HANSER

Leseprobe

zu

„Programmieren lernen mit KOTLIN“

von Christian Kohls et al.

Print-ISBN: 978-3-446-46702-6
E-Book-ISBN: 978-3-446-46711-8
E-Pub-ISBN: 978-3-446-46723-1

Weitere Informationen und Bestellungen unter
http://www.hanser-fachbuch.de/978-3-446-46702-6
sowie im Buchhandel

Inhalt

Vorwort

Mit diesem Buch können Sie ohne Vorkenntnisse in die Programmierung einsteigen. Dabei werden Sie verschiedene Ansätze kennenlernen und praktisch anwenden. Nach der Lektüre des Buches können Sie kleinere Softwareprojekte entwickeln, also zum Beispiel eigene Ideen umsetzen, Aufgaben und Problemstellungen verstehen und lösen sowie Softwarespezifikationen in lauffähige Programme überführen. Sie können einfache Algorithmen selbst entwickeln und Standardalgorithmen und Datenstrukturen umsetzen. Sie können Apps für Android-Systeme entwickeln oder Programme für Server und Desktop-Rechner schreiben.

Die Welt des Programmcodes ist unsichtbar. Wir haben festgestellt, dass einige Konzepte besonders schwer zu begreifen sind und oft falsche Vorstellungen existieren. Es wurde daher großer Wert darauf gelegt, möglichst viele Konzepte mit Metaphern, praktischen Anwendungsbeispielen und Bildern zu veranschaulichen. Dabei bauen wir auf unseren langjährigen Erfahrungen in der Programmierausbildung auf. Am Ende des Buches können Sie Apps mit einer grafischen Benutzerschnittstelle entwickeln und aus unsichtbarem Code eine visuell ansprechende App entwickeln.

Dieses Buch richtet sich vor allem an Einsteiger und Anfänger. Es werden keine Vorkenntnisse vorausgesetzt. Gleichzeitig denken wir, dass auch fortgeschrittene Entwickler und Umsteiger von anderen Programmiersprachen von diesem Buch profitieren werden.

Alle Codebeispiele und zusätzliche Übungsaufgaben finden Sie im Download-Portal von Hanser-Plus. Den Zugangscode finden Sie am Anfang des Buches.

An dieser Stelle wollen wir uns bei unserer Lektorin Sylvia Hasselbach und unserem Korrektor Jürgen Dubau bedanken. Zudem danken wir Sandy Neumann für die Gestaltung zahlreicher Grafiken. Für den fachlichen Austausch möchten wir uns bei unseren Teamkollegen an der TH Köln bedanken. Insbesondere bei David Petersen, der wesentliche Inspirationen zu diesem Buch beigetragen hat.

Viel Spaß beim Coden und Entwickeln!

Christian Kohls, Alexander Dobrynin, Florian Leonhard Im Juli 2020

Kotlin vereint dabei viele erprobte Konzepte aus den letzten Jahrzehnten. Kotlin setzt konsequent auf Mechanismen, die sich bewährt haben und die in der Programmierpraxis zu einer erhöhten Produktivität führen – und dadurch letztlich mehr Freude bereitet. Ja, Kotlin macht Spaß!

1.1 Eine Sprache für viele Plattformen

Mit Kotlin können Sie für viele verschiedene Plattformen entwickeln.

Android: Kotlin gehört inzwischen zu den primären Entwicklungssprachen für die Android-Plattform von Google. Damit hat sie bereits jetzt eine hohe Relevanz, da sich Apps für Android mit Kotlin schneller, besser und leichter entwickeln lassen.

JVM: Programme, die in Kotlin geschrieben werden, können für die Java-Plattform kompiliert werden. Das heißt: Die Kotlin-Programme werden in Java-Bytecode übersetzt und laufen dann auf jeder Java Virtuellen Maschine (JVM). Diese JVMs gibt es für verschiedene Betriebssysteme. Ihre Programme können also auf verschiedenen Rechnersystemen ausgeführt werden.

JavaScript: Kotlin kann aber auch nach JavaScript übersetzt werden. Das heißt: Sie können die Programmierung von Webseiten auch mit Kotlin durchführen. Dies gilt sowohl für die Programmierung des Clients (also JavaScript, das im Browser ausgeführt wird) als auch des Servers (also JavaScript, das auf dem Server ausgeführt wird).

Kotlin Native und iOS: Darüber hinaus gibt es mit Kotlin Native den Ansatz, ein Kotlin-Programm direkt für eine bestimmte Rechnerarchitektur zu kompilieren. Es wird also nicht Bytecode für eine virtuelle Maschine wie die JVM erzeugt, sondern echter Maschinencode für die jeweilige Plattform (z. B. Mac, Windows oder Linux). Mit Kotlin Native können Sie auch Apps für iOS entwickeln.

1.2 Deshalb ist Kotlin so besonders

Hier noch einmal die wichtigsten Gründe, die für Kotlin sprechen:

Einfacher Einstieg: Kotlin lässt sich schneller lernen. Es ist verständlicher, und die Sprachelemente sind prägnanter, also ausdrucksstärker. Sie können sich besser auf die wesentlichen Konzepte fokussieren, da unnötiger „Boilerplate Code“ entfällt. Das ist Code, der eigentlich nicht nötig ist, z. B. weil er keine neuen Sachverhalte ausdrückt. Ein Beispiel sind die vielen setter- und getter-Methoden, die man in anderen Programmiersprachen wie zum Beispiel Java schreiben muss.

Eleganter: Viele Programme lassen sich mit Kotlin eleganter schreiben. Mehr noch: Man muss die Programme eleganter schreiben. Sie lernen also gleich den richtigen Stil und können diesen auch in anderen Sprachen einsetzen.

Effiziente Entwicklung: Vieles lässt sich in Kotlin kürzer und effizienter ausdrücken. Das spart nicht nur Zeit beim Schreiben des Quellcodes. Durch kurze, übersichtliche Programme lassen sich Fehler besser vermeiden, und Sie behalten besser den Überblick.

Effiziente Ausführung: Kotlin stellt viele optimierte Algorithmen für immer wieder anfallende Aufgaben zur Verfügung. Das Verarbeiten oder Sortieren von Listen ist in Kotlin sehr gut gelöst und leicht nutzbar.

Erprobte Konzepte: Kotlin baut auf erprobten Konzepten auf. Die Sprache enthält konzeptionell nichts Neues, dafür (fast) alles Gute aus den letzten 50 Jahren Softwareentwicklung. Die Sprache ist praktisch und effizient, bietet mehr Sicherheit und vereint objektorientierte und funktionale Programmierkonzepte.

Erklärt sich selbst: Der Quellcode ist lesbarer für Menschen. Sie können den Code von anderen Entwicklern (und womöglich Ihren eigenen Code selbst nach ein paar Wochen) viel besser verstehen.

Error-less: Viele Fehler, die man in anderen Sprachen leicht machen kann, werden in Kotlin gar nicht erst zugelassen oder zumindest leichter vermieden. So gibt es in Kotlin beispielsweise keine sogenannten *Null-Fehler* mehr.

Erfrischend: Viel Entwickler berichten, dass das Entwickeln mit Kotlin wieder mehr Spaß macht! Und tatsächlich bekommt man beim Entwickeln mit Kotlin leuchtende Augen. Und manchmal auch tränende Augen – vor Freude über Kotlin und Ärger darüber, dass früher so vieles anstrengender war.

1.3 Darauf dürfen Sie sich freuen

Aufbau des Computers: Programme laufen auf dem Prozessor eines Computers. Dabei verändern sie Daten im Speicher. Um zu verstehen, wie aus dem Quellcode, den Sie schreiben, lauffähige Programme auf dem Rechner werden, schauen wir uns zunächst den grundlegenden Aufbau von Computern an. Dabei werden wir sehen, wie aus lauter Nullen und Einsen bunte Bilder werden können. Da wir Daten im Speicher liegen haben, werden wir uns auch mit der Organisation des Speichers beschäftigen.

Basics: Zur Steuerung von Programmen benötigen wir ein paar grundlegende Zutaten. Dazu gehören Ausdrücke und Anweisungen, um dem Rechner zu sagen, was er ausführen soll. Mit Variablen können wir Daten speichern und später wieder darauf zugreifen. Mit Kontrollstrukturen können wir steuern, wie ein Programm abläuft. Somit können wir auf Benutzereingaben und verschiedene Zustände reagieren. Mit Funktionen werden wir wiederverwendbare Codebausteine definieren, aus denen sich komplexe Programme zusammensetzen lassen.

Objekte und Klassen: Wir werden Grundlagen über Objekte und Abstraktion als Klassen kennenlernen. Wie können wir ermitteln, welche Eigenschaften von Objekten relevant sind? Wie abstrahiere ich von konkreten Objekten in der Welt auf eine allgemeine Klasse? Klassen beschreiben die Struktur, die Eigenschaften und die Verhaltensweisen von allen Objekten, die zu einer Klasse gehören. Man spricht von Objektinstanzen (oder einfache Instanzen), wenn man sich auf die Exemplare einer Klasse bezieht. Das Erzeugen eines neuen Objekts erfolgt über Konstruktoren. Wir werden uns ansehen, wie Konstruktoren die erste Version eines Objekts bauen. Zudem werden wir Objekte miteinander in Beziehung setzen. Zum Beispiel werden wir einen Film aus Budget, Schauspieler und Regisseur zusammensetzen, um dies in einem Spiel zu verwenden.

