular and promising a tool in many development areas, be it an Android development, desktop applications, or server-side solutions. The company which stands behind the language and has been investing into its development ever since is JetBrains, famous for its excellent software engineering tools, such as IntelliJ IDEA.

By November, 2019, Kotlin had reached version 1.3, acquiring a massive community, well-developed ecosystem and extensive tooling. Having overgrown an original intent of creating a better Java alternative, it now embraces multiple platforms, including Java Virtual Machine, Android, JavaScript and native applications. In 2017, Google had announced Kotlin as an officially-supported language of Android platform, which gave a tremendous boost to the language's popularity. Nowadays, a lot of companies – including Google, Amazon, Netflix, Pinterest, Uber and many others – are using Kotlin for production development, and the number of open positions for Kotlin developers is growing steadfastly.

It all became possible due to efforts devoted to careful language design and putting into action the primary traits, which make Kotlin such an excellent development tool. The language philosophy has mainly arisen based on the problems it was intended to solve back in 2010. By that moment, JetBrains had already accumulated an extensive Java code base for products centered around its IntelliJ platform, which apart of arguably the most known IntelliJ IDEA, had also included a set of minor IDEs dedicated to different technologies such as WebStorm, PhpStorm, RubyMine, etc.

The maintenance and growth of such codebase, however, was being hampered by Java itself due to its slow pace of evolution and lack of many useful features which at that moment had already been made available in such languages as Scala and C#. Having researched the JVM languages available at that moment, the company concluded that no existing language proved satisfiable for their needs and decided to invest resources into implementing their own language. The new language was eventually named Kotlin as a tribute to an island near Saint-Petersburg, Russia, where most of its development team is located.

So, what are those traits which have been shaping the language from the very beginning? In fact, we've already given the answer in its definition. The reason behind Kotlin is a need for multiparadigm language that emphasizes safety, conciseness and interoperability. Let's look at these traits in more detail.

Safe

For a programming language to be safe requires it to prevent programmer's errors. In practice, designing the language with respect to safety is a matter of tradeoff since error prevention typically comes at a cost: you give the compiler more detailed information about your program or let it spend more time reasoning about its correctness (probably both). One of the Kotlin design goals was to find a sort of a

golden mean: contriving a language with stronger safety guarantees than Java, but not so strong to frustrate a developer's productivity. And although Kotlin solution is by no means absolute, it has repeatedly proved to be an efficient choice in practice.

We'll discuss the various aspects of Kotlin safety as we go through the book. Here we'd like to point out some major features:

- Type inference which allows the developer to omit explicit declaration types in most cases (Java 10 introduced this for local variables);
- Nullable types regulate the usage of null and help to prevent infamous `NullPointerException`;
- Smart casts which simplify type casting, reducing the chance of casting errors at runtime

Multiparadigm

Initially, the meaning behind Kotlin multi paradigmality implied the support of functional programming in addition to conventional object-oriented paradigm that is typical of many mainstream programming languages such as Java. The functional programming is based around the idea of using functions as values: passing them as parameters or returning from other functions, declaring locally, storing in variables, etc. Another aspect of the functional paradigm is an idea of immutability, meaning that objects you manipulate can't change their state once created and functions can't produce side effects.

The major benefit of this approach is improved programming flexibility: being able to create a new kind of abstraction, you can write more expressive and concise code, thus increasing your productivity.

Note that although functional programming principles can be employed in many languages (Java's anonymous classes, for example, were obvious choice before introduction of lambdas), not every language has necessary syntactic facilities encouraging the writing of such code.

Kotlin, on the contrary, included necessary features right from the start. They include, in particular, functional types smoothly integrating functions into the language type systems and lambda expressions meant to create functionally-typed values from code blocks. The standard library as well as external frameworks provides extensive API, facilitating the functional style. Nowadays, many of that also apply to Java which had

introduced functional programming support starting with Java 8. It's expressiveness, though, still somewhat falls behind that of Kotlin's.

We'll cover the basics of functional programming in *Chapter 5*, but its applications and examples will accompany us throughout all the remaining of the book.

Over its growth, the language also began to exhibit two more programming paradigms. Thanks to the ability to design APIs in a form of **domain-specific languages (DSLs)**, Kotlin can be used in a declarative style. In fact, many Kotlin frameworks provide their own DSLs for specific tasks with no need to sacrifice type-safety or expressive power of the general-purpose programming language. For example, exposed framework includes a DSL for defining database schema and manipulating its data, whereas kotlinc.html gives a concise and type-safe alternative to HTML template languages. In *Chapter 11, Domain-Specific Languages*, we'll discuss these examples in more detail as well as learn how to create our own DSLs.

One more paradigm, namely, concurrent programming, entered the language with the introduction of coroutines. Although concurrency support by itself is present in many languages, including Java, the Kotlin features a rich set of programming patterns which enable a new programming approach. We'll cover the basics of concurrency in *Chapter 13, Concurrency*.

All in all, the presence of multiple paradigms greatly increase a language expressive power, making it a more flexible and multi-purposed tool.

Concise and expressive

Developer productivity is largely tied with the ability to quickly read and understand the code, be it some other developer's work or maybe your own after a significant time has passed. In order to understand what a specific piece of code does, you need to also understand how it's related to other parts of your program. That's why reading existing code generally takes more time than writing a new one and that's why language conciseness, an ability to clearly express programmer's intents without much information noise is a crucial aspect of language efficiency as a development tool.

The designers of Kotlin did their best to make language as concise as possible, eliminating a lot of notorious Java boilerplate, such as field getters and setters, anonymous classes, explicit delegation and so on. On the other hand, they made sure the conciseness is not overtly abused – unlike Scala, for example, Kotlin doesn't allow the programmer to define custom operators, only redefine existing ones since the former tends to obfuscate the operation meaning. In the course of the book we'll see

numerous implications of this decision and how useful it turned to be.

Another aspect of Kotlin's conciseness is tightly related to the DSLs (see *Chapter 11, Domain-Specific Languages*), which greatly simplifies the description of specific programming domains with a minimum of syntactic noise.

Interoperable

Java interoperability was a major point in Kotlin design, since Kotlin code wasn't meant to exist in isolation, but to cooperate as smoothly as possible with existing codebase. That's why Kotlin designers made sure that not only existing Java code could be used from Kotlin, but that Kotlin code could also be used from Java with little to no effort. The language also includes a set of features specifically meant to tune interoperability between Java and Kotlin.

As the language outgrew the JVM and spread to other platforms, interoperability guarantees were extended to cover interaction with JavaScript code for JS platform and C/C++/Objective C/Swift code for native applications.

We'll devote *Chapter 12, Java Interoperability*, to discuss Java interoperability issues and how Kotlin and Java can be mixed together in the same project.

Multiplatform

Multi platformity wasn't an original intent of Kotlin designers, but rather manifested itself as a result of language evolution and adaptation to the needs of the development community. While JVM and Android remain primary targets of Kotlin development, nowadays the supported platforms also cover:

- JavaScript, including browser and Node.js applications as well as JavaScript libraries;
- Native applications and libraries for macOS, Linux and Windows

Since 1.3, Kotlin supports multiplatform development with major use cases being sharing the code between Android and iOS applications and creating multiplatform libraries for JVM/JS/Native world.

Kotlin Ecosystem

Throughout its evolution, Kotlin has given rise to a rich set of libraries and frameworks covering most of the aspects of software development. Here we try to give you an

overview of the available tools which hopefully, will serve as a guide in this ocean of possibilities. Note, however, that as ecosystem is continuously growing, the state-of-the-art presented in this book at the moment it's being written will inevitably fall out of date, so don't hesitate looking for it by yourself. A good starting point is a community-updated list of libraries and frameworks available on the Awesome Kotlin website: <https://kotlin.link>.

It's also worth noting that thanks to well-conceived Java interoperability, Kotlin applications may benefit from a whole lot of existing Java libraries. In some cases, they have specific Kotlin extensions allowing one to write more idiomatic code.

Coroutines

Thanks to the concept of suspendable computations, Kotlin is able to support concurrency-related programming patterns such as `async/await`, `futures`, `promises` and `actors`. The coroutines framework provide a powerful, elegant and easily scalable solution to concurrency problems in Kotlin application, whether it be a server-side, mobile or a desktop one. The major features of the coroutines include, among others:

- A lightweight alternative to threads;
- Flexible thread dispatching mechanism;
- Suspendable sequences and iterators;
- Sharing memory via channels;
- Using actors to share mutable state via message sending.

We'll cover the basics of coroutine API in the *Chapter 13, Concurrency, dealing with concurrency issues in Kotlin*.

Testing

A part of the familiar Java testing frameworks such as JUnit, TestNG and Mockito which can be employed in a Kotlin application with little effort, the developers can enjoy the power of Kotlin-tailored frameworks providing useful DSLs for testing purposes, be it test definitions or mocking your objects. In particular, we'd like to point out:

- Mockito-Kotlin, an extension for popular Mockito framework simplifying object mocking in Kotlin;

- Spek, a behavior-driven testing framework supporting Jasmine-and Gherkin-styled definition of test cases;
- KotlinTest, a ScalaTest-inspired framework which supports flexible test definitions and assertions.

In *Chapter 14, Testing with Kotlin*, we'll pay more attention to the features provided by Spek and Mockito and consider how to use them in your projects.

Android Development

Android is one of the major and most-actively growing application areas of Kotlin. This has become especially relevant after Google announced Kotlin as a first-class Android language implying, in particular, that Android tooling is now being designed and developed with due regard to the Kotlin features as well. Besides the excellent programming experience brought by the Android Studio plugin, Android developers can benefit from the smooth interoperability with many popular frameworks such as Dagger, ButterKnife and DBFlow. Among Kotlin-specific Android tools, we'd like to pay attention to Anko and Kotlin Android Extensions.

Kotlin Android Extensions is a compiler plugin whose main feature is data-binding, which allows you to use XML-defined views as if they're implicitly defined in your code, thus avoiding the infamous `findViewById()` calls. Besides that, it supports view caching and an ability to automatically generate `Parcelable` implementations for user-defined classes. Thanks to that there is no need to employ external frameworks like ButterKnife in pure Kotlin projects.

Anko is a Kotlin library, simplifying the development of Android applications. In addition to numerous helpers, it includes a domain-specific language (Anko Layouts) for composing dynamical layouts accompanied by the UI preview plugin for Android Studio as well as database query DSL based around Android SQLite.

We'll cover some of these features in *Chapter 15, Android Applications*, which introduces the reader to Kotlin-powered Android development.

Web Development

Web/Enterprise application developers can also benefit from Kotlin. Popular frameworks such as Spring 5.0 and Vert.x 3.0 include Kotlin-specific extensions that allow using their functionality in more Kotlin-idiomatic way. Besides that you can employ pure Kotlin solutions using a variety of frameworks:

- Ktor, a JetBrains framework for creating asynchronous server and client applications;
- `kotlinx.html`, a domain-specific language for building HTML documents
- Kodein, a dependency injection framework

We'll discuss specifics of building web applications and microservices using Ktor and Spring in the *Chapters 16 and 17*.

Desktop applications

Developers of desktop applications for JVM platform can employ TornadoFX, a JavaFX-based framework. It provides helpful domain-specific languages for building GUI and style description via CSS, supports FXML markup and MVC/MVP architecture. It also comes with an IntelliJ plugin, simplifying generation of TornadoFX projects, views and other components.

Getting started with Kotlin

Now you should have an idea of the Kotlin ecosystem, and the only thing we need to discuss before we can start exploring the language is how to set up a working environment.

Setting up an IntelliJ project

Although, Kotlin by itself, like most programming languages, is not tied to a particular IDE or text editor, the choice of development tools has a great impact on developer's productivity. As of now, JetBrains IntelliJ platform provides the most powerful and comprehensive support of Kotlin development lifecycle. Right from the start, Kotlin IDE has been developed in tight integration with the language itself, which helps it to stay up-to-date with Kotlin changes. For that reason, we recommend using it for your own projects and will employ it for the examples in the book.

Since IntelliJ IDEA 15 Kotlin support is bundled into the IDE distribution, so you won't need to install any external plugins to facilitate Kotlin development. For this book we've been using IntelliJ IDEA 2019.1, released in March, 2019.

If you don't have an IDEA installed, you can download the latest version from www.jetbrains.com/idea/download, and follow installation instructions at <https://www.jetbrains.com/help/idea/install-and-set-up-product.html>. IDEA

comes in two editions: Community, which is free and open-source, and Ultimate, which is a commercial product. The major difference is that Ultimate edition includes a set of features related to the development of web and enterprise applications as well as database tools. You can find more detailed list of changes at the download page. For this book we won't need the Ultimate feature, so IDEA Community is more than enough.

If you haven't opened a project in the IntelliJ before, you'll see a welcome screen on startup where you can click **Create New Project** option to go directly to the project wizard dialog. Otherwise IntelliJ opens the recently edited project(s); in this case, choose **File | New | Project...** in the application menu.

Project types are grouped into categories you can see in the left pane. The exact set of categories and projects depends on the installed plugin, but for now we're interested in Kotlin category which is available out-of-the box, thanks to the bundled Kotlin plugin. Clicking on it you'll see the list of available project templates (*Figure 1.1*).

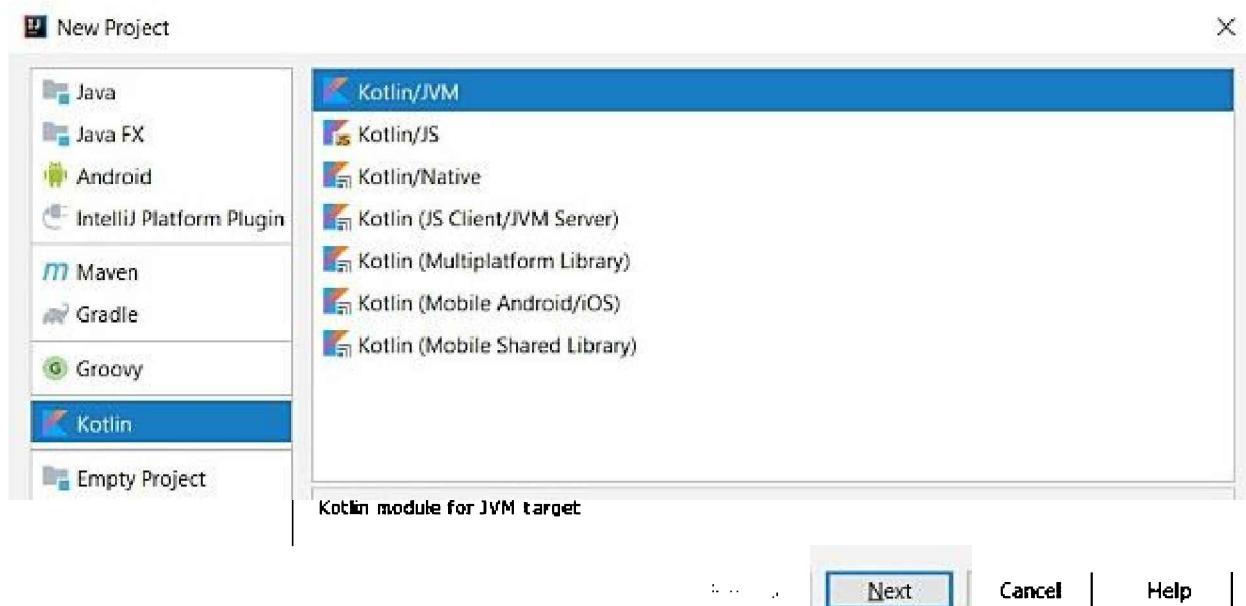


Figure 1.1: New Project wizard

As of version 1.3.21 (which we've used when writing this book), the Kotlin plugin supports creation of projects targeting JVM, JavaScript, native applications as well as several cases of multiplatform projects such as mobile applications, targeting both Android and iOS. The platform determines both the type of compiler artifacts (bytecode for the JVM, .js files for JavaScript and platform-specific executables for

Kotlin/Native) and a set of available dependencies your project can use; a project targeting JavaScript, for example, can't access classes from Java class library. As the book continues, our primary interest will be the JVM applications. So, for this example we'll choose Kotlin/JVM option and click **Next**.

On the next step (*Figure 1.2*) you need to provide a project name and location, which is a root directory for project-related content, including its source files. Note that IntelliJ suggests location automatically, based on the project name you've typed, but you can change it if necessary.

Since our project targets the JVM platform, we also have to specify a default JDK to use for project compilation. That would allow our project to use classes from Java standard library as well as compile Java sources in mixed-language projects. In the *Chapter 12, Java Interoperability*, we'll cover such projects in more detail and also discuss how to introduce Kotlin support to existing Java projects.

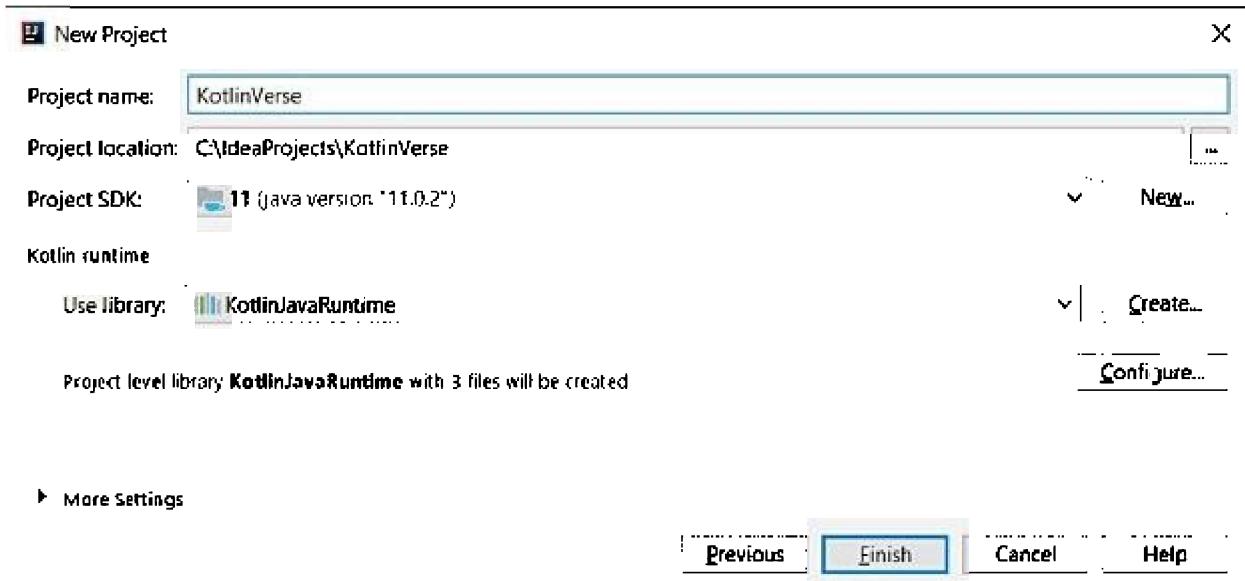


Figure 1.2: Project name, location and dependencies

We recommend using JDK 8 or higher. For this example, we've chosen the latest (as of the book's writing) release version of JDK 11 available on www.oracle.com. Usually, IntelliJ auto-detects the JDK installed on your machine, but if that doesn't happen or none of preconfigured JDKs in the Project SDK list suits your purposes, you can add a new one by clicking the New button and specifying a path to JDK root directory. You may need to install JDK beforehand; to do that download it from <https://www.oracle.com/technetwork/java/javase/downloads/index.html>, and

follow the installer's instructions.

One more thing worth mentioning is a Kotlin runtime library that IntelliJ has preconfigured for our project. By default, the project will reference a library in the IDE plugin directory which means it's upgraded automatically whenever you update the plugin itself. If you, however, want your project to depend on a particular version of Kotlin runtime, only updating it manually out of necessity, you may change that behavior: to do that click on **Create** button and choose **Copy** to have the option of specifying a directory name where the library is to be kept.

Now to the final step: clicking the **Finish** button will get IntelliJ to generate and open an empty project. By default, IntelliJ presents it in a two-panel view: Project tool window on the left and editor area occupying most of the remaining area. The editor is initially empty since we haven't opened a single file yet, so we'll first focus on the Project window and use it to create a new Kotlin file.

If Project tool window is absent, you can bring it by clicking on the **Project** button (usually it's on the left side of window border) or by using the shortcut Alt+1 (Meta+1).

The Project window shows hierarchical structure of your project. Let's expand the root nodes and see what it contains (*Figure 1.3*). Currently we're mostly interested in three items:

- `src` directory, which serves as a content root containing project source files;
- `out` directory, where the compiler puts generated bytecode;
- External libraries, which lists all libraries the project depends on.

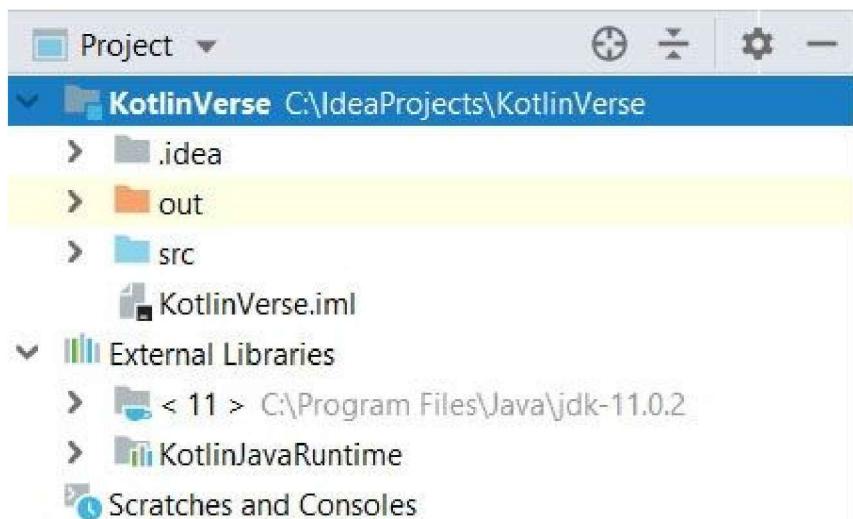


Figure 1.3: Project structure tool window

Now, right-click the `src` directory and select **New | Kotlin File/Class** command. In the dialog that follows, type a file name `main.kt`, make sure that the **Kind** field is set to **File**, and click **OK**. You'll see an updated Project window that shows a new file which is at the same time opened in the editor. Note that Kotlin source files must have `.kt` extension.

At last we're ready to write an actual code. Let's type the following in the editor window (*Figure 1.4*):

```
fun main() {  
    println("Hello, KotlinVerse!")  
}
```

The code above defines the `main` function which serves as an entry-point for Kotlin application. The function body consists of a single statement, a call to the standard library function `println()` writing its argument to the program's standard output with new line added at the end.

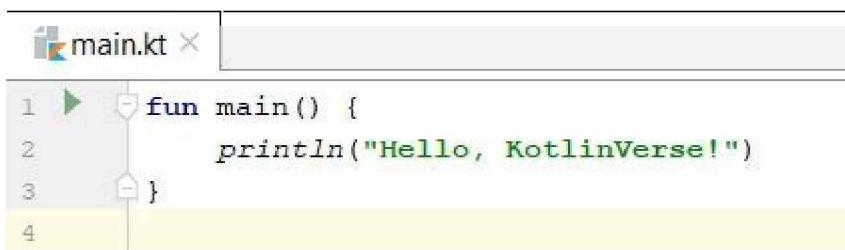


Figure 1.4: Hello, World program

Java developers will surely recognize a similarity between this code and the following Java program:

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("Hello, World!")  
    }  
}
```

In fact, JVM's version of Kotlin `println()` function is just a call to `System.out.println()`. Since JVM entry-point must be a static class method, you might be wondering how Kotlin application starts up without defining a single class.

The answer is that although we haven't defined a class explicitly, the Kotlin compiler will create one behind the scenes, putting there the JVM's entry-point which in turn would call our `main()` function. We'll get back to these so called facade classes in *Chapter 12, Java Interoperability*, because they constitute a major aspect of Kotlin/Java interoperability.

Also note that unlike JVM entry-point, which is supposed to take an array of command-line arguments as its parameter, our `main()` function has no parameters at all. This comes in handy for the cases when command-line arguments are not used. If necessary, however, you can still define entry-point using these arguments:

```
fun main() {  
    println("Hello, KotlinVerse!")  
}
```

Parameterless `main()` is in fact a recent feature introduced in Kotlin 1.3. In earlier language versions, the only acceptable entry point was the one taking `String<Array>` argument.

You might've noticed a small green triangle at the left of `main()` definition. This line marker indicates that function `main()`, being an entry-point, is executable. Clicking on that marker brings up a menu which allows you to run or debug its code. Let's choose **Run MainKt** option and see what happens (*Figure 1.5*).

MainKt, by the way, is a name of the compiler-generated facade class we've mentioned a few paragraphs ago. Upon choosing the **Run** command, IntelliJ compiles our code and executes the program. Run tool window, which is opened upon program startup, gets automatically linked to its standard I/O streams serving as a built-in console. If you've done everything correctly, the program will print **Hello, Kotlin Verse** to the console and terminate successfully.

Figure 1.5: Running a program

If you look inside the `out` directory, you should see `.class` files generated by the Kotlin compiler from our source program.

Congratulations! You now have an understanding of how to set up and run a Kotlin project in the IntelliJ IDEA environment and are ready to delve into the language fundamentals. `KotlinVerse`, here we go!

Using REPL

Kotlin plugin for IntelliJ provides an interactive shell which allows you to evaluate program instructions on-the-fly: this can be used for quick testing of your code or experimenting with library functions. It also gets quite handy for those who are just learning the Kotlin language. This feature is called REPL. The meaning behind this name is Read/Evaluate/Print Loop because that's what the shell does: reading code the user has typed, evaluating it, printing the result (if any) and looping the whole thing over. In order to access the REPL, select **Tools | Kotlin | Kotlin REPL**.

You can type Kotlin code in the REPL window just like you do it in the editor. The major difference is that each piece of code is compiled and executed right after you enter it. Once the code is typed, you press **Ctrl+Enter** (**Command+Return**) telling the IDE to process your input. Let's try it out with:

```
println("Hello from REPL")
```

As far as you can see, IntelliJ responds with printing `Hello from REPL` to the

console which, in this case is shared with REPL window.

The printing of the string above is in fact a side effect of the `println()` function which by itself doesn't return any result to the calling program. If we, however, attempt to evaluate some expression, which does have some meaningful result, the output is slightly different. Let's try entering `1+2*3` (*Figure 1.6*):

```
Type :help for help, :quit for quit
println("Hello from REPL")
Hello from REPL

1+2*3
=
7
```

Figure 1.6: Using REPL

The REPL gives us the expression result which is 7. Note the difference in font and = icon as opposed to the `println()` example. This is to signify that 7 is an actual result of the code you've typed. To sum it up, we advise you to get acquainted with this tool and use it throughout the book (and beyond) to experiment with any features you feel necessary.

Kotlin Playground

Besides REPL shell available in IntelliJ, it's worth mentioning a similar but more powerful online tool which lies somewhere in between a REPL and a full-fledged IDE. The tool in question is the Kotlin Playground. To give it a try, open <https://play.kotlinlang.org> in your browser (*Figure 1.7*).

The Kotlin Playground is basically an online environment which allows you to explore the language with no need for an actual IDE, which still has some of its intelligent features at your disposal, including code editor, syntax and error highlighting, code completion and console program runner.

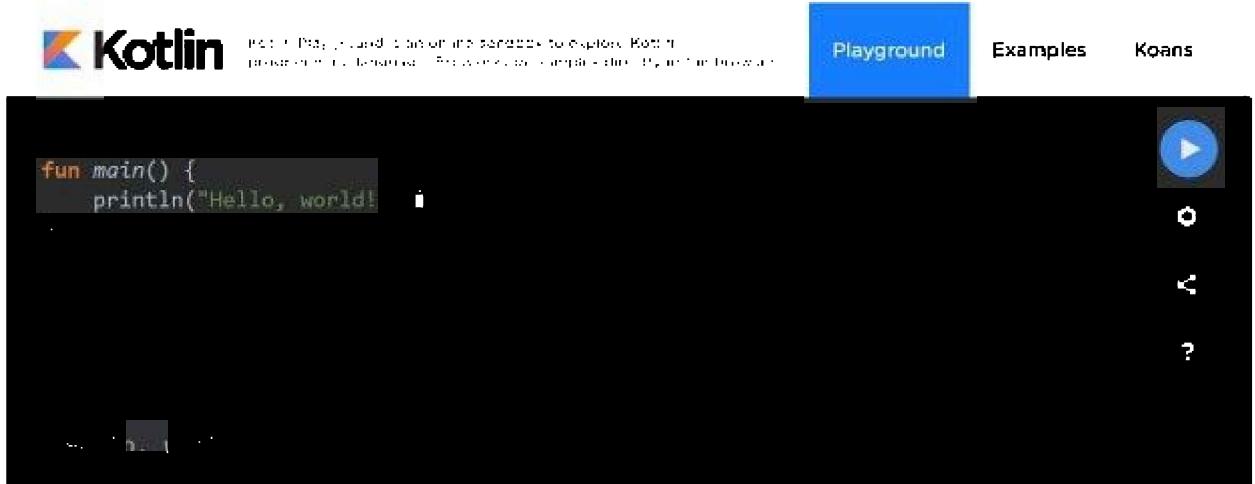


Figure 1.7: The Kotlin Playground

The Playground site also includes a bunch of examples and exercises to familiarize the developer with major Kotlin features. The exercises, also known as Kotlin Koans, take a form of failing test cases which must be fixed in order to pass (Figure 1.8):

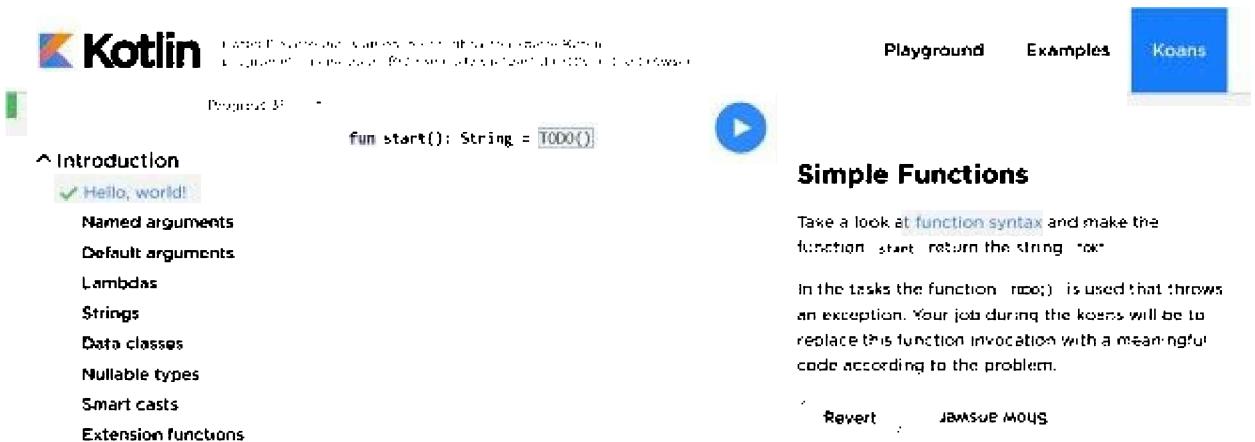


Figure 1.8: Kotlin Koans

We strongly recommend going through these examples as a valuable complement to the book itself.

Setting up an Eclipse project

Kotlin tooling is not limited to IntelliJ: thanks to the Eclipse plugin, the developers preferring that IDE can use Kotlin as well. Although language support in Eclipse is not as extensive as in IntelliJ, it still provides a lot of code assistance features for the

developer's benefit, such as code highlighting, completion, program execution and debugging, basic refactorings and more.

If you don't have Eclipse, it can be freely downloaded from www.eclipse.org/downloads. After running the installer, choose **Eclipse IDE for Java Developers** (or **Enterprise Java Developers**) and follow the instructions. For that tutorial we've used Eclipse 4.10 released in December, 2018.

Unlike IntelliJ IDEA, Eclipse doesn't come with bundled Kotlin support, which means that the plugin must be installed from the Eclipse Marketplace before we can get to writing the code. To do that select **Help | Eclipse Marketplace**...and search for Kotlin plugin (*Figure 1.9*).

After you click **Install** button, the IDE will download and install the plugin. Make sure to accept license agreements and restart Eclipse in order to complete the installation.

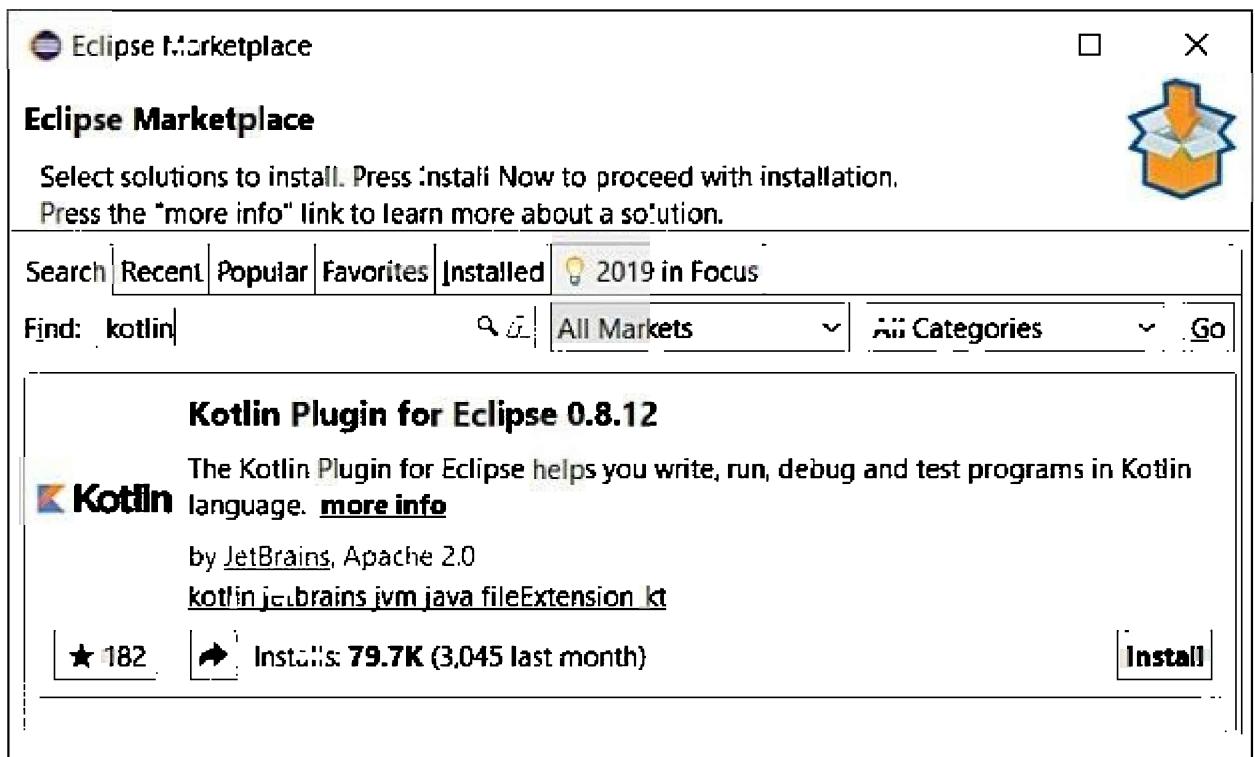


Figure 1.9: Installing Kotlin plugin from Eclipse Marketplace

Now we can get to setting up a project. First, we switch IDE to Kotlin perspective using **Window | Perspective | Open Perspective | Other...** command and choosing Kotlin in the dialog that follows. Besides the layout change, this perspective

makes some Kotlin actions accessible directly from the application menu. So, in order to create a project we need to choose **File | New | Kotlin Project...**, specify a new project name, and click **Finish** (*Figure 1.10*):

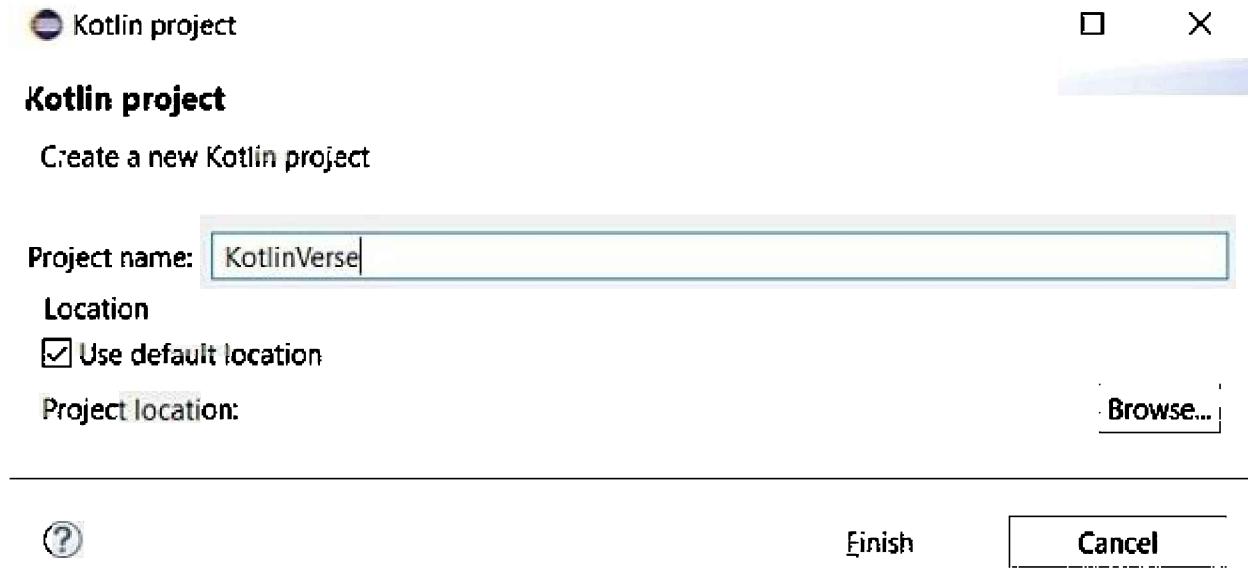


Figure 1.10: Creating Kotlin project

We're almost there! Expanding the Kotlin Verse node in the Package Explorer view, you can see the components of our newly created project: **Java Runtime Environment (JRE)** library, the Kotlin Standard Library and, as of yet empty, `src` directory where the source files are kept. Now let's create our first Kotlin file: right-click on the `src` directory and choose **New | Kotlin File**. Type file name and click **Finish** (*Figure 1.11*).

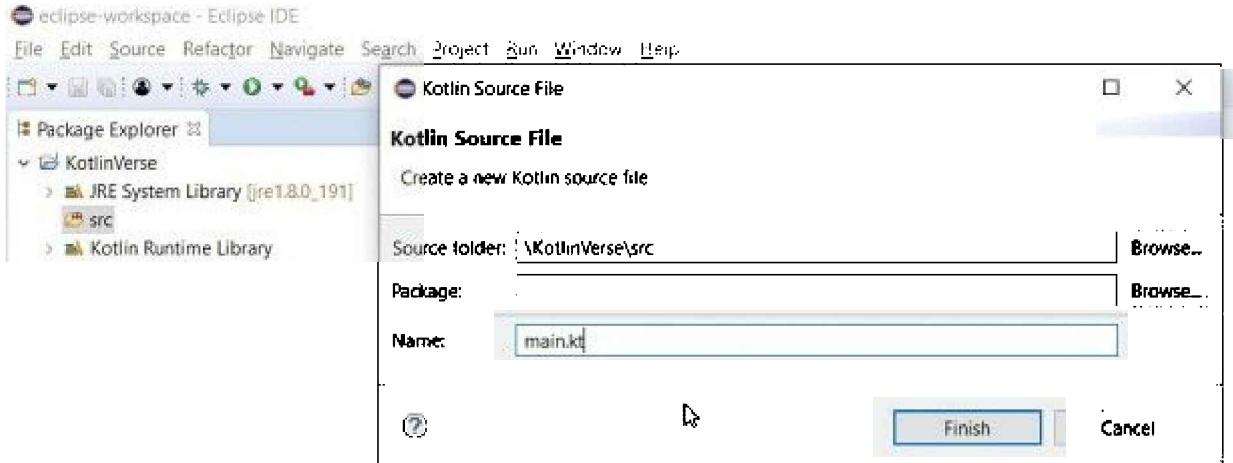


Figure 1.11: Creating Kotlin file

Eclipse automatically opens a new file in the editor window. Let's type Hello, World program, you can recognize from our earlier example (*Figure 1.12*).

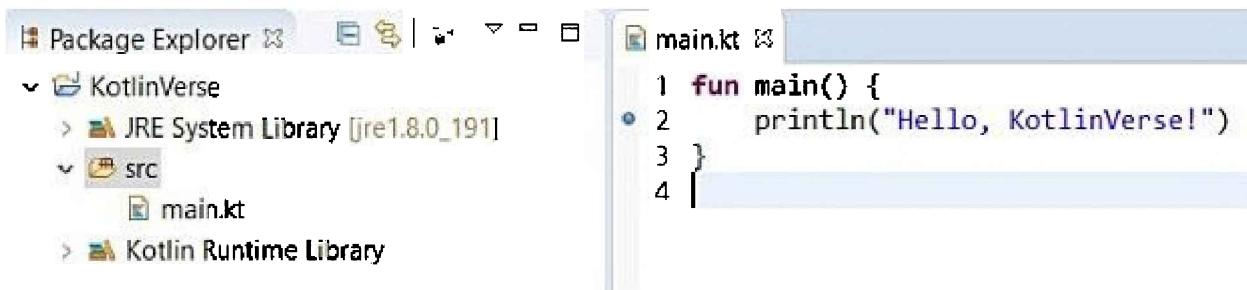


Figure 1.12: Hello, World in Eclipse

That's it! To run the program, you may use **Run | Run** command: Eclipse will compile your file to JVM bytecode and start resulting program, redirecting its output to the console view.

Conclusion

In this chapter we have learned the major aspects of Kotlin language, such as safety, conciseness and support for functional and object-oriented programming paradigms. Together with the support of multiple development platforms, such as JVM, Android, JavaScript and native applications, well-designed interoperability with Java or other platform-specific code, extensive ecosystem of tools, libraries and frameworks and fast-growing community makes Kotlin an excellent language that is definitely worth

learning.

We've also looked at the common tools you can use for getting started with Kotlin programming, including IntelliJ IDEA, Eclipse IDE and Kotlin Playground. Now we are ready to move ahead. In the next chapter, we'll focus on the anatomy of some basic syntactic structures like variables and expressions, as well as get acquainted with the basic Kotlin types.

CHAPTER 2

Language Fundamentals

In this chapter you'll learn the basic syntactic elements of Kotlin program and how to define and use variables. You'll get an understanding of Kotlin types which are used to represent numeric, character and Boolean values as well as their built-in operations, and get acquainted with more complex structures such as strings and arrays. Along the way, we'll also point out the major differences from Java syntax and type system which should ease the migration to Kotlin.

Structure

- Variable definitions;
- Mutable and immutable variables;
- Basic expressions: references, calls, unary/binary operators;
- Basic types and their operations: numbers, characters, Boolean values;
- String type: string literals and templates, basic String operations;
- Array types: array construction and basic operations.

Objective

Our objective will be to understand the basic Kotlin types and learn to program simple computations with numbers, Booleans, characters/strings and arrays.

Basic syntax

In this chapter we'll learn how to define and use local variables and get to know basic Kotlin expressions, such as references, calls and unary/binary operations.

Comments

Like Java, Kotlin supports three varieties of comments you can use to document your code:

- Single-line comments which start with // and continue till the end of line;
- Multi-line comments delimited by /* and */;
- KDoc multiline comments delimited by /** and */.

KDoc comments are used to generate rich text documentation similar to Javadoc.

```
/*
    multi-line comment
    /* nested comment */
*/
println("Hello") // single-line comment
```

Java vs. Kotlin: Unlike in Java, multiline comments in Kotlin can be nested.

Defining a variable

The simplest form of a variable definition in Kotlin takes the following form:

```
val timeInSeconds = 15
```

Let's consider the elements which make it up:

- The val keyword (from value);
- Variable identifier, which is a name you give to a new variable and use to refer to it later in the code;
- An expression which defines variable's initial value and follows after = sign.

Java vs. Kotlin: You might've noticed that we didn't put a semicolon (;) at the end of the variable definition. This is not a mistake: In Kotlin you can omit semicolon at the end of the line. In fact, this is a recommended code style: putting one statement per line you'll virtually never need to use semicolons in your code.

IDE Tips: IntelliJ enforces this code style by showing a warning for each unnecessary semicolon.

Suppose we want to write a program which asks user for two integer numbers and outputs their sum. Here is how it might look like in a Kotlin:

```
fun main() {
    val a = readLine()!!.toInt()
```

```
val b = readLine() !!.toInt()
println(a + b)
}
```

Let's look more closely at what it does:

- `readLine()` is a call expression which tells the program to execute the `readLine`, a standard Kotlin function which reads a single line from the standard input and returns it as a character string.
- `!!` is a not-null assertion operator which throws an exception if the `readLine()` result is null. Unlike Java, Kotlin tracks if a type can contain nulls and wouldn't allow us to call the `toInt()` function unless we make sure that nulls are ruled out. For now, we can simply ignore it since `readLine()` never returns null when reading from console. In the *Chapter 4, Working with Classes and Objects*, we'll discuss the issue of type nullability in more detail.
- We then call `toInt()` function on the result of `readLine()` call. `toInt()` is a method defined in the Kotlin's `String` class which converts the character string upon which it's called into the integer value. If the string in question does not correspond to a valid integer, `toInt()` fails with a runtime error which in this case just terminates our program. For now, we won't worry about that assuming that all user input is valid and postpone the issue of error handling till the next chapter.
- The result of `toInt()` call is assigned to the variable '`a`' we define in the same line.
- Similarly, we define the second variable '`b`' which is assigned an integer entered on the second line.
- Finally, we compute the sum of two integers, '`a + b`', and pass the result to `println()` function call which prints it to the standard output.

The variables we've introduced above are called local since they're defined in the body of a function (in our case, it's `main()`). Apart from that, Kotlin allows definition of properties which are similar to variables, but in general, can perform some computations on reading or writing. For example, as we'll see further, all strings in Kotlin have property `length` which contains the number of characters.

If you're familiar with Java, you've probably noticed that we didn't specify the type of our variables, and yet the program successfully compiles and runs (*Figure 2.1*). The

reason is a so called type inference, a language feature which in most cases allows the compiler to deduce type information from context. In this case, the compiler already knows that `toInt()` function returns a value of type `Int`, and since we assign `toInt()` result to our variable, it assumes that the variable must also be of type `Int`. Thanks to type inference, Kotlin remains a strongly-typed language, yet saving the developer from cluttering the code with unnecessary type annotations. Throughout the book we'll see various examples of how type inference can simplify programming in Kotlin.

Java vs. Kotlin: Java 10 has introduced a similar feature for local variables. Now you can write, say:

```
var text = "Hello"; // String is inferred automatically
```

In Kotlin, however, type inference is not limited to local variables and has a much wider application; we'll see in the upcoming chapters.

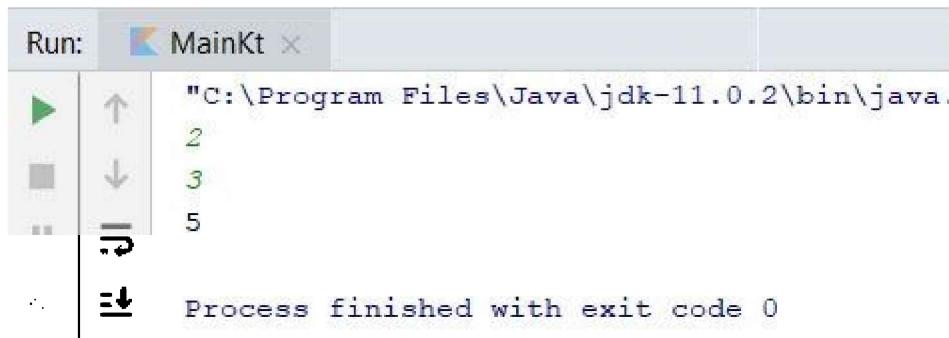


Figure 2.1: Running summation program in IntelliJ

You can also specify the type explicitly when necessary. To do that you put the type specification after the variable identifier separating them with a colon (:):

```
val n: Int = 100
val text: String = "Hello!"
```

Note that in this case, the initial value must belong to the variable type. The following code will produce compilation error:

```
val n: Int = "Hello!" // Error: assigning String value to Int
variable
```

IDE Tips: IntelliJ allows you to see the type which the compiler has inferred for any expression or variable. To do it select expression of interest in editor

or simply put the caret at variable identifier and press **Ctrl + Shift + P** (**Command + Shift + P**):

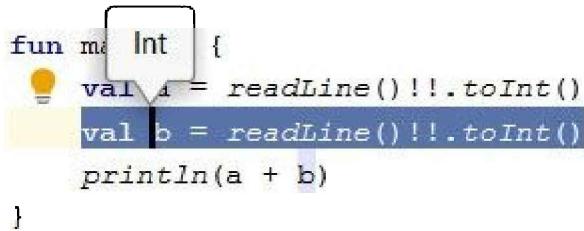


Figure 2.2: Show Expression Type action

On top of this, you can add or remove explicit types using simple actions. Just place editor caret at the variable identifier, press **Alt + Enter** and choose **Specify type explicitly** or **Remove explicit type specification** respectively (the latter also works on the type specification itself).

It's possible to omit an initial value and initialize a variable later, in a separate statement. This could be helpful if computing an initial value can't be put into a single expression. In this case you'll have to specify variable type explicitly:

```
val text: String
text = "Hello!"
```

Note that a variable must be initialized before you can read its value. If the compiler can't guarantee that a variable is definitely initialized before use, it will report an error:

```
val n: Int
println(n + 1) // Error: variable n is not initialized
```

Identifiers

Identifiers are names you give to the entities defined in the program, like variables or functions. Kotlin identifiers come in two flavors. This first one is quite similar to Java identifiers and can be an arbitrary string of characters conforming to the following rules:

- It may only contain letters, digits and underscore characters (_) and may not start with a digit;

- It may consist entirely of underscores: names like `_`, `__`, `___` and so on are reserved and can't be used as identifiers;
- It may not coincide with a hard keyword.

Hard keywords (like `val` or `fun`) are considered keywords regardless of where they are put in code. Soft keywords (like `import`), on the other hand, are parsed as keywords only in the specific context, outside of which they can be used as a normal identifiers. You can find the full list of hard and soft keywords at kotlinlang.org/docs/reference/keyword-reference.html.

Letters and digits, like in Java, are not limited to ASCII, but include national alphanumeric characters as well. It is, however, considered a good practice to use names based on English words.

Java vs. Kotlin: Note that unlike Java, dollar signs (\$) are not allowed in Kotlin identifiers.

Second form, a quoted identifier, is an arbitrary non-empty character string enclosed inside backquotes (`):

```
val `fun` = 1
val `name with spaces` = 2
```

Quoted identifiers may not contain new lines and backquotes themselves. On top of that they must satisfy platform-specific requirements. In the Kotlin/JVM code, for example, such identifiers may not contain any of the following characters since they are reserved by the JVM itself: `.`, `;`, `[`, `]`, `/<>`, `:`, `\`.

For a better readability, this feature shouldn't be abused, though. It primarily exists for Java interoperability because Java declaration names might coincide with Kotlin keywords (`fun`, for example, is a keyword in Kotlin, but not in Java) and Kotlin code should be able to use them if necessary. One more use case is a naming of test case methods which we'll see in the Chapter 14, *Testing with Kotlin*.

Mutable variables

The variable we've considered so far are in fact immutable: in other words, you can't reassign their value once they're initialized. In this regard they resemble final variables in Java. You should aim at declaring all variables immutable as much as practical since using such immutable variables and avoiding functions with side effects facilitates functional style and simplifies reasoning about your code.

If necessary, however, you can still define a mutable variable by using `var` keyword (from `variable`) instead of `val`. The basic syntax remains the same, but now we can change variable's value as many time as we like. The operation (`=`) we use to change variable's value is called the assignment. We've already seen its use for initialization of immutable variables.

```
var sum = 1  
sum = sum + 2  
sum = sum + 3
```

Note that variable type specified or inferred at its declaration stays the same whether it's mutable or not. Assigning the value of wrong type is a compile-time error:

```
var sum = 1  
sum = "Hello" // Error: assigning String value to Int  
variable
```

Additionally, Kotlin supports so called augmented assignments which combine changing the variable's value with one of binary operators: `+`, `-`, `*`, `/`, `%`. For example:

```
val result = 3  
result *= 10 // result = result * 10  
result += 6 // result = result + 6
```

Such assignments are available whenever corresponding binary operators make sense for a given variable.

Java vs. Kotlin: As opposed to Java, Kotlin assignments are statements, rather than expressions and do not return any value. This, in turn, means that in Kotlin you can't form assignment chain similar to Java's `a = b = c`. Such assignments are forbidden in Kotlin because they are considered error-prone and rarely useful. This includes augmented assignments as well.

There are two more operations that are concerned with changing a variable's value: increment (`++`) and decrement (`--`). Their most obvious usage is increasing or decreasing a numeric variable by 1. Like in Java, these operations come in prefix and postfix form:

```
var a = 1  
println(a++) // a is 2, 1 is printed
```

```

println(++a) // a is 3, 3 is printed
println(--a) // a is 2, 2 is printed
println(a--) // a is 1, 2 is printed

```

These examples demonstrate that while both the prefix and the postfix operations modify the variable, the former's result is a new value, while the latter return variable's value before change.

Expressions and operators

Kotlin expression we've used in the examples above can be divided into the following categories:

- Literals representing specific values of particular type (like 12 or 3.56);
- Variable/property references and function calls (a, readLine(), "abc".length, "12".toInt());
- Prefix and postfix unary operations (-a, ++b, c--);
- Binary operations (a + b, 2 * 3, x < 1).

Every expression has a definite type which describes a possible range of values and allowed operations. For example, literal 1 has a type Int, while readLine() !! call is of type String.

Also note that variable references and function calls may have a dot-separated receiver expression, like in readLine() !!.toInt(): this means that we are using toInt() function defined for the type String (which is the type of readLine() !!) in the context of readLine() !!.

Unary and binary operations have different precedence which determine the order of evaluation: for example, in 2 + 3*4 expression, we first evaluate 3*4 and then add the result to 2, getting 14. The order can be changed with parentheses, so (2 + 3)*4 will be equal to 5*4, or 20 instead. The precedence of operators that we're going to consider in this chapter can be summarized in the following table:

Category	Operations	Examples
Postfix	++ --	a*b++ // a*(b++) ++b-- // ++(b--) a*b.foo() // a*(b.foo())
Prefix	+ - ++ -- !	+a*b// (+a)*b

		$\text{++}a * b // (\text{++}a) * b$ $!a \mid\mid b // (!a) \mid\mid b$
Multiplicative	$*$ / $\%$	$a * b + c // (a * b) + c$ $a - b \% c // a - (b \% c)$
Additive	$+$ -	$a + b$ and $c // (a + b)$ and c
Infix	Named operators	$a < b$ or $b < c // (a < (b \text{ or } b)) < c$ $a == b$ and $b == c // (a == b)$ and $(b == c)$
Comparison	$<>$ $<=$ $>=$	$a < b == b < c // (a < b) == (b < c)$ $a < b \&\& b < c // (a < b) \&\& (b < c)$
Equality	$==$ $!=$	$a == b \mid\mid b != c // (a == b) \mid\mid (b != c)$
Conjunction	$\&\&$	$a \mid\mid b \&\& c // a \mid\mid (b \&\& c)$ $a \&\& b \mid\mid c // (a \&\& b) \mid\mid c$
Disjunction	$\ $	
Assignment	$=$ $+=$ $-=$ $*=$ $/=$ $\%=$	$a = b \% c // a = (b \% c)$ $a *= a + b // a *= (a + b)$

Table 2.1: Operation precedence

Binary operators with the same priority are evaluated from left to right. For example:

```
a.foo().bar()                                // (a.foo()).bar()
a*b%c                                         // (a*b)%c
(a == 1) or (b < 1) and (c > 1) // ((a == 1) or (b < 1)) and
(c > 1)
```

Over the course of the book, we'll introduce additional operations refining this table.

Basic types

In this section we'll consider Kotlin types which describe simple values like numbers, characters and Booleans. If you're familiar with Java, you can think of them as counterparts of Java primitive types, but the analogy is not perfect. In Java, you have clear distinction between primitive types like `int`, whose values are stored directly in the pre-allocated memory of the method and class-based reference types like `String`, whose values are just references to dynamically-allocated data of corresponding class. In Kotlin, the distinction is somewhat blurred since the same type – say, `Int` – can be represented as either primitive or reference one depending on the context. Java

includes special boxing classes which can be used to wrap primitive value, but Kotlin performs boxing implicitly when necessary.

Java vs. Kotlin: As opposed to Java, all Kotlin types are ultimately based on some class definition. This, in particular, means that even primitive-like types, such as Int, have some member functions and/or properties. You can, for example, write `1.5.toInt()` to call `toInt()` operation upon 1.5, a value of type Double, which converts it to integer number.

Types can form hierarchies based on a notion of subtyping: in essence when we say that type A is a subtype of B, we mean that any value of A can be used in any context which requires a value of B. For example, all Kotlin types disallowing null values are direct or indirect subtypes of a built-in type Any, so the following code is correct although it forces a value of 1 to become boxed:

```
val n: Any = 1 // Ok: Int is a subtype of Any
```

Integer types

There are four basic Kotlin types representing integer numbers (*Table 2.2*):

Counterpart	Size (in bytes)	Range	Java Counterpart
Byte	1	-128 .. 127	byte
Short	2	-32768 .. 32767	short
Int	4	-231 .. 231 - 1	int
Long	8	-263 .. 263 - 1	long

Table 2.2: Integer types

The simplest form of a literal expressing some value of an integer type is just a decimal number:

```
val n = 12345
```

Since Kotlin 1.1 you can, like in Java 7+, put underscores inside numeric literals for better readability. This comes in handy when literals are rather big:

```
val n = 34_721_189
```

A literal itself has either type Int or Long depending on its size. You can, however,

assign literals to variables of smaller types as well, provided they fit into the expected range:

```
val one: Byte = 1                                // OK
val tooBigForShort: Short = 100_000              // Error: too big
for Short
val million = 1_000_000                          // OK: Int is
inferred
val tooBigForInt: Int = 10_000_000_000          // Error: too
big for Int
val tenBillions = 10_000_000_000                  // OK: Long is
inferred
val tooBigForLong = 10_000_000_000_000_000        // Error: too
big for Long
```

Adding l or L suffix forces the literal type to become Long:

```
val hundredLong = 100L                         // OK: Long is inferred
val hundredInt: Int = 100L                      // Error: assigning Long to Int
```

You can also specify literals in binary or hexadecimal numeral system prefixing them with 0b and 0x respectively:

```
val bin = 0b10101 // 21
val hex = 0xF9      // 249
```

Note that numeric literal may not start with 0 unless it's 0 itself. Some programming languages (including Java) use zero-prefixed literals to denote octal numbers which are not supported in Kotlin as they're rarely useful and are often a misleading feature. For that reason, zero-prefixed numbers are forbidden so as to not confuse developers accustomed to the octal notation.

```
val zero = 0           // OK
val zeroOne = 01 // Error
```

Negative numbers like -10 are not technically literals, but rather unary-minus expressions applied to literals:

```
val neg = -10
val negHex = -0xFF
```

Each integer type defines a pair of constants containing its minimum (`MIN_VALUE`) and maximum (`MAX_VALUE`) values. To use them, just qualify constant with type name:

```
Short.MIN_VALUE      // -32768  
Short.MAX_VALUE      // 32767  
Int.MAX_VALUE + 1   // -2147483648 (integer overflow)
```

Floating-Point types

Like Java, Kotlin provides support for IEEE 754 floating-point numbers with types `float` and `double`. Their Java counterparts, as you might've guessed, are `float` and `double`, respectively.

The simplest form of a floating-point literal is a decimal number which consists of integer and fractional parts, separated by a dot:

```
val pi = 3.14  
val one = 1.0
```

The integer part may be empty in which case it's assumed to be zero. The fractional part, however, is mandatory:

```
val quarter = .25 // 0.25  
val one = 1.          // Error  
val two = 2       // No error, but that's integer literal
```

Kotlin also supports scientific notation where you follow decimal number with an exponent part signifying the power of ten:

```
val pi = 0.314E1      // 3.14 = 0.314*10  
val pi100 = 0.314E3     // 314.0 = 0.314*1000  
val piOver100 = 3.14E-2 // 0.0314 = 3.14/100  
val thousand = 1E3        // 1000.0 = 1*1000
```

Note that in scientific notation, fractional part is optional.

Java vs. Kotlin: Unlike Java 6+, Kotlin doesn't support hexadecimal literals for `Float` and `Double` types.

By default, literals have the `double` type. Tagging them with `f` or `F` forces the type to

float: (in this case fractional part is also optional):

```
val pi = 3.14f  
val one = 1f
```

Java vs. Kotlin: In Java you can tag a literal with d or D, forcing its type to be Double (like 1.25d). Kotlin, however, has no such suffix: Double type is only assigned by default.

Note that float literals are not automatically converted to Double type. The following code will lead to compile-time error:

```
val pi: Double = 3.14f // Error
```

Float and double define a set of constants representing some special values of these types:

- MIN_VALUE, MAX_VALUE: The smallest/the largest positive finite value representable by the type;
- NEGATIVE_INFINITY, POSITIVE_INFINITY: Negative/positive infinite value, which is respectively the smallest/the largest value of the type;
- NaN: not a number value for uncertainties like 0/0.

