Kapitel 1
Hello World!

Traditionell ist *Hello World!* das erste Programm in jeder Programmieranleitung bzw. in jedem Programmierbuch. Die Aufgabe dieses Programms besteht darin, die Zeichenkette »Hello World!« auf dem Bildschirm bzw. in einem Terminalfenster auszugeben.

Tatsächlich besteht der Sinn des Hello-World-Programms natürlich nicht darin, eine Zeichenkette auszugeben, sondern vielmehr darin, die Syntax und Werkzeuge einer neuen Programmiersprache erstmals auszuprobieren. Dementsprechend geht es auch in diesem Kapitel weniger um Code, sondern vielmehr darum, woher die Programmiersprache Kotlin stammt und mit welchen Werkzeugen Kotlin-Code verfasst und ausgeführt wird.

1.1 Über Kotlin

Die erste Version der Programmiersprache Kotlin wurde 2011 veröffentlicht. Seit 2016 gilt Kotlin als stabil. Die Sprache Kotlin wurde und wird vor allem von der Firma JetBrains entwickelt. Diese Firma ist für ihr großes Angebot an Entwicklungsumgebungen bekannt: *IntelliJ IDEA* und *Android Studio* für Java und Kotlin, *PyCharm* für Python, *Rider* für .NET-Sprachen, *DataGrip* für Datenbanken und SQL etc.

Der Name »Kotlin«

Die tschechische Firma JetBrains hat in mehreren Ländern Niederlassungen. Kotlin wurde von dem in St. Petersburg ansässigen IntelliJ-Entwicklungsteam konzipiert. Die neue Programmiersprache wurde nach der vor der Stadt gelegenen Insel *Kotlin* benannt. Die Namensgebung ist also ein dezenter Hinweis auf Java: Dieser Name bezeichnet ja auch gleichermaßen eine Programmiersprache und eine Insel.

JetBrains hat Kotlin mit dem Anspruch entworfen, ein »besseres Java« zu schaffen. Das ist in vielerlei Hinsicht gelungen: Einerseits ist die Syntax deutlich klarer, andererseits bietet Kotlin eine Menge Features, die in Java fehlen oder erst später realisiert wurden und daher nicht mehr perfekt in die Sprache integriert werden konnten.

Naturgemäß hatte JetBrains den großen Vorteil, dass es sowohl aus den Design-Fehlern Javas lernen als auch Features von anderen Sprachen übernehmen bzw. adaptieren konnte. (Die erste Version von Java erschien 1996, also 15 Jahre vor Kotlin. In IT-Maßstäben gerechnet, stammt Java aus der Steinzeit …)

Gleichzeitig wollte JetBrains aber nicht das Rad neu erfinden. Deswegen hat sich die Firma entschieden, Kotlin zwar nicht syntaktisch, wohl aber auf einer unteren Ebene kompatibel zu Java zu machen: Kotlin-Code wird normalerweise zu einem sogenannten »Bytecode« kompiliert, der von jeder Java Virtual Machine (JVM) ausgeführt werden kann. Das hat gleich zwei Vorteile:

- Zum einen setzt Kotlin auf ein bewährtes, extrem weit verbreitetes Fundament auf. Kotlin-Programme laufen damit überall, wo auch Java-Programme ausgeführt werden können.
- Zum anderen kann Kotlin auf alle für Java entwickelten Bibliotheken zurückgreifen. Damit steht Ihnen ein riesiges Software-Angebot zur Auswahl. Gleichzeitig unterstützt diese Kompatibilität die schrittweise Migration eines Projekts von Java nach Kotlin.

Sehr beliebt ist Kotlin gegenwärtig bei Android-App-Entwicklern. Das liegt auch daran, dass sich das Android-Universum zunehmend schwer damit tut, aktuelle Java-Versionen zu unterstützen. (Der von Oracle eingeführte halbjährliche Release-Zyklus macht die Sache nicht gerade einfacher.) Kotlins moderne Sprachfeatures funktionieren dagegen selbst dann, wenn das Kotlin-Programm auf uralten Java Virtual Machines ausgeführt wird. Insofern erleichtert Kotlin den Entwickleralltag erheblich.

Google hat vielleicht noch einen Grund, sich über den Erfolg von Kotlin zu freuen: Ein jahrelanger Rechtsstreit zwischen Oracle und Google über die Java-APIs hat die Atmosphäre zwischen den beiden Technologie-Giganten vergiftet. Google ist vermutlich nicht traurig, wenn sich das Android-Ökosystem dank Kotlin weiter von Java löst.

Kotlin nur als Sprache für App-Entwickler zu bezeichnen, greift aber zu kurz. Kotlin kommt auch serverseitig (im sogenannten »Backend«) immer öfter zum Einsatz, oft ergänzend zu bereits vorhandenem Java-EE-Code.

Dokumentation

Kotlin ist im Internet umfassend dokumentiert. Ich weise hier nur auf die wichtigsten Seiten hin:

https://kotlinlang.org/docs/reference
https://kotlinlang.org/docs/reference/faq.html
https://stackoverflow.com/questions/tagged/kotlin

Kotlin JS und Kotlin/Native

Kotlin-Programme können auch zu JavaScript-Code bzw. nativ kompiliert werden. Das hat den Vorteil, dass Kotlin-Programme unabhängig von einer JVM ausgeführt werden können. Damit ist aber natürlich der Nachteil verbunden, dass weder die Java-Standardbibliothek aus dem JDK noch irgendeine eine andere Java-Klassenbibliothek genutzt werden kann. Selbst innerhalb der Kotlin-Klassenbibliothek gibt es Einschränkungen, weil auch dort nicht jede Funktion/Methode/Klasse für jede Plattform (also JVM/JS/Native) zur Verfügung steht.

In diesem Buch gehe ich davon aus, dass Sie Ihre Kotlin-Programme mit einer JVM ausführen.

Open-Source-Lizenz

JetBrains hat für Kotlin die liberale Open-Source-Lizenz Apache 2.0 gewählt. Die Lizenz erlaubt die Verwendung von Kotlin auch für kommerzielle Produkte und zwingt Sie nicht dazu, eigenen Code selbst wieder als Open-Source-Code freizugeben. (Das gilt nicht nur für in Kotlin entwickelte Programme, sondern auch für Produkte, die direkt von Kotlin abgeleitet sind – also z. B. eine weitere Programmiersprache.) Die Lizenzbestimmungen finden Sie unter:

https://www.apache.org/foundation/license-faq.html

Der Quellcode von Kotlin ist auf GitHub verfügbar:

https://github.com/jetbrains/kotlin

1.2 Installation

Allererste Experimente mit Kotlin können Sie ohne jede Installation auf der Webseite *https://try.kotlinlang.org* durchführen. Um Kotlin-Programme auf Ihrem eigenen Rechner auszuführen, müssen Sie aber Java und eine Entwicklungsumgebung installieren. Dieser Abschnitt gibt dazu einige Tipps.

Welche Entwicklungsumgebung?

Die ideale Entwicklungsumgebung zum Erlernen von Kotlin ist definitiv *IntelliJ*. Damit können Sie unkompliziert und mit minimalem Overhead kleine Testprogramme entwickeln.

Mit der Programmierung erster Android-Apps warten Sie am besten, bis Sie Kotlin ein wenig kennengelernt haben und mit seiner Syntax vertraut sind. Dann können Sie entweder bei IntelliJ bleiben und dessen Android-Plugin verwenden oder auf *Android Studio* umsteigen.

> Intern verwenden IntelliJ und Android Studio dieselbe Basis und sind insofern weitestgehend kompatibel zueinander. IntelliJ hat den Vorteil, universeller verwendbar zu sein. Demgegenüber ist Android Studio stärker darauf fokussiert, Apps zu entwickeln. Weitere Informationen dazu finden Sie unter:
>
> *https://blog.jetbrains.com/idea/2013/05/intellij-idea-and-android-studio-faq*
> *https://stackoverflow.com/questions/30779596*
>
> Ich empfehle Ihnen, *beide* Entwicklungsumgebungen zu verwenden. Für die ersten Kapitel dieses Buchs, in denen es darum geht, die Syntax von Kotlin kennenzulernen, ist IntelliJ die erste Wahl. IntelliJ hilft Ihnen, sich auf Kotlin zu konzentrieren, ohne vom Android-Overhead erschlagen zu werden.
>
> Sobald Sie in Kapitel 21, »Hello Android!«, mit der App-Programmierung starten, wechseln Sie zu Android Studio. Dieses Programm ist aktuell der De-facto-Standard für alle Android-App-Entwickler.

Java (JDK-Installation)

Wie gesagt verwendet Kotlin Java als Fundament. Aus diesem Grund muss zur Entwicklung von Kotlin-Code auf jeden Fall ein *Java Development Kit* (JDK) auf Ihrem Rechner installiert sein.

Prinzipiell begnügt sich Kotlin dabei mit relativ alten Versionen (Java 8 oder sogar Java 6). Sinnvoller ist allerdings die Installation eines aktuellen JDKs. Ich habe meine Tests auf der Basis von Java 11 LTS durchgeführt.

Der Download des JDKs bei Oracle ist in den vergangenen Jahren zunehmend mühsam geworden. Oracle verlangt eine (kostenlose) Registrierung. Danach ist zwar ein Download möglich, Updates erhalten aber nur zahlende Kunden.

Insofern sollten Sie einen der vielen alternativen Anbieter in Erwägung ziehen, die vollständig kompatible JDKs auf Basis von *OpenJDK* zusammenstellen. Das OpenJDK basiert auf dem Open-Source-Code von Java. In vielen Linux-Distributionen kommt das OpenJDK standardmäßig zum Einsatz.

Downloads sowie weitere Informationen finden Sie hier:

https://www.oracle.com/technetwork/java/javase/downloads/index.html
https://adoptopenjdk.net
https://aws.amazon.com/de/corretto
https://stackoverflow.com/questions/52431764

Toolbox (Installation von JetBrains-Produkten)

Anstatt IntelliJ und/oder Android Studio manuell herunterzuladen und einzurichten, sollten Sie einmalig die ungemein praktische *Toolbox* installieren. Dieses winzige Programm der Firma JetBrains unterstützt Sie in der Folge dabei, diverse Entwicklungsumgebungen zu installieren und später auch zu aktualisieren (siehe Abbildung 1.1). Aus meiner Sicht ist die Toolbox absolut empfehlenswert und spart eine Menge Zeit und Mühe. Diese Empfehlung gilt für alle Betriebssysteme, aber ganz besonders für Linux, wo die manuelle Installation bzw. die Durchführung von Updates vergleichsweise mühsam ist.

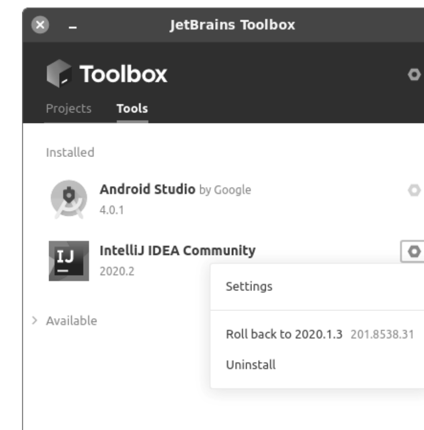

Abbildung 1.1 Die Toolbox erleichtert die Installation und Aktualisierung von IntelliJ, Android Studio und anderen JetBrains-Produkten.

Für Windows und macOS finden Sie unter *https://www.jetbrains.com/toolbox* jeweils ein einfaches Setup-Programm für die Toolbox. Wenn Sie unter Linux arbeiten, laden Sie ein komprimiertes TAR-Archiv herunter. Es enthält als einzige Datei den Binärcode des Programms. Mit den folgenden Terminalkommandos packen Sie das Archiv aus und starten das Programm erstmalig:

```
cd Downloads
tar xzf jetbrains-toolbox-<nnn>.tar.gz
./jetbrains-toolbox-<nnn>/jetbrains-toolbox
```

Die Toolbox installiert sich dann selbstständig in das folgende Verzeichnis (relativ zum Home-Verzeichnis):

```
AppData\Local\JetBrains                  (Windows)
.local/share/JetBrains                   (Linux)
Library/Application Support/JetBrains    (macOS)
```

Im JetBrains-Verzeichnis landen später auch IntelliJ, Android Studio sowie gegebenenfalls weitere JetBrains-Programme.

Sie können die Toolbox in der Folge im Dock verankern (unter Gnome mit dem Kommando ZU FAVORITEN HINZUFÜGEN) bzw. in den Einstellungen einen automatischen Start beim Login aktivieren. Im Toolbox-Fenster verwalten Sie nun alle installierten JetBrains-Produkte und können bei Bedarf unkompliziert Updates durchführen.

Achten Sie bei der Installation von IntelliJ darauf, dass Sie die kostenlose *Community*-Version auswählen. Die *Ultimate Edition* stellt zusätzliche Funktionen zur Verfügung, die sich aber speziell an professionelle Entwickler richten und zum Erlernen von Kotlin nicht erforderlich sind.

IntelliJ manuell installieren

Falls Sie sich gegen die Toolbox-Variante entscheiden, laden Sie die Intellij-IDEA (*Integrated Development Environment Application*, im Folgenden kurz *IntelliJ*) von dieser Webseite herunter:

https://www.jetbrains.com/idea/download

Auch für den Download gilt: Die kostenlose Community Edition ist ausreichend. Unter macOS und Windows ist die Installation selbsterklärend. Unter Linux müssen Sie das `.tar.gz`-Archiv in einem beliebigen Verzeichnis auspacken. Der erste Start muss aus dem Terminal heraus erfolgen. Dabei wird IntelliJ so verankert, dass das Programm in Zukunft auch über das Startmenü ausgeführt werden kann.

```
cd
tar xzf Downloads/ideaIC-<nnn>.tar.gz
./idea-IC-<nnn>/bin/idea.sh
```

Unter Ubuntu Linux können Sie IntelliJ auch mit dem Programm *Ubuntu Software* als Snap-Paket installieren. Das ist bequem und hat den Vorteil, dass Sie in Zukunft alle Updates automatisch erhalten. Die Snap-Installation ist allerdings auch mit vielen Nachteilen verbunden (größerer Platzbedarf, Probleme mit Zugriffsrechten etc.). Sofern Ihr Linux-Wissen ausreicht, um die oben skizzierte manuelle Installation durchzuführen, empfehle ich Ihnen ausdrücklich diesen Weg!

1.3 »Hello World!« mit und ohne IDE ausführen

Sie können erste Kotlin-Experimente ohne jede Installation im Webbrowser ausführen (siehe Abbildung 1.2). Auf der Website *https://play.kotlinlang.org* finden Sie zudem viele Beispiele, die mit der Syntax von Kotlin vertraut machen.

Abbildung 1.2 Kotlin im Webbrowser ausprobieren

> **Übungen zum Erlernen von Kotlin**
>
> Wenn Sie die grundsätzliche Syntax von Kotlin anhand von Übungen erlernen möchten, werfen Sie einen Blick auf die Seite *https://play.kotlinlang.org/koans/overview*. Rund 40 als Rätsel formulierte Aufgaben machen Sie mit der Syntax von Kotlin vertraut. Die Aufgaben setzen allerdings schon etwas Routine mit einer objektorientierten Programmiersprache voraus, idealerweise mit Java.

»Hello World!« in IntelliJ

In IntelliJ starten Sie ein neues Projekt mit FILE • NEW • PROJECT. Im ersten Schritt des Assistenten wählen Sie in der linken Spalte den Typ KOTLIN und in der rechten Spalte PROJEKT TEMPLATE: JVM • CONSOLE APPLICATION aus (siehe Abbildung 1.3). Das BUILD-SYSTEM belassen Sie für einfache Testprogramme bei der Voreinstellung INTELLIJ. Außerdem müssen Sie Ihrem Projekt natürlich einen Namen geben und ein Verzeichnis auswählen, in dem es gespeichert werden soll.

> **Build-System**
>
> Das *Build-System* ist dafür verantwortlich, aus allen Code-Dateien und Bibliotheken das fertige Programm zu kompilieren. In einfachen Fällen kümmert sich IntelliJ selbst um diesen Prozess.

Erst wenn Sie in einem Projekt externe Bibliotheken nutzen wollen, müssen Sie ein externes Build-System verwenden. IntelliJ stellt Ihnen *Gradle* oder *Maven* zur Wahl. In diesem Buch gehe ich nur auf Gradle ein. Was Gradle ist und worin die Unterschiede zwischen den beiden Varianten Gradle Kotlin und Gradle Groovy bestehen, erkläre ich Ihnen in Anhang A.2, »Gradle«.