Vererbung und Polymorphie Dies ist ein weiteres zentrales Thema der objektorientierten Programmierung. Klassen stehen in einer Vererbungshierarchie zueinander. Klassen können andere Klassen erweitern, wobei die Eigenschaften und Methoden einer Oberklasse an die Unterklasse vererbt werden. Wenn die übergeordnete Klasse Tasse die Eigenschaft „Volumen“ und die Methode „trinken“ besitzt, dann wird auch die untergeordnete Klasse Kaffeetasse diese Eigenschaften haben, sie kann aber um spezifische Eigenschaften (z. B. „Kaffeesorte“) und Methoden (z. B. „istNochHeiss“) ergänzt werden. Dabei kann das Verhalten von Methoden auch durch Überschreiben geändert werden (z. B. weil man aus einer Thermotasse anders trinkt). Eng verbunden mit der Definition von Klassen und Schnittstellen sind Typkonzepte und Polymorphie. Polymorphie bedeutet Vielgestaltigkeit. Wenn Sie an „Fahrzeug“ denken, dann kann dieses Fahrzeug sehr viele unterschiedliche Gestalten haben: Auto, Fahrrad, Kutsche. Dennoch können Sie allgemeine Eigenschaften und Methoden nutzen, wenn diese Gestalten auf einer abstrakteren Ebene vom gleichen Typ sind. So lässt sich für alle Fahrzeuge sagen: „fahre los“. Egal, ob es sich um eine Kutsche oder ein Fahrrad handelt. Die Umsetzung wird aber ganz unterschiedlich aussehen.

Robustheit: Damit wir auch ordentliche Programme schreiben können, werden wir uns mit verschiedenen Konzepten zur Robustheit von Code beschäftigen. Wie können Sie auf Fehler und Ausnahmesituationen reagieren? Was passiert, wenn eine Datei gelesen werden soll, die gar nicht existiert? Was passiert, wenn ein Server nicht erreichbar ist? Ausnahmezustand! Wir werden lernen, wie wir eine Operation versuchen können und das Scheitern schon einplanen und auffangen. Das funktioniert bestens für Situationen, wo Sie selbst keinen Einfluss darauf haben, ob etwas wie gewünscht funktioniert. Dann können Sie nämlich darauf reagieren. Denn das Auffangen ist viel komplizierter als die Fehlerkorrektur. Und vor allem wollen Sie bei der Entwicklung ja auch, dass Fehler erkannt und somit beseitigt werden können. Wir werden uns die Vor- und Nachteile für verschiedene Software-Designoptionen anschauen. Aus diesen Überlegungen heraus haben sich über die Jahre hinweg Entwurfsmuster entwickelt, die gute Lösungen für bestimmte Aufgaben darstellen.

Datensammlungen: Wir schauen uns verschiedene Möglichkeiten an, um Datensammlungen (z. B. Listen, Verzeichnisse, Paare, Mengen) abzubilden und Operationen darauf auszuführen. Zudem werden wir selbst Datenstrukturen entwickeln, um Objekte zu speichern. Dies wird unter anderem am Beispiel einer verketteten Liste illustriert.

UI Design und App-Entwicklung: Die Android-Plattform stellt spezielle Bibliotheken zur Verfügung, um grafische Oberflächen zu gestalten und die Funktionen von Android-Geräten zu nutzen. Wir werden eine grafische Oberfläche bauen und Code definieren, um auf Eingabeevents (Klick auf einen Button) zu reagieren.

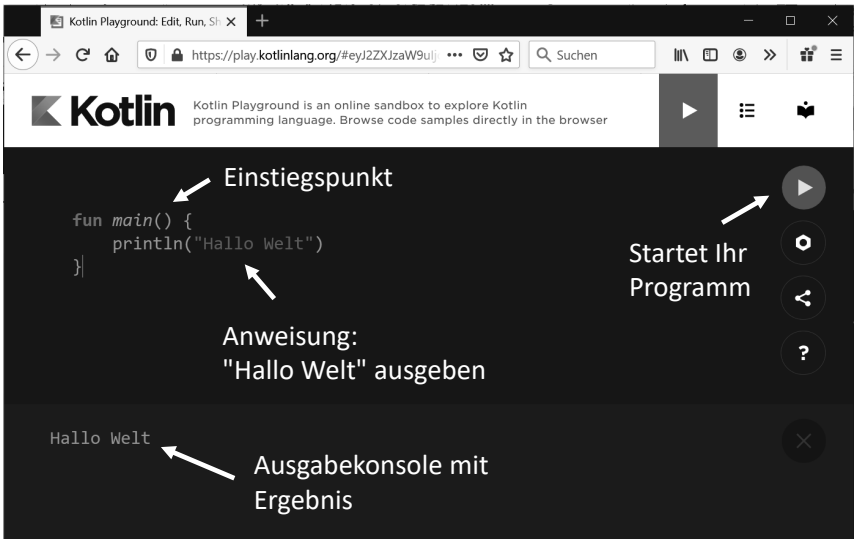
Nebenläufigkeit: Häufig müssen mehrere Dinge gleichzeitig ausgeführt werden. Bei Spielen sollen zum Beispiel gleichzeitig mehrere Figuren bewegt werden. Auf Ihrem Smartphone sollen gleichzeitig mehrere Bilder aus dem Internet heruntergeladen werden, während Sie weiterhin mit dem Programm interagieren. Für diese Nebenläufigkeit führt Kotlin das Konzept der *Coroutines* ein. Wir werden uns anschauen, wie sich nebenläufige Programme gestalten lassen und auf welche Stolpersteine geachtet werden muss.

7 Los geht's

Wie wir gesehen haben, müssen wir uns nicht mit lauter Nullen und Einsen auseinandersetzen. Stattdessen helfen uns Compiler dabei, einen Quelltext zu übersetzen. Der Quelltext kann sich in einer oder mehreren Dateien auf Ihrem Rechner befinden. Der Compiler übersetzt diese Dateien in Byte- oder Maschinencode.

Das Schreiben der Quelltexte kann in Prinzip mit jedem Editor geschehen. Ja, selbst der einfache Texteditor von Windows ist dafür geeignet, ein Kotlin-Programm zu schreiben, denn es handelt sich erst einmal um reine Textdateien. Diese Textdateien können wir dann über die Kommandozeile kompilieren. Jetzt benötigen wir für die Ausführung noch eine JVM. Diese können wir ebenfalls über die Kommandozeile starten und dabei angeben, welches kompilierte Programm ausgeführt werden soll. Das klingt kompliziert und nicht besonders komfortabel.

Zum Glück gibt es zwei Lösungen. Zum einen können Sie eine Online-Umgebung nutzen, um die ersten kleinen Programme zu schreiben. Darin können Sie sofort loslegen, ohne irgendetwas auf Ihrem Rechner zu installieren. Für größere Programme werden wir dann eine sogenannte *integrierte Entwicklungsumgebung* verwenden, die zahlreiche weitere Vorteile bietet. Doch jetzt legen wir erst einmal mit der Online-Umgebung los:



Wenn Sie auf *https://play.kotlinlang.org* gehen, dann können Sie direkt im Web-Browser Ihre ersten Kotlin-Programme schreiben.

Jedes Programm startet in einer `main`-Funktion. Mehr zu Funktionen erfahren Sie später. Wichtig ist erst einmal, dass es immer diese `main`-Funktion gibt und man darin Programmcode schreiben kann. Machen wir das doch direkt und geben etwas auf der Konsole aus:

```
println("Hallo Welt")
```

Dieser Programmcode wird beim Programmablauf ausgeführt. Wenn Sie auf den Run-Button klicken, dann wird das Programm kompiliert, also in ausführbaren Code übersetzt, und direkt ausgeführt. Das funktioniert allerdings nur, wenn das Programm fehlerfrei ist.

Die Online-Umgebung ist eine gute Spielwiese zum Ausprobieren. Für die effektive Entwicklung benötigen Sie jedoch eine integrierte Entwicklungsumgebung, die Sie bei der Softwareentwicklung unterstützt.

7.1 Integrierte Entwicklungsumgebungen

Eine integrierte Entwicklungsumgebung (*Integrated Development Environment*, kurz IDE) unterstützt Sie bei der Entwicklung von Programmen. Sie bietet einen sehr leistungsfähigen Editor zum Schreiben des Programmcodes und verwaltet Ihre Programmstruktur. Zudem unterstützt Sie die IDE bei der Fehlersuche. Syntaxfehler werden sofort angezeigt. Zudem können Sie Ihren Programmcode mit einem Debugger schrittweise durchlaufen und so Fehler suchen. Damit werden wir uns später ausführlicher beschäftigen. Die IDE hilft uns auch dabei, die richtigen Operationen und Funktionen für bereits existierende Typen zu finden, denn der Editor bietet eine Codevervollständigung. Das bedeutet: Der Editor zeigt an, welche Operationen für ein bestimmtes Objekt möglich sind. Auch unsere eigenen Variablennamen müssen wir nicht vollständig von Hand eintippen. Stattdessen reichen die Anfangsbuchstaben, und der Editor schlägt vor, welcher Name verwendet werden soll.

Betrachten wir zunächst noch einmal den Quellcode. Dieser könnte, wie gesagt, auch in einem einfachen Texteditor entwickelt werden. Doch dieser bietet uns keinerlei Unterstützung beim Schreiben. So weist der einfache Texteditor nicht darauf hin, wenn wir uns vertippt haben und syntaktisch falsche Anweisungen schreiben. Von modernen Textverarbeitungsprogrammen sind wir inzwischen Besseres gewöhnt. Diese heben Rechtschreib- und Grammatikfehler hervor und korrigieren diese teils noch beim Tippen. Allerdings helfen uns die Textverarbeitungen eben nur beim Schreiben von Texten in natürlicher Sprache.

