Kurze Übersicht zu Dart

Inhalt

Vorbemerkung 3

I. Einleitung 3

1. Was ist Dart ? 3
2. DartPad 3

b) lokaler Computer *Fehler! Textmarke nicht definiert.*

3. Das Main-Konzept 3

II. Variablen, Kommentare und Data Typen 4

1. Kommentare 4
2. Datentypen 4
3. Grundlegende Dart-Typen 4
4. Das dynamische Schlüsselwort 5
5. Boolesche Werte 5
6. Enum 5
7. Tpyzuweisung mit „of“ 6
8. Sondertypen 6

III. Operatoren 6

1. Arithmetische Operatoren 6
2. Gleichheits-Operatoren 6

Vergleichs-Operatoren 6

3. Typ-Test-Operator 7
4. Logische Operatoren 7
5. Zuweisungs-Operator 7
6. Ternärer Operator 7

a) Bedingung ? expr1 : expr2 7

b) expr1 ?? expr2 7

IV. Strings 7

2. Runen 8

IV. Unveränderlichkeit (Immutability) . 8

V. Nullability 9

1. Null-fähiger Operator ?? 9
2. Der ?. Operator 9

VI. Kontroll Strukturen 9

1. Bedingungen 9

a) If-Aussagen 10

b) Sonst-Aussagen 10

2. While-Schleifen 10

a) Testen der While-Schleife 10

b) Ausprobieren der Do-While-Schleife 10

c) continue und break 10

3. For-Schleifen 11
4. for each 11
5. switch – case 11

VII. Sammlungen (Collections) 12

1. Listen 12

a) Arbeiten mit Listenelementen 12

2. Maps 12
3. Record 13
4. Iterable 14

VIII. Funktionen 14

1. Funktionen definieren 14
2. Arbeiten mit Funktionen 14
3. Verschachtelungsfunktionen (Nesting Functions) 14
4. Optionale Parameter 15
5. Benannte Parameter und Standardwerte 15
6. Anonyme Funktionen 15
7. Verwendung anonymer Funktionen 16

IX. Klassen (OOP) 16

1. Definition 16
2. Konstruktor 17
3. this 17
4. Getter und Setter 17
5. Vererbung 17

a) extends 17

b) implements 17

c) with (Mixins) 17

6. Generika 17

X. Typdef 18

XI. Kurzschreibweisen 18

1. Lambda 18
2. Fat Arrow 18

XII. Ausnahmebehandlungen 18

1. Try-Catch 18
2. Eigene Ausnahmen 18

XIII. Datum/Zeit 18

XIV. Pakete 18

XV. Debugger 18

1. Assert....18

XVI. Asynchrone Programmierung.....18

1. Await und future....18
2. Streams....18

XVII. Dateihandling.....19

1. Datei in Dart schreiben....19

a) Einleitung*....*19

b) Datei in Dart schreiben*....*19

c) Fügen Sie neue Inhalte zu vorherigen Inhalten hinzu*....*19

d) CSV-Datei in Dart schreiben*....*19

2. Eine Datei in Dart einlesen....20

XVIII. Isolate 20

XIX. – leer- 20

XX. Weiterführende Tipps.....20

Vorbemerkung

Bei Programmieren brauchte ich als Anfänger eine kurze Übersicht, auf der ich schnell etwas zur Syntax von Dart nachlesen kann. Zu diesem Zweck habe ich den nachfolgenden Text erstellt und verwende ihn – old-school - in ausgedruckter Form stets beim Programmieren 😉.

Die Basis für diese kurze Übersicht zu Dart war ein Artikel von Jonathan Sande(updated) und Joe Howard. Ihr findet diesen auf https://www.kodeco.com/22685966-dart-basics. Die Ausführungen zum Lesen und Schreiben von Dateien bei XVII. stammen von der Internetseite https://dart-tutorial.com/file-handling-in-dart/write-file-in-dart/. Dort finden sich auch viele weitere Ausführungen zu Dart.

I. Einleitung

1. Was ist Dart ?

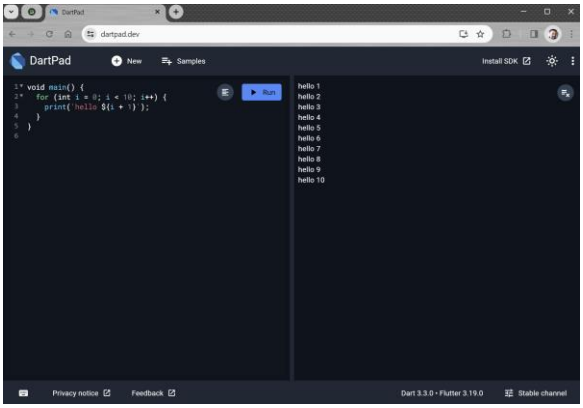
Dart hat viele Ähnlichkeiten mit anderen Sprachen wie Java, C#, Swift und Kotlin. Zu den Funktionen gehören:

1. Statisch typisiert
2. Typinferenz
3. Zeichenfolgen-Ausdrücke (Strings)
4. Multi-Paradigma mit objektorientierter und funktionaler Programmierung
5. Null-sicher (null-safety, Nullability)

Dart ist für die Entwicklung schneller Apps auf einer Vielzahl von Plattformen optimiert.

2. DartPad

Um schnell mit Dart loszulegen, verwenden am besten das Open-Source-Tool DartPad über die Adresse *https://dartpad.dev/*, mit dem Sie Dart-Code über einen Webbrowser schreiben und testen können. Es macht auch immer Sinn zum Testen einzelner Befehle.



DartPad ist wie eine typische IDE aufgebaut. Es umfasst die folgenden Komponenten:

- *Editor-Bereich*: Befindet sich auf der linken Seite. Ihr Code wird hier angezeigt.
- *Schaltfläche "AUSFÜHREN"*: Führt Code im Editor aus.
- *Konsole*: Befindet sich oben rechts und zeigt die Ausgabe an.
- *Dokumentationsbereich*: Dieser befindet sich unten rechts und zeigt Informationen zum Code an.
- *Beispiele*: In dieser Dropdown-Liste finden Sie einige Beispielcodes.
- *Versionsinformationen*: Ganz unten rechts zeigt DartPad an, welche Versionen von Flutter und Dart derzeit verwendet werden.

Wenn Sie möchten, können Sie das Dart SDK **lokal** auf Ihrem **Computer** installieren. Eine Möglichkeit hierfür ist die Installation des Flutter SDK. Durch die Installation von Flutter wird auch das Dart SDK installiert. Um das Dart SDK direkt zu installieren, besuchen Sie https://dart.dev/get-dart.

3. Das Main-Konzept

Dart-Programme beginnen mit einem Aufruf von `main`. Die Syntax von Dart für `main` ähnelt der anderer Sprachen wie C, Swift oder Kotlin.

```
void main() {
}
```

`main` ist eine `Funktion` (siehe auch VIII.) In diesem Fall wird `void` für den Rückgabewert angegeben, was bedeutet, dass `main` nichts zurückgibt.
Die Klammern nach `main` geben an, dass es sich um eine Funktionsdefinition handelt. Die geschweiften Klammern enthalten den Programmcode (body) der Funktion. In `main` fügen Sie daher den Dart-Code für Ihr Programm hinzu.

II. Variablen, Kommentare und Data Typen

Das erste, was Sie zu *main* hinzufügen, ist eine Variablenzuweisungsanweisung. *Variablen* enthalten die Daten, mit denen das Programm arbeitet.