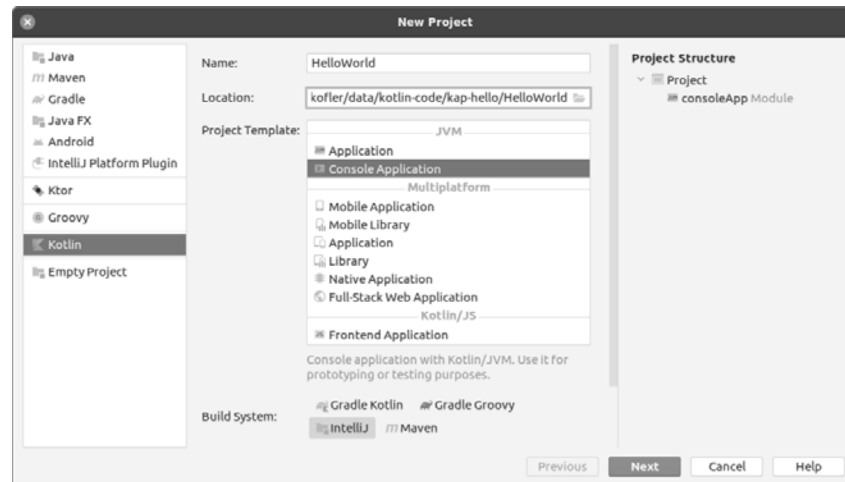

Abbildung 1.3 Ein neues Kotlin-Projekt in IntelliJ erzeugen

Im zweiten Schritt stellen Sie Template: Console Application und Test framework: None ein (siehe Abbildung 1.4). Die erste Einstellung bewirkt, dass das neue Projekt bereits eine `main`-Funktion enthält und sofort getestet werden kann. Die zweite Option gibt an, dass Sie keine automatisierten Unit-Tests nutzen wollen. Als Target JVM version geben Sie eine Version an, die nicht höher ist als die des JDKs auf Ihrem Rechner. (Umgekehrt bedeutet das: Wenn Sie hier 8 angeben, läuft Ihr Programm später auf jedem JDK ab der Version 8.)

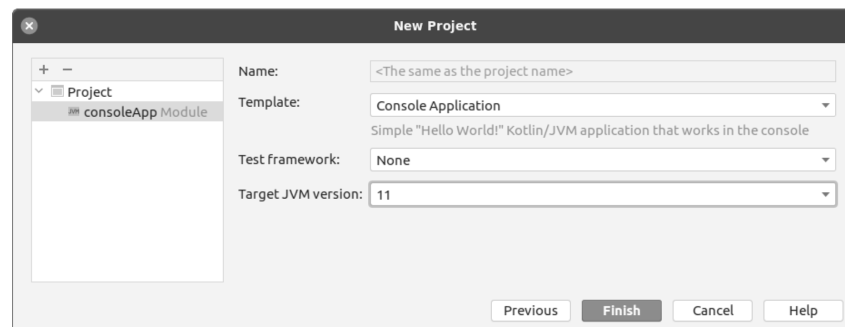

Abbildung 1.4 Ein neues Kotlin-Projekt in IntelliJ erzeugen (Schritt 2)

IntelliJ erzeugt nun ein neues Projekt, dessen einzige Datei aber gut im Verzeichnis `src/main/kotlin/main.kt` versteckt ist (siehe Abbildung 1.5). Die Datei enthält bereits den fertigen Hello-World-Code. Unter Umständen zeigt IntelliJ auch die Fehlermeldung Projekt SDK is not defined aus. In diesem Fall klicken Sie auf den Button Setup SDK und wählen das auf Ihrem Rechner installierte JDK aus.

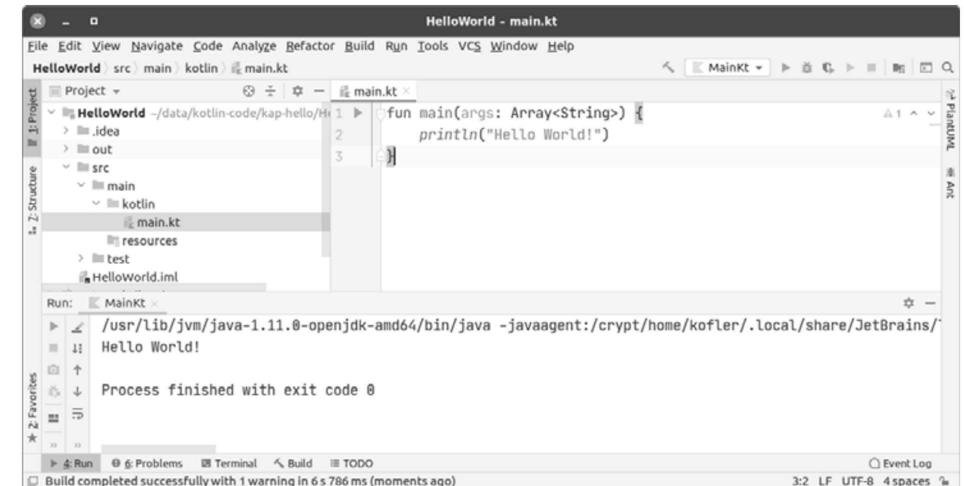

Abbildung 1.5 »Hello World!« in IntelliJ

Falls Sie beim Einrichten des Projekts die Option Template: Console Application übersehen haben, müssen Sie die Code-Datei selbst einrichten. Dazu klappen Sie im Teilfenster Project das Projekt auf, wählen das dort befindliche Verzeichnis `src/main/kotlin` aus und führen Sie dann File • New • Kotlin File/Class aus. (Die Aktion kann auch über ein entsprechendes Kontextmenü des `src`-Eintrags ausgeführt werden.) Sie können die neue Kotlin-Datei nach Gutdünken benennen. Für den Startpunkt eines Projekts sind Namen wie `Main` oder `App` üblich. IntelliJ fügt dem Dateinamen automatisch die Kennung `.kt` hinzu.

In die neue Code-Datei fügen Sie nun die folgenden drei Zeilen mit dem Hello-World-Code ein:

```
fun main() {
    println("Hello World!")
}
```

Dazu kurz einige Erläuterungen:

- `fun` leitet die Definition einer Funktion oder Methode ein.
- Die `main`-Funktion gilt immer als Startpunkt für ein Kotlin-Programm. Sie kann optional den Parameter `args: Array<String>` enthalten. Damit können beim Auf-

ruf des Programms aus einem Terminal oder einer Konsole Daten übergeben werden.

Für dieses Beispiel können Sie den `args`-Parameter entfernen. Wenn Sie das nicht tun, weist IntelliJ Sie darauf hin, dass Sie diesen Parameter ignorieren. Diese Warnung stört aber nicht und beeinträchtigt auch die Funktion Ihres Programms nicht.

- `println` gibt den zwischen den runden Klammern angegebenen Text am Bildschirm aus.

Jetzt gilt es noch, den Code zu kompilieren und auszuführen. Dazu klicken Sie auf den winzigen grünen Button, der im Code-Fenster links von `fun main` eingeblendet ist.

Sobald Run das erste Mal funktioniert, merkt sich IntelliJ, dass `main.kt` die Datei mit dem Startpunkt Ihres Projekts ist. In Zukunft reicht es aus, einfach auf das Run-Dreieck in der Symbolleiste zu klicken, wenn Sie Ihr Programm neuerlich starten möchten. Alle Ausgaben erscheinen im Teilfenster Run (siehe Abbildung 1.5).

Kotlin-Interpreter

Wenn Sie rasch einige Kotlin-Kommandos ausprobieren möchten, ist das REPL-Fenster von IntelliJ eine große Hilfe (siehe Abbildung 1.6). Sie öffnen dieses Teilfenster mit Tools • Kotlin • REPL. Das Akronym REPL steht dabei für *read-eval-print loop*. Innerhalb des REPL-Fensters erwartet IntelliJ einzelne Kotlin-Anweisungen, die Sie dann mit Strg+↵ ausführen können.

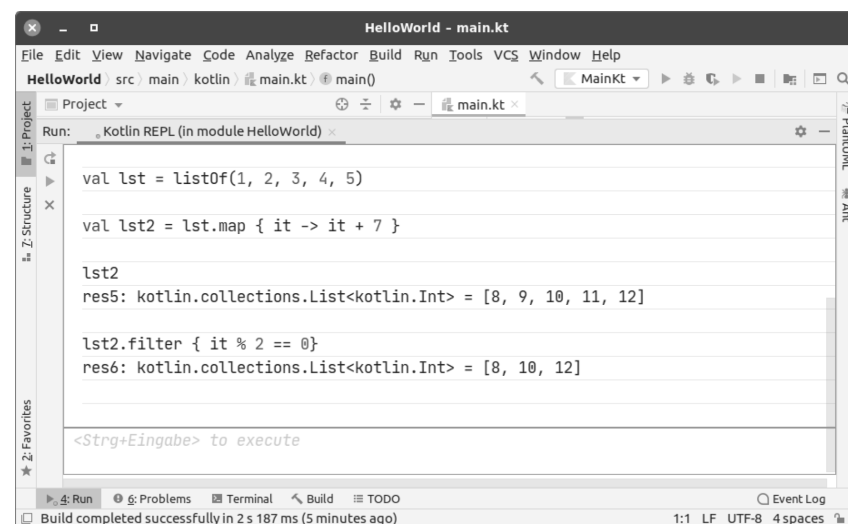

Abbildung 1.6 Ein einfacher Kotlin-Interpreter in IntelliJ

Kapitel 15
Exceptions

Was passiert, wenn Ihr Code auf ein Listenelement zugreifen möchte, das es gar nicht gibt, die Zeichenkette `"xy"` in eine Zahl umzuwandeln versucht oder eine Datei öffnen möchte, für die das Programm keine Leserechte hat? Solange Sie sich nicht um eine Fehlerabsicherung kümmern, zeigt die Entwicklungsumgebung eine Fehlermeldung an und beendet das Programm. Bei einer Android-App ist die Sache noch trister: Die App stürzt ohne jeden Hinweis auf den Fehler ab.

Hinter den Kulissen verwendet Kotlin ein ähnliches Fehlerkonzept wie Java: Das Programm erzeugt ein Objekt, das den Fehler beschreibt, und wechselt in einen speziellen Ausnahmezustand, d. h. in eine sogenannte *Exception*. Umgangssprachlich ist meist die Rede davon, dass das Programm eine Exception auslöst oder *wirft* (wörtlich übersetzt aus der englischen Nomenklatur: *to throw an exception*).

Dieses Kapitel erläutert, was Exceptions sind, wie Sie selbst derartige Fehlerzustände auslösen und wie Sie Ihren Code gegen Fehler absichern können. Vor allem der letzte Aspekt ist natürlich essenziell für jedes Programm, das Sie nicht nur zum Vergnügen oder zu Lernzwecken entwickeln, sondern das später tatsächlich von anderen Personen genutzt werden soll. Dazu verpacken Sie den fehleranfälligen Code in eine `try-catch`-Konstruktion. Das gibt Ihnen die Möglichkeit, ein Programm bzw. eine App trotz eines Fehlers fortzusetzen.

Keine Checked Exceptions

In Java gibt es zwei Arten von Exceptions:

- Bei den sogenannten *Unchecked Exceptions* (z. B. bei einer `IndexOutOfBoundsException`) ist die Fehlerabsicherung optional.
- Beim Aufruf einer Methode, die eine *Checked Exception* (z. B. eine `FileNotFoundException`) auslösen kann, sind Sie als Programmierer(in) dagegen gezwungen, sich um die Fehlerabsicherung zu kümmern: Entweder sichern Sie Ihren Code direkt durch `try-catch` ab oder Sie kennzeichnen Ihre eigene Methode mit `throws` und delegieren so die Absicherung an eine höhere Ebene.

Kotlin hat diese Unterscheidung nicht von Java übernommen, vielmehr sind alle Exceptions *unchecked*. (Im weiteren Verlauf dieses Buches betone ich das nicht mehr, ich schreibe einfach »Exceptions«.) Somit ist in Kotlin die Absicherung von Code immer freiwillig.

Ist diese Vereinfachung in Kotlin nun ein Vorteil oder ein Nachteil? Vor allem Entwickler, die von Java zu Kotlin wechseln, sind diesbezüglich zwiegespalten. Persönlich habe ich den Zwang zur Fehlerabsicherung immer als eine Art Hinweis betrachtet: »Vorsicht, hier kann etwas schiefgehen!« Gerade in Testprogrammen, wo die »Absicherung« in der Regel ohnedies nur in Form einer `println`-Anweisung erfolgt, ist es aber durchaus angenehm, dass der `try-catch`-Zwang aufgehoben ist.

Naturgemäß hindert Sie niemand daran, Kotlin-Code genau wie Java-Code abzusichern. Aber weil Kotlin Sie nicht dazu zwingt, besteht die Gefahr, dass Sie auf eine Absicherung vergessen – weil Ihnen gar nicht klar ist, dass an dieser Stelle ein Fehler passieren könnte.

Der Grund für den entspannteren Umgang mit Fehlern in Kotlin hat damit zu tun, dass viele Leute sowohl in der Java Community als auch außerhalb große Zweifel haben, ob *Checked Exceptions* eine gute Idee sind. Diverse andere Programmiersprachen haben dieses Konzept nicht von Java übernommen. Hintergrundartikel zu diesem Thema finden Sie hier:

https://kotlinlang.org/docs/reference/exceptions.html
https://stackoverflow.com/questions/47733803

15.1 Fehlerabsicherung

Im Folgenden gehe ich davon aus, dass Sie die Funktion `inputDouble` verwenden möchten, die Ihnen bei der Eingabe von Fließkommazahlen hilft. Diese Funktion sei unveränderlich vorgegeben (z. B. als Teil einer Bibliothek, die Sie nutzen):

```
// Beispielprojekt try-catch
fun inputDouble(msg: String): Double {
    print("$msg ")
    val inp = readLine()  // Datentyp String?
    return inp!!.toDouble()
}
```

Die Funktion zeigt den Text `msg` an, wartet dann auf eine Eingabe und erzwingt schließlich deren Umwandlung in eine `Double`-Zahl. Dabei können zwei Dinge schiefgehen:

- Wenn die Eingabe sich nicht als Fließkommazahl interpretieren lässt (z. B. eine leere Eingabe oder `"abc"`), tritt eine `NumberFormatException` aus. Beachten Sie, dass `toDouble` nicht lokalisiert ist, also das US-Fließkommaformat mit einem Dezimalpunkt erwartet.
- Wenn die Eingabe leer ist (also `inp == null` gilt), dann löst der Operator `!!` eine `KotlinNullPointerException` auf. Das ist zum Glück unwahrscheinlich: Dieser Fall kann nur eintreten, wenn die Standardeingabe auf eine Datei umleitet und das Ende dieser Datei erreicht wurde.

Wenn Sie `inputDouble` selbst entwickeln, würden Sie die Funktion natürlich besser implementieren, beispielsweise so:

```
fun safeInputDouble(msg: String): Double {
    print("$msg ")
    // Endlosschleife, bis eine gültige Eingabe vorliegt
    while(true) {
        val inp = readLine()
        if (inp == null)
            // Ende der Eingabedatei erreicht, weitere Versuche
            // sind zwecklos, not-a-number zurückgeben
            return Double.NaN
        else {
            val result = inp.toDoubleOrNull()
            if (result != null)
                return result
        }
        print("Ungültig. Bitte wiederholen Sie die Eingabe! ")
    }
}
```

Aber, wie gesagt, wir nehmen an, dass Sie auf `inputDouble` keinen Einfluss haben. Sie können daher nur die Codeteile absichern, in denen Sie `inputDouble` aufrufen.

»try-catch«

Die Syntax zum Absichern von Code, in dem Fehler auftreten können, ist ganz einfach: Sie verpacken den Code in einen `try`-Block und reagieren auf einen möglichen Fehler im `catch`-Block.

```
try {
  // fehleranfälliger
  // Code
} catch (e: Exception) {
  // wird nur ausgeführt, wenn oben ein Fehler
  // aufgetreten ist; e enthält ein Exception-Objekt
  // mit der Beschreibung des Fehlers
}
// in jedem Fall geht es hier weiter!
```

Ganz egal, ob ein Fehler auftritt oder nicht: Der Code wird auf jeden Fall mit der nächsten Anweisung nach der `try-catch`-Konstruktion fortgesetzt!