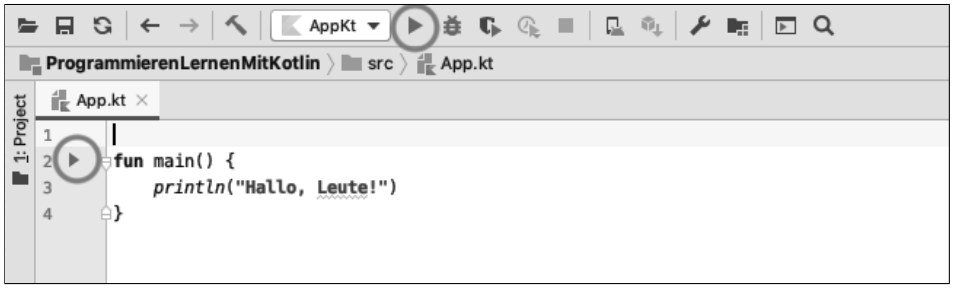
Eine integrierte Entwicklungsumgebung spricht dagegen die Sprache des Programmierens. Eine IDE unterstützt uns dabei, syntaktisch richtige Programme zu schreiben. Dies geschieht auf unterschiedliche Weise:

- Hervorheben durch syntaktische Fehler
- Vorschläge zum Ausbessern der Fehler
- Syntax-Highlighting zum Hervorheben verschiedener Programmelemente
- Vorschläge für die Verwendung von bereits existierenden Variablen oder Funktionen
- Codevervollständigung für bereits existierende Namen und typische Konstrukte

Die IDE analysiert schon beim Programmieren, welche Struktur unser Code hat. Dadurch kann die IDE Vorschläge unterbreiten, welche bereits existierenden Elemente wir wieder verwenden können. Das ist sehr praktisch, da uns so stets eingeblendet wird, welche Funktionen uns zur Verfügung stehen. Denn einen Großteil des Codes, den wir einsetzen, werden wir nicht selber schreiben. Stattdessen setzen wir Standardbibliotheken ein, die effiziente und getestete Funktionen bereitstellen. Die IDE hilft uns dabei zu entscheiden, welche Funktionen wir einsetzen können und wie diese aufgerufen werden. Wenn Sie große Projekte starten, dann hilft Ihnen eine IDE auch bei der Umstrukturierung des Programmcodes. Es gibt noch zahlreiche weitere Annehmlichkeiten, die Sie während der Entwicklung Schritt für Schritt schätzen lernen.

Wir werden eine IDE namens *IntelliJ* für einen Großteil des Buches verwenden. In Teil VII, „Android“, verwenden wir *Android Studio*. Dabei handelt es sich um eine spezielle, vorkonfigurierte Variante von IntelliJ, die für die Android-Plattform optimiert wurde.

Eine Aufgabe der IDE ist es zudem, alle erforderlichen Programmteile zu übersetzen und mit den eingesetzten Bibliotheken zu verknüpfen. Wenn Sie Ihr geschriebenes Programm laufen lassen möchten, dann müssen Sie nur noch den Play-Button drücken. Der Play-Button am linken Rand des Quellcodes erscheint neben allen `main`-Funktionen, die Sie in Ihrem Projekt haben. Klicken Sie auf diesen Button, um das Projekt zu kompilieren und diese `main`-Funktion als Einstiegspunkt zu verwenden. Wenn Sie dagegen auf den Play-Button oben in der Toolbar klicken, dann wird stets die zuletzt gewählte `main`-Funktion wieder als Einstiegspunkt verwendet. Daher müssen Sie auch beim Ausführen eines Programms in einem neu angelegten Projekt auf den Play-Button links neben der `main`-Funktion klicken. So legen Sie gleichzeitig fest, welche `main`-Funktion ausgeführt werden soll.



Wenn das Quellprogramm fehlerfrei ist, dann wird es in einem Rutsch kompiliert, alles Nötige eingebunden und dann auch gleich ausgeführt. Wenn beim Übersetzen Fehler auftreten, können Sie mit einem Klick an die richtige Stelle springen. Die IDE stellt also eine enge Verknüpfung zwischen dynamischer Programmausführung und statischem Quelltext her. Dies kann sogar noch einen Schritt weiter gehen, wenn Sie mit dem Debugger arbeiten.

Syntaktische Fehler können bereits vor dem Kompilieren von der IDE angezeigt werden. So weist Sie die IDE darauf hin, wenn Sie Variablen und Daten verwenden, die es gar nicht gibt. Genauso werden Sie darauf hingewiesen, wenn Sie Operationen ausführen wollen, dies es nicht gibt oder die für bestimmte Werte nicht erlaubt sind. Das sind sogenannte Typfehler.

Doch wir haben auch gesehen, dass es eine Reihe von Fehlern gibt, die sich erst zur Laufzeit zeigen. Um diese Fehler zu beseitigen, ist es meist notwendig, genau nachzuvollziehen,

was eigentlich beim Ablauf des Programms geschieht. Werden die Variablen so gesetzt und verändert, wie wir dies im Kopf geplant haben? Oder haben wir etwas irgendwo vergessen?

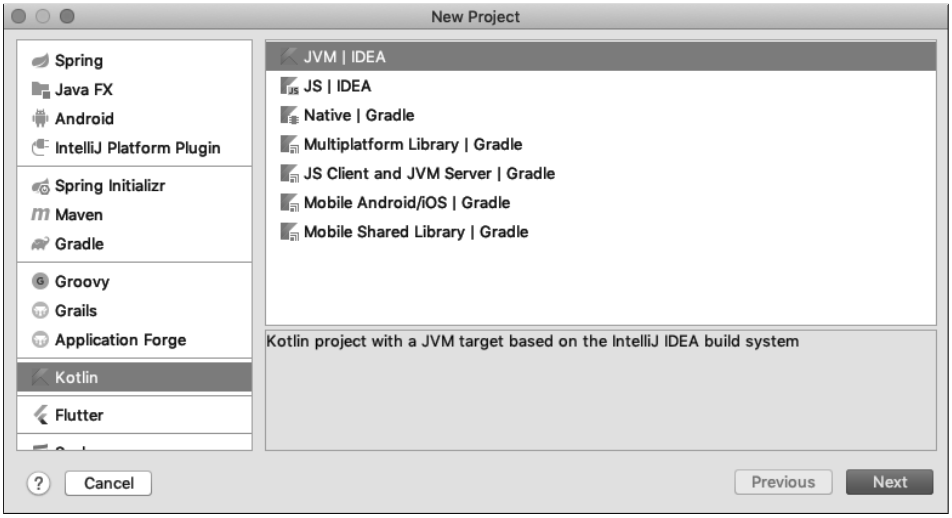
Dies lässt sich gut nachvollziehen, wenn man den Ablauf wirklich durchspielt. Und zwar nicht nur im Kopf, sondern tatsächlich durch Ausführen des Programms. Genau dafür stellen IDEs einen *Debugger* zur Verfügung. Wie Sie mit dem Debugger arbeiten, werden Sie in Kapitel 23, „Debugger", erfahren.

Gerade für umfangreichere Programme ist das Debuggen unerlässlich, um einen Fehler aufzuspüren. Daher wollten wir es bereits an dieser Stelle kurz erwähnen. Wir haben häufig beobachtet, dass gerade Programmiereinsteiger oft versuchen, das Problem eines fehlerhaften Programms rein theoretisch und durch Nachdenken zu lösen. Ein Fehler lässt sich aber viel, viel einfacher lösen, wenn man geradezu sieht, was falsch läuft. Genau hierfür ist der Debugger da. Und er hilft Ihnen auch dabei, Programmieren zu lernen.

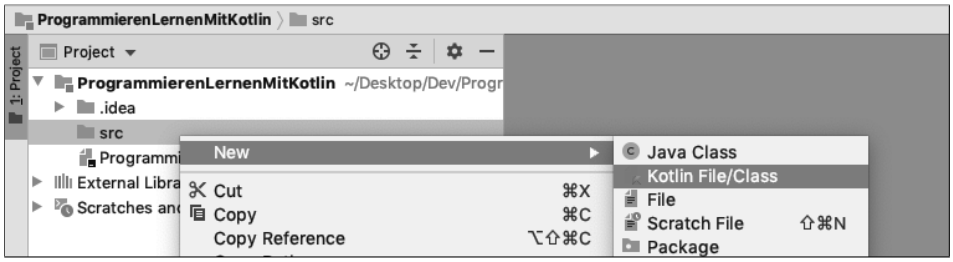
7.2 Projekt anlegen

Die IntelliJ Community Edition ist kostenlos, und Sie sollten diese auf Ihrem Rechner installieren, um darin alle Beispiele in diesem Buch als Projekt anzulegen.

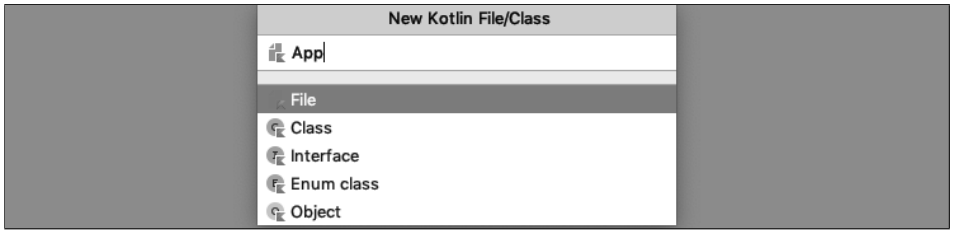
So legen Sie ein neues Projekt an. Gehen Sie im Menü auf File -> New -> Project und wählen links *Kotlin* und rechts *JVM / IDEA* aus:



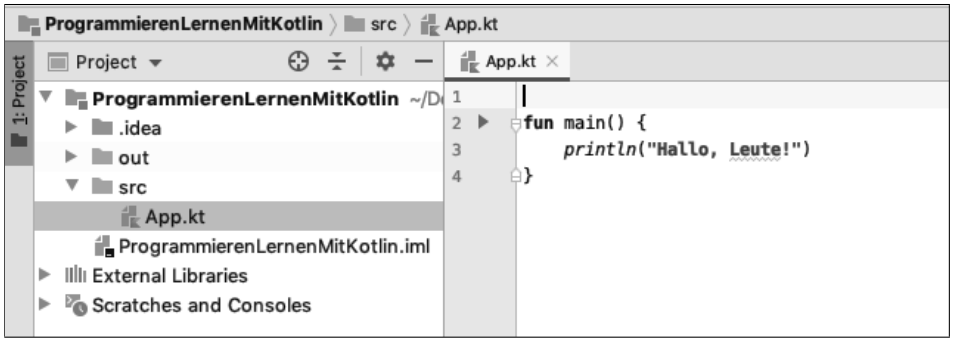
Danach vergeben Sie einen Namen für das Projekt und klicken alle Punkte durch, bis das Projekt erstellt wurde. In dem neuen Projekt gibt es links einen Projekt-Browser. Dort finden Sie den *src*-Ordner. Darin werden alle Quelldateien (Source Code) angelegt. Klicken Sie mit der rechten Maustaste auf den src-Ordner und erstellen Sie eine neue Kotlin-Datei:



Sie können der Datei einen beliebigen Namen geben, wie z. B. `App`:



Jetzt können Sie Ihr erstes eigenes Programm schreiben:



Klicken Sie auf den grünen Run-Pfeil, um das Programm auszuführen. Wenn alles richtig eingegeben wurde, dann erscheint unten in der Ausgabe `"Hallo, Leute!"`:

Dies kann beim ersten Mal etwas dauern, da die Entwicklungsumgebung erstmalig alle benötigten Dateien vorbereitet (der sogenannte Build-Prozess).