```
println(Float.MIN_VALUE)           // 1.4E-45  
println(Double.MAX_VALUE)          //  
1.7976931348623157E308  
println(Double.POSITIVE_INFINITY)    // Infinity  
println(1.0/Double.NEGATIVE_INFINITY) // -0.0  
println(2 - Double.POSITIVE_INFINITY) // -Infinity  
println(3 * Float.NaN)            // NaN
```

Arithmetic operations

All numeric types support basic arithmetic operations:

Operation	Meaning	Examples	
+ (unary)	The same value	+2	// 2
- (unary)	Opposite value	-2	// -2
+	Addition	2 + 3 // 5 2.5 + 3.2 // 5.7	

-	Subtraction	1 - 3 // -2 3.4 - 1.8 // 1.6
*	Multiplication	3 * 4 // 12 3.5 * 1.5 // 5.25
/	Division	7/4 // 1 -7/4 // -1 7/(-4) // -1 (-7)/(-4) // 1 6.5/2.5 // 2.6 -6.5/2.5 // -2.6 6.5/(-2.5) // -2.6 (-6.5)/(-2.5) // 2.6
%	Remainder	7%4 // 3 -7%4 // -3 7%(-4) // 3 (-7)%(-4) // -3 6.5%2.5 // 1.5 -6.5%2.5 // -1.5 6.5%(-2.5) // 1.5 (-6.5)%(-2.5) // -1.5

Table 2.3: Arithmetic operations

The behavior of arithmetic operations is consistent with Java. Note that integer division operations give a result rounded to zero, while remainder has the same sign as the numerator. Floating-point operations are carried out according to IEEE 754 specification.

Numeric types support `++`/`--` operations which amount to increasing/decreasing the value by 1.

The result of unary `+-` operations has the same type as their argument with an exception for `Byte` and `Short` for which it's `Int`:

```
val byte: Byte = 1
val int = 1
val long = 1L
val float = 1.5f
val double = 1.5

-byte      // -1: Int
-int      // -1: Int
```

```
-long      // -1: Long
-float     // -1.5: Float
-double   // -1.5: Double
```

Each of the binary arithmetic operations comes in multiple variants covering all possible combinations of numeric types. Since there are 6 numeric types, it means $6^6 = 36$ versions for each operation. It allows combining different numeric types in arithmetic expressions without an explicit conversion. The result of such operation is taken to be a bigger one between its arguments where bigger means:

Double > Float > Long > Int > Short > Byte

For most types, it essentially means larger set of values, but that's not always the case: the obvious example is conversion from Long to Float which can lead to loss of precision. Note that the result type is never smaller than Int even when arguments have Byte or Short type.

Going ahead with our previous example we get:

```
byte + byte      // 2: Byte
int + byte       // 2: Int
int + int        // 2: Int
int + long       // 2: Long
long + double    // 2.5: Double
float + double   // 3.0: Double
float + int      // 2.5: Float
long + double    // 2.5: Double
```

Bitwise operations

Int and Long support a range of bit-level operations:

Operation	Meaning	Examples	Java Counterpart
Shl	Shift left	<pre>// 13: 0...00001101 13 shl 2 // = 52: 0... 00110100 // -13: 1...11110011 (-13) shl 2 // = -52: 1...11001100</pre>	<<
Shr	Shift right	<pre>// 13: 0...00001101 13 shr 2 // = 3: 0...00000011</pre>	>>

		// -13: 1...11110011 (-13) shr 2 // = -4: 1...11111100	
ushr	Shift right unsigned	// 13: 0...00001101 13 ushr 2 // = 3: 0...00000011 // -13: 1...11110011 (-13) ushr 2 // = 1073741820: 001...111100	>>>
And	Bitwise AND	// 13: 0...00001101 // 19: 0...00010011 13 and 19 // = 1: 0...00000001 // -13: 1...11110011 // 19: 0...00010011 -13 and 19 // = 19: 0...00010011	&
Or	Bitwise OR	// 13: 0...00001101 // 19: 0...00010011 13 or 19 // = 31: 0...00011111 // -13: 1...11110011 // 19: 0...00010011 -13 and 19 // = -13: 1...11110011	
xor	Bitwise XOR	// 13: 0...00001101 // 19: 0...00010011 13 xor 19 // = 30: 0...00011110 // -13: 1...11110011 // 19: 0...00010011 -13 xor 19 // = -32: 1...11100000	^
Inv	Bitwise inversion	// 13: 0...00001101 13.inv() // = -14: 1...11110010 // -13: 1...11110011 (-13).inv() // = 12: 0...00001100	~

Table 2.4: Bitwise operations

Note that `inv` is not a binary operation, but rather a simple method which is invoked using dot notation.

Since Kotlin 1.1 `and`, `or`, `xor` and `inv` are also available on `Byte` and `Short`.

Java vs. Kotlin: If you're familiar with Java you might know about the bitwise operators: `&`, `|`, `^`, `~`, `<<`, `>>`, and `>>>`. These operators are not currently supported in Kotlin: you have to use `and`, `or`, `xor`, `inv`, `shl`, `shr` and `ushr`, respectively, which have exactly the same semantics on JVM.

Char type

Char type represents a single 16-bit Unicode character. A literal of this type is just a character itself surrounded by single quotes:

```
val z = 'z'  
val alpha = 'α'
```

For a special character like newlines, Kotlin provides a set of escape sequences: \t (tab), \b (backspace), \n (newline), \r (carriage return), \' (single quote), \\" (double quote), \\ (backslash), \\$ (dollar sign):

```
val quote = '\''  
val newLine = '\n'
```

You can also put arbitrary Unicode characters into the literal using \u sequence followed by 4-digit hexadecimal character code:

```
val pi = '\u03C0' // π
```

Although internally, Char value is just a character code, the Char itself is not considered a numeric type in Kotlin. It does, however, support a limited set of arithmetic operations concerned with moving around the Unicode character set. This is what you can do with characters:

- Add/remove an integer with +/- operators which give a character shifted by corresponding number of steps;
- Subtract two characters getting number of steps between them;
- Increment/decrement a character with ++/-- operators;

Let's consider some examples:

```
var a = 'a'  
var h = 'h'  
/* 5th character after 'a' */      println(a + 5) // f  
/* 5th character before 'a' */     println(a - 5) // \  
/* distance between 'a' and 'h' */   println(h - a) // 7  
/* get character preceding 'h' */    println(--h)      // g  
/* get character following 'a' */   println(++a)      // b
```

Java vs. Kotlin: Note that while in Java results of all arithmetic operations on characters are implicitly converted to int, Char operations in Kotlin (except the difference of two characters) give Char as their result.

Numeric conversions

Each numeric type defines a set of operations to convert its value to any other numeric types as well as Char. Operations have self-explanatory names corresponding to the target type: `toByte()`, `toShort()`, `toInt()`, `toLong()`, `toFloat()`, `toDouble()` and `toChar()`. The same set of operations is available for Char values.

Java vs. Kotlin: Unlike Java, values of smaller-range types can not be used in the context where larger type is expected: you can't, for example, assign an Int values to Long variable. Following code will produce a compilation error:

```
val n = 100          // Int
val l: Long = n // Error: can't assign Int to Long
```

The reason behind this is an implicit boxing we've mentioned above. Since in general values of Int (or any other numeric type) are not necessarily represented as primitives, such widening conversions could potentially amount to producing a value of different boxing type, thus violating equality and leading to subtle errors. If the code above were considered correct, the instruction

```
println(l == n)
```

would print false, which is a rather unexpected result. In Java there is a similar issue related to boxing types:

```
Integer n = 100;
Long l = n;           // Error: can't assign Integer to Long
```

Conversion between integer types is lossless when target type has a larger range. Otherwise, it basically truncates extra most significant bits and reinterprets remainder as a target type. The same also goes for a conversion to/from Char type:

```
val n = 945
println(n.toByte())    // -79
println(n.toShort())  // 945
println(n.toChar())   // α
println(n.toLong())   // 945
```

Conversions involving floating-point types can, in general, lead to precision loss regardless of the target type: say, converting a very big Long value to Float can zero some lower digits. Converting floating-point numbers to integers is basically the same as rounding to zero.

```
println(2.5.toInt())           // 2
println((-2.5).toInt())        // -2
println(1_000_000_000_000.toFloat().toLong()) // 999999995904
```

Boolean type and logical operations

Kotlin has a Boolean type representing a logical operation which can be either true or false:

```
val hasErrors = false;
val testPassed = true;
```

Like in Java, Kotlin's Boolean is distinct from numeric types and can't be converted to numbers (and vice versa) neither implicitly, nor with some kind of built-in operations like `toInt()`. Developer is supposed to use comparison operations and conditionals (see below) to build Boolean values from non-Boolean ones.

Operations supported by Boolean include:

- `!`: Inversion;
- `or`, `and`, `xor`: Eager disjunction/conjunction and exclusive disjunction;
- `||`, `&&`: Lazy disjunction/conjunction.

Lazy operations have essentially the same semantics as their Java counterparts. Operation `||` does not evaluate its right argument if the first one is true. Similarly, operation `&&` doesn't evaluate its right arguments if the first one is false. This can be useful if evaluating right argument entails some side effects.

Java vs. Kotlin: Unlike Java, Kotlin doesn't have `&` and `|` operators. Their role is performed by `and` / `or` respectively.

Let's consider some examples using equality/inequality operations `==` and `!=` (more on them in the next section):

```
println((x == 1) or (y == 1))      // true
println((x == 0) || (y == 0))       // false
```

```
println((x == 1) and (y != 1))      // true
println((x == 1) and (y == 1))        // false
println((x == 1) xor (y == 1))        // true
println((x == 1) xor (y != 1))        // false
println(x == 1 || y/(x - 1) != 1)     // true
println(x != 1 && y/(x - 1) != 1)    // false
```

In the last two examples, using lazy operations is essential since an attempt to evaluate right argument when $x == 1$ will result in runtime error due to division by zero.

Note the precedence difference between eager/lazy conjunction and disjunction. Eager operations `and`, `or`, `xor` are named infix operators and thus have the same precedence and dominate `&&` operation which, in turn, dominates `||`. For example, the following expression:

```
a || b and c or d && e
```

is evaluated as:

```
a || (((b and c) or d) && e)
```

In doubtful cases we suggest using parentheses to clarify the meaning behind your code.

Comparison and equality

All types we've considered so far support standard set of comparison operations: `==` (`equals`), `!=` (`not equals`), `<` (`less than`), `<=` (`less than or equals`), `>` (`greater than`), `>=` (`greater than or equals`):

```
val a = 1
val b = 2
println(a == 1 || b != 1) // true
println(a >= 1 && b < 3) // true
println(a < 1 || b < 1) // false
println(a > b) // false
```

In general, equality operations `==` and `!=` are applicable to values of any type. There is, however, an exception for numeric types, `Char` and `Boolean`. Consider the following code:

```
val a = 1                  // Int
val b = 2L                 // Long
println(a == b)            // Error: comparing Int and Long
println(a.toLong() == b)   // Ok: both types are Long
```

Basically for such types Kotlin only permits `==` and `!=`, when both arguments are of the same type: you can't, for example, apply `==` when one argument is `Int` while another is `Long`. This is explained by the same reasoning we've seen for assignments: equality check would produce different results depending on whether values are boxed, and since boxing in Kotlin is implicit, it could lead to confusion if permitted for any pair of types.

Operations `<`, `<=`, `>`, `>=`, however, allow to compare any numeric types: just like arithmetic operations they are overloaded to cover all possible cases. So, you can write, for example:

```
1 <= 2L || 3 > 4.5
```

Note that `Char` and `Boolean` values also supports comparison operations, but can be compared only with a value of the same type:

```
false == true // false
false < true   // true
false > 1      // Error: comparing Boolean and Int
'a' < 'b'       // true
'a' > 0        // Error: comparing Int and Char
```

Note that `false` is assumed to be less than `true`, and `Char` values are ordered by their character code.

Java vs. Kotlin: In Java where boxed and unboxed values are represented by different types (such as `long` vs. `Long`), primitive numeric types (including `char`) can be freely compared with each other using `==/!=` as well as `</<=/>/>=` operators. Boolean values in Java, however, are not ordered and can only be checked for equality.

In the context of floating-point types comparison operations follow IEEE 754 standard. This, in particular, assumes specific treatment of `Nan` values:

```
println(Double.NaN == Double.NaN)           // false
println(Double.NaN != Double.NaN)          // true
println(Double.NaN<= Double.NaN)           // false
```

```
println(Double.NaN<Double.POSITIVE_INFINITY) // false  
println(Double.NaN>Double.NEGATIVE_INFINITY) // false
```

Basically, NaN is not equal to anything, including itself is considered neither less, nor greater than any other value, including infinities.

These rules, however, are only put in action when compiler knows statically that the value of interest has a floating-point type. In more generic cases which involve, for example, storing numbers in a collection, the compiler falls back to using equality and comparison rules imposed by boxed type. On JVM, this is equivalent to comparing instances of Double/Float wrapper types:

```
val set = sortedSetOf(Double.NaN,  
Double.NaN, Double.POSITIVE_INFINITY,  
Double.NEGATIVE_INFINITY, 0.0)  
println(set) // [-Infinity, 0.0, Infinity, NaN]
```

The code above creates a tree sorted internally by natural ordering of element type (on JVM this is basically a TreeSet) and prints its items. The output show that in this case:

- NaN is equal to itself, since only one such value was added to the set;
- NaN is considered the largest value of Double (even greater than positive infinity).

In the upcoming chapters we'll discuss the concepts of equality and ordering in more detail.

Strings

The String type represents strings of characters. Like in Java, Kotlin strings are immutable: in other words you can't change characters once a String object is created, only read them or create new strings based on the existing one. In this section we'll consider how to construct new strings and perform some basic manipulations with them.

String templates

The simplest way to define a string literal, as we've already seen in this chapter, is to enclose its content in double quotes just like in Java:

```
val hello = "Hello, world!"
```

If the string must contain some special symbols like newline character, you have to use one of the escape sequences (see the *Char type* section):

```
val text = "Hello, world!\nThis is \"multiline\" string"  
println("\u03C0 \u2248 3.14") // π ≈ 3.14
```

These literals are basically the same as in Java. In addition to them, Kotlin provides a much more powerful way to define a string which comes in useful when you want to compose it from various expressions. Suppose, for example, that we want to welcome the user with a message saying hello and printing current date and time:

```
import java.util.Date  
fun main() {  
    val name = readLine()  
    println("Hello, $name!\n Today is ${Date()}")  
}
```

Basically, you can replace any part of the string with a valid Kotlin expression by putting it inside `{}$`. If expression is a simple variable reference, like `name` in our example, you may just prefix it with a dollar sign. Such literal is called a string template.

Note that expressions in string templates may take any value: they are automatically converted to strings using `toString()` method available for any Kotlin type.

If you run the program and enter some name (say, `John`) you'll see something like this:

```
Hello, John!  
Today is Sat Dec 28 14:44:42 MSK 2019
```

The result may vary depending on your locale.

The import directive we've used in first line allows us to refer to the JDK class `Date` by its simple name instead of `java.util.Date`. In the next chapter we'll discuss the issue of imports and packages in more details.

One more variety of string literals is so called raw string. It allows you to write strings without escape sequences. Such literal is enclosed by triple quotes and may contain arbitrary characters, including newlines

```
val message = """  
Hello, $name!
```

```
Today is ${Date()}\n""".trimIndent()
```

The `trimIndent()` is a standard Kotlin function which removes common minimal indent.

In the rare cases when you still want to put some special character sequence into a raw string (like triple quotes), you have to embed them into `{}:`:

```
val message = """\n    This is triple quote:'${"\\""\\""}'\n""".trimIndent()
```

In JVM-targeted applications strings are represented by instances of JVM String class.

Basic String operations

Every `String` instance has `length` and `lastIndex` properties which contain the number of characters and last character index, respectively:

```
"Hello!".length          // 6\n"Hello!".lastIndex // 5 since indices start from zero
```

You can also access individual characters using indexing operator with zero-based index inside brackets. On JVM, passing invalid index will produce `StringIndexOutOfBoundsException` similarly to Java.

```
val s = "Hello!"\nprintln(s[0])      // H\nprintln(s[1])      // e\nprintln(s[5])      // !\nprintln(s[10]) // invalid index
```

You can concatenate strings using operator `+`. Second argument may, in fact, be any value which is automatically converted to string by calling its `toString()` function. In most cases, though, we suggest using string templates instead as they usually are more concise.

```
val s = "The sum is: " + sum // can be replaced by "The sum\nis $sum"
```

Strings can be compared for equality with operators `==` and `!=`. These operators

compare string content, so even two different instances are considered equal if they contain the same sequence of characters:

```
val s1 = "Hello!"  
val s2 = "Hel" + "lo!"  
println(s1 == s2) // true
```

Java vs. Kotlin: In Java == and != operators check referential equality, so you have to use equals() method to compare actual string content. In Kotlin == is basically a more convenient synonym for equals(), so usually there is no need to call equals() directly. Null-checking aside, the code above is equivalent to Java's s1.equals(s2). What about referential equality in Kotlin? For that you can use === and !== operators.

Strings are lexicographically ordered, so you can compare them using operators <, >, <= and >=:

```
println("abc" < "cba") // true  
println("123" > "34") // false
```

String also supports conversion functions for numeric types and Boolean: toByte(), toShort(), toInt(), toLong(), toFloat(), toDouble(), toBoolean(). Note that numeric conversions will produce a run-time error if string doesn't contain a well-formed number.

Here is a list of some additional useful functions String is able to offer:

isEmpty isNotEmpty	Check if string is empty	"Hello".isEmpty() // false "".isEmpty() // true "Hello".isNotEmpty() // true
substring	Extract substring	"Hello".substring(2) // "llo" "Hello".substring(1, 3) // "el" startsWith
endsWith	Check prefix/suffix	"Hello".startsWith("Hel") // true "Hello".endsWith("lo") // true
indexOf()	Get first occurrence index of character or substring	// search from start "abcabc".indexOf('b') // 1 "abcabc".indexOf("ca") // 2 "abcabc".indexOf("cd") // -1 // search from given index "abcabc".indexOf('b', 2) // 4 "abcabc".indexOf("ab", 2) // 3

Throughout the book (and *Chapter 7, Exploring Collections and I/O*, in particular) we'll see more examples of Kotlin String API. For more information we advise you to visit a documentation page at kotlinlang.org/api/latest/jvm/stdlib/kotlin/-string/index.html.

Arrays

Array is a built-in Kotlin data structure which allows you to store a fixed number of same-typed values and refer to them by index. They are conceptually similar to arrays in Java which are in fact used as their representation in Kotlin/JVM applications. In this section we'll consider how to define an array and access its data.

Constructing an array

The most general Kotlin type implementing an array structure is `Array<T>`, where `T` is a common type of its elements. In *Chapter 1, Kotlin: Powerful and Pragmatic*, we've already seen an example of `main()` function accepting a parameter of type `Array<String>` which holds command-line arguments passed to the program. If number of elements is known in advance, we can create an array using one of standard functions:

```
val a = emptyArray<String>()           // Array<String>
(zero elements)
val b = arrayOf("hello", "world") // Array<String> (2
elements)
val c = arrayOf(1, 4, 9)           // Array<Integer> (3
elements)
```

These functions are generic, meaning they refer to unknown element types which must be specified in the call. Thanks to type inference, however, compiler can figure out the unknown type in the second and third calls using arguments we pass: if we, for example, create array from a series of integer numbers, it obviously has `Array<Integer>` type. In the first call, however, compiler has no such information, so we have to specify element type in angular brackets of the call expression. For now, we'll just take this syntax for granted and postpone the detailed discussion of generic types and functions till the *Chapter 9, Generics*.

There is a more flexible way to create an array by describing how to compute an element with a given index. The code below generates an array containing squares of

integers from 1 to whatever the user has entered:

```
val size = readLine()!!.toInt()
val squares = Array(size) { (it + 1)*(it + 1) }
```

The construct inside braces, also called lambda, defines an expression to compute element value based on its index, which is represented by automatically declared variable `it`. Since array indices range from 0 to `size - 1`, its elements will take form 1, 4, 9 and so on. For now, we'll just use this syntax as is and come back to lambdas in the *Chapter 5, Leveraging Advanced Functions and Functional Programming*.

Using `Array<Int>` is a working, but an impractical solution since it will force the boxing of numbers. For that reason, Kotlin provides more efficient storage with specialized array types such as `ByteArray`, `ShortArray`, `IntArray`, `LongArray`, `FloatArray`, `DoubleArray`, `CharArray` and `BooleanArray`. On JVM, these types are represented by Java primitive arrays such as `int[]` or `boolean[]`. Each of them is accompanied by functions similar to `arrayOf()` and `Array()`:

```
val operations = charArrayOf('+', '-', '*', '/', '%')
val squares = IntArray(10) { (it + 1)*(it + 1) }
```

Java vs. Kotlin: Unlike Java, Kotlin doesn't have `new` operator, so a construction of array instance looks like an ordinary function call. Note also that in Kotlin you have to explicitly initialize array elements on its creation.

Using arrays

Array types are quite similar to `String`. In particular, they have `size` (analogous to `String's length`) and `lastIndex` properties and their elements can be accessed by indexing operator. Using invalid index will produce `IndexOutOfBoundsException` at runtime:

```
val squares = arrayOf(1, 4, 9, 16)
squares.size           // 4
squares.lastIndex // 3
squares[3]        // 16
squares[1]         // 4
```

Unlike string characters, though, array elements can be changed:

```
squares[2] = 100 // squares: 1, 4, 100, 16
squares[3] += 9    // squares: 1, 4, 100, 25
squares[0]--        // squares: 0, 4, 100, 25
```

Note that like in Java, array variable itself stores a reference to actual data. For that reason, assigning array variables basically shares the same set of data between variables:

```
val numbers = squares
numbers[0] = 1000      // mutates data shared between squares
and numbers
println(squares[0]) // prints 1000
```

If you want to create a separate array, use `copyOf()` function which can also produce array of a different size if necessary:

```
val numbers = squares.copyOf()
numbers[0] = 1000 // squares is not affected
squares.copyOf(2) // truncated: 1, 4
squares.copyOf(5) // padded with zeros: 1, 4, 9, 16, 0
```

Note that arrays with different types can't be assigned to each other. The following code will fail with a compilation error:

```
var a = arrayOf(1, 4, 9, 16)
a = arrayOf("one", "two") // Error: can't assign
Array<String> to Array<Int>
```

Java vs. Kotlin: In Java you can assign array of subtype to an array of its super type. Since arrays are mutable this can lead to problems at runtime:

```
Object[] objects = new String[] { "one", "two", "three" };
objects[0] = new Object(); // fails with ArrayStoreException
```

For that reason, Kotlin array type is not considered a subtype of any other array type (apart from itself) and such assignments are prohibited. So even though `String` is a subtype of `Any`, `Array<String>` is not a subtype of `Array<Any>`:

```
val strings = arrayOf("one", "two", "three")
val objects: Array<Any> = strings // Error
```

In fact, that's a specific case of a powerful variance concept which we'll cover in *Chapter 9, Generics*.

Although array length can't change after it's been created, you can produce new array by adding extra elements with + operation:

```
val b = intArrayOf(1, 2, 3) + 4           // add single  
element: 1, 2, 3, 4  
val c = intArrayOf(1, 2, 3) + intArrayOf(5, 6) // add another  
array: 1, 2, 3, 5, 6
```

Unlike strings, == and != operators on arrays compare references rather than elements themselves:

```
intArrayOf(1, 2, 3) == intArrayOf(1, 2, 3) // false
```

If you want to compare array content, you should use `contentEquals()` function:

```
intArrayOf(1, 2, 3).contentEquals(intArrayOf(1, 2, 3)) //  
true
```

IDE Tips: IntelliJ issues a warning when you try to compare arrays using == or != and suggests to replace it with `contentEquals()` call instead.

Some standard function which may be helpful when using arrays:

<code>isEmpty</code> <code>isNotEmpty</code>	Check if array is empty	<code>intArrayOf(1, 2).isEmpty()</code> // false <code>intArrayOf(1, 2).isNotEmpty()</code> // true
<code>indexOf</code>	Get first index of an array item	<code>intArrayOf(1, 2, 3).indexOf(2)</code> // 1 <code>intArrayOf(1, 2, 3).indexOf(4)</code> // -1

In the upcoming chapter we'll consider additional array functions. They will mostly be introduced in *Chapter 7, Exploring Collections and I/O*, which deals with Kotlin collections API.

Conclusion

In this chapter we've got the first taste of Kotlin: we've learned about variables and type inference and acquired an understanding of basic types as well as fundamental operations on numbers, characters and Booleans as well as constructing and manipulating more complex data in the form of strings and arrays. We've also seen

examples of how Kotlin design helps to avoid common programming mistakes known in the Java world. Having built this foundation, we're ready to take the next step. In *Chapter 3, Defining Functions*, we'll learn Kotlin control structures and how to structure your code with functions and packages.

CHAPTER 3

Defining Functions

The central topic of this chapter is a concept of function. We'll learn the basic function anatomy and address some important issues such as using named arguments, default values, and vararg-style functions. We'll also introduce you to the imperative control structure of the Kotlin language. The chapter will show you how to implement binary and multiple choices with if and when statements, discuss various forms of iteration and error handling. We'll see that many of these constructs are quite similar to the ones employed by Java (and, in fact, many other programming languages supporting an imperative paradigm) and learn the major differences that will ease migration to Kotlin for developers with Java experience. One more topic of interest is grouping related declarations with packages and using import directives for cross-package references.

Structure

- The anatomy of a Kotlin function: definition and call syntax, specifics of function parameters
- Control structures: conditionals, loops, error handling
- A Kotlin package structure and imports

Objective

The aim of this chapter is to get you acquainted with fundamentals of imperative programming in Kotlin using conditional, iterative, and error-handling control structures as well as the means to structure your code using functions and packages.

Functions

Similar to Java's methods, a Kotlin function is a reusable block of code that accepts some input data (called parameters) and may return an output value to its calling code. In this section, we'll learn to define functions and have a look at their anatomy.

The Anatomy of a Kotlin Function

Let's begin with a simple example and define a function that computes an area of a circle with a given radius:

```
import kotlin.math.PI

fun circleArea(radius: Double): Double {
    return PI*radius*radius
}

fun main() {
    print("Enter radius: ")
    val radius =readLine()!!.toDouble()
    println("Circle area: ${circleArea(radius)}")
}
```

Note that we've used a standard constant PI, which denotes an approximate value of π . The import directive at the start allows us to refer to PI by its simple name.

Now, let's look closely at what makes up the `circleArea` definition:

- The `fun` keyword (from function), which tells a compiler that what follows is a function definition;
- The function name, `circleArea`, which, like variable name, can be an arbitrary identifier;
- The comma-separated parameter list enclosed in parentheses: this tells a compiler which data can be passed to our function upon its call;
- The return type, `Double`, which is a type of value returned to the function caller;
- The function body that is enclosed in `{ }` block and describes implementation of our function.

Note that parentheses in a function definition and its call are mandatory even if the function has no parameters. For example:

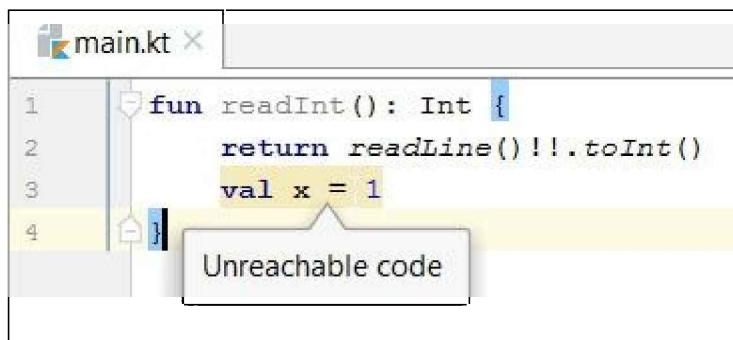
```
fun readInt(): Int {
    return readLine()!!.toInt()
}

fun main() {
    println(readInt())
```

```
}
```

Similar to Java, the function result is specified by the return statement, which terminates its execution passing control back to the caller. Any code placed after the return statement is effectively dead, that is, never gets executed.

Java versus Kotlin: As opposed to Java, the unreachable code in Kotlin is not a compile-time error. The compiler will, however, report a warning, and the IDE provides highlighting, which clearly marks which portion of your code is dead as shown on the *Figure 3.1*:



The screenshot shows a code editor window titled "main.kt". The code is as follows:

```
1 fun readInt(): Int {
2     return readLine() !!.toInt()
3     val x = 1
4 }
```

A yellow rectangular highlight covers the entire body of the function, from the opening brace at line 1 to the closing brace at line 4. Inside this highlight, the line "val x = 1" is also highlighted with a yellow background. A tooltip box labeled "Unreachable code" is positioned over the line "val x = 1".

Figure 3.1: Highlighting of the Unreachable code

In Kotlin, similar to Java, you have a block statement, which is basically a group of statements enclosed in { }. Statements are separated by either new line (which is the preferred style), or semicolons and are executed sequentially.

We've already employed blocks when writing a body of function, but in fact, they're used whenever we need to perform multiple statements in a context which syntactically requires just one – like a body of a loop or a branch of some conditional statement. Blocks are also used to improve code readability; for example, a loop body is often enclosed in block even when it consists of a single statement.

Block can contain definitions of local variables and functions. The scope of such declarations is limited to the block itself.

The parameter definition is basically an implicit local variable that is automatically initialized to the value passed in its call before executing the body.

Java versus Kotlin: Unlike Java's method parameters which are mutable by default and must be marked with the final modifier to forbid further changes in method body, Kotlin parameters are immutable. In other words, changing the parameter value inside a function body is a compilation error:

```
fun increment(n: Int): Int {  
    return n++ // Error: can't change immutable variable  
}
```

Note also that marking parameter with the `val` or `var` keyword is forbidden. The reason behind this is that parameter assignments are considered error-prone, while using parameters as immutable values lead to more clean and understandable code.

Kotlin follows call-by-value semantics, which means the parameter values are copied from the respective call arguments. In particular, it means that changes to some variable passed as the call argument (like `radius` in the main function above) do not affect the value of the parameter inside the called function. However, when parameter is a reference – for example, has an array type – what gets copied is only reference itself while the data behind it gets shared between the function and its caller. So, even though parameters themselves can't change inside the function, the data they reference in general may be mutable. For example, the following function takes an (immutable) reference to an integer array and modifies its first element affecting its caller as well:

```
fun increment(a: IntArray): Int {  
    return ++a[0]  
}  
  
fun main() {  
    val a = intArrayOf(1, 2, 3)  
    println(increment(a)) // 2  
    println(a.contentToString()) // [2, 2, 3]  
}
```

Note that unlike variables, the parameters always have explicit type since compilers cannot infer it from the function definition.

The return type, on the contrary, can be inferred from the types of function parameters, but still must be specified explicitly. The rationale behind this decision is that functions often have more than one exit point where their result is determined and it may be difficult for a programmer to understand the kind of value returned just by looking at the function definition. In this sense, explicit return types serve as a kind of documentation, which immediately tells you about values the function can produce.

This rule, however, has two exceptions that allow you to omit the return type in some cases. The first one is a function of the so-called unit type, which is a Kotlin

counterpart for Java's void and basically signifies a function that doesn't need a meaningful return value. The actual return value of such a function is a constant unit, which is also the single value of the built-in Unit type. If you skip the return type in your function definition, the Kotlin compiler automatically assumes that you're declaring a Unit function. In other words, the following definitions are equivalent to each other:

```
fun prompt(name: String) {  
    println("***** Hello, $name! *****")  
}  
  
fun prompt(name: String): Unit {  
    println("***** Hello, $name! *****")  
}
```

We've already seen such functions through the example of `main()`. Note that Unit functions do not need the `return` statement to specify their result, since it's always the same. You can, however, use the `return` statement to terminate function execution before it reaches the end of its body (the `return Unit` is valid, but redundant in this case).

Another exception is a so-called expression-body functions. If function can be implemented by a single expression, we can drop the `return` keyword and braces and write it in the following form:

```
fun circleArea(radius: Double): Double = PI*radius*radius
```

The preceding syntax is similar to a variable definition where you specify the initializing expression after the `=` symbol. Like variables, expression-body functions allow you to omit the result type:

```
fun circleArea(radius: Double) = PI*radius*radius // Double  
is inferred
```

Expression-body functions are considered simple enough to spare explicit type specification. But this feature should be used with care. Complex expressions are often worth to be written in a usual block form for better readability.

Note that if you try placing the `{}` block after the `=` sign to define a block-body function, you'll not get the expected result since the block in such a position is interpreted as a lambda (basically, a simplified syntax for an anonymous function).

Consider, for example, the following function:

```
fun circleArea(radius: Double) = { PI*radius*radius }
```

The definition above corresponds to a valid function which returns another function computing the circle area for a fixed value of radius. A similar definition with a return inside a block, on the other hand, will produce a compile-time error. For example:

```
fun circleArea(radius: Double) = {  
    return PI*radius*radius // expected function, but returning  
    Double  
}
```

The error is due to the type mismatch and the fact that return, as we'll see in the *Chapter 5, Leveraging Advanced Functions and Functional Programming*, is by default forbidden in lambdas.

Positional versus named arguments

By default, call arguments are mapped to parameters by their positions: the first argument corresponds to the first parameter, second to second, and so on. In Kotlin, such arguments are called positional, as shown in the following code:

```
fun rectangleArea(width: Double, height: Double): Double {  
    return width*height  
}  
  
fun main() {  
    val w = readLine()!!.toDouble()  
    val h = readLine()!!.toDouble()  
    println("Rectangle area: ${rectangleArea(w, h)}")  
}
```

Apart of positional arguments known in Java and many other languages, Kotlin supports the so-called named arguments that are mapped to parameters by explicit names rather than positions. For example, we could have written the call to `rectangleArea()` like this:

```
rectangleArea(width = w, height = h)
```

Or even like this:

```
rectangleArea(height = h, width = w)
```

With the named argument style, the actual argument order is irrelevant. Therefore, both calls have exactly the same semantics as `rectangleArea(w, h)`.

You can also mix positional and named arguments in the same call. Note, however, that once you've used a named argument, all subsequent arguments in the same call must also be named. Consider, for example, a function that swaps two characters in a string (the original string is, of course, unaffected since String values are immutable):

```
fun swap(s: String, from: Int, to: Int): String {
    val chars = s.toCharArray() // convert to array
    // Swap array elements:
    val tmp = chars[from]
    chars[from] = chars[to]
    chars[to] = tmp
    return chars.toString() // Convert back to
}

fun main() {
    println(swap("Hello", 1, 2)) // Hlelo
    println(swap("Hello", from = 1, to = 2)) // Hlelo
    println(swap("Hello", to = 3, from = 0)) // lelHo
    println(swap("Hello", 1, to = 3)) // Hlleo
    println(swap(from = 1, s = "Hello", to = 2)) // Hlelo
    // Incorrect mixing of positional and named arguments
    println(swap(s = "Hello", 1, 2)) // Compilation error
    println(swap(s = "Hello", 1, to = 2)) // Compilation
error
}
```

Overloading and default values

Kotlin functions, like Java methods, can be overloaded: in other words, you can define multiple functions sharing the same name. The overloaded function must have different parameters type so that the compiler can distinguish them while analyzing a call. For example, the following definitions comprise a valid overloading:

```
fun readInt() = readLine()!!.toInt()
```

```
fun readInt(radix: Int) = readLine()!!.toInt(radix)
```

The following pair of functions, however, leads to a compilation error since they only differ in return types:

```
fun plus(a: String, b: String) = a + b  
fun plus(a: String, b: String) = a.toInt() + b.toInt()
```

When choosing the function for a given call expression, the compiler follows an algorithm that is quite similar to the Java overload resolution:

1. Collect all functions which can be called with the given arguments according to parameter count and types.
2. Remove all less specific functions: a function is less specific if all of its parameter types are supertypes of corresponding parameters of some other function in the candidate list. This step is repeated until no less specific functions remain.
3. If the candidate list is reduced to a single function, it's considered the call target; otherwise, the compiler reports an error.

Consider the following function definitions:

```
fun mul(a: Int, b: Int) = a*b // 1  
fun mul(a: Int, b: Int, c: Int) = a*b*c // 2  
fun mul(s: String, n: Int) = s.repeat(n) // 3  
fun mul(o: Any, n: Int) = Array(n) { o } // 4
```

And the results of overload resolution for some calls:

```
mul(1, 2) // Choosing 1 between 1 and 4 since Int is a subtype of Any  
mul(1, 2L) // Error: no overload accepts (Int, Long)  
mul(1L, 2) // Choosing 4 as it's the only acceptable overload  
mul("0", 3) // Choosing 3 between 3 and 4 since String is a subtype of Any
```

If you want to call an overload that is otherwise consider less specific, you can explicitly cast some argument(s) to their supertypes using the type castg operation as shown in the following line:

```
mul("0" as Any, 3) // Choosing 4 as it's the only acceptable  
overload
```

We'll come back to the `as` operation in the *Chapter 8, Understanding Class Hierarchies*, where we'll take a closer look at subtyping and inheritance in Kotlin.

In Java, the overloaded methods often perform the same operation and differ only in the set of their parameters allowing the user to omit one or more arguments in a function call assuming that they take some default values. Looking at the pair of `readInt()` functions defined at the beginning of this section, we see that both of them parse the input string into an integer number with the second one being more general and allowing you to parse numbers in some range of numeral systems, while the first parses only decimals. In fact, we could have rewritten the first function in terms of the second one as:

```
fun readInt() = readInt(10)
```

In Kotlin, you don't need to use overloaded functions in such cases; thanks to a more elegant solution: you just need to specify default values for parameters of interest just as you would specify a variable initializer:

```
fun readInt(radix: Int = 10) = readLine()!!.toInt(radix)
```

Now, you can call this function with either a zero or one argument:

```
val decimalInt = readInt()  
val decimalInt2 = readInt(10)  
val hexInt = readInt(16)
```

Note that if some non-default parameters come after default ones, the only way to call such function with default parameter(s) omitted is to use named arguments:

```
fun restrictToRange(  
    from: Int = Int.MIN_VALUE,  
    to: Int = Int.MAX_VALUE,  
    what: Int  
) : Int = Math.max(from, Math.min(to, what))  
fun main() {  
    println(restrictToRange(10, what = 1))  
}
```

It is, however, considered a good style to put parameters with default values at the end of the parameter list.

Default values somewhat complicate the overloading resolution since some function may be called with different number of arguments. Consider the following definitions:

```
fun mul(a: Int, b: Int = 1) = a*b           // 1
fun mul(a: Int, b: Long = 1L) = a*b         // 2
fun mul(a: Int, b: Int, c: Int = 1) = a*b*c // 3
```

And the corresponding calls:

```
mul(10)          // Error: can't choose between 1 and 2
mul(10, 20)      // Choosing 1 between 1 and 3 as having
fewer parameters
mul(10, 20, 30) // Choosing 3 as the only acceptable
candidate
```

You can see that function 3 is considered less specific than 2 for the two-argument call, `mul(10, 20)`, since it basically extends the second signature by adding a third parameter `c`. If we, however, change, the first definition to:

```
fun mul(a: Number, b: Int = 1) = a*b
```

`mul(10, 20)` will resolve to the third function while the second one will be considered less specific due to the `Number` being a supertype of `Int`.

Varargs

We've already seen several examples of functions such as `arrayOf()`, which accept a variable number of arguments. This feature is also available for the functions you define in your own code. All you need to use it is place the `vararg` modifier before the parameter definition:

```
fun printSorted(vararg items: Int) {
    items.sort()
    println(items.contentToString())
}

fun main() {
    printSorted(6, 2, 10, 1) // [1, 2, 6, 10]
```

```
}
```

Inside the function itself such a parameter is available as an appropriate array type: for example, in case of our `printSorted()`, it's `IntArray`.

You can also pass an actual array instance instead of the variable argument list prefixing it with a spread operator *:

```
val numbers = intArrayOf(6, 2, 10, 1)
printSorted(*numbers)
printSorted(numbers) // Error: passing IntArray instead of
Int
```

Note that the spread creates an array copy; so, changes to elements of `items` parameter do not affect values of `numbers` elements:

```
fun main() {
    val numbers = intArrayOf(6, 2, 10, 1)
    printSorted(*numbers) // [1, 2, 6, 10]
    println(a.contentToString()) // [6, 2, 10, 1]
}
```

The copying, however, is shallow: if array elements are themselves references, copying that reference leads to sharing the data between the function and its caller:

```
fun change(vararg items: IntArray) {
    items[0][0] = 100
}
fun main() {
    val a = intArrayOf(1, 2, 3)
    val b = intArrayOf(4, 5, 6)
    change(a, b)
    println(a.contentToString()) // [100, 2, 3]
    println(b.contentToString()) // [4, 5, 6]
}
```

Declaring more than one parameter as `vararg` is forbidden. Such parameter, however, can accept any mix of command-separated ordinary parameters and spreads. Upon the call, they are merged into a single array preserving the original order:

```
printSorted(6, 1, *intArrayOf(3, 8), 2) // [1, 2, 3, 6, 8]
```

If the `vararg` parameter is not the last one, values for parameters coming after it can only be passed with named arguments notation. Similar to default values, it's considered good style to place the `vararg` parameter at the end of parameter list. `vararg` itself can't be passed as named argument unless you're using spread operator:

```
printSorted(items = *intArrayOf(1, 2, 3))
printSorted(items = 1, 2, 3) // Error
```

Note that parameters having default values do not mix very well with `vararg`. Placing defaults before `vararg` will force first values of `vararg` argument to be interpreted as values of preceding defaults unless you pass `vararg` in a named form which defeats the purpose of using `vararg` in the first place:

```
fun printSorted(prefix: String = "", vararg items: Int) { }
fun main() {
    printSorted(6, 2, 10, 1) // Error: 6 is taken as value of
    prefix
    printSorted(items = *intArrayOf(6, 2, 10, 1)) // Correct
}
```

Placing defaults after the `vararg` parameter, on the other hand, will require you to use named form for the defaults:

```
fun printSorted(vararg items: Int, prefix: String = "") { }
fun main() {
    printSorted(6, 2, 10, 1, "!") // Error: "" is taken as part
    of vararg
    printSorted(6, 2, 10, 1, prefix = "!") // Correct
}
```

`varargs` also affect the overload resolution. All other things being equal, a function with the `varargs` parameter is considered less specific than a function having a fixed number of parameters of the same type. For example, in following code the compiler will prefer the second overload when given a call with three arguments:

```
fun printSorted(vararg items: Int) {} // 1
fun printSorted(a: Int, b: Int, c: Int) {} // 2
fun main() {
    printSorted(1, 2, 3) // Choosing 2 between 1 and 2 as non-
    vararg function
```

```
printSorted(1, 2) // Choosing 1 as the only acceptable  
candidate  
}
```

Function scope and visibility

Kotlin functions can be broken down into three categories depending on where they're defined:

- Top-level functions declared directly in a file
- Member functions declared in some type
- Local functions declared inside of another function.

In this chapter, we'll focus on top-level and local functions while the member functions are postponed till *Chapter 4, Working with Classes and Objects*, where we'll deal with a concept of the Kotlin class.

So far we've only defined top-level functions such as `main()`. By default, these functions are considered public; in other words, they can be used anywhere in the project and not just in their enclosing file. Let's, for example, create two Kotlin files, `main.kt` and `util.kt` in the same directory. We can see that the `main()` function defined in the `main.kt` file calls the `readInt()` function defined in `util.kt` (*Figure 3.2*):

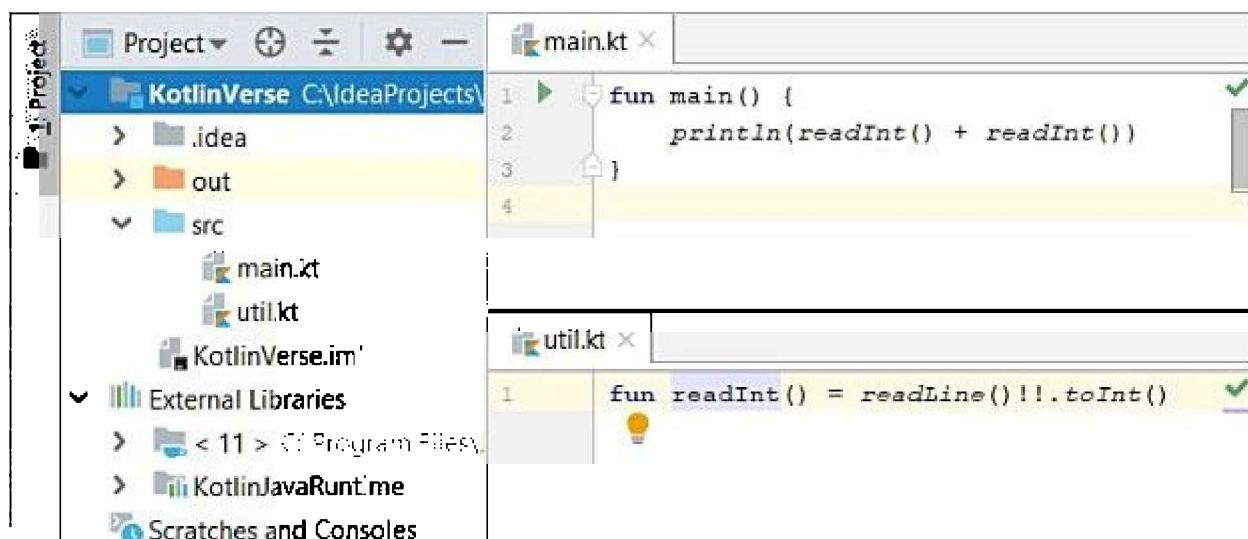


Figure 3.2: Calling a public function from another file

In some cases, you might want to hide some implementation details from other parts of your project, thus narrowing the function scope or the set of places in code where it can be used. To do this, you may prefix a top-level function definition with either of keywords – private or internal – which are called visibility modifiers.

Marking a function as private makes it accessible only in the containing file. For example, if we make `readInt()` private, we can still use it inside `util.kt`, but not from `main.kt` (*Figure 3.3*):

```
Project main.kt util.kt
```

```
fun main() {
    println(readInt() + readInt())
}

private fun readInt() = readLine()!!.toInt()

fun readIntPair() = intArrayOf(readInt(), readInt())
```

Figure 3.3: Calling a private function from another file

Using an internal modifier allows you to restrict function usages to its containing module. A Kotlin module is basically a group of files that are compiled together. Its specific meaning depends on the build system you use to assemble your project. However, in case of IntelliJ IDEA, it corresponds to a single IDE module. So, making a function internal allows you to use it from any other file in the same module, but not from other modules of your project.

IDE Tips: You can create a separate module using File | New | Module wizard similar to the New Project wizard we've discussed in Chapter 1, *Kotlin: Powerful and Pragmatic*.

You can also use a public modifier, but that's consider redundant since top-level functions are public by default.

A local function, like a local variable, is declared inside another function. The scope of such functions is limited to the enclosing code block:

```

fun main() {
    fun readInt() = readLine()!!.toInt()
    println(readInt() + readInt())
}
fun readIntPair() = intArrayOf(readInt(), readInt()) // Error

```

Local functions are able to access declarations available in enclosing functions, including their parameters:

```

fun main(args: Array<String>) {
    fun swap(i: Int, j: Int): String {
        val chars = args[0].toCharArray()
        val tmp = chars[i]
        chars[i] = chars[j]
        chars[j] = tmp
        return chars.toString()
    }
    println(swap(0, chars.lastIndex))
}

```

Note that local functions and variables may not have any visibility modifiers.

Java versus Kotlin: The Java language and JVM, in general, require all methods to be members of some class. So, you're probably wondering how Kotlin top-level and local functions can be compiled on the JVM platform. In *Chapter 1, Kotlin: Powerful and Pragmatic*, we've already seen that from the JVM view-point top-level `main()` function is in fact a static member of a special facade class generated per Kotlin file. For the local functions, the Kotlin compiler performs a similar trick that involves a declaration of special class (you can compare it with a local class in Java), which contains local function as its member and captures its context like variables and parameters of the enclosing function. Note that this implies some performance overhead as your program may need to create a new instance of such class upon every call of the local function. We'll come back to this issue in *Chapter 5, Leveraging Advanced Functions and Functional Programming* while discussing lambdas.

Packages and imports

A Kotlin package is a way to group related declaration. Any package has a name and may be nested into some other package. This concept is very similar to its Java

counterpart but has its own specifics, which we'll highlight in the upcoming sections.

The packages and directory structure

Similar to Java, you can specify the package name at the start of a Kotlin file making the compiler to put all top-level declarations listed in the file into the corresponding package. If the package is not specified, the compiler assumes that your file belongs to a default root package which has an empty name.

The package directive starts with a `package` keyword and contains dot-separated list of identifiers comprising qualified name of the package, which is basically a path to the current package in the project's package hierarchy starting from the root. For example, the following file:

```
package foo.bar.util
fun readInt() = readLine()!!.toInt()
```

Belongs to the package `util`, which is contained in the package `bar`, which, in turn, is contained in the package `foo`, while the following file is put into the package `util` which is contained right in the package hierarchy root:

```
package numberUtil
fun readDouble() = readLine()!!.toDouble()
```

Multiple files share the same package if they have the same package directives. In this case, the package will include all contents of these files combined.

Top-level declarations that comprise a package include types, functions, and properties. We've already acquainted ourselves with a top-level function definition and will see how to define types and properties in the upcoming chapter. Within the same package, you can refer to its declarations using their simple names. This is what we've been doing so far in our examples since all our files were implicitly put into the same root package. *Figure 3.4* shows you a similar example with a non-default package name:

```
util.kt
1 package foo.bar.kotlinVerse
2
3 fun readInt(radix: Int = 10) = readLine()!!.toInt(radix)

main.kt
1 package foo.bar.kotlinVerse
2
3 fun main() {
4     println(readInt(radix: 8))
5 }
```

Figure 3.4: Calling a function from the same package

What if a declaration you want to use belongs to a different package? In this case, you can still refer to it using its qualified name, which is basically a simple name prefixed with a qualified name of the enclosing package (Figure 3.5):

```
util.kt
1 package foo.bar.util
2
3 fun readInt(radix: Int = 10) = readLine()!!.toInt(radix)

main.kt
1 package foo.bar.main
2
3 fun main() {
4     println(foo.bar.util.readInt(radix: 8))
5 }
```

Figure 3.5: Using a qualified name to call a function from a different package

In general, this approach is not practical because it produces hard-to-read code with excessively long names. For this reason, Kotlin provides an import mechanism. By placing an import directive with a qualified declaration name once at the beginning of

your file, you can refer to it using a simple name (*Figure 3.6*).

```
util.kt
1 package foo.bar.util
2
3 fun readInt(radix: Int = 10) = readLine()!!.toInt(radix)

main.kt
1 package foo.bar.main
2
3 import foo.bar.util.readInt
4
5 fun main() {
6     println(readInt(radix = 8))
7 }
```

Figure 3.6: Using import directive

IDE tips: The IntelliJ plugin takes care of many tedious operations with imports. In particular, if you try to use some declaration that is located in another package but refer to it with a simple name only, the IDE will automatically bring up a popup suggesting to import it from relevant packages. It also highlights unused imports and allows you to optimize an entire import list by removing unused ones and sorting remaining directives with an **Optimize Imports** command available by the **Ctrl + Alt + O (Command + Alt + O)** shortcut.

Note that package hierarchy is a separate structure inferred purely from the package directives in the source files. It may coincide with a directory structure of the source file tree, but that's not necessary. For example, source files may reside in the same directory but belong to different packages and vice versa.

Java versus Kotlin: In Java, on the contrary, the package structure must be a direct reflection of the source tree directories in the compilation root. Any mismatch is treated as a compilation error.

It is, however, recommended to keep the directory and package structures matched as it simplifies the navigation between different parts of your project.

IDE tips: By default, the IntelliJ plugin enforces package/directory matching and displays a warning whenever it's violated. You've probably noticed that package directives in *Figure 3.6* are highlighted: that's because we've put the package directives that do not match the directory paths. By pressing Alt + Enter within the highlighted area, you can either change the directive itself, or move the containing file into its corresponding directory.

Using import directives

We've already seen how import directives allow you to avoid using qualified names and simplify your code. In this section, we'll look more closely at what kinds of import directives are available in Kotlin and how they differ from their Java counterparts.

The simplest form that we've already seen in the earlier examples allows you to import some specific declaration by specifying its qualified name:

```
import java.lang.Math      // JDK class
import foo.bar.util.readInt // top-level function
```

The import directive is not limited to top-level declarations such as classes or functions. It can also be used to import various member declarations such as nested classes or enum constants as shown in the following example:

```
import kotlin.Int.Companion.MIN_VALUE
fun fromMin(steps: Int) = MIN_VALUE + n // refer to MIN_VALUE
by simple name
```

In *Chapter 4, Working with Classes and Objects* and *Chapter 6, Using Special-Case Classes*, we'll address this issue in more details.

Java vs. Kotlin: Unlike Java, Kotlin doesn't have a separate constructor which imports type members similar to Java's `import static`. All declarations in Kotlin are imported using a general import directive syntax.

In some declarations, residing in different packages may have the same name. What if you need to use them in a single file? Suppose we have two `readInt()` functions in packages `app.util.foo` and `app.util.bar`, respectively. Trying to import them both as shown below won't solve the problem:

```
import app.util.foo.readInt
import app.util.bar.readInt
```

```
fun main() {
    val n = readInt() // Error: can't choose between two
variants of readInt()
}
```

You can always use the qualified name to distinguish between these alternatives, but Kotlin gives you a better solution, which is called an import alias. This feature allows you to introduce a new name for an imported declaration, which has an effect in the scope of the entire file:

```
import foo.readInt as fooReadInt
import bar.readInt as barReadInt
fun main() {
    val n = fooReadInt()
    val m = barReadInt()
}
```

Another form of an import directive allows you to import all declarations from a given scope. To use it you just need to place `*` at the end of qualified name. The syntax is pretty similar to Java as demonstrated by the following line:

```
import kotlin.math.* // import all declarations from
kotlin.math.package
```

Note that such on-demand import has a lower priority than import directive referring to some specific declarations. If we consider our example with two `readInt()` functions but change one of the import directives to on-demand, the specific one takes over, as shown in the following code:

```
import app.util.foo.readInt
import app.util.bar.*
fun main() {
    val n = readInt() // No ambiguity: resolves to
    app.util.foo.readInt
}
```

Conditionals

Conditional statements allow you to choose one of the two or more actions depending on the value of some condition. In Kotlin, they are represented by `if` and `when`

statements which can be roughly compared to Java's `if` and `switch`.

Making decisions with `if` statements

Using an `if` statement, you can select between two alternatives depending on the value of some Boolean expression. It has the same syntax as a similar statement in Java:

```
fun max(a: Int, b: Int): Int {  
    if (a > b) return a  
    else return b  
}
```

Basically, it performs the first statement when condition in parentheses is true and the second one (else-branch) otherwise. Else-branch may be absent in which the case statement does nothing if the condition is false. Each of the two branches may be a block statement which allows to execute multiple statements within the same alternative:

```
fun main(args: Array<String>) {  
    if (args.isNotEmpty()) {  
        val message = "Hello, ${args[0]}"  
        println(message)  
    } else {  
        println()  
    }  
}
```

Note that the condition must be an expression of the Boolean type.

The key difference from the `if` statement in Java is that Kotlin's `if` statement can also be used as an expression. For example, we could have written our `max` function in a simpler form:

```
fun max(a: Int, b: Int) = if (a > b) a else b
```

This is also true when one or both the branches are blocks, in which case the value of the entire conditional statement coincides with the last expression in the corresponding block:

```
fun main() {  
    val s = readLine()!!
```

```

val i = s.indexOf("/")
// Split line like 10/3 into 10 and 3 and perform the
division
val result = if (i >= 0) {
    val a = s.substring(0, i).toInt()
    val b = s.substring(i + 1).toInt()
    (a/b).toString()
} else ""
println(result)
}

```

Note that when `if` is used as an expression, both the branches must be present. The following code won't compile since it misses an `else`-branch:

```
val max = if (a > b) a
```

Java versus Kotlin: Kotlin doesn't have a ternary conditional operator which you might've used in Java. This is, however, mostly mitigated by the fact that `if` can be used as both a statement and an expression.

Sometimes it can be helpful to use `return` in an `if` expression. The `return` statement can be used as an expression of a special type `Nothing` which denotes a non-existing value. Basically, if some expression has the `Nothing` type, it indicates some break in the sequential control-flow of the program since such expression never reaches any definite value. In case of `return`, it means termination of an enclosing function. One useful aspect of the `Nothing` type is that it's considered a subtype of every the Kotlin type and thus, its expressions may be used in any context where expression is expected. Suppose we're given a qualified package name and want to know how it would look like if its simple name was changed. We can implement it in a following way:

```

fun renamePackage(fullName: String, newName: String): String
{
    val i = fullName.indexOf('.')
    val prefix = if (i >= 0) fullName.substring(0, i + 1) else ""
    return prefix + newName
}
fun main() {

```

```
    println(renamePackage("foo.bar.old", "new")) // foo.bar.new
}
```

Note that the value of `e` in `return e` is not the value of return expression, but rather a resulting value of the enclosing function. The return expression itself has no value just like any expression of the type `Nothing`. Also mind the difference between `Unit` and `Nothing`. As opposed to `Nothing`, `Unit` has a single instance which is generally used to denote the absence of any useful value, rather than an absence of any value at all.

Ranges, progressions, and in operation

Kotlin includes several built-in types that represent some interval of ordered values. They are particularly useful for iteration over numeric ranges using the `for` loop. In Kotlin, these types are collectively known as ranges.

The simplest way to construct a range is to use the`..` operation on numeric values. For example:

```
val chars = 'a'..'h'          // all characters from 'a' to 'h'
val twoDigits = 10..99         // all two-digit integers from 10 to
99
val zero2One = 0.0..1.0 // all floating-point numbers in the
range from 0 to 1
```

Using the `in` operation, you can check whether a given value fits into the range. This is basically equivalent to a pair of comparison:

```
val num = readLine()!!.toInt()
println(num in 10..99) // num >= 10 && num <= 99
```

There is also an opposite operation `!in` which allows you to write expressions such as `!(a in b)` in a simplified form:

```
println(num !in 10..99) // !(num in 10..99)
```

In fact, the`..` operation is available for all comparable types, including numeric, `Char`, `Boolean` and `String`. Basically whenever you can use `<=` or `>=`, you can also use`..` to construct a range as shown below:

```
println("def" in "abc".."xyz") // true
println("zzz" in "abc".."xyz") // false
```

Ranges produced by the.. operation are closed. This means they include both start and end points. There is another operation which lets you create semi-closed ranges with an excluded end point. This operation is only available for integer types and basically produces a range with a smaller end point. In the following example upper bound 100 is not included to the resulting range:

```
valtwoDigits = 10 until 100      // same as 10..99, 100 is excluded
```

Note that built-in ranges where end point is strictly less than start one are empty. For example:

```
println(5 in 5..5)           // true
println(5 in 5 until 5) // false
println(5 in 10..1)          // false
```

In general, it's not true if comparison on a given type is ill-behaved. In particular, if it's not transitive, there is a possibility that $x \in a .. b$ might be true even when $a > b$.

There is also a related concept of progression which is an ordered sequence of integer or Char values separated by some fixed step. Every range over these types is an ascending progression with step 1, but progressions in general will give you additional options. For example, you can define descending progression using the downTo operation as shown below:

```
println(5 in 10 downTo 1) // true
println(5 in 1 downTo 10) // false: progression is empty
```

You can also specify a custom progression step:

```
1..10 step 3           // 1, 4, 7, 10
15 down 9 step 2 // 15, 13, 11, 9
```

Progression step must be positive; therefore, if you want to construct a descending sequence, you should use a step together with the downTo operation as in the preceding example.

Progression elements are generated by successively adding a step to its starting point. Therefore, if the end point does not actually correspond to one of progression values. It's automatically adjusted to the nearest progression element:

```
1..12 step 3           // 1, 4, 7, 10: the same as 1..10 step 3
```

```
15 down 8 step 2 // 15, 13, 11, 9: the same as 15 downTo 9  
step 2
```

Using ranges you can extract a portion of a string or array. Mind the difference between the `substring()` function taking a closed integer range and the one taking a pair of indices where the end point is excluded:

```
"Hello, World".substring(1..4)           // ello  
"Hello, World".substring(1 until 4)       // ell  
"Hello, World".substring(1, 4)            // ell: like  
substring(1 until 4)  
IntArray(10) { it*it }.sliceArray(2..5)    // 4, 9,  
16, 25  
IntArray(10) { it*it }.sliceArray(2 until 5) // 4, 9, 16
```

Range and progression types are defined in the Kotlin standard library as a set of classes such as `IntRange`, `FloatRange`, `CharProgression`, `IntProgression`, and so on. You can find an exhaustive list of the classes together with their functions and properties on the documentation page for the `kotlin.ranges` package: kotlinlang.org/api/latest/jvm/stdlib/kotlin.ranges/.

In general, using ranges instead of comparisons involve a slight overhead since ranges are dynamically allocated objects. Compiler, however, tries to avoid creating actual objects when possible. For example, in the following program the `IntRange` instance is not created at runtime; instead it just compares 5 with entered values:

```
fun main() {  
    val a = readLine()!!.toInt()  
    val b = readLine()!!.toInt()  
    println(5 in a..b)  
}
```

So in terms of performance, it's equivalent to a pair of comparisons: `a <= 5 && 5 <= b`. Another major use case of range/progression optimization is a for loop.

IDE Tips: IntelliJ plugin includes a JVM bytecode viewer which can be useful to explore a low-level semantics of the Kotlin code. To open it, choose Tools | Kotlin | Show Kotlin Bytecode in the IDE menu. The viewer updates to reflect the bytecode of the current Kotlin file in the editor and automatically preselect portion of bytecode corresponding to the caret position in the

source code.

If you're not particularly familiar with the JVM bytecode, you can click Decompile button transforming it into a Java code. Note that due to specifics of the bytecode generated by the Kotlin compiler such decompiled code can be formally incorrect, but it still can give you a good understanding of the original Kotlin code's inner workings.

Ranges are not the only types supporting the `in`/`!in` operation: you can use for other types describing some kind of container such as strings or arrays:

```
val numbers = intArrayOf(3, 7, 2, 1)
val text = "Hello!"