Wenn Sie `inputDouble` verwenden möchten, um den Flächeninhalt eines Rechtecks auszurechnen, können Sie also so vorgehen:

```
// Beispielprojekt try-catch
try {
    val l = inputDouble("Länge:")
    val b = inputDouble("Breite:")
    println("Flächeninhalt: ${l*b}")
} catch (e: Exception) {
    println("Es ist ein Fehler aufgetreten: $e")
    e.printStackTrace()
}
println("Hier geht's weiter, auch nach einem Fehler.")
```

Die Variable `e`, die Sie natürlich auch anders nennen können, enthält im Fall eines Fehlers ein Objekt, das von der Klasse `Exception` abgeleitet ist. Die Umwandlung in eine Zeichenkette liefert den genauen Fehlertyp (z. B. `NumberFormatException`) samt einer kurzen Fehlerbeschreibung. Die Methode `printStackTrace` gibt aus, wo im Code der Fehler passiert ist und welche Abfolge von Funktions- und Methodenaufrufen zu dieser Stelle geführt hat.

Reaktion auf unterschiedliche Fehler

Wenn im `try`-Block verschiedene Fehler auftreten können und Sie auf diese Fehler auf unterschiedliche Weise reagieren möchten, geben Sie entsprechend mehrere `catch`-Blöcke an:

```
try {
    // fehleranfälliger Code
} catch (e: Exception1) {
    // Reaktion auf den Fehler Exception1
} catch (e: Exception2) {
    // Reaktion auf den Fehler Exception2
} catch (e: Exception) {
    // Reaktion auf alle anderen Fehler
}
```

Im Falle eines Fehlers wird der erste zutreffende `catch`-Block ausgeführt – und nur dieser! Bei der Formulierung der `catch`-Blöcke müssen Sie beachten, dass die Exception-Klassen hierarchisch strukturiert sind. Beispielsweise ist die `FileNotFoundException` von `IOException` abgeleitet, diese von der gewöhnlichen `Exception` und diese schließlich von `Throwable`. (Das folgende Listing gibt an, in welchen Paketen der Java-Standardbibliothek die Klassen definiert sind.)


```
kotlin.kotlin_builtins.Throwable
   java.lang.Exception
      java.io.IOException
         java.io.FileNotFoundException
```

Bei der Formulierung der `catch`-Blöcke müssen Sie mit spezifischen Fehlern beginnen und dürfen die allgemeine `Exception`-Klasse erst zum Schluss nennen. Das folgende Beispiel ruft die Methode `writeText` auf, um eine Textdatei zu (über-)schreiben. Mehr Informationen zum Umgang mit Dateien und ganz allgemein zum Thema Input/Output folgen in Kapitel 18, »Dateien verarbeiten (I/O, JSON, XML)«.

```
// Beispielprojekt try-catch
try {
    // fehleranfälliger Code zum Dateizugriff
    val f = File("/test.txt")         // Linux + macOS
    // val f = File("C:\\test.txt")   // Windows
    f.writeText("Zeile 1\nZeile 2\n")
} catch (e: FileNotFoundException) {
    println("Auf die Datei kann nicht zugegriffen werden.")
} catch (e: IOException) {
    println("Es ist ein anderer I/O-Fehler aufgetreten.")
} catch (e: Exception) {
    println("Es ist ein anderer Fehler aufgetreten,")
    println("der nichts mit I/O zu tun hat.")
}
```


Hätten Sie die obige `catch`-Kaskade mit `catch (e: Exception)` eingeleitet, wäre immer dessen Fehlerverarbeitungscode ausgeführt werden. Jede Fehlerklasse ist von `Exception` abgeleitet, deswegen trifft `catch (e: Exception)` auf jeden Fehler zu.

»finally«

Optional können Sie die `try-catch`-Konstruktion mit einem `finally`-Block abschließen. Der Code im `finally`-Block wird *immer* ausgeführt, ganz egal, ob vorher ein Fehler aufgetreten ist oder nicht. Damit eignet sich der `finally`-Block perfekt für Aufräumarbeiten, also z. B. für das Schließen geöffneter Dateien, Datenbank- oder Netzwerkverbindungen.

```
try {
    // fehleranfälliger Code
} catch (e: Exception) {
    // Reaktion auf Fehler
} finally {
    // Aufräumarbeiten, die auf jeden Fall ausgeführt
    // werden sollen
}
```

Wenn es einen finally-Block gibt, ist catch optional – d. h., auch try-finally ohne catch ist zulässig.

Vielleicht fragen Sie sich, was nun der Unterschied zwischen den beiden folgenden Code-Varianten ist:

```
try {
    blockA
} catch (e: Exception) {
    blockB
} finally {
    blockC
}
```

```
try {
    blockA
} catch (e: Exception) {
    blockB
}
blockC
```

blockC wird doch bei beiden Varianten ausgeführt, oder? Das stimmt meistens, aber nicht immer. Wenn Sie die gesamte Konstruktion innerhalb von blockA oder blockB durch break/continue (innerhalb einer Schleife) oder durch return (innerhalb einer Funktion/Methode) verlassen, dann kümmert sich Kotlin im ersten Fall darum, vorher noch blockC auszuführen. Im zweiten Fall wird blockC dagegen nicht mehr ausgeführt! Der folgende Code dient als Beweis:

```
// Beispielprojekt try-catch
fun testFinally() {
    try {
        println("try-Block")
        return  // verlässt die Funktion
    } finally {
        println("finally-Block")
    }
}
// Ausgabe beim Aufruf von testFinally():
// try-Block
// finally-Block (!)
```

»try-catch« als Ausdruck

Abweichend von der Java-Syntax kann try-catch in Kotlin auch aus Ausdruck mit einem Ergebnis formuliert werden:

```
val result = try { block1 } catch (e: Exception) { block2 }
```

Wenn alles gut geht, bestimmt die letzte Anweisung in block1 das Ergebnis. Tritt dagegen ein Fehler auf, enthält result den Rückgabewert des letzten Statements in block2.

Der Datentyp von result ergibt sich aus der Kombination aller möglichen Rückgabedatentypen. Üblich ist es, wie im folgenden Beispiel im Fall eines Fehlers null zurückgeben. Der resultierende Datentyp ist dann einfach *nullable*. Es ist aber auch

erlaubt, dass die Blöcke vollkommen unterschiedliche Datentypen zurückzugeben – dann lautet der Ergebnisdatentyp Any. Das folgende Beispiel zeigt nochmals die Berechnung einer Rechtecksfläche, diesmal in Form eines try-Ausdrucks. area hat den Datentyp Double?.

```
// Beispielprojekt try-catch
val area = try {   // Datentyp Double?
    val l = inputDouble("Länge:")
    val b = inputDouble("Breite:")
    l * b
} catch (e: Exception) {
    println("Es ist ein Fehler aufgetreten.")
    null
}
println("Flächeninhalt: $area")
```

Grundsätzlich sind try-catch-Ausdrücke auch mit finally erlaubt. Der Code im finally-Block wird ausgeführt, hat aber keinen Einfluss auf das zurückgegebene Resultat.

Ressourcen automatisch freigeben (»use«)

In Java besteht die Möglichkeit, Ressourcen (also Dateien, Datenbankverbindungen etc.) als Parameter von try zu öffnen. Java kümmert sich dann darum, die Ressourcen beim Abschluss der try-Konstruktion zu schließen, egal ob ein Fehler eintritt oder nicht:

```
// Java-Code
try(IOKlasse1 x = new IOKlasse1("dateiname1")) {
    // IO-Objekt verwenden
} catch(IOException e) {
    // Fehlerbehandlung
}
// x wird automatisch mit .close() geschlossen
```

In Kotlin gibt es keine vergleichbare try-Syntax. Stattdessen können Sie try-catch mit der Funktion use kombinieren. Diese Funktion wird auf ein Objekt angewendet, das die Schnittstelle Closable implementiert. use liefert das Ergebnis des als Lambda-Ausdruck angegebenen Blocks zurück und garantiert gleichzeitig, dass das Objekt mit close geschlossen wird – selbst dann, wenn innerhalb des Lambda-Ausdrucks ein Fehler auftritt. Beachten Sie aber, dass dieser Fehler nach close neuerlich ausgelöst wird. Deswegen müssen Sie use innerhalb einer try-Konstruktion ausführen. Typischerweise sieht der Codeaufbau dann so aus:

```
// Kotlin-Code
try {
    val x = IOKlasse1("dateiname")
    x.use { // Lambda-Ausdruck
        // IO-Objekt verwenden
        it.write(...)
        ...
    }
    // x wird automatisch mit .close() geschlossen
} catch (e: Exception) {
    // Fehlerbehandlung
}
```

Das folgende konkrete Beispiel erzeugt im lokalen Verzeichnis eine Textdatei und speichert darin mit einem `BufferedWriter` zwei Textzeilen. (Wie bereits erwähnt, folgen mehr Informationen zum Umgang mit Dateien in Kapitel 18, »Dateien verarbeiten (I/O, JSON, XML)«.)

```
// Beispielprojekt try-catch
val currentdir = System.getProperty("user.dir")
val tmpfile = Paths.get(currentdir, "tmp-file-kotlin.txt")
println(tmpfile)
try {
    val bw = Files.newBufferedWriter(tmpfile,
                                     StandardOpenOption.CREATE)
    bw.use {
        it.write("Zeile 1\n")
        it.write("Zeile 2\n")
    }
} catch (e: Exception) {
    println("Fehler: $e")
}
```

Fehleranfälligen Code mit »runCatching« verpacken

Anstatt eine `try-catch`-Konstruktion zu bilden, können Sie fehleranfälligen Code auch als Lambda-Ausdruck an die globale Funktion `runCatching` übergeben. `runCatching` liefert ein `Result<T>`-Objekt zurück, wobei `T` der Datentyp des Lambda-Ausdrucks ist. Das `Result`-Objekt enthält das Ergebnis des Lambda-Ausdrucks bzw. Informationen über den Fehler. Die folgenden Zeilen zeigen die prinzipielle Anwendung:

```
// Beispielprojekt try-catch
val result = runCatching {
    val tmp = (0..200).random()
```



```
    if (tmp > 100) {
        throw java.lang.Exception("Zu groß")
    }
    tmp
}
println(result.isSuccess)
println(result.isFailure)
if (result.isSuccess) {
    println("Ergebnis: ${result.getOrThrow()}")
} else {
    println("Fehler: ${result.exceptionOrNull()}")
}
```

Zur Auswertung stellt die `Result`-Klasse diverse Eigenschaften und Methoden zur Verfügung, von denen ich hier nur die wichtigsten nenne: `isSuccess` und `isFailure` verraten, ob der Lambda-Ausdruck ohne bzw. mit Fehler ausgeführt wurde. Beim Zugriff auf das Ergebnis helfen die Methoden `getOrThrow`, `getOrNull` sowie `getOrDefault`. Das `Exception`-Objekt können Sie mit `exceptionOrNull` auslesen.

15.2 Selbst Fehler auslösen (»throw«)

Mit `throw` können Sie selbst eine Exception auslösen (werfen). Von dieser Möglichkeit sollten Sie Gebrauch machen, wenn an eine von Ihnen entwickelten Methode oder Funktion unsinnige Daten übergeben werden, oder wenn Sie im Konstruktor einer Klasse erkennen, dass Sie mit den zur Verfügung stehenden Daten kein zweckmäßiges Objekt erzeugen können. Eine klare Fehlermeldung ist hier (speziell für andere Entwickler, die Ihre Klassen nutzen) hilfreicher als der Versuch, fehlerhafte Daten mit Defaultwerten oder Defaultzuständen zu kaschieren.

```
// Funktion für eine Integer-Division
fun divide(a: Int, b: Int): Int {
    if (b==0)
        throw IllegalArgumentException("b darf nicht 0 sein!")
    return a / b
}
```

Nach Möglichkeit sollten Sie versuchen, eine der vielen vordefinierten Exceptions aus der Kotlin- oder Java-Standardbibliothek zu verwenden. Die im obigen Beispiel genutzte `IllegalArgumentException` bietet sich an, wenn Sie einen Fehler aufgrund eines fehlerhaften Parameters auslösen möchten.

Bei Bedarf können Sie für Ihr Programm unkompliziert eine eigene Exception-Klasse definieren. Die folgende `DbException`-Klasse ist für Fehler in einer Datenbankbiblio-

thek gedacht. Zusätzlich zu den gewöhnlichen Fehlerdaten besteht hier die Möglichkeit, eine Zeichenkette mit den Verbindungsdaten zur Datenbank zu übergeben.

```
class DbException(msg: String) : Exception(msg) {
    val connectionData: String? = null
}
```

Der Datentyp »Nothing«

Funktionen, die auf jeden Fall eine Exception auslösen und somit nie regulär verlassen werden, können mit dem Datentyp `Nothing` deklariert werden:

```
fun neverReturns(s: String): Nothing {
    if (s.length<10)
        throw IllegalArgumentException("Zeichenkette zu kurz")
    else
        throw IllegalArgumentException("Zeichenkette zu lang")
}
```

`Nothing` ist laut Kotlin-Dokumentation ein Wert, der nie existiert – beinahe schon ein philosophisches Mysterium:

https://kotlinlang.org/api/latest/jvm/stdlib/kotlin/-nothing.html

Die Annotation »Throws«

Anders als in Java ist es in Kotlin nicht erforderlich, Methoden zu kennzeichnen, die selbst einen Fehler auslösen können. (Java sieht dazu das Schlüsselwort `throws` vor.) Wenn Sie allerdings in einem Projekt Code mehrerer Sprachen mischen (z. B. Java und Kotlin oder Swift und Kotlin) und in Kotlin eine Methode entwickeln, die aus einer Java/Swift-Klasse heraus aufgerufen werden soll, dann können Sie Ihrer Methode die Annotation `@Throws` voranstellen. Das folgende Listing gibt ein Beispiel für diese Syntax:

```
// Annotation, nur sinnvoll, wenn die Methode
// von Java aus aufgerufen werden soll
@Throws(DbException::class)
fun connectToDatabase(hostname: String) {
    ...
}
```

Die Annotation `@Throws` ist Teil der Kotlin-Standardbibliothek und ist direkt im `kotlin`-Package definiert (also `kotlin.Throws`).

15.3 Arbeitstechniken

Losgelöst von der Kotlin-Syntax möchte ich Ihnen hier noch einige grundsätzliche Arbeitstechniken bzw. Überlegungen mit auf den Weg geben.

Am besten sind Fehler, die gar nicht auftreten

`try-catch` ist kein Zaubermittel gegen schlampige Programmierung. Prinzipiell können in einem Programm zwei Arten von Fehlern auftreten:

- Zur ersten Gruppe zählen vermeidbare Fehler, also der Zugriff auf ein nicht vorhandenes Element einer Liste oder die Division durch 0. Zwar kann `try-catch` auch solche Fehler abfangen; aber derartige Fehler sind immer ein Zeichen dafür, dass Sie schlampig gearbeitet haben. Nehmen Sie sich die Zeit, Ihren Code in Ruhe und für alle Sonderfälle zu testen!
- Die zweite Gruppe sind Fehler, die außerhalb Ihrer Verantwortung liegen: Ein Download scheitert, weil der externe Server nicht antwortet; die Datenbankverbindung funktioniert nicht mehr, weil jemand ein Netzwerkkabel entfernt hat; die Datei kann nicht gespeichert werden, weil der Benutzer die SD-Karte oder den USB-Stick gelöst hat, usw. Dank `try-catch` kann Ihr Programm oder Ihre App in solchen Situationen angemessen reagieren und vielleicht sogar einen Hinweis auf die Fehlerursache geben.

Alles oder nichts

Grundsätzlich würde es ausreichen, mit `try-catch` exakt den Aufruf der Methode abzusichern, die einen Fehler verursachen kann. In der Praxis ist es aber üblicher, den Begleitcode mit in den `try`-Block zu integrieren. Das macht den Code lesbarer. Außerdem sind nun auch Anweisungen abgesichert, bei denen Sie keine Probleme erwarten, die aber unter Umständen dennoch Probleme verursachen können.

Der Nachteil einer allzu großzügigen Absicherung besteht darin, dass es damit zunehmend schwierig wird, die tatsächliche Fehlerursache zu erkennen und spezifisch darauf zu reagieren.

Orientierung an der Zielgruppe

Wenn Sie an einer App arbeiten, sind Endanwender Ihre Zielgruppe. Eine sorgfältige Absicherung Ihres Codes verhindert, dass Ihre App bei Bedienungsfehlern, Netzwerkproblemen etc. sang- und klanglos abstürzt und dabei womöglich auch noch Daten verloren gehen.