Um den Flächeninhalt eines Kreises zu berechnen, müssen wir den Radius mit 2 potenzieren und das Ergebnis mit PI multiplizieren. Das erste Argument der `pow`-Funktion ist der Radius `3.5`. Das zweite Argument ist der Exponent `2.0`.

Die Variable `PI` kommt aus der `Math`-Bibliothek von Kotlin. Der Wert von `PI` ist die Kreiszahl mit 20 Nachkommastellen.

Evaluieren wir einmal den Ausdruck:

```
1  > PI * pow(3.5, 2.0)
2  > PI * pow(3.5, 2.0)
3  > 3.141592653589793 * pow(3.5, 2.0)
4  > 3.141592653589793 * pow(3.5, 2.0)
5  > 3.141592653589793 * pow(3.5, 2.0)
6  > 3.141592653589793 * pow(3.5, 2.0)
7  > 3.141592653589793 * 12.25
8  > 38.48451000647496
```

Die Schritte ähneln dem Beispiel davor sehr. Die Besonderheit ist hier die Variable. Eine Variable wird evaluiert, indem der Name durch den zugewiesenen Wert ersetzt wird. Im Fall von `PI` ist das die Kreiszahl.

Natürlich gibt es noch viel mehr unterschiedliche Ausdrücke, die wir im Laufe des Buches kennenlernen. Allerdings haben Sie jetzt schon einmal eine Orientierung dafür, wie Ausdrücke zu einem Wert evaluiert werden.

Beim Programmieren folgen wir vielen Regeln aus der Mathematik. Grundsätzlich evaluieren wir gemäß der Präzedenz von links nach rechts. Bei Operatoren oder Funktionen werden zuerst die Operanden bzw. die Argumente evaluiert. Die Präzedenz gibt die Gewichtung an. Die Multiplikation ist beispielsweise höher gewichtet als die Addition. Die Präzedenz können wir mit Klammern beeinflussen.

8.3 Zusammenspiel von Werten und Typen

Werte und Typen tauchen immer in Kombination auf. Es gibt keinen Wert ohne Typ und keinen Typ ohne Wert[1].

Werte sind wichtig, weil wir damit *konkrete Daten* erzeugen. Ohne Werte sind die meisten Funktionen für uns nutzlos. Die `pow`-Funktion benötigt zum Beispiel zwei Werte vom Typ `Double`. Ansonsten können wir die Funktion nicht verwenden. Doch Werte sind nicht nur Zahlen. Hinter Zeichenketten, Bildern, Videos usw. verbergen sich Werte. Für den Computer sind alle Daten eine Folge von Nullen und Einsen. Für uns haben diese Daten eine bestimmte Bedeutung. Der Unterschied zwischen einer Zahl, einer Zeichenkette und einem Bild ist die semantische Auslegung der Daten. Wir verwenden Typen unter anderem dafür, damit der Computer unsere Bedeutung teilt und unsere Absicht versteht.

[1] Ausnahmen bestätigen die Regel. Der Typ Nothing hat beispielsweise keinen Wert. Mehr dazu in Abschnitt 9.7, „Nothing“.

Technisch gesehen legt der Typ fest, wie der Computer eine Kombination aus Nullen und Einsen interpretiert. Das ist ein Detail, mit dem wir uns in sogenannten höheren Programmiersprachen wie Kotlin nicht auseinandersetzen müssen. Für uns legt der Typ fest, welche Werte erlaubt und welche Operationen mit Werten des Typs möglich sind.

8.3.1 Typprüfungen durch den Compiler

Wie spielen nun Werte und Typen zusammen? Betrachten wir dafür einmal den Vergleichs-Operator (==). In den letzten beiden Abschnitten haben wir mit diesem Operator einmal zwei Werte vom Typ `Int` und einmal zwei Werte vom Typ `Double` miteinander verglichen. Das ist möglich, weil der Vergleichs-Operator für **unterschiedliche Typen definiert** ist. Wir können sogar zwei `Boolean`- oder zwei `String`-Werte miteinander vergleichen. Voraussetzung ist, dass beide Operanden **denselben Typ** haben. Was nicht funktioniert, sind Vergleiche, wo die Operanden unterschiedliche Typen haben. Schauen wir uns dazu ein paar Beispiele an:

```
fun main() {
    3.5 == 1
    "Hallo" == true
    false == 0
}
```

IntelliJ zeigt bei allen drei Zeilen einen Fehler an. Die Fehlermeldung des ersten Ausdrucks ist: „*Operator '==' cannot be applied to 'Double' and 'Int'*“. Die Fehlermeldung kommt vom Compiler. Der Compiler überprüft jeden Ausdruck unter anderem auf die Typkompatibilität. Der Compiler schaut sich in diesem Beispiel die Typen der Operanden und den Typ des Operators an. Wenn alle drei Typen kompatibel sind, wird der Ausdruck ausgewertet. Wenn nicht, wird ein entsprechender Typfehler angezeigt. Die Prüfung auf Typkompatibilität ist bei diesem Beispiel noch relativ einfach. Der Compiler schaut nach, ob ein Vergleichs-Operator für den Vergleich von beispielsweise `Double` und `Int` definiert ist.

Übrigens:

Genauer gesagt kommen die Fehlermeldungen vom sogenannten *Type-Checker*, der ein Teil des Compilers ist. Der Einfachheit halber sprechen wir hier immer vom Compiler. Der Compiler selbst besteht aus einer Reihe von Komponenten, die miteinander verbunden sind (eine sogenannte Pipeline). Jede Komponente hat eine bestimmte Aufgabe und meldet ihre eigenen Fehler. Der Type-Checker ist eine davon.

In einigen Programmiersprachen würden die obigen Ausdrücke übrigens kompilieren. Es wäre zum Beispiel denkbar, dass der Compiler den Ausdruck `3.5 == 1` zu `3.5 == 1.0` umwandelt. So findet ein Vergleich von zwei `Double`-Werten statt. Es gibt aber auch Sprachen, die zwar die obigen Vergleiche erlauben, aber nicht zu einem `Boolean` auswerten. Stattdessen werden die Ausdrücke zu einem undefinierten Wert oder Ähnlichem ausgewertet.

Die gleichen Typprüfungen finden natürlich auch bei Funktionsaufrufen statt. Schauen wir uns dazu ein paar Aufrufe von `round` an:

```
fun main() {
    round(1)
    round(false)
    round("Hallo")
}
```

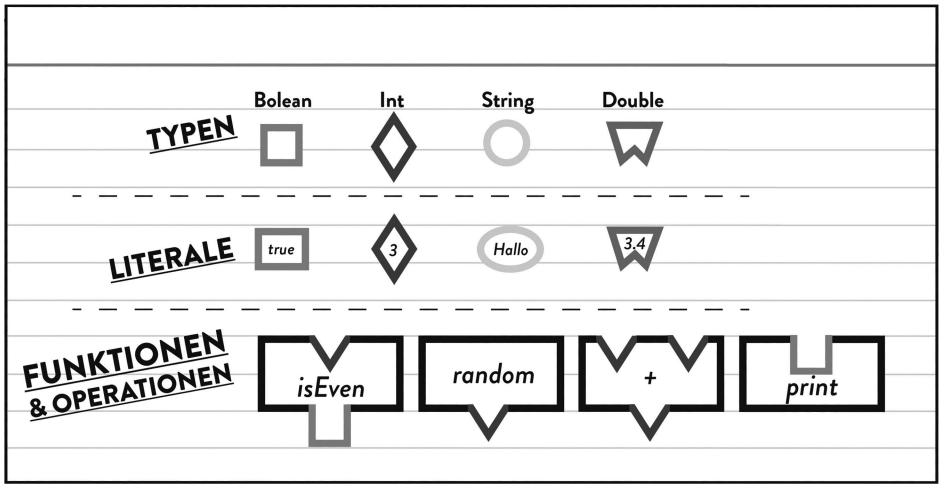
Auch das Programm kompiliert nicht. Die Fehlermeldung des ersten Ausdrucks ist: *„The integer literal does not conform to the expected type Double“*. Die anderen Ausdrücke haben entsprechend andere Fehlermeldungen. Die `round`-Funktion akzeptiert ausschließlich Werte vom Typ `Double` oder `Float`. Das haben die Autoren der `round`-Funktion so festgelegt. Wie soll denn auch eine Ganzzahl gerundet werden? Geschweige denn ein `Boolean` oder ein `String`? Der Compiler schützt uns vor Fehlern. Ein gutes Typsystem sorgt dafür, dass wir ausdrucksstark und gleichzeitig fehlerfrei (hinsichtlich der Typen) programmieren können.

Achtung:

Ein Programm kann erst ausgeführt werden, wenn es erfolgreich kompiliert. Dafür reicht es nicht, dass das Programm syntaktisch fehlerfrei ist. Zudem muss auch die Typkompatibilität gegeben sein. Das ist wichtig, denn der Compiler verspricht, dass alle Ausdrücke bei einem erfolgreich getypten Programm fehlerfrei evaluieren.