Sie können sich eine Variable als ein Schublade im Arbeitsspeicher Ihres Computers vorstellen, die einen Wert enthält. Jede Schublade hat einen Namen, der der Name der Variablen ist. Um eine Variable mit Dart zu kennzeichnen, verwenden Sie das Schlüsselwort `var` .

Fügen Sie eine neue Variable zu `main` hinzu:

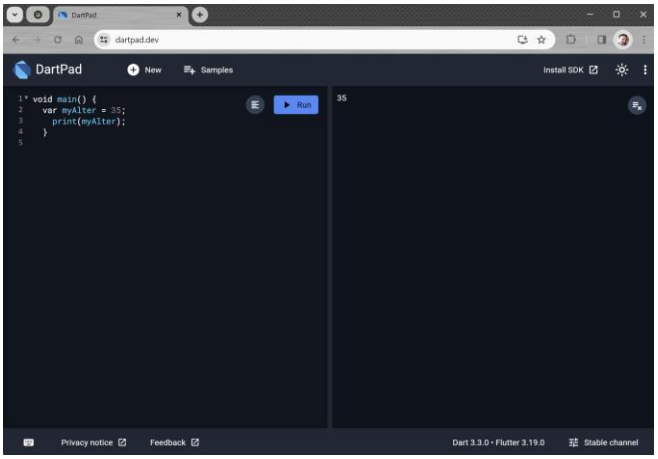
```
var myAlter = 35;
```

Jede Dart-Anweisung endet mit einem Semikolon, genau wie Anweisungen in C und Java. Im obigen Code haben Sie die Variable `myAlter` erstellt und auf *35* festgelegt.

Sie können den integrierten `Druck` in Dart verwenden, um die Variable auf der Konsole auszugeben. Fügen Sie diesen Aufruf nach der Variablen hinzu:

```
print(myAlter); // 35
```

Klicken Sie in DartPad auf RUN, um den Code auszuführen. Der Wert der Variablen, *35*, wird im Ausgabebereich (Console) angezeugt.



1. Kommentare

Kommentare in Dart sehen genauso aus wie die in C und in anderen Sprachen: Text, der auf `//` folgt, ist ein einzeiliger Kommentar, während Text innerhalb von `/* ... */` ein mehrzeiliger Kommentarblock ist. Hier ein Beispiel:

```
//Dies ist ein einzeiliger Kommentar.

print(myAlter); //Dies ist ein
Kommentar.

/*
 Dies ist ein mehrzeiliger
Kommentarblock. Dies ist für lange Zeit
nützlich
 Kommentare, die sich über mehrere
Zeilen erstrecken.
 */
```

2. Datentypen

Dart ist *statisch typisiert*, d. h., jede Variable in Dart hat einen Typ, der beim Kompilieren des Codes bekannt sein muss. Der Variablentyp kann sich beim Ausführen des Programms **nicht** ändern. C, Java, Swift und Kotlin sind ebenfalls statisch typisiert.
Dies steht im Gegensatz zu Sprachen wie Python und JavaScript, die *dynamisch typisiert* sind. Das bedeutet, dass Variablen verschiedene Arten von Daten enthalten können, wenn Sie das Programm ausführen. Sie müssen den Typ nicht kennen, wenn Sie den Code kompilieren.
Klicken Sie im Editorfenster auf `myAlter` und sehen Sie im Feld "*Documentation*" nach . Sie sehen, dass Dart *abgeleitet* hat, dass `myAlter` ein `int` ist, da es mit dem ganzzahligen Wert *35* initialisiert wurde.
Wenn Sie nicht explizit einen Datentyp angeben, verwendet Dart *den Typrückschluss* (type inference), um ihn zu bestimmen, genau wie Swift und Kotlin.

Dart verwendet den Typrückschluss auch für andere Typen als `int`. Geben Sie eine Variable pi ein, die gleich 3,14 ist:

```
var pi = 3.14;

print(pi);  3.14
```

Dart leitet ab, dass `Pi` ein `double` ist, da Sie einen Gleitkommawert verwendet haben, um es zu initialisieren. Alternativ können Sie den Typ deklarieren, anstatt den Typrückschluss zu verwenden.

3. Grundlegende Dart-Typen

Dart verwendet die folgenden Grundtypen:

int: Ganze Zahlen
double: Gleitkommazahlen
bool: Boolesche Werte
String: Zeichenfolgen

Hier ist ein Beispiel für jeden Dart-Typ:

int (num)	42
double (num)	3.14159
bool	true/false
String	'Hello' + "World"
dynamic	42/3.14/true/'Hello'

`Int` und `double` werden beide von einem Typ mit dem Namen `num` abgeleitet. `num` verwendet das `Schlüsselwort dynamic`, um die dynamische Eingabe im statisch typisierten Dart zu imitieren.

Ersetzen Sie dazu `var` durch den Typ, den Sie verwenden möchten:

```
int deinAlter = 27;

print(deinAlter);  27
```

4. Das dynamische Schlüsselwort

Wenn Sie das Schlüsselwort `dynamic` anstelle von `var` verwenden, erhalten Sie eine dynamisch typisierte Variable:

```
dynamic numberOfKittens;
```

Hier können Sie `numberOfKittens` durch die Verwendung von in Anführungszeichen zu einem `String` machen. Weitere Informationen zum String-Typ finden Sie weiter unten in diesem Tutorial.

```
numberOfKittens = 'Es gibt keine
Kätzchen!';

print(numberOfKittens); //Es gibt keine
Kätzchen!
```

`numberOfKittens` hat einen Typ, da Dart über eine statische Typisierung verfügt. Dieser Typ ist jedoch `dynamisch`, was bedeutet, dass Sie ihm andere Werte mit anderen Typen zuweisen können. So können Sie der `print-Anweisung` einen int-Wert zuweisen.

```
numberOfKittens = 0;

print(numberOfKittens);  //0
```

5. Boolesche Werte

Der bool-Typ enthält Werte von `true` oder `false`.

```
bool areThereKittens = false;

print (areThereKittens); // false


numberOfKittens = 1;

areThereKittens = true;

print(areThereKittens); //true
```

6. Enum

Enum steht für Aufzählung.

```
enum Fruit {
 apple, banana
}

main() {
 var a = Fruit.apple;
 switch (a) {
 case Fruit.apple:
 print('it is an apple');
 break;
  default:
    print("keine Frucht");
 }
 // get all the values of the enums
 for (Fruit value in Fruit.values) {
 print(value);
 }
 // get the second value
 print(Fruit.values[1]);
}
```

Ausgabe:

```
it is an apple
Fruit.apple
Fruit.banana
Fruit.banana
```

7. Tpyzuweisung mit „of“

- kommt noch -

8. Sondertypen

Der Unterstrich “_” als erstes Zeichen einer Variabeln oder auch Funktion kennzeichnet diese als“ *private*“.

III. Operatoren

Dart verfügt über alle üblichen Operatoren, die Sie aus anderen Sprachen wie C, Swift und Kotlin kennen.

Hinweis: Dart ermöglicht auch das *Überladen von Operatoren*, wie in C++ und Kotlin, aber das würde den Rahmen dieses Tutorials sprengen. Um mehr über das Thema zu erfahren, besuchen Sie die Wikipedia-Seite zum Überladen von Operatoren.

Als Nächstes sehen Sie sich die einzelnen Operatoren an.

1. Arithmetische Operatoren

Arithmetische Operatoren funktionieren genau so, wie Sie es erwarten würden. Probieren Sie sie aus, indem Sie Ihrem DartPad eine Reihe von Operationen hinzufügen:

```
print (40 + 2); // 42

print (44 - 2); // 42

print (21 * 2); // 42

print (84 / 2);  // 42.0
```

Für die Division, selbst mit ganzen Zahlen, leitet Dart ab, dass die resultierende Variable ein `Double` ist. Deshalb bekommst du *42.0* statt 42 für die letzte *print* Anweisung.