Ganz anders ist die Lage, wenn Sie an Klassen oder Bibliotheken arbeiten, die für andere Programmierer gedacht sind: Hier ist es im Gegenteil oft sinnvoll, bei der

Übergabe ungültiger Parameter oder bei einer nicht zweckmäßigen Anwendung von Methoden explizit einen Fehler auszulösen. Exceptions sind hier ein Kommunikationsmittel, um anderen Entwicklern klarzumachen, dass sie Ihre Klassen fehlerhaft anwenden.

Kapitel 17
Asynchrone Programmierung

Von *asynchroner* oder *nebenläufiger Programmierung* spricht man, wenn Teile des Programms in unterschiedlichen Teilprozessen parallel zueinander laufen. Gewöhnlicher Code wird immer synchron ausgeführt. Das bedeutet, dass die Anweisungen nacheinander ausgeführt werden:

```
func1()  // func2() wird erst gestartet,
func2()  // wenn func1() fertig ist
```

Es gibt mehrere Gründe, Code asynchron zu formulieren:

- Eine Teilaufgabe soll nicht das restliche Programm blockieren. Während eine App Daten herunterlädt, soll die Benutzeroberfläche weiter bedienbar bleiben (und idealerweise eine Möglichkeit bieten, den Download abzubrechen).
- Die vielen Cores einer CPU sollen optimal genutzt werden. Insbesondere sollen Server-Anwendungen möglichst effizient auf viele, quasi gleichzeitig eintreffende Anfragen reagieren können.
- Rechenintensive Aufgaben können unter Umständen durch die parallele Nutzung mehrerer Cores beschleunigt werden. »Gewöhnliche« (also synchrone) Algorithmen lasten dagegen nur einen Core aus und nutzen daher die Hardware nicht optimal. Allerdings ist die Verteilung von Aufgaben über mehrere Prozesse oft schwierig und mitunter gar nicht möglich.

Auf Sprachebene gibt es in Kotlin lediglich das Schlüsselwort `suspend` (wörtlich »aussetzen«) zur Kennzeichnung von Funktionen, die asynchron ausgeführt werden sollen. Alle weiteren Hilfsmittel, die zur Ausführung asynchronen Codes erforderlich sind, befinden sich in der Zusatzbibliothek `kotlinx.coroutines`.

Dieses Kapitel gibt eine Einführung in den Umgang mit Koroutinen, ohne aber auf alle Sonderfälle, Varianten und asynchrone Kommunikationsmechanismen einzugehen, die in der umfassenden Dokumentation behandelt werden:

https://kotlinlang.org/docs/reference/coroutines/coroutines-guide.html
https://github.com/Kotlin/kotlinx.coroutines/tree/master/docs
https://github.com/Kotlin/kotlinx.coroutines/blob/master/ui/coroutines-guide-ui.md

17.1 Hello Coroutines!

Damit Sie die Coroutines-Bibliothek möglichst einfach in einem IntelliJ-Projekt ausprobieren können, verwenden Sie beim Einrichten des neuen Projekts die Variante Gradle Groovy und wählen im Listenfeld Project Template die Option JVM • Console Application. (Gradle ist ein in der Java- und Kotlin-Welt stark verbreitetes Build-Tool. Es hilft dabei, Projekte zu kompilieren, die auf mehrere externe Bibliotheken zugreifen – siehe Anhang A.2.)

Nun öffnen Sie per Doppelklick die Datei `build.gradle` und bauen dort den Block `dependencies` mit der folgenden `implementation`-Zeile ein:

```
// Datei build.gradle
...
dependencies {
    // vorhandene Einträge belassen, die folgende
    // Anweisung in EINER Zeile hinzufügen:
    implementation
      'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.9'
}
```

`implementation` und die nachfolgende Zeichenkette müssen in derselben Zeile angegeben werden und wurden im obigen Listing nur aus Platzgründen auf zwei Zeilen verteilt. Bis dieses Buch erschienen ist, gibt es sicher schon eine neuere Version der Coroutines-Bibliothek. Die aktuelle Versionsnummer finden Sie auf der GitHub-Seite der Bibliothek:

https://github.com/Kotlin/kotlinx.coroutines

IntelliJ erkennt die Änderung an `build.gradle` und blendet den Button Load Gradle Changes ein. Sie müssen diesen Button anklicken, damit Ihre Ergänzungen in `build.gradle` wirksam werden (siehe Abbildung 17.1).

Schließlich fügen Sie in die schon vorhandene Datei `src/main/kotlin/main.kt` den folgenden Beispielcode ein und führen ihn aus:

```
import kotlinx.coroutines.*

fun main() {
    // launch startet eine Koroutine im Hintergrund
    GlobalScope.launch {
        delay(1000L) // in ms
        println("Coroutines!")
    }
    // der Code hier wird sofort ausgeführt
    print("Hello ")
```



```
    Thread.sleep(2000L)  // sonst endet das Programm, bevor
                         // der mit launch gestartete Code
                         // fertig ist
}
// Ausgabe: Hello Coroutines!
```

Das Programm gibt sofort »Hello« und nach einer Sekunde Verzögerung »Coroutines!« aus. Die Verzögerung ergibt sich aus dem `delay`-Aufruf, wobei die Zeitangaben in Millisekunden erfolgen.

Vorzeitiges Programmende

Beim Test von asynchronem Code müssen Sie darauf achten, dass Ihr Programm nicht vorzeitig endet. Damit enden auch alle noch nicht fertigen Koroutinen! Im obigen Listing wird das durch `Thread.sleep` verhindert.

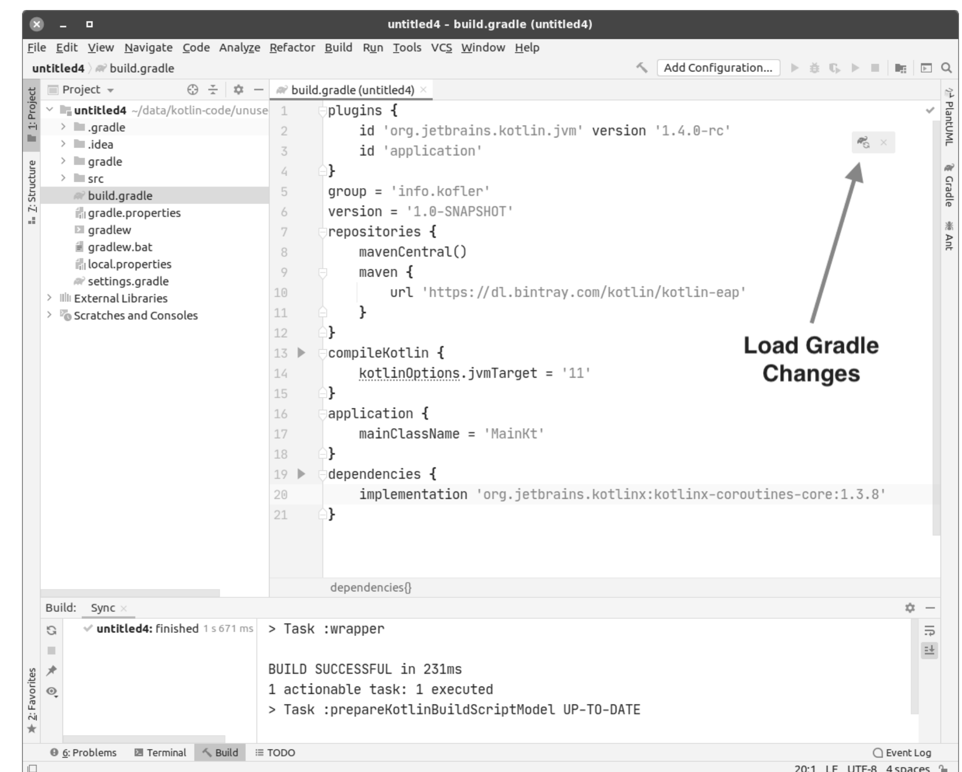

Abbildung 17.1 Die Bibliothek »coroutines« muss in den Abschnitt »dependencies« eingebaut werden.

Nomenklatur: Von Threads, Scopes und Dispatchern

Das Listing des vorigen Abschnitts war kurz und in seiner Funktion leicht nachzuvollziehen, gleichzeitig kamen aber drei verschiedene Begriffe/Konzepte vor: `launch` startet eine Koroutine. `GlobalScope` gibt an, in welchem Scope (Bereich) diese Koroutine ausgeführt werden soll. Und `Thread` bezieht sich auf den Thread, in dem die `main`-Funktion des Programms läuft.

Bevor Sie in die asynchrone Programmierung einsteigen können, müssen Sie sich mit deren Nomenklatur anfreunden. Im Folgenden gehe ich davon aus, dass Sie Ihren Kotlin-Code mit der JVM ausführen. Kotlin setzt in diesem Fall auf das Multithreading-Fundament auf, das durch Java bzw. durch die JVM vorgegeben ist. Für Kotlin JS bzw. Kotlin Native gelten zum Teil abweichende Regeln.

- **Threads** der JVM werden durch die Multi-Threading-Funktionen des Betriebssystems verarbeitet. Jeder Thread hat einen eigenen Stack-Speicher für Verwaltungsinformationen. Auf dort befindliche Daten können andere Threads nicht zugreifen. Unabhängig davon teilen sich alle Threads den gemeinsamen Heap-Speicher, in dem Kotlin- bzw. Java-Objekte, Listen, Arrays usw. abgelegt werden.

  `Thread.sleep(n)` pausiert den gerade aktuellen Thread um `n` Millisekunden. Die Klasse `Thread` und die statische Methode `sleep` sind in der Java-Standardbibliothek definiert und haben nichts mit den Koroutinen von Kotlin zu tun. (Innerhalb von Koroutinen müssen Sie anstelle von `sleep` die Methode `delay` aufrufen.)

- **Koroutinen** werden hingegen nicht vom Betriebssystem verwaltet, sondern direkt von Kotlin. Wenn die Codeausführung von einer Koroutine zur nächsten wechselt, ist dabei kein CPU-Kontext-Switch erforderlich (wie dies bei Threads der Fall ist). Koroutinen besitzen zudem keinen eigenen Stack. Zusammengefasst bedeutet das, dass Koroutinen mit Ressourcen wesentlich sparsamer umgehen als Threads. In Kotlin können Sie Tausende von Koroutinen starten, ohne Angst haben zu müssen, dass der damit verbundene Overhead Ihr System zum Stillstand bringt.

  Hinter den Kulissen greifen natürlich auch Koroutinen auf Threads zurück. Allerdings verwendet Kotlin dabei einen vergleichsweise kleinen Pool von Threads, den es nach Bedarf erzeugt. Innerhalb eines Threads können viele Koroutinen ausgeführt werden. Die Anzahl der aktiven Threads zum gegenwärtigen Zeitpunkt finden Sie bei Bedarf mit `Thread.activeCount()` heraus.

- Jede Koroutine muss in einem Kontext ausgeführt werden, dem sogenannten **Scope** (wörtlich »Anwendungsbereich«). Kotlin-intern ist die Schnittstelle `CoroutineScope` für die Verwaltung dieser Kontextdaten zuständig.

  Im Einführungsbeispiel stellte das Objekt `GlobalScope` diesen Kontext zur Verfügung. Mit `launch` wurde eine Art »Top-Level«-Koroutine gestartet.

  Das Problem des `GlobalScope`-Kontexts besteht darin, dass in ihm ausgeführte Koroutinen unter Umständen erst mit dem Ende des gesamten Programms gestoppt werden – z. B. wenn ein im Hintergrund gestarteter Download »hängt«. Die Verwendung des `GlobalScope` ist deswegen (außer in winzigen Beispielen wie in diesem Kapitel) selten empfehlenswert.

  Besser ist es, einen eigenen `CoroutineScope` für ein inhaltlich zusammenhängendes Objekt zu verwenden (also z. B. einen Scope für alle Koroutinen, die für eine Android-Aktivität benötigt werden). Endet die Lebensdauer der Objekts, können alle im Kontext dieses Objekts gestarteten und vielleicht noch immer aktiven Koroutinen beendet werden (siehe auch Abschnitt 17.7, »Asynchroner Code in Android-Apps«).

- Mit jedem Scope ist ein sogenannter **Dispatcher** verbunden. Diese Funktion entscheidet, von welchem Thread bzw. von welchen Threads die Koroutinen tatsächlich ausgeführt werden. Kotlin kennt einige vordefinierte Dispatcher:
  - `Dispatchers.Main` ist für Code vorgesehen, der im Main-Thread ausgeführt werden soll.
  - `Dispatchers.IO` ist für Code gedacht, der im Hintergrund auf Dateien oder Netzwerk-Ressourcen zugreift.
  - `Dispatchers.Default` kommt zur Anwendung, wenn Sie CPU-intensive Aufgaben im Hintergrund erledigen möchten. Dieser Dispatcher wird standardmäßig verwendet, wenn Sie nicht explizit einen anderen Dispatcher auswählen. Daher sind die beiden folgenden Ausdrücke gleichwertig:

    ```
    GlobalScope.launch { ... }
    GlobalScope.launch(Dispatchers.Default) { ...}
    ```
  - `Dispatchers.Unconfined` startet Koroutinen im gerade aktuellen Thread. Wenn die Koroutine angehalten und später fortgesetzt wird, gibt es keine fixe Thread-Zuordnung mehr. Der Hintergrundprozess kann in einem beliebigen Thread fortgesetzt werden. Der Einsatz dieses Dispatchers in eigenen Projekten ist nicht empfohlen.

Die statischen Methoden der Java-Klasse `Thread` verraten den Namen des gerade aktiven Threads sowie die Anzahl der Threads:

```
GlobalScope.launch {
    println("Thread-Anzahl: ${Thread.activeCount()}")
    println("Thread-Name:   ${Thread.currentThread().name}")
    // Ausgabe:
    // Thread-Anzahl: 4
    // Thread-Name:   DefaultDispatcher-worker-1
}
```

`Dispatchers.IO` und `Dispatchers.Default` teilen Threads im gleichen Pool miteinander. Lassen Sie sich also nicht davon irritieren, dass `Thread.currentThread().name` in beiden Fällen eine Bezeichnung der Art `DefaultDispatcher-worker-<nnn>` liefert.

Mehr Details

Eine viel detailliertere Beschreibung, wie Koroutinen durch Kotlin ausgeführt werden und wie Sie dabei auf Scopes und Dispatcher Einfluss nehmen können, finden Sie in der Dokumentation zu `kotlinx.coroutines`:

https://github.com/Kotlin/kotlinx.coroutines/blob/master/docs/coroutine-context-and-dispatchers.md

17.2 Koroutinen ausführen

Kotlin bietet diverse Möglichkeiten zum Start neuer Koroutinen bzw. Jobs. (Die beiden Begriffe werden im Weiteren synonym verwendet.) In diesem Abschnitt stelle ich Ihnen die wichtigsten Verfahren vor, um asynchrone Jobs zu starten.

Jobs ausführen (»launch«)

Die schon bekannte Methode `launch` führt den üblicherweise als Lambda-Ausdruck formulierten Code im Hintergrund aus. `launch` wird auf ein `CoroutineScope`-Objekt angewendet. In den folgenden Beispielen kommt der Einfachheit halber `GlobalScope` zum Einsatz:

```
GlobalScope.launch {
    // Code wird im Hintergrund durch Threads
    // des Default-Dispatchers ausgeführt.
}
```

Wenn Ihr Code nicht in den Threads des Default-Dispatchers laufen soll, geben Sie den gewünschten Dispatcher explizit als Parameter von `launch` an. Dabei sind oft auch verschachtelte Konstruktionen sinnvoll. (Das innere `launch`-Kommando bezieht sich dabei automatisch auf den dank `GlobalScope.launch` schon vorhandenen Scope.)

```
GlobalScope.launch(Dispatchers.IO) {
    // IO- oder Netzwerk-Aufgaben im Hintergrund
    // ausführen
    launch(Dispatchers.Main) {
        // nach der Fertigstellung Ergebnis
        // im Main-Thread verarbeiten
    }
}
```


Achten Sie auf den Kontext!

Wenn Sie `launch`-Methoden verschachteln, kommt für jeden `launch` der Kontext und damit auch der Dispatcher der nächsthöheren Ebene zur Anwendung. Wenn Sie das nicht wollen, müssen Sie entweder den gewünschten Scope voranstellen (`myscope.launch { ... }`) oder wie im obigen Beispiel den gewünschten Dispatcher in Klammern angeben.