8.3.2 Typen als Bausteine

Stellen Sie sich jeden Typen wie einen Baustein mit einer einzigartigen Form vor. Auf Grundlage dessen gibt es für alles, was mit Typen interagiert, entsprechende Bausteine. Es gibt verschiedene Bausteine für Funktionen und Operationen, die so geformt sind, dass sie einen bestimmten Typ akzeptieren und einen bestimmten Typ zurückgeben. Zudem gibt es Bausteine für diverse Literale. Schauen wir uns einmal ein paar ausgewählte Bausteine an:

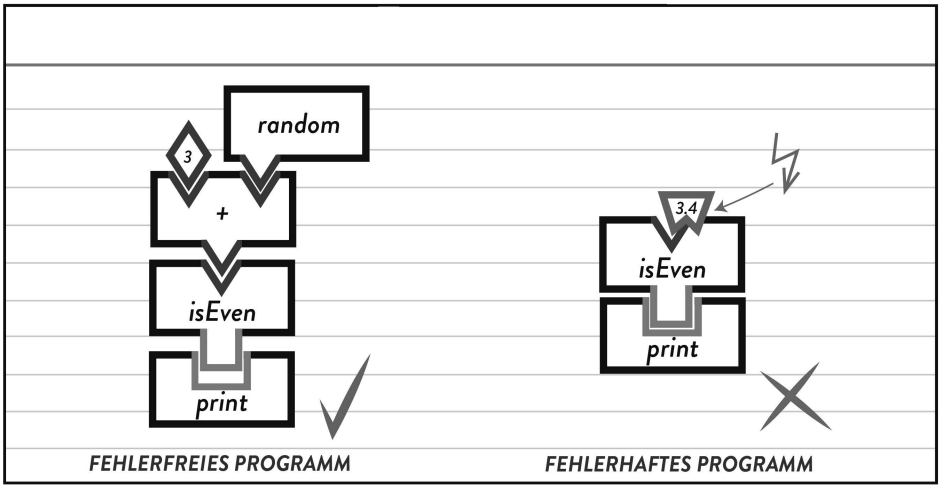


Die Bausteine lassen sich nur zusammenstecken, wenn die Typen passen. Genau das ist die Aufgabe des Typsystems. Der Compiler überprüft, ob die angewendeten Ausdrücke valide sind, indem die Typeigenschaften der im Ausdruck verwendeten Funktionen, Operationen und Werte auf Kompatibilität überprüft werden.

Achtung:

Das ist eine sehr vereinfachte Form, wo jede Funktion einen konkreten Typen erwartet. Kotlin und andere Programmiersprachen verfügen über das Konzept von *polymorphen* Typen und Funktionen. Hierbei ist ein Typ zu mehreren Typen konform, und eine Funktion kann mit unterschiedlichen Typen aufgerufen werden. Ein Beispiel hierfür ist der +-Operator. Dieser kann Werte von unterschiedlichen Typen wie `Int`, `String`, `Double` und viele weitere addieren.

Schauen wir uns jetzt zwei kleine Programme an, wo die Bausteine aufeinander angewendet werden, sprich wo mit den Bausteinen programmiert wird. Das linke Programm kompiliert fehlerfrei, während das rechte einen Typfehler hat:



Sie können sich gut vorstellen, dass bestimmte Bausteine nicht aufeinander passen, so wie es auch im echten Leben ist. Der `isEven`-Baustein akzeptiert ausschließlich einen Baustein vom Typ `Int`. Beim linken Programm wird das Ergebnis des +-Operators direkt an die `isEven`-Funktion übergeben. Diese Bausteine können zusammengesteckt werden, weil sie kompatibel sind. Anders beim rechten Programm: Dort wird ein Baustein vom Typ `Double` auf einen Baustein gesteckt, der ausschließlich Bausteine vom Typ `Int` erlaubt. Das funktioniert nicht.

Auf Grundlage unserer Erkenntnisse können wir Typen folgendermaßen definieren.

Definition *Typ*:

Der Typ legt fest, welche Werte erlaubt und welche Funktionen und Operationen mit Werten des Typs möglich sind. Der Typ wird durch den Compiler ermittelt. Der Compiler überprüft bei jedem Ausdruck, ob dieser mit den Typregeln kompatibel ist.

Da wir nun wissen, wie die Begriffe Ausdruck, Wert und Typ zusammenhängen, schauen wir uns eine nicht vollständige Aufzählung davon an, was alles ein Ausdruck in Kotlin ist. Im Laufe des Buches werden Sie alle Punkte kennengelernt haben.

- Ein Literal wie `1`, `1.5`, `true` oder `"Hallo"`
- Das Auslesen einer Variable
- Der Aufruf und das Ergebnis einer Funktion
- Der Aufruf eines Operators mit seinen Operanden
- Die Verzweigung zu einem bestimmten Wert mithilfe eines if- oder when-Ausdrucks
- Das Erzeugen eines Objekts
- Eine beliebige Kombination aus all diesen Ausdrücken

Der letzte Satz ist besonders wichtig. Ausdrücke sind eine *rekursive Struktur*. Das bedeutet, dass Ausdrücke beliebig ineinander verschachtelt werden können. Ein Ausdruck kann beispielsweise aus mehreren Ausdrücken bestehen. Das haben wir bei Ausdrücken wie `sqrt(8.0)== pow(8.0, 1.0 / 2.0)` gesehen. Je nach Zählweise kann man bei diesem Beispiel bis zu 8 einzelne Ausdrücke identifizieren. Damit wir als Programmierer unter einem Ausdruck das Gleiche verstehen wie ein Computer, folgt die Evaluation von Ausdrücken bestimmten Regeln.

14.3 Funktionen und Methoden

Wir haben nun sehr ausführlich gesehen, wie sich Objekte initialisieren lassen. In Kotlin ist dies sehr wichtig, da hiermit der Zustand von Objekten zu Beginn festgelegt wird. Wir haben bereits gesehen, dass es oft besser ist, Werte nicht mehr zu verändern. Daher kommt es in Kotlin oft vor, dass Objekte erzeugt werden, deren Zustand sich nicht mehr verändert. Der Zustand eines Objekts kann sich jedoch auch verändern, wenn einzelne Eigenschaften als `var` definiert sind.

Das Verändern des Objektzustands geschieht in Funktionen, die mit den Objekten arbeiten. Wir werden sehen, wie sich Funktionen direkt für Klassen definieren lassen.

14.3.1 Objekte als Parameter

Wir haben viele Möglichkeiten kennengelernt, Gläser als Objekte derselben Klasse zu erzeugen. Auch eine Flasche können wir als weitere Klasse abbilden:

```
class Glass(var content: Int, val capacity: Int)
class Bottle(var content: Int)
```

Wir können nun auch ein Objekt der Klasse `Bottle` erzeugen:

```
fun main() {
    val bottle = Bottle(800)
}
```

Die Variablen `glass1`, `glass2` und `bottle` enthalten Referenzen auf verschiedene Objekte. Diese Referenzen können als Argumente an eine Funktion übergeben werden. Diese Funktion erhält die Referenz als Parameter. Diese Parameter sind unveränderliche Variablen. Allerdings können diese konstanten Referenzen auf Objekte verweisen, die sich sehr wohl ändern lassen.

Wir können nun also eine Funktion schreiben, in der wir die Flüssigkeit von einem `Bottle`-Objekt in ein `Glass`-Objekt umfüllen:

```
fun calculateAmount(contentBottle: Int, contentGlass: Int, maxGlass: Int): Int {
    val capacity = maxGlass - contentGlass
    return if (contentBottle >= capacity) capacity else contentBottle
}

fun decant(bottle: Bottle, glass: Glass) {
    val amount = calculateAmount(bottle.content, glass.content, glass.capacity)

    bottle.content -= amount
    glass.content += amount
}
```

Der Vorteil von Funktionen ist, dass sie mehrfach wiederverwendet und mit unterschiedlichen Argumenten aufgerufen werden können:

```
fun main() {
    decant(bottle, glass1)
    decant(bottle, glass2)
}
```

14.3.2 Methoden: Funktionen auf Objekten ausführen

Neben Eigenschaften lassen sich auch Funktionen für eine Klasse festlegen. Klassen sind nicht nur zum Kapseln von Eigenschaften geeignet, sondern auch zum Kapseln von Funktionen. Im Sinne der Objektorientierung sollten wir Funktionen, die sich auf bestimmte Klassen beziehen, auch zum Teil dieser Klasse werden lassen.

Definition *Methoden*:

Die Funktionen einer Klasse nennt man Methoden. Es handelt sich dabei um Funktionen, die mit den Zustandsdaten eines Objekts arbeiten. Im Sinne der Objektorientierung sollten wir Funktionen, die sich auf bestimmte Klassen beziehen, auch zum Teil dieser Klasse werden lassen.

Man kann sich das so vorstellen: Eine Klasse haben wir als Formularvorlage angesehen. Nun legen wir zusätzlich noch Gebrauchsanleitungen fest, was man mit Objekten dieser Klasse anstellen kann. Die Gebrauchsanleitung müssen wir nur einmal festlegen, aber das spezifische Verhalten ändert sich, je nachdem, auf welches Objekt wir sie anwenden. Auf dieses Objekt kann innerhalb der Funktion über einen Parameter `this` zugegriffen werden, der automatisch in dieser Funktion zur Verfügung steht. Durch diesen impliziten `this`-Parameter wird eine Funktion zu einer Methode für die Objekte einer Klasse (auch Mitgliedsfunktion oder Member-Function genannt).

Nehmen wir an, wir wollen eine Gebrauchsanleitung festlegen, wie man die Daten auf dem ausgefüllten Formular (also die Zustandsdaten eines Objekts) vorlesen soll. Dazu könnten wir in der Klasse folgende Funktion festlegen:

```
class Glass(var content: Int, val capacity: Int) {
    var drinkName = "Wasser"

    fun readOut() {
        println("Es sind noch $content ml. $drinkName im Glas.")
        println("Die maximale Füllmenge sind $capacity ml.")
    }
}
```

Beachten Sie, dass diese Funktion im Klassenrumpf festgelegt wurde. Diese Anleitung gehört also fest zur Klasse. Daher kann man sie auch für Objekte der Klasse `Glass` aufrufen. Man sagt auch, dass die Funktion `readOut()` auf `Glass` definiert ist.