println(2 in numbers)      // true
println(9 !in numbers)    // true
println(4 in numbers)      // false
println('a' in text)       // false
println('H' in text)       // true
println('h' !in text)      // true
```

In terms of precedence, the range operation `..` fits between additive and infix ones, while `in`/`!in` are placed between infix and comparisons. In other words, the relevant portion of the table from *Chapter 2, Language Fundamentals*, will now look as follows:

Additive	<code>+ -</code>	<code>a + b..c - d</code> // <code>(a + b)..(c - d)</code>
Range	<code>..</code>	<code>a..b step c</code> // <code>(a..b) step c</code> <code>a in b..c</code> // <code>a in (b..c)</code>
Infix	Named operators	<code>a < b or b < c</code> // <code>(a < (b or b)) < c</code> <code>a == b and b == c</code> // <code>(a == b) and (b == c)</code> <code>a in b or a in c</code> // <code>(a in (b or a)) in c</code>
Named check	<code>in !in</code>	<code>a < b in c</code> // <code>a < (b in c)</code> <code>a !in b > c</code> // <code>(a !in b) > c</code>
Comparison	<code><><= >=</code>	<code>a < b == b < c</code> // <code>(a < b) == (b < c)</code> <code>a < b && b < c</code> // <code>(a < b) && (b < c)</code>

Table 3.1: Operation precedence

Operations `until`, `downTo` and `step` have the same precedence as any other

named infix operation (`and`, `or`, `xor`, to name a few).

When statements and multiple choice

Since `if` statements can choose between two options, one way to implement a multiple choice is combining several `if` statements into a cascade-like structure, which sequentially checks all conditions of interest. Suppose we want to convert a decimal number between 0 and a corresponding hexadecimal digit:

```
fun hexDigit(n: Int): Char {  
    if (n in 0..9) return '0' + n  
    else if (n in 10..15) return 'A' + n - 10  
    else return '?'  
}
```

Kotlin, though, provides you a more concise construct to select among multiple alternatives, which is called a `when` statement. Using this construct, we can rewrite the preceding function into the following form:

```
fun hexDigit(n: Int): Char {  
    when {  
        n in 0..9 -> return '0' + n  
        n in 10..15 -> return 'A' + n - 10  
        else -> return '?'  
    }  
}
```

Basically, the `when` statement is a block which is preceded by the `when` keyword and consists of zero or more branches of the general form `condition -> statement` as well as an optional `else` branch. The statement execution proceeds according to the following rule: the program subsequently evaluates conditions in the order they're written until it finds the one that evaluates to true. If such a condition is found, the program executes `statement`-part of the corresponding branch. If all conditions evaluate to false, the program executes the `else`-branch (if there is one).

IDE Tips: The IntelliJ plugin provides an intention action for automatic conversion between nested `if` and `when`. To access it, press Alt + Enter with a caret placed in the `if/when` keyword and choose “Replace `if` with `when`” or “Replace `when` with `if`” action respectively.

Similar to if, a when statement can be used as expression. In this case, the else branch is mandatory since when should be able to provide some definite value for each possible case:

```
fun hexDigit(n: Int) = when {  
    n in 0..9 -> '0' + n  
    n in 10..15 -> 'A' + n - 10  
    else -> '?'  
}
```

Java versus Kotlin: Kotlin's when is similar to Java's switch statement, which can also select among multiple options. The crucial difference, however, is that when allows you to check arbitrary conditions, while switch can only choose among values of a given expression. Besides, the switch statement in Java follows so-called fall-through semantics: when some condition is matched, the program executes its statements as well as statements in subsequent branches unless execution is explicitly stopped by the break statement. Kotlin's when executes only statements in the matched branch and never falls through the entire when block.

The when statement has another form that is suitable for multiple checks involving equality and in operations. Consider the following function:

```
fun numberDescription(n: Int): String = when {  
    n == 0 -> "Zero"  
    n == 1 || n == 2 || n == 3 -> "Small"  
    n in 4..9 -> "Medium"  
    n in 10..100 -> "Large"  
    n !in Int.MIN_VALUE until 0 -> "Negative"  
    else -> "Huge"  
}
```

Since all conditions of the preceding expression are either equality, the in or !in operation with the same left operand, n, we can express the same logic by making n the subject expression of when and rewriting it in the following form:

```
fun numberDescription(n: Int, maxLarge: Int = 100): String =  
when (n) {  
    0 -> "Zero"  
    1, 2, 3 -> "Small"
```

```

in 4..9 -> "Medium"
in 10..max -> "Large"
!inInt.MIN_VALUE until 0 -> "Negative"
else -> "Huge"
}

```

IDE Tips: The IntelliJ plugin can transform one form of when expression into another eliminating and introducing subject expressions when necessary. You access these actions by placing a caret on the when keyword and pressing Alt + Enter; you can then choose a command depending on the statement form: either “Introduce ... as subject of when”, or “Eliminate argument of when”.

This form of a when statement is distinguished by subject expression, which is written in parentheses after the when keyword. A branch of such statement can start with either in/!in, an arbitrary expression or else keyword (there is also the is/!is branch, which we’ll defer to *Chapter 8, Understanding Class Hierarchies*). The execution is similar to the first form of when and proceeds as follows:

- Subject expression is evaluated: suppose its value is subj.
- The program successively evaluates conditions of branches until it finds the one which is true: the in/!in branch is treated as the in/!in expression with subj as its left operand, while the free-form expression e is interpreted as an equality operation subj == e.
- If such condition is found, the program executes corresponding statement; otherwise, else-statement is executed if it’s present.

The subject form allows you to write multiple conditions in a single branch, separating them by commas (1, 2, 3 -> small branch in the preceding example). During condition evaluations, these commas are effectively treated as logical OR (||).

Note that expressions in the branches of subject when are not necessarily Boolean: they may have an arbitrary type as long as corresponding operations (== or in/!in) are applicable.

Java versus Kotlin: Since Java 12 switch has acquired an expression form which is very similar to the subject form of Kotlin’s when. It, however, has some limitations: in particular, switch doesn’t support range checks (unlike the in/!in operation in Kotlin) and can be applied only to limited set of types: integers, enums, and strings.

Note also that when branches can use arbitrary expressions and are not limited to constants.

Since Kotlin 1.3, when statements allow you to bind subject expression to a variable using the syntax:

```
fun readHexDigit() = when(val n = readLine()!!.toInt()) { //  
    define n  
    n in 0..9 -> '0' + n  
    n in 10..15 -> 'A' + n - 10  
    else -> '?'  
}
```

Such a variable can only be used inside the when block and cannot be declared as var.

Loops

Kotlin supports three control structures that repeat the same sequence of instructions for a given set of data or till some condition is satisfied. The while and do-while loops have the same structure as the corresponding Java statements, and the for loop is very similar to Java's for-each. All loops in Kotlin are statements rather than expressions and so do not have any value as such, only side effects.

The while/do-while loop

Suppose we want to compute a sum of integers entered by a user. Let's agree that zero will serve as a stop-value after which we cease reading the input and report the result:

```
fun main() {  
    var sum = 0  
    var num = 0  
    do {  
        num = readLine()!!.toInt()  
        sum += num  
    } while (num != 0)  
    println("Sum: $sum")  
}
```

The do-while loop is evaluated according to the following rules:

1. Execute the loop body between do and while keywords.
2. Evaluate the condition coming after the while keyword, and if it's true, go back to step 1; otherwise, proceed to the statement after the loop.

Note that the loop body is always executed at least once since condition is checked afterwards.

There is another form of loop which also executes its body while condition holds true, but checks that condition before running instructions in the body. It means that if condition is false upon entering the loop, its body will never execute.

Suppose we want to write the program that generates some number and then asks user to guess it giving hints if the guess was wrong (for example, too small or too big) and stopping when the guess is right:

```
import kotlin.random.*  
fun main() {  
    val num = Random.nextInt(1, 101)  
    var guess = 0  
    while (guess != num) {  
        guess = readLine()!!.toInt()  
        if (guess < num) println("Too small")  
        else if (guess > num) println("Too big")  
    }  
    println("Right: it's $num")  
}
```

The number is generated using the Random.nextInt() function from the standard library.

These examples clearly demonstrate that the while/do-while statements in Kotlin are essentially the same as in Java.

Iterables and for loop

Kotlin for loop allows you to iterate over collection-like values, which can contain or produce multiple elements. We can, for example, use a for loop to sum array elements:

```
fun main() {  
    val a = IntArray(10) { it*it } // 0, 1, 4, 9, 16, ...
```

```
var sum = 0
for (x in a) {
    sum += x
}
println("Sum: $sum") // Sum: 285
}
```

The loop consists of three parts:

1. Iteration variable definition (`x`);
2. A container expression (`a`), which produces values to iterate over;
3. A loop body statement (`{sum += x}`), which is executed on each iteration.

The iteration variable is accessible only inside the loop body and is automatically assigned a new value upon the start of each iteration. Note that a loop variable is not marked with `val` or `var` keywords like you do with an ordinary variable and is implicitly immutable: in other words, you can't change its value inside of the loop body. In the simplest case, the loop variable definition is a simple identifier. You can specify its type, though, but that's rarely needed in practice:

```
for (x: Int in a) {
    sum += x
}
```

Java versus Kotlin: Kotlin for loop is quite similar to Java's for-each loop which will give you a simple syntax for iteration over any `Iterable` instance – be it an array, list, set, or a user-defined type. Kotlin, however, doesn't have a counterpart for an ordinary Java for loop, which requires you to explicitly declare, initialize, check, and update an iteration variable. In Kotlin, such iterations are just special cases of the for loop statement you've seen in the preceding code.

You can use a `for` loop to iterate over string characters. Let's, for example, write our own function which parses a binary string representation of a positive number to `Int`:

```
fun parseIntNumber(s: String, fallback: Int = -1): Int {
    var num = 0
    if (s.length !in 1..31) return fallback
    for (c in s) {
        if (c !in '0'..'1') return fallback
    }
    return num
}
```

```

    num = num*2 + (c - '0')
}
return num
}

```

When a string in question doesn't represent a valid number or doesn't fit into, the function returns some fallback value.

Java versus Kotlin: In Java, the direct iteration on string is not possible, so you have to use some workaround such as iterating over its indices or converting a string to a character array first.

What about ordinary iteration over numeric intervals? For that purpose, we'll use the progressions that we introduced in the previous section. Suppose we want to double all array elements with even indices:

```

val a = IntArray(10) { it*it } // 0, 1, 4, 9, 16, ...
for (i in 0..a.lastIndex) {           // 0, 1, 2, 3, ...
    if (i % 2 == 0) {             // 0, 2, 4, 6, ...
        a[i] *= 2
    }
}

```

We can simplify this loop even further using progression with a custom step:

```

for (i in 0..a.lastIndex step 2) { // 0, 2, 4, 6, ...
    a[i] *= 2
}

```

Strings and arrays have `indices` property which contain a range of character or item indices:

```

val a = IntArray(10) { it*it } // 0, 1, 4, 9, 16, ...
for (i in a.indices step 2) {           // 0, 2, 4, 6, ...
    a[i] *= 2
}

```

The real beauty of the for loop comes from the fact that the compiler doesn't just support some limited set of disparate use cases such as numeric ranges or collections, but provides a unified mechanism that allows you to iterate over all kinds of values. The only thing required of a container expression is the `iterator()` function which returns an `Iterator` object capable of extracting element values. We'll postpone

detailed discussions of iterators till *Chapter 7, Exploring Collections and I/O*, but for now it, suffices to know that many standard Kotlin types already have built-in iterators: that's why for loops work just as well for progressions, arrays, and strings. As we'll see further, using an extension mechanism allows you to attach the iterator to any type you like thus extending a range of possible expressions to iterate.

Java versus Kotlin: The Java `for-each` loop is similar to the Kotlin `for` statement in a sense that it can be applied to any subtype of `Iterable`. The Kotlin `for` loop convention is more flexible though, as it doesn't require container to be of any particular type; all a `for` loop needs is a presence of the `iterator()` function.

Changing loop control-flow: break and continue

Sometimes it's convenient to alter an ordinary control-flow of the loop: for example, it may be convenient to check an exit condition not at the start or end of a loop iteration, but somewhere in the middle. For that purpose, Kotlin includes a pair of expressions:

- `break`, which immediately terminates iterating, forcing the execution to continue from the next statement after the loop;
- `continue`, which stops current iteration and jumps to the condition check.

In other words, these statements have the same semantics as their Java counterparts. Consider, for example, our Guess the Number program. We could have used a `break` statement to write it as follows:

```
import kotlin.random.*  
fun main() {  
    val num = Random.nextInt(1, 101)  
    while (true) {  
        val guess = readLine()!!.toInt()  
        if (guess < num) println("Too small")  
        else if (guess > num) println("Too big")  
        else break  
    }  
    println("Right: it's $num")  
}
```

Note that the loop condition became unnecessary since all exit checks now happen in its body. Thanks to this; we can also move `guess` variable into the loop.

Java versus Kotlin: Like return, break and continue statements in Kotlin can be used as expressions of the type Nothing. We could, for example, have rewritten the preceding program to calculate the message text before printing:

```
import kotlin.random.*  
fun main() {  
    val num = Random.nextInt(1, 101)  
    while (true) {  
        val guess = readLine()!!.toInt()  
        val message =  
            if (guess < num) "Too small"  
            else if (guess > num) "Too big"  
            else break  
        println(message)  
    }  
    println("Right: it's $num")  
}
```

This feature shouldn't be abused though, as in more complex expressions it might, in fact, hinder the understanding of your code.

Suppose that we want to count a number of times each English letter occurs in a given string. In the following example, we will use a continue expression to jump out of the iteration when the character is not a letter before trying to access an array:

```
fun countLetters(text: String): IntArray {  
    val counts = IntArray('z' - 'a' + 1)  
    for (char in text) {  
        val charLower = char.toLowerCase()  
        if (charLower !in 'a'..'z') continue  
        counts[charLower - 'a']++  
    }  
    return counts  
}
```

Java versus Kotlin: In Java, a break is also used to stop the execution of the remaining branches in a switch statement. Since when expressions don't follow a fall-through semantics, the break statements do not serve the same purpose in Kotlin. However, using break or continue inside when is prohibited since such code might be

confusing – especially for programmers migrating to Kotlin from java. Also `continue` is reserved for explicit fall-through semantics that is expected to be implemented in some a future language version. If we've replaced the `if` cascade from our Guess the Number game with a single `when` expression, the compiler would have reported an error:

```
val message = when {
    guess < num -> "Too small"
    guess > num -> "Too big"
    else -> break // Error
}
```

The work around is to use a labeled `break/continue`, which we'll cover in the next section.

Nested loops and labels

When using nested loop statements, the simple `break/continue` expressions we've seen in the previous section are always applied to nearest enclosing loop. In some cases, you might want them to affect control-flow of the outer loop: to do this, Kotlin provides a statement labeling that is similar to Java's albeit with a slightly different syntax.

Suppose we want to write a function that searches a given subarray in an array of integer similar to how `indexOf()` does for strings:

```
fun indexOf(subarray: IntArray, array: IntArray): Int {
    outerLoop@ for (i in array.indices) {
        for (j in subarray.indices) {
            if (subarray[j] != array[i + j]) continue@outerLoop
        }
        return i
    }
    return -1
}
```

Here we attach a label to the outer loop and use `continue@outerLoop` to terminate current iteration of the outer loop that looks for a subarray offset as soon as we see the first mismatch between subarray and array elements. At this point, we know that it

makes no sense to check remaining subarray items and the search must continue starting from a next offset.

In Kotlin, you can attach a label to any statement, but `break` and `continue` specifically require that labels to be attached to loop. If it's not the case, the compiler reports an error. A label name, like that of variable or function, can be an arbitrary identifier.

Java versus Kotlin: Note the syntactic difference between the label definition and usage in Kotlin and Java:

```
loop@ while(true) break@loop // Kotlin  
loop: while(true) break loop // Java
```

Labeling, among others, allows you to use `break/continue` inside when expressions, which are in turn nested into a loop body. Thanks to this, we can write Guess the number program from the previous section as follows:

```
import kotlin.random.*  
fun main() {  
    val num = Random.nextInt(1, 101)  
    loop@ while (true) {  
        val guess = readLine()!!.toInt()  
        val message = when {  
            guess < num -> "Too small"  
            guess > num -> "Too big"  
            else -> break@loop // Correct  
        }  
        println(message)  
    }  
    println("Right: it's $num")  
}
```

Tail-recursive functions

Kotlin supports an optimized compilation for so-called tail-recursive functions. Suppose we want to write a function that implements a binary search in an integer array. Assuming that array is presorted in the ascending order, let's write this search in a recursive form:

```
tailrec fun binIndexOf(
    x: Int,
    array: IntArray,
    from: Int = 0,
    to: Int = array.size
): Int {
    if (from == to) return -1
    val midIndex = (from + to - 1) / 2
    val mid = array[midIndex]
    return when {
        mid < x -> binIndexOf(x, array, midIndex + 1, to)
        mid > x -> binIndexOf(x, array, from, midIndex)
        else -> midIndex
    }
}
```

This definition concisely expresses an algorithm idea, but in general have a performance overhead and risks stack overflow compared to the more cumbersome non-recursive version. In Kotlin, however, you can tell a compiler to automatically translate a tail-recursive function into non-recursive code by adding the `tailrec` modifier. As a result, you get the best of both worlds: concise recursive function with no extra performance penalties. The preceding function, in particular, would be equivalent to the code:

```
tailrec fun binIndexOf(
    x: Int,
    array: IntArray,
    from: Int = 0,
    to: Int = array.size
): Int {
    var fromIndex = from
    var toIndex = to
    while (true) {
        if (fromIndex == toIndex) return -1
        val midIndex = (fromIndex + toIndex - 1) / 2
        val mid = array[midIndex]
        when {
```

```

        mid < x -> fromIndex = midIndex + 1
        mid > x -> toIndex = midIndex
        else -> return midIndex
    }
}
}
}

```

To be eligible for such transformation function must not perform any action after a recursive call: that's the meaning behind tail-recursive. If this requirement is not satisfied but function is still marked as `tailrec`, the compiler will issue a warning and the function will be compiled as a recursive one. For example, the following summation function is not tail-recursive because `sum(array, from + 1, to)` call is followed by addition:

```

tailrec fun sum(array: IntArray, from: Int = 0,      to: Int =
array.size): Int {
    // Warning: not a tail-recursive call
    return if (from < to) array[from] + sum(array, from
+ 1, to) else 0
}

```

The compiler will also report a warning if a function is marked as `tailrec` but contains no recursive calls:

```

tailrec fun sum(a: Int, b: Int): Int {
    return a + b // Warning: no tail-recursive calls
}

```

Exception handling

Exception handling in Kotlin is very similar to Java's approach. A function may terminate either normally, which means that it returns some value – possibly a trivial one of type `Unit` – or abnormally by throwing an exception object when some error occurs. In the latter case, the exception can be either caught and handled by its caller or propagated further up the call stack. Let's now consider exception-related control structures.

Throwing an exception

To signal an error condition, you use a throw expression with an exception object just like in Java. Let's revise our earlier `parseIntNumber()` function to throw an exception when its input is ill-formed rather than return some fallback value:

```
fun parseIntNumber(s: String): Int {  
    var num = 0  
    if (s.length !in 1..31) throw NumberFormatException("Not  
a number: $s")  
    for (c in s) {  
        if (c !in '0'..'1') throw NumberFormatException("Not  
a number: $s")  
        num = num*2 + (c - '0')  
    }  
    return num  
}
```

Java versus Kotlin: Unlike Java, creating a class instance (in this case, it's an exception) doesn't require any special keywords like Java's `new`. In Kotlin, a constructor invocation `NumberFormatException("Not a number: $s")` looks like an ordinary function call.

When an exception is thrown the following actions are taken:

- The program looks for an exception handler, which can catch a given exception. If such a handler is found, it gains a control.
- If no handler is found in the current function, its execution is terminated, the function is popped out the stack, and the whole search is repeated in the context of its caller (if any). We say that exception is propagated to the caller.
- If exception propagates uncaught to the entry point, the current thread terminates.

You can see that exception handling steps in Kotlin are basically the same as in Java.

Java versus Kotlin: In Kotlin, `throw` is an expression of type `Nothing` like `break` and `continue` that we've seen in one of the earlier sections. For example:

```
fun sayHello(name: String) {  
    val message =  
        if (name.isNotEmpty()) "Hello, $name"  
        else throw IllegalArgumentException("Empty name")
```

```
    println(message)
}
```

Handling errors with try statements

To handle an exception in Kotlin, you use a `try` statement that has essentially the same syntax as in Java. Consider the following function that return some default value when it can't parse an input string to number:

```
import java.lang.NumberFormatException
fun readInt(default: Int): Int {
    try {
        return readLine()!!.toInt()
    } catch (e: NumberFormatException) {
        return default
    }
}
```

The code which may throw an exception (in our case it's the `toInt()` call) is wrapped in the `try` block. A first form of `try` statement also includes at least one `catch` block that handles an exception of the appropriate type (for example, `NumberFormatException`). The exception to handle is represented by `exception` parameter, which you can use anywhere inside the `catch` block. When the code inside `try` block throws some exception, its execution terminates and the program chooses first `catch` block, which is able to handle it; if no such block is found, the exception propagates.

Java verus Kotlin: In Java 7 or later, a single `catch` block can handle multiple exceptions using syntax of the sort: `catch (FooException | BarException e)` `{ }`. In Kotlin, such handlers are not supported yet.

Since `catch` blocks are checked in order of their declaration, placing a block that can handle some exception type before a block that can handle one of its supertype is useless since any exception of that subtype will be caught by the preceding block. For example, since `NumberFormatException` is a subtype of `Exception`, the second `catch` block in the following function is effectively dead:

```
import java.lang.NumberFormatException
fun readInt(default: Int): Int {
    try {
```

```
        return readLine()!!.toInt()
    } catch (e: Exception) {
        return 0
    } catch (e: NumberFormatException) {
        return default // dead code
    }
}
```

Java versus Kotlin: Note that in Java, a similar statement will produce a compile-time error since Java explicitly forbids such kind of unreachable code.

The major difference between `try` statements in Java and Kotlin is that Kotlin's `try` can be used as an expression. The value of such an expression is either the value of a `try` block (if no exceptions are thrown), or the value of a `catch` block which manages to handle an exception:

```
import java.lang.NumberFormatException
fun readInt(default: Int) = try {
    readLine()!!.toInt()
} catch (e: NumberFormatException) {
    default
}
```

Java versus Kotlin: Unlike Java, Kotlin doesn't distinguish checked and unchecked exceptions. The rationale is that in large projects requiring explicit specification of possible exceptions in fact decrease productivity and produce an excessive boilerplate code.

Another form of the `try` statement uses the `finally` block, which allows you to perform some actions just before the program leaves the `try` block:

```
import java.lang.NumberFormatException
fun readInt(default: Int) = try {
    readLine()!!.toInt()
} finally {
    println("Error")
}
```

This block is useful to clean up some resources that might have been allocated before/in the `try` block, for example, to close a file or network connection. You may

also use catch and finally blocks within a single `try` statement.

Note that the value of the finally block doesn't affect the value of the entire try statement when it's used as an expression.

Java versus Kotlin: You're probably familiar with a try-with-resources statement which is introduced in Java 7 and allows you to perform automatic cleanup of resources such as file streams and network connections. Although Kotlin doesn't have a special construct for this purpose, it does provide a library function `use` which solves the same task. We'll look at it more closely in *Chapter 7, Exploring Collections and I/O*.

Conclusion

Let's summarize what we've done in this chapter. We've acquired the knowledge of fundamental control structures that constitute the algorithmic basis of imperative programming. We've learned how to define and use functions facilitating the reuse of common pieces of program code. Finally, we've discussed structuring your program by grouping related functions into packages. Now, you have all the necessary knowledge for exploiting imperative and procedural paradigms within the Kotlin language.

In the next chapter, we will move toward an object-oriented programming. We'll look at defining classes and objects, get an understanding of class initialization, learn to declare and use properties, and address the issue of handling null values in Kotlin.

Questions

1. What's an expression-body function? In what cases is it worth using it instead of a block one?
2. How would you decide whether to use default parameters values or function overloading?
3. What are the pros and cons of using named arguments?
4. How to declare function with variable number of arguments? What're the differences between varargs in Kotlin and Java?
5. What the types `Unit` and `Nothing` are used for? Compare them with the Java's `void`. Can you use a `Nothing`-or `Unit`-typed variables?
6. Try to explain the meaning of code like `return return 0`. Why is it considered valid but redundant?
7. Can you have a function without `return` statements?

8. What's the local function? How would you emulate such functions in Java?
9. What's the difference between public and private top-level functions?
10. How can you group your code using packages? Point out the key differences from the packages in Java.
11. What's an import alias? Can you use something similar to Java's static imports in Kotlin?
12. How does an `if` statement/expression work? Compare it with Java's `if` statement and conditional operation (`? :`).
13. Describe the basic algorithm of `when` statement. How is it different from Java's `switch`?
14. How would you implement a Java's counting for loop like `for (int i = 0; i < 100; i++)`?
15. What are the loop statements in Kotlin? What's the difference between `while` and `do ... while`? Why is `for` loop used?
16. How to change loop's control flow using `break` and `continue` statements?
17. Give an overview of an exception handling process. What are the major differences from Java? Compare `try` statements in Java and Kotlin.

CHAPTER 4

Working with Classes and Objects

In this chapter, we will get a taste of object-oriented programming in Kotlin and see how to define our own types using classes. We'll address major topics such as initialization of class instances, using visibility for hiding implementation details, implementing singletons with object declarations, and utilizing different kinds of properties for various effects beyond a simple storage of data: lazy computations, deferred initialization, custom read/write behavior, and so on. A related topic you'll also get to know in this chapter is a type nullability which allows the Kotlin compiler to distinguish between values that are null and those that are not.

Structure

- Class definition and members
- Constructors
- Member visibility
- Nested and local classes
- Nullable types
- Using non-trivial properties
- Objects and companions

Objective

The objectives of this chapter is to:

- Introduce the reader to the basics of object-oriented programming in Kotlin using classes and objects
- Learn to handle nullable values
- Get an understanding of how to use different varieties of properties.

Defining a class

A class declaration introduces a new type with custom set of operations. The readers familiar with Java or some other object-oriented programming language such as C++ will surely find class declaration familiar. In this section, we will discuss a basic class structure, initialization of newly allocated instances, the issue of visibility, and special kinds of classes declared inside other classes or function bodies.

By default, a class declaration defines a referential type. In other words, the values of such types are references pointing to the actual data of a particular class instance. Similarly, Java instances themselves are created explicitly with a special constructor call and freed automatically by a garbage collector after program loses all references to them. Kotlin 1.3 had introduced a concept of inline class, which allows defining non-referential types as well. We'll address this topic in *Chapter 6, Using Special-Case Classes*.

A class anatomy

Similarly to Java, a Kotlin class is defined using a `class` keyword with a name followed by a class body, which is a block containing definitions of members. Let's define a class that holds some information about a person:

```
class Person {  
    var firstName: String = ""  
    var familyName: String = ""  
    var age: Int = 0  
    fun fullName() = "$firstName $familyName"  
    fun showMe() {  
        println("${fullName()}: $age")  
    }  
}
```

This definition tells us that every instance of the `Person` class will have properties `firstName`, `familyName`, `age`, and two functions `fullName()` and `showMe()`. The simplest property variety is basically a variable associated with a particular class instance: you can compare it with a class field in Java. In more general cases, a property may involve arbitrary computations: their values may be generated on the fly rather than stored in a class instance, or computed lazily, taken from a map, and so on. The common feature of all properties is a reference syntax that allows us to use them like a variable:

```
fun showAge(p: Person) = println(p.age) // reading from a
```

```
property
fun readAge(p: Person) {
    p.age = readLine()!!.toInt() // assignment to a property
}
```

Note that since `property` is associated with a particular class instance, we have to qualify it with an expression (like `p` in the preceding code): it's called a receiver and signifies an instance which you use to access a property. The same also goes for member functions that are often called methods:

```
fun showFullName(p: Person) = println(p.fullname()) // calling a method
```

The receiver can be thought of as an additional variable available for all class members. Inside a class, you can refer to it using this expression. In most cases, it's assumed by default, so you don't have to write it explicitly to access members of the same class. For example, our first example could have been written as:

```
class Person {
    var firstName: String = ""
    var familyName: String = ""
    var age: Int = 0
    fun fullName() = "${this.firstName} ${this.familyName}"
    fun showMe() {
        println("${this.fullName()}: ${this.age}")
    }
}
```

Sometimes, though, this is necessary; for example, you can use it to distinguish between a class property and a methods parameter with the same name:

```
class Person {
    var firstName: String = ""
    var familyName: String = ""
    fun setName(firstName: String, familyName: String) {
        this.firstName = firstName
        this.familyName = familyName
    }
}
```

Java versus Kotlin: Unlike Java's fields, Kotlin properties doesn't violate encapsulation since you're free to change their implementation – for example, add a custom getter or setter – without the need to change the client code. In other words, the `firstName` reference remains valid regardless of how the property is implemented. In the following section, we'll see how such custom properties are defined.

Note that the underlying field used by the property is always encapsulated and can't be accessed outside the class definition – and in fact, outside the property definition itself.

A class instance must be explicitly created before you can access its method. This is accomplished by a constructor call which has a form of an ordinary function call: the difference is that you use a class name instead of a function's:

```
fun main() {  
    val person = Person() // Create a Person instance  
    person.firstName = "John"  
    person.familyName = "Doe"  
    person.age = 25  
    person.showMe() // John Doe: 25  
}
```

When you use a constructor call, the program first allocates a heap memory for a new instance and then executes a constructor code which initializes an instance state. In the preceding example, we're relying on a default constructor, which doesn't take any parameters: hence no arguments in a constructor call. In the next section, we'll see how to define custom constructors that allow running your own initialization code.

Kotlin classes are public by default, which means, they may be used in any part of your code. Similarly to top-level functions, you may also mark the top-level classes as private or internal, limiting their visibility scope to the containing file or compilation module, respectively.

Java versus Kotlin: In Java, on the contrary, class visibility is by default limited to the containing package. You have to mark its definition with explicit `public` modifier to make it visible everywhere.

Note also that in Kotlin, you don't need to name your source file exactly as a public class it contains. You may also define multiple public classes in a single file. If a file contains exactly one class, though, the file and the class usually do have the same name, but in Kotlin, it's more like a matter of code style than a strict requirement (as opposed to Java).

Class properties may be immutable just like local variables. In such cases, however, we need a way to provide some actual values for them during initialization, lest all instances get stuck with the same values, like in the following code:

```
class Person {  
    // all instance will have the same value of firstName  
    val firstName = "John"  
}
```

This may be accomplished by using a custom constructor which brings us to the next topic.

Constructors

A constructor is a special function that initializes a class instance and is invoked upon the creation of instance. Consider the following class:

```
class Person(firstName: String, familyName: String) {  
    val fullName = "$firstName $familyName"  
}
```

Note the parameter list we've added after the `class` keyword: these parameters are passed to class when the program creates its instance and can be used to initialize properties and perform some other work:

```
fun main() {  
    val person = Person("John", "Doe") // Create new Person  
    instance  
    println(person.fullName)           // John Doe  
}
```

Java versus Kotlin: Note that Kotlin doesn't use a special keyword (like Java's `new`) to denote a constructor call.

The parameter list in the class header is called a primary constructor declaration. A primary constructor doesn't have a single body like a function: instead, its body consists of property initializers as well as initialization blocks taken in the order they appear in the class body. The initialization block is a block statement prefixed with the `init` keyword: such blocks can be used for a non-trivial initialization logic, which you need on class instantiation. For example, the following class would print a message

each time its primary constructor is called:

```
class Person(firstName: String, familyName: String) {  
    val fullName = "$firstName $familyName"  
    init {  
        println("Created new Person instance: $fullName")  
    }  
}
```

A class may contain multiple `init` blocks: in that case, they are executed sequentially together with property initializers.

Note that initialization block may not contain `return` statements:

```
class Person(firstName: String, familyName: String) {  
    val fullName = "$firstName $familyName"  
    init {  
        if (firstName.isEmpty() && familyName.isEmpty())  
            return // Error  
        println("Created new Person instance: $fullName")  
    }  
}
```

So far, we've always specified a property initial value in its initializer. In some cases, though, you may need more complex initialization logic which can't be fit into a single expression. For this reason, Kotlin permits initialization of properties inside the `init` blocks:

```
class Person(fullName: String) {  
    val firstName: String  
    val familyName: String  
    init {  
        val names = fullName.split(" ")  
        if (names.size != 2) {  
            throw IllegalArgumentException("Invalid name:  
$fullName")  
        }  
        firstName = names[0]  
        familyName = names[1]  
    }  
}
```

```

    }
}

fun main() {
    val person = Person("John Doe")
    println(person.firstName) // John
}

```

In the preceding example, the `init` block splits `fullName` into an array of space-separated substrings and then uses them to initialize `firstName` and `familyName` properties.

The compiler ensures that every property is definitely initialized: if it can't guarantee that every execution path in the primary constructor either entails initialization of all member properties, or throws an exception, you will get a compilation error as follows:

```

class Person(fullName: String) {
    // Error: properties may be uninitialized
    val firstName: String
    val familyName: String
    init {
        val names = fullName.split(" ")
        if (names.size == 2) {
            firstName = names[0]
            familyName = names[1]
        }
    }
}

```

Primary constructor parameters may not be used outside property initializers and `init` blocks. For example, the following code is wrong since `firstName` is not available inside the member function:

```

class Person(firstName: String, familyName: String) {
    val fullName = "$firstName $familyName"
    fun printFirstName() {
        println(firstName) // Error: first name is not available
here
    }
}

```

```
}
```

A possible solution would be to add member properties holding values of constructor parameters:

```
class Person(firstName: String, familyName: String) {  
    val firstName = firstName // firstName refers to  
    constructor parameter  
    val fullName = "$firstName $familyName"  
    fun printFirstName() {  
        println(firstName) // Ok: firstName refers to member  
        property here  
    }  
}
```

Kotlin, however, provides out-of-the-box solution, which allows you to combine property and constructor parameter in a single definition:

```
class Person(val firstName: String, familyName: String) {  
    // firstName refers to parameter  
    val fullName = "$firstName $familyName"  
    fun printFirstName() {  
        println(firstName) // firstName refers to member  
        property  
    }  
}  
fun main() {  
    val person = Person("John", "Doe")  
    println(person.firstName) // firstName refers to property  
}
```

Basically, when you mark primary constructor parameter with the `val` or `var` keyword, you also define a property which is automatically initialized with a parameter value. When you refer to such definition in a property initializer or init block, it means a constructor parameter; in any other context, it's a property.

IDE Tips: The IntelliJ plugin can detect the code when you initialize the member property by value of constructor parameter and convert it to the `val/var` parameter (*Figure 4.1*):

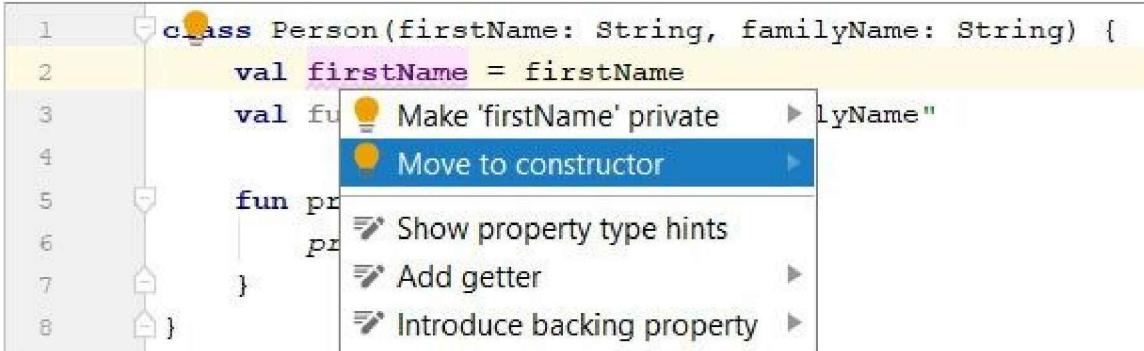


Figure 4.1: Converting property to the `val/var` parameter

Note that using `val/var` parameters you can define a class which has non-trivial members and an empty body:

```
class Person(val firstName: String, val familyName: String =  
"") {  
}
```

In such cases, Kotlin allows you to omit the body entirely. In fact, that's a recommended code style enforced by the IntelliJ plugin:

```
class Person(val firstName: String, val familyName: String =  
"")
```

Similarly to functions, you can use default values and `vararg` for constructor parameters:

```
class Person(val firstName: String, val familyName: String =  
"") {  
    fun fullName() = "$firstName $familyName"  
}  
class Room(vararg val persons: Person) {  
    fun showNames() {  
        for (person in persons) println(person.fullName())  
    }  
}  
fun main() {  
    val room = Room(Person("John"), Person("Jane", "Smith"))  
    room.showNames()
```

```
}
```

In some cases, you need to provide multiple constructors which initialize a class instance in different ways. Many of them are covered by a single primary constructor with default parameters, but sometimes that's not enough. In Kotlin, this problem can be solved by secondary constructors. A secondary constructor syntax is similar to that of function definition with the `constructor` keyword in place of a function name:

```
class Person {  
    val firstName: String  
    val familyName: String  
    constructor(firstName: String, familyName: String) {  
        this.firstName = firstName  
        this.familyName = familyName  
    }  
    constructor(fullName: String) {  
        val names = fullName.split(" ")  
        if (names.size != 2) {  
            throwIllegalArgumentException("Invalid name:  
$fullName")  
        }  
        firstName = names[0]  
        familyName = names[1]  
    }  
}
```

Although a secondary constructor can't be given a return type, it has a form of effectively `Unit`-typed function. In particular, you can use return statements inside its body (as opposed to the `init` blocks).

If class doesn't have a primary constructor, then every secondary constructor invokes property initializers and `init` blocks before executing its own body: this ensures that common initialization code runs exactly once upon class instantiation regardless of which secondary constructor is called.

An alternative option is to make a secondary constructor call another secondary constructor using a constructor delegation call:

```
class Person {  
    val fullName: String
```

```
constructor(firstName: String, familyName: String) :  
    this("$firstName $familyName")  
constructor(fullName: String) {  
    this.fullName = fullName  
}  
}
```

A constructor delegation call is written after colon (:) separating it from the constructor parameter list and looks like an ordinary call with the `this` keyword in place of the function name.

When a class has a primary constructor, all secondary constructors (if any) must delegate either to it, or to some other secondary constructor. We can, for example, turn one secondary constructor from our example to the primary one:

```
class Person(valfullName: String) {  
    constructor(firstName: String, familyName: String) :  
        this("$firstName $familyName")  
}
```

Note that secondary constructors may not declare property-parameters using `val`/`var` keywords:

```
class Person {  
    constructor(valfullName: String) // Error  
}
```

There is also a separate issue of using secondary constructors in combination with class inheritance to call superclass constructors. We'll deal with it in *Chapter 8, Understanding Class Hierarchies*.

Member visibility

Class members may have different visibility which determines their usage scope. This is a major part of class definition since visibilities allow you to enforce encapsulation of implementation-specific details effectively hiding them from the outside code. In Kotlin, a class member visibility is represented by one of the following modifier keywords:

- `public`: A member may be used anywhere; this is assumed by default. So

usually, there is no need to use the `public` keyword explicitly.

- `internal`: A member is accessible only within the compilation module containing its class.
- `protected`: A member is accessible within the containing class and all of its subclasses; we'll postpone the detailed discussion of this case till *Chapter 8, Understanding Class Hierarchies*, dealing with class inheritance;
- `private`: A member is accessible only within the containing class body.

The meaning of these modifiers is in fact quite similar to the ones we've seen for top-level functions and properties.

Java versus Kotlin: In Java, the default visibility is package-private meaning that a member is accessible anywhere within the containing package. If you want a member to be public, you have to explicitly mark it with the `public` modifier. In Kotlin, on the opposite, class members (and, in fact, all non-local declarations) are public by default. Note also that currently Kotlin doesn't have a direct counterpart for Java's package-private visibility.

In the following code, properties `firstName` and `familyName` are declared as `private` and thus inaccessible to the `main()` function. The `fullName()` function, on the other hand, is `public`:

```
class Person(private val firstName: String,
private val familyName: String) {
    fun fullName() = "$firstName $familyName"
}
fun main() {
    val person = Person("John", "Doe")
    println(person.firstName)      // Error: firstName is not
accessible here
    println(person.fullName()) // Ok
}
```

The visibility modifiers are supported for functions and properties – both declared in the class body and as primary constructor parameters – as well as primary and secondary constructors. If you want to specify visibility for a primary constructor, you have to also add an explicit `constructor` keyword:

```
class Empty private constructor() {
```

```

    fun showMe() = println("Empty")
}
fun main() {
    Empty().showMe() // Error: can't invoke private
constructor
}

```

Note that the `Empty` class can't be instantiated since its only constructor is private and so is not available outside of the class body. In the *Objects* section, we'll see how a constructor hiding can be used together with the so-called companion objects to create factory methods.

Nested classes

Apart from functions, properties, and constructors, the Kotlin classes may include other classes as its members. Such classes are called nested. Let's consider an example:

```

class Person (val id: Id, val age: Int) {
    class Id(val firstName: String, val familyName:
String)
    fun showMe() = println("${id.firstName} ${id.familyName},
$age")
}
fun main() {
    val id = Person.Id("John", "Doe")
    val person = Person(id, 25)
    person.showMe()
}

```

Note that outside of the containing class body, references to nested classes must be prefixed with outer class name, like `Person.Id` in the preceding code.

Like other members, nested classes may have different visibilities. Being members of their containing class, they also may access its private declarations:

```

class Person (private val id: Id, private val age: Int) {
    class Id(private val firstName: String,
            private val familyName: String) {
        fun nameSake(person: Person) = person.id.firstName ==
    }
}

```

```

    firstName
}
fun showMe() = println("${id.firstName} ${id.familyName},
$id.age")
}

```

Java versus Kotlin: Unlike Java, the outer class may not access private members of its nested classes.

The nested class may be marked as `inner` to be able to access current instance of its outer class:

```

class Person(val firstName: String, val familyName: String) {
    inner class Possession(val description: String) {
        fun showOwner() = println(fullName())
    }
    fun fullName() = "$firstName $familyName"
}
fun main() {
    val person = Person("John", "Doe")
    // Possession constructor call
    val wallet = person.Possession("Wallet")
    wallet.showOwner() // John Doe
}

```

Note how the call of the inner class constructor is qualified with an outer class instance: `person.Possession("Wallet")`. Similarly, to other member references, qualification may be omitted if the instance in question is this:

```

class Person(val firstName: String, val familyName: String) {
    inner class Possession(val description: String) {
        fun showOwner() = println(fullName())
    }
    // the same as this.Possession("Wallet")
    val myWallet = Posession("Wallet")
}

```

In general, this always means the innermost class instance, so inside an inner class body it refers to the current instance of an inner class itself. When you need to

reference the outer instance from an inner class body, you may use a qualified form of this expression:

```
class Person(val firstName: String, val familyName: String) {  
    inner class Possession(val description: String) {  
        fun getOwner() = this@Person  
    }  
}
```

The identifier coming after the @ symbol is a name of the outer class.

Java versus Kotlin: Nested classes in Kotlin and Java are very similar. The major difference is a default behavior in the absence of additional modifiers: while Java classes are inner by default and must be explicitly marked as static if you do not want their objects to be associated with instances of outer class, Kotlin classes are not. In other words, the following Kotlin code:

```
class Outer {  
    inner class Inner  
    class Nested  
}
```

is basically equivalent to the Java declaration:

```
public class Outer {  
    public class Inner {  
    }  
    public static class Nested {  
    }  
}
```

Local classes

Similar to Java, Kotlin classes can be declared inside of the function body. Such local classes can only be used inside the enclosing code block:

```
fun main() {  
    class Point(val x: Int, val y: Int) {  
        fun shift(dx: Int, dy: Int): Point = Point(x + dx, y  
+ dy)
```

```

        override fun toString() = "($x, $y)"
    }
    val p = Point(10, 10)
    println(p.shift(-1, 3)) // (9, 13)
}
fun foo() {
    println(Point(0, 0)) // Error: can't resolve Point
}

```

Similarly to local function, Kotlin local classes can access declarations from the enclosing code. In particular, they capture local variables which can be accessed and even modified inside of the local class body.

```

fun main() {
    var x = 1
    class Counter {
        fun increment() {
            x++
        }
    }
    Counter().increment()
    println(x) // 2
}

```

Java versus Kotlin: Unlike Kotlin, Java doesn't allow modification of captured variables. Moreover, all such variables must be explicitly marked as final when used inside the anonymous class. Note, however, that the ability to change captured variables in Kotlin comes with a certain price. In order to share variables between anonymous object and its enclosing code, the Kotlin compiler encloses their values inside of special wrapper objects. The Java's equivalent of the Counter example above would look like this:

```

import kotlin.jvm.internal.Ref.IntRef;
class MainKt {
    public static void main(String[] args) {
        final IntRef x = new IntRef(); // create wrapper
        x.element = 1;
        final class Counter {

```

```

        public final void increment() {
x.element++;                      // modify shared data
    }
}
(new Counter()).increment();
System.out.println(x.element); // read shared data
}
}

```

Note that immutable variables have no such overhead since they don't require any wrapper.

Unlike nested classes, local classes can't have visibility modifiers: their scope is always limited by the enclosing block.

Local classes may contain all the members permitted in any other classes such as functions, properties, constructors, or nested classes. Note, however, that their nested classes must always be marked as inner:

```

fun main(args: Array<String>) {
    class Foo {
        val length = args.length
        inner class Bar {
            val firstArg = args.firstOrNull()
        }
    }
}

```

Allowing non-inner classes would lead to somewhat counterintuitive behavior where outer class can access local state (such as the `args` variable shown above) while its nested class, being non-inner, cannot.

Nullability

Similarly to Java, referential values in Kotlin include special constant `null` which represents a null reference, that is, a reference which doesn't correspond to any allocated object. Null doesn't behave like any other reference: in Java, you can assign the null to a variable of any referential type, but can't use any methods or properties defined for the corresponding type as any attempt to access null members results in `NullPointerException` (NPE for short). The worst part is that such errors only

reveal themselves at runtime as compiler can't detect them using static type information.

A significant advantage of the Kotlin type system is its ability to make clear distinction between referential types which allow null values and those which do not. This feature shifts the problem to compilation time and helps you to mostly avoid the notorious `NullPointerException`.

In this section, we'll discuss types which are used to represent nullable values and basic operations you can use for dealing with nulls. In *Chapter 12, Java Interoperability*, we'll also address nullability issues related to Java-Kotlin interoperability.

Nullable types

One of the major features of Kotlin type system is its ability to distinguish between types that include null values and those which do not. In Java, all reference types are assumed to be nullable: in other words, the compiler can't guarantee that a particular variable of reference type can't hold null.

In Kotlin, however, all references types are non-nullable by themselves. So, you can't store a null in a variable of, say, the `String` type. Consider the following function that checks if a given string contains only letter characters:

```
fun isLetterString(s: String): Boolean {
    if (s.isEmpty()) return false
    for (ch in s) {
        if (!ch.isLetter()) return false
    }
    return true
}
```

If we try to pass null for the `s` parameter, we'll get a compilation error:

```
fun main() {
    println(isLetterString("abc")) // Ok
    println(isLetterString(null)) // Error
}
```

The reason is that argument in the second call has a nullable type, but `String` doesn't accept nulls, so the call is forbidden. You don't need to write any additional checks in the `isLetterString()` itself to ensure that no null is passed or worry that it may

throw NPE on trying to dereference its parameter: the Kotlin compiler prevents such errors at compilation time.

Java versus Kotlin: In Java, on the opposite, passing null into the following function is completely acceptable from the compiler's point of view, but produces NullPointerException at runtime:

```
class Test {  
    static boolean isLetterString(String s) {  
        for (int i = 0; i < s.length; i++) {  
            if (!Character.isLetter(s.charAt(i))) return  
false;  
        }  
        return true;  
    }  
    public static void main(String[] args) {  
        // Compiles but throw an exception at runtime  
        System.out.println(isEmpty(null))  
    }  
}
```

What if you need to write a function that may accept the null value? In this case, you mark the parameter type as nullable by placing the question mark after it:

```
fun isBooleanString(s: String?) = s == "false" || s == "true"
```

Types like String? are called nullable types in Kotlin. In terms of the type system, every nullable type is a supertype of its base type that enlarges its original set of values by including null. This, in particular, means that nullable variable can be always assigned a value of the corresponding non-nullable type, but the opposite is, of course, false:

```
fun main() {  
    println(isBooleanString(null)) // Correct  
    val s: String? = "abc" // Correct  
    val ss: String = s // Error  
}
```

Note that the last assignment in the preceding example is incorrect: even though the variable s doesn't hold the null value at runtime, the compiler has to be conservative

since it can only use a static type information which tells it that the variable `s` is nullable because we've explicitly marked it as such.

At runtime, non-nullable values do not actually differ from the nullable ones: the distinction exists on the compilation level only. The Kotlin compiler doesn't use any wrappers (such as optional class introduced in Java 8) to represent non-nullable values, so no additional runtime overheads are involved.

Primitive types such as `Int` or `Boolean` also have nullable versions. Bear in mind, though, that such types always represent boxed values:

```
fun main() {
    val n: Int = 1      // primitive value
    val x: Int? = 1 // reference to a boxed value
}
```

The smallest nullable type is `Nothing?` which doesn't contain any other value aside of the null constant: this is the type of the null itself and a subtype of any other nullable type. The largest nullable type `Any?` is also the largest type in the whole Kotlin type system and is considered a supertype of any other type, nullable or not.

Nullable types don't retain methods and properties available for their base types. The reason is that usual operations such as calling a member function or reading a property don't make sense for the null value. If we change out the `isLetterString()` function by replacing its parameter type with `String?` but leave everything else untouched, we'll get a compilation error as now all usages of `s` in the function body become incorrect:

```
fun isLetterString(s: String?): Boolean {
    // Error: isEmpty() is not available on String?
    if (s.isEmpty()) return false
    // Error: iterator() is not available on String?
    for (ch in s) {
        if (!ch.isLetter()) return false
    }
    return true
}
```

Note that you can't use the `for` loop to iterate over nullable strings since `String?` doesn't have the `iterator()` method.

In fact, nullable types may have their own methods and properties, thanks to the Kotlin extension mechanism. In *Chapter 5, Leveraging Advanced Functions and Functional Programming*, we'll address this issue in more details. One example is a string concatenation, which also works for values of the `String?` type:

```
fun exclaim(s: String?) {
    println(s + "!")
}

fun main() {
    exclaim(null) // null!
}
```

So how do we fix the code like the `isLetterString()` function to correctly process nullable values? To do the job, Kotlin suggests several options, which we'll cover in the following sections.

Nullability and smart casts

The most straightforward way to process a nullable value is to compare it with `null` using some kind of conditional statement:

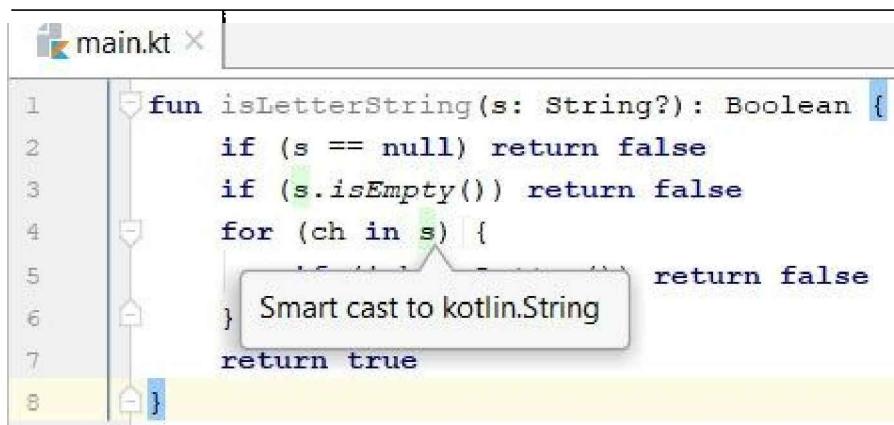
```
fun isLetterString(s: String?): Boolean {
    if (s == null) return false
    // s is non-nullable here
    if (s.isEmpty()) return false
    for (ch in s) {
        if (!ch.isLetter()) return false
    }
    return true
}
```

Although we haven't changed the type of `s` itself, adding the check against `null` somehow makes the code compilable. This is possible, thanks to a helpful Kotlin feature which is called a smart cast. Basically, whenever you make an equality check against `null`, the compiler knows that in one control-flow branch, the value of interest is exactly `null`, while in another, it's definitely not `null`. It then uses this information to refine the value type, implicitly casting it from nullable to non-nullable, hence the name smart cast. In the preceding example, the compiler understands that since the branch corresponding to `s == null` being `true` ends with a `return statement`, the

code coming after the `if (s == null) return false` never executes when `s` is null. As a result, the variable `s` is assumed to have non-nullable type `String` in the remaining piece of the function body.

Smart casts are not limited to nullability: in *Chapter 8, Understanding Class Hierarchies*, we'll see how they enable safe type casting in the context of class hierarchies.

IDE Tips: The IntelliJ plugin has a special highlighting for variable references affected by smart casts: thanks to it, you can easily distinguish such variables just by looking at your code. It also shows the refined type in the reference tooltip (*Figure 4.2*):



The screenshot shows the IntelliJ IDEA code editor with a file named `main.kt`. The code defines a function `isLetterString` that takes a nullable string `s` and returns a boolean. Inside the function, there is a loop that iterates over the characters in `s`. A tooltip appears over the expression `ch in s`, indicating a "Smart cast to kotlin.String". The code is as follows:

```
fun isLetterString(s: String?): Boolean {
    if (s == null) return false
    if (s.isEmpty()) return false
    for (ch in s) {
        if (ch !in 'A'..'Z' && ch !in 'a'..'z') return false
    }
    return true
}
```

Figure 4.2: Smart cast highlighting

Smart casts also work inside of other statements or expressions concerned with condition checking such as when expressions and loops:

```
fun describeNumber(n: Int?) = when (n) {
    null -> "null"
    // n is non-nullable in the following branches
    in 0..10 -> "small"
    in 11..100 -> "large"
    else -> "out of range"
}
```

The same goes for right-hand sides of `||` and `&&` operations:

```
fun isSingleChar(s: String?) = s != null && s.length == 1
```

Note that in order to perform a smart cast, compiler has to ensure that the variable in

question doesn't change its value between the check and the usage. In particular, immutable local variables we've seen so far permit smart casts without limitations since they can't change the value after initialization. Mutable variables, however, may prevent smart casts when modified between the null check and the usage:

```
var s = readLine() // String?  
if (s != null) {  
    s = readLine()  
    // No smart cast below as variable has changed its value  
    println(s.length) // Error  
}
```

Mutable properties never permit smart casts since, in general, they may be changed by other code at any time. In *Chapter 8, Understanding Class Hierarchies*, we'll discuss these rules and their exceptions in more details.

Not-null assertion operator

We've already come across the !! operator in our earlier examples involving the `readLine()` function. The !! operator, also called a not-null assertion, is a postfix operator which throws a `KotlinNullPointerException` (on JVM, it's a subclass of the well-known `NullPointerException`) when its argument is null and returns it unchanged when it's not. The resulting type is a non-nullable version of the original type. Basically, it reproduces the behavior of the Java program, which throws an exception on attempt to dereference the null value. The following example demonstrates such behavior:

```
val n = readLine()!!.toInt()
```

In general, this operation should be avoided because null values usually require some reasonable response instead of simply throwing an exception. Sometimes, though, its usage is justified. Consider, for example, the following program:

```
fun main() {  
    var name: String? = null  
    fun initialize() {  
        name = "John"  
    }  
    fun sayHello() {
```

```
    println(name !! .toUpperCase())
}
initialize()
sayHello()
}
```

In this case, the not-null assertion is an appropriate solution since we know that the `sayHello()` function called after the name is assigned a non-null value. The compiler, however, can't recognize that such usage is safe and won't refine the variable type to `String` inside the `sayHello()`, so one solution is to ignore its alerts and use non-null assertion. Note, however, that even in cases like this, it often makes sense to use less blunt tools for dealing with nulls or even rewriting the control-flow of your code in such a way that compiler can employ smart casts.

Using not-null assertion on a non-null receiver is not considered an error. Such code, though, is redundant and should be avoided.

IDE Tips: The IntelliJ plugin comes with an inspection which highlights and suggests to remove redundant usages of the `!!` operator.

Like any other postfix operator, the not-null assertion has the highest possible precedence.

Safe call operator

We've already mentioned that values of the nullable types don't allow you to call methods available for the corresponding non-nullable type. There is, however, a special safe-call operation which allows you to circumvent this restriction. Let's consider one of our earlier examples:

```
fun readInt() = readLine() !! .toInt()
```

This function works fine as long as your program uses console as its standard I/O. If, however, we've started the program piping some file as standard input, it could've failed with `KotlinNullPointerException` if the file in question was empty. Using the safe call operator, we can rewrite it into the following form:

```
fun readInt() = readLine()?.toInt()
```

The code above is basically equivalent to the function:

```
fun readInt(): Int? {
    val tmp = readLine()
    return if (tmp != null) tmp.toInt() else null
}
```

In other words, the safe call operator behaves like an ordinary call when its receiver (left-hand operand) is not null. When its receiver is null, however, it doesn't perform any call and simply returns null. Similarly, to `||` and `&&` operations, safe calls follow a lazy semantics: they do not evaluate call arguments if the receiver is null. In terms of precedence, the `?.` operator takes the same level as an ordinary call operator `(.)`.

The pattern *do something meaningful when receiver is not null or return the null otherwise* happens quite often in practice, so safe calls can greatly simplify your code by relieving you from unnecessary `if` expressions and temporary variable declarations. One useful idiom is to chain safe calls into something like this:

```
println(readLine()?.toInt()?.toString(16))
```

Note that since the safe call operator may return null, its type is always the nullable version of the corresponding non-safe call. We have to take this into account on the call site of our new `readInt()` function:

```
fun readInt() = readLine()?.toInt()
fun main() {
    val n = readInt() // Int?
    if (n != null) {
        println(n + 1)
    } else {
        println("No value")
    }
}
```

Like not-null assertions, safe calls can be applied to non-nullable receivers. Such code, however, is completely redundant as it behaves exactly like a simple dot-call `(.)`.

IDE Tips: The IntelliJ plugin automatically highlights redundant usages of the `?.` operator and suggests to replace them with ordinary calls.

Elvis operator

One more useful tool for dealing with nullable values is a null coalescing operator ?: that allows you to provide some default value in place of null. It's usually called the Elvis operator due to its resemblance to an emoticon of Elvis Presley. Let's consider an example:

```
fun sayHello(name: String?) {  
    println("Hello, " + (name ?: "Unknown"))  
}  
fun main() {  
    sayHello("John") // Hello, John  
    sayHello(null) // Hello, Unknown  
}
```

In other words, the result of this operator is the left argument when it's not null and the right one otherwise. Basically, the sayHello() function above is equivalent to the following code:

```
fun sayHello(name: String?) {  
    println("Hello, " + (if (name != null) name else  
"Unknown"))  
}
```

The Elvis operator is useful in combination with safe calls to substitute a default value when the receiver is null. In the following code, we substitute a zero when the program's standard input is empty:

```
val n = readLine()?.toInt() ?: 0
```

One more handy pattern is to use control-flow breaking statement like return or throw as a right argument of Elvis. This serves as an abbreviation of the corresponding if expression:

```
class Name(val firstName: String, val familyName: String?)  
class Person(val name: Name?) {  
    fun describe(): String {  
        val currentName = name ?: return "Unknown"  
        return "${currentName.firstName}  
${currentName.familyName}"  
    }  
}
```

```

fun main() {
    println(Person(Name("John", "Doe")).describe()) // John
    Doe
    println(Person(null).describe()) // Unknown
}

```

IDE Tips: The IntelliJ plugin has a special inspection that detects null-checking if expressions that can be replaced with the Elvis operator (*Figure 4.3*).

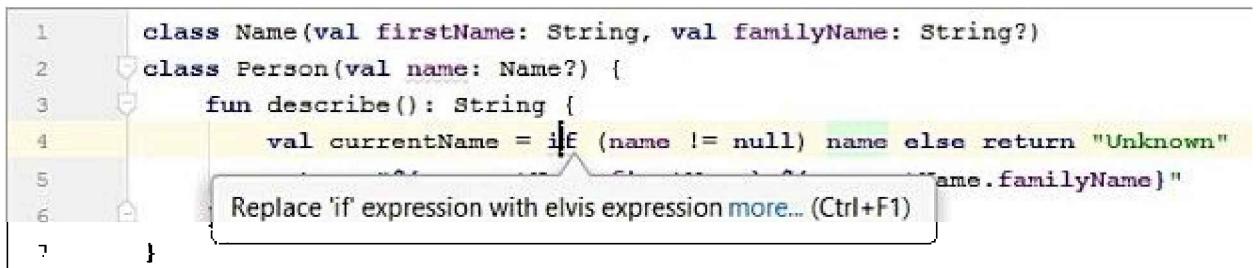


Figure 4.3: Replacing if expression with Elvis operator

In terms of precedence, the Elvis operator occupies an intermediate place between infix operations like or and in!/in operators yielding, in particular, to comparison/equality operators, ||, &&, and assignments.

Properties: beyond simple variables

In the first section, we've introduced you to an idea of property as a variable bound to a particular class instance or file facade similarly to Java field. In general, however, Kotlin properties possess far richer capabilities which go beyond simple variables offering you the means to control how the property value is read or written. In this section, we'll have a closer look at non-trivial property semantics.

Top-level properties

Similarly to classes or functions, properties may be declared at the top-level. In this case, they serve as a sort of global variables or constants:

```

val prefix = "Hello, " // top-level immutable property
fun main() {
    val name = readLine() ?: return
}

```

```
    println("$prefix$name")
}
```

Such properties may have one of top-level visibilities (public/internal/private). They also may be used in import directives:

```
// util.kt
package util
val prefix = "Hello, "
// main.kt
package main
import util.prefix
fun main() {
    val name = readLine() ?: return
    println("$prefix$name")
}
```

Late initialization

Sometimes, the requirement to initialize class properties upon its instantiation may be unnecessarily strict. Some properties can only be initialized later, after the class instance is already created, but before its actual use: they might be, for example, specified in some initialization method like unit test setup or assigned via dependency injection. One solution would be to assign some default value (for example, null) which basically means an uninitialized state in the constructor and provide an actual value when necessary. Consider, for example, the following code:

```
import java.io.File
class Content {
    var text: String? = null
    fun loadFile(file: File) {
        text = file.readText()
    }
}
fun getContentSize(content: Content) = content.text?.length
?: 0
```

We assume that `loadFile()` is called elsewhere to load string content from some file. The drawback of this example is that we have to deal with a nullable type while

the actual value is supposed to be always initialized before access and thus non-null. Kotlin provides a built-in support for this kind of pattern via the `lateinit` keyword. Let's apply it to our example:

```
import java.io.File
class Content {
    lateinit var text: String
    fun loadFile(file: File) {
        text = file.readText()
    }
}
fun getContentSize(content: Content) = content.text.length
```

Property with a `lateinit` marker works just like an ordinary property short of a single difference: on attempt to read its value, the program will check if the property is initialized, and throw `UninitializedPropertyAccessException` if it's not. This behavior is somewhat similar to an implicit `!!` operator.

There are some requirements a property must satisfy in order to be eligible for late initialization. First, it must be declared as mutable (`var`) since its value may be changed in different parts of your code. Second, it must have a non-nullable type and may not represent a primitive value like `Int` or `Boolean`. The reason is that internally the `lateinit` property is represented as a nullable variable with `null` reserved to mean uninitialized state. Finally, the `lateinit` property may not have an initializer since such construct would have defeated the purpose of declaring it `lateinit` in the first place.

Kotlin 1.2 has introduced a couple of `lateinit`-related improvements. In particular, it's now possible to use late initialization for top-level properties and local variables:

```
lateinit var text: String
fun readText() {
    text = readLine()!!
}
fun main() {
    readText()
    println(text)
}
```

Another improvement is an ability to check whether a `lateinit` property is

initialized before trying to access its value. We'll discuss how to do it in *Chapter 10, Annotations and Reflection*, which deals with the Kotlin reflection API.

Using custom accessors

The properties we've seen so far had essentially behaved like ordinary variables stored either in an instance of some Kotlin class, or in the context of a file (which on JVM is also represented as an instance of special facade class). The real power of Kotlin properties, however, comes from their ability to combine variable-and function-like behavior in a single declaration. This is achieved with custom accessors, which are special functions invoked when property value is accessed for reading or writing.

In the following example, we define custom getter, that is, the accessor which is used to read a property value:

```
class Person(val firstName: String, val familyName: String) {  
    val fullName: String  
    get(): String {  
        return "$firstName $familyName"  
    }  
}
```

The getter is placed at the end of a property definition and basically looks like a function albeit with a keyword `get` instead of a name. Whenever such property is read, the program automatically invokes its getter:

```
fun main() {  
    val person = Person("John", "Doe")  
    println(person.fullName) // John Doe  
}
```

Similar to functions, accessors support an expression-body form:

```
val fullName: String  
get() = "$firstName $familyName"
```

Note that a getter may not have any parameter, while its return type, if present, must be the same as the type of the property itself:

```
val fullName: Any  
get(): String { // Error
```

```
        return "$firstName $familyName"  
    }
```

Since Kotlin 1.1, you can just omit explicit property type and rely on the type inference instead:

```
val fullName  
    get() = "$firstName $familyName" // String is inferred
```

The value of the `fullName` property we've introduce above is computed upon each access. Unlike `firstName` and `familyName`, it doesn't have a backing field and thus doesn't occupy memory in a class instance. In other words, it's basically a function that simply has a property form. In Java, we'd usually introduce a method such as `getFullName()` for the same purpose. The rule regarding backing fields is as follows: the backing field is generated when property has at least one default accessor or a custom accessor, which explicitly mentions the field. Since immutable properties have only one accessor, a getter, and in our example, it doesn't reference the backing field directly, the `fullName` property will have no backing field.

What about the direct field reference? It's useful when you want your property to be based on some stored value, but still need to customize access. For example, we could use it to log property reads as follows:

```
class Person(val firstName: String, val familyName: String,  
age: Int) {  
    val age: Int = age  
    get(): Int {  
        println("Accessing age")  
        return field  
    }  
}
```

The backing field reference is represented by the `field` keyword and is valid only inside an accessor's body.

When a property doesn't use a backing field, it can't have an initializer because an initializer is basically a value assigned directly to the backing field upon initialization of a class instance. That's why we didn't add initializer for the `fullName` definition above: being a computed property, it doesn't need one.

Since property with a customer getter behaves like a parameterless function, albeit

with a slightly different syntax, this poses a question how you should choose between both constructs in a particular case. The official Kotlin coding conventions recommend using a property instead of a function when the computation doesn't result in throwing an exception, the value is cheap enough or cached, and different invocations produce the same result unless the state of the containing class instance is not changed.

Mutable properties defined with the `var` keyword have two accessors: a getter for reading and a setter for writing. Let's consider an example:

```
class Person(val firstName: String, val familyName: String) {
    var age: Int? = null
    set(value) {
        if (value != null && value <= 0) {
            throw IllegalArgumentException("Invalid age: $value")
        }
        field = value
    }
}
fun main() {
    val person = Person("John", "Doe")
    person.age = 20           // calls custom setter
    println(person.age) // 20, uses default getter
}
```

A property setter must have a single parameter of the same type as the property itself. The parameter type is usually omitted since it's always known in advance. By convention, the parameter is named `value`, but it's possible to choose a different name if you like.

Note that the property initializer does not trigger a setter call since the initializer value is assigned to the backing field directly.

Since mutable properties have two accessors, they always possess a backing field unless both accessors are custom and do not reference it via the `field` keyword. For example, the preceding `age` property has a backing field due to a default getter and direct mention in the setter, while the following property does not:

```
class Person(var firstName: String, var familyName: String) {
```

```

var fullName: String
get(): String = "$firstName $familyName"
set(value) {
    val names = value.split(" ") // Split string space-separated words
    if (names.size != 2) {
        throw IllegalArgumentException("Invalid full name: '$value'")
    }
    firstName = names[0]
    familyName = names[1]
}

```

Property accessors may have their own visibility modifiers. It can be useful if you, say, want to forbid changing your property outside of its containing class, thus making it effectively immutable for the outside world. If you don't need a non-trivial implementation of an accessor, you can abbreviate it by a single get/set keyword:

```

import java.util.Date
class Person(name: String) {
    var lastChanged: Date?
    private set // can't be changed outside Person class
    var name: String = name
    set(value) {
        lastChanged = Date()
        field = value
    }
}

```

Java versus Kotlin: From the JVM point of view, a Kotlin property, in general, corresponds to a one or two access or methods (like `getFullName()` and `setFullName()`) possibly backed by a private field. Although the method themselves are not available in the Kotlin code, they can be called from the Java classes and comprise a major point in Java/Kotlin interoperability. In *Chapter 12, Java Interoperability*, we'll discuss this issue in more details. Private properties, on the other hand, by default have no accessor methods generated since they can't be used outside of containing class or file: access to such properties is optimized to refer to their

backing fields directly.

Custom accessors are not allowed for `lateinit` properties since their accessors are always generated automatically. They are also not supported for properties declared as primary constructor parameters, but that can be solved by using ordinary non-property parameter and assigning its value to a property in the class body just like we did with the preceding `val` age.

Lazy properties and delegates

In a previous section, we've seen how to implement late initialization using the `lateinit` modifier. In many cases, though, we'd like to defer value computation until its first access. In Kotlin, this can be achieved with `lazy` properties. Let's consider an example:

```
import java.io.File
val text by lazy {
    File("data.txt").readText()
}
fun main() {
    while (true) {
        when (val command = readLine() ?: return) {
            "print data" -> println(text)
            "exit" -> return
        }
    }
}
```

The preceding `text` property is defined as `lazy`: we specify how it's initialized in the block coming after by the `lazy` clause. The value itself is not computed until we first access it in the `main()` function when a user types an appropriate command. After initialization, the property value is stored in a field and all successive attempts to access it will just read the stored value. If we have, for example, defined a property with a simple initializer:

```
val text = File("data.txt").readText()
```

The file would be read right on the program start, while the property with a getter like this:

```
val text get() = File("data.txt").readText()
```

Would reread the file every time the program tries to access the property value.

You can also specify property type explicitly, if necessary:

```
val text: String by lazy { File("data.txt").readText() }
```

This syntax is, in fact, a special case of the so-called delegated property that allows you to implement a property via a special delegate object which handles reading/writing and keeps all related data if necessary. The delegate is placed after `by` keyword and can be an arbitrary expression which returns object conforming to specific convention. In our example, `lazy {}` is not a built-in language construct, but rather just a call to standard library function with a lambda supplied (we've already seen a similar example in *Chapter 2, Language Fundamentals*, while discussing creation of array instances).

Kotlin provides some delegate implementations out of the box: aside of enabling `lazy` computations standard delegates allow you to create observable properties which notify a listener before/after every change of their value and to back properties by a map instead of storing them in separate fields. In this section, we'll give you a basic taste of delegates in the context of lazy properties and defer their comprehensive treatment till *Chapters 7, Exploring Collections and I/O*, and *Chapter 11, Domain-Specific Languages*, where we'll consider standard delegates available in the Kotlin library and the means to design your own delegates, respectively.

Note that unlike `lateinit` properties, `lazy` properties may not be mutable: they don't change the value once initialized:

```
var text by lazy { "Hello" } // Error
```

By default, `lazy` properties are thread-safe: in a multi-threaded environment, the value is computed by a single thread, and all threads trying to access a property will ultimately get the same result.

Since Kotlin 1.1, you can use delegates for local variables. This, in particular, allows you to define `lazy` variable in a function body:

```
fun longComputation(): Int {...}  
fun main(args: Array<String>) {  
    val data by lazy { longComputation() } // lazy local  
    variable
```

```
    val name = args.firstOrNull() ?: return
    println("$name: $data") // data is only accessed when
name is not null
}
```

Note that delegated properties currently do not support smart casts. Since delegates may have an arbitrary implementation, they are treated similar to properties with custom accessors. It also means that you can use smart casts with local delegated variables:

```
fun main() {
    val data by lazy { readLine() }
    if (data != null) {
        // Error: no smart cast, data is nullable here
        println("Length: ${data.length}")
    }
}
```

The `lazy` properties / local variables are not an exception: currently, you can't apply smart casts to them even though their values do not actually change after initialization.

Objects

In this section, we'll discuss the concept of object declarations. An object declaration in Kotlin is a kind of mix between a class and a constant which allows you to create singletons – classes which has exactly one instance. We'll also look at object expressions which play a role similar to Java's anonymous classes.

Object declarations

Kotlin has a built-in support of the singleton pattern which basically ensures that some class can only have a single instance. In Kotlin, you declare a singleton similar to class, but using the `object` keyword instead:

```
object Application {
    val name = "My Application"
    override fun toString() = name
    fun exit() { }
```

```
}
```

Such object declaration can be used as both a class and a value representing its instance. For example, look at the following code:

```
fun describe(app: Application) = app.name // Application as a type
fun main() {
    println(Application) // Application as a value
}
```

Note that using an object as a type is usually meaningless since such types have exactly one instance so you can just as well refer to that instance itself.

Object definitions are thread-safe: the compiler ensures that even if you concurrently access the singleton from different execution threads, there is still exactly one shared instance and initialization code is run only once.

The initialization itself happens lazily upon the loading of the singleton class, which usually happens when the program first refers to the object instance.

Java versus Kotlin: In Java, singletons have to be emulated using ordinary class declarations which is usually achieved through combination of private constructors and some static state. Such object declarations can have different features depending on the implementation details, the most common being lazy versus eager and thread-safe versus non-thread-safe singletons. Looking at the JVM bytecode of the Application object, we see that it basically amounts to the following Java class:

```
public final class Application {
    private static final String name = "My Application";
    public static final Application INSTANCE;
    private Application() { }
    public final String getName() {
        return name;
    }
    public final void exit() { }
    static {
        INSTANCE = new Application();
        name = "My Application";
    }
}
```

```
    }  
}
```

Note that the `INSTANCE` variable is not accessible in the Kotlin code itself, but can be used in Java classes referring to the Kotlin's singleton. In *Chapter 12, Java Interoperability*, we'll consider this issue in more details.

Similar to classes, object declarations can include member functions and properties as well as initializer blocks, but may not have primary or secondary constructors: an object instance is always created implicitly so constructor calls make no sense for objects.

Classes in the object body can't be marked as inner. Instances of inner classes are always associated with corresponding instance of their enclosing class, but object declarations have only one instance which makes `inner` modifier effectively redundant: that's the reason why it is forbidden.

Object members can be imported and later referred by their simple names similar to top-level declarations. Suppose, for example, the `Application` object is defined in a separate file:

```
import Application.exit  
fun main() {  
    println(Application.name) // using qualified reference  
    exit() // using simple name  
}
```

You may not, however, import all of the object members at once using an on-demand import:

```
import Application.* // Error
```

The reason behind such restriction is that object definitions, like any other classes, include common methods such as `toString()` or `equals()` which would be imported too if on-demand import was allowed.

Like classes, objects can be nested into other classes or even into other objects. Such declarations are also singletons which have exactly one instance per entire application. If you need a separate instance per enclosing class, you should use an inner class instead. You can't, however, put objects inside functions as well as local or inner classes because such definitions, in general, would depend on some enclosing context and thus couldn't be singletons. Locally-scoped objects can be created using an object

expression, which we'll consider further in this chapter.

Java versus Kotlin: In the Java world, you can often come across a so called utility class. It's essentially a class which doesn't have instances (usually by means of private constructor) and instead serves as a kind of grouping for related methods. This pattern proves to be useful in Java, but it's generally discouraged in Kotlin, although you can certainly declare utility-style classes if that's what you want. The reason is that, unlike Java, Kotlin has top-level declarations that can be grouped together using packages thus freeing you from the need to use special classes and reducing the boilerplate.

Companion objects

Similar to nested classes, nested objects can access private members of the enclosing class given its instance. A useful implication of this is an ability to easily implement the factory design pattern. There are cases when using constructor directly is unwanted: you can't, for example, return null or instances of different types (conforming to the class type) depending on some prechecks, since a constructor call always returns an instance of its class or throws an exception. A possible solution is to mark constructor as private, making it inaccessible outside of the class, and define a nested object with a function which serves as a `Factory` method and calls the class constructor when necessary:

```
class Application private constructor(val name: String) {
    object Factory {
        fun create(args: Array<String>): Application? {
            val name = args.firstOrNull() ?: return null
            return Application(name)
        }
    }
}
fun main(args: Array<String>) {
    // Direct constructor call is not permitted
    // val app = Application(name)
    val app = Application.Factory.create(args) ?: return
    println("Application started: ${app.name}")
}
```

Note that in this case, we have to refer to the object name every time we call the factory method unless it's imported using the import

`Application.Factory.create` directive. Kotlin allows you to solve this problem by turning the `Factory` object into the companion. A companion object is basically a nested object marked with the `companion` keyword. Such object behaves just like any other nested object with one exception: you can refer to its members by the name of enclosing class without mentioning the name of companion object itself. Using companions, we can make our previous example a slightly more concise:

```
class Application private constructor(val name: String) {  
    companion object Factory {  
        fun create(args: Array<String>): Application? {  
            val name = args.firstOrNull() ?: return null  
            return Application(name)  
        }  
    }  
}  
  
fun main(args: Array<String>) {  
    val app = Application.create(args) ?: return  
    println("Application started: ${app.name}")  
}
```

Although it's considered redundant, you can still refer to the `companion object` members using its name:

```
val app = Application.Factory.create(args) ?: return
```

IDE Tips: IntelliJ automatically warns you about unnecessary references to companion and suggests removing them from the code (Figure 4.4):



Figure 4.4: Redundant companion reference

For the companion object, you can also skip the name in the definition itself. This is the recommended approach:

```

class Application private constructor(val name: String) {
    companion object {
        fun create(args: Array<String>): Application? {
            val name = args.firstOrNull() ?: return null
            return Application(name)
        }
    }
}

```

When the companion name is omitted, the compiler assumes the default name `Companion`.

Note that companion name must be mentioned explicitly when you import its members:

```

import Application.Companion.create // OK
import Application.create           // Error

```

A class may not have more than one companion:

```

class Application {
    companion object Factory
    companion object Utils      // Error: only one companion
is allowed
}

```

It's also an error to use the `companion` modifier for a top-level object or an object nested into another object: in the former case, you lack a class definition to bind the companion to, while in the latter, the companion is basically redundant.

Java versus Kotlin: Companion objects in Kotlin may be considered a counterpart of Java's static context: like statics, companion members share the same global state and can access any member of the enclosing class regardless of its visibility. The crucial difference, however, is that their global state is an object instance. This gives much more flexibility than Java's statics as companion objects may have supertypes and passed around like any other object. In *Chapters 8, Understanding Class Hierarchies* and *Chapter 11, Domain-Specific Languages*, we'll see how companion objects can be combined with inheritance and language conventions to produce a more expressive code.

Note also that `init` blocks in companion objects can be used similar to Java static initializers.

Object expressions

Kotlin has a special kind of expression that creates a new object without an explicit declaration. This object expression is very similar to a Java anonymous class. Consider the following example:

```
fun main() {
    fun midPoint(xRange: IntRange, yRange: IntRange) = object {
        val x = (xRange.first + xRange.last)/2
        val y = (yRange.first + yRange.last)/2
    }
    val midPoint = midPoint(1..5, 2..6)
    println("${midPoint.x}, ${midPoint.y}") // (3, 4)
}
```

An object expression looks just like an object definition without a name and being an expression can be, for example, assigned to a variable like in the preceding example. Note that unlike classes and object expressions, named objects can't be declared inside functions:

```
fun printMiddle(xRange: IntRange, yRange: IntRange) {
    // Error
    object MidPoint {
        val x = (xRange.first + xRange.last)/2
        val y = (yRange.first + yRange.last)/2
    }
    println("${MidPoint.x}, ${MidPoint.y}")
}
```

The rationale behind this decision is that object definitions are supposed to represent singletons, while local objects, if they were allowed, in general, would have to be created anew upon every call of the enclosing function.

Since we've defined no explicit type for the object returned by the `midPoint()` function, you might be wondering what the return type is. The answer is a so called anonymous object type which represents a class with all the members defined in the object expression and a single instance. This type is not denotable in the language itself: it's just an internal representation of the object expression type used by the Kotlin compiler. We can still use expressions of anonymous type similar to any other

class instances: for example, to access its members as evidenced by the `println()` call above.

IDE Tips: If we try to look at object expression type using **Show Expression Type** action (Ctrl + Shift + P/ Cmd + Shift + P), IntelliJ would show us the `<anonymous object>` placeholder (*Figure 4.5*):

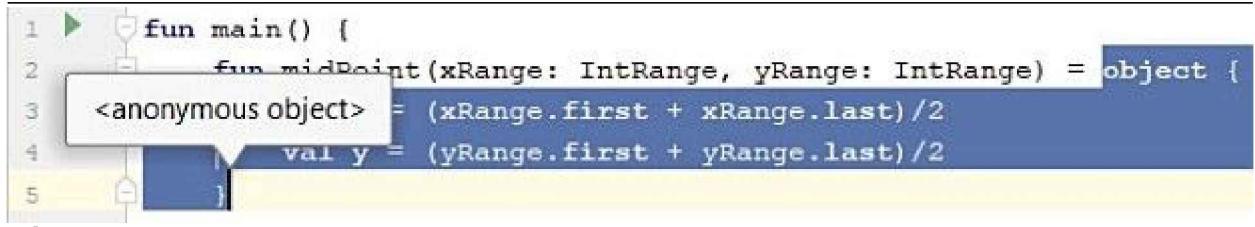


Figure 4.5: Anonymous object type

This example also demonstrates that function with object expression body have an anonymous return type and the same is also true for local variables and properties:

```
fun main() {  
    val o = object { // anonymous object type is inferred  
        val x = readLine()!!.toInt()  
        val y = readLine()!!.toInt()  
    }  
    println(o.x + o.y) // can access x and y here  
}
```

Note, however, that anonymous types are only propagated to local or private declarations. If we, for example, were to declare the `midPoint()` function as a top-level one, we would get a compile-time error on attempt to access object members:

```
fun midPoint(xRange: IntRange, yRange: IntRange) = object {  
    val x = (xRange.first + xRange.last)/2  
    val y = (yRange.first + yRange.last)/2  
}  
fun main() {  
    val midPoint = midPoint(1..5, 2..6)  
    // Error: x and y are unresolved  
    println("${midPoint.x}, ${midPoint.y}")  
}
```

Now, the return type of the `midPoint()` function is not an anonymous type of our object expression, but rather its denotable supertype. Since our object has no explicit supertype, it's assumed to be `Any`. That's why the `midPoint.x` reference becomes unresolved.

Similar to local functions and classes, object expressions can capture variables from the enclosing code. Mutable captured variables can be modified in the object's body: in this case, the compiler creates necessary wrappers to share the data similar to the local classes.

```
fun main() {
    var x = 1
    val o = object {
        fun change() {
            x = 2
        }
    }
    o.change()
    println(x) // 2
}
```

Note that unlike object declarations which are initialized lazily, object expressions are initialized immediately after their instance is created. For example, the code below will print `x = 2` since at the point, the `x` variable is read, the initialization code in the object expression has already been executed:

```
fun main() {
    var x = 1
    val o = object {
        val a = x++;
    }
    println("o.a = ${o.a}") // o.a = 1
    println("x = $x")      // x = 2
}
```

Just like Java's anonymous classes, object expressions are most useful when combined with class inheritance: they give you a concise way to describe a small modification based on existing class without explicit subclass definition. We'll consider them in *Chapter 8, Understanding Class Hierarchies*.

Conclusion

In conclusion, let's summarize the things we've learned in this chapter. We now have a basic understanding of how to define and use custom types based on Kotlin classes, how to properly initialize class instances and use singleton objects. We've learned to use different kinds of properties to program custom read/write behavior. Finally, we're now able to employ powerful type nullability mechanism for improving our program safety.

We'll revisit object-oriented aspects of Kotlin in the upcoming chapters. In particular, in *Chapter 6, Using Special-Case Classes*, will deal with special classes covering common programming patterns, while *Chapter 8, Understanding Class Hierarchies*, will address the issue of inheritance and building class hierarchies.

For the next chapter, we'll switch to a different topic and get to know another major paradigm powering Kotlin development: the functional programming. We will introduce you to lambdas, discuss higher-order functions and show how to use extension functions and properties for adding new features to existing types.

Questions

1. Describe a basic class structure in Kotlin. How does it compare to Java classes?
2. What is a primary constructor?
3. What is a secondary constructor? How would you decide which constructor(s) a class should contain and whether secondary constructor are necessary?
4. What are the supported member visibilities in Kotlin? How do they differ from the visibilities in Java?
5. What's the difference between inner and non-inner nested classes in Kotlin? Compare them with Java's counterparts.
6. Can you define a class inside a function body? What are the limitations?
7. What's the gist of the late initialization mechanism? What's the advantage of using `lateinit` compared to a nullable property?
8. What are custom property accessors? Compare them with conventional getter and setter methods in Java.
9. Can you define an effectively read-only property which behaves like a `val` for the class client? What about an effectively write-only property?

10. How can you achieve lazy computation with delegated properties? Compare `lazy` and `lateinit` properties.
11. What is object declaration? Compare Kotlin objects with common singleton implementation used in the Java development.
12. What are limitations of object declarations as compared to classes?
13. What's the difference between ordinary object and a companion one?
14. Compare Kotlin companion objects with Java's statics.
15. What is the Kotlin's counterpart of Java's anonymous classes? How do you use one?

CHAPTER 5

Leveraging Advanced Functions and Functional Programming

In this chapter, we'll address some advanced issues related to using functions and properties. The first half is devoted to the fundamentals of functional programming in Kotlin. We'll introduce you to a concept of a higher-order function, describe how to construct functional values using lambdas, anonymous functions and callable references and show how inline functions can help you to use functional programming with almost zero runtime overheads. In the second half, we'll consider a matter of extension functions and properties which allow you to add new features to existing types without their modification.

Structure

- Lambdas and higher-order functions
- Functional types
- Callable references
- Inline functions
- Non-local returns control-flow
- Extension functions and properties
- Extension lambdas
- Scope functions

Objective

Learn to make use of functional Kotlin features with higher-order functions, lambdas, and callable references as well as employ extension functions and properties for enriching existing types.

Functional programming in Kotlin

In this section, we'll introduce you to the Kotlin features enabling the support of functional paradigm. Functional programming is based around the idea of presenting the program code as a composition of functions manipulating immutable data. Functional languages allow treating functions like first-class values, which means that they have the same basic capabilities as value of any other type: in particular, they can be assigned to / read from variables as well as passed to / returned from functions. This enables definition of so-called higher-order functions that manipulate other functional values like data providing flexible mechanism for code abstraction and composition.

Higher-order functions

In the previous chapter, we've already seen some examples of using lambdas to perform computations. For example, the array constructor call takes a lambda, which computes an array element given its index:

```
val squares = IntArray(5) { n -> n*n } // 0, 1, 4, 9, 16
```

In this section, we'll take a more detailed look at lambdas and higher-order functions.

Suppose that we want to define a function that computes a sum of elements in an integer array:

```
fun sum(numbers: IntArray): Int {
    var result = numbers.firstOrNull()
    ?: throw IllegalArgumentException("Empty array")
    for (i in 1..numbers.lastIndex) result += numbers[i]
    return result
}
fun main() {
    println(sum(intArrayOf(1, 2, 3))) // 6
}
```

What if we want to generify this function to cover other kinds of aggregates such as product or min/max value? We can keep the basic iteration logic in the function itself and extract computation of intermediate values into a functional parameter, which can be supplied at the call site:

```
fun aggregate(numbers: IntArray, op: (Int, Int) -> Int): Int
{
```

```

var result = numbers.firstOrNull()
?: throw IllegalArgumentException("Empty array")
for (i in 1..numbers.lastIndex) result = op(result,
numbers[i])
return result
}
fun sum(numbers: IntArray) =
    aggregate(numbers, { result, op -> result + op })
fun max(numbers: IntArray) =
    aggregate(numbers, { result, op -> if (op > result) op
else result })
fun main() {
    println(sum(intArrayOf(1, 2, 3))) // 6
    println(sum(intArrayOf(1, 2, 3))) // 3
}

```

What distinguishes the `op` parameter is a functional type (`Int, Int`) \rightarrow `Int` describing values, which can be called like functions. In our example, the `op` parameter may accept functional values which accept a pair of `Int` values and return some `Int` as their result.

At the call site in `sum()` and `max()` functions, we pass a lambda expression which denotes such functional values. It's basically a definition of local function without a name which uses a kind of simplified syntax. For example, in the expression:

```
{ result, op -> result + op }
```

`result` and `op` play the role of function parameters while the expression after \rightarrow computes the result. No explicit `return` statement is necessary in this case, and parameter types are inferred automatically from the context.

Let's now examine these features in more detail.

Functional types

The functional type describes values that can be used like functions. Syntactically such type is similar to a function signature and contains the following two components:

1. List of parentheses-enclosed argument types that determine which data can be passed to the functional value;

2. A return type which determines the type of result returned by the value of the functional type.

Note that the return type must be always specified explicitly even if it's the `Unit`.

For example, the type `(Int, Int) -> Boolean` represents a function that takes a pair of integers as its input and returns a Boolean value as a result. Note that unlike function definition, the return type and argument list in a function type notation are separated by the `->` character instead of a colon (`:`).

The value of the functional type can be invoked just like an ordinary function: `op(result, numbers[i])`. An alternative way is to use an `invoke()` method, which takes the same arguments:

```
result = op.invoke(result, numbers[i])
```

Java versus Kotlin: In Java 8+, any interface with a **single abstract method (SAM)** may be considered a functional type given appropriate context and instantiated with a lambda expression or method reference. In Kotlin, however, functional values always have a type of the form `(P1, ..., Pn) -> R` and cannot be implicitly cast to an arbitrary SAM interface. So while the following code is valid in Java:

```
import java.util.function.Consumer;
public class Main {
    public static void main(String[] args) {
        Consumer<String> consume = s ->System.out.println(s);
        consume.accept("Hello");
    }
}
```

The similar code in Kotlin would not compile:

```
import java.util.function.Consumer
fun main() {
    // Error: type mismatch
    val consume: Consumer<String> = { s ->println(s) }
    consume.accept("Hello")
}
```

Kotlin, however, does support simplified conversion between function types and SAM interfaces declared in Java for the sake of Kotlin/Java interoperability. We'll see examples of this conversion in *Chapter 12, Java Interoperability*.

The parameter list may be empty if a function represented by a functional type do not take any parameters:

```
fun measureTime(action: () -> Unit): Long {  
    val start = System.nanoTime()  
    action()  
    return System.nanoTime() - start  
}
```

Note that parentheses around parameter types are mandatory even if the function type has a single parameter or none at all:

```
val inc: (Int) -> Int = { n -> n + 1 } // Ok  
val dec: Int -> Int = { n -> n - 1 } // Error
```

The values of functional types are not limited to function parameters. In fact, they may be used on equal terms with any other type. For example, you can store the functional value in a variable:

```
fun main() {  
    val lessThan: (Int, Int) -> Boolean = { a, b -> a < b }  
    println(lessThan(1, 2)) // true  
}
```

Note that if you omit a variable type, the compiler won't have enough information to infer types of lambda parameters:

```
val lessThan = { a, b -> a < b } // Error
```

In such cases, you'll have to specify parameter types explicitly:

```
val lessThan = { a: Int, b: Int -> a < b } // Ok
```

Just like any other type, a functional type may be nullable. In this case, we enclose the original type in parentheses before adding a question mark:

```
fun measureTime(action: ((() -> Unit)?)): Long {  
    val start = System.nanoTime()  
    action?.invoke()  
    return System.nanoTime() - start  
}
```

```
fun main() {
    println(measureTime(null))
}
```

If we don't do that, the effect would be different: `() -> Unit?` would describe functions that return a value of `Unit?`.

Functional types may be nested in which case they represent higher-order functions themselves:

```
fun main() {
    val shifter: (Int) -> (Int) -> Int = { n -> { i ->i + n }
}
val inc = shifter(1)
val dec = shifter(-1)
println(inc(10)) // 11
println(dec(10)) // 9
}
```

Note that `->` is right-associative, so `(Int) -> (Int) -> Int` actually means `(Int) -> ((Int) -> Int)`, that is, a function which takes an `Int` and returns another function which maps an `Int` to an `Int`. If we want it to mean the function which takes an `Int-to-Int` function and returns an `Int`, we have to use parentheses:

```
fun main() {
    val evalAtZero: ((Int) -> (Int)) -> Int = { f -> f(0) }
    println(evalAtZero { n -> n + 1 }) // 1
    println(evalAtZero { n -> n - 1 }) // -1
}
```

A functional type may include optional names for its parameters. They can be used for documentation purpose to clarify the meaning of a functional value this type represent:

```
fun aggregate(
    numbers: IntArray,
    op: (resultSoFar: Int, nextValue: Int) -> Int
): Int {...}
```

IDE Tips: IntelliJ IDEA allows you to see these parameters name using Parameter Info feature which gives you hints about a function signature when you press Ctrl + P (Cmd + P) inside its call (refer to the *Figure 5.1* for an example):

```
1 fun aggregate(numbers: IntArray, op: (resultSoFar: Int, nextValue: Int) -> Int): Int {  
2     var result = numbers.firstOrNull()  
3     ?: throw IllegalArgumentException("Empty array")  
4     for (i in 1..numbers.lastIndex) result = op(result, numbers[i])  
5     return result  
6 }  
7  
8     numbers: IntArray op: (resultSoFar: Int, nextValue: Int) -> Int  
9  
10    fun sum(numbers: IntArray) = aggregate(numbers) { result, op -> result + op }
```

Figure 5.1: Viewing functional parameter names with “Parameter Info”

Lambdas and anonymous functions

How do we construct a particular value of a functional type? One way is to use a lambda expression which basically describes a function without giving it a name. Let's, for example, define two more functions which compute sum and maximum value using the earlier `aggregate()` declaration:

```
fun sum(numbers: IntArray) =  
aggregate(numbers, { result, op -> result + op })  
fun max(numbers: IntArray) =  
aggregate(numbers, { result, op -> if (op > result) op else  
result })  
fun main() {  
    println(sum(intArrayOf(1, 2, 3))) // 6  
    println(sum(intArrayOf(1, 2, 3))) // 3  
}
```

The expression:

```
{ result, op -> result + op }
```

is called lambda expression. Similar to a function definition it consists of the following:

- The parameter list: `result, op`
- A list of expressions or statements which comprises the lambda body: `result + op`

```
+ op
```

Unlike function definition, you can't specify a return type: it's inferred automatically from the lambda body. Also, the last expression in the body is treated as lambda result, so you don't need to use an explicit return statement at the end.

Note that the lambda parameter list is not enclosed in parentheses. Parentheses around lambda parameters are reserved for so-called de-structuring declarations, which we'll cover in *Chapter 6, Using Special-Case Classes*.

When lambda is passed as the last argument, it can be placed outside parentheses. This is, in fact, the recommended code style we've already seen in the examples of array construction calls and lazy properties:

```
fun sum(numbers: IntArray) =  
    aggregate(numbers) { result, op -> result + op }  
fun max(numbers: IntArray) =  
    aggregate(numbers) { result, op -> if (op > result)  
op else result }
```

IDE Tips: IntelliJ plugin warns you about the cases when lambda can be passed outside an ordinary argument list and can automatically perform the necessary code changes.

When lambda has no arguments, an arrow symbol `->` can be omitted:

```
fun measureTime(action: () -> Unit): Long {  
    val start = System.nanoTime()  
    action()  
    return System.nanoTime() - start  
}  
val time = measureTime{ 1 + 2 }
```

Kotlin also has a simplified syntax for lambdas with a single parameter: in such cases, we can omit both the parameter list and an arrow and refer to the parameter by the predefined name `it`:

```
fun check(s: String, condition: (Char) -> Boolean): Boolean {  
    for (c in s) {  
        if (!condition(c)) return false  
    }
```

```

        return true
    }
}

fun main() {
    println(check("Hello")) { c ->c.isLetter() } // true
    println(check("Hello")) { it.isLowerCase() } // false
}

```

IDE Tips: The IntelliJ plugin allows you to convert lambda with it into a lambda with an explicit parameter and vice versa. These actions are available via the Alt + Enter menu when caret is positioned on either the parameter reference or parameter definition (see the *Figure 5.2*):

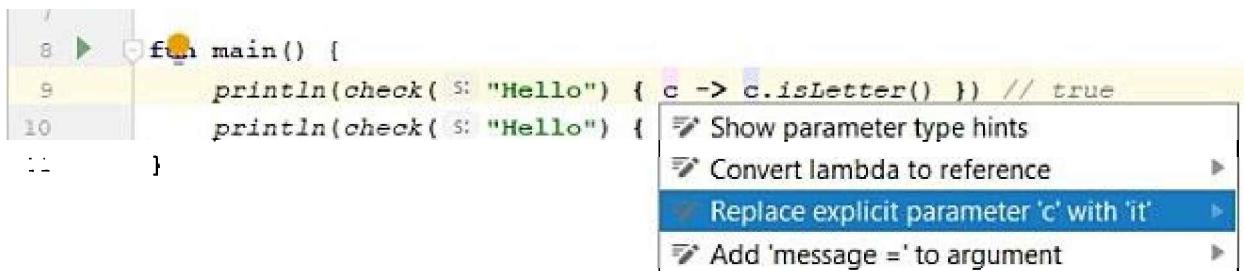


Figure 5.2: Converting explicit parameter to it

Since Kotlin 1.1, you can put underscore symbols (`_`) in place of an unused lambda parameters:

```

fun check(s: String, condition: (Int, Char) -> Boolean) : Boolean {
    for (i in s.indices) {
        if (!condition(i, s[i])) return false
    }
    return true
}

fun main() {
    println(check("Hello")) { _, c ->c.isLetter() } // true
    println(check("Hello")) { i, c ->i == 0 || c.isLowerCase() } // true
}

```

Another way to specify a functional value is to use an anonymous function:

```
fun sum(numbers: IntArray) =  
    aggregate(numbers, fun(result, op) = result + op)
```

An anonymous function has almost the same syntax as an ordinary function definition, albeit with a few differences; which are as follows:

- An anonymous function doesn't have a name, so the `fun` keyword is immediately followed by a parameter list;
- Similar to lambdas, you can omit explicit specification of parameter types if they can be inferred from the context
- Unlike a function definition, an anonymous function is an expression, so it can be, for example, passed to function as an argument or assigned to a variable (this parallels similar difference between object definitions and anonymous object expressions).

Unlike lambdas, anonymous functions allow you to specify the return type. In this regard, they follow the same rules as function definitions: the return type is optional (and can be inferred) if a function has an expression body, and must be explicit (unless it's the `Unit` type) when using a block body:

```
fun sum(numbers: IntArray) =  
    aggregate(numbers, fun(result, op): Int { return result + op  
})
```

Note that unlike lambdas anonymous functions can't be passed outside the argument list.

IDE Tips: The IntelliJ plugin include actions for automatic conversion between lambdas and anonymous functions. To access it, you need to place an editor caret on lambda's opening brace or `fun` keyword and press `Alt + Enter` (as shown in the *Figure 5.3*):

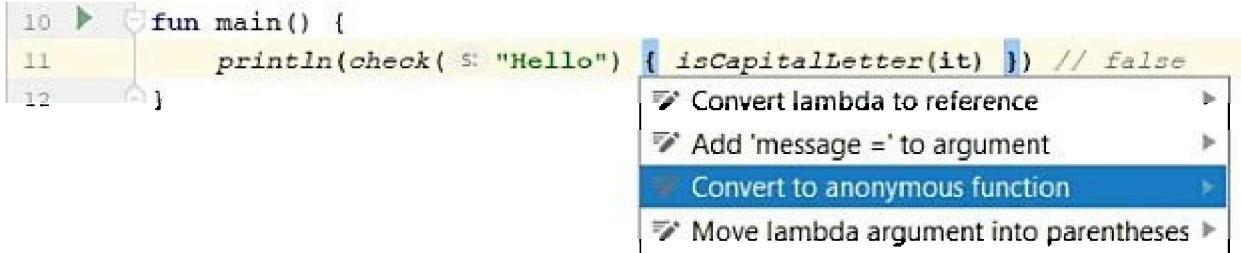


Figure 5.3: Converting lambda expression to an anonymous function

Similar to local functions, lambdas and anonymous functions can access their closure, or variables defined in their containing declaration. In particular, they can change mutable variables from the outer scope:

```
fun forEach(a: IntArray, action: (Int) -> Unit) {
    for (n in a) {
        action(n)
    }
}
fun main() {
    var sum = 0
    forEach(intArrayOf(1, 2, 3, 4)) {
        sum += it
    }
    println(sum) // 10
}
```

Java versus Kotlin: Java lambdas, on the contrary, may not modify any outer variables. This is similar to the case of modifying outer variables from local classes and anonymous objects we've discussed in *Chapter 4, Working with Classes and Objects*.

Callable references

In the previous section, we've seen how to construct a new functional value using lambdas and anonymous functions. But what if we already have a function definition and want to, for example, pass it as a functional value into some higher-order function? We can, of course, wrap it in a lambda expression as shown in the following code:

```
fun check(s: String, condition: (Char) -> Boolean): Boolean {
```

```

        for (c in s) {
            if (!condition(c)) return false
        }
        return true
    }
fun isCapitalLetter(c: Char) = c.isUpperCase() &&c.isLetter()
fun main() {
    println(check("Hello") { c ->isCapitalLetter(c) }) // false
}

```

In Kotlin, however, there is a much more concise way to use an existing function definition as an expression of a functional type. This is achieved through the use of callable references:

```

fun main() {
    println(check("Hello", ::isCapitalLetter)) // false
}

```

The `::isCapitalLetter` expression denotes a function value which behaves exactly like the `isCapitalLetter()` function it refers.

IDE Tips: The IntelliJ plugin provides a pair of actions which can transform lambda expressions to callable references (if possible) and vice versa. These actions can be accessed via Alt + Enter menu (as shown in the *Figure 5.4*):

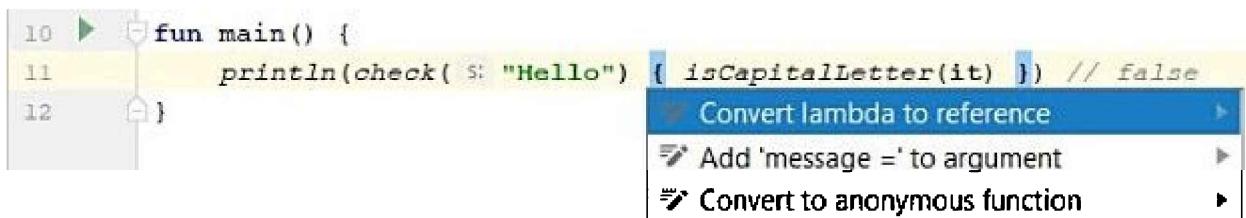


Figure 5.4: Converting lambda to a callable reference

The simplest kind of a callable reference is based on a top-level or local function. To compose such reference, you just need to prefix the function name with the `::` operator:

```

fun evalAtZero(f: (Int) -> Int) = f(0)
fun inc(n: Int) = n + 1
fun main() {

```

```
fun dec(n: Int) = n - 1
println(evalAtZero(::inc)) // 1
println(evalAtZero(::dec)) // -1
}
```

Callable reference may only mention a function by its simple name, so if a top-level function is located in another package, it must be imported first.

On applying the `::` operator to a class name, you get a callable reference to its constructor as follows:

```
class Person(val firstName: String, val familyName: String)
fun main() {
    val createPerson= ::Person
    createPerson("John", "Doe")
}
```

Another form of the `::` operator introduced in Kotlin 1.1 is called a bound callable reference. You can use it to refer to a member function in a context of a given class instance:

```
class Person(val firstName: String, val familyName: String) {
    fun hasNameOf(name: String) = name.equals(firstName,
ignoreCase = true)
}
fun main() {
    val isJohn = Person("John", "Doe")::hasNameOf
    println(isJohn("JOHN")) // true
    println(isJohn("Jake")) // false
}
```

There is also a third form that allows you to refer to a member function without binding it to a particular instance. We'll discuss this in the *Callable reference with receiver* section.

Note that callable references by themselves are not able to distinguish between overloaded functions. You have to provide an explicit type if compiler is not able to choose a particular overload:

```
fun max(a: Int, b: Int) = if (a > b) a else b
fun max(a: Double, b: Double) = if (a > b) a else b
```

```
val f: (Int, Int) -> Int = ::max // Ok
val g = ::max                         // Error: ambiguous
reference
```

The ability to specify a particular function signature in a callable reference may be added in a future version of Kotlin. For this reason, using parentheses after callable reference is currently reserved to accommodate a possible refinement of syntax. If you want to use a callable reference in a call, you have to enclose it in parentheses:

```
fun max(a: Int, b: Int) = if (a > b) a else b
fun main() {
    println(::max(1, 2)) // 2
    println(::max(1, 2)) // Error: this syntax is
reserved for future use
}
```

Callable reference can also be constructed for Kotlin properties. Such references, however, are not functional values by themselves, but rather a reflection objects containing property information. Using the `getter` property, we can access the functional value corresponding to the getter function. For a `var` declaration, the `setter` property similarly allows you to refer to setter:

```
class Person(var firstName: String, var familyName: String)
fun main() {
    val person = Person("John", "Doe")
    val readName = person::firstName.getter           // reference to getter
    val writeFamily = person::familyName.setter // reference to setter
    println(readName())                          // John
    writeFamily("Smith")
    println(person.familyName)                  // Smith
}
```

Callable references to local variables are currently not supported, but may be added in a future version.

Java versus Kotlin: Readers familiar with Java would probably recognize the similarity between Kotlin callable references and method references introduced in Java 8.

Although their semantics is indeed very similar, there are some important differences. First, callable references are more varied due to the fact that Kotlin supports declarations that have no direct counterparts in Java, such top-level and local functions as well as properties. Second, while Kotlin callable references are first-class expressions, Java's method references only make sense in the context of some functional interface: they don't have a definite type of their own. On top of that, a callable reference is not just a functional value but also a reflection object that you can use to obtain function or property attributes at runtime. In *Chapter 10, Annotations and Reflection*, we'll address the reflection API in more detail.

Inline functions and properties

Using higher-order functions and functional values is fraught with a certain performance overheads since each function is represented as an object. Moreover, when a lambda or an anonymous function in question uses variables from outer scope, it has to be created anew each time you pass it into a higher-order call to reflect a change of context. Invocations of functional values have to be dispatched through virtual calls that choose function implementation at runtime as compiler in general has no way to infer it statically.

Kotlin, however, provides a solution that can reduce runtime penalties of using functional values. The basic idea is to inline a higher-order function at its usage replacing a call with a copy of its body. To distinguish such functions, you need to mark them with the `inline` modifier.

Suppose, for example, that we have a function that searches a value in an integer array given a predicate it must satisfy:

```
inline fun indexOf(numbers: IntArray, condition: (Int) ->
Boolean): Int {
    for (i in numbers.indices) {
        if (condition(numbers[i])) return i
    }
    return -1
}
fun main() {
    println(indexOf(intArrayOf(4, 3, 2, 1)) { it < 3 }) // 2
}
```

Since the `indexOf()` function is inlined, the compiler will substitute its body instead of the function call. This means that the `main()` function will be basically equivalent to the code:

```
fun main() {  
    val numbers = intArrayOf(4, 3, 2, 1)  
    var index = -1  
    for (i in numbers.indices) {  
        if (numbers[i] < 3) {  
            index = i  
            break  
        }  
    }  
    println(index)  
}
```

Although inline functions can increase the size of the compiled code, when used reasonably, they can boost performance especially when a function in question is relatively small. Many higher-order functions provided by the Kotlin standard library we'll see in *Chapter 7, Exploring Collections and I/O* are inline.

Note that unlike some programming languages supporting function inlining (such as C++), the `inline` modifier in Kotlin is not an optimization hint which may be ignored depending on the compiler decision. Kotlin functions marked with `inline` are always inlined when it's possible, and when inlining can't be performed, usage of an `inline` modifier is considered a compilation error.

The preceding example demonstrates that the `inline` modifier affects not just a function it's applied to, but also the functional values that serve as its parameters. This in turn restricts all possible manipulations with such lambdas inside an inline function. Since inlined lambdas won't exist as a separate entity at runtime, they can't be, for example, stored in a variable or passed to a non-inline function. There are only two things we can do with an inlinable lambda: call it or pass as inlinable argument into another inline function:

```
var lastAction: () -> Unit = {}  
inline fun runAndMemorize(action: () -> Unit) {  
    action()  
    lastAction = action // Error
```

```
}
```

For the same reason, it's not allowed to inline values of a nullable functional type:

```
inline fun forEach(a: IntArray, action: ((Int) -> Unit)?) {  
    // Error  
    if (action == null) return  
    for (n in a) action(n)  
}
```

In such cases, we can forbid inlining of a particular lambda argument by marking it with a `noinline` modifier:

```
// Error  
inline fun forEach(a: IntArray, noinline action: ((Int) ->  
Unit)?) {  
    if (action == null) return  
    for (n in a) action(n)  
}
```

Note that when a function has no inlinable parameters, it's usually not worth inlining at all since substituting its body at call site will unlikely make a significant difference at runtime. For this reason, the Kotlin compiler marks such functions with a warning.

What if we try to use private members in a public inline function? Since the body of inline function is substituted instead of a call, it might let some external code to break encapsulation. To avoid this, Kotlin forbids references to private members, which may be leaked to the external code:

```
class Person(private val firstName: String,  
            private val familyName: String) {  
    inline fun sendMessage(message: () -> String) {  
        println("$firstName $familyName: ${message()}" ) //  
Error  
    }  
}
```

Note that if we'd marked the `sendMessage()` function or its containing class with the `private` modifier, the code would've compiled since references to private members in the `sendMessage()` body wouldn't have leaked outside the `Person`

class.

Starting from version 1.1, Kotlin supports inlining of property accessors. This may be useful for improving performance of reading/writing a property by eliminating a function call. In the following code all calls of the `fullName` getter are inlined:

```
class Person(var firstName: String, var familyName: String) {  
    var fullName  
    inline get() = "$firstName $familyName" // Inline getter  
    set(value) { ... } // Non-inline setter  
}
```

Besides inlining individual accessors, you may also mark a property itself with the `inline` modifier. In this case, the compiler will inline both getter and setter (if property is mutable):

```
class Person(var firstName: String, var familyName: String) {  
    inline var fullName // Inline getter and setter  
    get() = "$firstName $familyName"  
    set(value) { ... }  
}
```

Note that inlining is only supported for properties without a backing field. Also, similar to functions, you may not refer to private declarations if your property is public:

```
class Person(private val firstName: String,  
            private val familyName: String) {  
    inline var age = 0 // Error: property has a backing  
    field  
    // Error: firstName and familyName are private  
    inline val fullNameget() = "$firstName $familyName"  
}
```

Non-local control flow

Using higher-order functions raises some issues with instructions that break normal control flow such as the `return` statement. Consider the following code:

```
fun forEach(a: IntArray, action: (Int) -> Unit) {  
    for (n in a) action(n)
```

```
}
```

```
fun main() {
    forEach(intArrayOf(1, 2, 3, 4)) {
        if (it < 2 || it > 3) return
        println(it) // Error
    }
}
```

The intention was to return from lambda before printing a number if it doesn't fit into a range. However, this code won't compile: this happens because a `return` statement by default is related to the nearest enclosing function defined with `fun`, `get`, or `set` keywords. So, in our example, we're trying to return from the `main()` function instead. Such a statement, also known as non-local return, is forbidden because on JVM there is no efficient way that would allow a lambda to force return of its enclosing function. One way to solve the problem is to use an anonymous function instead:

```
fun main() {
    forEach(intArrayOf(1, 2, 3, 4), fun(it: Int) {
        if (it < 2 || it > 3) return
        println(it)
    })
}
```

If we do want to return from a lambda itself, we need to qualify the `return` statement with a context name similar to labeled `break` and `continue`. In general, the context name can be introduced by labeling a function literal expression. For example the code below assigns the `myFun` label to the lambda in the variable initializer:

```
val action: (Int) -> Unit = myFun@ {
    if (it < 2 || it > 3) return@myFun
    println(it)
}
```

When lambda is passed as an argument to a higher-order function, however, it's possible to use that function's name as a context without introducing an explicit label:

```
forEach(intArrayOf(1, 2, 3, 4)) {
    if (it < 2 || it > 3) return@forEach
    println(it)
}
```

```
}
```

Qualified returns are available in ordinary functions as well. You can use a function name as a context although usually such qualification is redundant:

```
fun main(args: Array<String>) {
    if (args.isEmpty()) return@main
    println(args[0])
}
```

When lambda is inlined, we can use return statements to return from the enclosing function. This is possible because the lambda body is substituted into the call site together with a body of the corresponding higher-order function, so the return statement would be treated as if it was placed directly in the body of `main()`:

```
inline fun forEach(a: IntArray, action: (Int) -> Unit) { ... }
fun main() {
    forEach(intArrayOf(1, 2, 3, 4)) {
        if (it < 2 || it > 3) return // Return from main
        println(it)
    }
}
```

There is a special case with calling inlinable lambda not directly in the body of a function it's passed to, but in a separate execution context like a local function or a method of the local class. Even though such lambdas are inlined, they are not able to force return of the caller function since even after inlining, they would occupy different frames of execution stack. For these reasons, such usages of functional parameters are forbidden by default:

```
private inline fun forEach(a: IntArray, action: (Int) ->
Unit) = object {
    fun run() {
        for (n in a) {
            action(n) // Error
        }
    }
}
```

To allow them, we need to mark a functional parameter with a `crossinline` modifier,

which leaves the functional value inlined but forbids using non-local returns inside a corresponding lambda:

```
private inline fun forEach(
    a: IntArray, crossinline action: (Int) -> Unit
) = object {
    fun run() {
        for (n in a) {
            action(n) // Ok
        }
    }
}

fun main() {
    forEach(intArrayOf(1, 2, 3, 4)) {
        if (it < 2 || it > 3) return // Error
        println(it)
    }
}
```

Non-local control flow issues may also arise when using `break` and `continue` statements since they can target a loop enclosing the lambda. Currently, they are not supported even if the lambda in question is inlined, although such support may be added in a future language version:

```
while (true) {
    forEach(intArrayOf(1, 2, 3, 4)) {
        if (it < 2 || it > 3) break // Error
        println(it)
    }
}
```

Extensions

The need to extend an existing class is quite common in practice: as a program evolves, a developer may want to add new functions and properties to classes, thus extending their API. But sometimes, simply adding new code to class is not an option since a class in question may be a part of some library and its modification will require significant efforts if feasible at all. Putting all possible methods into a single class may also be

impractical as not all of them are used together and therefore, worth decoupling into several program units.

In Java, such extra methods are often packed into utility classes. A common example is `java.util.Arrays` and `java.util.Collections` classes, which contain methods extending capabilities of Collection interfaces. The problem with such classes is that they often produce unnecessary boilerplate. For example, a typical usage of utility methods in Java may look like this:

```
int index = Collections.indexOfSubList(
    Arrays.asList("b", "c", "a"),
    Arrays.asList("a", "b")
)
```

Apart from clattering the source code, such calls do not allow you to make use of autocompletion available for class members in major IDEs such as IntelliJ and Eclipse.

That's the primary motivation behind Kotlin extensions that allow you to use functions and properties defined outside of a class as if they were its members. Supporting the open/closed design principle, they allow you to extend existing classes without modifying them.

Extension functions

Extension function is basically a function that can be called as if it were a member of some class. When you define such function, you put a type of its receiver before its name, separating them with a dot. Suppose we want to enrich the `String` type with a function that truncates the original string so that its length does not exceed given threshold. A possible definition can look like this:

```
fun String.truncate(maxLength: Int): String {
    return if (length <= maxLength) this else substring(0,
maxLength)
}
```

Once defined, this function can be used just like any member of the `String` class:

```
fun main() {
    println("Hello".truncate(10)) // Hello
    println("Hello".truncate(3))   // Hel
}
```

Note that inside of the extension function body, the receiver value can be accessed via this expression similar to class members. Members and extensions of receiver can also be accessed implicitly without this just like we've done with a `substring()` function call in the `truncate()` definition.

It's worth pointing out that the extension function by themselves are not able to break through the receiver type encapsulation. For example, since the extension function is defined outside of the class, it can't access its private members:

```
class Person(val name: String, private val age: Int)
fun Person.showInfo() = println("$name, $age") // Error:
can't access age
```

The extension function, however, may be declared inside a class body making it a member and extension at the same time. Such a function is allowed to access private members just like any other function in the class body:

```
class Person(val name: String, private val age: Int) {
    // Ok: age is accessible
    fun Person.showInfo() = println("$name, $age")
}
```

We'll see how to use such functions later in this chapter.

IDE Tips: The IntelliJ plugin can convert a class member to an extension. This can be achieved with the Convert member to extension action available in Alt + Enter menu when caret is positioned on the member name (the example is shown in *Figure 5.5*):

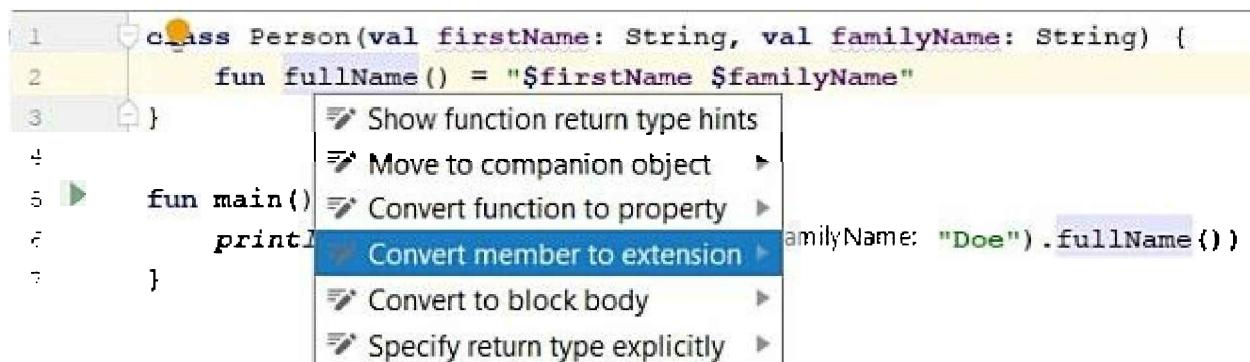


Figure 5.5: Converting member function to extension

Extension functions may be used in bound callable references similar to class members:

```
class Person(val name: String, val age: Int)
fun Person.hasName(name: String) = name.equals(this.name,
ignoreCase = true)
fun main() {
    val f = Person("John", 25)::hasName
    println(f("JOHN")) // true
    println(f("JAKE")) // false
}
```

What if you have function with the same signature defined both as a class member and as an extension? Consider the following code:

```
class Person(val firstName: String, val familyName: String) {
    fun fullName() = "$firstName $familyName"
}
fun Person.fullName() = "$familyName $firstName"
fun main() {
    println(Person("John", "Doe").fullName()) // ???
}
```

In this example, we have two `fullName()` functions defined on the `Person` class, which differ in whether they put `familyName` first or last. When faced with such ambiguity on the call site, the compiler always chooses the member function, so the preceding code will print `John Doe`. It will also issue a warning telling you that the extension function `fullName()` is shadowed by a member of the `Person` class and thus can't be called. IDE provides an appropriate highlighting as well (see *Figure 5.6*):

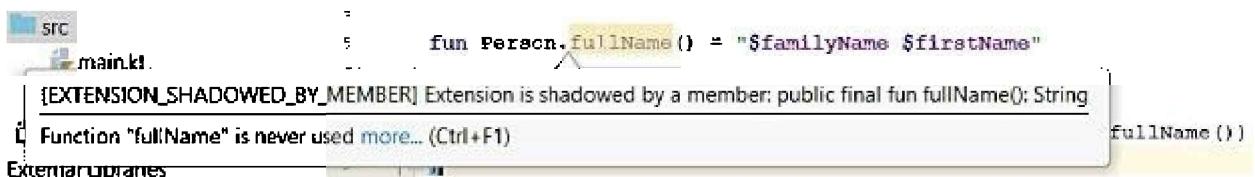


Figure 5.6: “Shadowed extension” warning

Favoring members over extensions prevents accidental modification of existing class behavior which otherwise could have led to hard-to-find errors. If it weren't the case, we could have, for example, defined:

```
package bad
fun Person.fullName() = "$familyName $firstName"
```

And then the meaning of the `Person("John", "Doe").fullName()` call would depend on whether the following statement is present in its containing file:

```
import bad.fullName
```

This also protects members of built-in and JDK classes.

Note that the extension shadowing has a flipside: if you define an extension function first, and then add corresponding member to the class, the original call will change its meaning. This is however, considered acceptable since class members comprising its primary API are supposed to change less frequently than its extension functions. This also simplifies interoperability with a Java code, which doesn't have extensions at all.

Extension functions may be local. In particular, they may be nested into other extension functions: in such cases, this expression means receiver of the innermost function. If you need to refer to the receiver of outer function instead, you may use a qualified form of this which specifies the function name explicitly. This is also true for members of local classes or anonymous objects declared inside an extension function body:

```
private fun String.truncator(max: Int) = object {
    val truncated
        get() = if (length <= max) this@truncator else
            substring(0, max)
    val original
        get() = this@truncator
}
fun main() {
    val truncator = "Hello".truncator(3)
    println(truncator.original)      // Hello
    println(truncator.truncated)   // Hel
}
```

The syntax is basically the same as we've seen in the case of inner classes.

When a top-level extension function is defined in another package, it must always be imported before you can make a call. For example:

```

// util.kt
package util
fun String.truncate(maxLength: Int): String {
    return if (length <= maxLength) this else substring(0,
maxLength)
}
// main.kt
package main
import util.truncate
fun main() {
    println("Hello".truncate(3))
}

```

The reason is that such a function can't be invoked by a qualified name since the qualifier position is taken by the receiver expression. Non-extension function, however:

```

fun truncate(s: String maxLength: Int): String {
    return if (s.length<= maxLength) s else s.substring(0,
maxLength)
}

```

could've been called as `util.truncate("Hello", 3)` without an import directive.

Java versus Kotlin: You might've been wondering how extension functions are represented on the Java Virtual Machine (JVM). The answer is actually quite simple: extension functions are compiled into methods with an additional parameter, which represent receiver expression. If we look at the bytecode generated for the preceding `truncate()` function, we'll see that it's basically equivalent to the following Java code:

```

public final class UtilKt {
    public static String truncate(String s, int maxLength) {
        return s.length() <= maxLength
            ? s
            : s.substring(0, maxLength)
    }
}

```

Which corresponds to the non-extension Kotlin function:

```
fun truncate(s: String, maxLength: Int) =  
    if (s.length <= maxLength) s else s.substring(0,  
maxLength)
```

In other words, extension functions are essentially a syntactic sugar over ordinary functions, which allows you to call them like a class members.

IDE Tips: The IntelliJ plugin includes an action which automatically converts the extension function to a non-extension one by changing its receiver to a parameter. To do this, you need to place caret at the receiver type, choose the **Convert receiver to parameter** from Alt + Enter menu, and enter a new parameter name (as shown in *Figure 5.7*). There is also an opposite action, **Convert parameter to receiver**, which transforms an arbitrary function parameter to its receiver. The latter action is available using Alt + Enter when caret is positioned on the parameter name.

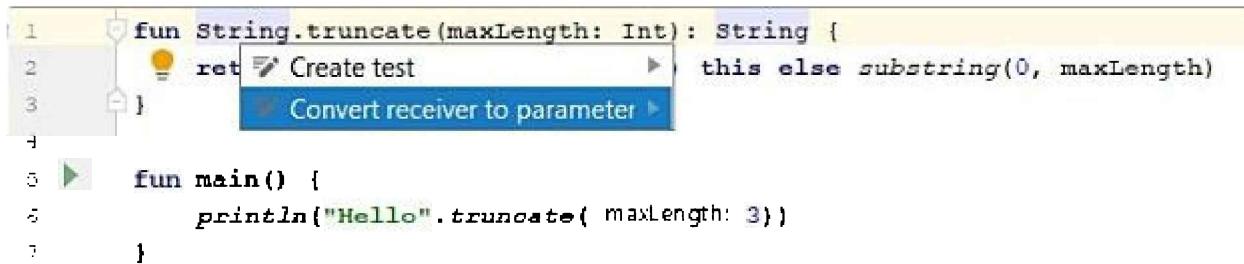


Figure 5.7: “Convert receiver to parameter” action

It's worth noting that extension, unlike member functions and properties, can be defined for a nullable receiver type. Since nullable types do not have their own members, this mechanism allows you to enrich them by introducing extension functions from outside. Such extensions can then be invoked without a safe call operator:

```
// Nullable receiver  
fun String?.truncate(maxLength: Int): String {  
    if (this == null) return null  
    return if (length <= maxLength) this else substring(0,  
maxLength)  
}
```

