## 3.11 Exceptions und Errors

Dart kennt und nutzt zwei offizielle Fehlertypen: *Exceptions* und *Errors*. Das kann auf den ersten Blick verwirren. Wir sehen uns an, in welchem Fall jeder Typ zu benutzen ist. Außerdem führt dieser Abschnitt zur Fehlerbehandlung in Dart das `try-catch`-Konstrukt ein.



### 3.11.1 Exceptions

Eine *Exception* ist aus vielen anderen Programmiersprachen als Konzept bereits bekannt: Tritt ein Ausnahmefehler auf, so wird er entweder am Auftrittspunkt behandelt oder weiter nach oben »geworfen«, also zum Aufrufer des jeweiligen Auftrittspunkts. Dieses Weiterwerfen kann sich bis hin zur Hauptprogrammschleife durchziehen. Wird eine Exception nicht behandelt, so bricht das Programm ab.

Eine Exception lässt sich in Dart mit dem `try-catch`-Konstrukt »fangen« und im `catch`-Block behandeln:

```
try {
  // Der übergebene String ist nicht
  // in eine Zahl verwandelbar. Eine
  // FormatException ist die Folge.
  final number = int.parse('abc');
} catch (e) {
  // Wir haben die Exception gefangen
  print(e);
}
```

**Listing 3.166** Mit »try-catch« eine Exception fangen

Im `catch`-Block behandeln wir die Exception, indem wir sie ausgeben.



**Exceptions schlucken**

Das »Schlucken« einer Exception – also sie zu fangen und anschließend unbehandelt zu lassen (mit einem leeren `catch`-Rumpf) – ist nicht zu empfehlen: Nur allzu leicht ließe sich eine Fehlerursache hierdurch aus den Augen verlieren, was eine unnötige Fehlersuche zur Folge hat.

### 3.11.2 Errors

Eingangs habe ich neben Exceptions auch *Errors* erwähnt. Bisher haben Sie nur die Exception gesehen. Errors sollen in Dart einen Fehler des Programmierers verdeutlichen, beispielsweise wenn eine (dynamisch aufgelöste) Methode mit den falschen

Parametern aufgerufen wird oder auf einem Objekt etwas ausgeführt werden soll, das ihm gar nicht bekannt ist. Errors grenzen sich damit von den Exceptions ab, die Probleme aufzeigen, mit denen zu rechnen war, etwa (wie zuvor zu sehen) wenn eine Zeichenkette nicht in eine Zahl verwandelt werden kann. Ein Error wird zum Beispiel geworfen, wenn wir auf das erste Element einer leeren `List` zugreifen wollen:

```
try {
  // Wir erzeugen eine leere Liste,
  // aber greifen auf den ersten Index zu.
  // Ein RangeError ist die Folge.
  final list = [];
  final element = list[0];
} catch(e) {
  // Wir haben den Error gefangen.
  print(e);
}
```

**Listing 3.167** Mit »try-catch« einen Error fangen

Wir übergeben dem Operator `[]`der `List` einen ungültigen Index. Weil dieser das Argument einer Methode ist (vergleiche Abschnitt 3.6, ein Operator ist eine Funktion) und an dieser Stelle ungültig ist, wird hier ein `RangeError` geworfen, der sich von `ArgumentError` ableitet.

Ein syntaktischer Unterschied zwischen Exception und Error besteht darin, dass ein Error als Basisklasse vererbt, eine Exception hingegen nur als Interface implementiert werden kann. Beispielimplementierungen wären:

```
// Ein Error wird geerbt.
class CustomError extends Error {}

// Eine Exception wird hingegen implementiert (Interface).

class CustomException implements Exception { }
```

**Listing 3.168** Implementierung von Errors und Exceptions

Beim ersten Auftreten eines Errors wird ihm automatisch von Dart ein *Stacktrace* mitgegeben, da der Getter `stacktrace` des Error-Objekts gesetzt wird. Dieses Konstrukt leitet Sie durch die Aufrufe von Funktionen bis hin zum Auslöser eines Fehlers. Ein Beispiel wäre etwa:

```
Unhandled exception:

#0      main (main.dart:27:2)
```





```
#1      _delayEntrypointInvocation.<anonymous closure> (dart:isolate-patch/
isolate_patch.dart:283:19)
#2      _RawReceivePortImpl._handleMessage (dart:isolate-patch/isolate_
patch.dart:184:12)
```

**Listing 3.169** Ein exemplarischer Stacktrace

### 3.11.3 Fehler werfen: throw

Dart kann interessanterweise jegliches Objekt mit dem Schlüsselwort `throw` werfen. Bei `throw` handelt es sich um einen Ausdruck, der überall eingesetzt werden kann und der den jeweiligen Programmfluss an der Stelle abbricht. Wird das geworfene Objekt nicht gefangen, bricht das ganze Programm ab. Auch wenn keine syntaktischen Anforderungen an das zu werfende Objekt gestellt werden, wird von Dart empfohlen, eigene Objekte von `Error` oder `Exception` abzuleiten:

```
// Möglich, aber Dart rät davon ab
throw 'This is not a drill!';
// Empfohlen, liest automatisch
// Informationen wie den StackTrace aus
throw CustomError();

// Eine eigene Exception, um einen
// Ausnahmefall zu signalisieren
throw CustomException();
```

**Listing 3.170** Jedes Dart-Objekt kann geworfen werden.

### 3.11.4 Der zweite Parameter von catch

Bisher haben Sie den `catch`-Block stets mit einem Parameter, dem geworfenen Objekt, gesehen. Er bietet als zweiten Parameter, zur Diagnose, auch den Stacktrace, also den Auftrittsort und den Aufrufpfad dorthin. Wie zuvor behandelt, bekommen Sie beim Fangen eines Errors den Stacktrace im entsprechenden Getter gleich mitgeliefert. Für Exceptions ist dies nicht der Fall. An die Information könnten Sie also durch den zweiten Parameter von `catch` kommen. Ich habe zur Demonstration das Beispiel aus Listing 3.166 zur Einführung der Exceptions erweitert:

```
try {
  // Der übergebene String ist nicht
  // in eine Zahl verwandelbar. Eine
  // FormatException ist die Folge.
  final number = int.parse('abc');
} catch (e, s) { // wir greifen den 2. Parameter (s) ab
```

```
  // Wir haben die Exception gefangen
  print(e);
  // Der Stacktrace aus catch
  print(s);
}
```

**Listing 3.171** Mit dem zweiten Parameter von »catch« den Stacktrace auslesen

### 3.11.5 Spezifische Exceptions oder Errors fangen

Das Abfangen jeglicher Art von auftretender Exception oder auftretendem Error ist mit der Kombination aus `try-catch` kein Problem. Bedarf es einmal einer genaueren Fallunterscheidung, etwa weil Error A anders behandelt werden soll als Fehler B, so steht uns dafür die Kombination `try-on-catch` zur Verfügung. Über `on` mit Angabe der Klasse können Sie einen Pfad speziell für geworfene Objekte eines speziellen Typs erstellen. Die Anzahl der `on`-Pfade ist nicht beschränkt:

```
try {
  // Der übergebene String ist nicht
  // in eine Zahl verwandelbar. Eine
  // FormatException ist die Folge.
  final number = int.parse('abc');
}
on CustomException catch(e) {
  // Anweisungen für CustomException
}
on FormatException catch(e, stacktrace) {
  // Anweisungen für FormatException
}
catch (e, stacktrace) {
  // Alle anderen Errors, Exceptions oder Objekte

  print(e);
  // Der Stacktrace aus catch
  print(stacktrace);
}
```

**Listing 3.172** Mit »on-catch« spezifische Errors oder Exceptions fangen

### 3.11.6 Das Schlüsselwort finally

Sie können ein `try-catch`- zu einem `try-catch-finally`-Konstrukt ausbauen. Auch ist es möglich, nur `try-finally` einzusetzen. Ein `finally`-Block wird immer ausgeführt, wenn etwas gefangen wurde, und auch dann, wenn nicht. Somit ließe sich mit `try-`





`finally` sogar unabhängig von Errors und Exceptions eine Aufräumlogik implementieren:

```
try {
  // etwas ausführen
}
catch (e) {
  // Mögliche Fehler behandeln
}
finally {
  // in jedem Fall (mit oder ohne catch) auszuführen
}
```

**Listing 3.173** Der »finally«-Block

Der Rumpf von `finally` wird übrigens auch ausgeführt, wenn im `try`-Block eine `return`-Anweisung stehen sollte:

```
try {
 // ...
 return;
}
finally {
 // trotzdem ausgeführt
}
```

**Listing 3.174** »finally« wird auch nach »return« ausgeführt.

## 3.12 Asynchrone Programmierung

Moderne Apps sind völlig undenkbar ohne die *asynchrone Programmierung*. Ohne die parallele Verarbeitung von Instruktionen gäbe es keine flüssige Benutzeroberfläche bei gleichzeitiger Bewältigung von rechenintensiven Aufgaben oder solchen mit längeren Wartezeiten, etwa Requests an APIs oder Zugriffe auf den Speicher. Dann hätte wohl niemand Freude an seinem Smartphone, Computer oder Smart-Device.

Dementsprechend selbstverständlich macht Dart es Programmierern außerordentlich einfach, asynchrone Programmierung in die Anwendung einzubauen. In den folgenden Abschnitten werden Sie Futures, die Grundwährung der asynchronen Programmierung, die Schlüsselwörter *async* und *await* sowie die Verarbeitungsschleifen namens *Event Loops* im Zusammenspiel mit *Isolates* kennenlernen.

### 3.12.1 Event Loops

Dart spannt zur Ausführung von Code einen einzigen Thread auf. Darin wird eine sogenannte Event Loop gestartet und empfängt fortwährend Ereignisse (*Events*), die in Warteschlangen (*Queues*) eingereiht werden.

Sollten keine Events vorliegen, benutzt die *Runtime* die freie Kapazität zum Freigeben von Speicher oder Ressourcen durch den *Garbage Collector*.

Einer Event Loop arbeiten zwei Queues zu: die Event Queue und die *Microtask Queue*. Ein *Microtask* ist eine Aufgabe, die nachgelagert, aber mit hoher Priorität noch vor dem nächsten Durchlauf der Event Queue zu verarbeiten ist. Als Events und damit als Elemente der Event Queue gelten unter anderem Eingabe/Ausgabe, Rendering und Timer.

### 3.12.2 Future

Viele Programmiersprachen implementieren das Konzept der *Future*, das manchmal auch *Promise* genannt wird. Es definiert eine Struktur mit Informationen über den asynchronen Vorgang und seinen Status. Außerdem lässt sich je nach Ausgang der Operation der berechnete Wert oder ein Fehler aus einem Future auslesen.

#### Zustände eines Future

Ein Future ist generisch und definiert als `Future<T>`. Der generische Parameter ist der Typ des erwarteten Ergebnisses der Operation. Sie kann drei Zustände einnehmen: nicht abgeschlossen (`uncompleted`), abgeschlossen mit Wert (`completed with value`) und abgeschlossen mit Fehler (`completed with error`).

#### Futures aus APIs

Zumeist werden Sie Futures durch asynchrone Methoden oder Funktionen erhalten, etwa wenn Sie einen Aufruf (*Request*) an eine entfernte API senden und auf die Antwort (*Response*) warten.

Angenommen, Sie fragen eine solche API über einen beliebigen *HTTP*-Client an, der die Methode `get` anbietet; dann bekommen Sie wie folgt ein Objekt des Typs `Future` zurück:

```
Future future = httpClient.get();
```

**Listing 3.175** Rückgabe eines Futures von einer Methode







#### Die Konstruktoren von Future

Außerdem bietet Future eigene Konstruktoren. Möchten Sie zum Beispiel eine Operation erst nach einer spezifischen Zeit ausführen, ließe sich ein entsprechendes Future folgendermaßen erzeugen:

```
final durationToWait = Duration(seconds: 2);
final delayedFuture = Future.delayed(durationToWait);
```

**Listing 3.176** Erzeugen eines Futures durch einen Konstruktor

In Zeile 1 des oberen Beispiels ist eine Dauer angegeben, nach deren Ablauf eine mit dem Future assoziierte Operation durchgeführt wird. Eine solche ist aber hier nicht definiert. Der Code beschränkt sich auf den Verweis der Variablen `delayedFuture` auf das Future-Objekt, das vom Konstruktor `Future.delayed` zurückgegeben wird.

Wenn Sie sich bereits im Kontext eines Futures befinden, zum Beispiel innerhalb des Callbacks von `then`, oder in einer als `async` deklarierten Funktion (0), können Sie über `Future.error` einen Fehler zurückgeben und ein zum `throw-catch`-Konzept analoges Verhalten implementieren. Angenommen, es tritt während der Verarbeitung in einem `then`-Callback ein Fehler auf, so lässt sich der Umstand über den Konstruktor `Future.error` signalisieren und über einen nachgelagerten `catchError`-Callback abhandeln:

```
Future.delayed(durationToWait)
    .then((_) => Future.error('Fehler'))
    .catchError((error) => print(error));
```

**Listing 3.177** Verwendung von »Future.error«

Um eine Funktion oder Methode parallel als *Microtask* (siehe oben) ausführen zu lassen, übergeben Sie diese an `Future.microtask`. Eine so in ein Future eingewickelte Aktion wird in die *Event Loop* (siehe Abschnitt 3.12.1) eingefügt und ausgeführt, sobald sie an der Reihe ist. Währenddessen kann der Code an der Stelle des Aufrufs ungehindert weiterlaufen. Sie könnten beispielsweise eine Logausgabe in der Konsole ausgeben und mithilfe von `Future.microtask` asynchron in die Datenbank oder auf das Dateisystem schreiben lassen, ohne dass die Funktion, während sie eine Logmeldung abarbeitet, verlangsamt oder beeinträchtigt würde:

```
Future.microtask(() => () {
  // Parallel ausgeführte Operation
});
```

**Listing 3.178** Parallele Ausführung mit Microtasks

Müssen Sie ein Objekt des Typs `Future` zurückgeben, lassen sich Berechnungen über die Konstruktoren `Future.sync` oder `Future.value` einwickeln.

### Aktionen nach Abschluss eines Futures ausführen

Wie Sie eingangs erfahren haben, kann ein Objekt des Futures einen von drei möglichen Zuständen einnehmen. Die vorherigen Beispiele interessierten sich dafür noch nicht, sondern zeigten zunächst, wie Sie zu einer Instanz des Typs gelangen.

Anlehnend an das Beispiel in Listing 3.176 befände sich das Future `delayedFuture` nach dem Aufruf des Konstruktors für mindestens zwei Sekunden im Zustand `uncompleted`, denn die zugeordnete Aufgabe ist erst beendet, wenn die angegebene Zeit verstrichen ist. Wäre dem Future beispielsweise ein Request aufgetragen worden, wäre das Objekt ebenfalls so lange in diesem Zustand, bis die Aktion (also das Warten auf den Erhalt von Daten der entfernten API) abgeschlossen wäre.

Wenn Sie etwas nach Abschluss der Aufgabe ausführen möchten, stellen Sie dem Future einen Callback zur Verfügung, der automatisch aufgerufen wird. Das nächste Beispiel demonstriert einen entsprechenden Aufruf an `print`:

```
Future.delayed(durationToWait).then((_) => print('done!'));
```

**Listing 3.179** Einen Callback über »then« bereitstellen

Der Callback ist als *Lambda* (siehe Abschnitt 3.6.7) bereitgestellt und ignoriert den überreichten Parameter, indem dieser nach der Konvention für ignorierte Parameter mit einem simplen Unterstrich eingeführt wird. Auf der vom Konstruktor `Future.delayed` zurückgegebenen Instanz wird die Methode `then` aufgerufen, die das beschriebene Lambda aufnimmt. Wenn Sie das Beispiel innerhalb einer simplen `main`-Funktion ausführen, erhalten Sie nach Ablauf der angegeben Dauer von zwei Sekunden die Ausgabe `done!`:

```
void main() {

  final durationToWait = Duration(seconds: 2);
  Future.delayed(durationToWait).then((_) => print('done!'));
}
```

**Listing 3.180** Zeitlich gesteuerte Ausgabe des Strings »done!«

### Behandeln von Fehlern

Durch die Methode `then` behandeln Sie den Abschluss der Operation mit Wert. Damit bleibt noch der dritte Zustand, nämlich »abgeschlossen mit Fehler«, zu betrachten. Um einen Fehlerfall zu simulieren, benötigen Sie als Erstes ein Future, das während







der Bearbeitung der zugewiesenen Aktion über das Schlüsselwort `throw` einen Fehlerzustand signalisiert:

```
Future.delayed(
  // Der erste Parameter ist die Verzögerung.
  durationToWait,
  // Der zweite Parameter ist die auszuführende Aktion.
  () => throw 'Fehler während Bearbeitung',
)
// Callback, falls die Aktion fehlerfrei verläuft
.then((_) => print('Fehlerfrei abgeschlossen'))
// Die Behandlung eines Fehlers als Callback
.catchError((error) => print(error));
```

**Listing 3.181** Behandlung von Fehlern eines Futures

Sie bemerken, dass der Konstruktor `Future.delayed` im Vergleich mit vorherigen Aufrufen an dieser Stelle ein zweites Argument erhält, das dem optionalen Positional Parameter (siehe Abschnitt 3.6.2) `computation` zugewiesen wird. In dem zu verarbeitenden Lambda wird über `throw` ein Fehler erzeugt. Darauf folgt der Callback für den Abschluss mit Wert, den Sie aus dem vorherigen Abschnitt kennen. Um einen Fehler zu behandeln, fügt die nächste Zeile mit der Methode `catchError` einen Callback ein, der den Fehler ausgibt.