In »echten« Anwendungen mit vielen Koroutinen sollten Sie eigene `CoroutineScope`-Objekte erzeugen und Ihre Koroutinen in deren Kontext ausführen. Im einfachsten Fall sieht der Code wie folgt aus:

```
val myscope = CoroutineScope(Dispatchers.IO)
myscope.launch {
    // Code wird im Hintergrund durch Threads
    // des IO-Dispatchers ausgeführt.
}
```

In Android-Apps ist es oft zweckmäßig, den `CoroutineScope` mit einer Aktivität oder einem Fragment zu verbinden (siehe Abschnitt 17.7, »Asynchroner Code in Android-Apps«).

Jobs abwarten (join)

Die Methode `join` gibt Ihnen die Möglichkeit, den Abschluss einer Koroutine abzuwarten. `join` führt also die Ausführung einer asynchronen Koroutine (eines Hintergrund-Jobs) mit dem gerade laufenden Code zusammen.

Damit Sie `join` anwenden können, benötigen Sie eine Referenz auf den Job. Dazu müssen Sie das Ergebnis der `launch`-Methode speichern. Es handelt sich dabei um ein `Job`-Objekt.

Die Ausführung von `join` ist allerdings nur im Kontext eines `CoroutineScope`s erlaubt. Einen derartigen Kontext erhalten Sie am einfachsten mit `runBlocking`. (Diese Funktion stelle ich einige Seiten weiter noch ausführlich vor. Sie dient ganz allgemein dazu, asynchrone und synchrone Codeteile zu kombinieren.)

Das folgende Listing zeigt die Funktionsweise von `join`:

```
// Job im Hintergrund starten
val job = GlobalScope.launch {
    for(i in (1..10)) {
        println("Fortschritt: $i")
        delay((500..2000L).random())
    }
}
```

```
Thread.sleep(3000)  // andere Arbeiten erledigen (in diesem
                    // Fall: nur schlafen ...)
runBlocking {
    job.join()      // warten, bis der Hintergrund-Job fertig ist
}
println("Ende")
// Ausgabe:
// Fortschritt: 1
// ...
// Fortschritt: 10
// Ende
```

Eine Variante zu `join` ist `joinAll`: Die Methode wird auf ein `CoroutineScope`-Objekt angewendet und wartet das Ende aller Jobs im aktuellen Kontext ab.

Funktionen mit Ergebnis ausführen (»async«)

`launch` führt im Hintergrund eine Aufgabe durch, gibt aber kein Ergebnis zurück. Genau diese Aufgabe erfüllt die Methode `async`. Die Methode liefert als Ergebnis ein Objekt des Typs `Deferred<T>`, wobei `T` der Ergebnistyp des Lambda-Ausdrucks ist. Im folgenden Beispiel soll die Anzahl der unterschiedlichen Zeichen in einem langen Text ermittelt werden. `result` hat den Typ `Deferred<Int>`.

```
val longtxt = "lorem ipsum"
val result = GlobalScope.async(Dispatchers.IO) {
    longtxt.toSet().count()
}
// Programm wird hier sofort fortgesetzt
```

Wegen des asynchronen Aufrufs wird das Programm ohne Verzögerung fortgesetzt. Allerdings enthält `result` noch nicht das endgültige Ergebnis, sondern ein Objekt, das über den Zustand der Koroutine Auskunft gibt. Die Eigenschaft `isActive` verrät, ob die Koroutine noch ausgeführt wird. Die Methode `await` wartet, bis die Koroutine fertig ist, und gibt dann das endgültige Ergebnis zurück (allgemein im Typ `T`, beim obigen Beispiel als `Int`-Zahl). Allerdings darf `await` nur in einer *Suspending Function* (siehe Abschnitt 17.6, »Suspending Functions«) oder in einer Koroutine ausgeführt werden, nicht aber in »gewöhnlichem« Code. In der Praxis bietet sich zur Auswertung von `async`-Ergebnissen der Einsatz einer `runBlocking`-Konstruktion an (siehe einige Seiten weiter).

```
runBlocking {
    // wartet, bis das Ergebnis vorliegt, und gibt es aus
    println(result.await())
}
```


»getCompleted«

Im `kotlinx.coroutines`-Paket gibt es die experimentelle Funktion `getCompleted`, die auf ein `Deferred`-Objekt angewendet werden kann. Wenn die zugrunde liegende Koroutine bereits abgeschlossen ist, liefert `getCompleted` wie `await` das Ergebnis. Läuft die Koroutine dagegen noch (bzw. wurde diese vorzeitig abgebrochen), dann löst `getCompleted` eine Exception aus.

Wenn Sie mehrere `Deferred`-Ergebnisse in einer Aufzählung (Liste, Set etc.) speichern, liefert `awaitAll` eine entsprechende Liste mit den Ergebnissen. Die Funktionsweise ist ähnlich wie `deferredlist.map { it.await()}`. Der Unterschied besteht darin, dass `awaitAll` sofort abbricht, wenn bei irgendeinem Teilergebnis ein Fehler auftritt, während `map` die Liste sequenziell verarbeitet und seine Arbeit beim ersten fehlerhaften Element abbricht. Ein Anwendungsbeispiel für `async` und `awaitAll` finden Sie in Abschnitt 17.8, »Beispiel: Effizient numerisch integrieren«, wo ein Rechenalgorithmus parallelisiert wird.

Parallele Verarbeitung von Collections

Die Kotlin-Standardbibliothek sieht grundsätzlich keine parallele (*multi-threaded*) Verarbeitung von Listen und anderen Aufzählungen vor. Natürlich können Sie selbst entsprechende Algorithmen programmieren. Sinn macht dies aber in der Regel nur dann, wenn entweder sehr viele Elemente vorliegen (Millionen) oder wenn der Rechenaufwand für die Bearbeitung einzelner Elemente sehr groß ist. In diesem Fall besteht z. B. die Möglichkeit, die Bearbeitungsfunktion innerhalb von `map` mit `async` zu starten und die resultierende Aufzählung von `Deferred`-Elementen dann mit `waitAll` wieder zusammenzuführen:

```
val lst = List<Int>(100) { (0..100000).random() }
var result = listOf<Int>()
runBlocking {
    result = lst.map {
        GlobalScope.async(Dispatchers.Default) {
            myLongRunningFunc(it)
        }
    }.awaitAll()
}
println(result)
```

Eine alternative Vorgehensweise liefern parallele Streams, ein Konstrukt aus der Java-Klassenbibliothek. Der obige Code könnte damit wie folgt verkürzt werden:

```
result = lst.parallelStream()
    .map { myLongRunningFunc(it) }
    .toList())
```

Sie haben dabei aber keinerlei Einfluss, wie die Verarbeitung intern tatsächlich erfolgt. Hintergrundinformationen finden Sie wie üblich im Internet:

https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html
https://stackoverflow.com/questions/20375176

»runBlocking«

Während `launch` Code im Hintergrund startet und das Programm unmittelbar fortsetzt (*non-blocking*), können Sie mit der Funktion `runBlocking` Koroutinen ausführen und während dieser Zeit die weitere Ausführung des gerade aktiven Threads blockieren. Das widerspricht eigentlich der Idee der asynchronen Programmierung.

`runBlocking` ist aber oft ein nützliches Hilfsmittel, um *non-blocking* und *blocking* Code zu verbinden. Wie die Beispiele aus dem vorangegangenen »async«-Abschnitt gezeigt haben, erfordern auch Methoden wie `await` oder `join` den Einsatz von `runBlocking`.

Anders als bei `launch` und `async` stellen Sie `runBlocking` keinen Scope voran. `runBlocking` verwendet einen leeren Scope (das Objekt `EmptyCoroutineScope`) und stellt diesen dem Lambda-Ausdruck zur Verfügung. Deswegen können im Lambda-Ausdruck Methoden wie `launch`, `async` oder `delay` ohne Weiteres verwendet werden. Vergessen Sie aber nicht, den gewünschten Dispatcher anzugeben – sonst wird der gleiche Dispatcher verwendet, in dem auch `runBlocking` ausgeführt wird. Häufig ist das `Dispatchers.Main`, was eine echte Parallelisierung unmöglich macht.

Eine typische Anwendung von `runBlocking` besteht darin, dass Sie in einer Funktion mehrere Koroutinen im Hintergrund starten, aber den Abschluss aller dieser Hintergrundaktivitäten abwarten möchten:

```
runBlocking {
    launch(Dispatchers.IO)      { loadFile() }
    launch(Dispatchers.Default) { doSomething() }
    launch(Dispatchers.Default) { doSomethingElse() }
    // loadFile und doXxx  werden parallel im Hintergrund
    // ausgeführt
}
// hier geht es erst weiter, wenn loadFile und doXxx fertig sind
```

Der folgende Beispielcode illustiert den Code-Ablauf. Beachten Sie, dass die Ausgabe »Im Block, Position 2« erscheint, noch bevor die Koroutine 2 gestartet wurde. »Im Block, Position 3« zeigt, dass der serielle Code im Block fertig ist. Der Block kann aber erst abgeschlossen werden, wenn auch die im Hintergrund laufenden Koroutinen ihre Arbeit erledigt haben.


```
runBlocking {
    println("Im Block, Position 1")
    launch(Dispatchers.IO) {
        println("Ko1 startet, ${Thread.currentThread().name}")
        delay(2000L)
        println("Ko1 fertig")
    }
    launch(Dispatchers.Default) {
        println("Ko2 startet, ${Thread.currentThread().name}")
        delay(1000L)
        println("Ko2 fertig")
    }
    println("Im Block, Position 2")
    delay(500L)
    println("Im Block, Position 3")
}
println("Block fertig")
// Ausgaben:
// Im Block, Position 1
// Ko1 startet, DefaultDispatcher-worker-1
// Im Block, Position 2
// Ko2 startet, DefaultDispatcher-worker-3
// Im Block, Position 3
// Ko2 fertig
// Ko1 fertig
// Block fertig
```


Alternativ können Sie an `runBlocking` auch den Kontext eines eigenen `CoroutineScope`-Objekts übergeben. `launch` ohne Parameter verwendet dann automatisch den Dispatcher des eigenen Scopes:

```
// neuen Scope mit Default-Dispatcher erzeugen
val myscope = CoroutineScope(Dispatchers.Default)
// diesen Scope in runBlocking verwenden
runBlocking(myscope.coroutineContext) {
    launch(Dispatchers.IO)  { loadFile() }
    launch                  { doSomething() }
    launch                  { doSomethingElse() }
    // loadFile und doXxx  werden parallel im Hintergrund
    // ausgeführt
}
// hier geht es erst weiter, wenn loadFile und doXxx fertig sind
```

Beachten Sie, dass mit einer Schleife oder mit `repeat` ausgeführter Code innerhalb einer `runBlocking`-Konstruktion sequenziell ausgeführt wird:

```
// führt 10 print-Ausgaben durch
runBlocking {
    repeat(10) {
        // sequenziell verarbeiten
        delay((0..500L).random())
        println("Koroutine $it")
    }
}
// Fortsetzung, wenn repeat fertig ist
// Ausgabe: Koroutine 0, 1, 2 ... 9
```

Wenn die durch `repeat` gestarteten Koroutinen parallel ausgeführt werden sollen, müssen Sie zusätzlich `launch` einbauen!

```
runBlocking {
    repeat(10) {
        // in parallelen Koroutinen ausführen
        launch(Dispatchers.Default) {
            delay((0..500L).random())
            println("Koroutine $it")
        }
    }
}
// Fortsetzung, wenn repeat fertig ist
// Ausgabe z. B. Koroutine 4, 2, 0, 3, 5, 8, 6, ...
```

Zeit abwarten (»delay«)

Die aus den vorangegangenen Beispielen schon bekannte Methode `delay` hält die gerade aktuelle Koroutine für die angegebene Anzahl von Millisekunden an. Die Millisekunden müssen als `Long`-Wert übergeben werden.

Der für die Koroutine zuständige Thread wird durch `delay` nicht blockiert und kann in der Zwischenzeit andere Aufgaben übernehmen. Koroutinen können während der Wartezeit durch `cancel` beendet werden. `delay` verhält sich also gegenüber einer `cancel`-Aufforderung kooperativ, löst dabei aber eine Exception aus. Sie können die Exception entweder ignorieren oder für Aufräumarbeiten verwenden.

Anderen Prozessen Vorrang geben (»yield«)

`yield` unterbricht vorübergehend die Ausführung der aktuellen Koroutine und erlaubt dem Dispatcher, anderen Koroutinen Rechenzeit zuzuweisen. Die Methode kann wie `delay` nur in einer Koroutine (also in asynchronem Code) ausgeführt werden. `yield` überprüft wie `delay`, ob die aktuelle Koroutine noch aktiv ist. Wenn das nicht der Fall ist (`cancel`), wird die Koroutine mit einer Exception abgebrochen.

Scope anwenden, erzeugen oder variieren

Egal, ob Sie Koroutinen mit `async` oder `launch` ausführen – Sie müssen diese Methoden in jedem Fall auf einen `CoroutineScope` anwenden. Innerhalb einer Koroutine (also in asynchronem Code) ist bereits ein Scope vorhanden, außerhalb (also in synchronem Code) dagegen nicht. Es gibt verwirrend viele Möglichkeiten, neue Scopes zu erzeugen bzw. einen schon vorhandenen Scope abzuwandeln. Die folgende Aufzählung gibt einen Überblick:

- `GlobalScope` ist ein vordefiniertes Objekt (Singleton), um Jobs in einem globalen Kontext auszuführen. Standardmäßig kommt der Default-Dispatcher zum Einsatz.

  ```
  GlobalScope.launch { ... }
  ```

- Die Funktion `MainScope` erzeugt einen neuen `CoroutineScope` für Programme mit grafischen Benutzeroberflächen (siehe Abschnitt 17.7, »Asynchroner Code in Android-Apps«).

  ```
  MainScope().launch { ... }
  ```

- Der Konstruktor von `CoroutineScope` erzeugt einen neuen Scope für den angegebenen Dispatcher:

  ```
  val myscope = CoroutineScope(Dispatchers.IO)
  myscope.launch { ... }
  ```

- Die Funktion `coroutineScope` erzeugt ebenfalls einen neuen `CoroutineScope`. Anders als der Konstruktor kann die Funktion aber nur innerhalb einer Koroutine erzeugt werden, nicht in synchronen Codeabschnitten!

  ```
  GlobalScope.launch {
      // verwendet den globalen Scope
      ...
      coroutineScope {
          // führt diesen Code in einem neuen CoroutineScope
          // aus, der aber die Grundeigenschaften vom Scope der
          // höheren Ebene übernimmt
      }
  }
  ```

- Die Funktion `withContext` führt den nachfolgenden Code in einem anderen Kontext aus (z. B. mit einem anderen Dispatcher). Intern wird der Code gegebenenfalls in einem anderen Thread ausgeführt. `withContext` kann nur innerhalb einer Koroutine verwendet werden, d. h., es muss bereits einen `CoroutineScope` geben, aus dem dann ein neuer Scope erzeugt wird. Vorsicht: Die Angabe von `Dispatchers.Main` kann zu einem Fehler führen (*module with the Main dispatcher is missing*).

```
GlobalScope.launch {
    ...
    withContext(Dispatchers.IO) {
        // führt diesen Code in einem neuen CoroutineScope
        // mit anderem Dispatcher/Kontext aus
    }
}
```

- `runBlocking` ohne Parameter stellt für den nachfolgenden Lambda-Ausdruck einen leeren `CoroutineScope` zur Verfügung (genau genommen ein `EmptyCoroutineScope`-Objekt):

  ```
  runBlocking {
      launch { ...}
  }
  ```

  `runBlocking` ist für Testcode sowie zum Zusammenführen von synchronem und asynchronem Code gedacht. `runBlocking` soll dagegen *nicht innerhalb* von Koroutinen verwendet werden.

Je nachdem, woher Ihr Scope stammt, müssen Sie bei `launch` oder `async` explizit angeben, welcher Dispatcher verwendet werden soll. Für Hintergrundjobs ist `Dispatchers.IO` oder `Dispatchers.Default` die richtige Wahl. Wenn Sie Elemente einer (Android-)Benutzeroberfläche verändern wollen, müssen Sie `Dispatchers.Main` verwenden!