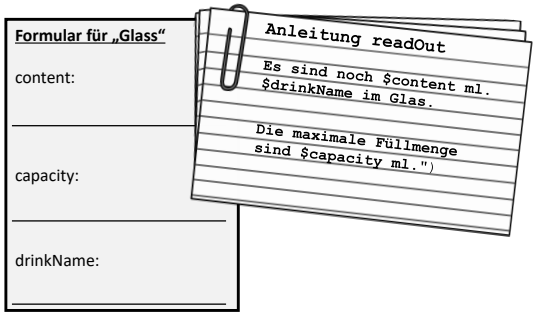
Der Aufruf `glass1.readOut()` erzeugt die Ausgabe:

```
> Es sind noch 40 ml. Cola im Glas.
> Die maximale Füllmenge sind 150 ml.
```

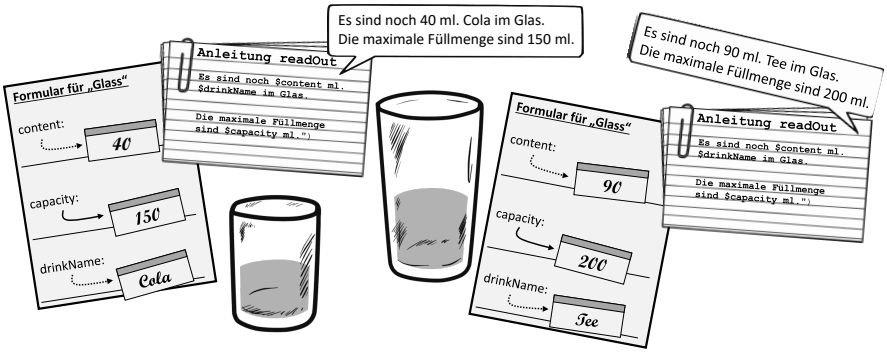
Der Aufruf `glass2.readOut()` erzeugt die Ausgabe:

```
> Es sind noch 90 ml. Tee im Glas.
> Die maximale Füllmenge sind 200 ml.
```

Die Ausgabe hängt also von den Eigenschaftswerten des Glases ab, für das die Methode aufgerufen wird. Man kann sich dies so vorstellen. Sie haben einmalig für die Klasse die Gebrauchsanleitung für das Vorlesen festgelegt:



Wenn man diese Anleitung nun auf verschieden ausgefüllte Formulare anwendet, dann wird Unterschiedliches vorgelesen:



Dieselbe Anleitung für unterschiedliche Objektexemplare führt zu unterschiedlichen Ausgaben. Wir haben also gesehen: Die Klasse legt die Eigenschaften und Methoden allgemein fest. Sie ist so etwas wie der Bauplan für konkrete Objekte. Wenn wir ein konkretes Exemplar einer Klasse erzeugen (oft als *Instanz* oder engl. *instance* bezeichnet), dann hat dieses Objekt konkrete Eigenschaftswerte. Die Eigenschaftswerte bestimmen den Zustand des Objekts. Jedes Objekt kann unterschiedliche Eigenschaftswerte haben, aber die Objekte einer Klasse haben die gleiche Eigenschaftsstruktur.

§

Definition *Mitglieder*:

Eine Klasse hat verschiedene Bestandteile. Dazu gehören Eigenschaften und Methoden. Diese werden auch Mitglieder (Member) der Klasse genannt. Die Klasse definiert, welche Eigenschaften und Methoden zur Verfügung stehen. Die Klasse legt auch fest, welchen Typ die Eigenschaften haben und welche Signatur die Methoden besitzen. Die konkrete Ausprägung einer Eigenschaft mit Datenwerten geschieht erst in den Objekten einer Klasse, d. h. in den konkreten Instanzen. ■

14.3.3 Von Funktionen zu Methoden

Bei der Klasse `Glass` haben wir bereits zusammengehörende Werte (`content` und `capacity`) zu einer Einheit zusammengefasst. Jetzt wollen wir auch noch die Funktionen, die aufgrund ihrer Semantik (also aufgrund ihrer Bedeutung) immer in Zusammenhang mit Objekten einer Klasse stehen, als Funktionen dieser Klasse festlegen. Nehmen wir einmal an, Sie möchten eine weitere Funktion definieren, mit der sich der zur Verfügung stehende Inhalt für ein Glas berechnen lässt. Damit diese Funktion für alle Gläser funktioniert, legen Sie für die Funktion ein bestimmtes Glas als Parameter fest:

```
fun remainingCapacity(glass: Glass): Int {
    return glass.capacity - glass.content
}
```

Die Funktion ruft man so auf:

```
fun main() {
    val glass1 = Glass(100, 200)
    val glass2 = Glass(50, 300)

    val remainingCapacity1 = remainingCapacity(glass1)
    val remainingCapacity2 = remainingCapacity(glass2)
}
```

Die Funktion `remainingCapacity` bezieht sich offensichtlich immer auf `Glass`-Objekte und niemals auf andere Objekte. Daher ist es sinnvoll, diese Funktion gleich zum Teil dieser Klasse werden zu lassen:

```
class Glass(var content: Int, val capacity: Int) {
    fun remainingCapacity(): Int {
        return this.capacity - this.content
    }
}
```

Die wesentliche Änderung und Vereinfachung ist, dass wir keinen Parameter mehr übergeben müssen. Denn der Aufruf der Methode bezieht sich bereits immer auf ein bestimmtes Objekt. Auf dieses Objekt lässt sich mit `this` zugreifen. Die Methode `remainingCapacity()` nimmt also von diesem Glas (`this`) den maximalen Inhalt und zieht von diesem Glas den aktuellen Inhalt ab.

Dieses Glas ist das Glas, für das wir die Methode aufrufen:

```
fun main() {
    // ...

    val remainingCapacity1 = glass1.remainingCapacity()
    val remainingCapacity2 = glass2.remainingCapacity()
}
```

Beim ersten Aufruf bezieht sich `this` also auf `glass1`, und beim zweiten Aufruf bezieht sich `this` auf `glass2`.

Vielleicht wundern Sie sich, was der Nutzen ist. Vorher haben wir der `remainingCapacity`-Funktion das betreffende `Glass` als Parameter übergeben. Jetzt rufen wir für das betreffende `Glass` die Funktion der Klasse auf.

Ein großer Vorteil ist, dass wiederum zusammengehörende Dinge als eine Einheit zusammengefasst werden. Die Funktion `remainingCapacity()` ist nun zum Teil der Klasse `Glass` geworden. So sehen Sie auch immer gleich, welche Funktionen für jedes `Glass` verfügbar sind. Die IDE (z. B. IntelliJ) unterstützt Sie sogar dabei, indem für ein Objekt einer Klasse eingeblendet wird, welche Methoden für dieses Objekt aufgerufen werden können.

Das Definieren von Methoden hat zudem den Vorteil, dass man sich diese Funktionen wie Benachrichtigungen oder Anfragen an ein Objekt vorstellen kann.

Der Aufruf `glass1.remainingCapacity()` bedeutet in etwa: „Hey Glas Nr. 1, wie viel Kapazität ist bei dir noch frei?“.

Methoden sind Funktionen einer Klasse!

Methoden ermöglichen es, eine Nachricht oder Anfrage an ein Objekt einer Klasse zu senden. Die Bearbeitung oder Beantwortung dieser Anfrage ist als Funktionalität in der Klassendefinition festgelegt. Die Anweisungen innerhalb einer Methode beziehen sich immer auf das Objekt, für das die Methode aufgerufen wurde.

Da Methoden sich ebenso wie Eigenschaften auf ein Objekt beziehen, bezieht sich auch der Zugriff auf Eigenschaften in den Methoden auf dieses Objekt.

Die Aufrufe `glass1.remainingCapacity()` und `glass2.remainingCapacity()` führen also zu unterschiedlichen Ergebnissen, wenn die Gläser unterschiedlich gefüllt sind.

Wenn eindeutig ist, dass die Eigenschaft der Klasse gemeint ist, kann das Keyword `this` weggelassen werden:

```
class Glass(var content: Int, val capacity: Int) {
    fun remainingCapacity(): Int {
        return capacity - content // Es geht auch ohne this
    }
}
```

Dies ist nicht der Fall, wenn ein Parameter einer Methode den gleichen Namen hat wie die Eigenschaft des Objekts. In Kotlin gibt es Gültigkeitsbereiche für die Namen von Eigenschaften, Methoden und Klassen. Diese Gültigkeitsbereiche nennt man auch Scope. Wenn ein neuer Name lokal festgelegt wird, überdeckt dieser eventuell einen Namen, der in einem äußeren Gültigkeitsbereich definiert wurde. Dies ist etwa der Fall, wenn die Eigenschaft der Klasse (allgemeinerer Namensraum) genauso heißt wie ein Parameter einer Methode (lokaler Namensraum). Dies hatten wir bereits in Abschnitt 12.8, „Shadowing von Variablen“, kennengelernt.

In diesem Fall muss mit `this` explizit gekennzeichnet werden, dass die Eigenschaft des Objekts und nicht der Parameter referenziert werden soll. Dadurch kann der Compiler zwischen der Eigenschaft der Klasse (allgemeinerer Namensraum) und dem Parameter (lokaler Namensraum) unterscheiden.