Wenn Sie das Beispiel laufen lassen, werden Sie feststellen, dass die einzige Ausgabe `Fehler während Bearbeitung` ist. Wie erwartet, rief das Objekt des Typs `Future` den Fehler-Callback auf und nicht jenen für einen erfolgreichen Abschluss.

#### Verkettung von Futures

Methoden von Future wie `then` oder `catchError` geben selbst wieder Objekte von Future zurück. Hierdurch können Sie Callbacks aneinanderketten, wie Sie es bereits bei Iterables in Abschnitt 3.9.4 gesehen haben.

Ein Äquivalent zum `try-catch-finally`-Konstrukt hinsichtlich der Callbacks wäre zum Beispiel:

```
final durationToWait = Duration(seconds: 2);
Future.delayed(
  durationToWait,
  () => throw 'Fehler während Bearbeitung',
)
.then((_) => print('Fehlerfrei abgeschlossen'))
.catchError((error) => print(error))
// Dieser Callback wird in jedem Fall ausgegeben,
// genauso wie es bei finally wäre
.then((_) => print('Abschluss'));
```

#### Wenn Sie mit der Geduld am Ende sind

In der Regel sollten Aktionen von Futures innerhalb einer erwarteten Zeitspanne durchlaufen sein, das müssen sie jedoch nicht. Es könnte etwa zu einem Deadlock kommen, also zu einer Situation, in der sich die parallele Ausführung verhakt hat oder der Zugriff auf Daten dauert länger als gedacht.

Natürlich möchten Sie einer Abarbeitung einen gewissen Zeitraum zugestehen, aber vielleicht müssten Sie irgendwann den Stecker ziehen, um die Bearbeitung zu beenden. Sie können einem Future über die Methode `timeout` eine Dauer einräumen. Kommt es innerhalb dieser Zeit nicht zum Abschluss, wird es terminiert.

Im folgenden Beispiel demonstriere ich die Funktionalität mit Rückgriff auf ein Objekt, das wir über `Future.delayed` erhalten haben:

```
final durationToWait = Duration(seconds: 2);
Future.delayed(durationToWait)
    .timeout(Duration(milliseconds: 500))
    .then((_) => print('Durchgelaufen'))
    .catchError((_) => print('Zeit abgelaufen'));
```

**Listing 3.182** Die Ausführung eines Futures mit »timeout« begrenzen

Wenn Sie den Code aus Listing 3.182 ausführen, werden Sie die Ausgabe `Zeit abgelaufen` sehen, da das Future für zwei Sekunden verzögert, aber ihm nur 500 Millisekunden Laufzeit eingeräumt wurden. Damit zeigt sich außerdem, dass – sollte es zu einem Timeout kommen – das Future in den Zustand »Abschluss mit Fehler« wechselt.

#### Das synchrone Future

Wenn eine Methode asynchrone Features verwendet – etwa *async* und *await*, die Sie in Abschnitt 3.12.3 kennenlernen – oder generell wenn die Funktion ein `Future` als Rückgabetyp fordert, dann müssten Sie ein synchron erworbenes Ergebnis in ein `Future` einwickeln. Das könnte explizit über den Konstruktor `Future.value` passieren. Oder Sie deklarieren die Funktion mit dem Schlüsselwort `async` (siehe ebenfalls Abschnitt 3.12.3) als asynchron. Hierdurch verlässt Dart die *Event Loop* (siehe Abschnitt 3.12.1) aber kurz, um unmittelbar mit dem nächsten Zyklus zurückzukehren.

In diesem Szenario können Sie das `SynchronousFuture` einsetzen, das einen Wert über den Konstruktor aufnimmt. Das Future ruft unmittelbar über `then` einen Callback auf, der die Ausführung, ohne aus der *Loop* herauszugehen, zurück zur aktuellen Zeile bringt:





```
Future<int> getNumberMaybeAsync() {
  return SynchronousFuture(42);
}
```

**Listing 3.183** Ein »SynchronousFuture« einsetzen

#### Weitere Möglichkeiten zur Verarbeitung von Ergebnissen

Zusätzlich zu der einzelnen Abhandlung von erfolgreichem und fehlerhaftem Abschluss eines Futures bietet sich die gemeinsame Verarbeitung über die Methode `whenComplete` an. Zur Verwendung beider Ergebnismöglichkeiten in der Form eines Streams (siehe Abschnitt 3.12.4 und Abschnitt 9.4) lässt sich `asStream` in der Future API verwenden.

### 3.12.3 async und await

Die im vorigen Abschnitt vorgestellten Konstruktoren und Methoden von Future haben Ihnen einen Eindruck von Darts Fähigkeiten in der asynchronen Programmierung geboten Dart überbaut die vorgestellte Future API mit den Schlüsselwörtern *async* und *await*, die Sie vielleicht von C# kennen. Mit ihrer Verwendung lassen sich Futures noch flüssiger lesen und einsetzen. Tatsächlich werden Sie beim Programmieren in Dart eher selten in Kontakt mit den Methoden `then` oder `catchError` von `Future` kommen.

Um diesen Standpunkt zu untermauern, möchte ich Ihnen demonstrieren, wie sich die zuvor verwendeten Beispiele mit verzögerter Ausgabe von Strings und die Fehlerbehandlung im Fall von auftretenden Exceptions oder Errors durch den Einsatz von async und await verändern:

```
Future<void> main() async {
  // die Verzögerung mit await abwarten
  await Future.delayed(durationToWait);
  // wie bei synchronem Code Errors und
  // Exceptions mit try/catch behandeln
  try {
    // Eine asynchrone Funktion mit await abwarten
    await Future.sync(() => throw 'Fehler');
    // synchroner Code
    print('Fehlerfrei abgeschlossen');
  }
  catch(e) {
    // wie gewöhnlich verarbeiten
    print(e);
```

```
  }
}
```

**Listing 3.184** Einsatz von »async« und »await«

#### Eine Funktion als async deklarieren

Eine Funktion oder Methode, in der mit `await` auf Futures oder Futures zurückgebende Funktionen gewartet werden soll, deklarieren Sie, wie in Zeile 1 von Listing 3.185 ersichtlich, mit dem Schlüsselwort `async`, das immer nach der Parameterliste einer Funktion eingefügt wird. Das gilt ebenfalls für die `=>`-Schreibweise:

```
Future<void> anAsyncFunction() async =>
  await Future.delayed(Duration(milliseconds: 50));

Future<void> bar(int a, String b) async {
  await Future.delayed(Duration(milliseconds: 50));
}
```

**Listing 3.185** »async« steht immer hinter der Parameterliste.

[!]

**Asynchrone Konstruktoren**

Dart unterstützt keine asynchronen Konstruktoren. Sollten Sie im Konstruktor asynchronen Code benötigen, etwa das Auslesen von Daten aus der Datenbank, so bleibt Ihnen aber die Möglichkeit, mit `Future.then` und `Future.catchError` entsprechende Callbacks einzubringen. Ein Durchlaufen des Konstruktors, bevor diese Callbacks aufgerufen wurden, lässt sich allerdings nicht vermeiden.

#### Asynchrone Getter

Da Getter auch Funktionen sind, können sie, weil sie keinen weiteren Einschränkungen hinsichtlich async und await unterliegen (wie zum Beispiel Konstruktoren), ebenfalls von den Schlüsselwörtern Gebrauch machen. Bei Gettern ist `async` hinter dem Bezeichner zu platzieren, weil es hier keine Parameterliste gibt. Sie könnten sogar den Aufruf an einer Datenbank direkt aus dem Rumpf eines Getters heraus ausführen:

```
Future<int> get numberOfRows async {
  // Die Datenbank abfragen.
  // Database.calculate ist eine gedachte Funktion.
  final rows = await Database.calculate();
  return rows;
}
```

**Listing 3.186** Ein Getter mit »async«







#### Rücksprung mit await

Der Quellcode wird stets synchron durchlaufen, bis ein `await` erreicht wird. Hier bricht der Kontrollfluss aus und berechnet die von dem Future eingewickelte Aktion im Hintergrund. Sobald sie entweder den Zustand »abgeschlossen mit Wert« oder »abgeschlossen mit Fehler« erreicht, kehrt der Kontrollfluss zurück zur Zeile mit `await`. Im Grunde stellt der Rücksprung ein neues *Event* in der *Event Loop* dar, die zu keiner Zeit stehen bleibt. Das bedeute zum Beispiel für eine Flutter-App, dass durch `async` und `await` kein Hängen der Benutzeroberfläche auftritt.

#### Rückgabetyp einer async-Funktion

Sie haben es in den oberen Beispielen bereits gesehen: Eine mit `async` deklarierte Funktion oder Methode gibt immer ein Future zurück. Das bedeutet, dass beispielsweise eine Funktion mit Rückgabetyp `int` oder `String` mit Wechsel auf `async` die Typen `Future<int>` und `Future<String>` zurückgibt.

### 3.12.4 Streams

*Reactive Programming* ist eine heutzutage häufig eingesetzte Technologie in mobilen Apps. Dart liefert mit seiner Implementierung `Stream<T>` das Grundwerkzeug, um reaktive Dart- und Flutter-Anwendungen zu schreiben.

**Reactive Programming**

*Reactive Programming* bedeutet: reagieren statt agieren. Das heißt, Sie registrieren sich einmal für den Erhalt von Informationen und werden anschließend automatisch über Ereignisse informiert. Es ist dann nicht mehr nötig, in gewissen zeitlichen Abständen nachzufragen, ob sich denn etwas getan hat.

Ereignisse werden an einer Stelle aufgenommen und anschließend an alle interessierten Parteien weitergeleitet. Als Bild hat sich dafür der Strom (*Stream*) durchgesetzt: Ganz so wie etwa beim Radio schalten Sie sich als Zuhörer (*Listener*) in den Strom (*Stream*) der Radiowellen ein, wenn Sie die Nachrichten oder Musik hören möchten.

Streams liefern Ihnen beim Programmieren viele Möglichkeiten, um Daten nach ihrem Eintreffen weiter zu filtern oder um Benachrichtigungen genauer zu definieren.

[+]

Mit dem `StreamController<T>` können Sie relativ einfach zu einer Implementierung von Streams in Ihrem Projekt kommen. Der Controller gewährt Zugriff auf den Eingang (`sink`) und Ausgang (`stream`) von Events. Der generische Parameter `T` beschreibt den Typ des Events. Möchten Sie einen `Stream<String>` erstellen, also einen Stream über String-Events, so instanziieren Sie entsprechend ein Objekt vom Typ `StreamController<String>`. Der folgende Code veranschaulicht dies:

```
final controller = StreamController<String>();
final sink = controller.sink;
final stream = controller.stream;

// Ein Event einfügen
sink.add('Erstes Event');

// ...
// auf Events hören
stream.listen((event) => print(event));
```

**Listing 3.187** Beispielimplementierung von »StreamController«

**Broadcasts**

Falls Sie sich nach der Implementierung eines `StreamController`s wundern, dass Sie eine Fehlermeldung beim Registrieren eines zweiten *Listeners* erhalten: Zunächst lässt der `StreamController` nur einen Listener zu. Möchten Sie mehrere Listener registrieren, so müssen Sie einen *Broadcast*-Stream erstellen. Hierfür bietet die Klasse `StreamController` den Konstruktor `broadcast`:

```
final broadcastController = StreamController<String>.broadcast();
```

Diese Fähigkeit hat aber ein Haken: Ein normaler Stream puffert alle aufgetretenen Ereignisse, bis sich ein Listener registriert. Weil ein Broadcast aber über mehrere Listener verfügen könnte, lässt sich eine Zwischenspeicherung nicht realisieren. Mehr Informationen hierzu finden Sie in Abschnitt 9.4.2, »Broadcasts«.

### 3.12.5 Isolates

Wenn Ihr Dart-Programm oder das darauf aufbauende Flutter-Programm startet, erzeugt die *Runtime* einen sogenannten *Isolate*. Der Name steht für eine Struktur, die über einen eigenen und abgeschlossenen *Speicherbereich* verfügt, da hierauf nur der eine und einzige Thread dieser Struktur zugreifen kann. Sie haben mit der *Event Loop* in Abschnitt 3.12.1 bereits die Verarbeitung von Ereignissen in einem Isolate kennengelernt.

Sollte der Bedarf bestehen, aufwendige Berechnungen auszulagern, liefert Dart über die Isolate-API Schnittstellen, um neue Instanzen zu erstellen. Auch wenn der Haupt-Isolate Ihrer Anwendung weitere Isolates erzeugt, kann er dennoch nicht auf deren Speicher oder Event Loop zugreifen. Die einzige Kommunikation zwischen Isolates erfolgt über Nachrichten und keineswegs über den Speicher.







Weil sich die Isolate-API in der Handhabung als sehr komplex herausstellt, sind einige Abstraktionen entstanden, die Ihnen die Ausführung einer Funktion ausgelagert in einem Isolate ermöglichen oder gleich ganze Isolate-Pools (*Thread Pools* oder *Worker Pools*) bereitstellen. Zu nennen sind an dieser Stelle die `compute`-Funktion von Flutter (*flutter/foundation.dart*) und die Packages `worker_manager` sowie `computer`. (Mehr zu Packages erfahren Sie in Kapitel 4, »Pubs: Abhängigkeiten komfortabel verwalten«.)

## 3.13 Die Library

Der Begriff *Library* ist Ihnen im Zusammenhang mit der Sichtbarkeit von Variablen, Funktionen und Klassen bereits in den vorangegangenen Abschnitten begegnet. Dabei stellte sich heraus, dass Elemente, deren Bezeichner ein Unterstrich (_) vorangestellt ist, nur in der jeweiligen Datei sichtbar sind.

Ein Dart-Projekt, das etwa über Android Studio oder Visual Studio Code erstellt wurde, gilt per se als Library. Sollten Sie ein neues Projekt erstellen, finden Sie darin unter anderem einige autogenerierte Dateien, die Paketkonfiguration (siehe Kapitel 4), eine Readme-Datei und den Unterordner *lib*, der den Quellcode beinhaltet.

### 3.13.1 Mini-Library

Dart betrachtet eine einzelne Datei als *Mini-Library*. Es wird empfohlen, jeweils eine Mini-Library pro Klasse anzulegen, es sei denn, dass die Elemente eng miteinander zusammenhängen. Das folgende Beispiel demonstriert die Mini-Library in der Datei *a.dart*. Wie Sie erkennen, befinden sich zwei Klassen darin. Die erste Klasse (`_Private`) ist aufgrund ihrer Sichtbarkeit auf die Datei beschränkt. Weil die Klasse `A` ebenfalls in *a.dart* liegt, ist es ihr möglich, den Typ `_Private` aufzulösen:

```
// Datei: a.dart

// Die Klasse _Private ist nur innerhalb der
// Mini-Library sichtbar
class _Private {

}

// Weil A in der gleichen Library definiert ist,
// kann A auf das Element _Private zugreifen
class A {
  final _Private somePrivate = _Private();
}
```

**Listing 3.188** Beispiel einer Mini-Library

### 3.13.2 Eine Library importieren

Um die in verschiedene Dateien aufgeteilten Elemente, soweit sie nicht in der Sichtbarkeit eingeschränkt sind, in anderen Dateien verwenden zu können, müssen Sie sie mit dem Schlüsselwort `import` in die jeweilige Mini-Library oder Datei hineinziehen.

#### Dateipfad

Exemplarisch greifen wir auf die Mini-Library aus dem obigen Abschnitt zurück (*a.dart*). Die importierende Datei oder Mini-Library ist *main.dart*, die den *Einsprungspunkt* (engl. *entry point*) für Dart anbietet:

```
// a.dart aus Listing l3.188
import 'a.dart';

void main() {
  // Durch den oberen Aufruf import
  // können wir die Klasse A auflösen.
  final a = A();
}
```

**Listing 3.189** Importieren von »a.dart«

Auf `import` kann jeweils ein *Dateipfad*, eine *Package-URI* (*Unique Resource Identifier*) oder ein Dart-Library-Schema folgen. Die Verwendung des Dateipfads haben Sie im oberen Beispiel bereits kennengelernt.