17.3 Koroutinen abbrechen

Jobs abbrechen (»cancel«)

Mit `cancel` können Sie eine mit `launch` gestartete Koroutine während ihrer Laufzeit dazu auffordern, ihre Tätigkeit einzustellen. Wie bei `join` setzt das voraus, dass Sie über eine Referenz auf den Job verfügen. Das folgende Listing zeigt, wie Sie einen Hintergrundprozess nach fünf Sekunden mit der `cancel`-Methode abbrechen:

```
// Beispielprojekt hello-coroutines, Funktion testCancel1()
val job = GlobalScope.launch {
    // die folgende Schleife im Hintergrund ausführen ...
    for(i in (1..10)) {
        val progress = "*".repeat(i)
        println("Fortschritt: $progress")
        delay((500..2000L).random())
    }
}
// ... und nach fünf Sekunden abbrechen
Thread.sleep(5000)
```



```
job.cancel()
println("Ende")
Thread.sleep(5000)
// Ausgabe:
// Fortschritt: *
// Fortschritt: **
// Fortschritt: ***
// Fortschritt: ****
// Ende
```

»cancel«-Kooperation

Bei der Anwendung von `cancel` müssen Sie eine Komplikation beachten: `cancel` kann das Ende einer Koroutine nicht erzwingen, sondern fordert den Job lediglich dazu auf, zu enden. Ob und wie schnell er darauf reagiert, ist dem Job überlassen.

Im obigen Beispiel hat `cancel` nur deswegen wunderbar funktioniert, weil der Hintergrundjob nur die Zeit mit `delay` totschlug. Die `delay`-Methode weist eine Besonderheit auf: Sie reagiert auf `cancel`-Aufforderungen kooperativ, beendet also das Warten *und* bricht die Koroutine ab.

Beim folgenden Beispiel, wo anstelle von `delay` eine Schleife massive CPU-Aktivität simuliert, funktioniert `cancel` dagegen nicht:


```
// Beispielprojekt hello-coroutines, Funktion testCancel2()
val job = GlobalScope.launch {
    // den folgenden Code im Hintergrund ausführen
    for(i in (1..10)) {
        val progress = "*".repeat(i)
        println("Fortschritt: $progress")
        var sum = 0L
        for (j in (0..500_000_000))
            sum += j
    }
}
Thread.sleep(2000)
println("Cancel")
job.cancel()        // evrsucht den Job abzubrechen.
println("Ende")     // Der Job läuft aber weiter, bis das
Thread.sleep(5000)  // Programm nach fünf Sekunden endet.
// Ausgabe:
// Fortschritt: *
// Fortschritt: **
// Fortschritt: ***
// Cancel
// Ende
```

```
// Fortschritt: ****
// Fortschritt: *****
// ...
```

`cancel` sendet also die Abbruchaufforderung. Das Programm wird im Vordergrund unmittelbar fortgesetzt (Ausgabe »Ende«). Im Hintergrund läuft aber auch der Job weiter, wahlweise bis zu seinem Abschluss oder bis das Programm endet. (Die Anweisung `Thread.sleep(5000)` ist hier entscheidend. Andernfalls würde das Testprogramm sowieso enden und die ausbleibende Reaktion auf die `cancel`-Aufforderung wäre nicht erkennbar.)

Wie können Sie also Ihre Koroutinen gestalten, damit diese in angemessener Zeit auf `cancel` reagieren? Innerhalb der Koroutine steht Ihnen die Eigenschaft `isActive` zur Verfügung. Sie müssen diese Eigenschaft hin und wieder überprüfen und den Code gegebenenfalls abbrechen. Vergessen Sie dabei nicht, dass Ihnen in Lambda-Ausdrücken leider `return` nicht zur Verfügung steht!

Das folgende Listing zeigt eine kooperative Variante der Koroutine. (Der restliche Code bleibt im Vergleich zum vorigen Beispiel unverändert.)

```
// Beispielprojekt hello-coroutines, Funktion testCancel3()
val job = GlobalScope.launch {
    for(i in (1..10)) {
        if (!isActive)  // wenn der Job abgebrochen werden soll,
            break       // i-Schleife verlassen
        val progress = "*".repeat(i)
        println("Fortschritt: $progress")
        var sum = 0L
        for (j in (0..500_000_000))
            sum += j
    }
}
```

Beachten Sie, dass auch im obigen Fall der Abbruch nicht unmittelbar erfolgt, sondern erst, wenn das nächste Mal die Bedingung `if (!isActive)` erreicht wird. Eine vielleicht gerade aktive `j`-Schleife wird vorher abgeschlossen.

Im obigen Beispiel wäre es ebenso denkbar, den `isActive`-Test in die `j`-Schleife einzubauen. Die vielen Tests würden aber die Rechenzeit unverhältnismäßig vergrößern.

»cancel«-Exceptions

Wenn die `cancel`-Aufforderung eintrifft, während `delay`, `yield` oder eine andere *cancelable* Funktion läuft, beendet diese Funktion nicht nur die Koroutine, sondern löst auch eine `JobCancellationException` aus. Es bleibt Ihnen überlassen, ob Sie darauf reagieren wollen oder nicht. Eine `try`-`finally`-Konstruktion gibt Ihnen in solchen Fällen die Möglichkeit, Aufräumarbeiten durchzuführen. Beachten Sie aber, dass *keine* Exception auftritt, wenn Sie selbst (»kooperativ«) auf `isActive` reagieren.


```
// Beispielprojekt hello-coroutines, Funktion testCancel4()
val job = GlobalScope.launch {
    // die folgende Schleife im Hintergrund ausführen ...
    try {
        for (i in (1..10)) {
            val progress = "*".repeat(i)
            println("Fortschritt: $progress")
            delay(500L)
        }
    } catch (e: Exception) {
        println("Exception $e")
    }
    finally {
        // Aufräumarbeiten durchführen
        println("Finally")
    }
}
```

In vielen Fällen können Sie den im obigen Code skizzierten `catch`-Block einfach weglassen (es sei denn, Sie erwarten auch andere Fehler). Die Exception-Absicherung hat hier nur die Aufgabe, innerhalb der Koroutine eventuell blockierte Ressourcen wieder freizugeben.

»cancel«-Varianten

Weil im Vorhinein nicht klar ist, wie lange es dauert, bis ein Job seine Arbeit einstellt, d. h., wie kooperativ er auf die `cancel`-Aufforderung reagiert, können Sie anstelle von `cancel` auch `cancelAndJoin` verwenden. Diese Methode leitet die `cancel`-Aufforderung weiter und wartet dann ab, bis die Koroutine tatsächlich beendet wird. (Um das nochmals klarzustellen: Die gewöhnliche `cancel`-Methode gibt lediglich die Abbruchaufforderung weiter. Anschließend wird der Code unmittelbar fortgesetzt, ganz egal, ob die Koroutine bereits beendet ist oder nicht.)

Die Methode `cancel` kann auch auf einen `CoroutineScope` angewendet werden und fordert in diesem Fall alle Koroutinen, die in diesem Scope gestartet wurden, zum Abbruch auf.

Wenn aus einer Koroutine heraus weitere Jobs gestartet wurden, also ein hierarchisches System von Koroutinen zusammengesetzt wurde, dann führt `cancel` auch zum Ende aller untergeordneten Jobs (*child jobs*). Alternativ können mit `cancelChildren` nur die untergeordneten Jobs beendet werden, während der Hauptjob (*parent job*) weiterläuft.

Asynchrone Funktionen abbrechen

`cancel` bzw. `cancelAndJoin` können auch auf ein `Deferred`-Objekt einer Funktion angewendet werden, die mit `async` gestartet wurde. Dabei gelten grundsätzlich die gleichen Regeln wie bei Koroutinen, die mit `launch` ausgeführt werden. Zu einem tatsächlichen Abbruch kommt es nur, wenn in der `async`-Koroutine *cancelable* Funktionen ausgeführt werden (z. B. `delay`, `yield`) oder wenn die Koroutine `isActive` beobachtet und ihre Arbeit selbst einstellt.

Ganz egal, ob die Koroutine dank `delay` automatisch beendet wird, ob Ihr Code die Koroutine aufgrund der Auswertung von `isActive` stoppt oder ob die Koroutine trotz `cancel`-Aufforderung bis zum Ende läuft – das resultierende `Deferred`-Objekt ist in jedem Fall unbrauchbar. Die Eigenschaft `isCancelled` liefert `true`; der Versuch, das Ergebnis mit `await` auszulesen, endet an dieser Stelle (also *nicht* in der Koroutine) mit einer Exception.

Koroutinen mit Timeouts

Mit der Funktion `withTimeout` können Sie einen Job starten, der nach einer vorgegebenen Zeit ein `cancel`-Signal erhält. Dabei sind allerdings einige Eigenheiten zu beachten:

- `withTimeout` kann nur im Kontext eines `CoroutineScope` genutzt werden, also z. B. in einer `runBlocking`-Konstruktion oder in einer anderen Koroutine.
- Der Lambda-Ausdruck wird im gleichen Thread ausgeführt wie die Funktion `withTimeout`. Anders als bei `launch` oder `async` erfolgt die Ausführung *nicht* im Hintergrund. Gegebenenfalls können Sie `launch` oder `async` innerhalb des `withTimeout`-Blocks aufrufen.
- `withTimeout` liefert weder ein Ergebnis noch ein `Job`-Objekt zurück.
- Wie bei `launch` und `async` kann `cancel` nicht erzwungen werden. Wenn im Lambda-Ausdruck weder eine *cancelable* Funktionen wie `delay` oder `yield` ausgeführt noch `isActive` ausgewertet wird, bleibt der Timeout wirkungslos und der Lambda-Ausdruck wird bis zum Ende ausgeführt.
- Wenn eine *cancelable* Funktion wie `delay` die Codeausführung abbricht, kommt es zu einer `TimeoutCancellationException`. Wenn Sie also `cancelable`-Funktionen verwenden, sollten Sie die ganze `withTimeout`-Konstruktion mit `try-catch` absichern.

Angesichts der vielen Einschränkungen gibt es nur wenige sinnvolle Anwendungen von `withTimeout`. Das folgende Listing demonstriert primär die Syntax:

```
runBlocking {
    try {
        withTimeout(1000L) {
```



```
            for (i in (1..10)) {
                println("Fortschritt: $i")
                delay(350L)
            }
        }
    } catch (e: Exception) {
        println("Exception $e")
    }
    println("weiter")
}
// Ausgabe:
// Fortschritt: 1
// Fortschritt: 2
// Fortschritt: 3
// Exception kotlinx.coroutines.TimeoutCancellationException:
//   Timed out waiting for 1000 ms
// weiter
```

Eine Variante zu `withTimeout` ist `withTimeoutOrNull`: Die Funktion liefert das Ergebnis des Lambda-Ausdrucks zurück oder `null`, wenn die Ausführung des Lambda-Ausdrucks durch eine *cancelable* Funktion abgebrochen wurde. Die Absicherung mit `try-catch` entfällt.

Kapitel 27

Hello Server!

Java wurde ursprünglich als universelle Sprache konzipiert, hat aber letztlich seine größte Verbreitung in drei recht speziellen Anwendungsgebieten gefunden:

- Zum Ersten wird Java an Schulen und Universitäten (leider) noch immer häufig als *First Programming Language* verwendet. Dabei gäbe es viele bessere Alternativen. (Persönlich finde ich Python ideal.)
- Zum Zweiten ist bzw. war Java *die* Programmiersprache für Android-Apps. Google empfiehlt zwar mittlerweile Kotlin anstelle von Java, aber natürlich wollen bzw. können nicht alle App-Entwickler von heute auf morgen ihren Code umstellen. Es ist abzusehen, dass Java und Kotlin jahrelang parallel im Einsatz sein werden.
- Und schließlich ist Java im sogenannten »Backend« omnipräsent. Dieser Begriff bezeichnet (oft aus vielen Rechnern zusammengesetzte) Server-Strukturen, die Geschäftsprozesse abbilden – z. B. das Einkaufssystem und die Lagerhaltung einer großen Lebensmittelkette oder das Abrechnungssystem eines Telekommunikationsanbieters. Java bzw. die Java Enterprise Edition (Java EE) wird dabei um diverse Bibliotheken und Frameworks für den Server-Einsatz erweitert. Besonders populär ist das *Spring Framework*, aber natürlich gibt es dazu diverse Varianten bzw. Ergänzungen.

Von Java zu Kotlin

Die letzten Kapitel dieses Buchs beschäftigen sich mit dem Thema, wie Kotlin im Backend eingesetzt werden kann. Aber bevor ich überhaupt richtig loslege, möchte ich zuerst darauf hinweisen, dass sich das Thema »Backend« nicht auf 100 Seiten abhandeln lässt. Angesichts der vielen Spielarten würde selbst ein ganzes Buch nicht ausreichen.

Grundsätzlich ist es ausgeschlossen, ein vorhandenes großes Backend-System einfach so von Java auf Kotlin umzustellen (ganz losgelöst von der Frage, warum man das tun sollte). Manche Unternehmen, die von den Vorteilen von Kotlin überzeugt sind, wählen deswegen einen pragmatischen Ansatz: Erweiterungen des Backends werden in Kotlin durchgeführt. Und bevor an einer vorhandenen Klasse umfangreiche Änderungen durchgeführt werden, wird versucht, auch diese Klasse in Kotlin-Code

umzuwandeln. Mit dieser Strategie verbleibt bewährter Code, der nicht verändert werden muss, in Java, während neuer bzw. geänderter Code Schritt für Schritt auf Kotlin umgestellt wird. Möglich ist das wegen der hohen Kompatibilität zwischen Java und Kotlin; diese erlaubt es, in einem Projekt nach Belieben Java- und Kotlin-Codedateien zu kombinieren.

In den folgenden Kapiteln gehe ich auf dieses Szenario nicht ein. Die Anforderungen sind zu speziell, um an dieser Stelle umfassend erläutert zu werden. Wie Sie Kotlin-Code in eine bestehende, über Jahre gewachsene Java-Basis integrieren, muss im Einzelfall geprüft werden. Wenn dann noch Frameworks wie Spring oder gar Eigenentwicklungen hinzukommen, erhöht sich die Komplexität noch einmal.

Stattdessen habe ich mich für einen anderen Ansatz entschieden: Ich konzentriere mich in den folgenden Kapiteln darauf, dass Sie Kotlin auf dem Backend für ein *neues Projekt* ausprobieren und dabei das von der Firma JetBrains entwickelte Microframework *Ktor* einsetzen wollen.

Ktor kann es im Funktionsumfang nicht mit Spring & Co. aufnehmen, weist dafür aber andere Vorteile auf: Ktor bietet volle Unterstützung für die moderne Syntax von Kotlin, es ist von Grund auf asynchron gedacht und es ist frei von Altlasten.

Microframeworks

Als *Microframeworks* werden eher minimalistische Bibliotheken für Webapplikationen bezeichnet, die sich unter anderem durch ein schlankes Design und kleinen Ressourcenbedarf auszeichnen. Ganz so winzig, wie der Begriff vermuten lässt, ist der Funktionsumfang von Ktor allerdings nicht. Ktor bietet weit mehr Features, als ich Ihnen in den folgenden Kapiteln im Detail vorstellen kann.

Deployment

Mit der Entwicklung einer Server-Anwendung ist es nicht getan: Sobald die Anwendung läuft, soll sie natürlich in den Produktivbetrieb überführt werden. Abermals muss ich passen: Prinzipiell ist es zwar nicht allzu schwierig, einen Linux-Server – oft in Kombination mit Docker – entsprechend einzurichten.

Das ist aber nur der erste Schritt: Jetzt stellen sich die Fragen, wie neue Versionen entwickelt und getestet werden (Beta-Betrieb, *Continous Integration*), wie Updates durchgeführt werden, wie das Zusammenspiel mit einem Versionsverwaltungs-Tool (in aller Regel *Git*) funktionieren soll usw. All diese Fragen führen weit über das eigentliche Thema dieses Buchs hinaus.

Voraussetzungen

In den folgenden Kapiteln setze ich natürlich voraus, dass Sie die Syntax von Kotlin bereits gut beherrschen. Darüber hinaus sollten Sie Kapitel 17, »Asynchrone Programmierung«, sowie Kapitel 19, »Datenbankzugriff (Exposed)«, gelesen und verstanden haben.