```
fun main() {
    val s = readLine()           // nullable String
    println(s.truncate(3)) // ?. is not necessary here
}
```

Note that if extension receiver has a nullable type, it's the responsibility of the extension function to handle a null value.

Extension properties

Similar to functions, Kotlin allows you to define extension properties, which can be accessed just like any member property. The syntax is also similar: to define an extension property, you prefix its name with a receiver type. Let's take a look at the following example:

```
val IntRange.leftHalf: IntRange
    get() = start..(start + endInclusive)/2
fun main() {
    println((1..3).leftHalf) // 1..2
    println((3..6).leftHalf) // 3..4
}
```

The preceding code defined an extension property `leftHalf` for the `IntRange` type: it computes the left half of the original range.

The crucial difference between a member and the extension property is that the latter can't have a backing field since there is no reliable way to add some extra state to a class instance. It means that extension properties can neither have initializers, nor use the `field` keyword inside their accessors. They also can't be `lateinit` since such properties rely on backing fields. For the same reason, an extension property must always have an explicit getter and, if mutable, an explicit setter:

```
val IntArray.midIndex
    get() = lastIndex/2
var IntArray.midValue
    get() = this[midIndex]
    set(value) {
        this[midIndex] = value
    }
```

```
fun main() {
    val numbers = IntArray(6) { it*it } // 0, 1, 4, 9, 16, 25
    println(numbers.midValue)         // 4
    numbers.midValue *= 10
    println(numbers.midValue)         // 40
}
```

Extension properties, however, can use delegates. Bear in mind, though, that the delegate expression can't access the property receiver; so, in general, there is no much point in declaring the `lazy` property as an extension since it would have the same value for each instance of receiver type as demonstrated by the following code:

```
valString.message by lazy { "Hello" }
fun main() {
    println("Hello".message) // Hello
    println("Bye".message)  // Hello
}
```

Object definitions can be considered an exception since they have only one instance:

`object Messages`

```
valMessages.HELLO by lazy { "Hello" }
fun main() {
    println(Messages.HELLO)
}
```

In general, it's possible to create a delegate, which is able to access the property receiver. We'll see how to do it in *Chapter 11, Domain-Specific Languages*.

Companion extensions

In *Chapter 4, Working with Classes and Objects*, we introduced the idea of companion object, which is a special nested object whose members can be accessed by the name of its containing class. This useful feature covers extensions as well.

In the following example, we will define an extension function for the companion object of the built-in `IntRange` class. The function can then be invoked via a class name:

```
fun IntRange.Companion.singletonRange(n: Int) = n..n
```

```
fun main() {
    println(IntRange.singletonRange(5)) // 5..5
    println(IntRange.Companion.singletonRange(3)) // 3..3
}
```

It is, of course, possible, to call such a function using a full companion name as well, such as `IntRange.Companion.singletonRange(3)`. The same idea also works for extension properties:

```
valString.Companion.HELLO
get() = "Hello"
fun main() {
    println(String.HELLO)
    println(String.Companion.HELLO)
}
```

Note that definition of extensions on the companion object is only possible if a class in question has explicit declaration of companion even if it's empty:

```
class Person(valfirstName: String, valfamilyName: String) {
    companion object
}
valPerson.Companion.UNKNOWN by lazy { Person("John", "Doe") }
```

We can't, on the other side, define an extension for the companion object of `Any` since it doesn't exist:

```
// Error: Companion is undefined
fun Any.Companion.sayHello() = println("Hello")
```

Lambdas and functional types with receiver

Similar to functions and properties, Kotlin allows you to utilize extension receivers for lambdas and anonymous functions. Such functional values are described by a special variety of functional types with receiver. Let's rewrite our `aggregate()` example to use a functional value with a receiver instead of a two-argument function:

```
fun aggregate(numbers: IntArray, op: Int.(Int) -> Int): Int {
    var result = numbers.firstOrNull()
    ?: throw IllegalArgumentException("Empty array")
    for (number in numbers) {
        result = op(result, number)
    }
    return result
}
```

```
    for (i in 1..numbers.lastIndex) result =
result.op(numbers[i])
    return result
}
fun sum(numbers: IntArray) = aggregate(numbers) { op -> this
+ op }
```

The receiver type is specified before the parameter type list and is separated with a dot:

```
Int.(Int) -> Int
```

In this case, any lambda passed as an argument gets an implicit receiver, which we can access using this expression:

```
{ op -> this + op }
```

Similarly, we can use an extension syntax for anonymous functions. The receiver type is specified just before the function's parameter list:

```
fun sum(numbers: IntArray) = aggregate(numbers, fun Int.(op:
Int) = this + op)
```

Unlike extension function definitions, a functional value with receiver can be called as a non-extension function with a receiver placed before all succeeding arguments. We could've written, for example:

```
fun aggregate(numbers: IntArray, op: Int.(Int) -> Int): Int {
    var result = numbers.firstOrNull()
    ?: throw IllegalArgumentException("Empty array")
    for (i in 1..numbers.lastIndex) {
        result = op(result, numbers[i]) // Non-extension call
    }
    return result
}
```

Basically, non-literal values of a functional type with receiver are freely interchangeable with values of the corresponding type where receiver is used as the first parameter as if they have the same type. This is possible because such values have essentially the same runtime representation:

```
val min1: Int.(Int) -> Int = { if (this < it) this else it }
val min2: (Int, Int) -> Int = min1
val min3: Int.(Int) -> Int = min2
```

Note, however, that while it's possible to invoke functional value with receiver as either extension or non-extension (with a receiver placed as the first argument), functional values without receivers can be invoked using non-extension syntax only:

```
fun main() {
    val min1: Int.(Int) -> Int = { if (this < it) this else
it }
    val min2: (Int, Int) -> Int = min1
    println(3.min1(2))      // Ok: calling min1 as extension
    println(min1(1, 2))   // Ok: calling min1 as non-extension
    println(3.min2(2))      // Error: Can't call min2 as
extension
    println(min2(1, 2)) // Ok: Calling min2 as non-extension
}
```

Lambdas with a receiver give you a powerful tool, which can be used for building DSL-like API. We'll address this issue in *Chapter 11, Domain-Specific Languages*.

Callable references with receiver

In Kotlin, you can also use callable references that define functional values with receivers. Such references may be based on either class member or extension declarations. Syntactically, they are similar to bound callable references, but qualified by a receiver type instead of an expression:

```
fun aggregate(numbers: IntArray, op: Int.(Int) -> Int): Int {
    var result = numbers.firstOrNull()
    ?: throw IllegalArgumentException("Empty array")
    for (i in 1..numbers.lastIndex) result =
result.op(numbers[i])
    return result
}
fun Int.max(other: Int) = if (this > other) this else other
fun main() {
    val numbers = intArrayOf(1, 2, 3, 4)
```

```

    println(aggregate(numbers, Int::plus)) // 10
    println(aggregate(numbers, Int::max))    // 4
}

```

In the preceding code, `Int::plus` refers to the member function `plus()` (which does exactly the same as the `+` operator) of the built-in class `Int`, while `Int::max` refers to the extension function defined in the containing file. The syntax is the same in both cases.

Thanks to implicit casting between extension and non-extension functional types we've mentioned in the previous section, it's also possible to use non-receiver callable references in the context where a functional type with a receiver is expected. For example, we could've passed a two-argument callable reference `::max` for a parameter of the type `Int.(Int) -> Int`:

```

fun aggregate(numbers: IntArray, op: Int.(Int) -> Int): Int {
    var result = numbers.firstOrNull()
    ?: throw IllegalArgumentException("Empty array")
    for (i in 1..numbers.lastIndex) result =
        result.op(numbers[i])
    return result
}
fun max(a: Int, b: Int) = if (a > b) a else b
fun main() {
    println(aggregate(intArrayOf(1, 2, 3, 4), ::max))
}

```

The converse is true as well: callable references with a receiver can be used when the expected functional type is a non-receiver one. In a slightly modified example, callable references to member and extension functions are used as values of a two-argument functional type `(Int, Int) -> Int`:

```

fun aggregate(numbers: IntArray, op: (Int, Int) -> Int): Int
{
    var result = numbers.firstOrNull()
    ?: throw IllegalArgumentException("Empty array")
    for (i in 1..numbers.lastIndex) result = op(result,
        numbers[i])
    return result
}

```

```
}

fun Int.max(other: Int) = if (this > other) this else other
fun main() {
    println(aggregate(intArrayOf(1, 2, 3, 4), Int::plus)) // 10
    println(aggregate(intArrayOf(1, 2, 3, 4),
Int::max))      // 4
}
```

Note that callable references are not supported for extension functions declared as class members as currently there is no way to specify multiple receiver types for a :: expression.

Scope functions

The Kotlin standard library includes a set of functions that allow you to introduce a temporary scope where you can refer to the value of a given context expression. Sometimes, this can be helpful to avoid an explicit introduction of local variables in the containing scope to hold an expression value and simplify the code. These functions are usually called scope functions.

Their basic effect is a simple execution of a lambda you provide as an argument. The difference comes from the combination of the following aspects:

- Whether the context expression is passed as a receiver or an ordinary argument;
- Whether the lambda is extension or not;
- Whether the function returns the value of lambda or the value of the context expression.

Overall, there are five standard scope functions: `run`, `let`, `with`, `apply`, `also`. In this section, we'll discuss how to use them to simplify your code. All scope functions are inline and thus, do not entail any performance overhead.

Note that scope functions should be used with care as abusing them can make your code less readable and more prone to errors. In general, it's worth avoiding the nested scope functions as you might easily get confused about the meaning of `this` or `it`.

run / with functions

The `run()` function is an extension that accepts an extension lambda and returns its

result. The basic use pattern is a configuration of an object state followed by a computation of a result value:

```
class Address {  
    var zipCode: Int = 0  
    var city: String = ""  
    var street: String = ""  
    var house: String = ""  
    fun post(message: String): Boolean {  
        "Message for ${zipCode}, $city, $street, $house":  
        $message"  
        return readLine() == "OK"  
    }  
}  
fun main() {  
    val isReceived = Address().run {  
        // Address instance is available as this  
        zipCode = 123456  
        city = "London"  
        street = "Baker Street"  
        house = "221b"  
        post("Hello!") // return value  
    }  
    if (!isReceived) {  
        println("Message is not delivered")  
    }  
}
```

Without a `run` function, we'd have to introduce a variable for the `Address` instance; thus, making it available for the rest of the function body, which may be undesirable if we need that instance for a single `post()` action. Using functions such as `run()` gives you more fine-grained control over visibility of local declarations.

Note that the result may also be of the type `Unit`:

```
fun Address.showCityAddress() = println("$street, $house")  
fun main() {  
    Address().run {
```

```

    zipCode = 123456
    city = "London"
    street = "Baker Street"
    house = "221b"
    showCityAddress()
}
}

```

The `with()` function is quite similar to `run()` : the only difference is that `with()` is not an extension, so the context expression is passed as an ordinary argument rather than a receiver. The common use of this function is a grouping of calls to member functions and properties of the context expression under the same scope:

```

fun main() {
    val message = with (Address("London", "Baker Street",
"221b")) {
        "Address: $city, $street, $house"
    }
    println(message)
}

```

In the preceding example, we'll make use of the fact that the members of the `this` instance can be accessed without a qualifier. Without the scope function, we'd have to write the following code:

```

fun main() {
    val addr = Address("London", "Baker Street", "221b")
    val message = "Address: ${addr.city}, ${addr.street},
${addr.house}"
    println(message)
}

```

In other words we'd have to introduce an additional variable and explicitly qualify all members of `Address` with a particular instance `addr`.

run without context

The Kotlin standard library also provides an overloaded version of `run()`, which doesn't have a context expression and just returns the value of lambda. The lambda

itself has neither receiver nor parameters.

The primary use case for this function is using a block in some context which requires an expression. Consider, for example, the following code:

```
class Address(val city: String, val street: String, val
house: String) {
    fun asText() = "$city, $street, $house"
}
fun main() {
    val address = Address("London", "Baker Street", "221b")
    println(address.asText())
}
```

What if we want to read address components from the standard input? We could've introduced a separate variable for each of them:

```
fun main() {
    val city = readLine() ?: return
    val street = readLine() ?: return
    val house = readLine() ?: return
    val address = Address(city, street, house)
    println(address.asText())
}
```

But that would place them in the same scope as any other local variable of `main()`, while variables such as `city` only make sense in the context of creating a particular `Address` instance. Inlining all the variables and getting something like the following is a rather bad choice leading to hard-to-read code:

```
fun main() {
    val address = Address(readLine() ?: return,
    readLine() ?: return,
    readLine() ?: return)
    println(address.asText())
}
```

Looking at such code, we can't immediately tell what each `readLine()` is supposed to mean. The idiomatic solution is given by `run()`:

```
fun main() {
    val address = run {
        val city = readLine() ?: return
        val street = readLine() ?: return
        val house = readLine() ?: return
        Address(city, street, house)
    }
    println(address.asText())
}
```

Since `run` is an inline function, we can use `return` statements inside its lambda to exit the outer function as if it's some built-in control structure.

Note that using a block statement by itself doesn't work since such a block is treated as lambda. That's the reason the `run()` function is added to the standard library:

```
fun main() {
    val address = {
        val city = readLine() ?: return      // Error: return
        is not allowed
        val street = readLine() ?: return // Error: return is
        not allowed
        val house = readLine() ?: return      // Error: return
        is not allowed
        Address(city, street, house)
    }
    println(address.asText()) // Error: no asText() method
}
let
```

The `let` function is similar to `run` but accepts single-argument lambda instead of an extension one. The value of context expression is thus represented by the lambda argument. The return value of `let` is the same as that of its lambda. This function is often used to avoid introduction of a new variable in the outer scope:

```
class Address(val city: String, val street: String, val
house: String) {
    fun post(message: String) {}
}
```

```
fun main() {
    Address("London", "Baker Street", "221b").let {
        // Address instance is accessible via it parameter
        println("To city: ${it.city}")
        it.post("Hello")
    }
}
```

Similar to other lambdas, you can introduce a custom parameter name for the purpose of readability or disambiguation:

```
fun main() {
    Address("London", "Baker Street", "221b").let { addr ->
        // Address instance is accessible via addr parameter
        println("To city: ${addr.city}")
        addr.post("Hello")
    }
}
```

A common use case of `let` is a concise way to pass a nullable value to a non-nullable function with a safety check. In the previous chapter, we've learned about a safe call operator, which allows you to invoke a function with a nullable receiver. But what if the value in question must be passed as an ordinary parameter. Consider the following example:

```
fun readInt() = try {
    readLine()?.toInt()
} catch (e: NumberFormatException) {
    null
}
fun main(args: Array<String>) {
    val index = readInt()
    val arg = if (index != null) args.getOrNull(index) else
    null
    if (arg != null) {
        println(arg)
    }
}
```

The `getOrNull()` function returns array item if the given index is valid and null otherwise. Since its parameter is non-nullable, we can't pass a result of the `readInt()` function to `getOrNull()`: that's why we need the if statement above to enable the smart cast to non-nullable type. We can, however, simplify the code by using `let`:

```
val arg = index?.let { args.getOrNull(it) }
```

The `let` call is only executed when `index` is not null, so the compiler knows that the `it` parameter is non-nullable inside the lambda.

apply / also functions

The `apply()` function is an extension which takes an extension lambda and returns the value of its receiver. A common use of this function is a configuration of the object state which, as opposed to `run()`, is not followed by an immediate computation of some result value:

```
class Address {  
    var city: String = ""  
    var street: String = ""  
    var house: String = ""  
    fun post(message: String) {}  
}  
fun main() {  
    val message = readLine() ?: return  
    Address().apply {  
        city = "London"  
        street = "Baker Street"  
        house = "221b"  
    }.post(message)  
}
```

There is also a similar function `also()`, which takes a single-argument lambda instead:

```
fun main() {  
    val message = readLine() ?: return  
    Address().also {
```

```

    it.city = "London"
    it.street = "Baker Street"
    it.house = "221b"
} .post(message)
}

```

Extensions as class members

In a previous section, we've discussed a possibility of declaring an extension function as a class member. Let's now take a closer look at such extensions.

When you define an extension function or property inside a class, such definition automatically gets two receivers as opposed to a single one for ordinary members and top-level extensions. The instance of the receiver type mentioned in extension definition is called the extension receiver, while the instance of the class containing the extension is called dispatch receiver. Both receivers can be denoted by this expression qualified with either the containing class name (for dispatch receiver), or the name of extension (for the extension receiver). Unqualified this expression, as usually, refer to the receiver of the nearest enclosing declaration. So usually, it's the same as an extension receiver unless you use it inside some local declaration such as class, nested extension function, or a lambda with a receiver.

Let's consider an example which illustrates both kinds of receivers:

```

class Address(val city: String, val street: String, val
house: String)
class Person(val firstName: String, val familyName: String) {
    fun Address.post(message: String) {
        // implicit this: extension receiver (Address)
        val city = city
        // unqualified this: extension receiver (Address)
        val street = this.city
        // qualified this: extension receiver (Address)
        val house = this@post.house
        // implicit this: dispatch receiver (Person)
        val firstName = firstName
        // qualified this: dispatch receiver (Person)
        val familyName = this@Person.familyName
        println("From $firstName, $familyName at $city,

```

```

$street, $house:")
    println(message)
}
fun test(address: Address) {
    // Dispatch receiver: implicit
    // Extension receiver: explicit
    address.post("Hello")
}
}

```

When we invoke the `post()` function inside `test()`, the dispatch receiver is supplied automatically since `test()` is a member of the `Person` class. The extension receiver, on the other hand, is passed explicitly as an `address` expression.

Similarly, we can call the `post()` function when the current instance of the `Person` class is supplied in a different way: for example, as an extension receiver or an instance of an outer class:

```

class Address(val city: String, val street: String, val
house: String)
class Person(val firstName: String, val familyName: String) {
    fun Address.post(message: String) { }
    inner class Mailbox {
        fun Person.testExt(address: Address) {
            address.post("Hello")
        }
    }
}
fun Person.testExt(address: Address) {
    address.post("Hello")
}

```

What if we have a receiver of the `Address` type instead? Suppose we want to call the `post()` inside the `Address` class body:

```

class Address(val city: String, val street: String, val
house: String) {
    fun test(person: Person) {
        person.post("Hello") // Error: method post() is not
}

```

```
defined
    }
}
class Person(val firstName: String, val familyName: String) {
    fun Address.post(message: String) { }
}
```

This doesn't work because dispatch receiver of the type `Person` must be already in scope. The problem can be solved using one of the scope functions that can wrap the `post()` call inside an extension lambda with a `Person` receiver:

```
class Address(val city: String, val street: String, val
house: String) {
    fun test(person: Person) {
        with(person) {
            // Implicit dispatch and extension receivers
            post()
        }
    }
}
class Person(val firstName: String, val familyName: String) {
    fun Address.post(message: String) { }
}
```

This trick can also be used to call `post()` outside of `Address` or `Person` class as well as their extensions:

```
class Address(val city: String, val street: String, val
house: String)
class Person(val firstName: String, val familyName: String) {
    fun Address.post(message: String) { }
}
fun main() {
    with(Person("John", "Watson")) {
        Address("London", "Baker Street",
"221b").post("Hello")
    }
}
```

These examples demonstrate that rules regarding functions and properties with double receivers can become quite confusing. For this reason, it's generally recommended to restrict their scope to containing declaration:

```
class Address(val city: String, val street: String, val
house: String)
class Person(val firstName: String, val familyName: String) {
    // Can't be used outside Person class
    private fun Address.post(message: String) { }
    fun test(address: Address) = address.post("Hello")
}
```

One particularly confusing and error-prone case that's worth avoiding is when both dispatch and extension receivers have the same type, as shown in the following example:

```
class Address(val city: String, val street: String, val
house: String) {
    fun Address.post(message: String) { }
}
```

An interesting example of the double-receiver member is an extension declared inside an object (in particular, a companion object). Such extensions may be imported and used similar to top-level ones:

```
import Person.Companion.parsePerson
class Person(val firstName: String, val familyName: String) {
    companion object {
        fun String.parsePerson(): Person? {
            val names = split(" ")
            return if (names.size == 2) Person(names[0],
names[1]) else null
        }
    }
    fun main() {
        // instance of Person.Companion is supplied implicitly
        println("John Doe".parsePerson()?.firstName) // John
    }
}
```

In most cases, though, using top-level extensions is more preferable since it leads to a more simple and readable code.

Conclusion

Let's sum up the basic things we have picked up in this chapter. We've learned to use functional types and higher-order functions to abstract and compose pieces of code in a form of functions. We've also seen various forms of constructing a functional value and discussed the capabilities of function inlining. Finally, we went through the major use cases of extension functions and properties that allow you to add new features to existing types.

In the next chapter, we'll revisit object-oriented programming and discuss special varieties of classes aimed at simplifying common programming patterns such as enumerations and data classes.

Questions

1. What's a higher-order function?
2. Describe the syntax of lambda expression. How do they compare with lambdas in Java?
3. What is a functional type? What's the difference between functional types in Kotlin and functional interfaces in Java?
4. Compare functional types with a receiver and the ones without.
5. What are the differences between lambdas and anonymous functions? When would you prefer an anonymous function over lambda?
6. Describe the pros and cons of inline functions. What are the limitations?
7. What is a callable reference? Describe callable reference forms. How do they compare with method references in Java?
8. Describe the behavior of return statements inside lambdas and anonymous functions. What is a qualified return statement?
9. Compare inlining modes of functional parameters: `default`, `noinline`, `crossinline`.
10. How to define an extension function? Do extensions modify the classes they apply to?

11. How would you use a companion object extension?
12. What are limitations of extension properties?
13. Describe the forms of `this` expression. What are the uses of qualified `this`?
14. What are the specifics of declaring extension functions inside classes?
15. What are the scope functions? How would you decide which scope function fits better for the particular task at hand?

CHAPTER 6

Using Special-Case Classes

In this chapter, we will discuss special kinds of classes designed to simplify implementation of some common programming patterns. Namely, we'll address the usage of enums to describe types with a restricted set of instances, the concise representation of data with data classes, and experimental lightweight wrappers with almost zero runtime overhead.

Structure

- Enum classes
- Data classes
- Inline classes

Objective

Learn to use special varieties of classes such as enums and data classes to solve common programming tasks. Get a basic understanding of inline classes and their usage on the example of unsigned integer types.

Enum classes

An enum (short of enumeration) class is a special variety of class which can represent a limited set of predefined constants. The simplest form is just a list of constant names enclosed inside the enum class body:

```
enum class WeekDay {  
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,  
    SUNDAY  
}  
fun WeekDay.isWorkDay() =  
    this == WeekDay.SATURDAY || this == WeekDay.SUNDAY  
fun main() {
```

```
    println(WeekDay.MONDAY.isWorkDay())      // false
    println(WeekDay.SATURDAY.isWorkDay()) // true
}
```

Enums allow more type-safe representation of limited value sets as compared to, say, integers or strings since you don't have to check whether a value in question is out of possible range. The compiler ensures that any variable of a particular enum type can take only one of the values specified in its body.

Java versus Kotlin: Kotlin enums are defined with a pair of the keyword `enum class` as opposed to just `enum` in Java. The `enum` keyword itself is soft and can be used as an identifier in any other context.

Note that being compile-time constants enum values are usually written in upper case. Enums are somewhat similar to object declarations, in a sense that they define a set of global constants representing instances of a particular type. Similar to objects, they are not permitted in contexts where there is no guarantee that such definition can be available as a global constant. You can't, for example, put an enum definition into an inner class or a function body:

```
fun main() {
    enum class Direction { NORTH, SOUTH, WEST, EAST } //
Error
}
```

Exhaustive when expressions

Just like values of any other type, enum variables may be compared against particular values using a `when` expression. There is, however, an additional benefit when using enums: you can omit an `else` branch if the `when` expression is exhaustive, that is, contains branches for all possible values of an enum type:

```
enum class Direction {
    NORTH, SOUTH, WEST, EAST
}
fun rotateClockWise(direction: Direction) = when (direction)
{
    Direction.NORTH -> Direction.EAST
    Direction.EAST -> Direction.SOUTH
}
```

```
    Direction.SOUTH -> Direction.WEST
    Direction.WEST -> Direction.NORTH
}
```

An exhaustive form of `when` expression decreases a chance of writing a code that may break on a context change like adding a new enum value. Suppose that we've added an `else` branch instead:

```
fun rotateClockWise(direction: Direction) = when (direction)
{
    Direction.NORTH -> Direction.EAST
    Direction.EAST -> Direction.SOUTH
    Direction.SOUTH -> Direction.WEST
    Direction.WEST -> Direction.NORTH
    else ->
    throw IllegalArgumentException("Invalid direction: $direction")
}
```

This code works fine until we add new values for the `Direction` enum:

```
enum class Direction {
    NORTH, SOUTH, WEST, EAST,
    NORTH_EAST, NORTH_WEST, SOUTH_EAST, SOUTH_WEST
}
```

Now, a call like `rotateClockWise(Direction.NORTH_EAST)` will throw an exception. If we, however, use an `else-free` form, an error can be caught at compile-time as the compiler will complain about non-exhaustive `when` expression in the `rotateClockWise()` body.

Java versus Kotlin: Note that unlike Java's `switch` statement which requires us to use unqualified names of enum values in `case` clauses, Kotlin enum constants used in a `when` expression must be qualified with a name of the enum class unless imported. Compare the preceding `rotateClockWise()` function with a similar Java method:

```
public Direction rotateClockWise(Direction d) {
    switch (d) {
        case NORTH: return Direction.EAST;
        case EAST: return Direction.SOUTH;
```

```

        case SOUTH: return Direction.WEST;
        case WEST: return Direction.NORTH;
    }
    throw new IllegalArgumentException("Unknown value: " +
d);
}

```

We can void explicit qualification of enum constants by importing them at the beginning of a containing file:

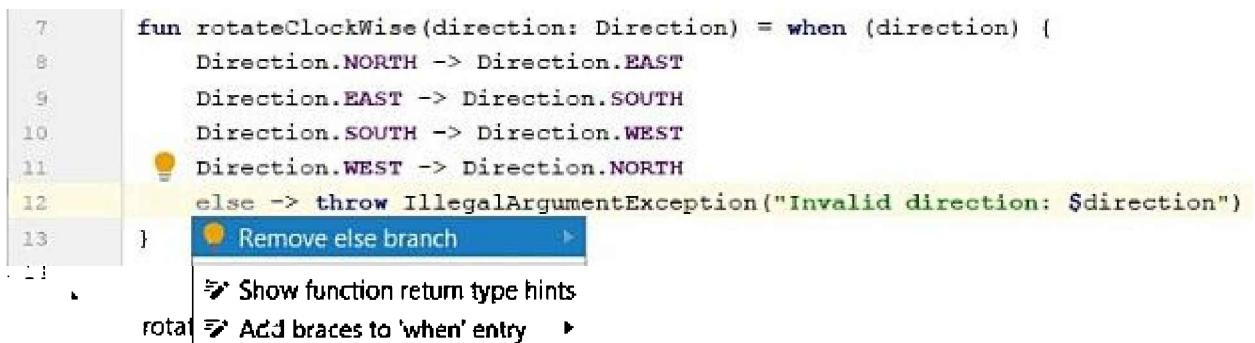
```

import Direction.*
enum class Direction {
    NORTH, SOUTH, WEST, EAST
}
fun rotateClockWise(direction: Direction) = when (direction)
{
    NORTH -> EAST
    EAST -> SOUTH
    SOUTH -> WEST
    WEST -> NORTH
}

```

Internally exhaustive when expressions include an implicit else branch that throws special exception of the NoWhenBranchMatchedException class when no branch matches a subject expression.

IDE Tips: The IntelliJ plugin can detect unnecessary else branches and suggest to drop them if the when expression is exhaustive (as shown on the Figure 6.1):



```

7     fun rotateClockWise(direction: Direction) = when (direction) {
8         Direction.NORTH -> Direction.EAST
9         Direction.EAST -> Direction.SOUTH
10        Direction.SOUTH -> Direction.WEST
11        Direction.WEST -> Direction.NORTH
12        else -> throw IllegalArgumentException("Invalid direction: $direction")
13    }

```

The screenshot shows the IntelliJ IDE interface with code completion suggestions. The code defines a `rotateClockWise` function using a `when` expression. The `else` branch of the `when` expression is highlighted with a yellow background. A context menu is open over this `else` branch, with the option "Remove else branch" highlighted in blue. Other options in the menu include "Show function return type hints" and "Add braces to 'when' entry".

Figure 6.1: Redundant else branch in an exhaustive when expression

Declaring enums with custom members

Similar to other classes, enums may have their own members. Besides this, you can define your own extension functions and properties as evidenced by a preceding example.

The enum class may include any definition permitted for an ordinary class, including functions, properties, primary and secondary constructors, initialization blocks, inner/non-inner nested classes, and objects (whether companion or not). Any such declaration in an enum class body must be placed after the enum constant list. The constant list itself, in this case, must be terminated by a semicolon (it's one of those rare cases when a semicolon in Kotlin can't be omitted). The members declared in an enum class body are available for all its constants:

```
enum class WeekDay {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,
    SUNDAY;
    val lowercaseName get() = name.toLowerCase()
    fun isWorkDay() = this == SATURDAY || this == SUNDAY
}
fun main() {
    println(WeekDay.MONDAY.isWorkDay()) // false
    println(WeekDay.WEDNESDAY.lowercaseName) // wednesday
}
```

When an enum class has a constructor, you have to place an appropriate call in the definition of each enum constant:

```
enum class RainbowColor(val isCold: Boolean) {
    RED(false), ORANGE(false), YELLOW(false),
    GREEN(true), BLUE(true), INDIGO(true), VIOLET(true);
    val isWarm get() = !isCold
}
fun main() {
    println(RainbowColor.BLUE.isCold) // true
    println(RainbowColor.RED.isWarm) // true
}
```

The enum constants may also have a body with their own definitions. Note, however, that an anonymous types introduced by such constants (we've already mentioned them in *Chapter 4, Working with Classes and Objects*) are not exposed to the outside code, which means that you can't access members introduced in the enum constant body outside of the body itself. The following code demonstrates this idea:

```
enum class WeekDay {  
    MONDAY { fun startWork() = println("Work week started")  
    },  
    TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY  
}  
fun main() = WeekDay.MONDAY.startWork() // Error
```

Such members are generally helpful when used to provide implementation of virtual methods in the enum class itself or some supertype. We'll defer such examples till *Chapter 8, Understanding Class Hierarchies*.

Note that currently all nested classes defined in an enum constant body must be inner.

Using common members of enum classes

All enum classes in Kotlin are implicit subtypes of the `kotlin.Enum` class, which contains a set of common functions and properties available for any enum value. Besides a few API difference, this class is quite similar to its Java counterpart, `java.lang.Enum`. On the JVM, it's indeed represented by the Java's `Enum`.

Any enum value has a pair of properties, `ordinal`, and `name`, which contain zero-based index of its definition in the enum class body and value name, respectively:

```
enum class Direction {  
    NORTH, SOUTH, WEST, EAST;  
}  
fun main() {  
    println(Direction.WEST.name)          // WEST  
    println(Direction.WEST.ordinal) // 2  
}
```

Values of a particular enum class are comparable with each other according to the order of their definition in the enum body. Similar to Java, the enum equality is based on their identity:

```

fun main() {
    println(Direction.WEST == Direction.NORTH)           // false
    println(Direction.WEST != Direction.EAST)            // true
    println(Direction.EAST < Direction.NORTH)           // false
    println(Direction.SOUTH >= Direction.NORTH)         // true
}

```

The comparison operations on enum values basically work on their indices as given by the `ordinal` property.

Java versus Kotlin: Even though both Java and Kotlin enums implicitly implement the `Comparable` interface, you can't apply an operator like `<` or `>` to enum values in Java.

Each enum class also has a set of implicit methods which can be invoked on a class name similarly to members of a companion object. The `valueOf()` method returns a enum value given its name or throws an exception if a name is not valid:

```

fun main() {
    println(Direction.valueOf("NORTH"))                  // NORTH
    println(Direction.valueOf("NORTH_EAST")) // Exception:
    Invalid name
}

```

The `values()` method gives you an array of all enum values in the order of their definition. Note that the array is created anew on each call, so changes to one of them does not affect others:

```

enum class WeekDay {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY,
    SUNDAY
}
private val weekDays = WeekDay.values()
val WeekDay.nextDay get() = weekDays[(ordinal + 1) % weekDays.size]

```

Since Kotlin 1.1, you can use generic top-level functions `enumValues()` and `enumValueOf()` instead of `values()` and `valueOf()` methods, respectively:

```
fun main() {
```

```

    val weekDays = enumValues<WeekDay>()
    println(weekDays[2])                                // WEDNESDAY
    println(enumValueOf<WeekDay>("THURSDAY")) // THURSDAY
}

```

Data classes

Kotlin provides a useful feature to declare classes with a primary goal of storing some data. This feature, called data classes, allows you to use automatically generated implementations of some basic operations like equality or conversion to `String`. With data classes, you can also take advantage of destructuring declarations, which give you an option to extract class properties putting them into separate local variables with a single and concise language construct. In this section, we'll consider the possibilities of data classes.

Data classes and their operations

Consider, for example, the following class:

```

class Person(val firstName: String,
            val familyName: String,
            val age: Int)

```

What if we want to compare its instances by equality? Similar to Java, the value of referential types are by default considered equal if they have the same identity, that is, refer to the same object. The values of instance fields are not taken into account:

```

fun main() {
    val person1 = Person("John", "Doe", 25)
    val person2 = Person("John", "Doe", 25)
    val person3 = person1
    println(person1 == person2) // false, different
identities
    println(person1 == person3) // true, the same identity
}

```

If we need a custom equality for our class, we usually implement it with the `equals()` method (more on this in *Chapters 7, Exploring Collections and I/O*, and *Chapter 8, Understanding Class Hierarchies*) accompanied with corresponding `hashCode()`

method, which allows using class instances as keys in collections such as `HashMap`. For a certain variety of classes, called data classes, Kotlin can generate these methods automatically based on a list of class properties. Let's modify our example slightly:

```
data class Person(val firstName: String,  
                 val familyName: String,  
                 val age: Int)  
  
fun main() {  
    val person1 = Person("John", "Doe", 25)  
    val person2 = Person("John", "Doe", 25)  
    val person3 = person1  
    println(person1 == person2) // true  
    println(person1 == person3) // true  
}
```

Now, both comparisons yield `true` since the compiler automatically provides implementation of equality operation, which compares values of the properties declared in the primary constructor. This also applies to the hash code which depends on the same set of properties.

Note that comparisons of property values are based on their `equals()` method too, so how deep the equality goes depends on the type of the properties involved. Consider the following example:

```
data class Person(val firstName: String,  
                 val familyName: String,  
                 val age: Int)  
  
data class Mailbox(val address: String, val person: Person)  
  
fun main() {  
    val box1 = Mailbox("Unknown", Person("John", "Doe", 25))  
    val box2 = Mailbox("Unknown", Person("John", "Doe", 25))  
    println(box1 == box2) // true  
}
```

Since `String`, `Person`, and `MailBox` implement content-based equality, the comparison of `MailBox` instances depend on its own `address` property as well properties of the corresponding `Person` instance. If we, however, drop the `data` modifier before the `Person` class, the result will change since `Person` properties will be compared by their identity:

```

class Person(val firstName: String,
            val familyName: String,
            val age: Int)
data class Mailbox(val address: String, val person: Person)
fun main() {
    val box1 = Mailbox("Unknown", Person("John", "Doe", 25))
    val box2 = Mailbox("Unknown", Person("John", "Doe", 25))
    // false: Person instances have different identities
    println(box1 == box2)
}

```

The `hashCode()` method similarly returns an object hash code which depends on hash codes of all properties declared in the primary constructor.

Besides `equals()/hashCode()` generation, the data classes provide an implementation of the `toString()` method, which converts a class instance to a string:

```

fun main() {
    val person = Person("John", "Doe", 25)
    println(person) // Person(firstName=John, familyName=Doe, age=25)
}

```

Note that only properties declared as parameters of primary constructor are used in the equality/hash code/String conversion. Any other properties do not affect the result:

```

data class Person(val firstName: String, val familyName: String) {
    var age = 0
}
fun main() {
    val person1 = Person("John", "Doe").apply { age = 25 }
    val person2 = Person("John", "Doe").apply { age = 26 }
    println(person1 == person2) // true
}

```

Any data class implicitly provides the `copy()` function which allows you to create a

copy of the current instance with a possible change of some properties. It has the same signature as the data class primary constructor, but accompanies each parameter with a default equal to the current value of the corresponding property. The `copy()` function is usually invoked with a named argument syntax for better code readability:

```
fun Person.show() = println("$firstName $familyName: $age")
fun main() {
    val person = Person("John", "Doe", 25)
    person.show()                                // John Doe:
25
    person.copy().show()                         // John
Doe: 25
    person.copy(familyName = "Smith").show()      // John
Smith: 25
    person.copy(age = 30, firstName = "Jane").show() // Jane
Doe: 30
}
```

The ability to easily copy an instance encourages the usage of immutable data classes. Although `var` properties are allowed, it's often reasonable to design data classes as immutable. Using immutable data simplifies reasoning about your code and makes it less error-prone, especially in a multi-threaded projects. Besides this, immutability is a prerequisite for a proper usage of object as a map key: violating immutability in such cases may lead to quite unexpected behavior as we'll see in *Chapter 7, Exploring Collections and I/O*.

The Kotlin standard library includes two general-purpose data classes, which can be used to hold a pair or a triplet of values:

```
fun main() {
    val pair = Pair(1, "two")
    println(pair.first + 1)           // 2
    println("${pair.second}!") // two!
    val triple = Triple("one", 2, false)
    println("${triple.first}!") // one!
    println(triple.second - 1)       // 1
    println(!triple.third)          // true
}
```

Pairs can also be constructed using an infix operation to:

```
val pair = 1 to "two"  
println(pair.first + 1)           // 2  
println("${pair.second}!") // two!
```

Note that in most cases using custom data classes is more reasonable since they allow you to choose meaningful names for both a class and its properties, thus improving the code readability.

Apart from the autogenerated functions we've just seen, data classes give you a useful ability to extract their constituent properties into separate variables within a single definition. In the following section, we'll consider how to do it using destructuring declarations.

Destructuring declarations

Consider the following example:

```
import kotlin.random.Random  
  
data class Person(val firstName: String,  
                  val familyName: String,  
                  val age: Int)  
  
fun newPerson() = Person(readLine()!!,  
                        readLine()!!,  
                        Random.nextInt(100))  
  
fun main() {  
    val person = newPerson()  
    val firstName = person.firstName  
    val familyName = person.familyName  
    val age = person.age  
    if (age < 18) {  
        println("$firstName $familyName is under-age")  
    }  
}
```

We're extracting values of Person properties and using them in a subsequent computation. But since Person is a data class, we can use much more concise syntax to define corresponding local variables:

```
val (firstName, familyName, age) = person
```

This is a destructuring declaration that generalizes a local variable syntax by allowing you to use a parentheses-enclosed list of identifiers instead of a single variable name. Each name corresponds to a separate variable definition, which is initialized by a corresponding property from a data class instance written after the = sign.

Note that properties are mapped to the variables according to their position in the data class constructor rather than their names. So, while the code:

```
val (firstName, familyName, age) = Person("John", "Doe", 25)
println("$firstName $familyName: $age")
```

produces the expected result John Doe: 25, the following lines:

```
val (familyName, firstName, age) = Person("John", "Doe", 25)
println("$firstName $familyName: $age")
```

will give you Doe John: 25.

IDE Tips: For this specific case, when variable names in destructureing declaration match data class properties but are written in a wrong order, the IntelliJ plugin reports a warning, which may help to locate a source of possible bug. It's recommended to either rename a variable so that it does match the property or change its position in destructureing declaration (see an example in *Figure 6.2*):



Figure 6.2: Wrong order of variables in a destructureing declaration

Destructuring declaration as a whole may not have a type. It's possible, however, to specify explicit types for a component variable whenever it's necessary:

```
val (firstName, familyName: String, age) = Person("John",
"Doe", 25)
```

Destructuring declaration may include fewer components than there are properties in

a data class. In this case, missing properties at the end of constructor are not extracted:

```
val (firstName, familyName) = Person("John", "Doe", 25)
println("$firstName $familyName") // John Doe
val (name) = Person("John", "Doe", 25)
println(name) // John
```

What if you need to skip some properties that come at the beginning or in the middle? Since Kotlin 1.1, you can replace unused components with the `_` symbol similar to unused parameters of a lambda:

```
val (_, familyName) = Person("John", "Doe", 25)
println(familyName) // Doe
```

By replacing `val` with `var` you will get a set of mutable variables:

```
var (firstName, familyName) = Person("John", "Doe", 25)
firstName = firstName.toLowerCase()
familyName = familyName.toLowerCase()
println("$firstName $familyName") // john doe
```

Note that `val/var` modifier applies to all components of destructuring declarations, so you may either declare all variables mutable, or declare them all immutable without intermediate options.

Destructuring can also be used in the `for` loops:

```
val pairs = arrayOf(1 to "one", 2 to "two", 3 to "three")
for ((number, name) in pairs) {
    println("$number: $name")
}
```

Since Kotlin 1.1, it's possible to destructure a lambda parameter:

```
fun combine(person1: Person,
            person2: Person,
            folder: ((String, Person) -> String)): String {
    return folder(folder("", person1), person2)
}
fun main() {
```

```

val p1 = Person("John", "Doe", 25)
val p2 = Person("Jane", "Doe", 26)
// Without destructuring:
println(combine(p1, p2) { text, person -> "$text
${person.age}" })
// With destructuring:
println(combine(p1, p2) { text, (firstName) -> "$text
$firstName" })
println(combine(p1, p2) { text, (_, familyName) -> "$text
$familyName" })
}

```

Note that unlike an ordinary lambda parameter list, destructured parameters are enclosed in parentheses.

Since destructuring declarations are currently only supported for local variables, they can't be declared in a class body or at the top level in a file:

```

data class Person(val firstName: String,
                 val familyName: String,
                 val age: Int)
val (firstName, familyName) = Person("John", "Doe", 25) // Error

```

Note that as of now destructuring declarations can't be nested:

```

data class Person(val firstName: String,
                 val familyName: String,
                 val age: Int)
data class Mailbox(val address: String, val person: Person)
fun main() {
    val (address, (firstName, familyName, age)) =
        Mailbox("Unknown", Person("John", "Doe", 25)) // Error
}

```

While data classes provide out-of-the-box destructuring support, in general, it may be implemented for any Kotlin type. In *Chapter 11, Domain-Specific Languages*, we'll discuss how to do it using operator overloading conventions.

Inline classes

Creating wrapper classes is quite common in programming practice: after all, this is the gist of the well-known adapter design pattern. Let's suppose that, for example, we want our program to have a concept of currency. Although money quantity is essentially a number, we'd prefer not to mix it with other numbers, which may have a very different meaning. So, we're introducing some wrapper classes and utility functions:

```
class Dollar(val amount: Int) // amount in cents
class Euro(val amount: Int)      // amount in cents
fun Dollar.toEuro() = ...
fun Euro.toDollar() = ...
```

The problem with such approach is a runtime overhead, which comes from the necessity to create an extra object whenever we're introducing a new monetary amount. The problem becomes even more significant when the wrapped value is primitive as we have in the case of our currency classes; since direct manipulation of numeric values doesn't require any object allocation at all. Using wrapper classes instead of primitives prevent many optimizations and takes a toll on the program performance.

To solve such issues, Kotlin 1.3 has introduced a new variety of classes which is called the inline class.

Defining an inline class

To define an inline class, you need to add the `inline` keyword before its name:

```
inline class Dollar(val amount: Int) // amount in cents
inline class Euro(val amount: Int)      // amount in cents
```

Such a class must have a single immutable property declared in the primary constructor. At runtime, a class instance will be represented as a value of this property without creating any wrapper object. That's the origin of the term inline class: similar to inline functions whose bodies are substituted in place of their calls, the data contained in an inline class are substituted in place of its usages.

Inline classes may have their own properties and functions:

```
inline class Dollar(val amount: Int) {
```

```

fun add(d: Dollar) = Dollar(amount + d.amount)
val isDebt get() = amount < 0
}
fun main() {
    println(Dollar(15).add(Dollar(20)).amount) // 35
    println(Dollar(-100).isDebt) // true
}

```

Inline class properties, however, may not have any state. The reason is that state would have to be inlined together with a property in the primary constructor, and currently, the Kotin compiler supports only single-property inlining. This means that no backing fields, no `lateinit`, or delegated (including `lazy`) properties are possible. Inline class properties may have only explicit accessors like the `isDebt` in our example.

It's possible to define `var` properties in the `inline` class body, although it usually makes a little sense because the inline class may not have a mutable state.

Another restriction is an inability to use initialization blocks. This is explained by the fact that inline class constructor may not execute any custom code since at runtime, so the constructor call to `Dollar(15)` must behave just like a simple mention of the number 15.

In *Chapter 2, Language Fundamentals*, we've mentioned that primitive values may be implicitly boxed if the program tries to use them in some context, which requires a reference to a real object such as assigning them to a variable of nullable type. The same also applies to inline classes: for the sake of optimization, the compiler will prefer using unwrapped values whenever possible. When it's not an option, however, the compiler will fall back to using your class as if it wasn't an inline one. For a good approximation of the compiler behavior, you may use the following rule of thumb: an inline class instance can be inlined whenever it's used as exactly the value of corresponding type without casting to something else. Consider the following example:

```

fun safeAmount(dollar: Dollar?) = dollar?.amount ?: 0
fun main() {
    println(Dollar(15).amount) // inlined
    println(Dollar(15)) // not inlined: used as
Any?
    println(safeAmount(Dollar(15))) // not inlines: used as

```

```
Dollar?  
}
```

One more point worth noting is an experimental status of inline classes. At the moment, the design of this language feature is not finalized and may change in a future version. For this reason, any definition of inline classes in Kotlin 1.3 is by default accompanied with a compiler warning. This warning may be suppressed by passing a special command-line argument `-XXLanguage:+InlineClasses` to the Kotlin compiler.

IDE Tips: When using IntelliJ, you can automatically enable or disable inline classes (or any other experimental language features such as unsigned integers) in your project by choosing an appropriate action from the Alt + Enter menu on the highlighted element (as shown in *Figure 6.3*):

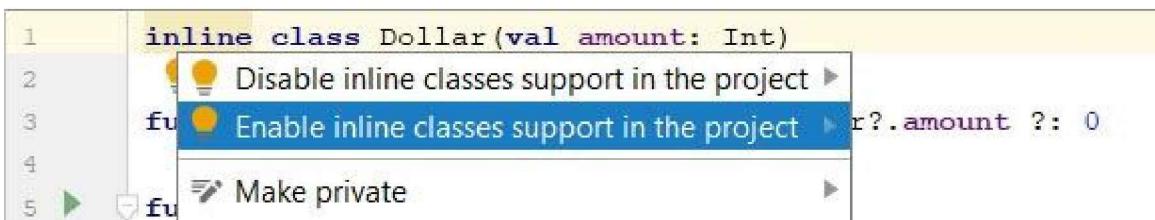


Figure 6.3: Enabling inline classes in an IntelliJ project

Unsigned integers

Since version 1.3, the Kotlin standard library includes a set of unsigned integer types implemented on top of built-in signed types using inline classes. Just like inline classes, in general, these types comprise an experimental feature, so currently their usages produce a warning unless you explicitly permit them in your project (see *Figure 6.4*):

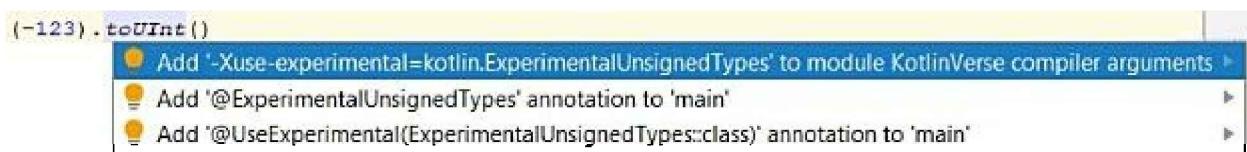


Figure 6.4: Enabling unsigned types support in an IntelliJ project

The name of each unsigned type is similar to the name of each signed counterpart with an extra U letter (*Table 6.1*):

Type	Size	Range
------	------	-------

	(in bytes)	
UByte	1	0 to 255
UShort	2	0 to 65535
UInt	4	0 to 232 - 1
ULong	8	0 to 264 - 1

Table 6.1: Unsigned integer types

To denote an unsigned value, you can add `u` or `U` suffix to an integer literal. The type of the literal is determined by its expected type such as the type of variable initialized by this value. If no expected type is specified, the literal type is supposed to be either `UInt` or `ULong` depending on its size:

```
val uByte: UByte = 1u           // explicit UByte
val uShort: UShort = 100u // explicit UShort
val UInt = 1000u           // UInt inferred automatically
val uLong: ULong = 1000u      // explicit ULong
val uLong2 = 1000uL        // explicit ULong due to L suffix
```

Signed and unsigned types are not compatible with each other, though. Therefore, you can't, for example, assign an unsigned value to a variable of a signed type and vice versa:

```
val long: Long = 1000uL      // Error
```

Unsigned and signed types can be converted one into another using one of the `toXXX()` methods:

```
println(1.toUByte())      // 1,          Int -> UByte
println((-100).toUShort()) // 65436, Int -> UShort
println(200u.toByte())     // -56,        UInt -> Byte
println(1000uL.toInt())    // 1000,       ULong -> Int
```

The unsigned types API is quite similar to that of signed integer types. In particular, any pair of unsigned values can be combined by arithmetic operators `+`, `-`, `*`, `/`, `%`:

```
println(1u + 2u) // 3
println(1u - 2u) // 4294967295
println(3u * 2u) // 6
println(5u / 2u) // 2
```

```
println(7u % 3u) // 1
```

You can't, however, combine signed values with unsigned ones:

```
println(1u + 2) // Error  
println(1 + 2u) // Error
```

Also, unlike signed types, unsigned integers do not support unary minus operation. This makes sense since they can't denote negative values:

```
println(-1u) // Error
```

Unsigned values can be used in increment/decrement expressions and augmented assignments:

```
var UInt: UInt = 1u  
++UInt  
UInt -= 3u
```

And support basic bitwise operations such as inversion, AND, OR, and XOR:

```
val ua: UByte = 67u      // 01000011  
val ub: UByte = 139u // 10001011  
println(ua.inv())        // 10111100: 188  
println(ua or ub)        // 11001011: 203  
println(ua xor ub)       // 11001000: 200  
println(ua and ub)       // 00000011: 3
```

UInt and ULong also supports left and right bitwise shifts:

```
val ua = 67u           // 0..0001000011  
println(ua shr 2) // 0..0000010000: 16  
println(ua shl 2) // 0..0100001100: 268
```

Note that the bit count is specified as a value of ordinary Int rather than UInt. Also, there is no separate ushr operation for unsigned right shifts because for unsigned integers, it behaves exactly like shr.

Similar to ordinary integers, unsigned values can be compared using <, >, <=, >=, == and != operations:

```
println(1u < 2u)           // true  
println(2u >= 3u)          // false
```

```
println(2u + 2u == 1u + 3u) // true
```

The Kotlin standard library also includes a set of auxiliary types which represent arrays of unsigned integers: `UByteArray`, `UShortArray`, `UIntArray`, `ULongArray`. These are also inline classes backed by the corresponding array classes like `IntArray`. Unsigned array types can be constructed similar to arrays we've faced so far:

```
val uBytes = ubyteArrayOf(1u, 2u, 3u)
val squares = UIntArray(10) { it*it }
```

There are also unsigned counterparts for range and progression types, which can be constructed using the `..` operator as well as operations such as `until` or `downTo`:

```
1u .. 10u           // 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
1u .. 10u step 2    // 1, 3, 5, 7, 9
1u until 10u        // 1, 2, 3, 4, 5, 6, 7, 8, 9
10u downTo 1u       // 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
10u downTo 1u step 3 // 10, 7, 4, 1
```

Conclusion

This chapter has introduced us to some special varieties of classes aimed at solving particular programming problems. We've learned to use enums to describe limited sets of objects with common functions and properties and seen how to employ data classes to concisely define simple holders of data as well as use destructuring for extraction of data classes properties. Finally, we've taken a look at possibilities of experimental inline classes introduced in Kotlin 1.3 for the purpose of creating lightweight wrappers and examined unsigned integer types based on the Kotlin inline classes.

In the next chapter, we will focus on the Kotlin standard library. In particular, we will cover basic collection types, give more extensive treatments to arrays and strings, consider I/O and networking capabilities as well as some useful utility functions.

Questions

1. What is an enum class? What built-in operations are available for enums?
2. What's the specifics of using when expressions with enum classes?
3. How can you define a enum class with custom functions or properties?

4. What is a data class? Which operations are generated automatically for any data class? How to copy a data class instance?
5. What is a destructuring declaration? Where can you use one?
6. What's the purpose of inline classes? What requirements a class must satisfy in order to be inline?
7. Describe Kotlin unsigned types and their built-in operations. What's their specifics as compared to signed integers?

CHAPTER 7

Exploring Collections and I/O

In this chapter, we'll take a look at two major components of the Kotlin standard library. The first part will be devoted to the collections API: here we'll discuss common collection types together with their basic operations and give a comprehensive treatment of various manipulations with collections and their data such as element access utilities, testing collection predicates, filtering and extracting collection parts, aggregation, transformation, and ordering. In the second part, we'll focus on the I/O API and talk about utilities that simplify both creation of I/O streams and access to their data as well some common filesystem operations.

Structure

- Collections
- Files and I/O streams

Objective

Get an understanding of Kotlin collection types and learn to use the Kotlin standard library for concise and idiomatic manipulations with collection data as well as use I/O stream API extensions.

Collections

A collection is an object designed to store a group of elements. In *Chapter 2, Language Fundamentals*, we've already discussed one example of such objects, namely, arrays that allow you to keep a fixed number of elements belonging to some common type. The Kotlin standard library, though, provides far richer collection capabilities, including both various classes based on different data structures (such as arrays, linked lists, hash tables and so on) as well as the comprehensive API for manipulating collections and their data: filtering, aggregation, transformation, ordering, and so on. In this section, we'll give a detailed account of what a collection library can offer to Kotlin developers.

It's worth noting that almost all collection-manipulating operations are inline functions.

Therefore, the ease of their use doesn't involve any performance penalties related to function calls and lambdas.

Collection types

Collection types in Kotlin can be divided into four basic categories: arrays, iterables, sequences, and maps. Since arrays have already been a major topic in *Chapter 2, Language Fundamentals*, in this section, we'll focus on the remaining three categories.

Similar to arrays, collection types are generic: when specifying a type of particular collection, you also need to specify a type of its elements: for example, `List<String>` means a list of string, while `Set<Int>` means a set of Int values.

The outline of basic collection types can be represented by the following diagram (*Figure 7.1*):

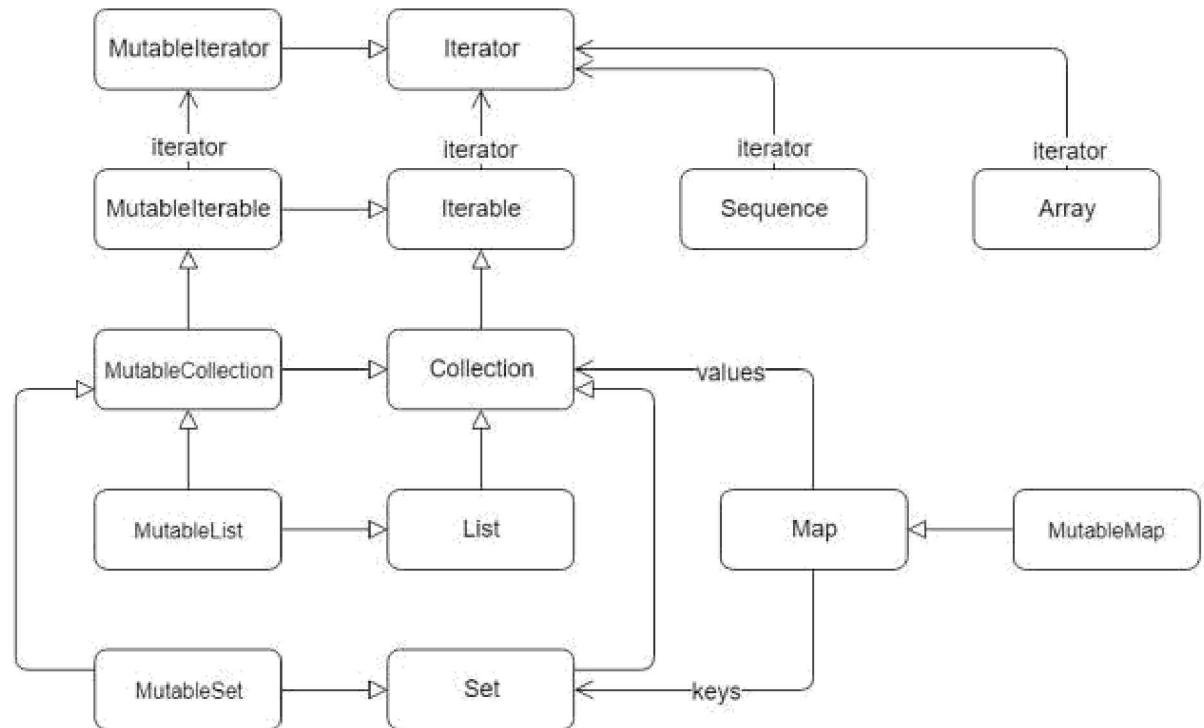


Figure 7.1: Kotlin collection types

Iterables

An iterable, represented by the `Iterable<T>` type, represents collection, which is generally both eager and stateful. The statefulness means that such collection stores contained elements in its instance rather than keeping some generator function which

can retrieve them lazily. Eagerness, on the other hand, means that collection elements are initialized at the moment of its creation instead of being computed lazily at some later point.

The iterable type itself is quite similar to its Java counterpart: it provides a single `iterator()` method which returns an object capable of traversing its elements. This allows to use Kotlin's `for` loop with any iterable:

```
val list = listOf("red", "green", "blue") // Create new list
for (item in list) {
    print(item + " ")
} // Prints red green blue
```

Java versus Kotlin: The Kotlin's `Iterator` type is basically the same as Java's: it contains two methods: `hasNext()` which checks whether the iterator has reached the end of collection, and `next()` which returns the next collection element. The only difference is the absence of the `remove()` method which is moved to `MutableIterator` instead.

A major feature of Kotlin iterables, as compared to Java, is a distinction between mutable and immutable collections. The content of immutable collections can't change after they are created, while mutable collections can generally be updated at any time by adding or removing elements. Note that collection mutability has nothing to do with mutability of a variable keeping a reference to collection instance: it means the ability to change the data this reference points to. You can, for example, keep a mutable collection in an immutable variable; in this case, you can't change the variable making it refer to some other collection, but can, for example, add or remove collection elements:

```
val list = ArrayList<String>()
list.add("abc") // Ok: changing collection data
list = ArrayList<String>() // Error: can't reassign immutable variable
```

The basic type of mutable iterable is represented by the `MutableIterable` interface, which can create a `MutableIterator` instance.

A useful feature of immutable collection types is their covariance. It means that if `T` is a subtype of `U`, then `Iterable<T>` is a subtype of `Iterable<U>`. This is also true for other collection-related types such as `Iterator`, `Collection`, `List`, `Set`, and `Map`. It, in particular, allows you write the following code:

```
fun processCollection(c: Iterable<Any>) { ... }
```

```
fun main() {
    val list = listOf("a", "b", "c") // List<String>
    processCollection(list)      // Ok: passing List<String> as
List<Any>
}
```

This, however, does not work for mutable collection. Otherwise, we could have written the code which, for example, adds integer to a list of string.