#### Package-URI

*Packages*, mit denen Sie sich zu einem späteren Zeitpunkt noch beschäftigen werden, sind kurz gefasst Sammlungen aus Dart-Librarys, die in einem festgelegten Format zur Wiederverwendung oder Integration bereitgestellt werden. Von einem Package zur Verfügung gestellte Librarys bzw. Dateien können Sie über das genannte Package-URI in Ihre Sichtbarkeit holen. Angenommen, Sie möchten die Datei *utils.dart* aus dem Package `example_package` integrieren, dann könnten Sie sie wie folgt in der `main`-Funktion verwenden:

```
import 'package:example_package/utils.dart';

void main() {
  // Funktionen aus utils.dart verwenden ...
}
```

**Listing 3.190** Eine Datei aus einem Package importieren







**Dateien Ihres Projekts über eine Package-URI einbinden**

Dart- und Flutter-Projekte sind in der Form eines *Packages* (Pakets) strukturiert, das wir in Abschnitt 3.14 näher beleuchten. Technisch steht Ihnen daher nichts im Wege, projektlokale Dateien statt über den Dateipfad über die Package-URI einzubinden. Ihnen bieten sich am angenommenen Beispiel der Datei *a.dart*, die in Ihre *main.dart*-Datei importiert werden soll, zwei Möglichkeiten:

```
// Import über Dateipfad …
import 'a.dart';
// … und über die Package-URI
import 'package:ihr_paket/a.dart';
```

Die einhellige Empfehlung in Dart lautet, für Dateien des eigenen Packages oder Projekts stets Dateipfade zu verwenden. Die Package-URI ist tatsächlich nur für das Importieren von Dateien aus zu benutzenden Packages einzusetzen.

#### Dart-Library

Die dritte und letzte Variante eines Imports ist diejenige, um mit Dart ausgelieferte Librarys in Ihren Dart-Code einzubinden. Beispiele dafür sind `dart:collection`, `dart:convert` oder `dart:math`.

Möchten Sie eine solche Abhängigkeit verwenden, etwa `dart:math`, so würde die Anweisung folgendermaßen aussehen:

```
import 'dart:math';
```

## 3.14 Struktur eines Projekts

Im Verlauf der Entwicklung einer App werden nicht selten Hunderte Elemente auf dem *Dateisystem* erzeugt, etwa Programmcode, Konfigurationen oder Bilder. Deshalb möchte ich Sie kurz mit den Konventionen der Projektstruktur in Dart und dem Umgang mit mehreren Dateien vertraut machen.

Dart-Projekte teilen sich in zwei sehr ähnliche Kategorien auf: in das *Library-Package* und in das *Application-Package*. Ersteres zielt auf die Verwendung als integrierbares *Package* ab, beispielsweise verteilt über *https://pub.dev*, während Letzteres ein App-Projekt darstellt. Der Begriff *Package* setzt an dieser Stelle keinen Umgang mit Paketverwaltung und -versionierung voraus, die wir in Kapitel 4 behandeln werden. Hier beschränkt er sich auf die Strukturierung der Dateien, die nach einem festgelegten Schema vorgenommen wird.

Wenn Sie ein neues Dart-Projekt erstellen, sollte das von Ihrer IDE erstellte Projekt folgendermaßen aussehen:

```
Projektname:          hello_world
-> .dart_tool         # generierter und Dart-eigener Ordner
-> .gitignore         # Git-typische Ignore-Datei
-> .idea              # generiert, falls mit Android Studio erstellt
-> .metadata          # generierte Metadaten zum Projekt
-> .packages          # generierte Abhängigkeitenzusammenfassung
-> README.md          # Die Readme des Projekts
-> hello_world.iml    # generiert, falls mit Android Studio erstellt
-> bin                # Ordner für Skripte (meist nur in Library)
-> lib                # Hauptordner für Quellcode
-> pubspec.lock       # generiert für pubspec.yaml (Kapitel 4)
-> pubspec.yaml       # Bibliotheksdatei (Konfiguration)(Kapitel 4)
-> test               # Unterordner für automatische Tests
```

**Listing 3.191** Die Struktur eines Dart-Projekts

Die wichtigsten Ordner für Sie sind *bin*, *lib* und *test*. Mit *bin* kommen Sie wahrscheinlich nur in Berührung, wenn Sie ein *Library-Package* erstellen. Alle in diesem Ordner hinterlegten Skripte können, falls Ihr Paket über `pub` (vgl. Kapitel 4) integriert wurde, mit dem Kommando `pub run <script_name>` ausgeführt werden. Innerhalb von *lib* halten Sie Ihre öffentlichen Quellcode-Dateien, also jene, die Sie anderen über `import` (siehe Abschnitt 3.13.2) zur Verfügung stellen. Dieser Aspekt kann allerdings bei einem *Application-Package* vernachlässigt werden.

**Quellcode außerhalb von lib**

Sie sind keineswegs auf den Ordner *lib* Ihres Projekts beschränkt. Ihren Quellcode können Sie ganz nach Ihrer Präferenz im Projektverzeichnis ablegen. Möchten Sie aber, dass Ihre Dateien von anderen Packages oder Apps gefunden werden können, wenn sie Ihr Package integrieren, dann müssen Sie sich unterhalb von *lib* befinden.

In Abschnitt 3.13.2 habe ich Ihnen den Import über die Package-URI vorgestellt. Er kann nur auf Quellcode zugreifen, der unterhalb von *lib* platziert wurde. Das können Sie, wenn Sie neugierig sind, auch über die Datei *.packages* nachprüfen, die automatisch vom *pub-Paketmanager* erstellt wird. Alle dort hinterlegten Pfade verweisen jeweils auf den *lib*-Unterordner eines Packages in Ihrem *pub-cache*.

Der letzte der drei wichtigsten Ordner ist *test*. Er stellt die Anlaufstelle für automatische Tests dar, etwa für Unit-Tests (siehe Abschnitt 20.5.1) oder für Widget-Tests (siehe Abschnitt 20.5.2).

# Kapitel 5
# Widgets

*»Ich suche die Einheit in allem, um mit ihr alles zu durchdringen.«*
*– Konfuzius*



In Flutter geschriebene grafische Oberflächen bestehen zum allergrößten Teil aus *Widgets*, die nichts anderes als deklarative Beschreibungsfunktionen darstellen. Die imperative Schreibweise, die von manchen auf C, C++ oder Java aufbauenden UI-Frameworks genutzt und in der die Oberfläche schrittweise über Instruktionen definiert wird, unterscheidet sich davon grundlegend, wie Sie in diesem und den folgenden Kapiteln sehen werden.

Das kürzlich von Apple eingeführte *SwiftUI* folgt ebenfalls dem deklarativen Ansatz, während die »althergebrachte« Art unter Verwendung von *UIViews*, Controllern und Models den imperativen Ansatz benutzt.

Im Folgenden möchte ich Ihnen anhand kleinerer Beispiele das Widget, seine Handhabung und seine Vorzüge vorstellen. Sie werden lernen, wie Sie konstante Widgets einsetzen sowie Widgets mit einem Status (*State*), der das Widget verändert. Außerdem werden die beiden Design-Stile *Material* und *Cupertino* (iOS-Stil) vorgeführt.

**Imperativ und deklarativ?**

Diese beiden Arten der Programmierung durch prägnante Spezifizierungen auseinanderzuhalten, ist eine Herausforderung in sich. Imperativer und deklarativer Dart-Code ist schließlich technisch das Gleiche und besteht jeweils aus Funktionen, Objekten und Variablen.

Im imperativen Programmierstil erstellen Sie alle Bestandteile einer Oberfläche als Instanzen von Objekten und verändern diese schrittweise. Dabei ist die Reihenfolge der Instruktionen vorgegeben. Möchten Sie etwa die Hintergrundfarbe einer Page (auch Screen oder View) ändern, so greifen Sie auf das Objekt zu und setzen dort die Farbe:

```
final app = getCurrentApp();
final page = app.getCurrentPage();
page.color = Colors.red;
```

Soll die Farbe anschließend geändert werden, womöglich durch ein Button-Ereignis, so würden Sie einen Button-Handler definieren, der in dessen Funktionsrumpf die Anpassung durchführt.

Wenn Sie dem deklarativen Ansatz folgen, beschreiben Sie stets einen Zustand der Oberfläche. Was soll auf ihr zu sehen sein und welche Parameter wirken auf sie ein? Flutters Widgets als deklarative Elemente ähneln einer mathematischen Funktion. Das obere gedankliche Beispiel aufnehmend, ließe sich die Page in Pseudocode beschreiben als:

```
F(x) = App(currentPage: Page(color: x)) mit x = Colors.red
```

Wäre in diesem Fall eine andere Farbe gewünscht, so würde nur der Parameter `x` geändert, sodass zum Beispiel `x = Colors.green` sei. Beim nächsten Durchlauf der Funktion, der vom gedachten Button und dem zugehörigen Berührungsereignis ausgelöst werden könnte, wechselt der Zustand entsprechend. Wichtig ist hier der Unterschied, dass der Button-Handler nicht die Farbe setzen würde, sondern nur darüber benachrichtigt, dass eine Neuberechnung durchgeführt werden soll.

Losgelöst von Dart oder Code lässt sich der Unterschied womöglich auch anhand einer Analogie erklären: Stellen Sie sich vor, Sie möchten von Ort A nach Ort B gelangen. Sie haben dazu zwei Möglichkeiten:

- Der imperative Ansatz verlangt von Ihnen, dass Sie jede Richtungsänderung in ein System eingeben und somit den Kurs festlegen. Sie müssen also definieren, *wie* Sie ans Ziel kommen möchten.
- Im deklarativen Fall nennen Sie dem System nur den Zielort, und das System berechnet anschließend mit dem Navigationssystem die Streckenführung. Sie teilen demnach lediglich mit, *wohin* (oder *was*) Sie möchten. Das gedanklich aufwendige *Wie* soll das System errechnen.

## 5.1 Ein erstes Widget

An dieser Stelle möchte ich mit Ihnen eine erste minimale Flutter-App bauen. Hierbei gehen wir durch die einzelnen Schritte, die zur Anzeige eines Textfelds auf dem Bildschirm nötig sind. Wir benutzen zunächst den *Material Design*-Stil. Nachdem Sie Ihre ersten Widgets kennengelernt und platziert haben, schwenken wir in späteren Abschnitten per *Hot-Reload* auch auf den iOS-Stil, Cupertino, um.

Erstellen Sie als Erstes, wie ich in Abschnitt 2.2 jeweils für Visual Studio Code und Android Studio vorgeführt habe, ein neues Projekt, um den Flutter-Code aufzunehmen. Falls nach der Fertigstellung die Datei *main.dart* nicht von selbst ausgewählt sein sollte, wählen Sie sie über die Projektübersicht aus, die sich normalerweise auf der linken Seite befindet.

In der geöffneten Datei sehen Sie eine Menge Code. Sie können, um Ihre Einstellungen und Ihre *Toolchain* zu überprüfen, einen Testlauf durchführen, indem Sie das Projekt auf einen *Android Emulator* oder *iOS Simulator* übertragen. Zumeist werde ich





im Buch der Einfachheit halber vom »Simulator« sprechen, ich beziehe mich dabei aber auf beide Technologien. Ihr Projekt sollte sich wie in Abbildung 5.1 auf dem virtuellen Gerät zeigen:


**Abbildung 5.1** Die Standardvorlage eines Flutter-Projekts wurde ausgeführt.

**Das Projekt lässt sich nicht ausführen?**

Bitte überprüfen Sie, ob die IDE oder Flutter Ihre virtuellen Geräte erkannt hat. Wechseln Sie dazu zur Konsole, und führen Sie das Kommando `flutter doctor -v` aus. Im unteren Teil der folgenden Ausgabe sollten Sie mindestens ein verbundenes Gerät (*connected device*) sehen, ähnlich wie hier im Beispiel mit einem Android Emulator:

```
Connected device (1 available)
    • Android SDK built for x86 (mobile) • emulator-5554 • android-x86
    • Android 10 (API 29) (emulator)
```

Wenn die Ausgabe kein Gerät erkannt hat, prüfen Sie Ihren *AVD Manager* im Fall von Android oder den iOS Simulator auf eine laufende Instanz.

Nach dem erfolgreichen Test steht Ihnen nichts weiter im Wege. Löschen Sie in der Datei *main.dart* alles unterhalb von Zeile 5 bzw. unter der Funktion `main`:

```
import 'package:flutter/material.dart';

void main() {
  runApp(MyApp());
}

// alles Folgende löschen …
```

**Listing 5.1** Der Ausgangspunkt der Flutter-App

Werfen wir einen kurzen Blick auf den Code, der noch in der Datei übrig geblieben ist. Die Funktion `main` ist Ihnen bereits aus Abschnitt 3.2 vertraut, in dem Sie sich mit der Programmiersprache Dart beschäftigt haben. Im Rumpf wird eine bisher unbekannte Funktion aufgerufen, die aus dem Package *flutter* importiert wurde, wie Sie der Zeile 1 von Listing 5.1 entnehmen. Der Name ist an dieser Stelle treffend gewählt, da die Funktion das übergebene Widget als Wurzel (*Root*) in den Widget-Baum (*Tree*) übernimmt und Ihre App darauf aufbaut. Übergeben werden soll das Widget `MyApp`, das Sie noch definieren müssen.

Nun wird Ihnen der Dart Analyzer aufzeigen, dass er das Element `MyApp` nicht auflösen kann. Darum kümmern Sie sich als Nächstes, indem Sie Ihre eigene Klasse `MyApp` schreiben. Fügen Sie den folgenden Code unterhalb der Funktion `main` ein:

```
class MyApp extends StatelessWidget {

}
```

**Listing 5.2** »MyApp« als »StatelessWidget«

Damit haben Sie die Fehlermeldung gegen eine neue ausgetauscht, denn nun sehen Sie die Meldung `Missing concrete implementation of 'StatelessWidget.build`. Die Klasse `StatelessWidget`, die Sie über das Schlüsselwort `extends` erben, deklariert die Methode `build`, bietet jedoch keine Implementierung an. Das ergibt durchaus Sinn, da die abstrakte Basisklasse `StatelessWidget`, die als Vorlage für alle zustandslosen Widgets benutzt wird, nicht wissen kann, wie Ihre Benutzeroberfläche aussehen soll. In den Abschnitten `5.1.1` und `5.2` werden wir das Thema »zustandslose (*stateless*) und zustandsbehaftete (*stateful*) Widgets« vertiefen.

Schreiben Sie, um die Fehlermeldung zu beseitigen, die Methode `build` innerhalb der Klasse `MyApp`:





```
@override
Widget build(BuildContext context) {
  return MaterialApp(
    home: Scaffold(
      body: Text(
        'Willkommen',
      ),
    ),
  );
}
```

**Listing 5.3** Implementierung der Methode »build«



**Methoden autogenerieren lassen**

Moderne IDEs weisen meist nicht nur auf einen Fehler hin, sondern liefern zusätzlich eine Lösung. Fehlt (wie im aktuellen Beispiel) eine Methode, dann können Sie einen sogenannten *Stub* erstellen lassen.

Falls Sie den Code aus Listing 5.3 noch nicht implementiert haben, zeigen Visual Studio Code und Android Studio jeweils einen Hinweis in Form einer gestrichelten Linie unter dem Klassennamen »MyApp«. Wenn Sie den Mauszeiger darüber halten, liefern die IDEs einen Hinweisdialog wie in Abbildung 5.2, aus dem heraus Sie einen solchen Methoden-Stump erstellen können:

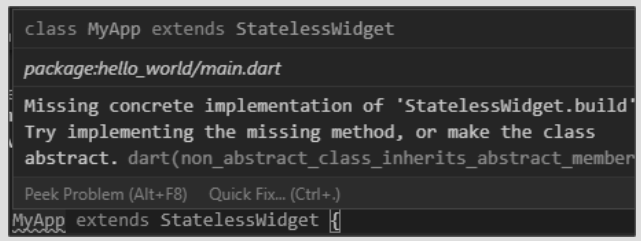


**Abbildung 5.2** Hinweisdialog in Visual Studio Code

Über die Schaltfläche Quick Fix... • Create 1 missing override(s) erstellen Sie zum Beispiel diesen Code für die Methode `build`:

```
@override
Widget build(BuildContext context) {
  // TODO: implement build
  throw UnimplementedError();
}
```

**Listing 5.4** Die Methode »build«, autogeneriert von der IDE

Solche Hilfen erhalten Sie außerdem durch die Tastenkombination Cmd + . . Das gleiche Prozedere können Sie in Android Studio ausführen. Wenn Sie den Mauszeiger über der Fehlermeldung ruhen lassen, erscheint ebenfalls ein Hinweis (siehe Abbildung 5.3), und die entsprechende Tastenkombination lautet hier Alt + ↵.