Hinweis: DartPad zeigt das Ergebnis von "84 / 2" als 42 in der Konsole an, da es die Ausgabe auf der Konsole so formatiert, dass nur die signifikanten Ziffern angezeigt werden. Wenn Sie die gleiche Anweisung in einem Dart-Programm aus dem Dart SDK drucken, erhalten Sie *42.0* als Ergebnis.

Operator	Meaning
+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulo
-expr	Unary Minus (reverse the sign of expression)
~/	Divide (Return Integer)
++	Increment Operator
--	Decrement Operator

2. Gleichheits-Operatoren

Dart verwendet *doppelte* (`==`) `gleich-` und `ungleich`-Operatoren *(*`!=`) :

```
print(42 == 43); // false

print(42 != 43); //true
```

Vergleichs-Operatoren

Dart verwendet die typischen Vergleichsoperatoren:

Operator	Meaning
>	Greater Than
<	Less Than
>=	Greather Than or Equal To
<=	Less Than or Equal To

Hier sind einige Beispiele:

```
print(42 < 43); // true
print(42 >= 43); //false
```

Darüber hinaus werden auch die üblichen zusammen-gesetzten Arithmetik-/Zuweisungsoperatoren verwendet:

```
var Wert = 42.0;

Wert += 1; print(Wert);  //43.0

Wert -= 1; print(Wert);  //42.0

Wert *= 2; print(Wert);  //84.0

Wert /= 2; print(Wert);  //42.0
```

Zusammengesetzte Arithmetik-/Zuweisungsoperatoren führen zwei Aufgaben aus. += addiert den Wert rechts zur Variablen auf der linken Seite und weist dann das Ergebnis der Variablen zu.

Eine Kurzform von `+= 1` ist **++:**

```
Wert++;

print(Wert);  //43.0
```

Und Dart hat den üblichen Modulo-Operator **(%),** um den **Rest** zurückzugeben, nachdem eine Zahl durch eine andere geteilt wurde:

```
print (392 % 50);  //42
```

3. Typ-Test-Operator

Operator	Meaning
is	True if the object has specified type
is!	True if the object has not specified type

4. Logische Operatoren

Dart verwendet die gleichen logischen Operatoren wie andere Sprachen, einschließlich `&&` für *AND* und `||` für *OR*.

Operator	Meaning
&& (AND)	Returns true if all conditions are true
|| (OR)	Returns true if any one condition is true
! (NOT)	Returns the inverse of the result

^ (XOR) Gibt *true* zurück, wenn weder beide wahr noch beide unwahr sind.

```
print((41 < 42) & (42 < 43)); // true

drucken((41 < 42) || (42 > 43)); //rtrue
```

Der Negationsoperator ist das *Ausrufezeichen*, das *false* in *true* und *true* in **false** umwandelt.

```
print(!( 41 < 42)); //false
```

In der Dart-Dokumentation finden Sie eine vollständige Liste der unterstützten Operatoren.

5. Zuweisungs-Operator

Dart verfügt über zwei Zuweisungsoperatoren.

Operator	Meaning
=	Assign value from right operand to left operand
??=	Assign the value only if the varible is null

6. Ternärer Operator

Dart verfügt über zwei **spezielle Operatoren**, mit denen Sie kleine Ausdrücke auswerten können, die möglicherweise eine ganze if-else-Anweisung erfordern.

a) Bedingung ? expr1 : expr2

Wenn die Bedingung wahr ist, wird ***expr1*** ausgewertet und gibt seinen Wert zurück. Andernfalls wird ***expr2*** ausgewertet und gibt den Wert zurück.

```
void main() {
  var a = 25;
  var res = a > 15
        ? "Wert größer als 15"
        : "Wert kleiner oder gleich 15";
  print(res);
}

//Ausgabe: Wert größer als 15
```

b) expr1 ?? expr2

Wenn ***expr1*** hier ungleich NULL ist, wird der Wert zurückgegeben. Andernfalls wird der Wert von ***expr2*** ausgewertet und zurückgegeben.

```
void main() {
  var a = 15;
  var b = Null;
  var res = a ?? b;
  print ( res );
}

//Ausgabe: 15
```

IV. Strings

Der Dart-Zeichenfolgentyp ist `String`. Zeichenfolgen werden in Dart mit Text ausgedrückt, der entweder in *einfache* oder *doppelte Anführungszeichen* eingeschlossen ist.
Sie können entweder `var` und automatische Typesierung oder `String` verwenden, um eine Zeichenfolgenvariable zu erstellen:

```
var firstName = 'Albert';
```

```
String lastName = "Einstein";
```

Ähnlich wie bei Sprachen wie Kotlin und Swift können Sie den Wert eines Ausdrucks in eine Zeichenfolge einbetten, indem Sie das Dollarzeichen verwenden: *${expression}*. Wenn es sich bei dem Ausdruck um einen Bezeichner handelt, können Sie das *{ } weglassen*. Fügen Sie Folgendes hinzu:

```
var physicist = "$firstName $lastName
mag die Zahl ${84 / 2}";

print(physicist);
//Albert Einstein mag die Zahl 42
```

`$firstName` und `$lastName` werden durch die Variablenwerte ersetzt. Das gibt das berechnete Ergebnis zurück.

Besonderheit: Escaping Strings

Die in Dart verwendeten Escape-Sequenzen ähneln denen, die in anderen C-ähnlichen Sprachen verwendet werden. Sie verwenden z. B. **`\n`** für einen Zeilenumbruch.
Wenn die Zeichenfolge Sonderzeichen enthält, verwenden Sie `\` , um sie mit Escapezeichen zu versehen:

```
var quote = 'If you can\'t explain it
simply\nyou don\'t understand it well
enough.';

print(quote);

// If you can't explain it simply
// you don't understand it well enough.
```

In diesem Beispiel werden einfache Anführungszeichen verwendet, daher wird eine Escapesequenz \' benötigt, um die Apostrophe für *can't* und *don't* in die Zeichenfolge einzubetten. Sie müssten den Apostroph nicht mit Escapezeichen versehen, wenn Sie stattdessen doppelte Anführungszeichen verwenden würden.

Wenn Sie Escapesequenzen innerhalb der Zeichenfolge anzeigen müssen, können Sie *unformatierte Zeichenfolgen* verwenden, denen das Präfix r vorangestellt ist.

```
var rawString = r"Wenn du es nicht
einfach erklären kannst\n verstehst du
es nicht gut genug.";

print(rawString);

//Wenn du es nicht einfach erklären
kannst\n verstehst du es nicht gut
genug.
```

Hier behandelte Dart `'\n'` als normalen Text, da die Zeichenfolge mit `r begann`.

2. Runen

Die Runen sind ganzzahlige Unicode-Codepunkte eines Strings. Die Zeichen einer Zeichenfolge werden in UTF-16 codiert. Dekodierung von UTF-16, das Surrogate kombiniert, ergibt Unicode-Codepunkte. Ähnlich wie in der Go-Terminologie verwendet Dart den Namen „Rune“ für eine Ganzzahl, die einen Unicode-Code-Wert darstellt. Verwende die runes-Eigenschaft, um die Runen einer Zeichenfolge abzurufen.