```
fun processCollection(c: MutableCollection<Any>) { c.add(123)
}
fun main() {
    val list = arrayListOf("a", "b", "c") // ArrayList<String>
    processCollection(list)           // !!!
}
```

In *Chapter 9, Generics*, we'll address the issue of covariance in more detail.

Collections, Lists, and Sets

An important subcategory of iterables is represented by the `Collection` interface and its mutable subtype, `MutableCollection`. This is a basic class for many standard implementation of iterables. `Collection` inheritors, in turn, typically belong to one of the following kinds:

- A list (represented by interfaces `List` and `MutableList`) is an ordered collection of elements with index-based element access. Common implementations of lists are `ArrayList` with fast random access by index and `LinkedList`, which can quickly add or remove elements, but requires linear time to find existing element by its index.
- A set is a collection of unique elements. Element ordering varies depending on the implementation:
 - `HashSet` is based on hash table implementation and orders elements according to their hash codes. In general, such ordering depends on particular implementations of the `hashCode()` method and so can be considered unpredictable;
 - `LinkedHashSet` is also based on a hash table, but retains the insertion order: in other words, elements are iterated in the same order as they've been inserted into the set.

- `TreeSet` is an implementation based on binary search trees; it maintains a stable element ordering according to some comparison rule, which may be implemented by elements themselves (if they inherit from the `Comparable` interface), or provided in a form of a separate `Comparator` object.

On the JVM platform, concrete classes implementing these interfaces are represented by corresponding JDK collections. Well-known Java classes like `HashMap` or `ArrayList` are seamlessly integrated into the Kotlin library.

Java versus Kotlin: In the Kotlin code, there is usually no need to use classes from the `java.util` package. Most standard collections such as `ArrayList` can be referred via aliases in the `kotlin.collections` package, which is automatically imported to all Kotlin files.

Sequences

Similar to iterables, sequences provide the `iterator()` method, which can be used to traverse their content. The intent behind them, however, is different since a sequence is supposed to be lazy. Most sequence implementations do not initialize their elements at the moment of instantiation but compute them only on demand. Many sequence implementations are also stateless meaning that they keep only a constant amount of data required to lazily generated collection elements. Iterables, on the other hand, usually spends amount of memory proportional to the number of elements.

Unlike iterables, most of the sequence implementations are internal and are not intended to be used directly. Instead new sequences are created by special functions, which we'll discuss in the upcoming sections.

Java versus Kotlin: Readers familiar with Java might recognize the similarity between sequences and streams introduced in Java 8. Since Kotlin 1.2, the standard library provides an `asSequence()` extension function, which can be used to wrap Java stream into a Kotlin sequence.

Maps

A map is set of key-value pairs where keys are unique. Although the map is not a subtype of `Collection` by itself, its content may be presented as such: you can, in particular, get a set of all keys, a collection of all values and a set of key-value pairs represented by `Map.Entry` and `MutableMap.MutableEntry` interfaces.

Since maps contain two different kinds of elements (keys and values), their types have two parameters: for example, `Map<Int, String>` is a map which associates Int keys

with String values.

Standard implementations of maps include `HashMap`, `LinkedHashMap`, and `TreeMap` which have properties similar to that of corresponding implementations of `Set`.

`AbstractMap` and `AbstractMutableMap` classes can be used as placeholders to implement your own maps.

Comparables and Comparators

Similar to Java, Kotlin supports `Comparable` and `Comparator` types which can be used in some collection operations. `Comparable` instances possess a natural ordering: each of them has the `compareTo()` function which can be used to compare it with other instances of the same type. So by making your type an inheritor of `Comparable`, you automatically allow operations such as `<` and `>`, and on top of it, you can apply ordering operations for collection with a corresponding element type. Suppose we want our `Person` class to have a natural order based on the full name. The implementation would look like this:

```
class Person(
    val firstName: String,
    val familyName: String,
    val age: Int
) : Comparable<Person> {
    val fullName get() = "$firstName $familyName"
    override fun compareTo(p: Person) =
        fullName.compareTo(p.fullName)
}
```

The convention about the `compareTo()` function is the same as in Java: it returns a positive number when current instance is greater than other, a negative number when it's smaller, and zero when both instances are equal. An implementation of `compareTo()` is supposed to be compatible with the `equals()` function.

In many cases, a given class can be compared in multiple ways. For example, we can order a collection of `Person` instances by their first or family names only, by age or by various combination of these properties. For this reason, the Kotlin library provides a concept of comparator. Similar to Java, an instance of the `Comparator<T>` class provides the `compare()` function which takes two instances of a type `T` and returns comparison result following the same convention as `compareTo()`. In Kotlin, comparators can be concisely constructed based on a comparison lambda:

```
val AGE_COMPARATOR = Comparator<Person>{ p1, p2 ->
    p1.age.compareTo(p2.age)
}
```

Alternatively, you can use `compareBy()` or `compareByDescending()` functions to provide a comparable value to be used instead of the original object:

```
val AGE_COMPARATOR = compareBy<Person>{ it.age }
val REVERSE_AGE_COMPARATOR = compareByDescending<Person>{
    it.age }
```

The comparator instance can then be passed into some ordering-aware function such as `sorted()` or `max()`. You can find examples in the upcoming sections on aggregating functions and collection ordering.

Creating a Collection

In *Chapter 2, Language Fundamentals*, we've already seen how to create array instances using either constructors or standard functions such as `arrayOf()`. Many standard collection classes may be constructed in a similar way. For example, classes such as `ArrayList` or `LinkedHashSet` can be created by an ordinary constructor call just like in the Java:

```
val list = ArrayList<String>()
list.add("red")
list.add("green")
println(list) // [red, green]
val set = HashSet<Int>()
set.add(12)
set.add(21)
set.add(12)
println(set) // [12, 21]
val map = TreeMap<Int, String>()
map[20] = "Twenty"
map[10] = "Ten"
println(map) // {10=Ten, 20=Twenty}
```

We also have functions similar to `arrayOf()` which take a variable argument list and produce an instance of some standard collection class:

- `emptyList() / emptySet()`: An instance of an immutable empty list/set (similar to `emptyXXX()` methods of the JDK Collections class);
- `listOf() / setOf()`: Creates a new immutable list/set backed by the argument array (for lists, it's basically the same as Java's `Arrays.asList()`);
- `listOfNotNull()`: Creates a new immutable list with nulls filtered out;
- `mutableListOf() / mutableSetOf()`: Creates a default implementation of a mutable list/set (internally, it's `ArrayList` and `LinkedHashSet`, respectively);
- `arrayListOf()`: Creates a new `ArrayList`;
- `hashSetOf() / linkedSetOf() / sortedSetOf()`: Creates a new instance of `HashSet`/`LinkedHashSet`/`TreeSet`, respectively;

Let's consider some examples:

```

valemptyList = emptyList<String>()
println(emptyList)      // []
emptyList.add("abc")   // Error: add is unresolved

valsingletonSet = setOf("abc")
println(singletonSet)      // [abc]
singletonSet.remove("abc") // Error: remove is unresolved

valmutableList = mutableListOf("abc")
println(mutableList) // [abc]
mutableList.add("def")
mutableList[0] = "xyz"
println(mutableList) // [xyz, def]

valsortedSet = sortedSetOf(8, 5, 7, 1, 4)
println(sortedSet) // [1, 4, 5, 7, 8]
sortedSet.add(2)
println(sortedSet) // [1, 2, 4, 5, 7, 8]

```

Similar functions are also provided for constructing maps:

- `emptyMap()`: An instance of immutable empty map;
- `mapOf()`: Creates a new immutable map (internally, it's `LinkedHashMap`);
- `mutableMapOf()`: Creates a default implementation of a mutable map (internally, it's `LinkedHashMap`);

- `hashMapOf() / linkedMapOf() / sortedMapOf()` : Creates a new instance `HashMap / LinkedHashMap / TreeMap`;

Note that the preceding map functions take a variable argument list of the `Pair` objects, which can be concisely constructed by the `to infix` operation:

```
valemptyMap = emptyMap<Int, String>()
println(emptyMap)           // {}

emptyMap[10] = "Ten" // Error: set is unresolved

valsingletonMap = mapOf(10 to "Ten")
println(singletonMap)        // {10=Ten}
singletonMap.remove("abc") // Error: remove is unresolved

valmutableMap = mutableMapOf(10 to "Ten")
println(mutableMap) // {10=Ten}
mutableMap[20] = "Twenty"
mutableMap[100] = "Hundred"
mutableMap.remove(10)
println(mutableMap) // {20=Twenty, 100=Hundred}

val sortedMap = sortedMapOf(3 to "three", 1 to "one", 2 to
"two")
println(sortedMap) // {1=one, 2=two, 3=three}
sortedMap[0] = "zero"
println(sortedMap) // {0=zero, 1=one, 2=two, 3=three}
```

Alternatively, you can create a mutable map and fill it using the `set()` method or indexing operator to avoid creation of excessive `Pair` instances.

Lists can also be constructed similar to arrays by specifying its size and a function that maps index to the element value:

```
println(List(5) { it*it }) // [0, 1, 4, 9, 16]

val numbers = MutableList(5) { it*2 }

println(numbers) // [0, 2, 4, 6, 8]
numbers.add(100)
println(numbers) // [0, 2, 4, 6, 8, 100]
```

The simplest way to create a sequence of known elements is to use a `sequenceOf()`

function, which takes a variable number of arguments. Alternatively, you can convert an existing collection such as an array, iterable, or map into a sequence by calling the `asSequence()` function:

```
println(sequenceOf(1, 2, 3).iterator().next())
// 1
println(listOf(10, 20, 30).asSequence().iterator().next()) // 10
println(
    mapOf(1 to "One", 2 to
    "Two").asSequence().iterator().next()
)
// 1=One
```

Note that calling `asSequence()` on a map gives you a sequence of map entries.

Another option is to create a sequence based on some generator function. This case is implemented by a pair of `generateSequence()` functions. The first one takes a parameterless function, which computes next sequence element. Sequence generation proceeds until this function returns null. For example, the following code create a sequence that reads the program input until it encounters a non-number or the input is exhausted:

```
val numbers = generateSequence{ readLine()?.toIntOrNull() }
```

The second `generateSequence()` function takes an initial value and single-parameter function, which generates new sequence element based on the previous one. Just like in the first case, a generation stops when this function returns null:

```
// Infinite sequence (with overflow): 1, 2, 4, 8, ...
val powers = generateSequence(1) { it*2 }
// Finite sequence: 10, 8, 6, 4, 2, 0
val evens = generateSequence(10) { if (it >= 2) it - 2 else
null }
```

Since Kotlin 1.3, one more way to construct a sequence is to use a special builder which allows you to provide sequence elements in parts. The builder is implemented by the `sequence()` function, which accepts an extension lambda with the `SequenceScope` receiver type. This type introduces a set of functions which can be used to append elements to a new sequence:

- `yield()`: Add a single element;

- `yieldAll()` : Add all elements of the specified iterator, iterable, or sequence.

Note that elements are added lazily: the `yield()`/`yieldAll()` calls are executed only when a corresponding chunk of sequence is requested. Consider the following example:

```
val numbers = sequence {
    yield(0)
    yieldAll(listOf(1, 2, 3))
    yieldAll(intArrayOf(4, 5, 6).iterator())
    yieldAll(generateSequence(10) { if (it < 50) it*3 else
null })
}
println(numbers.toList()) // [0, 1, 2, 3, 4, 5, 6, 10, 30, 90]
```

The sequence builder implemented by `sequence()`/`yield()`/`yieldAll()` functions is, in fact, an example of suspendable computations, a powerful Kotlin feature which gets especially useful in multi-threaded applications. We'll defer its detailed treatment till *Chapter 13, Concurrency*.

Final group of functions we'd like to mention in this section deals with collection conversion: for example, they allow you to create a list based on the content of array or turn a sequence into a set:

```
println(
    listOf(1, 2, 3, 2, 3).toSet()
) // [1, 2, 3]
println(
    arrayOf("red", "green", "blue").toSortedSet()
) // [blue, green, red]
println(
    mapOf(1 to "one", 2 to "two", 3 to "threen").toList()
) // [(1, one), (2, two), (3, threen)]
println(
    sequenceOf(1 to "one", 2 to "two", 3 to "threen").toMap()
) // {1=one, 2=two, 3=threen}
```

You can find a complete list of the conversion functions in the standard library reference at kotlinlang.org/api/latest/jvm/stdlib. Conversion functions follows a certain convention: namely, functions whose name starts with `to` (such as `toList()` or

`toMap()`) create a separate copy of the original collection, while those which start with `as`(such as `asList()`) create a view that reflects any change in the original collection.

IDE Tips: Do not hesitate using IDE completion (available by **Ctrl + Space/Cmd + Space**) to help you with choosing a conversion function or any other method (see example on the *Figure 7.2*):

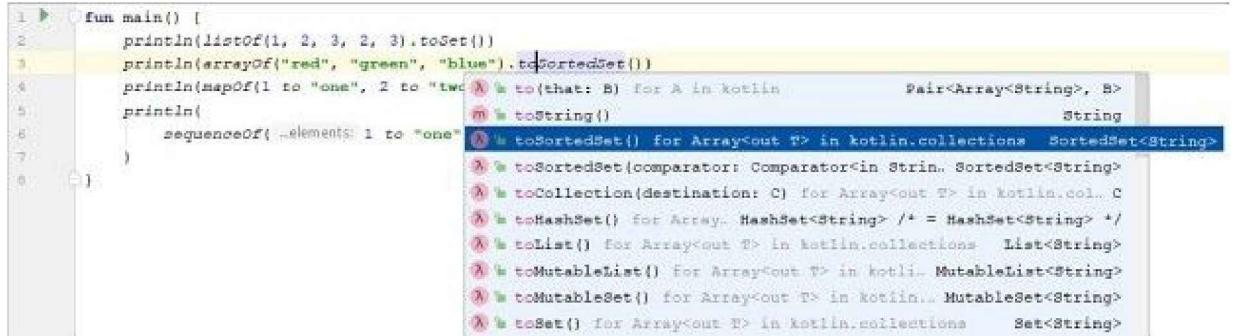


Figure 7.2: Using completion to choose completion function

New collections may also be created on the base of existing ones through operations such as filtering, transformation, or sorting. We'll cover such cases in the upcoming sections.

Basic operations

In this section, we'll take a look at basic operations available for Kotlin collection types.

One common operation supported by all collections is iteration. Arrays, iterables, sequences, and maps support the `iterator()` function. Although the instance of `Iterator` object returned by this function can definitely be used to traverse collection elements, this is rarely needed in practice since Kotlin provides more concise ways to do the same job.

In particular, the presence of the `iterator()` function allows us to use the `for` loop with any collection as we've already seen in the *Iterables* section. One thing worth pointing out is that the map iterator returns instances of `Map.Entry`. In Kotlin, the map entries support destructuring, which allows writing map iteration like this:

```
val map = mapOf(1 to "one", 2 to "two", 3 to "three")
for ((key, value) in map) {
    println("$key -> $value")
}
```

This also goes for lambdas that accept a map entry as their parameter.

An alternative is to use the `forEach()` extension function which execute a supplied lambda for each collection element:

```
intArrayOf(1, 2, 3).forEach { println(it*it) }
listOf("a", "b", "c").forEach { println("'$it'") }
sequenceOf("a", "b", "c").forEach { println("'$it'") }
mapOf(1 to "one", 2 to "two", 3 to "three").forEach { (key,
value) ->
println("$key -> $value")
}
```

If you want to additionally take element indices into account, there is a more general `forEachIndexed()` function:

```
listOf(10, 20, 30).forEachIndexed { i, n ->println("$i:
${n*n}") }
```

Basic features of the Collection type include:

- The `size` property, which gives you a number of elements;
- The `isEmpty()` function, which returns true if collection has no elements;
- The `contains()/containsAll()` functions, which check whether a collection contains a specific elements or all elements of another collection.

A call to the `contains()` function may be replaced by the `in` operator:

```
val list = listOf(1, 2, 3)
println(list.isEmpty()) // false
println(list.size) // 3
println(list.contains(4)) // false
println(2 in list) // true
println(list.containsAll(listOf(1, 2))) // true
```

Note that behavior of `contains()/containsAll()` depends on the proper implementation of the `equals()` method. If you use instances of your own classes as collection elements, make sure to implement content-based equality when necessary.

The `MutableCollection` type introduces methods for adding and removing elements. Consider the following example:

```
val list = arrayListOf(1, 2, 3)
list.add(4) // Add single: [1, 2,
3, 4]
list.remove(3) // Remove single: [1, 2, 4]
list.addAll(setOf(5, 6)) // Union: [1, 2, 4, 5, 6]
list.removeAll(listOf(1, 2)) // Difference: [4,
5, 6]
list.retainAll(listOf(5, 6, 7)) // Intersection: [5, 6]
list.clear() // Remove all: []

You can also use += and -= operators instead of
add()/remove()/addAll()/removeAll() calls:
```

```
list += 4
list -= 3
list += setOf(5, 6)
list -= listOf(1, 2)
```

Both mutable and immutable collections support + and - operator, which produce a new collection leaving the original untouched:

```
println(listOf(1, 2, 3) + 4) // [1, 2, 3, 4]
println(listOf(1, 2, 3) - setOf(2, 5)) // [1, 3]
```

You can also use += and -= with immutable collection, but with a very different semantics: for immutable collection, they act as abbreviations for assignments and thus can be applied only to mutable variables:

```
val readOnly = listOf(1, 2, 3)
readOnly += 4 // Error: can't assign to val
var mutable = listOf(1, 2, 3)
mutable += 4 // Correct
```

Such code, however, should be avoided in general since it implicitly creates a new collection object on each assignment, which may affect a program performance.

IDE Tips: The IntelliJ plugin warns you about such assignments suggesting to use mutable collection instead of an immutable one (see *Figure 7.3*).

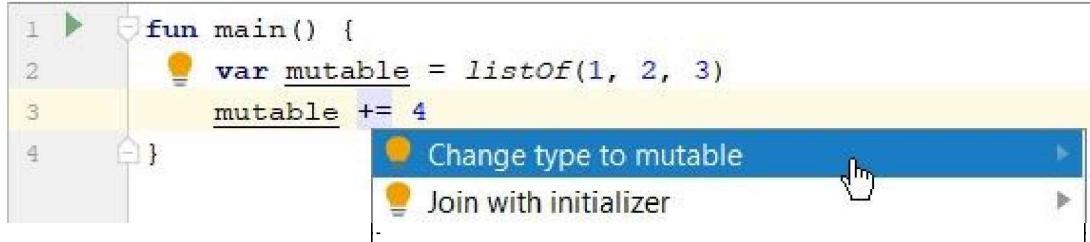


Figure 7.3: Replacing immutable collection with a mutable one

The list introduces some methods to access its elements by index similarly to arrays:

```
val list = listOf(1, 4, 6, 2, 4, 1, 7)  
println(list.get(3))          // 2  
println(list[2])              // 6  
println(list[10])             // Exception  
println(list.indexOf(4))       // 1  
println(list.lastIndexof(4))   // 4  
println(list.indexOf(8))       // -1
```

Note that indexing notation is generally more preferable than the call to the `get()` method. When a list is mutable, its elements may also be changed by index:

```
val list = arrayListOf(1, 4, 6, 2, 4, 1, 7)  
list.set(3, 0)      // [1, 4, 6, 0, 4, 1, 7]  
list[2] = 1         // [1, 4, 1, 0, 4, 1, 7]  
list.removeAt(5)   // [1, 4, 1, 0, 4, 7]  
list.add(3, 8)     // [1, 4, 1, 8, 0, 4, 7]
```

The `subList()` function creates a wrapper over particular segment of the list specified by start (inclusive) and end (exclusive) indices. The view shares data with original collection, and in the case of mutable list reflects changes in its data:

```
val list = arrayListOf(1, 4, 6, 2, 4, 1, 7)  
val segment = list.subList(2, 5) // [6, 2, 4, 1]  
list[3] = 0  
println(segment[1]) // 0  
segment[1] = 8  
println(list[3])    // 8
```

Sets do not introduce any additional operations by themselves. Their implementation of common Collection methods, however, ensure that no duplicates are added to a

collection.

The methods of the Map interface allow you to retrieve a value by key as well as provide access to the full key set and value collection. Let's consider an example:

```
val map = mapOf(1 to "I", 5 to "V", 10 to "X", 50 to "L")
println(map.isEmpty())                                // false
println(map.size)                                     // 4
println(map.get(5))                                    // V
println(map[10])                                     // X
println(map[100])                                    // null
println(map.getOrDefault(100, "?")) // ?
println(map.getOrElse(100) { "?" }) // ?
println(map.containsKey(10))                           // true
println(map.containsValue("C"))                         // false
println(map.keys)                                     // [1, 5, 10, 50]
println(map.values)                                    // [I, V, X, L]
println(map.entries)                                   // [1=I, 5=V, 10=X, 50=L]
```

`MutableMap` introduces basic modification methods as well support of + and - operators:

```
val map = sortedMapOf(1 to "I", 5 to "V")
map.put(100, "C")                                    // {1=I, 5=V, 100=C}
map[500] = "D"                                       // {1=I, 5=V, 100=C, 500=D}
map.remove(1)                                         // {5=V, 100=C, 500=D}
map.putAll(mapOf(10 to "X")) // {5=V, 10=X, 100=C, 500=D}
map += 50 to "L"                                     // {5=V, 10=X, 50=L, 100=C,
500=D}
map += mapOf(2 to "II",
            3 to "III")                                // {2=II, 3=III, 5=V, 10=X,
50=L, 100=C, 500=D}
map -= 100                                           // {2=II, 3=III, 5=V, 10=X, 50=L,
500=D}
map -= listOf(2, 3)                                    // {5=V, 10=X, 50=L, 500=D}
```

The comment about += and -= operators with respect to mutable and immutable collection is also valid for maps. Note also that while + operators take key-value pairs, - operators take keys.

Accessing collection elements

Besides basic collection operations, the Kotlin standard library contains a set of extension functions simplifying an access to individual collection elements, which we'll discuss in this section.

The `first()`/`last()` functions return respectively the first and the last element of a given collection throwing a `NoSuchElementException` instance if the collection is empty. There are also safe versions called `firstOrNull()`/`lastOrNull()`, which return `null` when no elements are found:

```
println(listOf(1, 2, 3).first())           // 1
println(listOf(1, 2, 3).last())             // 3
println(emptyArray<String>().first())      // Exception
println(emptyArray<String>().firstOrNull()) // null
val seq = generateSequence(1) { if (it > 50) null else it * 3 }
println(seq.first())                      // 1
println(seq.last())                      // 81
```

These function may also be passed a predicate in which case they will look for the first or the last element matching the corresponding condition:

```
println(listOf(1, 2, 3).first { it > 2 })    // 3
println(listOf(1, 2, 3).lastOrNull { it < 0 })  // null
println(intArrayOf(1, 2, 3).first { it > 3 })   // Exception
```

The `single()` function returns the element of a singleton collection. If a collection is empty or contains more than one element, `single()` throws an exception. Its safe counterpart, `singleOrNull()`, returns `null` in both cases:

```
println(listOf(1).single())           // 1
println(emptyArray<String>().singleOrNull()) // null
println(setOf(1, 2, 3).singleOrNull())   // null
println(sequenceOf(1, 2, 3).single())    // Exception
```

The `elementAt()` function allows you to retrieve collection element by its index. It generalizes the `get()` function of lists and can be applied to any array, iterable, or sequence. Bear in mind, though, that applying this function to a non-random access list will, in general, take time proportional to the value of index.

In the case of invalid index, `elementAt()` throws an exception. There are also variants

that provide a safer behavior when an index violates collection bounds: `elementAtOrNull()` which simply returns null and `elementAtOrElse()` which returns the value of supplied lambda:

```
println(listOf(1, 2, 3).elementAt(2))                                // 3
println(sortedSetOf(1, 2,
3).elementAtOrNull(-1))                      // null
println(arrayOf("a", "b", "c").elementAtOrElse(1) { "???" })
// b
val seq = generateSequence(1) { if (it > 50) null else it * 3
}
println(seq.elementAtOrNull(2))                                // 9
println(seq.elementAtOrElse(100) { Int.MAX_VALUE
})                // 81
println(seq.elementAt(10))                                // Exception
```

One more thing to mention is a support of destructuring for arrays and lists which allows you to extract up to 5 first elements. Note, however, that destructuring will throw an exception if you try to extract more elements than there is in a collection:

```
val list = listOf(1, 2, 3)
val (x, y) = list      // 1, 2
val (a, b, c, d) = list // Exception
```

Collective conditions

Checking that some collection satisfies certain condition is a quite common task. For this reason, the Kotlin library includes a set of functions implementing basic checks such as testing given predicate against collection elements.

The `all()` function returns true if all collection elements satisfy a given predicate. This function can be applied to any collection object, including arrays, iterables, sequences, and maps. In the case of maps, the predicate parameter is a map entry:

```
println(listOf(1, 2, 3, 4).all { it < 10 })          // true
println(listOf(1, 2, 3, 4).all { it % 2 == 0 }) // false
println(
mapOf(1 to "I", 5 to "V", 10 to "X")
```

```

.all { it.key == 1 || it.key % 5 == 0 }
) // true
// 1, 3, 9, 27, 81
val seq = generateSequence(1) { if (it < 50) it*3 else null }
println(seq.all { it % 3 == 0 }) // false
println(seq.all { it == 1 || it % 3 == 0 }) // true

```

The `none()` function tests the opposite condition: it returns true when there is no collection elements satisfying a predicate:

```

println(listOf(1, 2, 3, 4).none { it > 5 }) // true
println(
    mapOf(1 to "I", 5 to "V", 10 to "X").none { it.key % 2 ==
0 } // false
)
// 1, 3, 9, 27, 81
val seq = generateSequence(1) { if (it < 50) it*3 else null }
println(seq.none { it >= 100 }) // true

```

One more function of this kind is `any()`, which returns true when a predicate is satisfied by at least one collection element:

```

println(listOf(1, 2, 3, 4).any { it < 0 }) // false
println(listOf(1, 2, 3, 4).any { it % 2 == 0 }) // true
println(
    mapOf(1 to "I", 5 to "V", 10 to "X").any { it.key == 1 }
) // true
// 1, 3, 9, 27, 81
val seq = generateSequence(1) { if (it < 50) it*3 else null }
println(seq.any { it % 3 == 0 }) // true
println(seq.any { it > 100 }) // false

```

For an empty collection, the `all()` and `none()` functions return true while `any()` returns false. All three functions can be expressed in terms of one another using the relationships reminiscent of the De Morgan's law:

```

c.all{ p(it) } == c.none { !p(it) }
c.none{ p(it) } == !c.any { p(it) }

```

Bear in mind that `all()`, `none()`, and `any()` may run forever when applied to an

infinite sequence. For example, the following code will never terminate:

```
// 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, ...
val seq = generateSequence(0) { (it + 1) % 5 }
println(seq.all { it < 5 })
```

The `any()` and `none()` functions also have overloads which do not take any parameter and simply check if the collection in question is empty:

```
println(emptyList<String>().any())      // false
println(emptyList<String>().none()) // true
println(listOf(1, 2, 3).any())           // true
println(listOf(1, 2, 3).none())         // false
```

These overloads generalize `isNotEmpty()/isEmpty()` functions, which are available for arrays, instances of `Collection` and `Map` types, but not for arbitrary iterables or sequences.

Aggregation

Aggregation is a computation of a single value based on the collection content such as summing up collection elements or finding a maximum value. The Kotlin library provides a set of functions, which can be used for this purpose. In a previous section, we've covered a group of functions which test some collective conditions such as `any()` or `all()`: they can be considered a special kind of aggregates computing a Boolean value.

Aggregate functions, in general, shouldn't be applied to infinite sequences as they (with an exception of `count()` below) will never return in such a case.

The aggregation functions can be divided into three basic groups. The first one includes functions that compute commonly used aggregates such as `sum`, `min`, or `max`. Let's take a closer look at what they can do.

The `count()` function gives you the number of elements in a collection. It can be applied to any collection object, including arrays, iterables, sequences, and maps, and thus, generalizes the `size` property available for arrays, maps, and `Collection` instances:

```
println(listOf(1, 2, 3, 4).count())          // 4
println(mapOf(1 to "I", 5 to "V", 10 to "X").count()) // 3
// 1, 3, 9, 27, 81
val seq = generateSequence(1) { if (it < 50) it*3 else null }
```

```
println(seq.count()) // 5
```

Note that `count()` throws an exception if number of elements exceeds `Int.MAX_VALUE`. This, in particular, happens when `count()` is invoked on an infinite sequence:

```
// 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, . . .
val seq = generateSequence(0) { (it + 1) % 5 }
// Throws an exception after iterating through Int.MAX_VALUE
elements
println(seq.count())
```

The `count()` function has an overload that takes a predicate applied to collection elements. In this case, it returns the number of collection elements satisfying a given condition:

```
println(listOf(1, 2, 3, 4).count { it < 0 }) // 0
println(listOf(1, 2, 3, 4).count { it % 2 == 0 }) // 2
println(
    mapOf(1 to "I", 5 to "V", 10 to "X").count { it.key == 1 })
// 1
// 1, 3, 9, 27, 81
val seq = generateSequence(1) { if (it < 50) it*3 else null }
println(seq.count { it % 3 == 0 }) // 4
println(seq.count { it > 100 }) // 0
```

The `sum()` function computes an arithmetic sum for a numeric array, iterable, or sequence:

```
println(listOf(1, 2, 3, 4).sum()) // 10
println(doubleArrayOf(1.2, 2.3, 3.4).sum()) // 6.9
// Summing 1, 3, 9, 27, 81
val seq = generateSequence(1) { if (it < 50) it*3 else null }
println(seq.sum) // 121
```

The type of return value depends on the element type of original collection similar to an ordinary `+` operation: for example, summing up the collection of bytes will get you an `Int`, while applying `sum()` to, say, a `LongArray` will result in a value of `Long`.

Summation can also be applied to a collection of an arbitrary element type, provided it can be converted to a number. This can be achieved with `sumBy()` and

`sumByDouble()`, which takes a conversion function as a parameter: the difference is that `sumBy()` converts collection elements to `Int` values (or `UInt` when applied to a collection of unsigned integers), while `sumByDouble()` converts them to `Double`:

```
println(listOf(1, 2, 3, 4).sumByDouble { it/4.0 })           // 2.5
println(arrayOf("1", "2", "3").sumBy { it.toInt() }) // 6
// X, XX, XXX, XXXX, XXXXX
val seq = generateSequence("X") {
    if (it.length >= 5) null else it + "X"
}
println(seq.sumBy { it.length })                                // 15
```

The `average()` function similarly computes an arithmetic average of a numeric array, iterable, or sequence. The result is always a value of `Double`:

```
println(listOf(1, 2, 3, 4).average())                     // 2.5
println(doubleArrayOf(1.2, 2.3, 3.4).average()) // 2.3000000000000003
// Averaging 1, 3, 9, 27, 81
val seq = generateSequence(1) { if (it < 50) it*3 else null }
println(.average())                                         // 24.2
```

When collection is empty, the `average()` function always returns `Double.NaN`. For a non-empty collection, `c.average()` is essentially the same as `c.sum().toDouble()/c.count()`. Similar to the `count()` function, the `average()` function will throw an exception if collection contains more than `Int.MAX_VALUE` elements.

The `min()` and `max()` functions compute respectively the smallest and the largest value for an array/iterable/sequence of comparable values:

```
println(intArrayOf(5, 8, 1, 4, 2).min())                  // 1
println(intArrayOf(5, 8, 1, 4, 2).max())                  // 8
println(listOf("abc", "w", "xyz", "def", "hij").min()) // abc
println(listOf("abc", "w", "xyz", "def", "hij").max()) // xyz
// 1, -3, 9, -27, 81
val seq = generateSequence(1) { if (it < 50) -it * 3 else null }
println(seq.min())                                         // -27
println(seq.max())                                         // 81
```

Similar to summation, min/max can be computed for collections of non-comparable elements by providing a function that converts them to comparables. This behavior is implemented in `minBy()` and `maxBy()` functions:

```
class Person(val firstName: String,  
val familyName: String,  
val age: Int) {  
    override fun toString() = "$firstName $familyName: $age"  
}  
fun main() {  
    val persons = sequenceOf(  
        Person("Brook", "Watts", 25),  
        Person("Silver", "Hudson", 30),  
        Person("Dane", "Ortiz", 19),  
        Person("Val", "Hall", 28)  
    )  
    println(persons.minBy { it.firstName }) // Brook Watts:  
25  
    println(persons.maxBy { it.firstName }) // Val Hall: 28  
    println(persons.minBy { it.familyName }) // Val Hall: 25  
    println(persons.maxBy { it.familyName }) // Brook Watts:  
28  
    println(persons.minBy { it.age }) // Dane Ortiz: 19  
    println(persons.maxBy { it.age }) // Silver Hudson:  
30  
}
```

Alternatively, you use `minWith()`/`maxWith()`, which accepts a comparator instance instead of a conversion function. In the following example, we'll use different comparators to impose ordering by full name with first name coming either first, or last:

```
class Person(val firstName: String,  
val familyName: String,  
val age: Int) {  
    override fun toString() = "$firstName $familyName: $age"  
}  
valPerson.fullNameget() = "$firstName $familyName"  
valPerson.reverseFullNameget() = "$familyName $firstName"
```

```

val FULL_NAME_COMPARATOR = Comparator<Person>{ p1, p2 ->
    p1.fullName.compareTo(p2.fullName)
}
val REVERSE_FULL_NAME_COMPARATOR = Comparator<Person>{ p1, p2
->
    p1.reverseFullName.compareTo(p2.reverseFullName)
}
fun main() {
    val persons = sequenceOf(
        Person("Brook", "Hudson", 25),
        Person("Silver", "Watts", 30),
        Person("Dane", "Hall", 19),
        Person("Val", "Ortiz", 28)
    )
    // Brook Hudson: 25
    println(persons.minWith(FULL_NAME_COMPARATOR))
    // Val Ortiz: 28
    println(persons.maxWith(FULL_NAME_COMPARATOR))
    // Dane Hall: 19
    println(persons.minWith(REVERSE_FULL_NAME_COMPARATOR))
    // Silver Watts: 30
    println(persons.maxWith(REVERSE_FULL_NAME_COMPARATOR))
}

```

All variants of min/max aggregates return null when applied to an empty collection.

The second group of aggregate functions deals with combining collection elements into strings. The basic function is `joinToString()`, which in its simplest form doesn't take any parameter:

```
println(listOf(1, 2, 3).joinToString()) // 1, 2, 3
```

By default, the elements are converted to String using their `toString()` method and concatenated together with space-commas which serve as separators. In many cases, though, you'd need a custom conversion which can be supplied by a lambda parameter. Suppose we want to present our values in a binary numeral system:

```
println(listOf(1, 2, 3).joinToString { it.toString(2) }) // 1,
10, 11
```

Besides this, it's possible to specify the following optional parameters:

- `separator`: A string inserted between elements (, by default);
- `prefix` and `postfix`: string inserted at the beginning and the end of resulting string, respectively (both empty by default);
- `limit`: A maximum number of elements to show (-1 by default which means the number is not limited);
- `truncated`: When the limit is non-negative, this parameter specifies a string which is added instead of skipped elements (... by default).

The `joinToString()` function is available for any array, iterable, and sequence. Here is an example illustrating different options:

```
val list = listOf(1, 2, 3)
println(list.joinToString(prefix = "[", postfix = "]")) // [1,
2, 3]
println(list.joinToString(separator = "|")) // 1|2|3
println(list.joinToString(limit = 2)) // 1, 2,
...
println(list.joinToString(
    limit = 1,
    separator = " ",
    truncated = "???" // 1 ???
))
```

The Kotlin library also includes a more general function `joinTo()`, which appends characters to the arbitrary `Appendable` instance such as `StringBuilder` instead of producing a new string:

```
import java.lang.StringBuilder
fun main() {
    val builder = StringBuilder("joinTo: ")
    val list = listOf(1, 2, 3)
    println(list.joinTo(builder, separator = "|")) // joinTo:
1|2|3
}
```

The third group we will cover in this section allows implementing your own custom

aggregates based on functions which combine a pair of values. This group is represented by `fold()`/`reduce()` functions and their varieties.

The `reduce()` function takes a two-parameter function where the first parameter contains an accumulated value and the second one contains a current collection element. The aggregation proceeds as follows:

1. Initialize an accumulator to the value of the first element
2. For each successive element, combine the current value of accumulator with an element and assign the result back to the accumulator
3. Return the value of the accumulator

If collection is empty, the `reduce()` function throws an exception since the accumulator can't be initialized.

Let's consider an example. In the following code, we use `reduce()` to compute a product of numbers and concatenation of strings:

```
println(intArrayOf(1, 2, 3, 4, 5).reduce { acc, n -> acc * n
}) // 120
println(listOf("a", "b", "c", "d").reduce { acc, s -> acc + s
}) // abcd
```

If the aggregation rule depends on the element indices, you may use a `reduceIndexed()` function, which passes the current index as the first parameter of the aggregator operation. Suppose we want to modify the preceding example to sum only elements on odd positions:

```
// 8
println(intArrayOf(1, 2, 3, 4, 5)
.reduceIndexed { i, acc, n -> if (i % 2 == 1) acc * n else acc
})
// abd
println(listOf("a", "b", "c", "d")
.reduceIndexed { i, acc, s -> if (i % 2 == 1) acc + s else acc
})
```

Note that the first element is processed regardless of our constraint. If you want to choose the initial value by yourself, you can use `fold()`/`foldIndexed()` functions instead of `reduce()`/`reduceIndexed()`. On top of it, they allow you to use the accumulator of a type which differs from that of collection elements:

```

println(
    intArrayOf(1, 2, 3, 4).fold("") { acc, n -> acc + ('a' + n
- 1) }
) // abcd
println(
    listOf(1, 2, 3, 4).foldIndexed("") { i, acc, n ->
        if (i % 2 == 1) acc + ('a' + n - 1) else acc
    }
) // bd

```

Unlike `reduce()`, `fold()` doesn't fail on empty collection since the initial value is supplied by the programmer.

The `reduce()`/`reduceIndexed()` and `fold()`/`foldIndexed()` functions are available for any array, iterable, or sequence. Each of these function has a counterpart which processed elements in a reverse order starting from the last one. Such functions have a Right word in their name and are available only for arrays and lists since these objects provide an easy way to traverse them backward:

```

println(
    arrayOf("a", "b", "c", "d").reduceRight { s, acc -> acc +
s }
) // dcba
println(
    listOf("a", "b", "c", "d").reduceRightIndexed { i, s, acc
->
        if (i % 2 == 0) acc + s else acc
    }
) // dca
println(
    intArrayOf(1, 2, 3, 4).foldRight("") { n, acc -> acc +
('a' + n - 1) }
) // dcba
println(
    listOf(1, 2, 3, 4).foldRightIndexed("") { i, n, acc ->
        if (i % 2 == 0) acc + ('a' + n - 1) else acc
    }
) // ca

```

Mind the difference in the parameter order for lambdas passed to left and right varieties of fold/reduce: in the left version, the accumulator comes before the current element, while in the right version, the order is reversed.

Filtering

The Kotlin standard library provides a bunch of extension functions that can be used to filter collections leaving out elements, which do not satisfy a given condition. A filtering operation does not modify an original collection: it either produces an entirely new one or puts all accepted elements into some existing mutable collection distinct from the original one.

The most basic filtering operation is given by the `filter()` function: its predicate takes the current element as its single parameter and returns true if that element is accepted, and false otherwise. The function is applicable to arrays, iterables, maps, and sequences with the return type determined as follows:

- Filtering `Array<T>` or `Iterable<T>` gives you a `List<T>`;
- Filtering a `Map<K, V>` gives you a `Map<K, V>`;
- Filtering a `Sequence<T>` gives you a `Sequence<T>`.

This function is also applicable to primitive array types such as `IntArray` with the result being a `List` with a corresponding boxed element type such as `List<Int>`. Bear this in mind as applying `filter()` to such arrays will force boxing of filtered elements.

Let's consider an example of applying `filter()` to various collection objects:

```
// List: [green, blue, green]
println(
    listOf("red", "green", "blue", "green").filter {
it.length> 3 }
)
// List: [green, blue]
println(setOf("red", "green", "blue", "green").filter {
it.length> 3 })
// List: [green, blue, green]
println(
    arrayOf("red", "green", "blue", "green").filter {
it.length> 3 }
)
```

```

// List: [2, 4]
println(byteArrayOf(1, 2, 3, 4, 5).filter { it % 2 == 0 })
// Map: {X=10, L=50}
println(
    mapOf("I" to 1, "V" to 5, "X" to 10, "L" to 50)
.filter { it.value > 5 }
)
// Sequence
val seq = generateSequence(100) {
    if (it != 0) it/3 else null
}.filter { it > 10 }
// Converted to list: [100, 33, 11]
println(seq.toList())

```

Note that in the case of a map, the predicate parameter takes a value of corresponding map entry. If you want to filter only by key or value, you may use either the `filterKeys()` or `filterValues()` function:

```

val map = mapOf("I" to 1, "V" to 5, "X" to 10, "L" to 50)
println(map.filterKeys { it != "L" })      // {X=10, V=5, X=10}
println(map.filterValues { it >= 10 }) // {X=10, L=50}

```

The `filterNot()` function allows you to filter by a negative condition: in other words, a collection element is accepted when corresponding predicate returns false:

```

// [red]
println(listOf("red", "green", "blue").filterNot { it.length >
3 })
// {I=1, V=5}
println(
    mapOf("I" to 1, "V" to 5, "X" to 10, "L" to 50)
.filterNot { it.value > 5 }
)

```

Note that `filterKeys()` and `filterValues()` do not have a negative version like `filterNot()`.

If your filtering condition depends on element index as well as its value, you may use a `filterIndexed()` function whose lambda takes an additional index parameter. This function is available for arrays, iterables, and sequences, but not for maps:

```

val list = listOf("red", "green", "blue", "orange")
// [green, blue]
println(
list.filterIndexed { i, v -> v.length > 3 && i < list.lastIndex }
)
val seq = generateSequence(100) { if (it != 0) it/3 else null
}
// [33, 11, 3, 1]
println(seq.filterIndexed { i, v -> v > 0 && i > 0 }.toList())

```

The standard library also includes filtering functions based on some common conditions, which often arise in practice. One of them is `filterNotNull()`, which filters out null values. It always produces a collection with a non-null element type:

```

val list = listOf("red", null, "green", null, "blue")
// Error: it is nullable here
list.forEach { println(it.length) }
// Ok: it is non-nullable
list.filterNotNull().forEach { println(it.length) }

```

IDE Tips: IntelliJ includes an out-of-the-box inspection, which warns you about redundant `filterNotNull()` calls when the collection in question already has a non-nullable element type. You can easily drop extra filter using the Alt + Enter menu (as shown on the *Figure 7.4*):

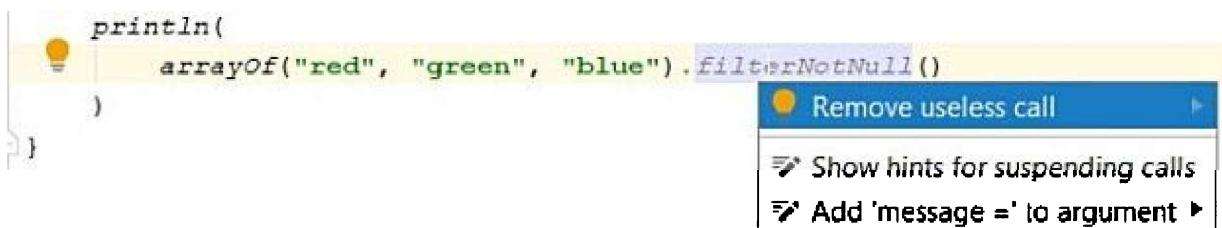


Figure 7.4: Removing useless filter

Another common case is covered by the `filterIsInstance()` function, which leaves only elements conforming to the specific type. The collection returned by this function have the same element type you specify in its call:

```

val hotchpotch = listOf(1, "two", 3, "four", 5, "six")
val numbers = hotchpotch.filterIsInstance<Int>()
val strings = hotchpotch.filterIsInstance<String>()

```

```
println(numbers.filter { it > 2 })           // [3, 5]
println(strings.filter { it != "two" }) // [four, six]
```

The filtering functions we've seen so far produce new immutable collections on each call. What if we need to put the filtering results into some existing mutable collection? In that case, we can use special versions of filter functions that take an additional parameter for the target collection where they put accepted values. The names of these function have To added to them:

```
val allStrings = ArrayList<String>()
// Added: green, blue
listOf("red", "green", "blue").filterTo(allStrings) {
    it.length > 3 }
// Added: one, two, three
arrayOf("one", null, "two", null,
        "three").filterNotNullTo(allStrings)
// abcde, bcde, cde, de, e,
val seq = generateSequence("abcde") {
    if (it.isNotEmpty()) it.substring(1) else null
}
// Added: abcde, bcde, cde
seq.filterNotTo(allStrings) { it.length < 3 }
// [green, blue, one, two, three, abcde, bcde, cde]
println(allStrings)
```

To versions are available for filter(), filterNot(), filterIndexed(), filterIsInstance(), and filterNotNull() functions. Note that an attempt to use the original collection as a target would, in general, lead to ConcurrentModificationException due to adding elements during the collection traversal:

```
val list = arrayListOf("red", "green", "blue")
list.filterTo(list) { it.length > 3 } // Exception
```

Besides various kinds of filtering, the Kotlin standard library includes the partition() function which splits the original collection into a pair where the first collection gets elements satisfying a given predicate, while the second gets those that do not. Consider the following example:

```
val (evens, odds) = listOf(1, 2, 3, 4, 5).partition { it % 2 }
```

```
== 0 }
println(evens) // [2, 4]
println(odds) // [1, 3, 5]
```

Unlike `filter()` and its varieties, `partition()` always returns a pair of lists even when applied to a sequence:

```
val seq = generateSequence(100) { if (it == 0) null else it/3 }
val (evens, odds) = seq.partition { it % 2 == 0 }
println(evens) // [100, 0]
println(odds) // [33, 11, 3, 1]
```

Note that `partition()` is not supported for maps.

Transformation

Various transformation functions included to the Kotlin standard library give you an ability to produce new collection by changing each element of existing ones according to a given rule and then combining the results in some way. These function can be divided into three basic categories: mapping, flattening, and association.

Mapping transformation applies a given function to each element of the original collection: the results then become elements of the new collection. The basic function of this kind is `map()`, which can be applied to any collection object, including arrays, iterables, sequences, and maps. The result is a sequence when applied to a sequence, and a list otherwise:

```
println(setOf("red", "green", "blue").map { it.length }) // [3, 5, 4]
println(listOf(1, 2, 3, 4).map { it*it }) // [1, 4, 9, 16]
println(byteArrayOf(10, 20, 30).map { it.toString(16) }) // [a, 14, 1e]
// 50, 16, 5, 1, 0
val seq = generateSequence(50) { if (it == 0) null else it / 3 }
println(seq.map { it*3 }.toList()) // [150, 48, 15, 3, 0]
```

You may also use the `mapIndexed()` function if your transformation needs to take element indices into account:

```
// [(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
println(List(6) { it*it }.mapIndexed { i, n ->i to n })
```

The `map()` and `mapIndexed()` functions also have variants that automatically filter out null values in the resulting collection. Semantically, they are similar to calling `filterNotNull()` after `map()` or `mapIndexed()`:

```
println(
    arrayOf("1", "red", "2", "green", "3").mapNotNull {
it.toIntOrNull() }
) // [1, 2, 3]
println(
    listOf("1", "red", "2", "green", "3").mapIndexedNotNull {
i, s ->
s.toIntOrNull()?.let { i to it }
}
) // [(0, 1), (2, 2), (4, 3)]
```

IDE Tips: IntelliJ can detect and simplify redundant usages of `mapNotNull()`/`mapIndexedNotNull()` suggesting to replace them by `map()`/`mapIndexed()` calls (see *Figure 7.5* for an example). On top of that, it warns you about explicit combination of `map()` and `filterNotNull()` calls which can be simplified to `mapNotNull()`. You can see an example on *Figure 7.6*.

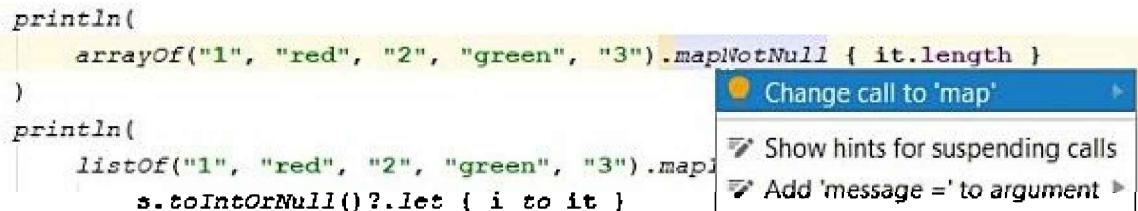


Figure 7.5: Simplifying redundant call to the `mapNotNull()` function

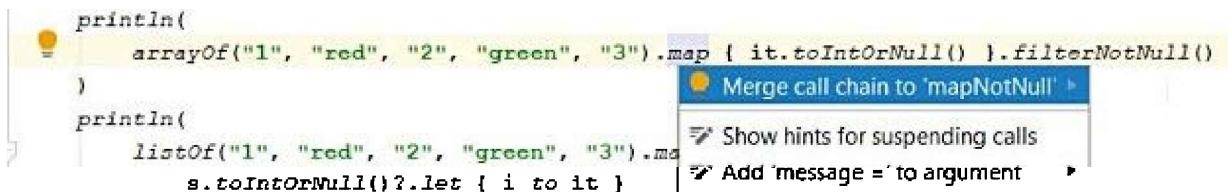


Figure 7.6: Merging `map-filterNotNull` chained call

The `map()` function can be applied to maps, in which case, the transformation takes

map entries as its input and produces a list. Additionally, you can use mapKeys() / mapValues() functions that transform only the keys or values, respectively, and return a new map:

```
val map = mapOf("I" to 1, "V" to 5, "X" to 10, "L" to 50)
// [I 1, V 5, X 10, L 50]
println(map.map { "${it.key} ${it.value}" })
// {i=1, v=5, x=10, l=50}
println(map.mapKeys { it.key.toLowerCase() })
// {I=1, V=5, X=a, L=32}
println(map.mapValues { it.value.toString(16) })
```

Each of the mapXXX() function also comes with a version which puts resulting element into some existing mutable collection rather than creating a new one. Similar to filters, these functions contain To in their names:

```
val result = ArrayList<String>()
listOf(1, 2, 3).mapTo(result) { it.toString() }
arrayOf("one", "two", "three").mapIndexedTo(result) { i, s ->
    "${i + 1}: $s"
}
sequenceOf("100", "?", "101", "?", "110").mapNotNullTo(result) {
    it.toIntOrNull(2)?.toString()
}
println(result) // [1, 2, 3, 1: s, 2: s, 3: s, 4, 5, 6]
```

The flattening operations transform each element of the original collection into a new collection and then glue resulting collection together. This kind of transformation is implemented by the flatMap() function, which produces a sequence when applied to a sequence and a list when applied to any other collection:

```
// [a, b, c, d, e, f, g, h, i]
println(setOf("abc", "def", "ghi").flatMap { it.asIterable() })
// [1, 2, 3, 4]
println(listOf(1, 2, 3, 4).flatMap { listOf(it) })
// [1, 1, 2, 1, 2, 3]
Array(3) { it + 1 }.flatMap { 1..it }
```

The `flatten()` function can be applied to any collection whose elements are collections themselves to glue them into a single object. It can be considered a simplified version of `flatMap()` with trivial transformation:

```
println(  
    listOf(listOf(1, 2), setOf(3, 4), listOf(5)).flatten()  
) // [1, 2, 3, 4, 5]  
println(Array(3) { arrayOf("a", "b") }.flatten()) // [a, b, a,  
b, a, b]  
println(  
    sequence {  
        yield(sequenceOf(1, 2))  
        yield(sequenceOf(3, 4))  
    }.flatten().toList()  
) // [1, 2, 3, 4]
```

IDE Tips: The IntelliJ plugin can detect trivial calls to `flatMap` suggesting to replace them with `flatten` (see Figure 7.7).

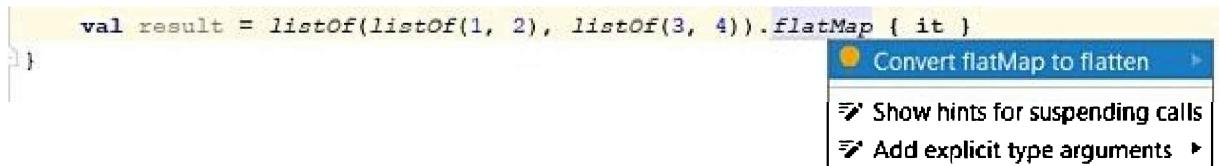


Figure 7.7: Replacing trivial `flatMap()` call with `flatten()`

Similar to `map()`, the `flatMap()` function has a version that appends resulting elements to an existing collection:

```
val result = ArrayList<String>()  
listOf(listOf("abc", "def"), setOf("ghi"))  
.flatMapTo(result) { it }  
sequenceOf(sequenceOf(1, 2), sequenceOf(3, 4))  
.flatMapTo(result) { it.map { "$it" } }  
println(result) // [abc, def, ghi, 1, 2, 3, 4]
```

One more transformation kind we'd like to cover in this section is an association that allows you to build maps based on a given transformation function and using original collection elements as either map keys or map values. The first case is implemented by the `associateWith()` function, which generates map values using original collection

as a source of keys:

```
println(
    listOf("red", "green", "blue").associateWith { it.length }
) // {red=3, green=5, blue=4}
println(
    generateSequence(1) { if (it > 50) null else it*3 }
    .associateWith { it.toString(3) }
) // {1=1, 3=10, 9=100, 27=1000, 81=10000}
```

Note that the `associateWith()` function is not applicable to arrays.

The `associateBy()` function similarly treats collection elements as values and uses supplied transformation function to produce map keys. Note that if there are multiple values corresponding to a single key, only one is retained in the resulting map:

```
// {3=red, 5=green, 4=blue}
println(listOf("red", "green", "blue").associateBy { it.length
})
// {1=15, 2=25, 3=35}
println(intArrayOf(10, 15, 20, 25, 30, 35).associateBy { it/10
})
// {1=1, 10=3, 100=9, 1000=27, 10000=81}
println(
    generateSequence(1) { if (it > 50) null else it*3 }
    .associateBy { it.toString(3) }
)
```

Finally, the `associate()` function transform collection element to produce both a key and value:

```
println(
    listOf("red", "green", "blue")
        .associate { it.toUpperCase() to it.length }
) // {RED=3, GREEN=5, BLUE=4}
println(
    intArrayOf(10, 15, 20, 25, 30, 35).associate { it to
    it/10 }
) // {10=1, 15=1, 20=2, 25=2, 30=3, 35=3}
println(
```

```

generateSequence(1) { if (it > 50) null else it*3 }
    .associate {
        val s = it.toString(3)
        "3^${s.length - 1}" to s
    }
) // {3^0=1, 3^1=10, 3^2=100, 3^3=1000, 3^4=10000}

```

Similar effect can also be achieved by `associateBy()` overloads which take separate transformation function for keys and values:

```

println(
    listOf("red", "green", "blue").associateBy(
        keySelector = { it.toUpperCase() },
        valueTransform = { it.length }
    )
) // {RED=3, GREEN=5, BLUE=4}

```

Association functions also have To variants (such as `associateByTo()`), which put produced entries into an existing mutable map.

Extracting subcollections

In the filtering section, we've discussed a set of functions that will allow you to extract a part of original collection retaining only elements satisfying a certain condition. In this section, we'll consider functions that serve a similar purpose but extract collection parts based on other criteria.

In the *Basic operations* section, we've mentioned the `subList()` function which gives you a view of a list segment. The `slice()` function performs a similar task but uses integer range instead of a pair of integers to represent segment bounds. On top of it, the `slice()` function can be applied to arrays as well as lists:

```

// 0, 1, 4, 9, 16, 25
println(List(6) { it*it }.slice(2..4)) // [4, 9, 16]

// 0, 1, 8, 27, 64, 125
println(Array(6) { it*it*it }.slice(2..4)) // [8, 27, 64]

```

In the case of list, it works similar to the `subList()` method producing a wrapper of the original collection which reflects a given segment. In the case of array, the result is a new list containing array elements with specified indices.

If you want to extract array segment as another array, you can use `sliceArray()` instead:

```
val slice = Array(6) { it*it*it
}.sliceArray(2..4).contentToString()
```

There is also a more general version of `slice()`/`sliceArray()` which takes an iterable of integers and uses them as indices. In other words, it allows you to extract an arbitrary subsequence of the original list or array:

```
println(List(6) { it*it }.slice(listOf(1, 2, 3))) // [1, 4, 9]
println(Array(6) { it*it*it }.slice(setOf(1, 2, 3))) // [1, 8,
27]
println(
    Array(6) { it*it*it }.sliceArray(listOf(1, 2,
3)).contentToString()
) // [1, 8, 27]
```

The `take()`/`takeLast()` functions are used to extract a given number of iterable or array elements starting from the first or the last one, respectively:

```
println(List(6) { it*it }.take(2))           // [0, 1]
println(List(6) { it*it }.takeLast(2))         // [16, 25]
println(Array(6) { it*it*it }.take(3))         // [0, 1, 8]
println(Array(6) { it*it*it }.takeLast(3))      // [27, 64, 125]
```

The `take()` function can also be applied to a sequence; in this case, it returns a new sequence containing first elements of the original one:

```
val seq = generateSequence(1) { if (it > 100) null else it*3 }
println(seq.take(3).toList()) // [1, 3, 9]
```

The `drop()`/`dropLast()` functions can be considered a complement to `take()`/`takeLast()`: they return the elements remaining when a given number of first/last ones is removed:

```
println(List(6) { it*it }.drop(2))           // [4, 9, 16, 25]
println(List(6) { it*it }.dropLast(2))          // [0, 1, 4, 9]
println(Array(6) { it*it*it }.drop(3))          // [27, 64,
125]
println(Array(6) { it*it*it }.dropLast(3))        // [0, 1, 8]
```

```
val seq = generateSequence(1) { if (it > 100) null else it*3 }
println(seq.drop(3).toList()) // [27, 81, 243]
```

The `take`/`drop` operations also come in versions that take a predicate on collection elements rather than a number: these versions take/drop elements only up to the first one violating a given condition:

```
val list = List(6) { it * it }
println(list.takeWhile { it < 10 }) // [0, 1, 4, 9]
println(list.takeLastWhile { it > 10 }) // [16, 25]
println(list.dropWhile { it < 10 }) // [16, 25]
println(list.dropLastWhile { it > 10 }) // [0, 1, 4, 9]

val seq = generateSequence(1) { if (it > 100) null else it*3 }
println(seq.takeWhile { it < 10 }.toList()) // [1, 3, 9]
println(seq.dropWhile { it < 10 }.toList()) // [27, 81, 243]
```

The `chunked()` functions introduced in Kotlin 1.2 allow you to split an iterable or sequence into lists (called chunks) whose size does not exceed a given threshold. The simplest form of `chunked()` takes just a maximum chunk size:

```
// 0, 1, 4, 9, 16, 25, 36, 49, 64, 81
val list = List(10) { it*it }
println(list.chunked(3)) // [[0, 1, 4], [9, 16, 25], [36, 49, 64], [81]]

// 1, 3, 9, 27, 81, 243, 729
val seq = generateSequence(1) { if (it > 300) null else it*3 }
println(seq.chunked(3).toList()) // [[1, 3, 9], [27, 81, 243], [729]]
```

Note `chunked()` returns a list of chunks when applied to an iterable, and a sequence of chunks when applied to a sequence.

The general version allows you to specify a function that transforms each chunk into an arbitrary value. The result is a list or sequence composed of transformation results. The following code replaces each chunk from the preceding example by sum of its elements:

```
// 0, 1, 4, 9, 16, 25, 36, 49, 64, 81
val list = List(10) { it*it }
```

```
println(list.chunked(3) { it.sum() })           // [5, 50, 149,  
81]  
  
// 1, 3, 9, 27, 81, 243, 729  
val seq = generateSequence(1) { if (it > 300) null else it*3 }  
println(seq.chunked(3) { it.sum() }.toList()) // [13, 351,  
729]
```

The `windowed()` function, also introduced in Kolin 1.2, allows you to extract all segments of a given slide visible through a kind of sliding window. Like `chunked()`, it produces a list of lists when applied to an iterable and a sequence of lists when applied to a sequence:

```
// 0, 1, 4, 9, 16, 25  
val list = List(6) { it*it }  
// [[0, 1, 4], [1, 4, 9], [4, 9, 16], [9, 16, 25]]  
println(list.windowed(3))  
  
// 1, 3, 9, 27, 81, 243  
val seq = generateSequence(1) { if (it > 100) null else it*3 }  
// [[1, 3, 9], [3, 9, 27], [9, 27, 81], [27, 81, 243]]  
println(seq.windowed(3).toList())
```

Similar to `chunked()`, you may supply a transformation function that aggregates elements of each window:

```
// 0, 1, 4, 9, 16, 25  
val list = List(6) { it*it }  
println(list.windowed(3) { it.sum() })           // [5, 14, 29,  
50]  
  
// 1, 3, 9, 27, 81, 243  
val seq = generateSequence(1) { if (it > 100) null else it*3 }  
println(seq.windowed(3) { it.sum() }.toList()) // [13, 39,  
117, 351]
```

Additionally, you may specify optional parameters that affect sliding window behavior:

- `step`: A distance between indices of first elements in a pair of adjacent windows (1 by default).
- `partialWindows`: This includes windows of smaller size at the end of the

collection (false by default).

Let's see an example using these options:

```
// 0, 1, 4, 9, 16, 25
val list = List(6) { it*it }
// Only elements with even indices (0 and 2) produce windows:
// [[0, 1, 4], [4, 9, 16]]
println(list.windowed(3, step = 2))
// Added two partial windows at the end:
// [[0, 1, 4], [1, 4, 9], [4, 9, 16], [9, 16, 25], [16, 25],
[25]]
println(list.windowed(3, partialWindows = true))
```

There is also a separate function for building two-element windows: `zipWithNext()`. Unlike `windowed()`, it produces lists and sequences of pairs rather than of lists:

```
// 0, 1, 4, 9, 16, 25
val list = List(6) { it*it }
// [(0, 1), (1, 4), (4, 9), (9, 16), (16, 25)]
println(list.zipWithNext())

// 1, 3, 9, 27, 81, 243
val seq = generateSequence(1) { if (it > 100) null else it*3 }
// [(1, 3), (3, 9), (9, 27), (27, 81), (81, 243)]
println(seq.zipWithNext().toList())
```

Accordingly, its aggregation version uses a function that takes a pair of collection elements instead of a list:

```
// [0, 4, 36, 144, 400]
println(List(6) { it*it }.zipWithNext { a, b -> a * b })
```

Ordering

The standard library includes functions that sort collection elements according to a given ordering. The simplest of them is the `sorted()` function that can be applied to any array/iterable/sequence of comparable values to sort them based on their natural ordering. The `sortDescending()` function is similar but sorts elements in reverse:

```
println(intArrayOf(5, 8, 1, 4, 2).sorted()) // [1, 2, 4, 5, 8]
```

```

println(
    intArrayOf(5, 8, 1, 4, 2).sortedDescending()
)
// [8, 5, 4, 2, 1]

println(
    listOf("abc", "w", "xyz", "def", "hij").sorted()
)
// [abc, def, hij, w, xyz]

println(
    listOf("abc", "w", "xyz", "def", "hij").sortedDescending()
)
// [xyz, w, hij, def, abc]

// 1, -3, 9, -27, 81
val seq = generateSequence(1) { if (it < 50) -it * 3 else null
}
println(seq.sorted().toList()) // [-27, -3,
1, 9, 81]
println(seq.sortedDescending().toList()) // [81, 9, 1,
-3, -27]

```

These functions return `Sequence` when applied to a sequence. Note, though, that the sequence returned is stateful and sorts the entire collection on the first attempt to access its elements.

When applied to an array or iterable, the result is always a `List`. For arrays, you can use a similar pair of `sortedArray()`/`sortedArrayDescending()` functions which return array instead of the list.

If collection elements are not comparable, you can still sort them by using one of `sorted()` alternatives which allows you to specify a custom ordering: `sortedBy()`/`sortedWith()`. The convention is similar to the one we've seen for the `min()`/`max()` aggregation functions: `sortedBy()` takes a function that converts collection elements to comparables, while `sortedWith()` takes a comparator. There is also a reversed version of `sortedBy()`, which is called `sortedByDescending()`:

```

class Person(val firstName: String,
val familyName: String,
val age: Int) {
    override fun toString() = "$firstName $familyName: $age"
}

valPerson.fullNameget() = "$firstName $familyName"
valPerson.reverseFullNameget() = "$familyName $firstName"

```