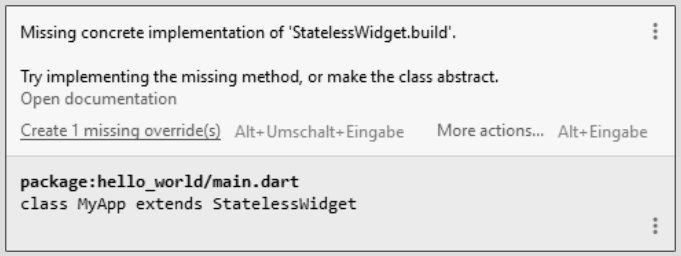


**Abbildung 5.3** Hinweisdialog in Android Studio

Die Methode `build` aus Listing 5.3 erwartet, von Ihnen mit dem Argument `context` vom Typ `BuildContext` aufgerufen zu werden. Ein `BuildContext` liefert Informationen zum Kontext Ihres Widgets, etwa wo es im *Widget-Tree* eingefügt wurde. Diesen Typ werden wir in Abschnitt 5.4 genauer betrachten. Die obere Implementierung von `build` benutzt den `context` allerdings nicht, sodass Sie Ihre Aufmerksamkeit erst einmal auf die nächsten Zeilen richten können.

Der `return`-Anweisung der Methode wird ein Widget vom Typ `MaterialApp` zugeführt, das Funktionalitäten einer App wie Navigation und Eigenschaften wie das *Theme* einwickelt und zur Integration in den Widget-Tree bereitstellt. Der Bezeichner `MaterialApp` lässt bereits erahnen, dass dieses Widget dem Designansatz *Material Design* folgt.

Die nächste Zeile beschreibt die Zuweisung des Widgets `Scaffold` an die Variable `home` der `MaterialApp`, die Sie soeben kennengelernt haben. Bei `Scaffold` handelt es sich erneut um ein Material Design-spezifisches Widget, das die anzuzeigende Oberfläche in *AppBar*, *Body* und *BottomNavigationBar* unterteilt und weitere Konfigurationen hinsichtlich des Themes übernimmt. Mit anderen Worten: Durch die Integration nur zweier Widgets, nämlich `MaterialApp` und `Scaffold`, erhalten Sie bereits das komplette *Look & Feel* einer App im Material Design-Stil! In Abschnitt 6.6.1, »Das Scaffold«, erfahren Sie mehr zu diesem Widget.

Das einzige explizit zugewiesene Element der Unterteilung durch `Scaffold` ist die Variable `body`. Ihr wird ein `Text`-Widget zugewiesen, das den String »Willkommen« anzeigen soll. Dieses Widget ist nicht auf das Material Design beschränkt, sondern kann in beiden Kontexten eingesetzt werden. Das für das jeweilige Design übliche Aussehen erhält `Text` über ein intern aufgelöstes `Theme`-Widget, das initial, wie oben angedeutet, von `MaterialApp` und `Scaffold` konfiguriert und angeboten wird. Das heißt aber nicht, dass Sie das `Theme` von einem Text nicht nach Ihren Wünschen anpassen könnten, wie Sie in Kapitel 11 sehen werden.





Mit der geschriebenen Methode `build` sollte sich bei erneutem Ausführen ein Anblick wie in Abbildung 5.4 zeigen:


**Abbildung 5.4** Das Widget auf dem Android-Emulator

Ihnen wird sofort auffallen, dass der Text »Willkommen« hinter der *Statusbar* des Geräts liegt. Das kommt daher, dass Flutter Zugriff auf den kompletten Bildschirm hat und nicht auf gewisse Bereiche beschränkt ist. Flutter hat jedoch keine Anweisung erhalten, den Text an einer bestimmten Stelle oder mit einem definierten Abstand zum oberen linken Punkt zu zeichnen.

In Apps, die der Material-Designrichtung folgen, finden Sie häufig sogenannte *AppBars*, die sich über den Kopf des Bildschirms erstrecken und zumeist weitere Menüs beherbergen, etwa über ein *Menu-Icon* (die drei vertikalen Striche). Momentan ist eine solche Leiste nicht definiert; sie würde den Inhalt aber unter der Statusbar von Android hervorholen. Gehen Sie zurück zur Methode `build`, und fügen Sie dem Widget `Scaffold` eine Standard-AppBar über den Konstruktor von `AppBar` hinzu:

```
@override
Widget build(BuildContext context) {
  return MaterialApp(
    home: Scaffold(
      appBar: AppBar(),
      body: Text(
        'Willkommen',
      ),
    ),
  );
}
```

**Listing 5.5** Das »Scaffold« mit »AppBar«

Nachdem Sie, wie im Code in Zeile 5 zu sehen ist, die `AppBar` definiert haben, führen Sie, um die Änderungen direkt auszuprobieren, einen *Hot-Reload* aus. Drücken Sie

dafür auf das Blitzsymbol, das Sie in Android Studio in der oberen rechten Region finden (siehe Abbildung 5.5).

**Abbildung 5.5** Der Hot-Reload-Blitz in Android Studio

Wenn Sie mit Visual Studio Code arbeiten, so haben Sie sicherlich die *Debug-Palette* in der oberen Mitte des Fensters bemerkt, in der sich der Hot-Reload, ebenfalls mit dem Blitz als Symbol, befindet (siehe Abbildung 5.6).

**Abbildung 5.6** Der Hot-Reload-Blitz in Visual Studio Code

Als Ergebnis des kurzen Eingriffs sollte sich jetzt auf Ihrem Bildschirm ein ähnliches Bild zeigen wie auf dem Screenshot aus Abbildung 5.7:

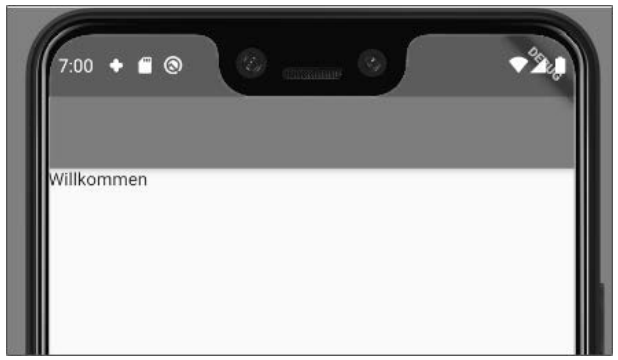


**Abbildung 5.7** Die App mit »AppBar«

Schon mit dieser kleinen Anpassung haben Sie die Vorteile von Flutter voll auskosten können: Die Änderungen waren mit dem Hot-Reload sofort auf Ihrem Testgerät. Das Übertragen und Anzeigen von Anpassungen auf physikalischen Geräten wie Ihrem iPhone oder Android Smartphone erfolgt übrigens genauso schnell.

**Die Komposition in Flutter**

Zusätzlich haben Sie gerade mit der *Komposition* eines der Grundprinzipien hinter Flutter angewendet: Anstatt eine eigene Erweiterung für das Widget zu bauen (etwa einen rechteckigen, farbigen Bildschirmbereich, der über dem Text eingefügt wird), haben Sie auf die bereits vorhandene Implementierung der `AppBar` als Widget zurückgegriffen und es mit Ihrem Widget kombiniert.

Flutter möchte dazu ermutigen und ruft auch dazu auf, stets flache Widgets zu bauen und miteinander zu kombinieren. Widgets erben somit nicht voneinander über lange





Ketten, wie Sie das vielleicht von anderen Frameworks, etwa *UIView* in iOS, kennen. Ein Widget basiert entweder auf `StatelessWidget` (siehe Abschnitt 5.1.1) oder `StatefulWidget` (siehe Abschnitt 5.2).

Wenn Sie einen Blick hinter die Kulissen von Flutter werfen und sich die Implementierungen von Widgets ansehen, finden Sie viele verschiedene kleine Widgets, die für eine Aufgabe zusammengebracht werden. Widgets folgen damit dem in Unix (auf dem z. B. Linux und macOS basieren) vorgelebten Ansatz, nach dem ein Programm nur eine Funktion ausführt: *Do one thing, do it well.*



Dem Grundsatz der *Komposition* folgend, definieren Widgets in Flutter keine Eigenschaften wie *Padding* oder *Alignment*. Um den Text »Willkommen«, der sich in Abbildung 5.7 in der oberen linken Ecke befindet, horizontal wie vertikal zu zentrieren, wenden Sie auf ihn stattdessen ein weiteres Widget an:

```
@override
Widget build(BuildContext context) {
  return MaterialApp(
    home: Scaffold(
      appBar: AppBar(),
      body: Center(
        child: Text(
          'Willkommen',
        ),
      ),
    ),
  );
}
```

**Listing 5.6** So wird der Text horizontal und vertikal zentriert.

Das Widget `Center` nimmt den Platz von `Text` ein und setzt die Variable `body` von `Scaffold`. Das `Text`-Widget wird danach der Variablen `child` von `Center` zugewiesen, die Sie in Widgets häufig antreffen werden. Diese Änderung demonstriert ebenfalls, wie die Widgets jeweils als eine Funktion auf einen Zustand wirken. Mithilfe von `Center` haben Sie demzufolge eine Zentrierung auf den `Text` angewendet.

Wenn Sie den Code so wie in Listing 5.6 angepasst und einen neuen Hot-Reload ausgeführt haben, sollte sich das Widget `Text` in der horizontalen und vertikalen Mitte des Bildschirms befinden (siehe Abbildung 5.8).

Möchten Sie allgemein eine Eigenschaft oder Funktion auf ein Widget anwenden, z. B. das Zentrieren, so müssen Sie das Ziel-Widget dementsprechend mit einem Eigenschafts- oder Funktions-Widget einwickeln.


**Abbildung 5.8** Der Text ist nun zentriert.

Angenommen, der weiße Bildschirm soll die Farbe Grau erhalten. Dann erreichen Sie das Ziel mit dem Widget `ColoredBox`:

```
@override
Widget build(BuildContext context) {
   return MaterialApp(
     home: Scaffold(
       appBar: AppBar(),
       body: ColoredBox(
         color: Colors.grey,
         child: Center(
           child: Text(
             'Willkommen',
           ),
         ),
       ),
```





```
     ),
   );
 }
```

**Listing 5.7** Der Bildschirm wird mit »ColoredBox« grau gefärbt.

Dieses Mal wird das zuvor eingebrachte Widget `Center` eingewickelt, und zwar von `ColoredBox`, das es ermöglicht, eine Farbe über die Variable `color` zuzuweisen, die anschließend den beanspruchten Bereich einfärbt, den auch das über `child` zugeführte Widget einnimmt. Führen Sie einen Hot-Reload aus. Der Bildschirm ist jetzt grau.

Erlauben Sie mir, noch ein weiteres Widget vorzustellen: `Padding`. Angenommen, Sie möchten den grauen Bereich auf jeder Seite etwa 20 *logische Pixel* vom Rand entfernt beginnen lassen, so wickeln Sie die `ColoredBox` entsprechend wie folgt ein:

```
@override
Widget build(BuildContext context) {
  return MaterialApp(
    home: Scaffold(
      appBar: AppBar(),
      body: Padding(
        padding: const EdgeInsets.all(20),
        child: ColoredBox(
          color: Colors.grey,
          child: Center(
            child: Text(
              'Willkommen',
            ),
          ),
        ),
      ),
    ),
  );
}
```

**Listing 5.8** Ein »Padding« erzeugt einen Rand um die »ColoredBox«.

Somit wird nun erneut ein anderes Widget der Variable `body` von `Scaffold` zugewiesen: das `Padding`. Mit diesem Typ werden wir uns in Abschnitt 6.1.4, »Padding«, noch genauer befassen. Weil ich dem nicht allzu viel vorwegnehmen möchte, sei an dieser Stelle nur angemerkt, dass Sie der Eigenschaft `padding` ein Objekt vom Typ `EdgeInsets` übergeben. Durch den Konstruktor bestimmen Sie für alle Seiten (`all`) einen Abstand über die besagten 20 logischen Pixel. Und damit haben Sie das Vorhaben schon umgesetzt! Ein weiterer Hot-Reload führt zu dem Stand aus Abbildung 5.9:


**Abbildung 5.9** Der mit »Padding« geschaffene Rand

### 5.1.1 Stateless Widgets

Damit haben Sie bereits Ihr erstes `StatelessWidget` eingesetzt, indem Sie die Klasse `MyApp` von dieser Basisklasse abgeleitet haben. Weil sich der Inhalt eines solchen Widgets nicht von innen heraus ändert, benötigt es kein *State*-Objekt (siehe Abschnitt 5.2.4), das Informationen über verschiedene *Renderingvorgänge* (siehe Abschnitt 5.4) hinweg festhalten kann. In einem `StatelessWidget` sollten sich generell keine nicht-`final`en Variablen befinden, weil sonst ein State implementiert würde – und genau dafür ist das `StatefulWidget` gedacht, das im nächsten Abschnitt betrachtet wird.

Außerdem beschränkt sich die Basisklasse auf die Methode `build`, die es, wie Sie es im vorigen Abschnitt gelernt haben, zu implementieren gilt. Damit zeigt sich das `StatelessWidget` als für Programmierer und auch für die *Rendering*-Pipeline sehr einfaches und performantes Element – perfekt geeignet für Konzepte wie Text, Bilder und andere sich nicht verändernde Inhalte.





## 5.2 Stateful Widgets

Ein `StatefulWidget` ist komplexer als das zuvor beschriebene `StatelessWidget` und bildet unter anderem den *Lifecycle* eines Oberflächenelements – etwa die Initialisierung, das Einhängen in den Widget-Tree oder das Aushängen und Löschen durch den *Garbage Collector* – in Callbacks ab.



**Der Widget-Tree**

Sie werden in Abschnitt 5.4.1 tiefer in Flutters Widget-Tree eintauchen, wo wir auch den *Element-Tree* und den *RenderObject-Tree* behandeln. Der *Widget-Tree* besteht aus allen Widgets, die inklusive der Wurzel (*root*) an die Funktion `runApp` übergeben wurden. Zum aktuellen Projektstand umfasst der *Widget-Tree* somit die folgenden Elemente:

```
MaterialApp
  -> Scaffold
    -> appBar
      -> AppBar
    -> body
      -> Padding
        -> ColoredBox
          -> Center
            -> Text
```

**Listing 5.9** Der aktuelle Widget-Tree

Mithilfe eines `StafefulWidgets` möchte ich mit Ihnen den Willkommensbildschirm, den wir in Abschnitt 5.1 geschrieben haben, zu einem Login-Bildschirm ausbauen, der zwei Eingabefelder und einen Button enthält. Diese drei Elemente werden zu einem Login-Widget zusammengefasst, das einen internen State besitzt, um überprüfen zu können, ob die Eingabefelder ausgefüllt wurden.

### 5.2.1 Die Ausgangssituation

Weil wir die letzten Widgets nur zu Demonstrationszwecken hinzugefügt haben, können Sie `Padding` und `ColoredBox` aus dem Widget-Tree entfernen. Das lässt sich händisch erledigen, aber auch mit Bordmitteln Ihrer IDE. Wenn Sie sich für den semiautomatischen Weg entscheiden, wählen Sie mit dem Cursor das Element aus, das Sie entfernen möchten, und drücken die Tastenkombination für die Hilfestellung, etwa Strg + . (Cmd + . unter macOS) in Visual Studio Code oder Alt + ↵ in Android Studio. Wählen Sie dann den Eintrag Remove this widget.

Sie sehen anschließend, wie Ihre IDE das Widget entfernt und gleichzeitig die neue Einrückung komplett übernimmt. Praktisch! Natürlich ist diese Funktion genauso

mit dem Mauszeiger erreichbar: Führen Sie ihn hierzu auf das Element, das Sie bearbeiten wollen, und klicken Sie es an. Wählen Sie in Ihrer IDE das Glühbirnen-Symbol (siehe Abbildung 5.10) aus, das am Anfang der Zeile erscheint. Dann erhalten Sie das gleiche Menü wie durch das Tastenkürzel zuvor.

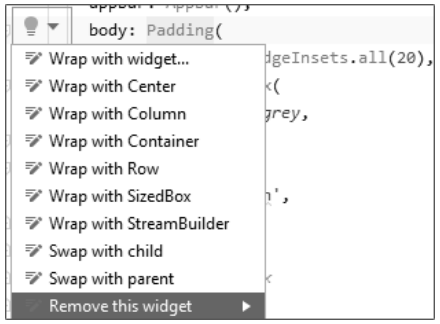


**Abbildung 5.10** Das Glühbirnen-Symbol in Android Studio

Nachdem Sie die Widgets `Padding` und `ColoredBox` nacheinander aus dem Widget-Tree entfernt haben, sollten Sie jetzt den folgenden Stand erzielt haben:

```
@override
Widget build(BuildContext context) {
  return MaterialApp(
    home: Scaffold(
      appBar: AppBar(),
      body: Center(
        child: Text(
          'Willkommen',
        ),
      ),
    ),
  );
}
```

**Listing 5.10** Die »build«-Methode nach dem Aufräumen

### 5.2.2 Ein neues (Stateful)Widget erstellen

Mit dem letzten Schritt haben Sie eine ordentliche Ausgangsbasis erreicht und können nun ein neues Widget erstellen, das die Login-Maske definieren soll. Diese Maske wird aus zwei vertikal angeordneten Eingabefeldern und einem darunter positionierten Button bestehen, wie Sie in Abbildung 5.11 sehen.