Beispiel:

```
const string = 'Dart';
final runes = string.runes.toList();
print(runes); // [68, 97, 114, 116]
```

Für einen Charakter außerhalb der Basic Multilingual Plane (Ebene 0), die aus einem Ersatzpaar, Runen, besteht kombiniere das Paar und gibt eine einzelne Ganzzahl zurück.
Zum Beispiel das Unicode-Zeichen für das Emoji „Mann“. ('👨', `U+1F468`) wird aus dem kombiniert Surrogate `U+d83d` und `U+dc684`

`Beispiel:`

```
const emojiMan = '👨';
print(emojiMan.runes); // (128104)
// Surrogate pairs:
for (final item in emojiMan.codeUnits)
{
print(item.toRadixString(16));
// d83d
// dc68
}
```

IV. Unveränderlichkeit (Immutability)

Dart verwendet die Schlüsselwörter `const` und `final` für Werte, die sich nicht ändern.
Verwenden Sie `const` für Werte, die zur Kompilierzeit bekannt sind. Verwenden Sie `final` für Werte, die zur Kompilierzeit noch nicht bekannt sein müssen, aber nach der Initialisierung nicht neu zugewiesen werden können.
Anmerkung: `final` verhält sich wie `val` in Kotlin oder `let` in Swift.
Sie können `const und final` anstelle von `var` verwenden und über den Typrückschluss den Typ bestimmen lassen:

```
const speedOfLight = 299792458;

print(speedOfLight);  //299792458
```

Hier erkennt Dart, dass `speedOfLight` ein `int` ist, wie

`final` gibt an, dass eine Variable *unveränderlich* ist, d. h., sie können keine neuen Werte zuweisen. Sie können den Typ explizit mit `final` oder `const` angeben:

```
finaler Planet = 'Jupiter';

Planet = 'Mars';

//Fehler: Planet kann nur einmal gesetzt werden

final String moon = 'Europa';

print('$planet hat einen Mond, $moon');

//Jupiter hat einen Mond, Europa
```

V. NULL-Safty (Nullability)

Wenn Sie in der Vergangenheit eine Variable nicht initialisiert haben, hat Dart ihr den Wert `null` gegeben, was bedeutet, dass *nichts* in der Variablen gespeichert wurde. Ab Dart 2.12 schließt sich Dart jedoch anderen Sprachen wie Swift und Kotlin an, um standardmäßig keine NULL-Werte mehr zulassen zu können.

Darüber hinaus garantiert Dart, dass ein Typ, der keine NULL-Werte zulässt, niemals einen NULL-Wert enthält. Dies wird als *sprechende Nullsicherheit* bezeichnet.

Wenn Sie eine Variable deklarieren möchten, müssen Sie sie normalerweise initialisieren:

```
String middleName = 'Mai';

print(middleName); //Mai
```

Da jedoch nicht jeder einen zweiten Vornamen hat, ist es sinnvoll, `middleName` zu einem Wert zu machen, der NULL-Werte zulässt. Um Dart mitzuteilen, dass Sie den Wert null zulassen möchten, fügen Sie **`?`** nach dem Typ.

```
String? middleName = null;

print(middleName); // null
```

Der Standardwert für einen Typ, der NULL-Werte zulässt, ist `null`, sodass Sie den Ausdruck wie folgt vereinfachen können:

```
String? middleName;

print(middleName); //null
```

1. Null-fähiger Operator ??

Dart verfügt über einige *NULL-fähige* Operatoren, die Sie verwenden können, wenn Sie mit NULL-Werten arbeiten.

Der doppelte Fragezeichen-Operator, **`??`**, ist wie der *Elvis-Operator* in Kotlin: Er gibt den linken Operanden zurück, wenn das Objekt nicht *null* ist. Andernfalls wird der rechte Wert zurückgegeben:

```
var name = middleName ?? "keine";

print(name); //keine
```

Da `middleName` *null* `ist`, weist Dart den rechten Wert `"keine" zu`.

2. Der ?. Operator

Die `?.` -Operator schützt Sie vor dem Zugriff auf Eigenschaften von NULL-Objekten. Sie gibt *null* zurück, wenn das Objekt selbst *null* ist. Andernfalls wird der Wert der Eigenschaft auf der rechten Seite zurückgegeben:

```
print(middleName?.length); //null
```

Wenn Sie in den Tagen vor der NULL-Sicherheit das Fragezeichen vergessen und middleName.length geschrieben haben, stürzte Ihre App zur Laufzeit ab, wenn middleName *null* war. Das ist jetzt kein Problem mehr, denn Dart sagt Ihnen jetzt sofort, wann Sie mit Nullwerten umgehen müssen.

VI. Kontroll Strukturen

Mit der *Ablaufsteuerung* können Sie festlegen, wann bestimmte Codezeilen ausgeführt, übersprungen oder wiederholt werden sollen. Sie verwenden *Bedingungen* und *Schleifen*, um die Ablaufsteuerung in Dart zu verarbeiten.

In diesem Abschnitt erfahren Sie mehr über:

- Bedingungen
- While-Schleifen
- Fortsetzen und Pause
- For-Schleifen

Hier erfahren Sie, was Sie über Ablaufsteuerungselemente in Dart wissen müssen.

1. Bedingungen

Die grundlegendste Form der Ablaufsteuerung ist die Entscheidung, ob bestimmte Teile des Codes ausgeführt oder übersprungen werden sollen, abhängig von den Bedingungen, die beim Ausführen des Programms auftreten.

Das Sprachkonstrukt für die Behandlung von Bedingungen ist die `if`/`else`-Anweisung. `if`/`else` in Dart sieht fast identisch aus wie in anderen C-ähnlichen Sprachen.

a) If-Aussagen

Angenommen, Sie haben ein veränderliches Tier, das derzeit ein Fuchs ist. Das sieht dann so aus:

```
var tier = 'Fuchs';
```

Sie können eine if-Anweisung verwenden , um zu überprüfen, ob es sich bei dem Tier um eine Katze oder einen Hund handelt, und dann den entsprechenden Code ausführen.

```
if (tier == 'Katze' || tier == 'Hund') {
  print('Das Tier ist ein Haustier.');
}
```

Hier haben Sie die Gleichheits- *und* OR-Operatoren verwendet, um ein *bool* innerhalb der Bedingung für die if-Anweisung zu erstellen .

b) Sonst-Aussagen

Mit einer *else*-Klausel können Sie alternativen Code ausführen, wenn die Bedingung false ist:

```
else {
  print('Tier ist KEIN Haustier.');
}
Das Tier ist KEIN Haustier.
```

Sie können auch mehrere `if`/`else`-Anweisungen zu einem `if/else if/else`-Konstrukt kombinieren :

```
if (tier == 'Katze' || tier == 'Hund') {
  print('Das Tier ist ein Haustier.');
} else if (tier == 'Elefant') {
  print('Das ist ein großes Tier.');
} else {
  print('Tier ist KEIN Haustier.');
}
Das Tier ist KEIN Haustier.
```

Sie können so viele `else/if`-Zweige zwischen `if` und `else` haben, wie Sie benötigen.

2. While-Schleifen

Mit Schleifen können Sie Code eine bestimmte Anzahl von Malen oder basierend auf bestimmten Bedingungen wiederholen. Sie behandeln bedingungsbasierte Wiederholungen mithilfe von *while-Schleifen*.