Ein Beispiel dafür sehen Sie hier:

```
class Glass(var content(1): Int, var capacity(3): Int) {
    fun add(content(2): Int) {
        val calculatedContent = this.content(1) + content(2)
        this.content(1) = if (calculatedContent <= capacity(3)) {
            calculatedContent
        } else {
            println("Das Glas ist voll")
            capacity(3)
        }
    }
}
```

Das Schlüsselwort `this` wird auch noch in anderen Situationen zwingend benötigt, z. B. wenn eine Methode eine Referenz auf das Objekt selbst zurückgeben oder als Argument weitergeben möchte. Hier ein Beispiel:

```
class Glass(var content: Int, val capacity: Int) {
    fun biggerGlass(other: Glass): Glass {
        return if (this.capacity >= other.capacity) {
            this
        } else {
            other
        }
    }
}
```

Die Methode `biggerGlas` vergleicht das `Glass`, für das die Methode aufgerufen wird, mit einem anderen `Glass`. In der `if`-Bedingung sieht man zunächst, warum die Verwendung von `this` mehr als guter Stil ist. Es wird zweimal auf `capacity` zugegriffen: einmal für dieses Glas (`this`) und einmal für das andere Glas (`other`). Je nachdem, wo mehr enthalten ist, soll entweder dieses (`this`) oder das andere (`other`) Glas zurückgegeben werden. Das aktuelle Glas können wir nur über `this` referenzieren.

`this` ist ein impliziter Parameter, der uns beim Aufruf von Methoden eines Objekts zur Verfügung steht. `other` ist ein von uns eingeführter Parameter, der beim Aufruf der Methode explizit angegeben werden muss.

Mit folgender Anweisung können Sie der Variablen `bigGlass` das größere Glas zuweisen:

```
fun main() {
    // ...

    val bigGlass = glass1.biggerGlass(glass2)
}
```

14.4 Datenkapselung

Eventuell ist Ihnen schon aufgefallen, dass die Implementierung der `Glass`-Klasse noch Fehlerquellen birgt. Eine davon ist zum Beispiel, dass zwar bei der `add`-Methode und im Konstruktor überprüft wird, ob die `capacity` überschritten wurde, es jedoch immer noch möglich ist, über andere Wege die `capacity` zu überschreiten.

Android diese Activity beim Öffnen der App startet, muss diese noch als Launcher-Activity konfiguriert werden. Dazu fügen Sie der Activity ein `-intentfilter`-Element mit einen `action`- und einem `categorie`- Element hinzu. Ihr Manifest sollte anschließend so aussehen:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.programmierenlernenmitkotlin.moviemaker">

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">

        <activity android:name=".StartActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Nun können Sie die App, wie am Anfang dieses Kapitels beschrieben, mit dem Emulator oder Ihrem eigenen Gerät starten und ausprobieren. Klicken Sie dazu auf den grünen Play-Button oben rechts in der Ecke. Sie sollten dann die von Ihnen erstellte StartActivity sehen. Damit dürfen wir Ihnen gratulieren, denn Sie haben soeben Ihre erste lauffähige App geschrieben. Falls Sie sich für die „Hello World"-Variante entschieden und sofort durchgestartet haben, sollten Sie nun noch einmal zu Abschnitt 33.2.1, „Anlegen des Layouts", zurückkehren.

33.3 Persistenz

Bevor Sie in der StartActivity das Geld des Spielers anzeigen können, sollten wir klären, wie Sie an diese Information kommen. Außerdem soll in der App das Budget des Spiels so gespeichert werden, dass es auch nach dem Beenden der App nicht verloren geht. Deshalb werden Sie nun etwas über die Persistierung von Daten in einer Android-App lernen. Wenn die Daten einfach nur in einem Objekt als Attribut gehalten werden, wie es aktuell der Fall ist, sind die Daten spätestens nach dem Beenden der App weg. Deswegen sollte man sich in Android, genauso wie in allen anderen Systemen, Gedanken darüber machen, wie Daten länger gespeichert werden können. Immerhin möchten Sie das Spiel fortsetzen und nicht jedes Mal von vorne beginnen.

In Android gibt es mehrere Möglichkeiten, Daten abzuspeichern. Die wohl verbreitetsten sind *Shared Preferences* und *SQLite*. Shared Preferences sind ein sogenannter *Key-Value Store*, in dem einfache Werte persistiert werden. Solche sind z. B. Einstellungen, die der Nutzer in der App vorgenommen hat. Key-Value bedeutet, dass jeder Wert (Value) einem Schlüssel (Key) zugeordnet wird. Das kommt Ihnen vielleicht bekannt vor, denn dieses Konzept haben Sie schon in Abschnitt 24.5, „Maps", kennengelernt. In diesem Fall ist der Schlüssel ein `String`, während ein Wert alles Beliebige sein und sich jederzeit ändern kann.

Über den Schlüssel werden die Daten anschließend auch abgefragt. Stellen Sie sich das einfach wie eine Tabelle mit zwei Spalten vor. In der linken Spalte tragen Sie den Key ein und in der rechten den dazugehörigen Wert. Wenn Sie also Informationen eines Spielers abspeichern wollten, würde das in etwa so aussehen:

Key	Value
playerName	Max Mustermann
playerMoney	1000000

Das Entscheidende ist, dass der Schlüssel immer gleich bleibt, während sich der Wert (z. B. *playerMoney*) jederzeit ändern kann.

Während diese Art der Datenspeicherung für einfache Daten oft ausreicht, ist sie für komplexere Daten nicht besonders geeignet. Die Tabelle würde beispielsweise so aussehen, wenn die App mehrere Spieler unterstützt:

Key	Value
player1Name	Max Mustermann
player1Money	1000000
player2Name	Marie Mustermann
player2Money	700000

Über eine solche Datenstruktur lassen sich komplexe Objektstrukturen, die Sie in diesem Buch kennengelernt haben, nicht besonders gut abbilden. Dafür ist eine Datenbank wie SQLite wesentlich besser geeignet. Diese können Sie sich auch wie eine Tabelle vorstellen, wo allerdings die Anzahl der Werte, die zu einem Key gehören, nicht limitiert sind:

ID	Name	Money
1	Max Mustermann	1000000
2	Marie Mustermann	700000

Das Arbeiten mit einer solchen Datenbank ist jedoch komplizierter. Die Unterscheidung sowie Nutzung von Datenbanken füllt ganze Bücher. Sie müssten lernen, wie man in Android Tabellen definiert, wie man Anfragen an diese Tabellen stellt und vieles mehr. Shared Preferences hingegen sind wesentlich einfacher zu benutzen und reichen für die App vollkommen aus. Aus diesem Grund werden wir hier nur darauf zurückgreifen. Sie werden jedoch in diesem Kapitel die Datenspeicherung so abkapseln, dass Sie diese später ganz einfach austauschen können.

33.3.1 Definition der Schnittstelle

Dazu legen Sie im Ordner *app/java/ihr/paket/name/moviemaker* eine neue Datei namens *GameDatabase.kt* an. In dieser Datei werden Sie zunächst ein Interface als Schnittstelle zur Datenbank erstellen:

```
interface GameDatabase {
    fun getAllGenres(): List<Genre>
    fun getAllActors(): List<Actor>
    fun getAllDirectors(): List<Director>
    fun getMoney(): Int
    fun saveMoney(budget: Int)
    fun getImageFor(person: Person): Drawable?
    fun getImageFor(genre: Genre): Drawable?
}
```

Die Klassen `Genre`, `Actor`, `Director` und `Person` importieren Sie aus dem *core*-Paket, das Sie zu Beginn dieses Kapitels erstellt haben. Die Klasse `Drawable` kommt aus der Android-Bibliothek und repräsentiert in diesem Fall ein Bild. Diese brauchen Sie hier, da in der App im Gegensatz zum Konsolenprogramm auch Bilder zu den einzelnen Personen und Genres angezeigt werden.

33.3.2 Implementierung mit Shared Preferences

Erstellen Sie in der gleichen Datei eine Klasse namens `SharedPreferencesDatabase`, die das `GameDatabase`-Interface implementiert. In dieser Implementierung der `GameDatabase` nutzen Sie die Shared Preferences von Android, um die Daten zu speichern und zu lesen. Wenn Sie dieses Verhalten austauschen wollen und stattdessen eine Datenbank verwenden möchten, können Sie einfach eine weitere Implementierung von `GameDatabase` erstellen und diese nutzen. Welche Implementierung Sie nutzen, wird keine weiteren Auswirkungen auf den Programmcode Ihrer App haben. Das ist der Vorteil von Interfaces.

Für den Zugriff auf die Shared Preferences benötigen Sie den `Context`, von dem aus Sie auf die Daten zugreifen. Diesen nehmen Sie im Konstruktor entgegen. Anschließend implementieren Sie die Methoden des Interfaces. Die `Genres`, `Actors` und `Directors` beziehen Sie einfach aus dem `GameData`-Objekt, wie Sie es auch zuvor getan haben. Diese Daten werden nicht gespeichert, sondern bei jedem App-Start vom `GameData`-Objekt neu erzeugt:

```
override fun getAllGenres() = GameData.genres
override fun getAllActors() = GameData.actors
override fun getAllDirectors() = GameData.directors
```

Das Geld des Spielers soll persistiert werden. Dafür müssen Sie sich überlegen, wie die Datei heißen soll, in der die Werte gespeichert werden. Zudem müssen Sie einen Key vom Typ `String` definieren. Am besten legen Sie dazu ein `companion object` mit zwei Konstanten für diese Werte an, sodass Sie anschließend von den beiden Methoden darauf zugreifen können:

```
companion object {
    private const val PREFERENCES_NAME = "de.moviemaker.gamedata"
    private const val MONEY_KEY = "money"
}
```

Nun können Sie die Methoden implementieren. Dazu rufen Sie in der `getMoney`-Methode die `getSharedPreferences`-Methode auf, die auf allen Objekten vom Typ `Context` definiert ist. Dieser Methode übergeben Sie den Namen der Datei. Außerdem geben Sie den Bearbeitungsmodus an. Dieser sollte `Context.MODE_PRIVATE` sein. Als Ergebnis bekommen Sie eine