Außerdem wird es in den folgenden Kapiteln viel um das HTTP-Protokoll (GET- und POST-Requests), um REST-APIs, Authentifizierungstechniken usw. gehen. Wenn Sie schon einmal Webapplikationen erstellt haben (egal, mit welcher Programmiersprache), werden Ihnen die meisten Begriffe vertraut sein. Andernfalls werden Sie wohl hin und wieder einen Blick in die Wikpedia werfen oder im Internet nach Grundlagentexten oder Videos zu diesen Themen suchen müssen! Als erster Startpunkt bietet sich dieser Wikipedia-Artikel an:

https://de.wikipedia.org/wiki/Representational_State_Transfer

27.1 Hello Ktor!

Ktor ist ein in und für Kotlin programmiertes Framework für asynchrone Webserver und dazugehörende Client-Apps. Ktor wird wie die Programmiersprache Kotlin von der Firma JetBrains entwickelt. Das Open-Source-Framework verwendet eine freie Lizenz, die den uneingeschränkten Einsatz auch in kommerziellen Projekten erlaubt. Ich habe im Internet keine Informationen über die Herkunft des Namens *Ktor* gefunden. Die Projektwebsite mit ausgezeichneter Dokumentation und vielen Beispielen sowie die GitHub-Seite finden Sie unter:

https://ktor.io
https://github.com/ktorio/ktor

Projekt einrichten mit dem Ktor-Plugin

Das Einrichten eines neuen Ktor-Projekts und die Anfangskonfiguration der Datei `build.gradle` gelingen am einfachsten mit dem gleichnamigen Ktor-Plugin für IntelliJ. Dieses muss zuerst installiert werden: Dazu öffnen Sie mit File • Settings die IntelliJ-Einstellungen, suchen im Dialogblatt Plugins nach *Ktor* und installieren das Plugin. Beachten Sie, dass das Plugin nicht kompatibel zu Android Studio ist. Wie Sie in einer Android-App die Ktor-Client-Funktionen nutzen können, erkläre ich Ihnen in Abschnitt 30.1.

Nach einem Neustart von IntelliJ führen Sie File • New • Project aus. Im Assistenten gibt es nun den neuen Projekttyp Ktor mit einer schier überwältigenden Fülle von Optionen (siehe Abbildung 27.1):

- Project = Gradle legt das Build-System fest. Diese Einstellung entspricht Gradle Groovy im Assistenten zum Einrichten gewöhnlicher Kotlin-Projekte.

  Zur Auswahl stehen auch GradleKotlinDsl (entspricht Gradle Kotlin) und Maven, ich berücksichtige in diesem Buch aber nur Gradle.

- Wrapper: Diese vorselektierte Option bestimmt, ob Gradle-Builds über ein Wrapper-Script durchgeführt werden sollen. Belassen Sie die Option, wie sie ist. Hintergründe zur Sinnhaftigkeit des Gradle-Wrapper können Sie hier nachlesen:

  https://docs.gradle.org/current/userguide/gradle_wrapper.html

- Using = Netty: In diesem Listenfeld wählen Sie eine *Network Application Engine* aus, die Ihrem Projekt grundlegende Webserver-Funktionen zur Verfügung stellt und als unsichtbares Fundament Ihrer Server-App dient. Ktor unterstützt aktuell *Netty*, *Jetty*, *Tomcat* und *CIO*. (CIO steht für *Coroutine-based I/O* und ist eine native Kotlin-Implementierung der Engine, die aktuell nur HTTP/1.x unterstützt, nicht aber HTTP/2. Der Vorteil von CIO besteht darin, dass dem Projekt keine weiteren Abhängigkeiten hinzugefügt werden.)

  Bleiben Sie bei *Netty*, solange Sie keine spezifischen Features anderer Engines benötigen. Wenn Ihre App Servlets unterstützen soll, verwenden Sie stattdessen *Tomcat*. Zum Verständnis, wozu die Engine bzw. das Framework überhaupt dient, werfen Sie einen Blick auf die folgende Stack-Overflow-Seite, die Netty und Jetty vergleicht:

  https://stackoverflow.com/questions/5385407

- Server- und Client-Features: Mit vielen weiteren Optionen legen Sie fest, welche Features von Ktor Sie nutzen möchten. Deren Aktivierung hat zwei Auswirkungen: Zum einen fügt das Plugin die erforderlichen `implementation`-Zeilen in den `dependencies`-Abschnitt von `build.gradle` ein, zum anderen baut das Plugin in vielen Fällen ein paar Zeilen Beispielcode in die Datei `Application.kt` ein. Beides ist hilfreich, aber natürlich lässt sich das später auch manuell erledigen.

  Wenn Sie eine Option im Assistenten anklicken, wird unten eine Kurzbeschreibung samt Link zur Dokumentation eingeblendet. Das hilft bei der Entscheidungsfindung. Für erste Tests reichen HTML DSL und CSS DSL vollkommen aus.

In zwei weiteren Dialogen können Sie nun den Package-Namen und den Projektnamen festlegen.

Anleitungen, wie Sie eigene Ktor-Projekte ohne IntelliJ und dessen Ktor-Plugin einrichten, finden Sie in der Ktor-Dokumentation:

https://start.ktor.io
https://ktor.io/docs/maven.html

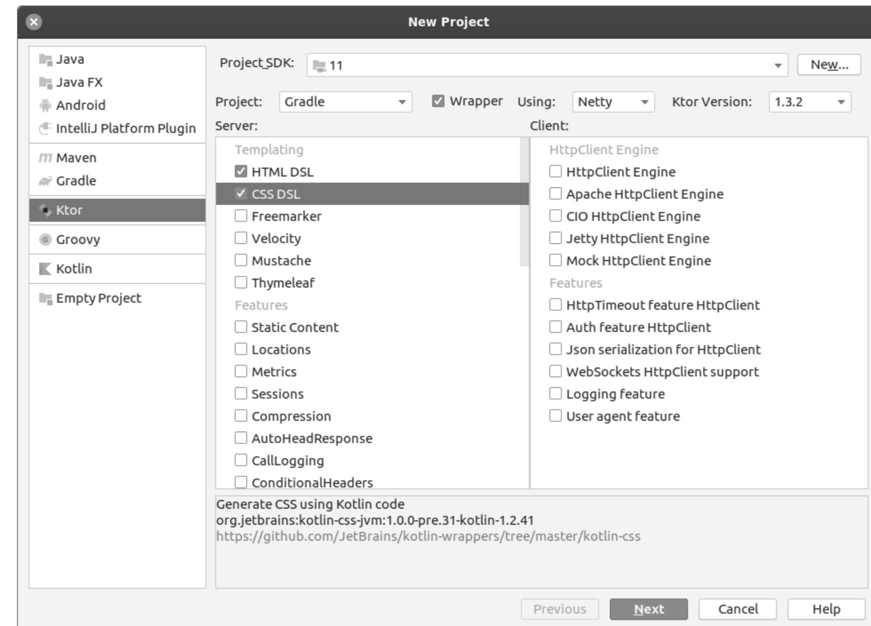

Abbildung 27.1 Unzählige Optionen beeinflussen das neue Projekt.

Projekt ausführen

Um den vom Plugin erzeugten Mustercode erstmalig auszuführen, öffnen Sie die Datei `Application.kt` und klicken auf den grünen Run-Button bei der `main`-Funktion. Damit fügen Sie dem Projekt die Run-Konfiguration hinzu.

IntelliJ zeigt nun im Run-Fenster an, welche IP-Adresse und welchen Port Ihre Web-App nutzt. Standardmäßig ist das *http://0.0.0.0:8080*. Wenn Sie diese Adresse in einem Webbrowser öffnen, reagiert die Server-App, indem sie den Text `HELLO WORLD!` sendet (siehe Abbildung 27.2).

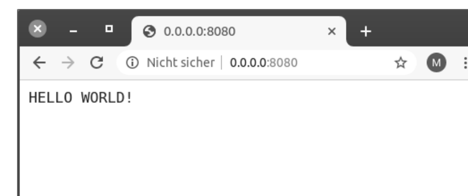

Abbildung 27.2 »Hello World!« im Webbrowser

Falls der Port 8080 auf Ihrem Rechner bereits von einer anderen Anwendung blockiert ist, endet das Programm mit einer Fehlermeldung. In diesem Fall öffnen Sie die Datei `resources/application.conf` und stellen einen anderen Port ein.

Mustercode

Die folgenden Zeilen zeigen den vom Ktor-Plugin erzeugten Mustercode:

```
// Startpunkt der Serer-App
fun main(args: Array<String>): Unit =
    io.ktor.server.netty.EngineMain.main(args)

// Reaktion der App auf verschiedene GET-Requests
@kotlin.jvm.JvmOverloads
fun Application.module(testing: Boolean = false) {
    routing {
        get("/") {
            call.respondText("HELLO WORLD!",
                contentType = ContentType.Text.Plain)
        }

        get("/html-dsl") {
            call.respondHtml {
                body {
                    h1 { +"HTML" }
                    ul {
                        for (n in 1..10) {
                            li { +"$n" }
                        }
                    }
                }
            }  // Ende des respondHtml-Lambda-Ausdrucks
        }      // Ende des get-Lambda-Ausdrucks
    }          // Ende des routing-Lambda-Ausdrucks
}              // Ende der Funktion module
```

Zum obigen Code sind einige Erklärungen angebracht:

- `main` ist wie üblich der Startpunkt des Programms. Dabei wird die Kontrolle einfach an die im Ktor-Plugin ausgewählte Engine (hier Netty) übergeben. Deren `main`-Methode erzeugt eine `Application`-Instanz, die während des gesamten Programmablaufs die Schnittstelle zu Ihrem eigenen Code ist. Die Server-App läuft, bis Sie explizit stoppen.
- Jede Ktor-App muss zumindest eine Erweiterungsfunktion für die `Application`-Klasse definieren. Wenn Sie das Ktor-Plugin verwenden, muss der Name dieser Funktion `module` lauten. (Das ist in der schon erwähnten Datei `resources/application.conf` festgelegt. In dieser Datei können Sie bei Bedarf weitere Modulfunktionen nennen.)

- Mit der in der `Application`-Klasse definierten Methode `routing` legen Sie in einem Lambda-Ausdruck fest, wie Ihre App auf diverse Requests reagieren soll. Typische Requests sind GET (HTTP-Dokument anfordern) oder POST (Formulardaten übermitteln). Entsprechend legen Sie im Lambda-Ausdruck mit `get` oder `post` fest, welcher Code für welche Adresse ausgeführt werden soll.
- `get("/")` bestimmt also, wie Ihr Server einen GET-Request beantworten soll, der direkt das Wurzelverzeichnis des Servers (also *http://0.0.0.0:8080*) betrifft. Der Server soll in diesem Fall ein reines Textdokument (kein HTML-Dokument) senden, das lediglich die Zeichenkette `"HELLO WORLD!"` enthält.
- Schon interessanter ist die Antwort auf *http://0.0.0.0:8080/html-dsl*: In diesem Fall erzeugt Ihr Programm ein HTML-Dokument, das aus einer Überschrift (`<h1>`) und einer Aufzählung (`<ul>`) mit mehreren Punkten (`<li>`) besteht. Spätestens jetzt ist die Vorliebe der Ktor-Entwickler für Lambda-Ausdrücke unübersehbar.

  Auf das Zusammensetzen von HTML-Dokumenten durch den verschachtelten Aufruf von DSL-Funktionen gehe ich in Abschnitt 28.4, »HTML- und CSS-Dokumente zusammensetzen«, näher ein.

Importe

Der vorhin präsentierte Beispielcode erfordert die folgenden Importe:

```
import io.ktor.application.*
import io.ktor.response.*
import io.ktor.routing.*
import io.ktor.http.*
import io.ktor.html.*
import kotlinx.html.*
```

Die Ktor-Dokumentation empfiehlt, nicht einzelne Klassen zu importieren, sondern mit `*` den gesamten Inhalt eines Pakets. Wenn Sie möchten, können Sie auch Ihr IntelliJ-Projekt dahingehend konfigurieren. Dazu öffnen Sie in den Einstellungen das Dialogblatt EDITOR • CODE STYLE • KOTLIN und aktivieren zweimal die Option USE IMPORT WITH *. Eine ausführlichere Anleitung finden Sie hier:

https://ktor.io/docs/code-style.html

27

Fehlende Bibliotheken

Wenn Sie an eigenen Projekten arbeiten, wird es immer wieder passieren, dass IntelliJ Klassen nicht findet und daher keinen passenden Import hinzufügen kann. Schuld ist dann fast immer eine fehlende Zeile im `dependecies`-Abschnitt von `build.gradle`. Recherchieren Sie, in welchem Paket Ihr Paket versteckt ist, und fügen Sie die entsprechende `implementation`-Zeile zu `build.gradle` hinzu.

Eine Hilfe bei der Suche nach dem richtigen Paket ist das Ktor-Plugin. Dieses können Sie leider nur für neue Projekte verwenden, nicht zur nachträglichen Veränderung vorhandener Projekte. Es spricht aber nichts dagegen, rasch ein Testprojekt einzurichten, dabei die erforderlichen Optionen auszuwählen und dann die `build.gradle`-Versionen Ihres vorhandenen Projekts und des Testprojekts miteinander zu vergleichen.

27.2 Beispiel: URL-Verkürzer

URL-Shortener-Dienste verkürzen lange Adressen von Webseiten zu einer Kurzform wie *https://tinyurl.com/qtmk82j* oder *https://bit.ly/2UELobH*. Das folgende Beispiel zeigt, wie Sie einen vergleichbaren Dienst mit wenigen Zeilen Code selbst realisieren können.

Ich habe mich bei der Implementierung auf das absolute Minimum beschränkt. Wenn Sie die Adresse *http://0.0.0.0:8080* besuchen, erscheint ein schmuckloses Formular, an das Sie Ihre URL übergeben (siehe Abbildung 27.3). Auf die Übertragung des Formulars antwortet die App mit der verkürzten URL, die außer dem Hostnamen der App lediglich 6 Zeichen enthält (siehe Abbildung 27.4). Wenn ein Anwender auf einen derartigen Kurz-Link klickt, zeigt die App, wohin dieser Link führt (siehe Abbildung 27.5). Erst ein weiterer Link führt zum gewünschten Ziel. (Die meisten »richtigen« URL-Shortener sparen sich diesen sicherheitstechnisch wünschenswerten Zwischenschritt.)

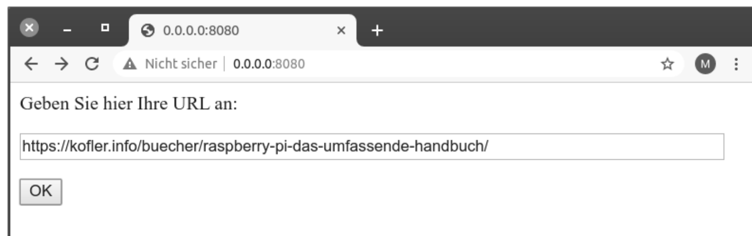

Abbildung 27.3 Formular für die ursprüngliche Adresse

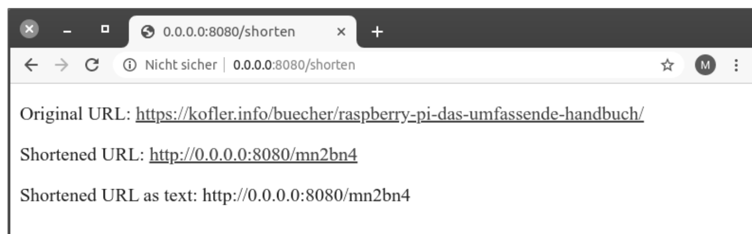

Abbildung 27.4 Anzeige der verkürzten Adresse

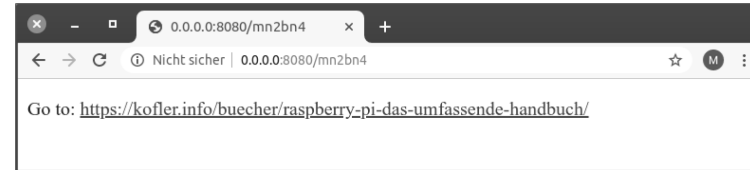

Abbildung 27.5 Weiterleitung an die Zieladresse

URL-IDs erzeugen

Als Startpunkt dient ein neues Ktor-Projekt, das Sie am einfachsten mit dem Plugin einrichten. Dabei aktivieren Sie als einzige Feature-Option HTML DSL. Die folgende Beschreibung des Codes ist bewusst relativ knapp gehalten: Auf die Grundlagen der Ktor-Programmierung, vom Routing bis zur Erzeugung neuer HTML-Dokumente, komme ich in Kapitel 28 noch ausführlich zu sprechen.