**Abbildung 5.11** Konzept, wie die Oberfläche später aussehen soll

Zum Erstellen eines neuen Widgets bieten sich in Ihrer IDE zwei Möglichkeiten: Sie können es entweder über eine kurze Zeichenfolge im Editor aus einer Vorlage heraus erstellen oder, falls Sie bereits einige Arbeiten an einem Teil des Widget-Tree durchgeführt haben, diesen direkt per Kommando in ein neues Widget überführen.

#### Ein Widget aus einem Snippet erstellen

Zuerst spielen wir das aus der Vorlage erstellte Widget durch. Setzen Sie den Cursor unterhalb der Klasse `MyApp` an, und tippen Sie »stful« in eine leere Zeile. In Visual Studio Code und Android Studio wird jetzt der Hilfsdialog mit dem Glühbirnen-Symbol erscheinen und den Eintrag New Stateful Widget anbieten. Durch einen Mausklick oder die ↹-Taste wählen Sie die vorgeschlagene Aktion aus. Auf Ihrem Bildschirm sollten neue Zeilen hinzugefügt worden sein, ähnlich dieser Ausgabe:

```
class  extends StatefulWidget {
  @override
  _State createState() => _State();
}

class _State extends State<> {
  @override
  Widget build(BuildContext context) {
    return Container();
  }
}
```

**Abbildung 5.12** Ein StatefulWidget wurde aus einer Vorlage erstellt.

Der Cursor müsste bereits an der rot markierten Stelle vor `extends` stehen und erwartet den Bezeichner des neuen Widgets. Tippen Sie »LoginMask«, und drücken Sie die Taste ↵. Haben Sie bemerkt, wie sich mit jedem Tastendruck auch der Inhalt zwischen den spitzen Klammern von `State<>` geändert hat?

Die nächste Haltestelle, die automatisch durch die Vorlage angesprungen werden sollte, ist der Inhalt der `build`-Methode, die Sie schon aus den vorigen Abschnitten kennen.

Im Unterschied zu einem `StatelessWidget` befindet sich beim `StatefulWidget` diese Methode aber nicht im Widget selbst, sondern in der Klasse `State`, die wir noch nicht besprochen haben. In der angesprungenen Methode steht ein Aufruf des Widgets `Container`, den Sie jetzt mit dem Code aus der `build`-Methode aus `MyApp` ersetzen:

```
class _LoginMaskState extends State<LoginMask> {
  @override
  Widget build(BuildContext context) {
    return Center(
      child: Text(
        'Willkommen',
      ),
    );
  }
}
```

**Listing 5.11** Der »LoginMaskState«

Weil sich `Center` und `Text` nun im `_LoginMaskState` befinden, können Sie sie aus `MyApp` entfernen und stattdessen das neue Widget `LoginMask` einsetzen:

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(),
        body: LoginMask(),
      ),
    );
  }
}
```

**Listing 5.12** »MyApp« benutzt die soeben erstellte »LoginMask«.

Der nächste Hot-Reload überträgt die Anpassungen auf das Gerät, und wie Sie sehen, hat es keine visuellen Änderungen gegeben. Alles ist beim Alten, trotz neuen Codes!

**Das Trailing Comma**

Ist Ihnen im Code oben in Listing 5.12 aufgefallen, dass Zeilen, in denen Variablen gesetzt werden, wie etwa `appBar: AppBar()` oder `body: LoginMask()`, mit einem Komma beendet werden?

Der Aufruf von `Scaffold` geht an den Konstruktor dieses Typs und initialisiert über die Named Parameters (siehe Abschnitt 3.6.2) die beiden Variablen `appBar` und `body`. Konstruktoren oder genereller gesagt Funktionen können mit und ohne ein *Trailing Comma* geschrieben werden, also ein Komma, das auf den letzten Parameter folgt.

In den Codebeispielen dieses Buchs werden Sie oft meine Vorliebe für (horizontal) kurze Zeilen bemerken, die der Spalte in einer Zeitung ähneln sollen. Auch heute noch wenden Print-Medien das bewährte visuelle Mittel an, um die Augenbewegungen zu minimieren und so das Lesen zu erleichtern.

Außer mit der Standardzeilenlänge Ihrer IDE, die wahrscheinlich bei 120 Zeichen liegt, können Sie die Länge Ihrer Zeilen mit dem Trailing Comma beeinflussen. Befindet sich hinter dem letzten Parameter oder auch z. B. dem letzten Element einer *Collection* ein Komma, umbricht der *Dart Formatter* automatisch die Zeile und sorgt für einen angenehm linksbündigen Lesefluss.

Den Dart Formatter können Sie in Ihrer jeweiligen IDE über das ganze Projekt oder einzelne Dateien laufen lassen. In Android Studio formatieren Sie die geöffnete Datei, indem Sie in der Menüleiste Code • Reformat Code with dartfmt auswählen. In Visual Studio Code führen Sie einen Rechtsklick auf den Editorbereich der geöffneten Datei aus und wählen Format Document.







### Mit dem Extract Refactoring zum neuen Widget

Bevor wir uns näher mit dem State-Objekt beschäftigen, möchte ich Ihnen die angekündigte zweite Möglichkeit vorstellen, um ein Widget zu erstellen. Wie erwähnt, können Sie ein Widget aus dem Teil des Widget-Trees herausziehen. Somit würden Sie das »stful«-Snippet, mit dem Sie zuvor das `StatefulWidget` Template generiert haben, sparen und gleichzeitig auch das Ausschneiden der zu bewegenden Widgets – hier `Center` und `Text`.

Da Sie zuvor schon eine manuelle Extraktion durchgeführt haben, müssten Sie den vorherigen Zustand noch einmal wiederherstellen, sodass ihre `build`-Methode erneut wie in Listing 5.10 aussieht. Anschließend können Sie mit der zweiten Möglichkeit beginnen.

Die betrachteten IDEs lassen Sie die nötigen Schritte beinahe analog ausführen, hier kommt es nur auf dem Weg zum jeweiligen Kontextmenü zu einem kleinen Unterschied.

Um aus einem Teil des Widget-Trees in Android Studio ein neues Widget zu erstellen, klicken Sie mit der rechten Maustaste auf das Widget, das Sie zusammen mit seinen Nachfolgern extrahieren möchten. In Ihrem Fall ist dies `Center`. Das Kontextmenü, das nun erscheint, enthält den Eintrag Refactoring und in dessen Untermenü finden Sie den Punkt Extract Flutter Widget. Klicken Sie darauf. In dem Dialogfenster, das sich nun öffnet (siehe Abbildung 5.13), tippen Sie den Bezeichner des Ziel-Widgets ein. Verwenden Sie erneut den Namen »LoginMask«.

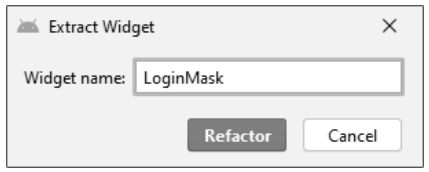


**Abbildung 5.13** Der »Extract Widget«-Dialog

Dem generierten Code sehen Sie an, dass er jedoch kein `StatefulWidget` ist. In der Tat wurde ein `StatelessWidget` generiert, aber die extrahierten Widgets stehen in der `build`-Methode an der richtigen Stelle:

```
class LoginMask extends StatelessWidget {
  const LoginMask({
    Key key,
  }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Center(
      child: Text(
        'Willkommen',
      ),
    );
  }
}
```

**Listing 5.13** Das extrahierte Widget

Aus der Extraktion ist ein sehr ausführlicher Konstruktor des Widgets `LoginMask` hervorgegangen. Das Schlüsselwort `const` (siehe Abschnitt 3.4.2) wurde automatisch eingesetzt, da keine nicht als `final` eingeführten Variablen vorhanden sind (in diesem Fall keinerlei Variablen). Ein Named Parameter (siehe Abschnitt 3.6.2) `Key, key` wurde eingeführt, dessen zugewiesener Wert an die Basisklasse, also `StatelesssWidget`, über den Aufruf von deren Standardkonstruktors (`super`, siehe Abschnitt 3.7.3) weitergereicht wird. Keys werden Sie in Abschnitt 5.5 kennenlernen.

Mit einem weiteren automatisierten Schritt kommen Sie ans Ziel: Setzen Sie den Cursor auf den Bezeichner der entstandenen Klasse `LoginMask`. Mit Alt + ↵ oder dem Wählen des Glühbirnen-Symbols erreichen Sie den Hilfsdialog, der Ihnen die Konvertierung in ein `StatefulWidget` anbietet (siehe Abbildung 5.14). Wählen Sie den Eintrag aus.






**Abbildung 5.14** Der Hilfsdialog mit angebotener Konvertierung



Damit haben Sie den gleichen Stand wie zuvor mit manueller Extraktion erreicht. Erneut wurde ein `StatefulWidget` namens `LoginMask` mit `Center` und `Text` in der `build`-Method erstellt.

Mit Visual Studio Code führen Sie im Grunde die gleichen Schritte aus wie schon mit Android Studio. Der Unterschied liegt im Beginn, wenn Sie den Eintrag zur Extraktion auswählen. In Visual Studio Code befindet sich dieser direkt im Hilfsdialog unter dem erscheinenden Glühbirnen-Symbol (siehe Abbildung 5.15). Außerdem ist er über das Tastenkürzel Strg + . (oder Cmd + . unter macOS) erreichbar.

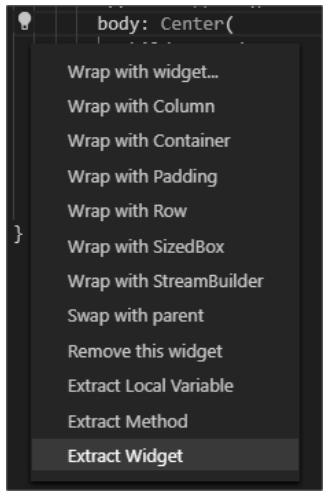


**Abbildung 5.15** Der Hilfsdialog zu »Extract Widget« in Visual Studio Code

Auch mit Visual Studio Code wird zuerst ein `StatelessWidget` erstellt, das Sie anschließend, wie im Beispiel zu Android Studio gesehen, über den Hilfsdialog in ein `StatefulWidget` mit dem Eintrag Convert To StatefulWidget umwandeln.

### Zwei Wege führen ans Ziel

Da Sie nun beide Wege kennengelernt haben, ist ein guter Zeitpunkt gekommen, um zu betrachten, wann Sie welche einsetzen sollten. Frei heraus: Die Entscheidung liegt ganz bei Ihnen. Verwenden Sie, womit Sie sich wohler fühlen. Ich persönlich greife auf das `stful`-Snippet zurück, wenn ich ein neues Widget von Grund auf baue, und auf die

Extraktion, wenn ich bereits auf einen stückweise oder großflächig ausdefinierten Teil des Widget-Trees treffe und diesen in ein eigenes Widget schieben möchte.

### 5.2.3 Das Widget verschieben

Sie haben es zu Ihrem ersten eigenen `StatefulWidget` geschafft! Zunächst bietet es sich an, das Widget in seine eigene Datei zu ziehen. Denn wie Sie in Abschnitt 3.13.1 erfahren haben, sollte je Klasse eine neue Mini-Library erstellt werden, außer es handelt sich um eng zueinander gehörende Klassen. Mit einem `StatefulWidget` und seinem State-Objekt liegt dieser Fall vor.

Markieren Sie den Code, der die Klassen `LoginMask` und `_LoginMaskState` umfasst, und schneiden Sie ihn über das Kontextmenü oder die Tastenkombination Strg + X (Cmd + X unter macOS) aus, damit er in die *Zwischenablage* Ihres Computers gespeichert wird.

Als Nächstes gilt es, eine neue Datei zu erstellen. Hierzu führen Sie einen Rechtsklick auf den Ordner *lib* in der Projektübersicht Ihrer IDE aus, die sich auf der linken Seite befinden sollte (siehe Abbildung 5.16. In dem Kontextmenü wählen Sie in Android Studio über den Eintrag New die Option Dart File aus. Als Namen können Sie zum Beispiel »login_mask« auswählen. Sie erinnern sich bestimmt, dass alle Dateinamen und Pfade in Dart in *snake_case* geschrieben werden (siehe Abschnitt 3.2.1). In Visual Studio Code wählen Sie New File und tippen entsprechend den Namen ein.

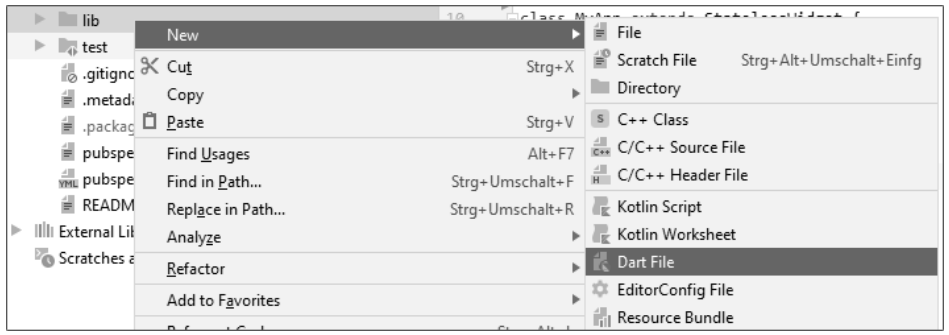


**Abbildung 5.16** Eine neue Datei in Android Studio anlegen

Die neu angelegte Datei wird automatisch geöffnet. Setzen Sie nun den Code aus der Zwischenablage über Strg + V (Cmd + V unter macOS) ein. Einige vom Dart Analyzer gefundene Fehler werden Ihnen aufgezeigt; unter anderem kann die Klasse `StatefulWidget` nicht mehr aufgelöst werden.

Diese Meldungen sind ganz normal, keine Sorge! Bisher haben Sie in der neuen Datei noch keine Flutter-Abhängigkeiten importiert. Wenn Sie zu *main.dart* zurückwechseln, werden Sie sehen, dass sich weit oben die folgende Zeile befindet:





```
import 'package:flutter/material.dart';
```

Diese Import-Anweisung muss ebenfalls in der neuen Mini-Library stehen. Zwar können Sie die Zeile kopieren und einfügen, solche Abhängigkeiten lassen sich mithilfe der IDE aber komfortabler lösen. Wechseln Sie, falls noch nicht geschehen, wieder zurück zur eben erstellten *login_mask.dart*.

Setzen Sie den Cursor auf das unterstrichene `StatefulWidget` neben dem Schlüsselwort `extends`, und wählen Sie über Alt + ↵ in Android Studio oder Strg + . (Cmd + . unter macOS) in Visual Studio Code den Hilfsdialog und darin die Option Import library 'package:flutter/material.dart' aus (siehe Abbildung 5.17). Zur Auswahl stehen daneben auch *cupertino.dart* und *widgets.dart*, die wir in Abschnitt 5.3 besprechen werden.

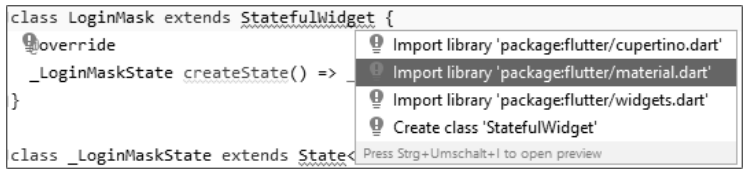


**Abbildung 5.17** Die Import-Anweisung über den Hilfsdialog einfügen

Mit der erfolgten Auswahl wurde am Anfang der Datei der fehlende Import eingefügt. Sofern Symbole wie Klassen oder Funktionen über die in *pubspec.yaml* eingebundenen Packages aufzulösen sind, können Sie `import`-Anweisungen über diese Hilfe auch in anderen Situationen einsetzen. Analog dazu importieren Sie in *main.dart*, aus der Sie das Widget entnommen haben, die neue Datei *login_mask.dart*.

### 5.2.4 Das State-Objekt

Nachdem jetzt alles an seinem Platz ist, beschäftigen wir uns mit dem State-Objekt `_LoginMaskState` und damit, wie es mit dem `StatefulWidget` `LoginMask` verbunden ist.

Zuvor haben Sie bereits gesehen, dass die `build`-Methode vom Widget zum State gewandert ist. Das hat den Grund, dass sie nicht nur beim Einbinden in den Widget-Tree aufgerufen wird, sondern ebenfalls durch andere Ereignisse, etwa interne Aktualisierungen wie den Ereignis-Handler eines Buttons. Außerdem können beide Objekte unterschiedliche Lebenszeiten aufweisen. Widgets werden beispielsweise häufig neu gebaut und anschließend mit dem existierenden State der Vorgängerinstanz wiedervereinigt.

Wenn ein `StatefulWidget` in den Widget-Tree eingefügt wird, ruft die *Runtime* die Methode `createState` auf, die wiederum den Konstruktor des zugehörigen States aufruft:

```
class LoginMask extends StatefulWidget {
  @override
  // Aufruf des State-Konstruktors
  _LoginMaskState createState() => _LoginMaskState();
}

class _LoginMaskState extends State<LoginMask> {
  // …
}
```

**Listing 5.14** Der Konstruktor von »_LoginMaskState« wird aufgerufen.

Wie an einigen zurückliegenden Stellen erwähnt, bietet das State-Objekt einige Callbacks, die den sogenannten *Lifecycle* (Lebenszyklus) eines `StatefulWidget`s betreffen. Im Folgenden möchte ich Ihnen jeweils eine kurze Beschreibung jeder dieser aufgerufenen Methoden geben, die Sie im Verlauf des Buchs im Code einsetzen werden.

### initState

Nicht selten muss oder darf Code nur einmal in der Lebenszeit eines Objekts aufgerufen werden. Für diese Fälle implementiert State den Callback `initState`, der angestoßen wird, sobald der State erstellt wird, etwa durch den Aufruf von `createState` in `StatefulWidget`.

Um auf `initState` aufzubauen, müssen Sie die Methode überschreiben. Hierzu definieren Sie die Methode erneut und fügen die `@override`-*Annotation* ein:

```
@override
void initState() {
  super.initState();

}
```

**Listing 5.15** Überschreiben der Methode »initState«

Bitte vergessen Sie nicht den Aufruf der Methode in der Basisklasse: `State`; rufen Sie entsprechend als Erstes mit dem Schlüsselwort `super` die entsprechende Implementierung auf.

**Autovervollständigung**

Ein großer Vorteil der Arbeit mit dem Editor einer IDE besteht darin, dass Sie die zugehörige Autovervollständigung nutzen können. Um `initState` nicht selbst schreiben zu müssen, tippen Sie die ersten Buchstaben, etwa »init«, in eine leere Zeile in der Klasse `_LoginMaskState`. Daraufhin erscheint der Hilfsdialog und bietet die Überschreibung einiger Methoden an (siehe Abbildung 5.18). Hier wählen Sie aus





aktuellem Anlass den Eintrag INITSTATE() { … }. Anschließend wird Ihnen die Arbeit dankenswerterweise abgenommen. Der generierte Code wird ähnlich wie das Ergebnis in Listing 5.15 aussehen.

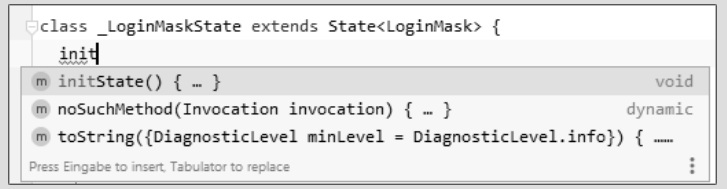


**Abbildung 5.18** Autovervollständigung zum Überschreiben von »initState«



### didChangeDependencies

An dieser Stelle sollen etwaige `InheritedWidget`-basierte Abhängigkeiten benutzt oder verwaltet werden. Diese Kategorie von Widgets lernen Sie in Kapitel 8 kennen. Beispiele dafür sind `Theme.of` oder das aus dem `provider`-Package bekannte `Provider.of`, mit dem Sie Abhängigkeiten in einer Art *Dependency Injection* in Ihre Widgets holen. Wie Sie noch sehen werden, können Sie selbstverständlich eigene `InheritedWidgets` erstellen.

Sobald sich eines dieser über den Widget-Tree aufgelösten Widgets ändert, wird der Callback von `didChangeDependencies` aufgerufen, außerdem geschieht dies, wenn sich die Position des aktuellen Widgets im Tree verändert hat.

Eine Beispielimplementierung von `didChangeDependencies` wäre:

```
@override
void didChangeDependencies() {
  super.didChangeDependencies();

  // Initialisierungen oder Anpassungen
}
```

**Listing 5.16** Die Überschreibung von »didChangeDependencies«

### build

Nach `didChangeDependencies` ist Ihr Widget vollständig in den Widget-Tree integriert und ruft unmittelbar danach die Methode `build` auf. Dies wird nicht die einzige Ausführung bleiben. Anders als beim `StatelessWidget` kann sich ein `StatefulWidget` aus vielerlei Gründen ändern. Animationen (siehe Kapitel 12) werden beispielsweise als `StatefulWidgets` implementiert. Jedes Element einer Animation ruft die `build`-Methode auf. Ebenfalls kann die Methode von Ihnen aufgerufen werden, etwa wenn sich Variablen im State geändert haben, die die Darstellung in der Oberfläche beeinflussen. Einen solchen Durchlauf starten Sie, indem Sie die Funktion `setState` einsetzen.

Ein Beispiel setzen Sie in Abschnitt 5.2.5 ein, um den Status der Login-Schaltfläche zu kontrollieren.

### didUpdateWidget

Ein State-Objekt ist nicht an eine Instanz eines Widgets gebunden. Sie können beispielsweise mithilfe von Keys (siehe Abschnitt 5.5) dafür sorgen, dass States über das Ende der Lebenszeit des zugehörigen Widgets hinaus konserviert und einem anderen Objekt des gleichen Typs zugewiesen werden können.

Den State zwischenzuspeichern, kann etwa beim Implementieren von scrollbaren Widgets sinnvoll sein. Nehmen wir an, ein Display kann nur 20 von 100 Elementen darstellen und der Nutzer der Anwendung scrollt durch die Liste. Er bewegt sich hinab und wieder hinauf, während Widgets erzeugt und abgebaut werden.

Sind die Widgets mit Keys versehen, die wie ein Identifikator fungieren, so weiß die Runtime ein neu erstelltes Widget mit einem vorherigen State zu vereinigen und spart sich somit die Mühe, ein weiteres Objekt zu erstellen und womöglich abermals aufwendige Logik auszuführen.

Der `didUpdateWidget`-Callback wird mit einer Referenz auf das vorherige Objekt aufgerufen, falls vorhanden. Wenn Abhängigkeiten zum alten Widget bestehen, müssen Sie die Verbindungen an dieser Stelle lösen, sodass Ressourcen freigegeben werden können.

```
@override
void didUpdateWidget(LoginMask oldWidget) {
  super.didUpdateWidget(oldWidget);

  // Ressourcen freigeben
}
```

**Listing 5.17** Beispielimplementierung von »didUpdateWidget« in »LoginMask«

### deactivate

Durch den Aufruf von `deactivate` bekommt Ihr Widget die Gelegenheit, darauf zu reagieren, dass es aus dem Widget-Tree entfernt wurde. Falls Referenzen auf Rendering-Objekte (siehe Abschnitt 5.4.3) oder positionsspezifische Eigenschaften abgefragt wurden und sich in Benutzung befinden, sollten Sie den State entsprechend aufräumen:

```
@override
void deactivate() {
  super.deactivate();

  // Aufräumen
}
```

**Listing 5.18** Überschreiben von »deactivate«







### dispose

Wenn ein `StatefulWidget` aus dem *Tree* entnommen, aber nicht wieder eingefügt wurde, ruft die Runtime darauf den Callback `dispose` auf. Hier bietet sich die Gelegenheit, aktuelle Operationen (etwa laufende Animationen, Timer oder Futures) zu beenden und so die Ressourcen freizugeben.

Nachdem `dispose` ausgeführt wurde, wird die Variable `mounted` eines `StatefulWidgets` auf `false` gesetzt. Hierüber können und sollten Sie nach asynchronen Aktivitäten den aktuellen Zustand Ihres Widgets überprüfen, bevor Sie eine Aktualisierung der Oberfläche anstoßen, etwa durch `setState`. Sollte `setState` auf einem Widget aufgerufen werden, dessen `mounted`-Variable den Wert `false` einnimmt, so wird die Anwendung mit einem Fehler reagieren.

```
@override
void dispose() {
  super.dispose();

  // Ressourcen freigeben
}
```

**Listing 5.19** Überschreiben von »dispose«

## 5.2.5 Die LoginMask mit State anreichern

Nachdem wir das State-Objekt und besonders dessen `setState`-Funktion näher betrachtet haben, möchte ich mit Ihnen die `LoginMask` so ausbauen, dass tatsächlich eine kleine Login-Maske entsteht, die, wenn Nutzername und Passwort gesetzt sind, die Login-Schaltfläche durch den Rückgriff auf den internen State und `setState` aktiviert.

### Mit initState initialisieren

Als Erstes werden die Variablen für den Zustand benötigt, die Sie gleich zuoberst innerhalb von `_LoginMaskState` einführen:

```
class _LoginMaskState extends State<LoginMask> {

  // Nimmt die Eingabe aus dem Eingabefeld
  // für den Username auf
  String _username;
```

```
  // Nimmt die Eingabe aus dem Eingabefeld
  // für das Passwort auf
  String _password;

  // Nimmt den Wert true ein, wenn sowohl
  // _username als auch _password nicht leer sind
  bool _canLogin;

// …
```

**Listing 5.20** Die Zustandsvariablen in »LoginMask«

Um den Callback `initState` in Aktion erleben zu können, wird die Initialisierung der soeben eingeführten Variablen dort durchgeführt und es ergibt sich folgende Implementierung:

```
// …

@override
void initState() {
  super.initState();

  _username = '';
  _password = '';
  _canLogin = false;
}
```

**Listing 5.21** »initState« im State von »LoginMask«

### Hintergrundlogik implementieren

Somit haben Sie bereits wichtige Bausteine in dem `StatefulWidget` integriert. Bevor wir die Definition der Oberfläche in `build` betrachten, führen wir noch die beiden Methoden ein, die nach dem Ende der Eingabe von Nutzername und Passwort aufgerufen werden, sowie eine weitere kleine Methode, die eine Überprüfung des Inhalts durchführt und entsprechend den Wert der Variablen `_canLogin` setzt:

```
@override
Widget build(BuildContext context) {
  // …
}

// Setzt _username und lässt anschließend die
// Eingaben prüfen
void _onSubmittedUsername(String username) {
```





```
   _username = username;
   _checkCanLogin();
}

// Setzt _password und lässt anschließend die
// Eingaben prüfen
void _onSubmittedPassword(String password) {
  _password = password;
  _checkCanLogin();
}

// Setzt die Variable _canLogin auf true, wenn
// _username und password jeweils gesetzt wurden
void _checkCanLogin() {
  // setState erwirkt einen neuen Durchlauf
  // der build-Methode
  setState(() {
    _canLogin = _username.isNotEmpty && _password.isNotEmpty;
  });
}
```

**Listing 5.22** Die Logik hinter den Eingabefeldern und dem Button

Wie Sie oben erkennen, erwarten `_onSubmittedUsername` und `_onSubmittedPassword` jeweils einen String, setzen dann die jeweilige Variable und stoßen die Prüfung der Bedingung hinter der Login-Schaltfläche an, die in `_checkCanLogin` durchgeführt wird. Dort ist ein Aufruf an `setState` eingebracht, der die Oberfläche zu einem Durchlauf der `build`-Methode auffordert. Falls die Variable `_canLogin` einen neuen Wert eingenommen hat, wird sich dementsprechend auch der Bildschirm aktualisieren.

### Die Oberfläche der LoginMask

Bis auf den Inhalt der `build`-Methode ist alles vorbereitet. Sehen wir uns also die Widgets an, die eine rudimentäre Login-Maske wie in Abbildung 5.11 abbilden.

Um die Widgets einigermaßen in Form zu bringen, muss ich an dieser Stelle etwas vorgreifen: Der Blick auf den folgenden Code wird das ein oder andere Widget enthüllen, das Sie bisher noch nicht kennengelernt haben. Sie werden diese Widgets allerdings später kennenlernen, speziell in Kapitel 6, »Layouting«, und Kapitel 10, »Mit dem Nutzer interagieren«.

Nun aber zum Code der `build`-Methode:

```
@override
Widget build(BuildContext context) {
  // Auf dem Bildschirm zentrieren
  return Center(
    // Künstlich auf eine feste Breite
    // von 300 logischen Pixeln verringern
    child: SizedBox(
      width: 300,
     // Die Column stapelt die in children
     // übergebenen Listenelemente vertikal
     // untereinander
      child: Column(
        // Die Column soll vertikal nur so viel
        // Höhe einnehmen, wie sie wirklich braucht
        mainAxisSize: MainAxisSize.min,
        // Die untereinander anzuzeigenden Widgets
        // der LoginMask
        children: [
          Text('Willkommen'),
          // Je ein TextField (Eingabefeld) für
          // Benutzername und Passwort
          TextField(onSubmitted: _onSubmittedUsername),
          TextField(onSubmitted: _onSubmittedPassword),
          // Eine rahmenlose (flache) Schaltfläche
          TextButton(
            onPressed: _canLogin ? () {} : null,
            // Auch der Text im Button ist ein Widget
            child: Text('Login'),
          ),
        ],
      ),
    ),
  );
}
```

**Listing 5.23** Die vollständige »build«-Methode

Offensichtlich bedarf es einiger Zeilen Code, um die Login-Maske zu realisieren. Weil sich viele Kommentare darin befinden, erscheint der Code aber umfangreicher, als er eigentlich ist. Gehen wir die Methode einmal von oben nach unten durch.

Das erste Widget ist `Center`, das Sie bereits aus vorherigen Beispielen kennen und das alles Nachfolgende vertikal und horizontal innerhalb des zur Verfügung stehenden Bereichs zentrieren wird.







Der Variable `child` ist das Widget `SizedBox` übergeben worden, mit dem Sie den verfügbaren Raum der nächsten absteigenden Widgets einschränken können. Ich habe es an dieser Stelle eingesetzt, um die Breite der Eingabefelder zu begrenzen, die ansonsten den ganzen Bildschirm einnehmen und eine horizontale Zentrierung erschweren.

Darunter befindet sich eine `Column`, die eine der meistgenutzten *Container* in Flutter ist. Mit ihr und anderen beschäftigen wir uns in Abschnitt 6.1.7. Dieses Widget besitzt keine Variable `child`, sondern den Plural `children`, weil es als `Container` nicht nur ein nachfolgendes Widget, sondern eine beliebige Anzahl von Widgets aufnehmen kann und diese vertikal untereinander anordnet. Die Variable `children` erwartet eine `List` (siehe Abschnitt 3.9.1) aus Widgets. Im Code ist eine dementsprechende Liste aus `Text`, zwei `TextField`s und einem `TextButton` übergeben worden.

Der `Text` ist genauso wie das Widget `Center` schon bekannt. Die `TextField`s haben wir bisher nicht eingesetzt. Das Beispiel beschränkt sich auf den Konstruktor des Eingabefelds und übergibt dort jeweils eine Methode, die aufgerufen wird, sobald der Nutzer die Texteingabe über den Fertig-Button der Tastatur beendet.

Der `TextButton` erhält die Zuweisung zweier Variablen. Als Erstes wird `onPressed` zugewiesen. Dort erkennen Sie den ternären Operator (siehe Abschnitt 3.10.1), der auf der Variablen `_canLogin` des State-Objekts operiert. In Flutter aktivieren Sie einen Button, indem Sie ihm eine Funktion zuweisen, die das Ereignis behandelt. Sollte `onPressed` also mit dem Wert `null` gesetzt werden, ist der Button deaktiviert. Sobald `_canLogin` den Wert `true` einnimmt, wird `TextButton` zu Demonstrationszwecken ein leeres Lambda zugewiesen.

Ebenfalls interessant ist die Zuweisung der Variable `child` des `TextButtons`. Dort wird nicht ein String zugewiesen, der den anzuzeigenden Text der Schaltfläche definieren würde, sondern mit `Text` ein Widget übergeben. Sie können genauso etwas anderes als Text verwenden, etwa ein Icon oder Bild, wie Sie in Abschnitt 10.1, »Button, TextField und Co«, noch sehen werden.

### Die Eingabemaske auf dem Gerät

Jetzt sollten Sie alle Elemente an Ort und Stelle haben, sodass Sie das Ergebnis auf einem Simulator sehen können. Falls Sie das Debugging seit dem letzten funktionierenden Stand in Abschnitt 5.1 nicht beendet haben, sollte ein erneuter Hot-Reload Ihre Änderungen bereits zur Anzeige bringen. Ansonsten starten Sie in Ihrer IDE eine neue Debugsitzung (Session) über das Startsymbol oder Debugsymbol. Letztendlich sollte sich bei Ihnen der Stand aus Abbildung 5.19 zeigen:

Wenn Sie die beiden Eingabefelder ausfüllen, indem Sie jeweils auf sie tippen und über die automatisch ausfahrende Tastatur Zeichen eingeben, sollte sich nach zweifacher Bestätigung des Fertig-Buttons die Login-Schaltfläche als aktiviert darstellen.

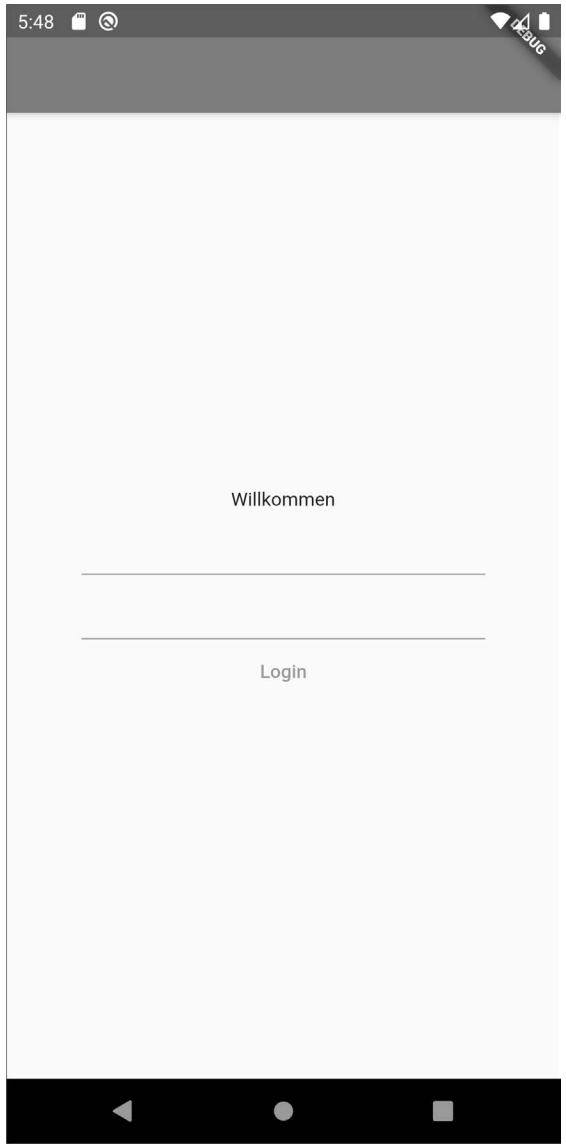


**Abbildung 5.19** Die fertige Login-Maske

## 5.3 Material Design und Cupertino Design

Flutter setzt nicht auf den jeweiligen grafischen Schnittstellen von Android oder iOS auf. Das heißt, wenn Sie ein Widget in Ihrer App darstellen, so ist dies keine *Android View* oder *iOS UIView*. Flutter nutzt stattdessen das Grafikframework *Skia*, eine in C/C++ geschriebene Engine, die Zeichenanweisungen an die *CPU* und *GPU* übermittelt.







Flutter beherrscht deswegen die komplette Oberfläche und ist nicht von Plattformgrenzen eingeschränkt. Das erlaubt es Ihnen, noch während die App debuggt wird, über einen simplen Hot-Reload zwischen Design-Stilen zu wechseln. Die beiden bekanntesten Stile sind aktuell das von Google verwendete Material Design und der von Apple auf i-Devices eingesetzte Stil, den Sie von iPhones oder iPads kennen.

Die bisherigen Beispiele bestanden größtenteils aus Widgets des Material Designs. Nichts hält Sie aber davon ab, Ihre Flutter-Apps im Stil von Apple darzustellen, der Cupertino genannt wird.

Diese beeindruckende Fähigkeit lässt sich anhand des derzeitigen Stands der `LoginMask` demonstrieren. Um die Anwendung, die sich momentan im Material Design zeigt, in eine Cupertino-App zu verwandeln, sind nur wenige Schritte nötig.

Zunächst müssen Sie an den Startpunkt der App zurückkehren, der sich in der Datei *main.dart* befindet. In der `build`-Methode von `MyApp` wird momentan ein Widget vom Typ `MaterialApp` zurückgegeben. Dies gilt es in `CupertinoApp` zu ändern, und gleichzeitig müssen Sie von `Scaffold` zu `CupertinoPageScaffold` wechseln, das allerdings keine Einteilung mehr in `appBar` und `body` vornimmt, sondern nur `child` anbietet. Nach den Änderungen gelangen Sie zum folgenden Stand:

```
@override
Widget build(BuildContext context) {
  return CupertinoApp(
    home: CupertinoPageScaffold(
      child: LoginMask(),
    ),
  );
}
```

**Listing 5.24** Wechsel von »MaterialApp« zu »CupertinoApp«

Weil *main.dart* noch nicht die Datei *cupertino.dart* aus dem Package `flutter` importiert, kann Ihre IDE die Cupertino-Typen nicht auflösen. Setzen Sie den Cursor auf einen der unterstrichenen Typen (siehe Abbildung 5.20), und öffnen Sie den Hilfsdialog über das Glühlampen-Symbol oder die jeweilige Tastenkombination. Sie ist in Android Studio Alt + ↵ und in Visual Studio Code Strg + . (Cmd + . unter macOS). Dort wählen Sie den Eintrag Import library 'package:flutter/cupertino.dart'.

Nach dem nächsten Hot-Reload wirft Ihnen Flutter einen Fehler entgegen und taucht den Hintergrund der App in Rot. Die Meldung besagt No Material widget found. Weil die Wurzel des Widget-Trees jetzt eine `CupertinoApp` mit entsprechendem `CupertinoPageScaffold` ist, fehlen einigen Widgets Ihrer App die Material Design Theme-Informationen.


**Abbildung 5.20** Im Hilfsdialog den Import »cupertino.dart« wählen

Flutter bietet einige Widgets, die im Kontext von Material Design und Cupertino verwendet werden können, etwa `Text`. Viele Widgets sind allerdings nur in dem einen oder anderen Stil verfügbar oder explizit für ihn geschrieben worden.

Um sich einen Überblick darüber zu verschaffen, welche Widgets in Ihrer `LoginMask` Bestandteile des Material Designs sind, wechseln Sie in die Datei *login_mask.dart*. Ändern Sie den Import in Zeile 1 von `material.dart` zu `widgets.dart`:

```
// Von Material...
import 'package:flutter/material.dart';

// ...zu Widget
import 'package:flutter/widgets.dart';
```

**Listing 5.25** Änderung der Import-Anweisung von »material« zu »widgets«

In Ihrer `build`-Methode von `_LoginMaskState` erkennen Sie jetzt, dass `Center`, `SizedBox`, `Column` und `Text` in beiden Stilen verwendet werden können, aber die Widgets `TextField` und `TextButton` aus *material.dart*, also dem Material Design gekommen sind.

Viele Widgets, die keine visuelle Repräsentation besitzen, die einem Stil entsprechen müsste, finden Sie also in *widgets.dart*. Jene Elemente hingegen, die vom jeweiligen Stil beeinflusst werden (etwa Buttons oder Widgets, die überhaupt nur in einem der beiden definiert werden), müssen bei einem Wechsel zwischen den Designwelten vollständig ausgetauscht werden.

Folglich müssen Sie für `TextField` und `TextButton` Pendants aus *cupertino.dart* verwenden. Tauschen Sie die Widgets mit `CupertinoTextField` und `CupertinoButton` aus. Denken Sie daran, dass auch in dieser Datei der Import von *cupertino.dart* erfolgen muss, sodass die Typen aufgelöst werden können.

Nach der Änderung sollte Ihre `build`-Methode folgendermaßen aussehen:

```
@override
Widget build(BuildContext context) {
  return Center(
    child: SizedBox(
      width: 300,
      child: Column(
```





```
        mainAxisSize: MainAxisSize.min,
        children: [
          Text('Willkommen'),
          CupertinoTextField(onSubmitted: _onSubmittedUsername),
          CupertinoTextField(onSubmitted: _onSubmittedPassword),
          CupertinoButton(
            onPressed: _canLogin ? () {} : null,
            child: Text('Login'),
          ),
        ],
      ),
    ),
  );
}
```

**Listing 5.26** »LoginMask« nach dem Wechsel zum Cupertino-Stil

Führen Sie jetzt einen Hot-Reload aus. Die Fehlermeldung sollte verschwunden und der visuelle Stil verändert sein (siehe Abbildung 5.21). Beachten Sie, wie sogar die Elemente für das Kopieren und Einfügen geändert wurden, obwohl sie eigentlich vom Betriebssystem selbst kommen. Mit Flutter sieht Ihre App auch auf Android wie eine waschechte iOS-App aus.

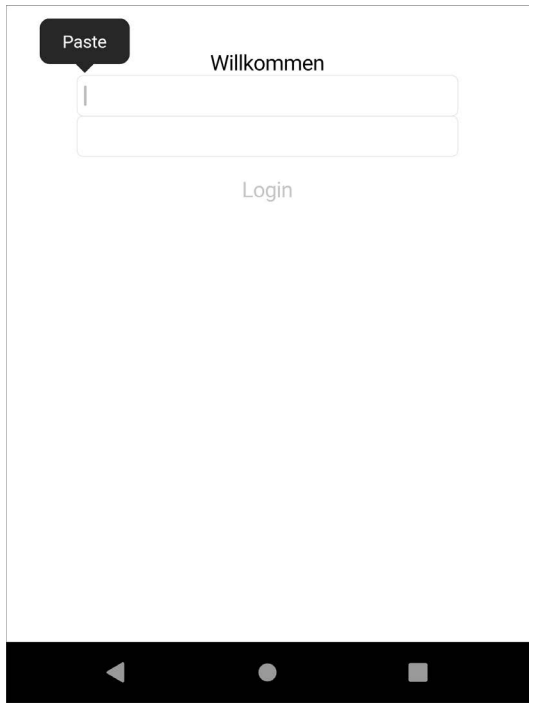


**Abbildung 5.21** Selbst das Kopieren sieht so aus wie unter iOS.

## 5.4 Vom Widget zum Rendering Object

Sie haben in den letzten Kapiteln Ihre ersten Erfahrungen mit Widgets in Flutter gemacht und aus ihnen ein `StatelessWidget` (`MyApp`) und ein `StatefulWidget` (`LoginMask`) zusammengesetzt. Bisher haben wir den Weg eines Widgets von der Definition zur Anzeige auf dem Bildschirm nur bis zur `build`-Methode verfolgt. In diesem Abschnitt möchte ich Ihnen einen Einblick in den Renderingprozess des Frameworks bieten und erklären, was es mit dem *Element-Tree* sowie dem *RenderObject-Tree* auf sich hat.

### 5.4.1 Widget-Tree

Der Begriff *Widget-Tree* ist zuvor einige Male im Zusammenhang mit ineinander verschachtelten Widgets gefallen. Er beginnt mit dem ersten Widget, das der Runtime über die Funktion `runApp` übergeben wird und besitzt wie ein Baum ein oder mehrere Enden, die sich, der Analogie folgend, nach Belieben verästeln und eine große Zahl Blätter hervorbringen können (siehe Abbildung 5.22).

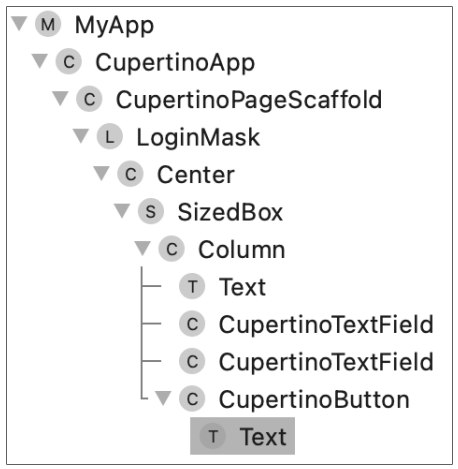


**Abbildung 5.22** Der Widget-Tree zur »LoginMask« im Cupertino-Stil

Der Widget-Tree kann in der Realität durchaus tiefer sein, als es im Code den Eindruck macht. Widgets, die in Flutter angeboten werden, könnten sich als einzelnes Widget darstellen, aber intern aus mehreren anderen Widgets bestehen und so den Tree vergrößern. Mit der `LoginMask` haben Sie zuvor ein eigenes Beispiel implementiert.

Wenn Sie die Datei *main.dart* betrachten, sehen Sie, dass Sie die Wurzel des Widget-Trees mit `MyApp` setzen. `MyApp` ist ein `StatelessWidget`, das aus `CupertinoApp`, gefolgt von `CupertinoPageScaffold` und letztlich `LoginMask` besteht.

Die `LoginMask` erscheint aus Sicht der *main.dart* als einzelnes Widget, wie Sie aber wissen und in Abbildung 5.22 bestätigt sehen, definiert es eine Vielzahl weiterer Widgets.





### 5.4.2 Element-Tree

Elemente, die den *Element-Tree* bilden, sind Ihnen an mancher Stelle schon begegnet. Sie haben sich allerdings hinter dem Namen `BuildContext` versteckt, der als Interface eingeführt wurde, sodass Sie nicht unmittelbar mit dem Konzept *Element* arbeiten.

Widgets sind unveränderlich (*immutable*). Sobald sich eine Änderung ergibt, etwa in der `LoginMask` durch die Anpassung des Werts »Willkommen« zu »Herzlich Willkommen« in `Text`, wird ein neues Widget erzeugt, das den aktuellen Zustand (State) abbildet.

Die nicht veränderbaren Widgets bringen zwar Leistungsverbesserungen in der Handhabung durch Flutter mit sich. Um nach einer solchen Änderung aber nicht den gesamten Tree erneut aufbauen zu müssen, benötigt Flutter einen Cache, der das letzte Abbild, also den letzten State, zwischenspeichert.

Diesen Cache stellt der Element-Tree dar, indem er für jedes Widget im Widget-Tree ein Element im Element-Tree einfügt. Werden Änderungen erkannt, verwendet Flutter die unveränderten Elemente erneut und aktualisiert lediglich die veränderten. Diese Strategie sorgt dafür, dass Flutter bei Teiländerungen nur Teile des Bildschirms neu zu zeichnen hat.

Kommen wir noch mal zum Fall einer Änderung des Widgets `Text`: Beim Vergleich mit dem Element-Tree wird dann erkannt, dass sich das `Text`-Widget geändert hat, aber der `runtimeType` ist noch der gleiche und führt dazu, dass eine Anpassung des entsprechenden *RenderObjects* (siehe Abschnitt 5.4.3) ausgeführt wird – in diesem Fall ein neuer Textinhalt. Hätten Sie statt eines geänderten Inhalts für `Text` ein ganz anderes Widget eingefügt, also mit abweichendem `runtimeType`, so wäre auch das Element im Tree zu ersetzen gewesen.

Flutter definiert zwei verschiedene Elemente: das `ComponentElement` und das `RenderObjectElement`. Ersteres sind jene Elemente, die keinen Einfluss auf *Layout* oder die *Renderphase* ausüben, etwa die `LoginMask` oder der in Flutter definierte `Container`, und die sich in Ihren `build`-Methoden aus anderen Widgets zusammensetzen. Letzteres repräsentiert ein Element, das aktiv am Layout oder Rendering teilnimmt, so zum Beispiel die Elemente der `Column` aus Listing 5.23, die ihre `children` gemäß Layout anordnet.

Was denken Sie? Ist das zugeordnete Element des Widgets `Text`, das Sie in `LoginMask` eingesetzt haben, ein `ComponentElement` oder ein `RenderObjectElement`? Wenn Sie einen Blick in die Definition von `Text` werfen, sehen Sie, dass es in seiner `build`-Methode das Widget `RichText` zurückgibt. `Text` ist demnach ein `ComponentElement`, weil es die Hülle für ein oder mehrere andere Widgets darstellt. Folgen Sie dem `RichText`-Widget zu dessen Definition.

**Die Definition eines Symbols finden**

In den beiden vorgestellten IDEs Visual Studio Code und Android Studio können Sie zum jeweiligen Definitionsort eines Symbols (d. h. einer Klasse, Funktion etc.) über eine Maus- und Tastenkombination oder auch mit einer reinen Tastenkombination springen.

Mit gedrückter Strg-Taste (Cmd unter macOS) und dem Linksklick Ihrer Maus auf ein Symbol springen Sie in beiden Entwicklungsumgebungen zum Ort der Definition, wenn er über die zur Verfügung stehenden Informationen aufgelöst werden kann.

Ausschließlich über die Tastatur kommen Sie mit Strg + B (Cmd + B unter macOS) in Android Studio und mit F12 in Visual Studio Code zum Ziel.

Zunächst fällt auf, dass `RichText` weder von `StatelessWidget` noch von `StatefulWidget` erbt, sondern von einer Klasse namens `MultiChildRenderObjectWidget`:

```
class RichText extends MultiChildRenderObjectWidget {
  // …
}
```

**Listing 5.27** »RichText« mit »extends«-Anweisung

Weiter finden Sie dort keine `build`-Methode vor, sondern `createRendererObject` und `updateRendererObject`. Spätestens an diesen beiden Eigenheiten können Sie zwischen `ComponentElement` und `RenderObjectElement` unterscheiden.

Mit dem Element-Tree werden Sie wahrscheinlich nur in seltensten Fällen in Kontakt kommen, etwa dann, wenn Sie aus der abstrakten Welt der Widgets in die der eigentlichen Rendering-Objekte vorstoßen müssen. Das könnte der Fall sein, wenn Sie die exakte Größe eines für ein Widget erstellten Render-Objekts brauchen, um ein anderes daran zu orientieren.

### 5.4.3 RenderObject-Tree

Im vorigen Abschnitt haben Sie den Unterschied zwischen `ComponentElements`, die als Widget-Container dienen, und den `RenderObjectElements` kennengelernt. Letztere heißen bereits nach dem im Folgenden betrachteten Gegenstand: dem `RenderObject`. Das `RenderObjectElement` ist also das Element, das von einem Widget zu einem tatsächlich gezeichneten Objekt führt.

Das *RenderObject* ist in Flutter eine äußerst abstrakte Idee, die nicht einmal ein *Koordinatensystem* definiert, das erst durch die von *RenderObject* abgeleitete *RenderBox* eingeführt wird. Diese *RenderBox* liefert den Grundbaustein für den *Layoutprozess* in Flutter. Darauf wiederum bauen spezialisierte Objekte auf, zum Beispiel *RenderParagraph* (Text), *RenderFlex* (beispielsweise eine `Column`, wie sie in Listing 5.23 eingesetzt wurde) oder *RenderImage*.





Flutter rechnet in einem einzigen Durchgang die Positionen und Größen von allen *RenderBox*-Objekten durch. Dabei beginnt es ganz oben im *RenderObject-Tree*. Das erste Objekt entspricht dem ersten des Element-Trees, das wiederum das erste Widget im Widget-Tree repräsentiert. Mit der RenderBox wird außer dem *kartesischen Koordinatensystem* ein sogenanntes *Box-Constraint-Model* eingeführt. Ein *Constraint* definiert die maximale und minimale Höhe und Breite des Objekts, dem es zugewiesen wird. Innerhalb des Trees reichen alle Eltern-Objekte jeweils ihren Kind-Objekten *Constraints* weiter. Diese berechnen damit ihre Größe innerhalb dieses Constraints und melden das Ergebnis wieder nach oben zurück.

Allgemein gilt, dass Constraints von oben nach unten und die Größen von unten nach oben gereicht werden. Nach einem Layoutdurchgang besitzen alle RenderBoxen, also RenderObjects, eine errechnete Größe, die innerhalb der Constraints ihrer jeweiligen Eltern-RenderBox liegt. Der RenderObject-Tree ist in diesem Moment bereit für das Rendering. Hierbei wird auf jedem Objekt die Methode `paint` aufgerufen, und das oberste RenderObject setzt den Baum zu einer *Szene* (*Scene*) zusammen, die an die *GPU* übergeben wird.

Die Grundmechanik des RenderObject-Trees, speziell die der Constraints, wird Ihnen in Kapitel 6 zum Layout von Widgets wieder begegnen.

## 5.5 Keys

Mit *Keys* (Schlüsseln) wird in Flutter nichts verschlüsselt, sondern sie dienen als *Identifier* (ID) für Widgets. Wenn Sie ein Widget von einem anderen Widget aus referenzieren wollen, so können Sie es mit einem Key erreichen und auf das verlangte Widget und seinen BuildContext (*Element*) zugreifen.

Durch den Einsatz eines Keys machen Sie ein Widget demnach identifizierbar. Dies trifft auch für die Engine hinter Flutter zu; die Zuordnung von Keys an Widgets macht diese über beliebig viele Frames hinweg nachverfolgbar. Wegen der Anwendung als ID muss ein Key in seinem Gültigkeitsbereich immer einzigartig sein und darf nicht doppelt vorkommen.

### 5.5.1 Widgets referenzieren

Denken Sie an die `LoginMask` zurück, die wir in Abschnitt 5.2 gebaut haben. Dieses Widget wird in *main.dart* innerhalb von `MyApp` benutzt. Sie könnten über einen Key von `LoginMask` auf `MyApp` zugreifen und zur reinen Demonstration die Variable `runtimeType` der `AppBar` ausgeben, die dem `Scaffold` übergeben wurde.

Zuerst müssen Sie einen geeigneten Key erstellen, der in `MyApp` zugewiesen wird. Ähnlich einer globalen Variablen in der Programmierung existiert für den globalen Zugriff der `GlobalKey`:

```
class MyApp extends StatelessWidget {

  // Einführung von appBarKey als globale Widget-ID
  static GlobalKey appBarKey = GlobalKey();

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        // das Zielwidget is die AppBar
        appBar: AppBar(key: appBarKey),
        body: LoginMask(),
      ),
    );
  }
}
```

**Listing 5.28** Einen »GlobalKey« zuweisen

Damit auf die Variable `appBarKey` der Klasse `MyApp` von außen zugegriffen werden kann, ist das Feld als statisch deklariert. Als Nächstes ist die `LoginMask` so anzupassen, dass bei einem Tap auf den Login-Button statt des Texts »Willkommen« die Ausgabe des `runtimeType`s aus `MyApp` erfolgt. Zur Aktualisierung wird außerdem `setState` verwendet, das Sie aus Abschnitt 5.2.5 kennen.

Wir benötigen eine neue Variable mit dem Bezeichner `_headline`, die der `LoginMask` hinzugefügt wird, dann in `initState` mit dem String »Willkommen« initialisiert und schließlich in der `build`-Methode benutzt wird:

```
class _LoginMaskState extends State<LoginMask> {

  // … andere Variablen

  String _headline;

  @override
  void initState() {
    super.initState();
```







```
    _username = '';
    _password = '';
    _canLogin = false;
    // Initialisierung der neuen Variablen
    _headline = 'Willkommen';
  }
  // …

  @override
  Widget build(BuildContext context) {
    return Center(
      child: SizedBox(
        width: 300,
        child: Column(
          mainAxisSize: MainAxisSize.min,
          children: [
            // Das String-Literal wird durch die
            // Variable _headline ausgetauscht
            Text(_headline),
    // …
    );
  }

  // …
}
```

**Listing 5.29** Die neue Variable »_headline«

Führen Sie jetzt einen Hot-Reload aus. Der Bildschirm auf Ihrem Simulator wird sich rot verfärben und melden, dass die Variable `_headline` nicht initialisiert wurde. An dieser Stelle sehen Sie, dass `initState` tatsächlich bloß einmal aufgerufen wird, und zwar, wenn das Widget zum ersten Mal in den Baum eingehängt wird. Ein Hot-Reload sorgt nur für ein Update der Oberfläche, setzt sie allerdings nicht zurück.

Um die App einem Neustart zu unterziehen, ohne das Debugging zu beenden, führen Sie einen *Hot-Restart* aus. Dieser befindet sich im Debugmenü Ihrer jeweiligen IDE, dargestellt durch ein Refresh-Icon gleich neben dem Blitz für den Hot-Reload. In Visual Studio Code werden Sie ihn unmittelbar in der Debug-Palette finden und in Android Studio im Debug-Tool-View (siehe Abbildung 5.23), das Sie, falls es nicht geöffnet sein sollte, mit Strg + 5 (Cmd + 5 unter macOS) öffnen.


**Abbildung 5.23** Die Debug-View in Android Studio

Führen Sie einen Hot-Restart aus. Der Fehler ist jetzt durch einen Neustart der App beseitigt, der nicht mehr als wenige Millisekunden benötigte. Damit fehlt zum Abschluss des kleinen Experiments nur noch der Zugriff auf den `GlobalKey` und das Setzen der Variablen `_headline`, sodass das Ergebnis – der Wert der `runtimeType`-Variablen – in der Oberfläche angezeigt wird. Passen Sie ein weiteres Mal wie folgt die `build`-Methode an:

```
@override
Widget build(BuildContext context) {
  return Center(
    child: SizedBox(
      width: 300,
      child: Column(
        mainAxisSize: MainAxisSize.min,
        children: [
          Text(_headline),
          TextField(onSubmitted: _onSubmittedUsername),
          TextField(onSubmitted: _onSubmittedPassword),
          TextButton(
            // Weiterhin der ternäre Operator mit _canLogin.
            // Falls erfüllt, dann Ausführen von setState
            onPressed: _canLogin ? () {
              setState(() {
                // Der GlobalKey wird über die statische Variable
                // MyApp.appBarKey aufgelöst
                _headline = MyApp.appBarKey.currentWidget.runtimeType
                  .toString();
              });
            } : null,
            child: Text('Login'),
          ),
        ],
      ),
    ),
  );
```

**Listing 5.30** Der »GlobalKey« aus »MyApp« wird benutzt.







Im oberen Code sehen Sie, wie über `MyApp.appBarKey` der `GlobalKey` aufgelöst wird. Über dessen Variable `currentWidget` bekommen Sie die Referenz zum Widget, dem der `Key` zugeordnet wird. Damit haben Sie von einem Widget auf ein anderes zugreifen können und lesen zum Beweis den `runtimeType` aus, der anschließend der neu eingeführten Variablen `_headline` der `LoginMask` zugeordnet wird. Der Aufruf an `setState` signalisiert eine Aktualisierung des Widgets.

Setzen Sie wie in Abschnitt 5.2.5 die beiden Eingabefelder, sodass der Login-Button freigeschaltet wird. Drücken Sie ihn anschließend. In der Oberfläche sollten Sie statt Willkommen jetzt wie in Abbildung 5.24 den String AppBar sehen.

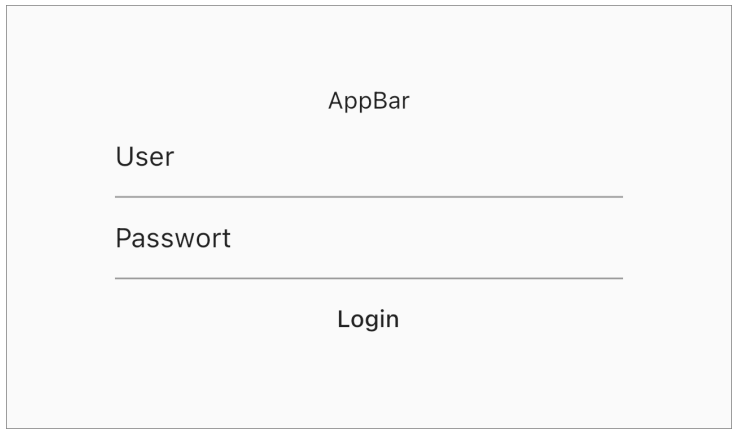


**Abbildung 5.24** Die App nach Tap auf den »Login«-Button

In diesem kleinen Beispiel haben Sie erfolgreich auf ein beliebiges Widget über einen zugewiesenen `GlobalKey` zugreifen können.

### 5.5.2 Widgets markieren

In Abschnitt 5.4.2 über den Element-Tree habe ich erklärt, wie er ein Abbild des Widget-Trees darstellt und wie Flutter wegen des Caches nur jene Widgets aktualisieren muss, die sich verändert haben.

Keys werden im Fall von Widget-*Containern* interessant, falls die Reihenfolge der eingefügten Objekte geändert wird. Stellen Sie sich vor, Sie hätten einen solchen Container, etwa eine `Column` (siehe Abschnitt 6.1.7), in folgendem Ausgangszustand:

```
// …
// a und b besitzen den
// ausgedachten runtimeType ExampleWidget
final a = ExampleWidget();
final b = ExampleWidget();
children: [
```

```
  a,
  b,
]
```

Nun wird die Reihenfolge geändert, sodass:

```
// …
children: [
  b,
  a,
]
```

Nach wie vor liegen auf beiden Plätzen Widgets mit den gleichen `runtimeType`s wie zuvor. Flutter erkennt dementsprechend nicht, dass das erste Element im Tree nicht mehr `a`, sondern `b` sein muss. Der Widget-Tree hat sich verändert, der Element-Tree jedoch nicht.

Um Flutter ein weiteres Kriterium zur Unterscheidung bereitzustellen, können Sie Keys einsetzen. Beim Vergleich zwischen dem Widget- und dem Element-Tree verwendet die Engine nicht nur den `runtimeType`, sondern ebenfalls die Eigenschaft `key` eines Widgets.

In Listing 5.13, als Sie die `LoginMask` aus `MyApp` extrahiert haben, wurde von der IDE automatisch ein ausführlicher Konstruktor generiert. Darin wurde auch die Variable `key` von `super` definiert. Wenn wir dies auf das gedachte Widget `ExampleWidget` übertragen, so ergibt sich:

```
class ExampleWidget extends StatelessWidget {
  const ExampleWidget ({
    Key key,
  }) : super(key: key);

  // …
}
```

Dementsprechend würde die Initialisierung der beiden Objekte `a` und `b` geändert in:

```
// …
// a und b besitzen den
// ausgedachten runtimeType ExampleWidget
final a = ExampleWidget(key: UniqueKey());
final b = ExampleWidget(key: UniqueKey());
children: [
  a,
  b,
]
```





Das obige Beispiel benutzt den `UniqueKey`, der einen Key mit nicht wiederkehrendem Wert (ähnlich einer Zufallszahl) darstellt. Jede Instanziierung unterscheidet sich also von der anderen. Er gehört zur Gruppe der `LocalKey`-Klassen. Andere Vertreter sind der `ObjectKey`, dem ein Objekt übergeben wird, mit dem es den Schlüsselwert erzeugt, und der `ValueKey`, dem ein beliebiger Wert als Schlüssel übergeben wird. `LocalKey`s müssen – anders als die `GlobalKeys` – nur innerhalb des eigenen Teilbaums einzigartig (*unique*) sein. Sie erinnern sich: `GlobalKey`s müssen über die ganze App hinweg unique sein.

Wenn Sie nach dieser Anpassung eine neue Sortierung durchführen, so erkennt Flutter jetzt, dass der `runtimeType` zwar der gleiche, der Key aber ein anderer ist, und sortiert auch die Elemente im Element-Tree und damit die RenderObjects korrekt.



**Widgets zum Neubauen zwingen**

Wenn Sie einmal das Problem haben sollten, dass Ihr Widget neu erstellt werden müsste, Flutter es aber nicht aktualisiert, obwohl die `build`-Methode durchlaufen wird, dann können Sie einen `UniqueKey` einsetzen! Den setzen Sie bei jedem Durchlauf mit einem Zufallswert oder Ähnlichem, sodass Flutter wie oben beschrieben zwar ein gleiches Widget am Platz, aber einen anderen Key vorfindet und dementsprechend das Widget ersetzt.

Eine solche Situation könnte beispielsweise auftreten, wenn das betroffene Widget innerhalb der `build`-Methode nicht unmittelbar mit Daten gefüllt wird, sondern nachgelagert Informationen (etwa über einen Callback) bezieht, die Ergebnisse intern im Speicher ablegt, daraufhin aber nicht seine Oberfläche aktualisiert.

## 5.6 Zusammenfassung

Sie haben Ihre ersten Erfahrungen mit einem eigenen Widget gesammelt und dabei die unterschiedlichen Aspekte von `StatelessWidget` und `StatefulWidget` kennengelernt – dem Fundament einer jeden Flutter-App. Sobald Sie in Ihrem Widget einen Zustand abbilden müssen, greifen Sie zu einem `StatefulWidget` und konfigurieren Ihr State-Objekt entsprechend Ihren Anforderungen. Bei statischen Informationen wie Texten oder Bildern setzen Sie das `StatelessWidget` ein, das Ihnen Leistungsverbesserungen durch *Caching* ermöglicht. Mit Keys erschließen Sie sich eine weitere Möglichkeit, um Verzögerungen bei der Darstellung zu vermeiden, und können darüber hinaus auf andere Widgets innerhalb des Widget-Trees zugreifen.

Dass Sie die verschiedenen Design-Stile miteinander kombinieren und mit Leichtigkeit vom einen zum anderen wechseln können, haben Sie anhand der in den letzten Abschnitten bearbeiteten Login-Maske ausprobiert. Sie wissen nun, dass einige der Widgets wie Container, Text oder SizedBox als übergreifende Widgets in beiden Stilen

benutzt werden können. Sollten Sie ein Element aus dem Material Design- oder Cupertino-Package benötigen, lösen Sie es über den entsprechenden Import auf.

Mit dem Element-Tree und dem RenderObject-Tree haben Sie die Zwischenschritte kennengelernt, die vom Widget bis zur Darstellung der Elemente auf dem Bildschirm durchlaufen werden.