Instanz der Klasse `SharedPreferences` zurück. Auf dieser können Sie die Methode `getInt` zum Lesen eines `Int`-Objekts aufrufen. Dieser Methode übergeben Sie den Key des Wertes, den Sie lesen möchten. Außerdem nimmt die Methode einen Standardwert entgegen, den Sie zurückbekommen, sofern kein Wert mit dem entsprechenden Key vorhanden ist:

```
override fun getMoney() = context
        .getSharedPreferences(PREFERENCES_NAME, Context.MODE_PRIVATE)
        .getInt(MONEY_KEY, 10_000_000)
```

Das Speichern von Werten erfolgt über den `SharedPreferencesEditor`. Auch hier holen Sie sich zunächst eine Instanz der Klasse `SharedPreferences`. Anschließend fragen Sie mit der `edit()`-Methode den Editor an. Über diesen können Sie mit `putInt` einen `Int`-Wert schreiben. Auch hier müssen Sie natürlich den Key sowie den zu schreibenden Wert übergeben. Um den Schreibprozess zu beenden, rufen Sie zum Schluss noch die `apply()`-Methode auf:

```
override fun saveMoney(money: Int) {
        context.getSharedPreferences(PREFERENCES_NAME, Context.MODE_PRIVATE)
                        .edit()
                        .putInt(MONEY_KEY, money)
                        .apply()
}
```

Nun fehlen nur noch die beiden `getImageFor`-Methoden. Hier soll ein entsprechendes `Drawable` für die jeweilige Person oder das jeweilige Genre zurückgegeben werden. Das Bereitstellen der Ressource übernimmt auch hier wieder Android für Sie. Dennoch müssen Sie Android die ID der Ressource mitteilen. Und die ist wiederum abhängig von dem Wert, der in Ihre Methode übergeben wird. Emma Thompson hat schließlich ein anderes Bild als Susi Sonnenschein. Das Mapping von Name auf ID machen wir in einer when-Anweisung. So könnten Sie bei der Person abhängig vom Namen und beim Genre abhängig vom Aufzählungswert die entsprechenden ID in die `getDrawable`-Methode des Context übergeben, die Ihnen dann das entsprechende Drawable zurückgibt:

```
override fun getImageFor(person: Person) = context.getDrawable(when
     ("${person.firstName} ${person.lastName}") {
    "Emma Thompson" -> R.drawable.person_emma_thompson
    "Susi Sonnenschein" -> R.drawable.person_susi_sonnenschein
    "Hanna Heiter" -> R.drawable.person_hanna_heiter
    "John Goodman" -> R.drawable.person_john_goodman
    "Fridolin Fröhlich" -> R.drawable.person_fridolin_froehlich
    "Steven Spielberg" -> R.drawable.person_steven_spielberg
    "Roland Emmerich" -> R.drawable.person_roland_emmerich
    "Lars Lustig" -> R.drawable.person_lars_lustig
    else -> 0
})

override fun getImageFor(genre: Genre) = context.getDrawable(when (genre) {
    Genre.ACTION -> R.drawable.genre_action
    Genre.ADVENTURE -> R.drawable.genre_adventure
    Genre.COMEDY -> R.drawable.genre_comedy
    Genre.CRIME -> R.drawable.genre_crime
    Genre.DOCUMENTATION -> R.drawable.genre_documentary
    Genre.DRAMA -> R.drawable.genre_drama
    Genre.HORROR -> R.drawable.genre_horror
    Genre.ROMANCE -> R.drawable.romance
    Genre.FANTASY -> R.drawable.genre_fantasy
    Genre.THRILLER -> R.drawable.thriller
})
```

33.3.3 Zentrales Instanziieren mit Extension Functions

Damit ist die Implementierung von `SharedPreferencesDatabase` fertig. Jetzt müssen Sie diese noch an der richtigen Stelle instanziieren. Das machen Sie am besten an einer zentralen Stelle, sodass Sie später nur an einer Stelle die Datenbank austauschen müssen, wenn Sie die Implementierung ändern wollen. Dazu könnten Sie eine *Extension-Property* auf dem `Context` definieren. Das ist auch sinnvoll, da die Datenbank ja abhängig vom Context ist. Alternativ könnten Sie auch eine globale `getDatabase`-Methode erstellen, die den Context entgegen nimmt und die Datenbankimplementierung zurückgibt. Wir nutzen die Extension-Property auf `Context`:

```
val Context.database: GameDatabase
    get() = SharedPreferencesDatabase(this)
```

So können Sie einfach `context.database` aufrufen, um eine Datenbankinstanz zu bekommen. Achten Sie jedoch darauf, dass dieses Property vom Typ `GameDatabase` und nicht `SharedPreferencesDatabase` ist, da Sie diese sonst nicht einfach durch eine andere `GameDatabase`-Implementierung austauschen können.

33.3.4 Zugriff auf die Datenbankinstanz

Nun sind Sie in der Lage, das aktuelle Budget des Spielers in der StartActivity anzuzeigen. Öffnen Sie dazu wieder Ihre StartActivity, wo Sie gleich den Wert des entsprechenden TextViews setzen werden. Um sicherzustellen, dass das TextView immer den aktuellen Wert hat, sollten Sie sich allerdings genau überlegen, zu welchem Zeitpunkt des Lifecycles Sie das TextView befüllen möchten. Wenn Sie dies in der `onCreate`-Methode machen, würde der Wert nur beim Erstellen der Activity gesetzt werden, nicht jedoch, wenn Sie aus einer anderen Activity zu dieser zurückkehren. Das ist normalerweise kein Problem, da der Wert im TextView auch nicht gelöscht wird, wodurch er nach wie vor zu sehen ist. Er wird jedoch ebenso nicht aktualisiert, wenn Sie zur `CreateMovieActivity` navigieren, einen Film erstellen und anschließend zur StartActivity zurückkehren (das machen wir gleich). Und genau in dieser Situation sollte sich das TextView aktualisieren. Deshalb ist hier die `onResume`-Methode die bessere Wahl. Dazu überschreiben Sie einfach die `onResume`-Methode Ihrer StartActivity.

Um nun den Wert auf das `TextView` zu setzen, benötigen Sie eine Referenz auf das `View`, das Android für Sie aus dem Layout erstellt hat. Sie können die Views Ihres Layouts sozusagen einfach mit `import kotlinx.android.synthetic.main.activity_start.*` importieren. Sie müssen natürlich darauf achten, dass Sie die Referenzen des richtigen Layouts hinzufügen. Wenn Sie die Views eines anderen Layouts importieren, wird es zur Laufzeit einen Fehler geben, da Android das entsprechende `View` nicht finden kann. Nun können Sie einfach über die ID des Views, die Sie in der Layout-Datei definiert haben, auf die `View`-Instanz zugreifen. Die Daten die Sie benötigen, können Sie über das von Ihnen im letzten Abschnitt definierte `database`-Attribut des `Context` abrufen. Dieses steht Ihnen auch hier zur Verfügung, da die Activity ja den Context erweitert. Ihre `onResume`-Methode sollte also nun so aussehen:

```
import kotlinx.android.synthetic.main.activity_start.*

override fun onResume() {
    super.onResume()
    current_budget_tv.text = this.database.getMoney().toString()
}
```

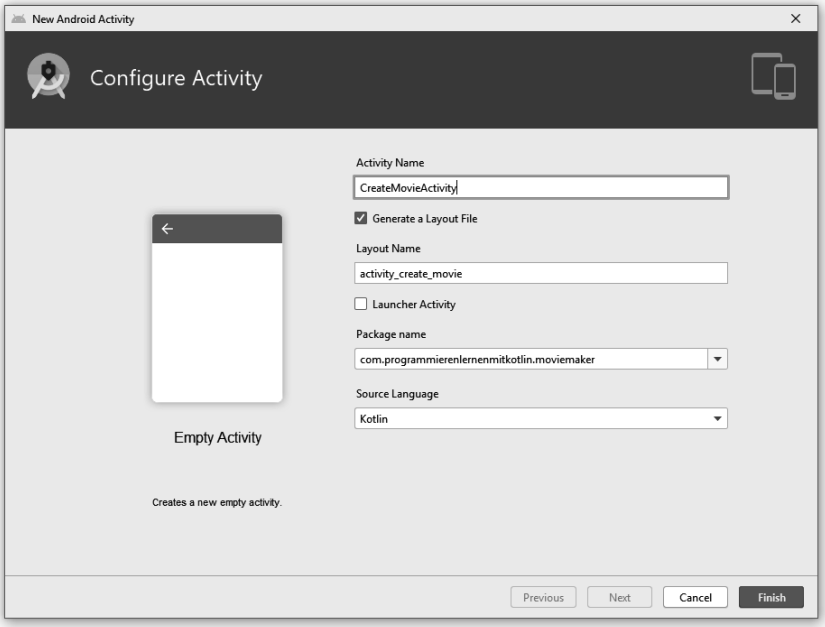
Wenn Sie jetzt Ihre App nochmals starten, sollten Sie den Standardwert im TextView sehen, den Sie in `SharedPreferencesDatabase` angegeben haben. Beim ersten Aufruf steht natürlich noch kein Wert in den Shared Preferences.

33.4 CreateMovieActivity erstellen (automatisiert)

Die StartActivity ist soweit fertig gestellt. Nun ist es an der Zeit die zweite Activity zu implementieren, die `CreateMovieActivity`, wo der Spieler die Produktion des Films zuerst plant und danach startet. Diesmal werden Sie sich allerdings beim Erstellen der Activity von Android Studio helfen lassen.

33.4.1 Erstellen der Activity

Dazu machen Sie einen Rechtsklick auf den Ordner, in dem sich auch die StartActivity befindet. Wählen Sie dann *New -> Activity -> Empty Activity*. Diese nennen Sie `CreateMovieActivity` und bestätigen die Auswahl:



Android Studio hat soeben die Activity mit dem Namen `CreateMovieActivity` für Sie erstellt, eine passende Layout-Datei mit dem Namen `activity_create_movie.xml` für Sie angelegt und die Activity im Manifest registriert.

Stichwortverzeichnis