Es gibt zwei Formen von while-Schleifen in Dart: `while` und `do-while`. Der Unterschied besteht darin, dass die Schleifenbedingung while vor dem Codeblock auftritt. In `do-while` tritt der Zustand danach auf. Das bedeutet, dass `Do-While-Schleifen` sicherstellen, dass der Codeblock mindestens einmal ausgeführt wird.

a) Testen der While-Schleife

Um dies auszuprobieren, erstellen Sie eine Variable i und initialisieren diese mit 1:

```
var i = 1;
```

Verwenden Sie als Nächstes eine while-Schleife, `i` auszugeben, während Sie es inkrementieren. Legen Sie die Bedingung auf `i` ist kleiner als 10 fest:

```
while (i < 10) {
  print(i);
  i++;
}
 // Ausgabe 1,2,3,4,5,6,7,8,9
```

Führen Sie den Code aus, und Sie werden sehen, dass die `while-Schleife` die Zahlen 1 bis 9 ausgibt.

b) Ausprobieren der Do-While-Schleife

Setze `i` zurück und füge dann eine `Do-While-Schleife` hinzu :

```
i = 1;
do {
  print (i);
  i++;
} while (i < 10);
// Ausgabe 1,2,3,4,5,6,7,8,9
```

Die Ergebnisse sind die gleichen wie zuvor. Dieses Mal wurde der Schleifenkörper jedoch einmal ausgeführt, bevor die Bedingung für das Beenden der Schleife überprüft wurde.

c) continue und break

Dart verwendet `continue`- und `break`-Schlüsselwörter in Schleifen und an anderer Stelle. Hier ist, was sie tun:

- *continue*: Überspringt den verbleibenden Code innerhalb einer Schleife und wechselt sofort zur nächsten Iteration.
- *break*: Stoppt die Schleife und setzt die Ausführung nach dem Hauptteil der Schleife fort.

Seien Sie vorsichtig, wenn Sie `continue` in Ihrem Code verwenden. Wenn Sie z. B. die `do-while-Schleife` von oben nehmen und fortfahren möchten, wenn `i` gleich 5 ist, kann dies zu einer *Endlosschleife* führen, je nachdem, wo Sie die `continue-Anweisung` platzieren :

```
i = 1;
do {
  print (i);
  if (i == 5) {
    continue;
  }
  ++i;
} while (i < 10);
// Ausgabe 1,2,3,4,5,5,5,5...
```

Die Endlosschleife tritt auf, weil Sie, sobald `i=5` ist, sie nie wieder inkrementieren, so dass die Bedingung immer wahr bleibt.

Wenn Sie dies ausführen, führt die Endlosschleife dazu, dass das Programm hängen bleibt. Verwenden Sie stattdessen `break`, damit die Schleife endet, nachdem `i` 5 erreicht hat:

```
i = 1;
do {
  print (i);
  if (i == 5) {
    break;
  }
  ++i;
} while (i < 10);
// Ausgabe 1,2,3,4,5
```

Führen Sie den Code aus. Nun endet die Schleife nach fünf Iterationen.

3. For-Schleifen

In Dart verwenden Sie `for-Schleifen,` um eine vorgegebene Anzahl von Malen zu wiederholen. `for-Schleifen` bestehen aus einer Initialisierung, einer Schleifenbedingung und einer Aktion. Auch hier ähneln sie `for-Schleifen` in anderen Sprachen.

Dart bietet auch eine `for-in-Schleife`, die über eine Sammlung von Objekten iteriert. Weitere Informationen dazu finden Sie später.

Um zu sehen, wie eine `for-Schleife` funktioniert, erstellen Sie eine Variable für die Summe:

```
var sum = 0;
```

Verwenden Sie als Nächstes eine `for-Schleife`, um einen Schleifenzähler von `i` bis 10 zu initialisieren . Sie überprüfen dann, ob `i` kleiner oder gleich 10 ist, und erhöhen i nach jeder Schleife.

Verwenden Sie innerhalb der Schleife eine zusammengesetzte Zuweisung, um `i` zur laufenden Summe hinzuzufügen:

```
for (var i = 1; i <= 10; i++) {
  Summe += i;
}
print("Die Summe ist $sum"); //Die Summe
beträgt 55
```

4. for each

```
void main (){

    list<int> liste = [3,4,5,6,7,5];
    list<int> liste2 = [];

    /* for(int i= 0; i< list.length ;
i++){

      if(liste[i]==3){
        liste2.add(liste[i]);
      }

      print(liste[i]);

    }*/

 liste.forEach((element) {
      element++;
      liste2.add(element);
    });

    print(liste);
    print(liste2);
}
```

5. switch – case

```
void main (){

    int alter = 16;
    String name = "Hans";
```

```
    print("vor dem switch");

    switch (name){
      case: "Peter":
        print("ich bin Peter");
        break;
      case: "Hans":
        print("ich bin Hans");
        break;
      default:
        print("keine ahnung");
    }
     print("nach dem switch");
}
```

VII. Sammlungen (Collections)

Sammlungen sind nützlich, um zusammengehörige Daten zu gruppieren. Dart enthält verschiedene Arten von Sammlungen, aber in diesem Tutorial werden die beiden häufigsten behandelt: `Liste` und `Karte`.

1. Listen

Listen in Dart ähneln *Arrays* in anderen Sprachen. Sie verwenden sie, um eine geordnete Liste von Werten zu verwalten. Listen sind nullbasiert, daher befindet sich das erste Element in der Liste an Index 0:

Hier ist eine Liste mit verschiedenen Desserts:

```
List<Strings> desserts = ['Kekse',
'Cupcakes', 'Donuts', 'Kuchen'];
```

Sie schließen die Elemente einer Liste in eckige Klammern ein: **`[ ]`**. Sie verwenden **Kommas**, um die Elemente zu trennen.

Am Anfang der Zeile sehen Sie, dass der Typ `Liste` ist. Sie werden feststellen, dass kein Typ enthalten ist. Dart leitet ab, dass die Liste den Typ Liste hat.

Hier ist eine Liste von ganzen Zahlen (int):

```
final numbers = [42, -1, 299792458,
100];
```

```
void main (){

  list<int> liste = [1,2,3];

  print(liste);
  print(liste[0]);
  print(liste.first);
  print(liste.length);
  print(liste.isEmpty);
  liste.add(3);
  print(liste);
}
```

a) Arbeiten mit Listenelementen

Um auf die Elemente einer Liste zuzugreifen, musst du die Indexnummer nach dem Namen der Listenvariablen in eckige Klammern setzen. Zum Beispiel:

```
final firstDessert = Desserts[0];
print(firstDessert); //Keks
```

Da Listenindizes nullbasiert sind, ist `desserts[0]` das erste Element der Liste.

Fügen Sie Elemente mit `add` bzw. `remove` hinzu und entfernen sie :

```
desserts.add('Kuchen');
print(desserts);
//[Kekse, Cupcakes, Donuts, Kuchen,
Kuchen]

desserts.remove('Donuts');
print(desserts);
/[Kekse, Cupcakes, Kuchen, Kuchen]
```

Zuvor haben Sie for-Schleifen kennengelernt. Die `for-in-Schleife` von Dart, die besonders gut mit Listen funktioniert. Probieren Sie es aus:

```
for (final dessert in desserts) {
  print('Ich liebe es, $dessert zu
essen.');
}
//Ich liebe es, Kekse zu essen.
//Ich liebe es, Cupcakes zu essen.
//Ich liebe es, Kuchen zu essen.
//Ich liebe es, Kuchen zu essen.
```

Sie müssen keinen Index verwenden. Dart durchläuft einfach jedes Element von *desserts* und weist es jedes Mal einer Variablen mit dem Namen dessert zu.

2. Maps

Wenn Sie eine Liste von Wertepaaren wünschen, ist `Map` eine gute Wahl. Dart Maps ähneln den *dictionaries* in Swift und den *maps* in Kotlin.