```

val FULL_NAME_COMPARATOR = Comparator<Person>{ p1, p2 ->
    p1.fullName.compareTo(p2.fullName)
}
val REVERSE_FULL_NAME_COMPARATOR = Comparator<Person>{ p1, p2
->
    p1.reverseFullName.compareTo(p2.reverseFullName)
}
fun main() {
    val persons = listOf(
        Person("Brook", "Hudson", 25),
        Person("Silver", "Watts", 30),
        Person("Dane", "Hall", 19),
        Person("Val", "Ortiz", 28)
    )
    println(persons.sortedWith(FULL_NAME_COMPARATOR))
    println(persons.sortedWith(FULL_NAME_COMPARATOR))
    println(persons.sortedWith(REVERSE_FULL_NAME_COMPARATOR))
    println(persons.sortedWith(REVERSE_FULL_NAME_COMPARATOR))
    println(persons.sortedBy { it.age })
    println(persons.sortedByDescending { it.age
})
}

```

All sorting functions we've seen so far put sorted elements into a new collection leaving the original one untouched. In the case of arrays and mutable lists, though, we can modify the original collection instead and sort its elements in place. This is implemented by `sort()` and `sortDescending()` functions:

```

val array = intArrayOf(4, 0, 8, 9, 2).apply { sort() }
println(array.contentToString()) // [0, 2, 4, 8, 9]
val list = arrayListOf("red", "blue", "green").apply { sort()
}
println(list) // [blue, green, red]

```

A separate group of functions can be used to reverse elements in iterables and arrays. The basic case is handled by the `reversed()` function, which returns a new list with original elements reversed:

```
println(intArrayOf(1, 2, 3, 4, 5).reversed()) //
```

```
[5, 4, 3, 2, 1]
println(listOf("red", "green", "blue").reversed()) // [blue,
green, red]
```

For arrays, you can also use the `reversedArray()` function, which produces a new array instead of list.

The `reverse()` function can be used to reverse elements of a mutable list or array without creating a new collection (note the similarity with `sort()` versus `sorted()/sortedArray()`):

```
val array = intArrayOf(1, 2, 3, 4, 5).apply { reverse()
}.contentToString()
println(array) // [5, 4, 3, 2, 1]

val list = arrayListOf("red", "green", "blue").apply {
reverse() }
println(list) // [blue, green, red]
```

The `asReversed()` function is similar to `reversed()` in a sense that it returns a new list. The list produced is, however, just a wrapper over the original. Both lists share the same data, which makes `asReversed()` more efficient in terms of memory usage. When applied to a mutable list, it returns a mutable wrapper. Changes in either list are automatically reflected in the other (unlike collection produced by the `reversed()` function):

```
val list = arrayListOf("red", "green", "blue")
valreversedCopy = list.reversed()
valreversedMirror = list.asReversed()
list[0] = "violet"
println(list)           // [violet, green, blue]
println(reversedCopy)   // [blue, green, red]
println(reversedMirror) // [blue, green, violet]
```

Note that `asReversed()` is available only for lists.

One more function we'd like to mention in this section is `shuffled()`. When applied to an iterable, it produces a new list with original elements rearranged in a random order:

```
println(listOf(1, 2, 3, 4, 5).shuffled())
```

Mutable lists can be similarly modified in-place using `shuffle()`:

```
arrayListOf(1, 2, 3, 4, 5).shuffle()
```

Note that sequences and arrays do not support either of these functions.

Files and I/O streams

In this section, we'll address the part of the Kotlin standard library that deals with input/output operations. The features we'll cover are based around the existing Java API for files, I/O streams, and URLs; in this regard, the Kotlin standard library provides a set of useful extension functions and properties that simplify the usage of I/O-related classes already present in the JDK.

Stream utilities

The Kotlin standard library includes a bunch of helper extensions for Java I/O streams. These functions simplify access to stream content and implement some more complex patterns such as copying and automatic stream finalization. In this section, we'll have a closer look at these features.

The following functions allow you to retrieve the entire stream content:

```
fun InputStream.readBytes(): ByteArray  
fun Reader.readText(): String  
fun Reader.readLines(): Line<String>
```

Mind the difference between the latter two functions and the `readLine()` method of the `BufferedReader` class: while `readLine()` retrieves a single line from the stream, `readText()`/`readLines()` reads the stream till the end and returns its entire content as either a single string, or a list of individual lines. Consider the following example:

```
import java.io.*  
  
fun main() {  
    FileWriter("data.txt").use { it.write("One\nTwo\nThree") }  
    // One  
    FileReader("data.txt").buffered().use {  
        println(it.readLine())  
        // One Two Three  
        FileReader("data.txt").use {  
            println(it.readText().replace('\n', ' '))  
        }  
    }  
}
```

```
// [One, Two, Three]
println(FileReader("data.txt").readLines())
}
```

Note that unlike `readText()`, the `readLines()` function automatically closes its stream on completion.

Kotlin allows direct iteration over buffered streams, although the API is a bit different for binary and text data. In the case of `BufferedOutputStream`, we have the `iterator()` function which, in particular, allows us to use such streams in the `for` loop to iterate over individual bytes:

```
FileInputStream("data.bin").buffered().use {
    var sum = 0
    for (byte in it) sum += byte
}
```

`BufferedReader`, on the other hand, provides the `lineSequence()` function which gives you a sequence over its lines:

```
FileReader("data.bin").buffered().use {
    for (line in it.lineSequence()) println(line)
}
```

Similar capabilities, albeit in a more indirect form, are available for an arbitrary Reader instance. The `forEachLine()` and `useLines()` functions allow you to iterate over individual lines. When using them, you don't have to worry about closing the stream since they perform it automatically:

```
import java.io.*
fun main() {
    FileWriter("data.txt").use { it.write("One\nTwo\nThree") }
    // One, Two, Three
    FileReader("data.txt").useLines {
        println(it.joinToString())
    }
    // One/Two/Three
    FileReader("data.txt").forEachLine { print("$it/") }
}
```

The difference is that `forEachLine()`'s lambda accepts current line and is invoked upon each iteration, while `useLines()`'s lambda takes a sequence over all lines.

It's also possible to transfer data between streams using the `copyTo()` function, which

has two overloaded versions for binary and text streams:

```
fun InputStream.copyTo(  
    out: OutputStream,  
bufferSize: Int = DEFAULT_BUFFER_SIZE  
) : Long  
fun Reader.copyTo(out: Writer,  
bufferSize: Int = DEFAULT_BUFFER_SIZE) : Long
```

The return value gives you an actual number of bytes or characters copied. The following sample demonstrates the usage of `copyTo()`:

```
import java.io.*  
fun main() {  
    FileWriter("data.txt").use { it.write("Hello") }  
    val writer = StringWriter()  
    FileReader("data.txt").use { it.copyTo(writer) }  
    println(writer.buffer) // Hello  
    val output = ByteArrayOutputStream()  
    FileInputStream("data.txt").use { it.copyTo(output) }  
    println(output.toString("UTF-8")) // Hello  
}
```

One more function we'll consider in this section will provide a safe way to work with streams and other resources that need explicit finalization. The `use()` function can be invoked on any instance of the `java.io.Closeable` type (and `java.lang.AutoCloseable` since Kotlin 1.2): it then executes a supplied lambda, properly finalizes the resource (whether an exception is thrown or not), and then returns the result of the lambda:

```
val lines = FileReader("data.bin").use { it.readLines() }
```

Java versus Kotlin: This function serves the same purpose as the `try-with-resources` statement introduced in Java 7.

The preceding code is roughly equivalent to the explicit `try` block:

```
val reader = FileReader("data.bin")  
val lines = try {  
    reader.readLines()  
} finally {
```

```
    reader.close()
}
```

IDE Tips: IntelliJ can automatically detect such `try` blocks and suggest converting them into the `use()` function calls using the Alt + Enter menu on the `try` keyword (as shown on *Figure 7.8*):

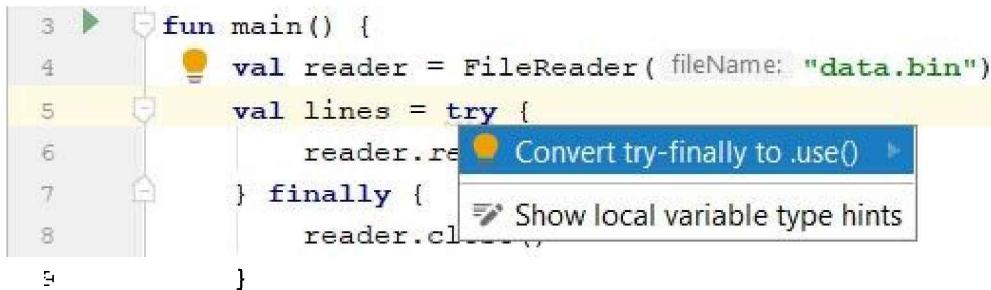


Figure 7.8: Converting explicit try block into the use() call

Creating streams

The standard library includes a set of functions simplifying creation of Java I/O streams. In this section, we'll see basic cases.

Using `bufferedReader()`/`bufferedWriter()` extensions, you can create a `BufferedReader`/`BufferedWriter` instance for a particular `File` object:

```
import java.io.File
fun main() {
    val file = File("data.txt")
    file.bufferedWriter().use { it.write("Hello!") }
    file.bufferedReader().use { println(it.readLine()) } // Hello!
}
```

There is also a similar pair of `reader()`/`writer()` extension functions, which create a `FileReader`/`FileWriter` object without bufferization.

The `printWriter()` function creates a `PrintWriter` instance suitable for a formatted output.

The `reader/writer-related` functions allow you to optionally specify encoding `charset` (defaulting to `UTF-8`), and the buffered versions have an extra optional parameter for the

buffer size. The default buffer size is given by the `DEFAULT_BUFFER_SIZE` constant, which currently corresponds to 8 kilobytes:

```
file.writer(charset = Charsets.US_ASCII).use {  
    it.write("Hello!") }  
file.bufferedReader(  
    charset = Charsets.US_ASCII,  
    bufferSize = 100  
) .use { println(it.readLine()) }
```

The `Charsets` object contains a set of constants for some standard charsets such as US-ASCII or different UTF variants.

If you want to work with a binary file, you can similarly use `InputStream()`/`OutputStream()` functions to create an appropriate stream instance:

```
import java.io.File  
fun main() {  
    val file = File("data.bin")  
    file.outputStream().use { it.write("Hello!".toByteArray()) }  
    file.inputStream().use {  
        println(String(it.readAllBytes())) }  
    } // Hello!  
}
```

Several functions give you an ability to create I/O streams based on the content of a `String` or `ByteArray`. The `byteInputStream()` creates an `ByteArrayInputStream` instance with a string as its source:

```
println("Hello".byteInputStream().read().toChar()) // H  
println("Hello".byteInputStream(Charsets.US_ASCII).read().toChar()) // H  
The reader() function similarly creates a StringReader instance:  
println("One\nTwo".reader().readLines()) // [One, Two]
```

The `InputStream()` function similarly constructs an instance `ByteArrayInputStream` using a byte array as its source:

```
println(byteArrayOf(10, 20, 30).inputStream().read())
```

It's also possible to use a portion of the byte array using an overloaded version of `inputStream()`, which takes an offset and a portion size:

```
val bytes = byteArrayOf(10, 20, 30, 40, 50)
println(
    bytes.inputStream(2, 2).readBytes().contentToString()
) // [30, 40]
```

The standard library also includes some extensions simplifying stream piping. The following set of functions can be used to construct a `Reader`, a `BufferedReader` or a `BufferedInputStream` objects based on the general instance of the `InputStream` class:

```
fun InputStream.reader(
    charset: Charset = Charsets.UTF_8
): InputStreamReader
fun InputStream.bufferedReader(
    charset: Charset = Charsets.UTF_8
): BufferedReader
fun InputStream.buffered(
    bufferSize: Int = DEFAULT_BUFFER_SIZE
): BufferedInputStream
```

Similar functions (named `writer()`, `bufferedWriter()` and `buffered()`) are also available for `OutputStream` in which case they pipe it to a `Writer`, a `BufferedWriter`, or a `BufferedOutputStream` instance respectively. The following example gives you a taste of them:

```
import java.io.FileInputStream
import java.io.FileOutputStream
fun main() {
    val name = "data.txt"
    FileOutputStream(name).bufferedWriter().use {
        it.write("One\nTwo" )
    }
    val line = FileInputStream(name).bufferedReader().use {
        it.readLine()
    }
    println(line) // One
```

```
}
```

The buffered() function is also defined for Reader and Writer:

```
fun Reader.buffered(bufferSize: Int = DEFAULT_BUFFER_SIZE):  
    BufferedReader  
fun Writer.buffered(bufferSize: Int = DEFAULT_BUFFER_SIZE):  
    BufferedWriter
```

URL utilities

The Kotlin library provides a couple of helper functions for retrieving data over network connections associated with URL objects:

```
fun URL.readText(charset: Charset =Charsets.UTF_8): String  
fun URL.readBytes(): ByteArray
```

The readText() function reads the entire content of an input stream corresponding to the URL instance using the specified charset. The readBytes() function similarly retrieves the content of a binary stream as an array of bytes.

Since both functions load an entire stream content blocking the calling thread till completion, they shouldn't be used to download large files.

Accessing file content

The Kotlin standard library allows you to access file content using special functions without an explicit mention of I/O streams. These functions are helpful in cases such as reading/writing an entire file, appending data to existing file, or line-by-line processing of a file.

The following functions allow you to manipulate a text content:

- `readText()` : Reads the entire content of a file as a single string;
- `readLines()` : Reads the entire content of a file splitting it by line separators and returning a list of strings;
- `writeText()` : Sets the file content to a given String rewriting it if necessary;
- `appendText()` : Adds a specified string to the content of a given file.

The usage of these functions is demonstrated by the following example:

```
import java.io.File  
fun main() {
```

```
val file = File("data.txt")
file.writeText("One")
println(file.readText())      // One
file.appendText("\nTwo")
println(file.readLines())    // [One, Two]
file.writeText("Three")
println(file.readLines())    // [Three]
}
```

Each of the text-related functions may accept an optional parameter of the Charset type specifying text encoding.

For the binary files, you may use similar functions that work with byte arrays instead of strings:

```
import java.io.File
fun main() {
    val file = File("data.bin")
    file.writeBytes(byteArrayOf(1, 2, 3))
    println(file.readBytes().contentToString()) // [1, 2, 3]
    file.appendBytes(byteArrayOf(4, 5))
    println(file.readBytes().contentToString()) // [1, 2, 3,
4, 5]
    file.writeBytes(byteArrayOf(6, 7))
    println(file.readBytes().contentToString()) // [6, 7]
}
```

Another group of functions allows you to process a file content in blocks without reading it entirely. This is helpful for handling large files which can't be efficiently put in memory as a whole.

The `forEachLine()` function allows you to process the text content line by line without reading an entire file. The following examples demonstrates how it works:

```
import java.io.File
fun main() {
    val file = File("data.txt")
    file.writeText("One\nTwo\nThree")
    file.forEachLine { print("/$it") } // /One/Two/Three
}
```

The `useLines()` function passes a line sequence to the given lambda which can compute some result, which is then returned by the `useLines()` call:

```
import java.io.File
fun main() {
    val file = File("data.txt")
    file.writeText("One\nTwo\nThree")
    println(file.useLines { lines ->lines.count { it.length> 3
} }) // 1
}
```

Similar to other text-related file functions, you may pass the optional `Charset` parameter to `forEachLine()` and `useLines()`.

To process a binary file, you can use the `forEachBlock()` function. Its lambda accepts a `ByteArray` buffer and an integer that tells how many bytes were read on the current iteration. The following code, for example, outputs the sum of all bytes in the `data.bin` file:

```
import java.io.File
fun main() {
    val file = File("data.bin")
    var sum = 0
    file.forEachBlock { buffer, bytesRead ->
        (0 until bytesRead).forEach { sum += buffer[it] }
    }
    println(sum)
}
```

By default the buffer size is implementation-dependent, but you may specify it as an optional `blockSize` parameter. Note that the buffer size can't be smaller than some implementation-specific threshold. In Kotlin 1.3, the default and minimum buffer sizes are 4096 and 512 bytes, respectively.

File system utilities

In this section, we'll discuss standard library functions that simplify file system operations such as copying and removing files as well as traversing the directory structure.

The `deleteRecursively()` function allows you to delete a given file together with

all its children, including nested directories. The result is true if deletion completes successfully, and false otherwise. In the latter case, the deletion may be partial, for example, if some nested directories can't be deleted. This function serves as a counterpart for the `makedirs()` method present in the Java API:

```
import java.io.File
fun main() {
    File("my/nested/dir").mkdirs()
    val root = File("my")
    println("Dir exists: ${root.exists()}") // true
    println("Simple delete: ${root.delete()}") // false
    println("Dir exists: ${root.exists()}") // true
    println("Recursive delete: ${root.deleteRecursively()}") // true
    println("Dir exists: ${root.exists()}") // false
}
```

The `copyTo()` function copies its receiver to another file and returns the copy:

```
import java.io.File
fun main() {
    val source = File("data.txt")
    source.writeText("Hello")
    val target = source.copyTo(File("dataNew.txt"))
    println(target.readText()) // Hello
}
```

By default the target file is not overwritten, so if it already exists, the `copyTo()` function throws `FileAlreadyExistsException`. You can, however, specify an optional `overwrite` parameter to enforce the file copying:

```
import java.io.File
fun main() {
    val source = File("data.txt").also { it.writeText("One") }
    val target = File("dataNew.txt").also {
```

```

it.writeText("Two") }
    source.copyTo(target, overwrite = true)
    println(target.readText()) // One
}

```

The `copyTo()` function may be applied to directories as well; however, it doesn't copy its files and subdirectories but simply creates an empty directory corresponding to the target path. If you want to copy the directory together with its content, there is a separate `copyRecursively()` function:

```

import java.io.File
fun main() {
    File("old/dir").mkdirs()
    File("old/dir/data1.txt").also { it.writeText("One") }
    File("old/dir/data2.txt").also { it.writeText("Two") }
    File("old").copyRecursively(File("new"))
    println(File("new/dir/data1.txt").readText()) // One
    println(File("new/dir/data2.txt").readText()) // Two
}

```

Similar to `copyTo()`, this function allows you to specify overwriting policy using the `overwrite` parameter (false by default). Additionally, you can set an action which is invoked on `IOException` when copying a particular file. This can be done using an optional `onError` parameter, which accepts a lambda of the type `(File, IOException) ->OnErrorHandler`. The result value determines how the `copyRecursively()` function would deal with a problematic file:

- `SKIP`: Skip the file and continue the copying;
- `TERMINATE`: Stop copying.

Being the last parameter, the `onError` lambda can be passed outside parentheses:

```

File("old").copyRecursively(File("new")) { file, ex -
>OnErrorAction.SKIP }

```

The default action is to rethrow a caught `IOException` instance back to the caller.

The `walk()` function implements traversal of the directory structure according to the depth-first search algorithm. The optional parameter specifies traversal direction:

- `TOP_DOWN`: Visit parent before children (default value);

- `BOTTOM_UP`: Visit children before parent.

The return value is a sequence of `File` instances. The following example demonstrates the usage of different traversal modes:

```
import java.io.File
import kotlin.io.FileWalkDirection.*
fun main() {
    File("my/dir").mkdirs()
    File("my/dir/data1.txt").also { it.writeText("One") }
    File("my/dir/data2.txt").also { it.writeText("Two") }
    println(File("my").walk().map { it.name }.toList())
    println(File("my").walk(TOP_DOWN).map { it.name
}.toList())
    println(File("my").walk(BOTTOM_UP).map { it.name
}.toList())
}
```

You can also use `walkTopDown()` and `walkBottomUp()` functions instead of `walk(TOP_DOWN)` and `walk(BOTTOM_UP)` calls, respectively.

The sequence returned by the `walk()` function belongs to the special `FileTreeWalk` class. Besides the common sequence functionality, this class allows you to specify additional traversal options. The `maxDepth()` function sets a maximum depth of the traversed subtree:

```
println(
    File("my").walk().maxDepth(1).map { it.name }.toList()
) // [my, dir]
```

The `onEnter()` and `onLeave()` functions set up actions performed when traversal enters and leaves a directory. The `onEnter()` call accepts a `(File) -> Boolean` lambda whose return value determines whether a directory (and its children) should be visited at all. The `onLeave()` call similarly accepts `(File) -> Unit` lambda. The `onFail()` function allows to specify an action that is called on `IOException` when trying to access the directory's children: the action takes a form of the `(File, IOException) -> Unit` lambda, which accepts problematic directory and a corresponding exception.

Since all four functions return the current instance of `FileTreeWalk`, they can be chained as shown in the following example:

```
println(  
File("my")  
.walk()  
.onEnter { it.name != "dir" }  
.onLeave { println("Processed: ${it.name}") }  
.map { it.name }  
.toList()
```

The preceding code would print the following:

```
Processed: my
```

```
[my]
```

Since the dir directory would be filtered out by the `onEnter()` action.

The default actions are as follows: always return true for `onEnter()`, do nothing for `onLeave()`, and throw an exception for `onFail()`. The maximum tree depth is `Int.MAX_VALUE` by default making it effectively unconstrained.

The `createTempFile()`/`createTempDir()` functions can be used to create a temporary file or directory, respectively:

```
valtmpDir = createTempDir(prefix = "data")  
valtmpFile = createTempFile(directory = tmpDir)
```

Both functions have the same set of parameters:

```
fun createTempDir(  
    prefix: String = "tmp",  
    suffix: String? = null,  
    directory: File? = null  
): File
```

The `createTempFile()` function is essentially the same as the JDK method `File.createTempFile()`.

Conclusion

In this chapter, we've got to know a major part of the Kotlin standard library aimed at manipulation of collections. We've got an understanding of collection types such as arrays, iterables, sequences, and maps. We have also discussed their basic API and operations covering various collection use cases such as accessing elements and subcollections, filtering, aggregation, transformations, and sorting. In the second part of

this chapter, we've taken a look at I/O utilities aimed at simplifying creation of streams, access to their data, and common file system operation such as deletion and copying.

In the next chapter, we'll revisit the subject of object-oriented programming and discuss how the concepts of class inheritance and delegation can be used in Kotlin applications.

Questions

1. Give an outline of collection types in Kotlin. What are the key differences from the Java collections library?
2. Which basic operations are provided by collection types?
3. Describe various ways to iterate over collection elements.
4. What common functions can be used to access collection elements?
5. What common aggregates are available in the Kotlin library?
6. Describe the `fold()` / `reduce()` operations.
7. What is the purpose of `all()` / `any()` / `none()` functions?
8. Describe collection filtering functions.
9. How one can extract a subcollection?
10. What standard transformations can be applied to collections? Describe features of mapping, flattening, and association.
11. Describe collection ordering utilities provided by the Kotlin standard library.
12. Describe stream creation and conversion utilities.
13. What functions can be used to access content of the file or I/O stream?
14. Describe the file system utility function.

CHAPTER 8

Understanding Class Hierarchies

This chapter continues the discussion of object-oriented aspects of Kotlin that were introduced in *Chapter 4, Working with Classes and Objects* and *Chapter 6, Using Special-Case Classes*. We will introduce the concept of class inheritance and explain how to define subclasses. We will also consider designing complex class hierarchies using abstract classes, interfaces and class delegation. The features of interest also include sealed classes that implement the concept of algebraic data types suited for definition of restricted class type hierarchies and type checking, enabling powerful Kotlin smart casts.

Structure

- Inheritance and overriding
- Type checking and casts
- Abstract classes
- Interfaces
- Sealed classes
- Delegation

Objective

Get an understanding of how inheritance and overriding works in Kotlin, and learn how to use Kotlin object-oriented capabilities to build class hierarchies.

Inheritance

In order to represent is-a relationship between domain concepts, most object-oriented languages use the concept of inheritance. When class A (a subclass or a derived class) inherits class B (a superclass or a base class) all instances of A are automatically considered instances of B. As a consequence of that, class A gets all members and extensions defined for B. This relation is transitive; if class B, in turn, inherits some

class C, A is also considered a subclass (albeit indirect) of C.

In Kotlin, like Java, classes support only single inheritance meaning that any class may not have more than one superclass. If you don't specify a superclass explicitly, the compiler automatically assumes that your class inherits from the built-in class Any. Thus, all classes in a given program form a well-defined inheritance tree, which is usually called a class hierarchy.

In the upcoming sections we shall discuss the basics of class inheritance in Kotlin: how to define a subclass, how superclass members are inherited and overridden, and the common methods that are available for any object via the Anyclass.

Declaring a subclass

To inherit from a given class you add its name preceded by: symbol after the primary constructor in your class definition:

```
open class Vehicle {  
    var currentSpeed = 0  
    fun start() {  
        println("I'm moving")  
    }  
    fun stop() {  
        println("Stopped")  
    }  
}  
open class FlyingVehicle : Vehicle() {  
    fun takeOff() {  
        println("Taking off")  
    }  
    fun land() {  
        println("Landed")  
    }  
}  
class Aircraft(val seats: Int) : FlyingVehicle()
```

Java vs. Kotlin: In Kotlin, there are no special keywords like extends and implements in Java. Instead, inheritance is always denoted by colon symbol (:). Please note the parentheses added after Vehicle and FlyingVehicle in the

definitions of their subclasses - this is a call to superclass constructor, where you put the necessary arguments to the super class initialization code.

You have probably noticed the `open` keyword near the `Vehicle` and `FlyingVehicle` definitions. This modifier marks corresponding classes as open for inheritance, thus, allowing them to serve as superclasses. The `Aircraft`, on the other hand, has no such modifier and by default is considered final. If you attempt to inherit from a final class, the compiler will report an error:

```
class Airbus(seats: Int) : Aircraft(seats) // Error: Aircraft  
is final
```

Java vs. Kotlin: Please note the difference between the default class behavior in Java and Kotlin:

In Java, any class is open by default and must be explicitly marked as final, if you want to forbid inheriting from it. However, in Kotlin, the default is final. If you want some class to be inheritable, you must declare it as open.

As seen in practice, classes that are not specifically designed with inheritance in mind, may suffer from the so-called fragile base class problem, when changes in a base class lead to an incorrect behavior in subclasses. This is because superclass no longer satisfies their assumptions. Hence, it is highly recommended to design and document inheritable classes carefully, making such assumptions explicit.

Instances of subclasses are also instances of their superclasses. They also inherit super class members:

```
val aircraft = Aircraft(100)  
val vehicle: Vehicle = aircraft // implicit cast to supertype  
vehicle.start() // calling Vehicle method  
vehicle.stop() // calling Vehicle method  
aircraft.start() // calling Vehicle method  
aircraft.takeOff() // calling FlyingVehicle  
method  
aircraft.land() // calling FlyingVehicle method  
aircraft.stop() // calling Vehicle method  
println(aircraft.seats) // accessing Aircraft own  
property
```

Certain kinds of classes have only limited support of inheritance. The data classes, for

example, are always final and can't be declared as open:

```
open data class Person(val name: String, val age: Int) //  
Error
```

Initially, it was forbidden to inherit data class from another class, but this limitation was removed in Kotlin 1.1.

On the other hand, inline classes can neither extend other classes, nor serve as superclasses themselves currently:

```
class MyBase  
open inline class MyString(val value: String) //  
Error  
inline class MyStringInherited(val value: String) : MyBase()  
// Error
```

Objects (including companions) can be freely inherited from open classes:

```
open class Person(val name: String, val age: Int) {  
    companion object : Person("Unknown", 0)  
}  
object JohnDoe : Person("John Doe", 30)
```

However, you cannot inherit from an object or declare it open, since each object is supposed to have only one instance.

A powerful feature of inheritance is a so-called, ad-hoc polymorphism, which allows you to provide different implementations of a superclass member for subclasses and choose them, depending on the actual instance class at runtime. In Kotlin, this can be achieved by overriding a member of superclass. Let's look at the following classes:

```
open class Vehicle {  
    open fun start() {  
        println("I'm moving")  
    }  
    fun stop() {  
        println("Stopped")  
    }  
}  
class Car : Vehicle() {
```

```

override fun start() {
    println("I'm riding")
}
}

class Boat : Vehicle() {
    override fun start() {
        println("I'm sailing")
    }
}

```

The `Vehicle` class provides common implementation of `start()` method, which is then overridden by its inheritors - `Car` and `Boat`. Please note, that the `start()` method in `Vehicle` class is marked as open, which makes it overridable in subclasses, while its implementations in `Car` and `Boat` are marked with `override` keyword. Calls on values of type `Vehicle` are dispatched depending on their runtime class. If you run the following code:

```

fun startAndStop(vehicle: Vehicle) {
    vehicle.start()
    vehicle.stop()
}

fun main() {
    startAndStop(Car())
    startAndStop(Boat())
}

```

You'll get:

```

I'm riding
Stopped
I'm sailing
Stopped

```

On the other hand, the `stop()` method is final. Since it is not explicitly marked as open, it cannot be overridden and is simply inherited by subclasses.

Java vs. Kotlin: It is worth pointing out two major differences between overriding in Kotlin and Java. First, like Java, Kotlin functions and properties are final by default and must be explicitly marked with `open` keyword to permit overriding in subclasses.

However, in Java, methods are implicitly open, so if you want to forbid their overriding, then you must do it with explicit `final` modifier. Second, overridden members in Kotlin must be accompanied by `override` keyword always. Failing to do so produces a compilation error. On the other hand, in Java, explicit marking of overriding methods is optional; although it's considered a good practice to use the `@Override` annotation. Enforcing explicit marking of overridden members in Kotlin helps in preventing the accidental override problem, where you add a member that just happens to match some super class and overrides its implementation, leading to unexpected program behavior and hard-to-find bugs.

It is worth pointing out an important difference between members and extensions. While class members can be overridden (provided they are not `final`) and thus, chosen based on the runtime class of a particular instance, extensions are always resolved statically. In other words, when calling an extension, the compiler always chooses it on the basis of statically known receiver type. Consider the following example:

```
open class Vehicle {  
    open fun start() {  
        println("I'm moving")  
    }  
}  
  
fun Vehicle.stop() {  
    println("Stopped moving")  
}  
  
class Car : Vehicle() {  
    override fun start() {  
        println("I'm riding")  
    }  
}  
  
fun Car.stop() {  
    println("Stopped riding")  
}  
  
fun main() {  
    val vehicle: Vehicle = Car()  
    vehicle.start() // I'm riding  
    vehicle.stop() // Stopped moving  
}
```

It is clear that the program calls `start()` defined in `Car` class, because it is resolved dynamically, depending on the runtime type of the `vehicle` variable (which is `Car`). However, the `stop()` is chosen depending on the static type of `vehicle` (which is `Vehicle`), so the function called is `Vehicle.stop()`.

Please note, that the signature of an overridden member must match that of its superclass version:

```
open class Vehicle {  
    open fun start(speed: Int) {  
        println("I'm moving at $speed")  
    }  
}  
  
class Car : Vehicle() {  
    override fun start() { // Error: wrong signature  
        println("I'm riding")  
    }  
}
```

However, you can replace return type with its supertype:

```
open class Vehicle {  
    open fun start(): String? = null  
}  
  
open class Car : Vehicle() {  
    final override fun start() = "I'm riding a car"  
}
```

If you declare the overridden member as `final`, it won't be overridden further in subclasses:

```
open class Vehicle {  
    open fun start() {  
        println("I'm moving")  
    }  
}  
  
open class Car : Vehicle() {  
    final override fun start() {  
        println("I'm riding a car")  
    }  
}
```

```
    }
}

class Bus : Car() {
    override fun start() { // Error: start() is final in Car
        println("I'm riding a bus")
    }
}
```

Properties can be overridden too. Apart from placing their implementations in subclass body, you also have an option to override them as primary constructor parameters:

```
open class Entity {
    open val name: String get() = ""
}

class Person(override val name: String) : Entity()
```

Immutable properties can be overridden by mutable ones:

```
open class Entity {
    open val name: String get() = ""
}

class Person() : Entity() {
    override var name: String = ""
}
```

Like Java, Kotlin has a special access modifier, which restricts members' scope to its inheritors. Such members are marked with `protected` keyword:

```
open class Vehicle {
    protected open fun onStart() { }

    fun start() {
        println("Starting up...")
        onStart()
    }
}

class Car : Vehicle() {
    override fun onStart() {
        println("It's a car")
```

```

    }
}

fun main() {
    val car = Car()
    car.start()      // Ok
    car.onStart()   // Error: onStart is not available here
}

```

Java vs. Kotlin: Please note the difference between the `protected` modifier in Kotlin and Java. While both languages permit access to protected members from inheritor classes, Java also allows using them from any code located in the same package. In Kotlin, that is forbidden. Currently, it does not have an access modifier that restricts the declaration scope to a containing package.

Sometimes an overridden version of a function or property needs to access its original version to reuse its code. In this case, you can prefix your member reference with `super` keyword (the syntax is similar to `this`, but you access an inherited member instead of the current one):

```

open class Vehicle {
    open fun start(): String? = "I'm moving"
}

open class Car : Vehicle() {
    override fun start() = super.start() + " in a car"
}

fun main() {
    println(Car().start()) // I'm moving in a car
}

```

IDE Tips: IntelliJ plugin includes a special action that can help you generate stubs for overriding members. To access it, you can use `Ctrl + O/Cmd + O` shortcut inside a class body. The IDE then presents a dialog to you, where you can choose the superclass members to override (as shown on the *Figure 8.1*).

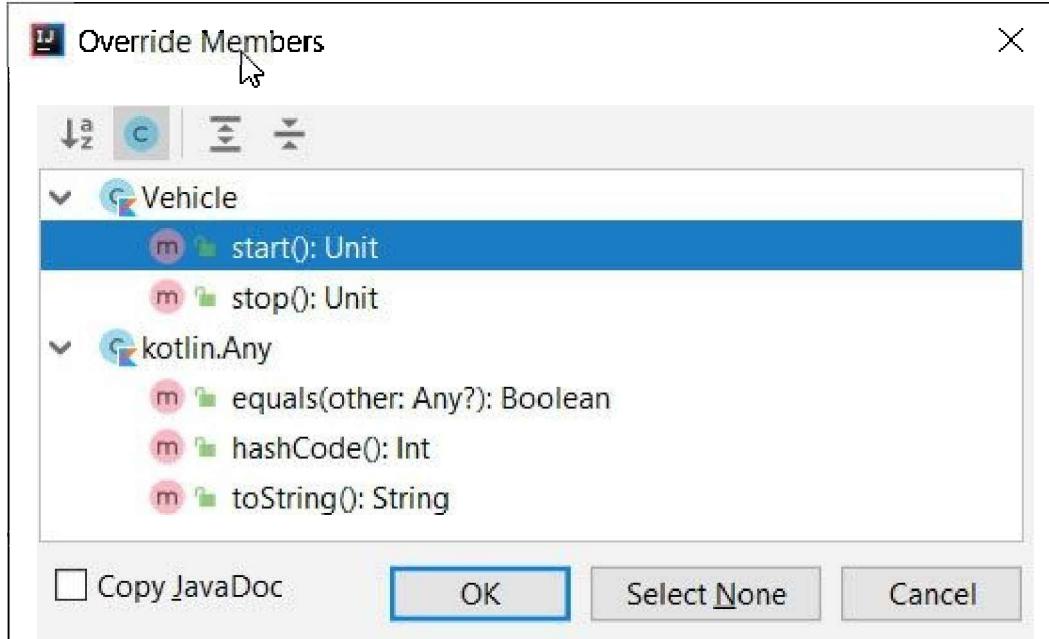


Figure 8.1: SEQ Figure * ARABIC1: Override Members dialog

Subclass initialization

In *Chapter 4 (Working with Classes and Objects)*, we have discussed how constructors are used to initialize an instance state of a particular class. While creating an instance of subclass, your program also needs to call the initialization code defined in its superclasses. The superclass initialization must come first, since it may create an environment used by a subclass code. In Kotlin, this order is enforced automatically: when your program attempts to create an instance of some class A, it gets a chain of its superclasses and then calls their constructors, starting from the hierarchy root (i.e. the Any class) and finishing with a constructor of A. Let's look at an example demonstrating the order of initialization:

```
open class Vehicle {  
  
    init {  
        println("Initializing Vehicle")  
    }  
}  
  
open class Car : Vehicle() {  
    init {  
        println("Initializing Car")  
    }  
}
```

```
    }
}

class Truck : Car() {
    init {
        println("Initializing Truck")
    }
}

fun main() {
    Truck()
}
```

When it is run, this program will print:

```
Initializing Vehicle
Initializing Car
Initializing Truck
```

This confirms the above idea that initialization proceeds from superclass to subclass.

We have already mentioned that parentheses coming after the superclass name in the subclass definition constitute a call to its constructor. Until now, we did not have to pass some arguments there, since the super classes in our examples have been using default constructors. What if, we need to provide them with some data as well? The simplest case is when a superclass has exactly one constructor:

```
open class Person(val name: String, val age: Int)
class Student(name: String, age: Int, val university: String)
:
Person(name, age)
fun main() {
    Student("Euan Reynolds", 25, "MIT")
}
```

In the example above, the primary constructor of the `Student` class passes three of its parameters to the constructor of the `Person` superclass using a so-called delegating call: `Person(firstName, familyName, age)`.

Just like the ordinary constructor calls, delegating calls are equally applicable to both primary and secondary constructors:

```
open class Person {
```

```

    val name: String
    val age: Int
    constructor(name: String, age: Int) {
        this.name = name
        this.age = age
    }
}
class Student(name: String, age: Int, val university: String)
:
Person(name, age)

```

What if, we want to use a secondary constructor in the `Student` class? In this case, the delegating call is specified after the constructor signature:

```

open class Person(val name: String, val age: Int)
class Student : Person {
    val university: String
    constructor(name: String, age: Int, university: String) :
super(name, age) {
        this.university = university
    }
}

```

The `super` keyword tells the compiler that our secondary constructor delegates to the corresponding constructor of the superclass. This syntax resembles delegation to another constructor of the same class, which is denoted by `this` keyword instead (see *Chapter 4, Working with Classes and Objects*). Another difference, as compared to call in primary constructor, is the absence of parentheses after the superclass name: `Person` instead of `Person()`. The reason being that our class does not have a primary constructor and delegating is put into a secondary one instead.

Java vs. Kotlin: Unlike Java, calls between constructors (whether they belong to the same class, or class and its superclass) are never put into the constructor body. In Kotlin, you use a delegating call syntax for that.

Please note that if a class has a primary constructor, its secondary constructor may not delegate to the superclass:

```

open class Person(val name: String, val age: Int)
// Error: call to Person constructor is expected

```

```
class Student() : Person {
    val university: String
    constructor(name: String, age: Int, university: String) :
        super(name, age) { // Error: can't invoke Person
constructor here
    this.university = university
}
}
```

An interesting case is when a superclass has different constructors and we want its subclass to support more than one of them. In this case, using secondary constructors becomes the only option:

```
open class Person {
    val name: String
    val age: Int
    constructor(name: String, age: Int) {
        this.name = name
        this.age = age
    }
    constructor(firstName: String, familyName: String, age: Int) :
        this("$firstName $familyName", age)
}
class Student : Person {
    val university: String
    constructor(name: String, age: Int, university: String) :
        super(name, age) {
            this.university = university
    }
    constructor(
        firstName: String,
        familyName: String,
        age: Int,
        university: String
) :
    super(firstName, familyName, age) {
```

```

        this.university = university
    }
}
fun main() {
    Student("Euan", "Reynolds", 25, "MIT")
    Student("Val Watts", 22, "ETHZ")
}

```

In fact, the use case above, was one of the primary reasons to add the secondary constructor to the language. This becomes important, especially if you consider the interoperability with Java code, which does not distinguish between primary and secondary constructors.

One more issue we would like to highlight in this section is the so-called leaking this problem. Consider the following code:

```

open class Person(val name: String, val age: Int) {
    open fun showInfo() {
        println("$name, $age")
    }
    init {
        showInfo()
    }
}
class Student(
    name: String,
    age: Int,
    val university: String
) : Person(name, age) {
    override fun showInfo() {
        println("$name, $age (student at $university)")
    }
}
fun main() {
    Student("Euan", "Reynolds", 25, "MIT")
}

```

If you run the program, the output will look like this:

Euan Reynolds, 25 (student at null)

Why does the university variable happen to be null? The reason is that the method `showInfo()` is invoked in the superclass initializer. It is a virtual function, so the program will call its overriding version in the `Student` class, but since the `Person` initializer runs before that of the `Student`, the `university` variable is not yet initialized at the moment of `showInfo()` call. The reason this situation is called leaking this is that the super class leaks the current instance to code, which in general may depend on the yet uninitialized part of the instance state. A more explicit example could look as follows:

```
open class Person(val name: String, val age: Int) {
    override fun toString() = "$name, $age"
    init {
        println(this) // potentially dangerous
    }
}
class Student(
    name: String,
    age: Int,
    val university: String
) : Person(name, age) {
    override fun toString() = super.toString() + "(student at
$university)"
}
fun main() {
    // Euan Reynolds, 25 (student at null)
    Student("Euan Reynolds", 25, "MIT")
}
```

The issue of leaking `this` poses a rare case when the variable of the non-nullable type in Kotlin may turn out null.

IDE Tips: IntelliJ plugin includes an inspection, which flags such calls and usages as potentially unsafe, displaying an appropriate warning (as shown on the *Figure 8.2*):

The screenshot shows a code editor with Kotlin code. Line 11 contains the code `showInfo()`. A yellow tooltip box appears over this line with the text "Calling non-final function showInfo in constructor more... (Ctrl+F1)". The code itself is:

```
1 open class Person(
2     val firstName: String,
3     val familyName: String,
4     val age: Int
5 ) {
6     open fun showInfo() {
7         println("$firstName $familyName, $age")
8     }
9
10 init {
11     showInfo()
12 }
```

Figure 8.2: SEQ Figure /* ARABIC2: Warning on non-final function call inside constructor

Type checking and casts

Since a variable of some class may refer to any instance of its subtypes at runtime, it's useful to have a means to check if a particular instance corresponds to a more specific type, and cast it to that type when necessary. For example, consider the following code:

```
val objects = arrayOf("1", 2, "3", 4)
```

From the compiler's point of view, `objects` is an array of `Any`, since `Any` is a minimal common supertype that covers all of its elements. What if we want to use some `String`-or `Int`-specific operations? Applying them to array elements directly won't work, since they have `Any` type and thus do not support more specific functions or properties:

```
for (obj in objects) {
    println(obj*2) // Error: * is not supported for Any
}
```

Kotlin provides a solution in the form of type checking and casting operators. The `is` operator returns true if its left operand has a given type. Let's change our example a little:

```
for (obj in objects) {  
    println(obj is Int)  
}
```

When you run the program above, it prints:

```
false  
true  
false  
true
```

As expected, the null value is considered an instance of any nullable type, but doesn't belong to non-nullable ones:

```
println(null is Int)          // false  
println(null is String?) // true
```

Kotlin also supports inverted operation which is expressed by `!is` operator:

```
val o: Any = ""  
println(o !is Int)          // true  
println(o !is String) // false
```

Please note, that the `is`/`!is` operators are only applicable when the static type of their left operand is a supertype of type at the right. The following check produced a compilation error, since it is meaningless to test an `Int` value against `String`, when the compiler knows, statically, that `String` is not an `Int` subtype:

```
println(12 is String) // Error
```

Both, `is` and `!is` operators, have the same precedence as `in` and `!in`.

Java vs. Kotlin: The `is` operator is very similar to Java's `instanceof`. However, please bear in mind that they diverge in their treatment of null. While `instanceof` always returns false when applied to null, the result of the `is` operator depends on whether its right-hand type is nullable or not.

In *Chapter 4 (Working with Classes and Objects)* we have introduced the concept of smart casts, which allowed us to automatically refine the variable's type from nullable to non-nullable, after comparing it with null. This useful feature is supported for the `is`/`!is` checks as well. For example:

```
val objects = arrayOf("1", 2, "3", 4)
var sum = 0
for (obj in objects) {
    if (obj is Int) {
        sum += obj // type of obj is refined to Int here
    }
}
println(sum) // 6
```

The `is`/`!is` checks and smart casts are also supported in the expressions where you can use them as a special kind of condition similar to `in`/`!in`:

```
val objects = arrayOf("1", 2, "3", 4)
var sum = 0
for (obj in objects) {
    when (obj) {
        is Int -> sum += obj // obj has Int type
here
        is String -> sum += obj.toInt() // obj has String
type here
    }
}
println(sum) // 10
```

We have already mentioned earlier, that the compiler permits a smart cast only when it can ensure that the variable type does not change before its check and the usage. Now we can express the smart cast rules more precisely.

First, smart casts are not allowed for properties and variables with custom accessors, since the compiler cannot guarantee that its return value will not change after the check. This also includes properties and local variables that use delegates:

```
class Holder {
    val o: Any get() = ""
}
fun main() {
    val o: Any by lazy { 123 }
    if (o is Int) {
        println(o*2) // Error: smart cast is
```

```
not possible
    }
    val holder = Holder()
    if (holder.o is String) {
        println(holder.o.length) // Error: smart cast is not
possible
    }
}
```

Open member properties also fall into this category, since they can be overridden in subtypes and can be given a custom accessor:

```
open class Holder {
    open val o: Any = ""
}
fun main() {
    val holder = Holder()
    if (holder.o is String) {
        println(holder.o.length) // Error: smart cast is not
possible
    }
}
```

Mutable local variables cannot be smartcast when their value is explicitly changed between the check and the read, or if they are modified in some lambda (the latter means that their value may change when lambda is invoked, which in general is unpredictable):

```
fun main() {
    var o: Any = 123
    if (o is Int) {
        println(o + 1)           // Ok: smart cast to Int
        o = ""
        println(o.length) // Ok: smart cast to String
    }
    if (o is String) {
        val f = { o = 123 }
        println(o.length) // Error: smart cast is not
```

```
possible
    }
}
```

Mutable properties, on the other hand, can't use smartcasts, since their value may be changed at any time by some other code.

It is worth noting that immutable local variables without delegates always support smartcasts, which is one more argument for preferring them over mutable ones.

However, when smartcasts are not available, we can use explicit operators to coerce a given value to some type. Kotlin supports two operators of this kind - `as` and its safe version `as?`. The difference lies in their treatment of values which do not conform to the target type. While `as` throws an exception, `as?` simply returns null:

```
val o: Any = 123
println((o as Int) + 1)           // 124
println((o as? Int)!! + 1)         // 124
println((o as? String ?: "").length) // 0
println((o as String).length)     // Exception
```

Please note the difference between expressions like `o as String?` and `o as? String`. They have the same value when `o` is a value of `String?` (including null), but behave differently when it is not:

```
val o: Any = 123
println(o as? String) // null
println(o as String?) // Exception
```

Also, do note that the attempt to cast null to non-nullable type produces an exception at runtime:

```
println(null as String) // Exception
```

Java vs. Kotlin: The `as` operator is like the Java cast expression, except the null treatment. In Java, casting always leaves null unchanged, while in Kotlin, the result depends on the nullability of the target type.

Common methods

The `kotlin.Any` class is a root of the Kotlin class hierarchy - every other class is its direct or indirect inheritor. When you don't specify an explicit superclass in your class

definition, the compiler automatically assumes that it is Any. Therefore, the members of this class are available for all values. Let's look at how it's defined:

```
open class Any {  
    public open operator fun equals(other: Any?): Boolean  
    public open fun hashCode(): Int  
    public open fun toString(): String  
}
```

The operator keyword here means that the `equals()` method can be invoked in an operator form (via `==` or `!=`). We will discuss operator syntax in *Chapter 11, Domain-Specific Languages*.

These methods define the basic operations that can be performed on any non-null value:

- Structural equality (`==` and `!=`);
- Computation of hash code, which is used by some collection types, like `HashSet` or `HashMap`;
- Default conversion to `String`;

Java vs. Kotlin: Readers familiar with Java will surely recognize the Any definition as somewhat of a minimalistic version of `java.lang.Object`. In fact, on the JVM, runtime values of Any are represented as `Object` instances.

In *Chapter 6, Using Special-Case Classes* we have already discussed an example of using referential equality, which the compiler automatically provides for any data class. Now, we'll see how to implement a custom equality operation for an arbitrary Kotlin class. Consider the following code:

```
class Address(  
    val city: String,  
    val street: String,  
    val house: String  
)  
open class Entity(  
    val name: String,  
    val address: Address  
)  
class Person(  
    val name: String,  
    val address: Address  
)
```

```

name: String,
address: Address,
val age: Int
): Entity(name, address)
class Organization(
    name: String,
    address: Address,
    val manager: Person
) : Entity(name, address)

```

By default, these classes implement the referential equality inherited from the `Any` class only. For example, if we try to use them as collection elements, we may face a problem, since two instances with equal properties are not considered equal themselves:

```

fun main() {
    val addresses = arrayOf(
        Address("London", "Ivy Lane", "8A"),
        Address("New York", "Kingsway West", "11/B"),
        Address("Sydney", "North Road", "129")
    )
    // -1
    println(addresses.indexOf(Address("Sydney", "North Road",
"129")))
}

```

This problem can be fixed by overriding the `equals()` method and implementing content-based equality. A simple implementation could look like this:

```

override fun equals(other: Any?): Boolean {
    if (other !is Address) return true
    return city == other.city &&
        street == other.street &&
        house == other.house
}

```

Now the `index()` call from the example above, can find our `Address` object and returns 2.

Please note that the `equals()` method is commonly used in its operator form `==` or `!=`. These operators may be applied to nullable values as well. When the left operand is null they simply compare the right one with null referentially. The original referential equality is implemented by `==`and `!=` operators. Their behavior, unlike that of `==`and `!=`, cannot be overridden in user code:

```
val addr1 = Address("London", "Ivy Lane", "8A")
val addr2 = addr1                                // the same instance
val addr3 = Address("London", "Ivy Lane", "8A") // different,
but equal
println(addr1 === addr2) // true
println(addr1 == addr2)   // true
println(addr1 === addr3) // false
println(addr1 == addr2)   // true
```

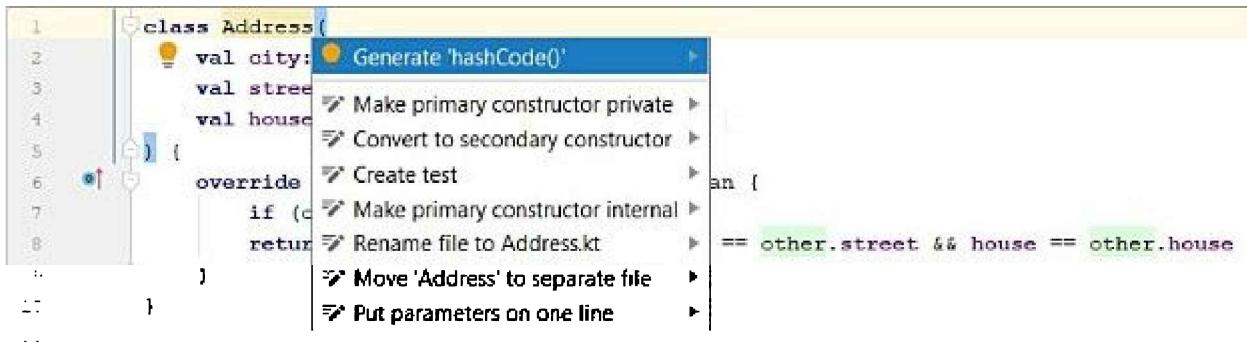
Java vs. Kotlin: In Java, `==`and `!=` operators implement referential equality, while content-based is expressed by an explicit call to `equals()`. The latter must also be guarded against the possible null value of its receiver object, to avoid NPE.

Just like in Java, a custom implementation of `equals()` method must be accompanied by a corresponding `hashCode()`. Both implementations must be related, so that any pair of equal objects (from the `equals()` point of view) always have the same hash code. This is because some collections (such as `HashSet`) use `hashCode()` to find a value in the hash table first, and then use `equals()` method to filter through all the candidates with the same hash code. If equal objects have different hash codes, then such collections will filter them out even before calling `equals()`. A possible `hashCode()` implementation, which is compatible with the `equals()` method above, can look like this:

```
override fun hashCode(): Int {
    var result = city.hashCode()
    result = 31 * result + street.hashCode()
    result = 31 * result + house.hashCode()
    return result
}
```

IDE Tips: IntelliJ plugin warns you about the classes that provide implementation of `equals()`, but not `hashCode()`, or vice versa. It also allows you to add the missing method by automatically generating some

reasonable implementation (see *Figure 8.3*):



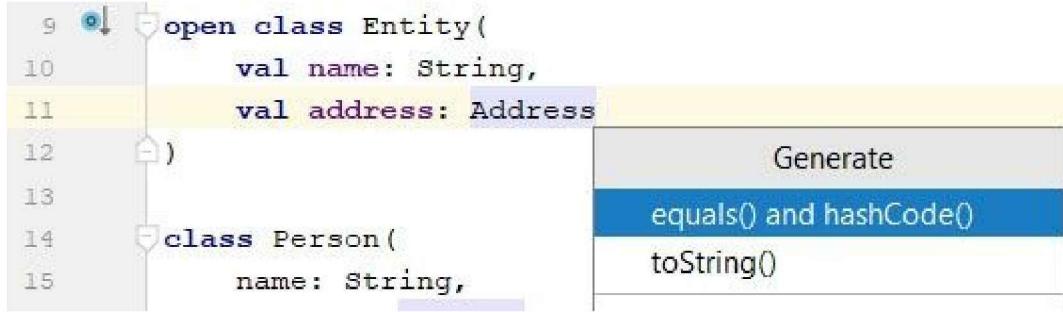
*Figure 8.3: SEQ Figure * ARABIC3: Using IDE inspection to generate missing hashCode() method*

The general requirements for `equals()` implementations are basically the same as Java:

- No non-null object must be equal to null;
- Each object must be equal to itself;
- Equality must be symmetric: $a == b$ must entail $b == a$;
- Equality must be transitive: $a == b$ and $b == c$ must entail that $a == c$;

IDE Tips: IntelliJ plugin can automatically generate implementations of `equals()` and `hashCode()` methods, based on the class properties. These methods are quite similar to the ones provided for data classes and would give reasonable equality behavior in most situations. In the remaining cases, you may use them as a good starting point for writing your own implementations.

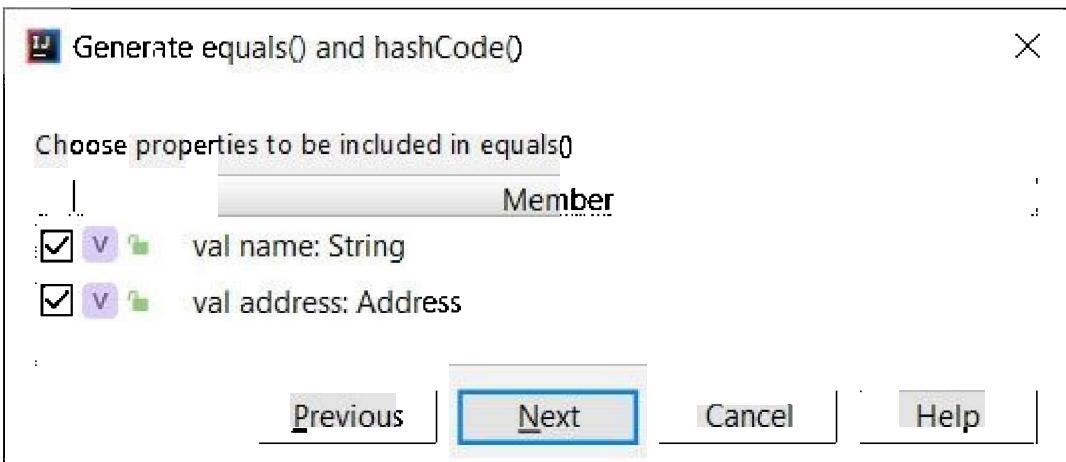
To generate methods, choose `equals()` and `hashCode()` in the Generate menu, which is brought up by the Alt + Insert shortcut inside a class definition (see *Figure 8.4*):



*Figure 8.4: SEQ Figure * ARABIC4: Generate menu*

When a class in question is an open one, the IDE will suggest that you generate methods that also support instances of its subclasses. If you agree, then the instances of different subclasses may happen to be equal, which is not always desirable. In our example, we will not use this option, since we want instances of `Person` and `Organization` to be distinct from each other.

You then proceed with choosing properties that should be used in the generated methods (as shown on the *Figure 8.5*). Please note that only the properties chosen for `equals()` may be used in `hashCode()`. This ensures that both the methods are compatible, in a sense that equal objects always have the same hash code.



*Figure 8.5: SEQ Figure * ARABIC5: Choosing properties for equals() method implementation*

Applying this action to the `Entity` class will produce the following code:

```

open class Entity(
    val name: String,
    val address: Address
) {
    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (javaClass != other?.javaClass) return false
        other as Entity
        if (name != other.name) return false
        if (address != other.address) return false
        return true
    }
    override fun hashCode(): Int {
        var result = name.hashCode()
        result = 31 * result + address.hashCode()
        return result
    }
}

```

Properties are compared by delegating them to their own implementation of `equals()` and `hashCode()`. Array types comprise of an exception - since they do not have their own content-based equality implementation, the generated code will use `contentEquals()` and `contentHashCode()` (or `contentDeepEquals()`/`contentDeepHashCode()`, when applied to the properties of multidimensional array types).

If superclass has its own non-trivial implementation of `equals()`/`hashCode()`, then the corresponding implementation in superclass will automatically include a call to its super counterpart. For example, on applying ‘Generate `equals()`/`hashCode()`’ to the `Person` class, we get:

```

class Person(
    name: String,
    address: Address,
    val age: Int
): Entity(name, address) {
    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (javaClass != other?.javaClass) return false

```

```

        if (!super.equals(other)) return false
        other as Person
        if (age != other.age) return false
        return true
    }
    override fun hashCode(): Int {
        var result = super.hashCode()
        result = 31 * result + age
        return result
    }
}

```

Just like Java, all Kotlin classes have a `toString()` method, which provide the default String representation of a given instance. By default, such representation is composed of a class name and an object's hash code. Therefore, in most cases, it is worth overriding to get a more readable information:

```

class Address(
    val city: String,
    val street: String,
    val house: String
) {
    override fun toString() = "$city, $street, $house"
}
open class Entity(
    val name: String,
    val address: Address
)
class Person(
    name: String,
    address: Address,
    val age: Int
): Entity(name, address) {
    override fun toString() = "$name, $age at $address"
}
class Organization(
    name: String,

```

```

address: Address,
val manager: Person?
) : Entity(name, address) {
    override fun toString() = "$name at $address"
}
fun main() {
    // Euan Reynolds, 25 at London, Ivy Lane, 8A
    println(Person("Euan Reynolds", Address("London", "Ivy
Lane", "8A"), 25))
    // Thriftocracy, Inc. at Perth, North Road, 129
    println(
        Organization(
            "Thriftocracy, Inc.",
            Address("Perth", "North Road", "129"),
            null
        )
    )
}

```

IDE Tips: IntelliJ also allows you to generate a simple `toString()` implementation, like the `equals()` / `hashCode()` methods. To do this, you just need to select the `toString()` option in the Generate menu (see *Figure 8.4*), and choose the properties you want to use in `toString()`. You may choose to generate the resulting string as either a single string template, or a concatenation expression. If the superclass has some nontrivial `toString()` implementation already, then you may additionally choose whether to add a `super.toString()` call.

Below is the result of applying Generate `toString()` action to our `Person` class:

```

class Person(
    val name: String,
    val age: Int,
    address: Address
): Entity(address) {
    override fun toString(): String {

```

```
        return "Person(name='$name', age=$age)  
${super.toString()}"  
    }  
}
```

The Kotlin standard library also includes the `toString()` extension that is defined for `Any?` type. This function simply delegates to the receiver's `toString()` member when it's not null, and returns the null string otherwise. This allows the use of `toString()` on both, nullable and non-null values.

Abstract classes and interfaces

So far, all the superclasses that we have seen, could have their own instances. However, sometimes, this is undesirable because classes may also represent abstract concepts, which do not have instances by themselves and are only instantiated through more specific cases. For example, our earlier example involved the `Entity` class subclassed by `Person` and `Organization`. While it makes sense to have objects representing particular persons and organizations, an entity by itself is an abstract notion. So it is meaningless to create an instance of just `Entity` rather than one of its specific subclasses. In the upcoming sections, we will deal with Kotlin aspects which allow us to define and use such abstract types.

Abstract classes and members

Like Java, Kolin also supports abstract classes, which cannot be instantiated directly but instead serve only as super types for other classes. In order to mark a class as abstract, you use a corresponding modifier keyword:

```
abstract class Entity(val name: String)  
// Ok: delegation call in subclass  
class Person(name: String, val age: Int) : Entity(name)  
val entity = Entity("Unknown") // Error: Entity can't be  
instantiated
```

As you can see in the example above, abstract classes may have their own constructors. The difference between abstract and non-abstract classes is that an abstract class constructor may only be invoked as a part of a delegation call in the subclass definition. In the following code, the secondary constructor delegates to the constructor of the abstract class:

```

abstract class Entity(val name: String)
class Person : Entity {
    constructor(name: String) : super(name)
    constructor(
        firstName: String,
        familyName: String
    ) : super("$firstName $familyName")
}

```

Another feature of abstract classes is that it allows you to declare abstract members. An abstract member defines a basic shape of function or property, such as its name, parameters and return type, but omits any implementation details. When a non-abstract class inherits such members from its abstract parent, they must be overridden and given an implementation:

```

import kotlin.math.PI
abstract class Shape {
    abstract val width: Double
    abstract val height: Double
    abstract fun area(): Double
}
class Circle(val radius: Double) : Shape() {
    val diameter get() = 2*radius
    override val width get() = diameter
    override val height get() = diameter
    override fun area() = PI*radius*radius
}
class Rectangle(
    override val width: Double,
    override val height: Double
) : Shape() {
    override fun area() = width*height
}
fun Shape.print() {
    println("Bounds: $width*$height, area: ${area() }")
}
fun main() {

```

```

// Bounds: 20.0*20.0, area: 314.1592653589793
Circle(10.0).print()
// Bounds: 3.0*5.0, area: 15.0
Rectangle(3.0, 5.0).print()
}