Der App fehlt ein echter Datenspeicher für die URLs. Diese werden stattdessen in einer `MutableMap` gespeichert, gehen also verloren, sobald das Programm endet:

```
// Beispielprojekt url-shortener, Datei Application.kt
// statt einer Datenbank
val urlMap = mutableMapOf<String, String>()
```

Jeder gespeicherten URL wird eine aus sechs zufälligen Zeichen zusammengesetzte Zeichenkette zugeordnet. Für die Generierung dieser Zeichenketten ist `randomId` zuständig. Die `do-while`-Schleife stellt sicher, dass die Funktion keine schon genutzte ID zurückgibt. Insgesamt gibt es bei den gewählten Parametern ca. 1,8 Milliarden Möglichkeiten, eine ID-Zeichenkette zu generieren. Nachdem etwa die Hälfte dieser IDs vergeben ist, würde `randomId` spürbar langsamer werden, weil die Suche nach einer freien ID zunehmend aufwendiger würde.

Falls die App auf einem richtigen Server (und nicht nur im Testbetrieb) ausgeführt wird, kann es theoretisch passieren, dass mehrere URLs gleichzeitig verkürzt werden sollen. Dann wird `randomId` möglicherweise in zwei unterschiedlichen Threads parallel ausgeführt und kann mit viel Pech zweimal die gleiche ID generieren. `urlMap[id] = "reserved"` versucht diesen Fall weitestgehend auszuschließen. Wirklich »wasserdicht« wäre hier aber nur eine Lösung auf Basis eines Datenbanksystems mit Transaktionen.

```
fun randomId() : String {
    val chars = "abcdefghijklmnopqrstuvwxyz0123456789"
    var id: String
    do {
        id = (1..6).map { chars.random() }.joinToString("")
    } while (urlMap.containsKey(id))
```

```
    urlMap[id] = "reserved"
    return id
}
```

Routing

Die App verarbeitet lediglich drei Requests:

- Ein GET-Request für die Startseite liefert das in der Variablen `shortenForm` gespeicherte HTML-Formular.
- Bei einem GET-Request in der Form *hostname/id* durchsucht das Programm die `urlMap` nach der gewünschten Adresse und zeigt diese gegebenenfalls an.
- Für die Verarbeitung des Formulars ist ein PUT-Request mit der Adresse *hostname/shorten* zuständig:

```
val shortenForm = """
    <html><body><form action='/shorten' method='post'>
    <p><label for='url'>Geben Sie hier Ihre URL an:</label>
    <p><input type='text' name='url' id='url' size='100'>
    <p><input type='submit' value='OK'>
    </form></body></html>
""".trimIndent()

fun Application.module(testing: Boolean = false) {
    routing {
        get("/") {
            // Formular anzeigen
            call.respondText(shortenForm, ContentType.Text.Html)
        }
        get("/{id}") {
            // Weiterleitung an die gewünschte Adresse
            lookupUrl()
        }
        post("/shorten") {
            // Formulardaten verarbeiten
            shortenUrl()
        }
    }
}
```

URL-Lookup

Um eine allzu unübersichtliche Verschachtelung von Lambda-Ausdrücken zu vermeiden, habe ich die URL-Suche bzw. Formularverarbeitung an zwei Funktionen ausgelagert: `lookupUrl` und `shortenUrl`. Deren Deklaration ist nicht ganz trivial:

- Das Schlüsselwort `inline` vermeidet den Overhead eines richtigen Funktionsaufrufs. Da die Funktionen jeweils nur an einer Stelle im Routing-Code genutzt werden, verschwendet `inline` auch keinen Platz im kompilierten Code.
- Die Funktionen sind als Erweiterungen zu `PipelineContext<Unit, ApplicationCall>` definiert. Das ist der Datentyp von `this` innerhalb der `get`- und `post`-Lambda-Ausdrücke.
- Der Routing-Code wird grundsätzlich in Koroutinen ausgeführt. Die beiden Hilfsfunktionen müssen daher mit `suspend` gekennzeichnet werden.

Die Kombination dieser drei Techniken ermöglicht es, dass `this` innerhalb von `lookupUrl` bzw. `shortenUrl` denselben Inhalt hat wie im Lambda-Ausdruck – und dass `this` in den Funktionen nicht explizit genannt werden muss. (Beispielsweise bezieht sich `call.request` auf das implizite Objekt `this`.) Hintergründe zu diesen doch schon recht fortgeschrittenen Syntaxspielarten von Kotlin können Sie bei Bedarf in Abschnitt 11.8, »Inline-Funktionen«, in Abschnitt 13.8, »Extensions«, sowie in Kapitel 17, »Asynchrone Programmierung«, nachlesen.

Nun aber zur eigentlichen Aufgabe von `lookupUrl`: Die Methode extrahiert aus der Request-URL die ID-Nummer. Wenn das gelingt und die ID in `urlMap` enthalten ist, generiert `lookupUrl` ein HTML-Dokument mit einem Link an die Zieladresse.

```
inline suspend fun
    PipelineContext<Unit, ApplicationCall>.lookupUrl()
{
    val id = call.parameters["id"]
    if (id is String && urlMap.containsKey(id)) {
        val href = urlMap[id]!!
        call.respondHtml {
            body {
                p { +"Go to: "
                    a(href) { +href }
                }
            }
        }
    } else {
        call.respondText { "Invalid link." }
    }
}
```

Wenn Sie anstelle der Anzeige des Links samt vollständiger Adresse eine sofortige Weiterleitung zur Zielseite wünschen, ersetzen Sie `respondHtml` durch `respondRedirect`:

```
call.respondRedirect(href)
```

Formularauswertung

Für die Auswertung der Formulardaten sowie für die Generierung und Speicherung des Short-Links ist `shortenUrl` zuständig. Dort wird zuerst überprüft, ob die Formulardaten korrekt sind: Es muss eine POST-Zeichenkette mit dem Namen `url` existieren, und aus dieser Zeichenkette muss sich ohne Fehler ein `URL`-Objekt erzeugen lassen.

Wenn diese Voraussetzungen erfüllt sind, überprüft die `shortenUrl`, ob die URL bereits in `urlMap` enthalten ist, und ermittelt den entsprechenden Key. Andernfalls erzeugt ein Aufruf von `randomId` eine neue ID. Die vollständige Adresse (Value) und die ID (Key) werden in der `urlMap` gespeichert. Als Feedback für den Anwender werden die ursprüngliche URL und die verkürzte URL als HTML-Dokument zurückgegeben. Die Short-URL wird sowohl als Link als auch in Textform angezeigt. Das vereinfacht es, die URL zu markieren und in die Zwischenablage zu kopieren. (In einer »echten« App könnten Sie diesen Vorgang mit einem Button und ein wenig JavaScript-Code für den Anwender weiter vereinfachen.)

```
inline suspend fun
    PipelineContext<Unit, ApplicationCall>.shortenUrl()
{
    val req = call.request
    val host = "%s://%s:%s".format(req.origin.scheme,
                                   req.host(), req.port())
    val data = call.receiveParameters()
    val url = data["url"]
    if (url is String) {
        // simple Validierung
        val result = runCatching { URL(url) }
        if (result.isSuccess) {
            var rid: String

            // Existiert die URL schon in der Map?
            val keys = urlMap.filterValues { it == url }.keys
            if (keys.size > 0)
                // ja, Key weiterverwenden
                rid = keys.first()
            else  {
                // nein, neuen Key erzeugen
                rid = randomId()
                urlMap[rid] = url
            }

            println(urlMap)
            val href = "$host/$rid"
            call.respondHtml {
                body {
                    p { +"Original URL: "
                        a(url) { +url }
                    }
                    p { +"Shortened URL: "
                        a(href) { +href }
                    }
                    p { +"Shortened URL as text: $href" }
                }
            }
            return
        }
    }
    call.respondText("Invalid data.")
}
```


27.3 Beispiel: URL-Verkürzer mit Datenbank-Server

Der Code des im vorigen Abschnitt präsentierten Beispiels ist leicht nachzuvollziehen, außerdem lässt sich die App ohne weitere Vorbereitungen sofort ausprobieren. Sie hat aber den Nachteil, dass sie wieder alles vergisst, wenn das Programm endet, und ist insofern nicht besonders realitätsnah. Deswegen präsentiere ich Ihnen hier eine verbesserte Version, die die URLs und die verkürzten Code in einer MySQL- oder MariaDB-Datenbank speichert.

Voraussetzungen

Bevor Sie das Beispielprojekt ausprobieren, benötigen Sie auf Ihrem Rechner oder in Ihrem lokalen Netzwerk einen MySQL- oder MariaDB-Server. (Empfehlenswert ist für solche Aufgabenstellungen Docker. Damit können Sie das Datenbanksystem für einen Test rasch einrichten und später ebenso schnell wieder löschen.)

Auf dem Datenbank-Server richten Sie nun eine leere Datenbank sowie einen Benutzer-Account mit Zugriffsrechten auf diese Datenbank ein. Am einfachsten führen Sie dazu in einem MySQL-Client die folgenden drei Kommandos aus:


```
CREATE DATABASE urldb DEFAULT CHARSET = utf8;
CREATE USER urluser@'%' IDENTIFIED BY 'geheim';
GRANT ALL ON urldb.* TO urluser@'%';
```

Die Datei »build.gradle«

Damit Ihr Ktor-Projekt das Exposed-Framework, den MySQL/MariaDB-Treiber sowie den Hikari-Connection-Pool verwenden kann, fügen Sie `build.gradle` die folgenden Zeilen hinzu. (Hintergrundinformationen finden Sie in Kapitel 19, »Datenbankzugriff (Exposed)«.)

```
// Beispielprojekt url-shortener-db, Datei build.gradle
dependencies {
  ...
  implementation "org.jetbrains.exposed:exposed-core:0.26.1"
  implementation "org.jetbrains.exposed:exposed-dao:0.26.1"
  implementation "org.jetbrains.exposed:exposed-jdbc:0.26.1"
  implementation "mysql:mysql-connector-java:8.0.19"
  implementation "com.zaxxer:HikariCP:3.4.2"
}
```

Datenbankverbindung herstellen

In `Application.module` stellen Sie die Verbindung zur Datenbank her. Weil der Webserver im realen Betrieb viele Anfragen parallel verarbeiten soll, ist es zweckmäßig, sich nicht auf *eine* Verbindung zu verlassen, sondern gleich einen ganzen Pool von Verbindungen zu verwenden.

```
// Beispielprojekt url-shortener-db, Datei Application.kt
fun Application.module(testing: Boolean = false) {
    // Verbindungs-Pool für die Datenbank
    val hikConfig = HikariConfig().apply {
         jdbcUrl = "jdbc:mysql://localhost/urldb"
         driverClassName = "com.mysql.cj.jdbc.Driver"
         username = "urluser"
         password = "geheim"
         maximumPoolSize = 12
    }
    Database.connect(HikariDataSource(hikConfig))

    // Tabelle erzeugen, wenn notwendig
    transaction {
        SchemaUtils.create(Urls)
    }

    // Verarbeitung von Requests
    routing {
       // wie bisher ...
    }
}
```


URL-Klasse

Damit das Exposed-Framework die Tabelle `Url` erzeugen kann, muss ein Singleton-Objekt die Tabelle beschreiben (`object Urls`). Die Tabelle besteht aus drei Spalten, einer ID-Spalte, einer Textspalte mit der vollständigen URL und einer weiteren Textspalte mit dem verkürzten Code.

Um einen effizienten Betrieb sicherzustellen, habe ich beide Spalten mit einem Index ausgestattet. Dabei ist allerdings eine MySQL/MariaDB-spezifische Einschränkung zu beachten: Indizes für Textspalten sind nur erlaubt, wenn die Textlänge in der Spalte auf 3072 Byte beschränkt ist. Da Exposed bei Textspalten in MySQL/MariaDB automatisch die Codierung UTF-8 verwendet und ein Zeichen bis zu drei Byte beanspruchen kann, ergibt sich eine maximale Textlänge für die URL von ca. 1000 Zeichen.

Sollte Ihnen das zu wenig sein, müssen Sie entweder auf den Index verzichten (was bei einem realen Einsatz aber nicht sinnvoll ist) oder das Datenbank-Design optimieren. Denkbar wäre, die Tabelle nicht durch Exposed, sondern manuell mit `CREATE TABLE` zu erzeugen und dabei einen (für URLs ausreichenden) 8-Bit-Zeichensatz zu verwenden. Noch besser wäre die parallele Speicherung eines Hash-Codes, um anhand dessen den URL-Abgleich durchzuführen. Das würde den Kotlin-Code aber natürlich komplexer machen, weil denkbar ist, dass zwei unterschiedliche URLs den gleichen Hash-Code aufweisen.

Für den Zugriff auf die Datensätze der Tabelle ist außerdem eine von `IntEntity` abgeleitete Klasse erforderlich (`class Url`). Hintergründe zur Notwendigkeit des `Urls`-Objekts und der `Url`-Klasse finden Sie in Abschnitt 19.3, »Data Access Objects (DAO)«.

```
// Beispielprojekt url-shortener-db, Datei Url.kt
object Urls : IntIdTable() {
    val fullurl = varchar("fullurl", 1000).index()
    val shortcode = varchar("shortcode", 10).index()
}
class Url(id: EntityID<Int>): IntEntity(id) {
    companion object : IntEntityClass<Url>(Urls)
    var fullurl by Urls.fullurl
    var shortcode by Urls.shortcode
}
```


URL-Lookup

Bei einem GET-Request der Form `/<id>` wird die Funktion `lookupUrl` ausgeführt. Da es sich dabei um eine Suspended Function handelt, muss anstelle der im Exposed-Framework üblichen Methode `transaction` deren asynchrone Variante `suspendedTransactionAsync` verwendet werden. `Url.find` durchsucht die Datenbank nach einem entsprechenden Eintrag und zeigt dann mit `respondHtml` einen Link zur Zielseite an:

```
// Beispielprojekt url-shortener-db, Datei Application.kt
suspend fun PipelineContext<Unit, ApplicationCall>.lookupUrl() {
   val id = call.parameters["id"] ?: "invalid data"
   suspendedTransactionAsync {
       val url = Url.find { Urls.shortcode.eq(id) }.firstOrNull()
       if (url is Url) {
           val href = url.fullurl
           call.respondHtml {
               println("now in respondHtml")
               body {
                   p { +"Go to: "
                       a(href) { +href }
                   }
               }
           }
       } else
           call.respondText { "Invalid link." }
   }
}
```

Formularauswertung

Etwas aufwendiger ist der Code zur Auswertung der Formulardaten: In den ersten Zeilen von `shortenUrl` wird überprüft, ob die per Formular übertragene URL korrekt und nicht zu lang ist. Nur unter dieser Voraussetzung wird eine Datenbanktransaktion gestartet: Wenn die URL bereits gespeichert wurde, wird der dabei verwendete Shortcode aus der Datenbank extrahiert und angezeigt; es ist nicht notwendig, einen neuen Eintrag zu erstellen.

Ist die URL hingegen nicht bekannt, wird in der `while`-Schleife ein neuer, zufälliger Shortcode generiert. Auch dabei muss sichergestellt werden, dass dieser Code nicht schon in der Datenbank verwendet wird. Anschließend kann der neue Eintrag in der `Url`-Tabelle gespeichert werden:

```
suspend fun PipelineContext<Unit, ApplicationCall>.shortenUrl() {
    val req = call.request
    val host = "%s://%s:%s".format(req.origin.scheme,
                                   req.host(), req.port())
    val data = call.receiveParameters()
    val url = data["url"]
    if (url !is String || url.length > 1000) {
        call.respondText("Invalid data.")
        return
    }
```



```
    // simple Validierung
    val result = runCatching { URL(url) }
    if (result.isFailure) {
        call.respondText("Invalid data.")
        return
    }
    suspendedTransactionAsync {
        val chars = "abcdefghijklmnopqrstuvwxyz0123456789"
        var id: String
        // Existiert URL schon in DB?
        val result = Url.find { Urls.fullurl.eq(url) }
        if (!result.empty()) {
            // ja, shortcode = id extrahieren
            id = result.first().shortcode
        } else {
            // nein, neuen Eintrag speichern
            do {
                id = (1..6).map { chars.random() }
                           .joinToString("")
            } while (!Url.find { Urls.shortcode.eq(id) }.empty())
            Urls.insert {
                it[fullurl] = url
                it[shortcode] = id
            }
        }
        // Short-Link anzeigen (Code wie im vorigen Beispiel)
        ...
    }
}
```