Hier ist ein Beispiel für eine Map in Dart:

```
Map<String, int> calories = {
  "Kuchen": 500,
  "Donuts": 150,
  "Cookies": 100,
};
```

Sie umgeben `Maps` mit geschweiften Klammern `{}`. Verwenden Sie Kommas, um die Elemente einer Map zu trennen.

Elemente einer Zuordnung werden als *Schlüssel-Wert-Paare* bezeichnet, wobei sich der Schlüssel auf der linken Seite eines Doppelpunkts und der Wert auf der rechten Seite befindet.

Sie finden einen Wert, indem Sie den Schlüssel verwenden:

```
final DonutCalories =
calories['Donuts'];
print(donutCalories);  //150
```

Der Schlüssel `"Donuts"` befindet sich in den eckigen Klammern hinter dem Kartennamen. In diesem Fall wird der Wert `150` zugeordnet.

Fügen Sie einer map ein neues Element hinzu, indem Sie den Schlüssel angeben und ihm einen Wert zuweisen:

```
calories['Eisbecher'] = 1200;
print(calories);
/{Kuchen: 500, Donuts: 150, Kekse: 100,
Eisbecher: 1200}
```

Wenn Sie diesen Code ausführen, sehen Sie die Map mit Ihrem neuen Dessert am Ende.

```
void main (){

  Map<String,String> map =
{"key1":"value1", "key2":"value2",};
  print(map.length);
  print(map.keys);
  print(map["key2"]);
}
```

3. Sets

Ein Set ist eine Sammlung von einzigartige Gegenständen. Du kannst keine doppelten Werte im Set speichern. Es ist ungeordnet, so dass es schneller als Listen sein kann, wenn mit einer großen Datenmenge gearbeitet wird. Set ist nützlich, wenn du eindeutige Werte speichern musst, ohne die Reihenfolge der Eingabe zu berücksichtigen, wie z.B.r Früchte, Monatsnamen, Tagesnamen usw. Es wird durch geschweifte Klammern {} dargestellt.

```
Set <variable_type> variable_name = {};

void main(){
  Set<String> fruits = {"Apfel",
"Orange", "Mango"};
  print(fruits);
}
```

Es gibt diverse Eigenschaften bei den Sets, wie dem nachfolgenden Beispiel entnommen werden kann.

```
void main() {
  Set<String> fruits = {"Apfel",
"Orange", "Mango", "Banane"};

  // Properties von Set
  print("Der erste Wert ist
${fruits.first}");
  print("Der letzte Wert ist
${fruits.last}");
  print("Ist fruits leer?
${fruits.isEmpty}");
  print("Ist fruits nicht leer?
${fruits.isNotEmpty}");
  print("Die Anzahl der fruits ist
${fruits.length}");
print(fruits.contains("Mango"));
  print(fruits.contains("Limone"));

fruits.add("Limone");
fruits.remove("Apfel");
print("Nach add/remove: $fruits");

}
//Ausgabe
Der erste Wert ist Apfel
Der letzte Wert ist Banane
Ist fruits leer? false
Ist fruits nicht leer? true
Die Anzahl der fruits ist 4
true
false
Nach add/remove: {Orange, Mango, Banane,
Limone}
```

Mit der .addAll() Methode können gleich mehrere Elements dem Set hinzugefügt werden.

```
Set<int> numbers = {1, 2, 3};
numbers.addAll([4,5]);
print("Jetzt: $numbers");
 //Jetzt: {1, 2, 3, 4, 5}
```

Auf alle Elemente des Sets zugreifen:

```
 for(String fruit in fruits){
   print(fruit);
 }
//Output Apple  - Orange  -Mango
```

Weitere Methoden sind:

.**clear**()	Entfernt alle Elemente des Set
.**difference**()	Erzeugt ein neues Set mit den Elementen die nicht in dem anderen Set sind: `set3=fruits.difference(fruits2)`
.**elementAt**()	Gibt den Wert des genannten Elementes zurück. `print(days.elementAt(2));`
.**intersection**()	Findet gemeinsame Elemente in zwei Sets. `set3= frus.intersection(frus2);`

3. Record

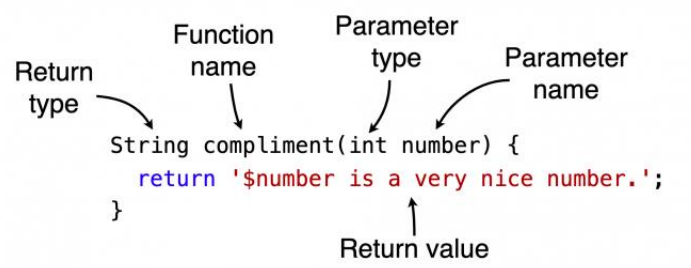
muss noch ergänzt werden.

4. Iterable

muss noch ergänzt werden.

VIII. Funktionen

Mit Funktionen können Sie mehrere zusammengehörige Codezeilen in einen einzelnen Bereich packen. Anschließend rufen Sie die Funktion auf, und vermeiden so, dass diese Codezeilen in Ihrer Dart-App mehrfach wiederholt werden.



Eine Funktion besteht aus den folgenden Elementen:

- Rückgabetyp
- Name der Funktion
- Parameterliste in runden Klammern
- Funktionskörper in geschweifen Klammern

1. Funktionen definieren

Der Code, den Sie in eine Funktion umwandeln, befindet sich in den geschweiften Klammern. Wenn Sie die Funktion aufrufen, übergeben Sie Argumente, die den Typen der Parameter der Funktion entsprechen.

Als Nächstes schreiben Sie eine neue Funktion, die prüft, ob ein bestimmter String eine *Banane* ist:

```
bool isBanane(String fruit) {
  return fruit == 'Banane';
}
```

Die Funktion verwendet `return`, um einen `bool-Wert` zurückzugeben. Das Argument, das Sie an die Funktion übergeben, wird geprüft und gibt einen `bool-Wert` zurück.

Diese Funktion gibt für jede Eingabe immer den gleichen Werttyp zurück. Wenn eine Funktion keinen Wert zurückgeben muss, können Sie den Rückgabetyp auf `void` festlegen. Das macht zum Beispiel `main`.

2. Arbeiten mit Funktionen

Sie können die Funktion aufrufen, indem Sie es wie eine Zeichenfolge übergeben. Sie können dann das Ergebnis dieses Aufrufs an `print` übergeben:

```
void main() {
  var fruit = "Apfel";
  print(isBanane(fruit)); //false
}
```

3. Verschachtelte Funktionen (Nesting Functions)

In der Regel definieren Sie Funktionen entweder außerhalb anderer Funktionen oder innerhalb von Dart-Klassen. Sie können jedoch auch Dart-Funktionen verschachteln. Sie können z. B. `isBanane` in `main` verschachteln.

```
void main() {
  bool isBanana(String fruit) {
```

```
    return fruit == 'Banane';
  }

  var fruit = 'Apfel';
  print(isBanane(fruit)); //false
}
```

Sie können das Argument auch ändern und es dann mit dem neuen Argument erneut aufrufen:

```
fruit = 'Banane';
print(isBanane(fruit));  //true
```

Das Ergebnis des Aufrufs der Funktion hängt von den übergebenen Argumenten ab.

4. Optionale Parameter

Wenn ein Parameter für eine Funktion *optional* ist, können Sie ihn in **eckige Klammern []** einschließen und den Typ NULL-Werte zulassen lassen:

```
String fullName(String first, String
last, [String? title]) {
  if (title == null) {
    return '$first $last';
  } else {
    return '$title $first $last';
  }
}
```

In dieser Funktion ist `title` optional. Der Standardwert ist `null`, wenn Sie ihn nicht angeben.