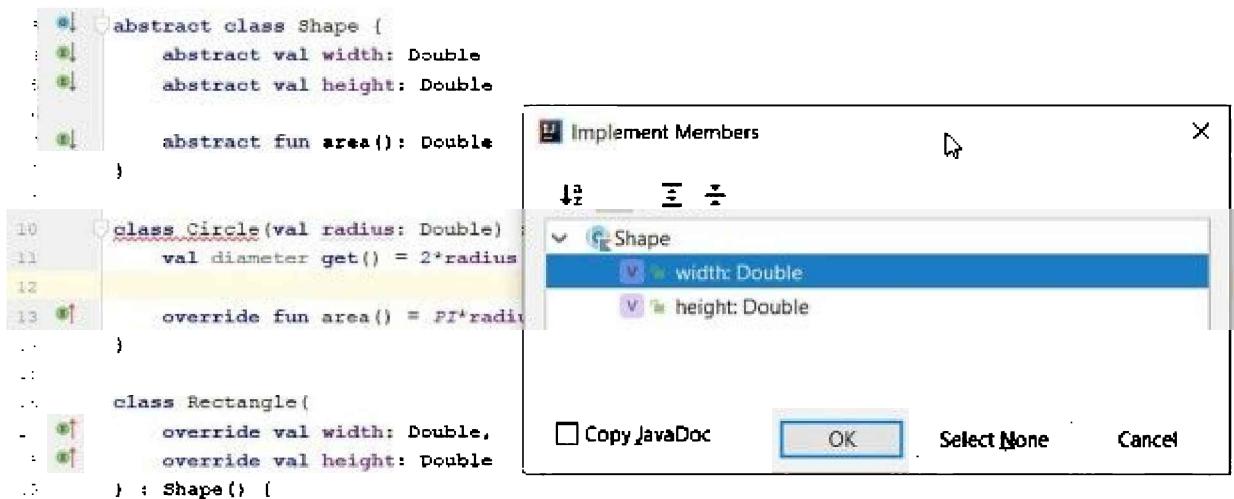
```

Since abstract members are not supposed to have an implementation by themselves, their definitions are subject to some limitations. In particular:

- Abstract properties may not have initializers, explicit accessors or by clause;
- Abstract functions may not have a body;
- Both abstract properties and functions must explicitly specify their return type, since it cannot be inferred automatically.

Please note that abstract members are implicitly open, so you don't need to explicitly mark them as such.

IDE Tips: Apart from the **Override Members** action that we have seen in the **Overriding Class Members** section, IntelliJ has a similar action known as **Implement Members**. The action is available by the **Ctrl + I** shortcut and produces a dialog similar to that of **Override Members**, but lists only those members that are yet to be implemented (see an example for **Circle** class in the *Figure 8.6*):



*Figure 8.6: SEQ Figure * ARABIC7: Implement Members dialog*

An alternate option is to use one of the quick-fixes available from the Alt + Enter menu invoked on class name or keyword (see the red highlighting on the *Figure 8.7*). Among other things, these quick fixes allow you to implement abstract properties such as constructor parameters (see Rectangle class for an example), or simply mark the current class as abstract.

Interfaces

Kotlin interfaces are conceptually quite similar to their Java counterparts, especially after the introduction of default methods in Java 8. In essence, an interface is a type that can contain methods and properties (both abstract and non-abstract), but can't define either instance state, or constructors.

Unlike classes, an interface definition is introduced by interface keyword:

```
interface Vehicle {  
    val currentSpeed: Int  
    fun move()  
    fun stop()  
}
```

Interface members are abstract by default. So if you do not provide an implementation (like in the code above), the `abstract` modifier is automatically assumed. You can write it explicitly, but that is considered redundant.

Interfaces can be supertypes for both, classes and other interfaces. When a non-abstract class inherits an interface, it must provide implementations for all abstract members (and may optionally override non-abstract ones). Similarly, class-to-class inheritance implementations of interface members must be marked with the keyword `override`:

```
interface FlyingVehicle : Vehicle {  
    val currentHeight: Int  
    fun takeOff()  
    fun land()  
}  
class Car : Vehicle {  
    override var currentSpeed = 0  
    private set  
    override fun move() {
```

```

        println("Riding...")
        currentSpeed = 50
    }
    override fun stop() {
        println("Stopped")
        currentSpeed = 0
    }
}

class Aircraft : FlyingVehicle {
    override var currentSpeed = 0
    private set
    override var currentHeight = 0
    private set
    override fun move() {
        println("Taxiing...")
        currentSpeed = 50
    }
    override fun stop() {
        println("Stopped")
        currentSpeed = 0
    }
    override fun takeOff() {
        println("Taking off...")
        currentSpeed = 500
        currentHeight = 5000
    }
    override fun land() {
        println("Landed")
        currentSpeed = 50
        currentHeight = 0
    }
}

```

Please note the absence of () after the supertype name in the definitions of all three types. This is explained by the fact that, unlike classes, interfaces have no constructors and thus, no code to call upon subclass initialization.

Java vs. Kotlin: Please note that in Kotlin all the possible cases of inheritance (class

from class, interface from interface and class from interface) are denoted by the same symbol (:) as opposed to Java, which requires using the implements keyword when the class inherits from an interface, and extends in all other cases.

Like Java, Kotlin interfaces are not allowed to inherit from classes. The Any class can be considered an exception, since it is implicitly inherited by each Kotlin class and interface.

Interface functions and properties may also have implementations:

```
interface Vehicle {  
    val currentSpeed: Int  
    val isMovingget() = currentSpeed != 0  
    fun move()  
    fun stop()  
    fun report() {  
        println(if (isMoving) "Moving at $currentSpeed" else  
"Still")  
    }  
}
```

These implementations are considered implicitly open and thus, can be overridden by inheritors. Marking an interface member as final is a compilation error:

```
interface Vehicle {  
    final fun move() {} // Error  
}
```

However, you may use the extension functions and properties as an alternative to final members:

```
fun Vehicle.relativeSpeed(vehicle: Vehicle) =  
    currentSpeed - vehicle.currentSpeed
```

Like classes, interface methods can also be overridden by inheriting interfaces:

```
interface Vehicle {  
    fun move() {  
        println("I'm moving")  
    }  
}
```

```
interface Car : Vehicle {  
    override fun move() {  
        println("I'm riding")  
    }  
}
```

IDE Tips: The Override Members and Implement Members actions that we have discussed in previous sections are also available inside of the interface bodies.

Since interfaces are not allowed to define state, they can't contain properties with backing fields. Properties with initializers and delegates are particularly forbidden:

```
interface Vehicle {  
    val currentSpeed = 0           // Error  
    val maxSpeed by lazy { 100 } // Error  
}
```

The interface is also implicitly abstract. However, unlike abstract classes, interfaces are forbidden to define any constructors:

```
interface Person(val name: String) // Error  
interface Vehicle {  
    constructor(name: String)           // Error  
}
```

Like Java, Kotlin interfaces also support multiple inheritance. Let's consider an example:

```
interface Car {  
    fun ride()  
}  
interface Aircraft {  
    fun fly()  
}  
interface Ship {  
    fun sail()  
}  
interface FlyingCar : Car, Aircraft
```

```

class Transformer : FlyingCar, Ship {
    override fun ride() {
        println("I'm riding")
    }
    override fun fly() {
        println("I'm flying")
    }
    override fun sail() {
        println("I'm sailing")
    }
}

```

Both, the `FlyingCar` interface and the `Transformer` class, inherit from more than one interface at once thus, getting all their members. In the case of non-abstract `Transformer` class, we also must implement all inherited members.

An interesting issue arises when a single type inherits from more than one different interface, that have members with the same signatures. In this case, they are effectively merged into single members, which are then inherited by a subtype. Let us suppose that our `Car` and `Ship` interfaces do not have a common supertype apart from `Any`:

```

interface Car {
    fun move()
}

interface Ship {
    fun move()
}

class Amphibia : Car, Ship {
    override fun move() {
        println("I'm moving")
    }
}

```

In the code above, both the variants of the `move()` method are abstract, so we have to implement it in the non-abstract `Amphibia` class. However, even if some of them do have implementations, the compiler will still force us to provide an explicit implementation to resolve a possible ambiguity:

```

interface Car {
    fun move() {
        println("I'm riding")
    }
}
interface Ship {
    fun move()
}
class Amphibia : Car, Ship {
    override fun move() {
        super.move() // Calling inherited implementation from
Car
    }
}
fun main() {
    Amphibia().move() // I'm riding
}

```

When more than one supertype provides an implementation of a merged member like this, the super-call itself becomes ambiguous. In this case, you may use an extended form of super qualified with a supertype name:

```

interface Car {
    fun move() {
        println("I'm riding")
    }
}
interface Ship {
    fun move() {
        println("I'm sailing")
    }
}
class Amphibia : Car, Ship {
    override fun move() {
        super<Car>.move()      // Call inherited implementation
in Car interface
        super<Ship>.move() // Call inherited implementation
}

```

```

    in Ship interface
    }
}
fun main() {
    /*
        I'm riding
        I'm sailing
    */
    Amphibia().move()
}

```

Java vs. Kotlin: Java 8 uses the qualified form of super for the same purpose: `Ship.super.move()`.

Since version 1.1, the Kotlin compiler can generate non-abstract interface members in the form of the Java 8 default methods. In *Chapter 12, Java Interoperability*, we shall discuss such interoperability issues in more detail.

The limitations concerning the use of state and constructors in interfaces are explained by their support of multiple inheritance. The primary goal was to avoid the infamous diamond inheritance problem. Consider the following classes:

```

interface Vehicle {
    val currentSpeed: Int
}

interface Car : Vehicle
interface Ship : Vehicle
class Amphibia : Car, Ship {
    override var currentSpeed = 0
    private set
}

```

If an instance state was allowed, the `Vehicle` interface may define `currentSpeed` as a state variable. As a result, the `Amphibia` class would inherit two copies of `currentSpeed` - one from the `Car` and the other from the `Ship` (both of which would inherit it from `Vehicle`). Kotlin design prevents the problem at the expense of disallowing state in interfaces. A restriction on the constructor definition is related to the importance of having a predictable initialization order of the program state. Allowing them for interfaces would require extending the initialization order rules

(see *Subclass initialization* section) to cover multiple inheritance, which can become quite cumbersome to follow, especially if some interfaces occur more than once in the supertype graph (like `Vehicle` in the example above).

Sealed classes

Sometimes, the concepts we want to represent in a program may come in a fixed set of variants. In the *Chapter 6 (Using Special-Case Classes)*, we have introduced an idea of enum class, which allows you to represent a predetermined set of constants with the same common type. For example, we can use it to represent a result of some computation as being either a success, or an error:

```
enum class Result {
    SUCCESS, ERROR
}

fun runComputation(): Result {
    try {
        val a = readLine()?.toInt() ?: return Result.ERROR
        val b = readLine()?.toInt() ?: return Result.ERROR
        println("Sum: ${a + b}")
        return Result.SUCCESS
    } catch (e: NumberFormatException) {
        return Result.ERROR
    }
}

fun main() {
    val message = when (runComputation()) {
        Result.SUCCESS -> "Completed successfully"
        Result.ERROR -> "Error!"
    }
    println(message)
}
```

However, in some cases, different variants may have their own attributes. For example, a state of successful completion may be accompanied by a produced result, while a state of error may carry some information about its cause. This is similar to the examples we have already discussed in this chapter. Concepts like these can be modeled with a class hierarchy, where the root abstract class expresses the concept in

general and its subclasses serve as representations of particular variants. Let's refine our example and add some members to the Success and Error cases:

```
abstract class Result {
    class Success(val value: Any) : Result() {
        fun showResult() {
            println(value)
        }
    }
    class Error(val message: String) : Result() {
        fun throwException() {
            throw Exception(message)
        }
    }
}
fun runComputation(): Result {
    try {
        val a = readLine()?.toInt()
        ?: return Result.Error("Missing first argument")
        val b = readLine()?.toInt()
        ?: return Result.Error("Missing second argument")
        return Result.Success(a + b)
    } catch (e: NumberFormatException) {
        return Result.Error(e.message ?: "Invalid input")
    }
}
fun main() {
    val message = when (val result = runComputation()) {
        is Result.Success -> "Completed successfully: ${result.value}"
        is Result.Error -> "Error: ${result.message}"
        else -> return
    }
    println(message)
}
```

This implementation is not flawless. It does not allow us to use the fact that the set of

Result variants is restricted to Success and Error. Nothing prevents some client code from adding a new subclass, for example:

```
class MyStatus: Result()
```

It is also the reason why we need to add the else clause to the when expression. The compiler cannot ensure that the result variable will always hold an instance of either Success, or Error, and forces us to deal with the remaining cases as well.

In Kotlin, we can overcome this problem, due to the courtesy of sealed classes. Let's change our class definition by adding the sealed modifier:

```
sealed class Result {  
    class Success(val value: Any) : Result() {...}  
    class Error(val message: String) : Result() {...}  
}
```

When the class is marked as sealed, its inheritors may be declared in either its body as nested classes and objects, or as top-level classes in the same file (the latter was introduced in Kotlin 1.1). Outside these scopes, the sealed class is effectively final and cannot be inherited from.

Please note that the sealed class is also abstract, hence you cannot create its instance directly. The idea is that any instance of a sealed class must be created through one of its subclasses:

```
val result = Result() // Error: can't instantiate an abstract class
```

In fact, sealed class constructors are private by default, and declaring them with some other visibility modifier is considered as a compile-time error.

Like enums, sealed classes support the exhaustive form of when expression that allows us to avoid the redundant else branches:

```
val message = when (val result = runComputation()) {  
    is Result.Success -> "Completed successfully:  
    ${result.value}"  
    is Result.Error -> "Error: ${result.message}"  
}
```

Please note that the inheritance restriction only covers direct subclasses of a sealed

class. The subclasses may have their own inheritors, provided that they are not final:

```
// Result.kt
sealed class Result {
    class Success(val value: Any) : Result()
    open class Error(val message: String) : Result()
}
// util.kt
class FatalError(message: String) : Result.Error(message)
```

Since Kotlin 1.1, sealed classes may extend other classes as well. This allows the classes to have subclasses, which are also sealed:

```
sealed class Result
class Success(val value: Any) : Result()
sealed class Error : Result() {
    abstract val message: String
}
class ErrorWithException(val exception: Exception): Error() {
    override val message: String get() = exception.message ?:
"""
}
class ErrorWithMessage(override val message: String): Error()
```

It is possible to use data classes as parts of sealed class hierarchy, due to the data class inheritance, which was also introduced in 1.1,. This allows us to combine the advantages of both, data and sealed classes. For example, consider the classes that represent the syntactic tree of simple arithmetic expressions:

```
sealed class Expr
data class Const(val num: Int): Expr()
data class Neg(val operand: Expr): Expr()
data class Plus(val op1: Expr, val op2: Expr): Expr()
data class Mul(val op1: Expr, val op2: Expr): Expr()
fun Expr.eval(): Int = when (this) {
    is Const -> num
    is Neg -> -operand.eval()
    is Plus -> op1.eval() + op2.eval()
    is Mul -> op1.eval() * op2.eval()
```

```

}

fun main() {
    // (1 + 2) * 3
    val expr = Mul(Plus(Const(1), Const(2)), Const(3))
    // Mul(op1=Plus(op1=Const(num=1), op2=Const(num=2)),
    op2=Const(num=3))
    println(expr)
    println(expr.eval()) // 9
    // 2 * 3
    val expr2 = expr.copy(op1 = Const(2))
    // Mul(op1=Const(num=2), op2=Const(num=3))
    println(expr2)
    println(expr2.eval()) // 6
}

```

Please note that the sealed modifier cannot be applied to interfaces. It means that the subclasses comprising a sealed hierarchy cannot inherit from some other class, since multiple class inheritance is forbidden in Kotlin.

A sealed class implementation may also be an object. Let us suppose that we want to refine our `Result` example to distinguish successful state without produced value:

```

sealed class Result {
    object Completed : Result()
    class ValueProduced(val value: Any) : Result()
    class Error(val message: String) : Result()
}

```

When all direct inheritors are objects, a sealed class effectively behaves like an enum.

IDE Tips: If you want to refactor an enum class into a sealed one, IntelliJ plugin can give you a good starting point, thanks to the corresponding intention action available via Alt + Enter menu (see *Figure 8.7*). As a result, enum constants are converted into singletons implementing an abstract sealed class.

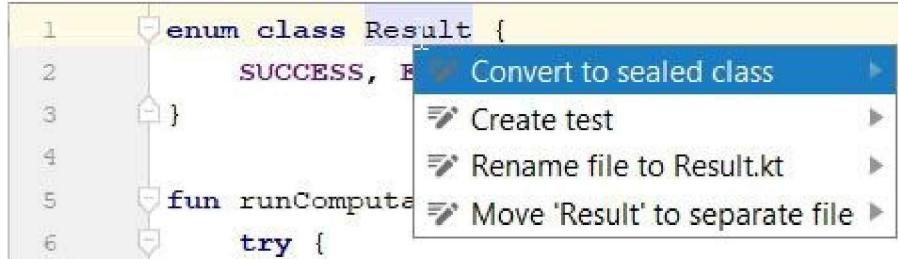


Figure 8.7: SEQ Figure * ARABIC8: Converting enum class to sealed class hierarchy

Apart from that, IntelliJ also supports a reverse transformation - if all direct inheritors of a sealed class are represented by object declarations, then you can turn it into an enum class by replacing its implementations by enum constants (as shown on the Figure 8.8):

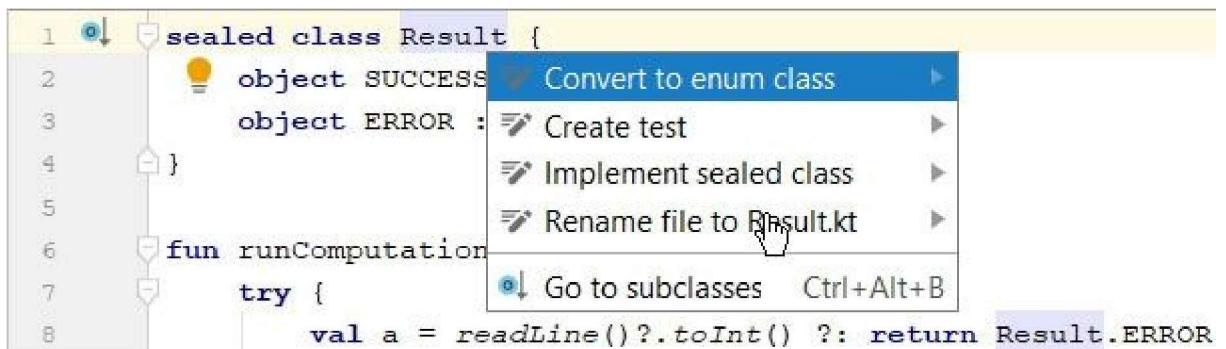


Figure 8.8: SEQ Figure * ARABIC9: Converting sealed class to enum

Delegation

In the preceding section, we have seen that the Kotlin classes are final by default. The goal is to encourage a thought through design of inheritable classes and prevent accidental inheritance from classes that are not supposed to have subclasses. This helps to mitigate the fragile base class problem that we have discussed above.

What if, we still need to extend or change the behavior of some existing class, but cannot inherit from it? In this case, we can use the well-known delegation pattern that allows us to reuse the existing classes. If we want to create an implementation of some interface, then we can take an instance of existing implementation, wrap it inside an instance of our class and delegate our methods to it, when necessary.

Let's consider an example. Suppose we have the following types:

```

interface PersonData {
    val name: String
    val age: Int
}
open class Person(
    override val name: String,
    override val age: Int
): PersonData
data class Book(val title: String, val author: PersonData) {
    override fun toString() = "'$title' by ${author.name}"
}
fun main() {
    val valWatts = Person("Val Watts", 30)
    val introKotlin = Book("Introduction to Kotlin",
valWatts)
    println(introKotlin) // 'Introduction to Kotlin' by Val
Watts
}

```

Now, let us suppose that we want writers to have pen names, allowing them to pose as another person:

```

class Alias(
    private val realIdentity: PersonData,
    private val newIdentity: PersonData
) : PersonData {
    override val name: String
    get() = newIdentity.name
    override val age: Int
    get() = newIdentity.age
}

```

We can now use this class to create person aliases:

```

fun main() {
    val valWatts = Person("Val Watts", 30)
    val johnDoe = Alias(valWatts, Person("John Doe", 25))
    val introJava = Book("Introduction to Java", johnDoe)
}

```

```
    println(introJava) // 'Introduction to Java' by John Doe
}
```

The problem of this approach is the amount of boilerplate code you have to generate to delegate all the necessary methods and properties to another object. Luckily for us, Kotlin has a built-in support for delegates. All you have to do is to specify a delegate instance after the `by` keyword, following a super interface name:

```
class Alias(
    private val realIdentity: PersonData,
    private val newIdentity: PersonData
) : PersonData by newIdentity
```

Now all members the `Alias` inherits from `PersonData` interface are implemented by delegating to the corresponding calls on `newIdentity` instance. We may also override some of them to change implementation behavior:

```
class Alias(
    private val realIdentity: PersonData,
    private val newIdentity: PersonData
) : PersonData by newIdentity {
    override val age: Int get() = realIdentity.age
}
fun main() {
    val valWatts = Person("Val Watts", 30)
    val johnDoe = Alias(valWatts, Person("John Doe", 25))
    println(johnDoe.age) // 30
}
```

In general, the delegate expression can be anything you can use in a class initialization. The compiler automatically creates a field to store the delegate value, when necessary. For example, we can drop `val` on `newIdentity`, making it a simple parameter:

```
class Alias(
    private val realIdentity: PersonData,
    newIdentity: PersonData
) : PersonData by newIdentity
```

We, however, can't delegate to a property defined in the class body:

```
class Alias(
    private val realIdentity: PersonData
) : PersonData by newIdentity { // Error: newIdentity is not
available
    val newIdentity = Person("John Doe", 30)
}
```

Combining delegation with object expressions can be useful to create an implementation with a slightly different behavior than the original object:

```
fun PersonData.aliased(newIdentity: PersonData) =
object : PersonData by newIdentity {
    override val age: Int get() = this@aliased.age
}
fun main() {
    val valWatts = Person("Val Watts", 30)
    val johnDoe = valWatts.aliased(Person("John Doe", 25))
    println("${johnDoe.name}, ${johnDoe.age}") // John Doe,
30
}
```

Please note that a class may only delegate an implementation of interface members. For example, the following code, produces an error since the Person is a class:

```
class Alias(
    private val realIdentity: PersonData,
    private val newIdentity: PersonData
) : Person by newIdentity // Error: only interfaces can be
delegated to
```

The bottom line is as follows:

Class delegation allows you to combine the advantages of composition and inheritance with minimal boilerplate, thus encouraging us to follow the well-known “composition over the inheritance” principle.

Conclusion

In this chapter, we have received an insight into the powerful inheritance mechanism of the Kotlin type system. We have discussed how to define subclasses, how class

initialization fits into the picture of class hierarchy and learned how to use member overriding for changing the base class behavior in subclasses. We have also learned how to employ tools aimed at the representation of abstract concepts, such as abstract classes and interfaces. Finally, we have explored the features implementing two useful inheritance-related patterns: sealed classes and delegation.

In the next chapter, we will focus on the topic of generics - a special feature of the Kotlin type system, giving you the ability to parameterize your declarations with unknown types that are provided later at use site.

Questions

1. How to define subclass in Kotlin? What conditions must a class satisfy in order to be inheritable?
2. Point out the major differences between class inheritance in Java and Kotlin.
3. How is a class instance initialized, when its class is an inheritor? How is the superclass initialization enforced in Java? Compare both the approaches.
4. Describe the purpose of the `is/as/as?` operators. How do they compare with the Java type checks and casts?
5. Name the common methods defined in the `Any` class. Describe the basic guidelines for their implementations.
6. What is an abstract class and an abstract class member? What are the rules that govern the abstract class/member implementations?
7. What are the differences between abstract classes and interfaces? Compare interfaces in Kotlin and Java.
8. What are the specifics of interface inheritance? Describe the differences between member overriding for classes and interfaces.
9. What is a sealed class hierarchy? How would you implement it in Java?
10. Describe how class delegation works in Kotlin.

CHAPTER 9

Generics

In this chapter we are going to discuss generics, which is a powerful feature of the Kotlin type system, allowing you to write a code that manipulates data of some unknown types. We will see how to define and use generics declarations, address issues of type erasure and reification concerned with generics representation at runtime and focus on an important concept of variance that can help you improve flexibility of generics by extending subtyping relation to different substitutions of the same generic type. A related topic that we are also going to highlight in this chapter is the concept of type alias that allows you to introduce alternative names for existing types.

Structure

- Generic declarations
- Type bounds and constraints
- Type erasure and reified type parameters
- Declaration-site variance
- Projections
- Type aliases

Objective

Learn the basics of generic declarations in Kotlin and their differences from Java, as well as get an understanding of how to use reified type parameters and variance to design more flexible generic APIs.

Type parameters

In the preceding chapters we've already seen quite a few examples of using generic types such as arrays and various collections classes, as well as generic functions and properties like `map()`, `filter()`, `sorted()` and so on. In this section we're going to discuss how you can generify your own code to improve its flexibility and make use

of more advanced features of the Kotlin type system.

Generic declarations

To make a declaration generic, we need to add one or more type parameters to it. Such parameters can then be used inside the declaration instead of ordinary types. When declaration is used – for example, when we construct an instance of class, or call a function – we need to supply actual types instead of type parameters:

```
val map = HashMap<Int, String>()
val list = arrayListOf<String>()
```

Sometimes these type arguments can be omitted, since compiler can infer them from context:

```
// use explicit type to infer type arguments of HashMap class
val map: Map<Int, String> = HashMap()
// use argument types of arrayListOf() call to infer its type
// arguments
val list = arrayListOf("abc", "def")
```

Java vs. Kotlin: Note the difference between passing type arguments to generic functions in Kotlin vs. generic methods in Java. While Java requires angle brackets to be put right after the dot, like in `Collections.<String>emptyList()`, in Kotlin such arguments are passed after the function name: `emptyList<String>`. Although when calling a class constructor, the syntax is similar: `new ArrayList<String>()` in Java vs. `ArrayList<String>()` in Kotlin.

also It should also be noted that Java supports automatic inference of type arguments when calling a class constructor, but Kotlin requires the use of a so-called diamond operator:

```
Map<Int, String> map = new HashMap<>() // not new HashMap()
!!!
```

The reason for this is a necessity to maintain a backward compatibility with older code written before generics were added in Java 5.

Let's see how to create generic declarations of our own.

Suppose we want to define a class representing a tree which can store values of a given type:

```

class TreeNode<T>(val data: T) {
    private val _children = arrayListOf<TreeNode<T>>()
    var parent: TreeNode<T>? = null
    private set
    val children: List<TreeNode<T>> get() = _children
    fun addChild(data: T) = TreeNode(data).also {
        _children += it
        it.parent = this
    }
    override fun toString() =
        _children.joinToString(prefix = "$data {", postfix =
    "}")
}
fun main() {
    val root = TreeNode("Hello").apply {
        addChild("World")
        addChild("!!!")
    }
    println(root) // Hello {World {}, !!! {}}
}

```

Type parameters of a class are written inside angle brackets, which are put right after the class name. Type parameters may have arbitrary names, but conventional code style is to use capital letters like T, U, V, etc. inside a class type parameters to define types of variables, properties or functions, or as argument types for other generic declarations.

Java vs. Kotlin: When generic class or interface is used to specify a data type, it must be accompanied by corresponding type arguments. Unlike Java, you cannot have a variable of type `TreeNode`: in Kotlin. You need to specify a type argument for T, like `TreeNode<String>` or `TreeNode<U>` where U is some other type parameter.

When you call a generic class constructor, explicit type arguments are often unnecessary, since in many cases, compiler can infer them from context: that is why we do not need to specify `<String>` in `TreeNode("Hello")` call above. An important exception is delegation call to super class constructor. Let's change our example a little:

```
open class DataHolder<T>(val data: T)
```

```
// Passing actual type as supertype argument
class StringDataHolder(data: String) : DataHolder<String>
    (data)
// Passing type parameter as supertype argument
class TreeNode<T>(data: T) : DataHolder<T>(data) { ... }
```

Unlike ordinary constructor call, compiler does not infer type arguments in delegation calls, so you always must provide them explicitly. Compare two cases:

```
// Error: need to explicitly specify DataHolder<String>
class StringDataHolder(data: String) : DataHolder(data)
// Ok: DataHolder<String> is inferred automatically
fun stringDataHolder(data: String) = DataHolder(data)
```

Please note that type parameters are not inherited; you pass them to supertype and similarly to constructor parameters, so T in TreeNode and T in DataHolder are separate declarations, in fact, we could've used different names for them:

```
class TreeNode<U>(data: U) : DataHolder<U>(data) { ... }
```

Functions and properties defined in generic classes may access their type parameters as demonstrated by addChild() and children definitions above. Additionally, you can make a property or a function generic by adding type parameters of its own:

```
fun <T>TreeNode<T>.addChild(vararg data: T) {
    data.forEach { addChild(it) }
}
fun <T>TreeNode<T>.walkDepthFirst(action: (T) -> Unit) {
    children.forEach { it.walkDepthFirst(action) }
    action(data)
}
val<T>TreeNode<T>.depth: Int
get() = (children.asSequence().map { it.depth }.max() ?: 0) +
1
fun main() {
    val root = TreeNode("Hello").apply {
        addChild("World", "!!!!")
    }
    println(root.depth) // 2
```

```
}
```

Please note that the type parameter list is placed after `fun` keyword rather than declaration name, as opposed to a generic class. Similarly, for generic class constructors, you may omit explicit type argument in a generic function calls when they can be inferred from the context.

Only extension properties may have their own type parameters. The reason for it is that non-extension property effectively represents a single value, hence it cannot be used to read/write values of different types that depend on supplied type arguments:

```
var <T> root: TreeNode<T>? = null // Error: T must be used in receiver type
```

It is forbidden to add type parameters to object declarations for the same reason:

```
object EmptyTree<T> // Error: type parameters are not allowed for objects
```

Property references do not support type arguments, so for generic properties they are always inferred using receiver type. Declaring generic property with type parameters, which are not actually used in its receiver, is a compile-time error for that very reason:

```
// Error: explicit type arguments are forbidden here
val minDepth = TreeNode("").depth<String>
// Error: T is not used in receiver type
val<T>TreeNode<String>.upperCaseData get() =
    data.toUpperCase()
```

Bounds and constraints

Type parameters do not impose any restrictions on their values by default and behave synonymous to `Any?` type. Sometimes though, implementation of generic class, function or property requires some additional information about the data they manipulate. Expanding our `TreeNode` example, suppose that we want to define a function, which computes an average value among all tree nodes. An operation of this kind is only applicable to numeric trees, so we want the tree elements to be values of `Number`. In order to do this, we declare a type parameter with `Number` as upper bound:

```
fun <T : Number>TreeNode<T>.average(): Double {
```

```

var count = 0
var sum = 0.0
walkDepthFirst {
    count++
    sum += it.toDouble()
}
return sum/count
}

```

When the type parameter has an upper bound, the compiler will check that the corresponding type arguments are subtypes of that bound. By default, upper bound is assumed to be `Any?` so if you don't specify it explicitly, a type parameter may accept any Kotlin type. The following calls are valid, since `Int` and `Double` are subtypes of `Number`:

```

val intTree = TreeNode(1).apply {
    addChild(2).addChild(3)
    addChild(4).addChild(5)
}
println(intTree.average()) // 3.0
val doubleTree = TreeNode(1.0).apply {
    addChild(2.0)
    addChild(3.0)
}
println(doubleTree.average()) // 2.0

```

Calling `average()` on tree of strings, however, produces a compilation error:

```

val stringTree = TreeNode("Hello").apply {
    addChildren("World", "!!!")
}
println(stringTree.average()) // Error: String is not subtype
of Number

```

Please note, that using final class as an upper bound is futile, since there are no other types that can be substituted for this type parameter. In this case, the compiler reports a warning:

```
// Can be replaced by a non-generic function
```

```
// fun TreeNode<Int>.sum(): Int {...}
fun <T : Int>TreeNode<T>.sum(): Int { // Warning
    var sum = 0
    walkDepthFirst{ sum += it }
    return sum
}
```

A type parameter bound may refer to the type parameter itself in which case it's called recursive. For example, if our tree contains instances of Comparable interface, we may find a node with the maximum value:

```
fun <T : Comparable<T>>TreeNode<T>.maxNode(): TreeNode<T> {
    val maxChild = children.maxBy { it.data } ?: return this
    return if (data >= maxChild.data) this else maxChild
}

fun main() {
    // Double is subtype of Comparable<Double>
    val doubleTree = TreeNode(1.0).apply {
        addChild(2.0)
        addChild(3.0)
    }
    println(doubleTree.maxNode().data) // 3.0
    // String is subtype of Comparable<String>
    val stringTree = TreeNode("abc").apply {
        addChildren("xyz", "def")
    }
    println(stringTree.maxNode().data) // xyz
}
```

Bounds can also refer to preceding type parameters. We can make use of this fact to write a function, which appends tree elements to a mutable list:

```
fun <T, U : T>TreeNode<U>.toList(list: MutableList<T>) {
    walkDepthFirst{ list += it }
}
```

Since U is a subtype of T, the function above may accept lists of more general elements. For example, we can append trees of Int and Double to a list of Number (which is their common supertype):

```

fun main() {
    val list = ArrayList<Number>()
    TreeNode(1).apply {
        addChild(2)
        addChild(3)
    }.toList(list)
    TreeNode(1.0).apply {
        addChild(2.0)
        addChild(3.0)
    }.toList(list)
}

```

Java vs. Kotlin: The upper bounds of Kotlin type parameters are quite similar to their Java counterparts, the major difference being the syntax: `T extends Number` in Java vs. `T : Number` in Kotlin.

A particularly common case is constraining type parameter to be not null. To do this we need to use non-nullable type as its upper bound:

```
fun <T: Any>notNullTreeOf(data: T) = TreeNode(data)
```

Type parameter syntax allows you to specify only one upper bound. In some cases, though, we may need to impose multiple restrictions on a single type parameter. This can be achieved by using a slightly more elaborate syntax of type constraint. Let us suppose that we have a pair of interfaces as showcased below:

```

interface Named {
    val name: String
}

interface Identified {
    val id: Int
}

```

Now suppose that we want to define a registry of objects that have both a name and an identifier:

```

class Registry<T> where T : Named, T : Identified {
    val items = ArrayList<T>()
}

```

The where clause is added before declaration body, and lists type parameters with their bounds.

Now that we've got a taste of generics syntax, we can move to the next topic that deals with generics representation at runtime.

Type erasure and reification

In the preceding examples, we have seen that type parameters can be used to specify types of variables, properties and functions inside generic declarations. There are cases, however, when type parameters cannot replace actual types. For example, consider the following code:

```
fun <T>TreeNode<Any>.isInstanceOf(): Boolean =  
    data is T && children.all{ it.isInstanceOf<T>() } // Error
```

The intention is to write a function that checks whether the given tree node and all of its children conform to the specific type `T`. The compiler, however, reports an error on `data is T` expression, and the reason is so-called type erasure.

The readers who are familiar with Java, will probably recognize similar limitations in Java generics as well. It comes from the fact that generics only appeared in Java 5, so newer versions of compiler and virtual machine had to maintain the existing representation of types for backward compatibility with older code. As a result, on JVM, information about type arguments is effectively erased from code (thus the type erasure term), and types like `List<String>` or `List<Number>` merge into the same type `List`.

In Kotlin, generics are available from the version 1.0, but due to JVM being its major platform, it suffers from the same type erasure problem. At runtime, generic code cannot distinguish between different versions of its parameter types, so checks like `data is T` above basically make no sense. The function `isInstanceOf()` has no way to know what `T` means when it's called. It is meaningless to use `is` operator for generic type with arguments for the same reason, although in this case the compiler will report either an error, or a warning, depending on whether the type arguments correspond to the type parameters:

```
val list = listOf(1, 2, 3) // List<Int>  
list is List<Number> // Warning: List<Int> is a subtype of  
List<Number>
```

```
list is List<String> // Error: List<Int> is not a subtype of  
List<String>
```

What if we just need to check that our value is a list, without clarifying its element type? We cannot just write `list` as `List`, because generic types in Kotlin must always be accompanied by type arguments. A correct check looks like this:

```
list is List<*>  
map is Map<*, *>
```

The `*` here, basically means some unknown type and replaces the single type argument. This syntax is, in fact, a special case of so-called projections, which we'll discuss at a later stage.

In some cases, though, the compiler has enough information to ensure that the type check is valid and doesn't report warnings/errors. In the following example, the check basically is concerned about the relationship between the `List` and `Collection` interfaces, rather than their particular types, such as `List<Int>` and `Collection<Int>`:

```
val collection: Collection<Int> = setOf(1, 2, 3)  
  
if (collection is List<Int>) {  
    println("list")  
}
```

Please note, that casts to generic types with non `*` arguments are permitted, but they always produce a warning, since their behavior involves a certain risk. While they allow you to work around limitations of generics, they also may defer actual type error till runtime. For example, both of the following expressions are compiled with warning, but the first completes normally, while the second one throws an exception:

```
val n = (listOf(1, 2, 3) as List<Number>) [0] // OK  
val s = (listOf(1, 2, 3) as List<String>) [0] // Exception
```

The exception in the latter case happens only when the value of a list element (which has type `Int`) is assigned to the variable of a (statically known) type `String`.

In Java, you mostly have to rely on casts or use reflection to work around type erasure. Both approaches have their drawbacks, since casts may mask a problem and lead to errors later. On the other hand, using reflection API may impact performance. Kotlin, however, offers you a third option which doesn't suffer from either of these

weaknesses.

Reification means that type parameter information is retained at runtime. How does the compiler circumvent type erasure? The answer is that reified type parameters are only available for inline functions: since function body is in-lined at call site, where type arguments are provided, the compiler always know which actual types correspond to type parameters in a particular in-lined call.

In order to reify the parameter, we need to mark it with a corresponding keyword. Let's use this feature to fix our `isInstanceOf()` function. Since inline functions Cannot be recursive, we will have to rewrite its implementation to some extent:

```
fun <T>TreeNode<T>.cancelableWalkDepthFirst(
    onEach: (T) -> Boolean
): Boolean {
    val nodes = Stack<TreeNode<T>>()
    nodes.push(this)
    while (nodes.isNotEmpty()) {
        val node = nodes.pop()
        if (!onEach(node.data)) return false
        node.children.forEach { nodes.push(it) }
    }
    return true
}
inline fun <reified T>TreeNode<*>.isInstanceOf() =
cancelableWalkDepthFirst{ it is T }
```

In the code above, we have extracted the actual tree traversal logic into separate non-inline function `cancelableWalkDepthFirst()` to prevent the in-lining of the loop itself. For example, now when we call this function, in the following way:

```
fun main() {
    val tree = TreeNode<Any>
    ("abc").addChild("def").addChild(123)
    println(tree.isInstanceOf<String>())
}
```

Compiler will inline `isInstanceOf()` substituting the actual type `String` instead of `T`, and the code that gets executed will look like this:

```
fun main() {
    val tree = TreeNode<Any>
    ("abc").addChild("def").addChild(123)
    println(tree.cancellableWalkDepthFirst { it is String })
}
```

As opposed to the approaches used in Java, reified type parameters give you both safe (no unchecked casts) and fast (thanks to in-lining) solution. However, please note, that using inline function tends to increase the size of the compiled code, but this issue can be mitigated by extracting heavy portions of the code into separate non-inline functions (like we did with `cancellableWalkDepthFirst()`). Also, since reified type parameters are only supported for inline functions, you cannot use them with classes or properties.

Reified type parameters still have their own limitations, which distinguish them from full-fledged types. It is currently not possible to call the constructor or access companion members via reified type parameters:

```
inline fun <reified T>factory() = T() // Error
```

Also, you cannot substitute non-reified type parameter instead of reified one:

```
fun <T, U>TreeNode<*>.isInstanceOfBoth() =
    isInstanceOf<T>() && isInstanceOf<U>()
```

The reason for this, again, is type erasure. Since we cannot know the actual types substituted for `T` and `U` in `isInstanceOfBoth()`, we have to find a way to safely inline either of `isInstanceOf()` calls.

This concludes our basic discussion of Kotlin generics. Now we'll move on to a more advanced topic of variance, which allows improving of the flexibility of generic declarations by controlling producer/consumer aspects of type behavior.

Variance

Variance is an aspect of generic type, which describes how its substitutions are related to each other in terms of subtyping. In the previous chapters we've already seen examples of generic types with different variance. Arrays and mutable collections, for example, do not preserve subtyping of their arguments, even though `String` is a subtype of `Any`, `Array<String>` is not considered a subtype of `Array<Any>` (neither `Array<Any>` is considered a subtype of `Array<String>`). Immutable

collections, like List or Set, on the other hand, do preserve subtyping, so List<String> is a subtype of List<Any>:

```
val objects: List<Any> = listOf("a", "b", "c") // Correct
```

Reasonable use of variance may improve the flexibility of your API without having to trade off its type safety. In the following sections, we will discuss the meaning of variance and how it is used with Kotlin generics.

Variance: distinguishing producers and consumers

Generics classes and interfaces can give rise to an unlimited set of types produced by substituting different type arguments instead of their type parameters. By default, all substitutions of a particular type are not considered subtypes of each other, regardless of relationships between their arguments. In this case, we say that the generic type is invariant (relative to some of its type parameters). For example, built-in Array class, mutable collection classes, as well as our TreeNode class, are all invariant. The following example shows that TreeNode<String> is not considered a subtype of TreeNode<Any>:

```
val node: TreeNode<Any> = TreeNode<String>("Hello") // Error
```

On the other hand, some types, like immutable collections, preserve subtyping of their arguments. In the following section we will discuss the language features, which allow you to control how subtyping affects your own generic classes. However, we first need to understand why some generic classes can preserve inheritance while others cannot.

The distinction is based on the way a type handles the values of its type parameter (say, T). All generic types may be divided into three categories:

1. Producers that only have operations which return values of T but never take them as input;
2. Consumers whose operations only take the values of T as input but never return them;
3. All the remaining types, which do not fall into either of the groups above.

It turns out that in general types from the last group, (ones that are neither producers, nor consumers) cannot preserve subtyping without breaking type safety. To understand why this happens, let's consider an example with our TreeNode class. For a moment, let us suppose that subtyping is permitted, and we can assign TreeNode<String> to TreeNode<Any>. Consider the following code:

```
valstringNode = TreeNode<String>("Hello")
valanyNode: TreeNode<Any> = stringNode
anyNode.addChild(123)
val s = stringNode.children.first() // ???
```

Now the problem is clear, since you can add the child of any type to `TreeNode<Any>`, assigning `stringNode` to `anyNode` makes it possible to add `Int` child to an original tree of `String`! If such an assignment was allowed, the program would fail with exception when trying to cast `stringNode.children.first()` to `String`. In other words, we would have violated the contract of `TreeNode<String>` by putting the integer value into one of its children nodes.

Java vs. Kotlin: Readers who are familiar with Java, would recognize a similarity with the infamous `ArrayStoreException`, which may happen due to array assignments. That is, in fact, the reason why in Kotlin array types do not preserve subtyping, as opposed to Java.

When we consider type A to be a subtype of type B, we assume that the values of A can be used in any context that requires a value of B. This is clearly not the case here: type `TreeNode<Any>` has an ability to add child nodes of any type, while `TreeNode<String>` doesn't. It can only add children of type `String`. That's the reason why `TreeNode<String>` cannot be a subtype of `TreeNode<Any>`.

Why are immutable collections like `List<T>` different? The reason is that they do not have operations like `addChild()`: their members only produce values of T, but never consume them. So, the basic contract of `List<Any>` is its ability to retrieve the values of Any. Similarly, the contract of `List<String>` is its ability to retrieve the values of String. But since String is a subtype of Any, that automatically makes `List<String>` capable to retrieve values of Any as well. In other words, subtyping of `List<String>` and `List<Any>` does not endanger type safety, and compiler permits us to make use of this property. We can say that such types are covariant with respect to their type argument. All producer-like types can be made covariant in Kotlin.

Many built-in immutable types like `Pair`, `Triple`, `Iterable`, `Iterator` and so on are covariant. In addition to that, the functional types are covariant with respect to their return types:

```
valstringProducer: () -> String = { "Hello" }
valanyProducer: () -> Any = stringProducer
println(anyProducer()) // Hello
```

Please note that covariance is not the same as immutability. Covariance (with respect to T) just forbids taking values of T as input, so it is possible to have a mutable type that can still be made covariant. For example, consider a putative list that can only delete its elements by index, but cannot add new ones:

```
interface NonGrowingList<T> {  
    val size: Int  
    fun get(index: Int): Int  
    fun remove(index: Int)  
}
```

It is clearly mutable, but behaves covariant. For example, NonGrowingList<String> is capable of everything the NonGrowingList<Any> can do.

The reverse is also true - types representing immutable objects may behave non-covariant. For example:

```
interface Set<T> {  
    fun contains(element: T): Boolean  
}
```

The type above might be immutable, but it's not a producer and hence cannot preserve subtyping. While Set<Any> can take any value as its input, Set<String> can take only strings.

What about consumer-like types? They obviously cannot preserve subtyping in keeping with the arguments above. It turns out, though, that they preserve subtyping in the opposite direction. To understand what it means, let's consider two substitutions of the Set<T> type above - Set<Int> and Set<Number>. The contract of Set<T> can be reduced to an ability to handle elements of T by the contains() function. So, Set<Number> can handle any Number and Set<Int> can handle any Int. But Int is a subtype of Number, so Set<Number> can handle any Int as well. In other words, Set<Number> behaves like a subtype of Set<Int>. In fact, in Kotlin, you can enable this subtyping by declaring T contravariant.

For example, function types are contravariant with respect to their argument types:

```
val anyConsumer: (Any) -> Unit = { println(it) }  
val stringConsumer: (String) -> Unit = anyConsumer  
stringConsumer("Hello") // Hello
```

So for a given generic type $X<T, \dots>$ we have the following options of variance with respect to T:

- X behaves like a producer: In this case we can declare T covariant, so that $X<A>$ will be a subtype of X, whenever A is a subtype of B;
- X behaves like a consumer: We then can make T contravariant: $X<A>$ will be a subtype of X, whenever B is a subtype of A;
- In all the remaining cases, T has to remain invariant;

In the following section we shall see how variance is expressed in Kotlin.

Variance at the declaration site

In Kotlin, variance of a type parameter can be specified in two ways: either in the declaration itself, or on its usage site when substituting type arguments. In this section, we shall focus on the first approach, which is known as ‘declaration-site variance’.

By default, type parameters are considered invariant, which means that their generic types do not preserve subtyping of the corresponding type arguments (as well its reversed version). For example, consider the simplified version of List type, with array-based immutable implementation:

```
interface List<T> {
    val size: Int
    fun get(index: Int): T
}
class ListByArray<T>(private varargval items: T) : List<T> {
    override val size: Int get() = items.size
    override fun get(index: Int) = items[index]
}
```

Suppose that we define a function, which takes a pair of lists and returns their concatenation delegating to either of the original List instances:

```
fun <T>concat(list1: List<T>, list2: List<T>) = object : List<T> {
    override val size: Int
    get() = list1.size + list2.size
```

```

        override fun get(index: Int): T {
            return if (index < list1.size) {
                list1.get(index)
            } else {
                list2.get(index - list1.size)
            }
        }
    }
}

```

Now, everything goes smoothly until we try to use this function to combine lists of related types, say, `List<Number>` and `List<T>`:

```

val numbers = ListByArray<Number>(1, 2.5, 3f)
val integers = ListByArray(10, 30, 30)
val result = concat(numbers, integers) // Error

```

The reason for this is an invariance of parameter `T`, due to which `List<Int>` is not considered a subtype of `List<Int>` (and vice versa). So we cannot pass a `List<Int>` variable into a function that expects `List<Number>`.

However, this is too restrictive. A quick glance at the `List` interface reveals that it actually behaves like a producer type. Its operations only return the values of `T`, but never take them as input. In other words, this type can be safely made covariant. To do this, we mark parameter `T` with `out` keyword:

```

interface List<out T> {
    val size: Int
    fun get(index: Int): T
}

```

Now the `concat()` call works as expected, because the compiler understands that `List<Int>` is a subtype of `List<Number>`.

The producer part is crucial here, because the compiler wouldn't let us define the parameter as covariant otherwise. Let us consider a mutable version of `List`:

```

interface MutableList<T> : List<T> {
    fun set(index: Int, value: T)
}

```

Trying to make `T` in a `MutableList` covariant will lead to a compilation error:

```
interface MutableList<out T> : List<T> {
    fun set(index: Int, value: T) // Error: T occurs in 'in' position
}
```

This happens because of the `set()` function, which takes an input value of `T`, thus acting as its consumer. The basic rule is as follows:

A type parameter may only be declared covariant if all of its occurrences happen to be in out positions, where an out position basically means a usage where its value is produced, rather than consumed - such as return type of a property or function, or covariant type argument of generic type. For example, the following type is valid, since all usages of parameter `T` are in out positions:

```
interface LazyList<out T> {
    // usage as return type
    fun get(index: Int): T
    // usage as out type argument in return type
    fun subList(range: IntRange): LazyList<T>
    // return part of functional type is 'out' position as well
    fun getUpTo(index: Int): () -> List<T>
}
```

Similarly, the in positions cover the usages where values are consumed like arguments of function calls and contravariant type arguments.

Please note, that the constructor parameters are exempt from these checks, because the constructor is called before an instance of generic type exists (it's called to create it in the first place). For that reason, we can make `ListByArray` implementation covariant as well:

```
class ListByArray<out T>(private varargval items: T) : List<T> { ... }
```

Similarly, we can use the `in` keyword to declare a type parameter contravariant. This is possible when its generic type acts as a consumer, that is, the type parameter itself has no usages in the out positions. For example:

```
class Writer<in T> {
    // usages as function argument
```

```

    fun write(value: T) {
        println(value)
    }
    // combining out List argument with in position as
    function argument
    // gives in position again
    fun writeList(values: Iterable<T>) {
        values.forEach { println(it) }
    }
}
fun main() {
    val numberWriter = Writer<Number>()
    // Correct: Writer<Number> can also handle integers
    val integerWriter: Writer<Int> = numberWriter
    integerWriter.write(100)
}

```

The `TreeNode` class from our earlier example cannot be made a covariant or contravariant, since its type parameter has usages in both in (e.g. `addChild()` function) and out positions (like `data` or `children` properties). We have no option, other than that of leaving it as invariant as it was originally. But what if we want to make a copy of a tree with all its children? Then, our `TreeNode` instance acts solely as a producer, since the only members we need for that task are the `data` and `children` properties. Can we somehow convince the Kotlin compiler that `TreeNode` is used covariantly in such a case? The answer is yes, and the language tool we need for that is a use-site variance, also known as a 'projection'.

Use-site variance with projections

Another way to specify a variance, is to place the `out/in` keyword before a type argument in particular, for the usage of generic type. This construct, also called a projection, is useful for types that are invariant in general, but can be used as either producers, or consumers, depending on the context.

Let us suppose that we want to implement a function that adds a copy of the existing tree to another tree, as a child. Let's start with invariant definition:

```

fun <T>TreeNode<T>.addSubtree(node: TreeNode<T>) : TreeNode<T>
{

```

```
    val newNode = addChild(node.data)
    node.children.forEach { newNode.addSubtree(it) }
    return newNode
}
```

This function works well, when both the trees have the same type:

```
fun main() {
    val root = TreeNode("abc")
    val subRoot = TreeNode("def")
    root.addSubtree(subRoot)
    println(root) // abc {def {}}
}
```

But what if we want to, say, add a tree of Int to a tree of Number? This operation is well-defined, since Int is a subtype of Number and adding Int-based nodes to a Number tree does not violate any assumptions about its type. But since `TreeNode<T>` is invariant and we've specified that both trees have the same element type T, the compiler won't let us do it:

```
val root = TreeNode<Number>(123)
val subRoot = TreeNode(456.7) // Error
```

The `TreeNode<T>` type has to remain invariant, since it contains both members that can return values of T (like `data` property) and those that take T values as their input (like `addChild()` function), so we cannot use the declaration-site variance here. However, in the context of `addSubtree()` function, a tree we pass as an argument is used as a producer, exclusively. This allows us to achieve our goal by marking the necessary type argument as out:

```
fun <T>TreeNode<T>.addSubtree(node: TreeNode<out T>): TreeNode<T> {
    val newNode = addChild(node.data)
    node.children.forEach { newNode.addSubtree(it) }
    return newNode
}

fun main() {
    val root = TreeNode<Number>(123)
    val subRoot = TreeNode(456.7)
```

```

    root.addSubtree(subRoot)
    println(root) // 123 {456.7 {}}
}

```

Alternatively, we could have introduced an additional type parameter bounded by the first one, to represent the elements of the added tree:

```

fun <T, U : T>TreeNode<T>.addSubtree(node: TreeNode<U>): TreeNode<T> {
    val newNode = addChild(node.data)
    node.children.forEach { newNode.addSubtree(it) }
    return newNode
}

```

Using out-projection, we can avoid extra type parameters and solve our problem in a concise way.

The `TreeNode<out T>` is called a projected type. The projection `out T` means that we do not know the actual type argument of `TreeNode`, except that it must be a subtype of `T`. You can think of `TreeNode<out T>` as a version of `TreeNode<T>`, which only exposes operations that act as producers with respect to `T`. For example, we can use properties such as `data`, `children`, `depth`, or functions like `walkDepthFirst()`, since they do not take values of `T` as their input. Consumer operations like `addChild()` member or `addChildren()` extension are available, but not actually usable, as any attempt to call them on out-projected type produces a compilation error:

```

fun processOut(node: TreeNode<out Any>) {
    node.addChild("xyz") // Error: addChild() is projected
out
}

```

The in-projections can be used in a similar manner, to enforce the usage of type as a consumer. For example, we could have written our tree adding the function in the following form:

```

fun <T>TreeNode<T>.addTo(parent: TreeNode<in T>) {
    val newNode = parent.addChild(data)
    children.forEach { it.addTo(newNode) }
}

```

Now, the receiver is a tree being added, while the parameter represents its new parent. Thanks to the in-projection, this kind of a function can add `TreeNode<T>` to a tree containing elements of any supertype of `T`:

```
fun main() {
    val root = TreeNode<Number>(123)
    val subRoot = TreeNode(456.7)
    subRoot.addTo(root)
    println(root) // 123 {456.7 {}}
}
```

Java vs. Kotlin: Kotlin projections play the same role as Java extends/super wildcards, essentially. For example, `TreeNode<out Number>` and `TreeNode<in Number>`, are equivalent to Java's `TreeNode<? extends Number>` and `TreeNode<? super Number>`, respectively.

Please note, that using projections, when the corresponding type argument has declaration-site variance, is meaningless. When the projection matches the parameter variance, the compiler reports a warning, since using projection in such a case is redundant. On the other hand, when projections do not match, the compiler considers this a compilation error. For example:

```
interface Producer<out T>{
    fun produce(): T
}

interface Consumer<in T> {
    fun consume(value: T)
}

fun main() {
    val inProducer: Producer<in String>      // Error:
conflicting projection
    val outProducer: Producer<out String> // out is redundant
    val inConsumer: Consumer<in String>      // in is
redundant
    val outConsumer: Consumer<out String> // Error:
conflicting projection
}
```

Just like Java wildcards, projections give you a possibility to use invariant types in a

more flexible way, by representing types constrained by either producer, or the consumer role. Additionally, Kotlin has a special way to denote a generic, whose argument can be replaced by any possible type - a star projection.

Star projections

Star projections, denoted by *, are used to indicate that argument type can be anything within its bounds. Since Kotlin only supports upper bounds for type parameters, this amounts to saying that type argument can be any subtype of the corresponding bounding type. Let's consider an example:

```
// Can be any list since its element type is only bounded by Any?  
val anyList: List<*> = listOf(1, 2, 3)  
// Can be any object comparable with itself (due to T : Comparable<T> bound)  
val anyComparable: Comparable<*> = "abcde"
```

In other words, a star projection effectively behaves like an out-projection applied to a type parameter bound.

Java vs. Kotlin: Star projection can be considered as a Kotlin counterpart of Java's ? wildcard, so `TreeNode<*>` in Kotlin, has basically the same meaning as `TreeNode<?>` in Java.

In the section on type erasure and refinement, we have seen that star-projected types can be used in type-checking operations:

```
val any: Any = ""  
any is TreeNode<*>
```

Since the type parameter of `TreeNode` is bounded by `Any?`, we can also write this using an explicit out projection:

```
any is TreeNode<out Any?> // Ok
```

However, if we try to replace `Any?` with some other type, the compiler will report an error, since such a check is impossible due to type erasure:

```
any is TreeNode<out Number> // Error
```

It is important to mind the difference between * and using type parameter bound as

non-projection argument, like in `TreeNode<*>` vs. `TreeNode<Any?>`. While `TreeNode<Any?>` is a tree that can contain the value of any type, `TreeNode<*>` represents the tree whose nodes are characterized by the same common type `T`, but that `T` is unknown to us. For that reason, we cannot use `TreeNode` operations that behave like consumer of `T` values. Since we do not know the actual type, we also do not know what values are acceptable for them. That is exactly the meaning of out-projection that we have discussed in the previous section.

To explain it briefly, star projections allow you to concisely represent generic type when particular arguments are not relevant or simply not known.

Please note, that when type parameter has more than one bound, `*` cannot be replaced with an explicit out projection. This is because type intersection is not denotable in the Kotlin source code:

```
interface Named {
    val name: String
}
interface Identified {
    val id: Int
}
class Registry<T> where T : Named, T : Identified
// the bound is intersection of Named and Identified
var registry: Registry<*>? = null
```

Another difference between `*` and explicit out is that, `*` are allowed for type parameters with declaration-site variance. In this case, the compiler doesn't report warnings/errors:

```
interface Consumer<in T> {
    fun consume(value: T)
}
interface Producer<out T> {
    fun produce(): T
}
fun main() {
    val starProducer: Producer<*> // the same as
    Producer<Any?>
    val starConsumer: Consumer<*> // the same as
```

```
Consumer<Nothing>
}
```

When applied to type argument in contravariant position (like in `Consumer<*>`) , star projection produces a type argument of Nothing. Thus we cannot pass anything to `consume()` function, because Nothing has no values.

Type aliases

In conclusion, we are going to discuss a language feature that is not directly related to generics, but comes very handy when you have to deal with complex generic types - the type aliases.

The reason for type aliases being added in Kotlin 1.1 is to give you a way to introduce alternative names for existing types. The primary goal of such a construct is to provide short names for otherwise long types, such as generic or functional ones. The definition of type alias is introduced with the `typealias` keyword, which is followed by an alias name and its definition separated by the `=` symbol:

```
typealias IntPredicate = (Int) -> Boolean
typealias IntMap = HashMap<Int, Int>
```

Now we can use the names above, instead of the right-hand sides of their definitions:

```
fun readFirst(filter: IntPredicate) =
    generateSequence{ readLine()?.toIntOrNull()
}.firstOrNull(filter)
fun main() {
    val map = IntMap().also {
        it[1] = 2
        it[2] = 3
    }
}
```

Another useful case is providing short names for nested classes:

```
sealed class Status {
    object Success : Status()
    class Error(val message: String) : Status()
}
```

```
typealias StSuccess = Status.Success  
typealias StError = Status.Error
```

Type aliases may have type parameters which allow us to introduce aliases for generic types, very similar to classes,:

```
typealias ThisPredicate<T> = T.() -> Boolean  
typealias MultiMap<K, V> = Map<K, Collection<V>>
```

You may also restrict their scope by using visibility modifiers:

```
private typealias MyMap = Map<String, String> // visible in  
current file only
```

Currently, (Kotlin 1.3) type aliases may only be introduced at the top-level. For example, it is not possible to declare them inside functions or as class members:

```
fun main() {  
    typealias A = Int // Error  
}
```

Another restriction is that you cannot declare bounds or constraints for type parameters of a generic type alias:

```
typealias ComparableMap<K : Comparable<K>, V> = Map<K, V> //  
Error
```

The important thing to note here is that, type aliases never introduce new types, they just give an additional way to refer to existing ones. This means that type aliases are completely interchangeable with their original types:

```
typealias A = Int  
fun main() {  
    val n = 1  
    val a: A = n  
    val b: Int = a  
}
```

As you already know, type aliases are not the only way to introduce new names for existing types, so it is useful to understand the major differences between language features that can be used for similar purpose.

For example, import aliases give you the ability to introduce alternative names as a part of import directives. They also support functions and properties like type aliases, but do not allow you to introduce generic aliases. Besides, their scope is always limited to the containing file, while public type aliases have a wider scope.

It is also possible to introduce a new type name by inheriting from a generic or functional type. This option allows you to define generic types, as well as control the new name visibility. The major difference with type aliases is that, these kind of definitions create a new type, namely a subtype of the original one, so their compatibility is one-way:

```
class MyMap<T> : HashMap<T, T>()
fun main() {
    val map: Map<String, String> = MyMap() // Ok, MyMap is
    subtype of Map
    val myMap: MyMap<String> = map           // Error
}
```

Besides, while you cannot inherit from a final class, you can introduce an alias for it.

Inline classes are also like type aliases in a sense that they may have the same runtime representation as their original type. However, the crucial difference is that, inline classes introduce new types that are not compatible with their originals. For example, a value of `UIInt` cannot be assigned to the variable of `Int` (and vice versa), without an explicit conversion.

Conclusion

This chapter has brought to us, the concept of generics, which gives you an additional tool for designing abstractions in the Kotlin code. Now you should be able to design your own generic APIs, as well as use more advanced concepts like reified type parameters and variance for writing more concise, efficient and type-safe code now. Among other things, we have introduced a useful feature of type aliasing that allows you to introduce alternative type names and can simplify handling of complicated generic and functional types.

In the following chapter, we are going to take a closer look at two interrelated topics. The first one would be annotations, which allow you to specify various metadata for your program elements. In Kotlin, annotations, among others, are used for fine-tuning of interoperability with code, which we will also cover in the *Chapter 12, Java*

Interoperability. The second major topic of *Chapter 10, Annotations and Reflection*, would be reflection that gives you an API to introspect a program structure and dynamically invoke your code.

Questions

1. How can you define a generic class, function or property in Kotlin?
2. Describe how to specify constraints for type parameters. How do they compare to Java's?
3. What is type erasure? Describe the limitations of type parameters vs. ordinary types.
4. How can you circumvent type erasure using reified type parameters? What are their limitations?
5. What is a variance? Why is variance important for a generic code?
6. Describe how declaration-site variance is used in Kotlin.
7. Compare use-site variance in Kotlin with Java wildcards
8. Describe the purpose of star projections.
9. Describe the type alias syntax. How do they compare with the related languages' features such as import aliases and inheritance?