Nun können Sie die Funktion mit oder ohne den optionalen Parameter aufrufen:

```
print(fullName('Joe', 'Howard'));
// Joe Howard

print(fullName('Albert', 'Einstein',
'Professor'));
//Prof. Dr. Albert Einstein
```

5. Benannte Parameter und Standardwerte

Wenn Sie mehrere Parameter haben, kann es verwirrend sein, sich daran zu erinnern, welcher welcher ist. Dart löst dieses Problem mit *benannten Parametern*, die Sie erhalten, indem Sie die Parameterliste mit **geschweiften Klammern**{} umgeben.

Diese Parameter sind standardmäßig *optional*, aber Sie können ihnen Standardwerte zuweisen oder sie mit dem Schlüsselwort `required` als erforderlich festlegen:

```
bool withinTolerance({required int
value, int min = 0, int max = 10}) {
  return min <= value && value <= max;
}
```

`value` ist erforderlich, während `min` und `max` mit Standardwerten optional sind.

Bei *benannten Parametern* können Sie Argumente in einer anderen Reihenfolge übergeben, indem Sie die Parameternamen mit einem Doppelpunkt angeben:

```
print(withinTolerance(min: 1, max: 5,
Wert: 11)); //false
```

Sie können die Parameter mit Standardwerten weglassen, wenn Sie die Funktion aufrufen.

```
print(withinTolerance(Wert: 5)); //true
```

6. Anonyme Funktionen

Dart unterstützt *first-class Funktionen*, was bedeutet, dass es Funktionen wie jeden anderen Datentyp behandelt. Sie können sie Variablen zuweisen, sie als Argumente übergeben und von anderen Funktionen zurückgeben.

Um diese Funktionen als Werte zu übergeben, lassen Sie den Funktionsnamen und den Rückgabetyp weg. Da es keinen Namen gibt, wird diese Art von Funktion als *anonyme Funktion bezeichnet*.

```
~~String compliment~~(int number) {
  return '$number is a very nice number.';
}
```

Sie können einer Variablen mit dem Namen `onPressed` eine anonyme Funktion wie folgt zuweisen:

```
final onPressed = () {
  print('Taste gedrückt');
};
```

`onPressed` hat einen Wert vom Typ `Function`. Die leeren Klammern geben an, dass die Funktion keine Parameter hat. Wie bei regulären Funktionen ist der Code in den geschweiften Klammern der Funktionstext.

Um den Code innerhalb des Funktionstexts auszuführen, rufen Sie den Variablennamen so auf, als wäre er der Name der Funktion:

```
onPressed(); gedrückte Taste
```

Sie können Funktionen, deren Hauptteil nur eine einzelne Zeile enthält, mithilfe der Pfeilsyntax (sog. fat arrow) vereinfachen. Entfernen Sie dazu die geschweiften Klammern und fügen Sie **=>** hinzu.

Hier ist ein Vergleich der obigen anonymen Funktion und einer refaktorierten Version:

```
//Ursprüngliche anonyme Funktion
final onPressed = () {
  print('Taste gedrückt');
};

//refaktoriert
final onPressed = () => print('Taste
gedrückt');
```

Das ist besser zu lesen.

7. Verwendung anonymer Funktionen

Sie werden häufig anonyme Funktionen in Flutter sehen wie die oben genannte, die als Rückrufe für UI-Ereignisse weitergegeben werden. Auf diese Weise können Sie den Code angeben, der ausgeführt wird, wenn ein Benutzer etwas tut, z. B. eine Schaltfläche drückt.

Häufig werden anonyme Funktionen auch mit Sammlungen (collections) verwendet. Sie können einer Liste (*list*) eine anonyme Funktion zuweisen, die für jedes Element der Liste eine Aufgabe ausführt. Zum Beispiel:

```
// 1
final drinks = ['Wasser', 'Saft',
'Milch'];
// 2
final bigDrinks = drinks.map(
 // 3
  (drink) => drink.toUpperCase()
);
// 4
print(bigDrinks); //(WASSER, SAFT,
MILCH)
```

Schauen wir uns die einzelnen Schritte an:

1. Definieren Sie eine Liste von Getränken, die Kleinbuchstaben enthalten.
2. `.map` übernimmt alle Listenwerte und gibt eine neue Auflistung zurück.
3. Als Parameter wird eine anonyme Funktion übergeben. In dieser anonymen Funktion verfügen Sie über ein drink-Argument, das jedes Element der Liste darstellt.
4. Der Text der anonymen Funktion konvertiert jedes Element in Großbuchstaben und gibt den Wert zurück. Da es sich bei der ursprünglichen Liste um eine Liste von Zeichenfolgen handelt, hat `drink` auch den Typ `String`.

Die Verwendung einer anonymen Funktion und deren Kombination mit `.map` ist eine bequeme Möglichkeit, eine Sammlung in eine andere umzuwandeln.

Anmerkung: Verwechseln Sie nicht die `.map` -Methode mit dem `Map` Typ.

Führen Sie den Code aus, um die resultierende Auflistung anzuzeigen.

IX. Klassen (OOP)

Klassennamen beginnen immer mit einem Großbuchstaben.

1. Definition

```
void main() {
  //Instanz von Car erzeugen
  Car car1 = Car();
  //Zugriff auf Werte+Funktionen mit .
  car1.color = "fäulnis";

  Car car2 = Car();
  car2.color = "blau";

  car1.sayColor();
  car2.sayColor();

  car1.drive();
}

class Car {
  //! Attribut
  late String color;

  //! methoden
  void drive() {
    print("Auto bewegt sich");
```

```
  }

  void sayColor() {
    print(this.color);
  }
}
```

2. Konstruktor

Automatisch (auch ohne Definition) wird immer ausgeführt **Car().** Es können auch **Parameter** übergeben werden.

- **a)** Einfache Var.: **Car(this.color, this.ps);**
- **b)** Named Parameter mit {}: **Car({this.color, this.ps});**
- c) Zwingende named Parameter mit **required**: **Car({required this.color, this.ps});**

```
void main() {
  Car car1 = Auto (color: "rot", ps:
300);
}

class Car {
  Car({required this.color, required
this.ps});

  final String color;
  final int ps;

  void drive() {
    print("Auto bewegt sich");
  }
}
```

3. this

4. Getter und Setter

5. Vererbung

a) extends

b) implements

c) with (Mixins)

```
void main() {
  Student student1 = Student();
  student1.setStudienjahr = 2;
  int studienjahr =
student1.getStudienjahr;
  print(studienjahr);

  student1.feiern();
  student1.setName = "Patrick";
  student1.laufen();
  student1.lernen();
}

class Person {
  late String _name;
  late int _alter;

  //Getter
  String get getName => this._name;
  int get getAlter => this._alter;

  //Setter
  set setName(String name) {
    this.name = name;
  }
  set setAlter(int alter) {
    this.alter = alter;
  }

  // methoden
  void laufen() {
    print("person läuft!");
  }
}

mixin Lernender {
  void lernen() {
    print("lernen!");
  }
}

class Student extends Person with
Lernender {
  late int _studienjahr;

  //Getter
  int get getStudienjahr =>
this.studienjahr;

  //Setter
  set setStudienjahr(int studienjahr) {
    this.studienjahr = studienjahr;
  }

  // methoden
  void feiern() {
    print("Party!");
  }
}
```

6. Generika

X. Typdef

XI. Kurzschreibweisen

1. Lambda

2. Fat Arrow

XII. Ausnahmebehandlungen

1. Try-Catch

2. Eigene Ausnahmen

XIII. Datum/Zeit

XIV. Pakete

XV. Debugger

1. Assert

muss noch ergänzt werden.

XVI. Asynchrone Programmierung

1. Await und future

```
void main() async {
  DataFetcher fetcher = DataFetcher();
  parallelTask();
  String data = await fetcher.getData();
  print(data);
}

// um wirklich parallelität zu
demonstrieren habe ich hier noch eine
Funktion gebaut
void parallelTask() async {
  for (int i = 0; i < 100; i++) {
    await
Future.delayed(Duration(seconds: 1));
    print(i);
  }
}

class DataFetcher {
  Future<String> _getDataFromCloud()
async {
    // get data from cloud
    // sleep hällt das programm auf
Future delayed stoppt nur die
spezifische Funktion
    await
Future.delayed(Duration(seconds: 4));
    print("get finished");
    return "data from Cloud";
  }

  Future<String>
_parseDataFromCloud({required String
cloudData}) async {
    // parse cloud data
    await
Future.delayed(Duration(seconds: 2));
    print("data parsing finished");
    return "parsed Data";
  }

  Future<String> getData() async {
    String _cloudDataRaw = await
_getDataFromCloud();
    String _parsedData = await
_parseDataFromCloud(cloudData:
_cloudDataRaw);

    //? alternative schreibweise :
    /*
    String _parsedData = await
_getDataFromCloud().then((_cloudDataRaw)
async {
      return await
_parseDataFromCloud(cloudData:
_cloudDataRaw);
    });
    */

    return _parsedData;
  }}
```

2. Streams

```
import 'dart:async';

void main() {
  Stream numberStream =
NumberGenerator().getStream.asBroadcastS
tream();
  //! broadcast -> ermöglicht mehrere
subscriber
  StreamSubscription sub1 =
numberStream.listen((event) {
    print(event);
  },
  onDone: () {},        //! was soll
passieren wenn stream geschlossen
  onError: (error) {},  //! was soll bei
einem error passieren
```

```
  cancelOnError: false  //! hey ich will
nach einen error weiter zuhören
  );
  /* sub1.pause(); //!pause
  sub1.resume(); //!weiter
  sub1.cancel(); //! subscription
beenden */

  StreamSubscription sub2 =
numberStream.listen((event) {
    print("sub2  :  $event");
  });
}

class NumberGenerator {
  int _counter = 0;

  StreamController<int> _controller =
StreamController<int>();

  Stream<int> get getStream =>
_controller.stream;

  NumberGenerator() {
    Timer.periodic(Duration(seconds: 1),
(timer) {
      _controller.sink.add(_counter);

      _counter++;
    });
  }
}
```

XVII. Dateihandling

1. Datei in Dart schreiben

Fundstelle: https://dart-tutorial.com/file-handling-in-dart/write-file-in-dart/

a) Einleitung

In diesem Abschnitt erfahren Sie, wie Sie eine Datei in der Dart-Programmiersprache schreiben, indem Sie die **File**-Klasse und die **writeAsStringSync()**-Methode verwenden.

b) Datei in Dart schreiben

Lassen Sie uns eine Datei mit dem Namen **test.txt** im selben Verzeichnis Ihres Dartprogramms erstellen und etwas Text hineinschreiben.

```
dart-Programm zum Schreiben in eine
Datei
import  'dart:io';

void main() {
  //open file
File  file = File('test.txt');
  //In Datei schreiben
  file.writeAsStringSync('Willkommen bei
test.txt Datei.');
  print('Datei geschrieben.');
}
```

Hinweis: Wenn Sie bereits Inhalte in **test.txt** Datei haben, wird diese entfernt und durch den neuen Inhalt ersetzt.

c) Fügen Sie neue Inhalte zu vorherigen Inhalten hinzu

Sie können **FileMode.append** verwenden , um dem vorherigen Inhalt neuen Inhalt hinzuzufügen. Angenommen, test.txt Datei enthält bereits Text.

Lassen Sie uns nun neue Inhalte hinzufügen.

```
//dart-Programm zum Schreiben in eine
vorhandene Datei
import  'dart:io';

void main() {
  //open file
File  file = File('test.txt');
  //In Datei schreiben
  file.writeAsStringSync('\nDies ist ein
neuer Inhalt.', mode: FileMode.append);
  print('Herzlichen Glückwunsch!! Neue
Inhalte werden zusätzlich zu den
vorherigen Inhalten hinzugefügt.");
}
```

d) CSV-Datei in Dart schreiben

Im folgenden Beispiel werden wir den Benutzer bitten, den Namen und die Telefonnummer von 3 Schülern einzugeben und in eine **CSV-Datei** mit dem Namen **students.csv** zu schreiben**.**

```
//Dart-Programm zum Schreiben in eine
CSV-Datei
import  'dart:io';

void main() {
  //open file
File  file = File("students.csv");
  //In Datei schreiben

file.writeAsStringSync('Name,Telefon\n')
;
  for (int i = 0; i < 3; i++) {
    //Name der Benutzereingabe
    stdout.write("Geben Sie den Namen
des Schülers ein ${i + 1}: ");
    String? name = stdin.readLineSync();
```

```
    stdout.write("Geben Sie das Telefon
des Schülers ${i + 1} ein: ");
    //Benutzereingabe Telefon
    String? phone =
stdin.readLineSync();

file.writeAsStringSync('$name,$phone\n',
Modus: FileMode.append);
  }
  print("Herzlichen Glückwunsch!! CSV-
Datei erfolgreich geschrieben.");
}
```

Ausgabe:

```
Geben Sie den Namen des Schülers 1 ein:
John
Geben Sie die Telefonnummer von Schüler
1 ein: 1234567890
Geben Sie den Namen des Schülers 2 ein:
Note
Geben Sie das Telefon von Student 2 ein:
0123456789
Geben Sie den Namen des Schülers 3 ein:
Elon
Geben Sie das Telefon von Schüler 3 ein:
0122112322
Glückwunsch!! CSV-Datei erfolgreich
geschrieben.
```

students.csv Datei sieht folgendermaßen aus:

```
Name,Telefon
John,1234567890
Marke,0123456789
Elon,0122112322
```

Hinweis: Sie können jeden Dateityp mit der **writeAsStringSync()**-Methode erstellen. Zum Beispiel **.html**, **.json**, **.xml** usw.

2. Eine Datei in Dart einlesen

XVIII. Isolate

```
import 'dart:isolate';

void main(){
  Isolate.spawn<IsolateModel>(heavyTask,
IsolateModel(35000, 500));
}

void heavyTask(IsolateModel
isolateModel){
  int total = 0;

  for(int i = 1; i <
isolateModel.iteration; i++){
    total += (i +
isolateModel.multiplier);
  }
  print(total);
}

class IsolateModel{
  final int iteration;
  final int multiplier;
  IsolateModel(this.iteration,
this.multiplier);
}
```

XIX. – leer-

XX. Weiterführende Tipps

Sie können die Codebeispiele über die Schaltfläche *Materialien herunterladen* oben oder unten in diesem Tutorial herunterladen.

Wenn Sie bereit sind, in Flutter einzutauchen, lesen Sie unser Tutorial "Erste Schritte mit Flutter".

Einen umfassenderen Einblick in Flutter erhalten Sie in unserem Buch Flutter Apprentice.

Sie können über die Grundlagen hinausgehen und mit unserem Buch Dart Apprentice mehr über objektorientierte Programmierung und asynchrone Programmierung erfahren.

Schauen Sie sich auch die offizielle Dart-Dokumentation an.