Extension

In Dart ermöglicht eine Extension, bestehende Klassen mit neuen Methoden und Eigenschaften zu erweitern, ohne die ursprüngliche Klasse zu verändern.

Situationserklärung

In Dart gibt es keine eingebaute Methode, um zwei Elemente innerhalb einer Liste direkt auszutauschen. Ich stehe jedoch vor der Aufgabe, eine App zu entwickeln, die häufig Befehle zum Austausch von Listenelementen benötigt. Anstatt jedes Mal eine separate Funktion dafür zu schreiben, möchte ich eine Lösung, die dies effizienter macht.

Hier kommen **Extensions** ins Spiel: Mit einer Extension kann ich der `List`-Klasse oder jeder anderen eingebauten Dart-Klasse zusätzliche Methoden hinzufügen, wie zum Beispiel eine Methode zum Austauschen von Elementen. Diese Lösung ist nicht nur auf Listen beschränkt, sondern kann auch auf alle anderen eingebauten Klassen in Dart sowie auf Widgets in Flutter angewendet werden.

Bauschritte

Um eine Extension zu erstellen, muss man zunächst das Schlüsselwort `extension` verwenden. Danach folgt der Name der Extension. Anschließend kommt das Schlüsselwort `on`, gefolgt von der eingebauten Klasse, auf der die Extension basieren soll.

Danach öffnet man geschweifte Klammern `{}`, und innerhalb dieser Klammern kann man die neuen Methoden wie Klassenmethoden definieren.

Ein wichtiger Hinweis: Innerhalb der geschweiften Klammern befindet man sich tatsächlich in der bereits gewählten Klasse (in unserem Fall die `List`-Klasse). Das bedeutet, dass alle Eigenschaften dieser Klasse direkt oder über `this` zugänglich sind. Zum Beispiel kann die Länge der Liste einfach über `this.length` oder `length` erreicht werden. Die Elemente der Liste können durch `this[index]` nicht nur gelesen, sondern auch verändert oder gelöscht werden.

Die neuen Methoden können entweder die Klasse selbst verändern, was dann `void`-Methoden wären, oder ein neues Objekt erstellen und zurückgeben, ähnlich wie es bei normalen Klassenmethoden der Fall ist.

Beispiel

```
extension ListAustauschExtension on List {
 ///- [indexTauschen] Tauscht zwei Elemente in der Liste aus, durch Angabe der
Indizes.
 void indexTauschen(int index1, int index2) {
    // Überprüfe, ob die Indizes innerhalb des gültigen Bereichs liegen.
    if (index1 < 0 || index1 >= length) {
      // Wirf eine RangeError-Instanz, wenn index1 außerhalb des gültigen Bereichs
liegt.
      throw RangeError('index1 ist außerhalb des gültigen Bereichs');
    } else if (index2 < 0 || index2 >= length) {
      // Wirf eine RangeError-Instanz, wenn index2 außerhalb des gültigen Bereichs
```

```
liegt.
      throw RangeError('index2 ist außerhalb des gültigen Bereichs');
      // Überprüfe, ob die Indizes gleich sind.
    } else if (index1 == index2) {
      return;
    }
    // Tausche die Elemente an den Indizes index1 und index2.
    final temp = this[index1];
    this[index1] = this[index2];
    this[index2] = temp;
  }

  ///- [wertTaushen] Tauscht zwei Werte in der Liste aus.
  void wertTaushen(var wert1, var wert2) {
    // Finde die Indizes der Werte in der Liste.
    final index1 = indexOf(wert1);
    final index2 = indexOf(wert2);
    // Überprüfe, ob die Werte in der Liste enthalten sind.
    if (index1 == -1 || index2 == -1) {
      // Wirf eine ArgumentError-Instanz, wenn ein Wert nicht in der Liste
enthalten ist.
      throw ArgumentError('Ein Wert ist nicht in der Liste enthalten');
    }
    // Tausche die Elemente an den Indizes index1 und index2.
    final temp = this[index1];
    this[index1] = this[index2];
    this[index2] = temp;
  }
}

 void main(){

  final List<int> numbers = [2, 6, 9, 4, 11, 66, 9, 44];
  print(numbers); // [2, 6, 9, 4, 11, 66, 9, 44]
  numbers.indexTauschen(1, 5);
  print(numbers); // [2, 66, 9, 4, 11, 6, 9, 44]
  numbers.wertTaushen(6, 66); // [2, 6, 9, 4, 11, 66, 9, 44]
 }
```

Wichtiger Hinweis: Bereits eingebaute Methoden der Klasse können in einer Extension nicht überschrieben werden.

```
  void add(var value) {
    insert(0, value); // Füge das Element am Anfang der Liste ein.
  }
   void main(){

  final List<int> numbers = [2, 6, 9, 4, 11, 66, 9, 44];
  print(numbers.add(100)); // [2, 6, 9, 4, 11, 66, 9, 44, 100] Element kommt immer
noch am Ende
```

```
  }
```

Operationen durch Extension erstellen

Mit einer Extension kann man nicht nur neue Methoden hinzufügen, sondern auch Operatoren definieren.

Situationsbeschreibung

Wir wollen es ermöglichen, dass Listen durch das `*`-Symbol mit einem Integer multipliziert werden können, sodass jedes Element der Liste mit diesem Integer-Faktor multipliziert wird. Dies soll jedoch nur für Listen vom Typ `int` gelten.

Die Schritte zum Erstellen dieser Operator-Overloading-Funktion sind sehr ähnlich zum Erstellen neuer Methoden. Anstelle eines Methodennamens verwendet man jedoch das Schlüsselwort `operator` gefolgt von dem zu überladenden Operator-Symbol. Der Rest bleibt gleich.

Beim Erstellen der Extension selbst sollte man die `List` als `List<int>` definieren:

```
extension Operationen on List<int> {}
```

Wichtiger Hinweis: Bereits eingebaute Operatoren für eine Klasse (in unserem Fall die List-Klasse) können zwar neu definiert, aber nicht überschrieben werden. Das bedeutet, wenn du eine +-Funktionalität für die List-Klasse erstellen möchtest, erhältst du keinen Kompilierungsfehler, aber dein Code wird nicht wie erwartet funktionieren, da der existierende +-Operator nicht überschrieben werden kann.

```
extension Operationen on List<int> {
  // Fügt ein Element am Anfang der Liste ein.
  // diese Methode ist sinnlos, da add() bereits in der Klasse List definiert ist.
  // Und man kann es auch nicht überschreiben.
  void add(var value) {
    insert(0, value);
  }
 // Berechnet die Summe aller Elemente in der Liste.
 // Diese Methode ist exclusive für List<int> definiert.
 // Wenn man sie auf einer Liste vom Typ List<dynamic> aufruft, kriegt man einen
Fehler.
  int addieren() {
    // reduce() ist eine Methode, die auf einer Liste angewendet wird und ein
einzelnes Ergebnis zurückgibt.
    return reduce((value, element) => value + element);
  }
  // Multipliziert jedes Element in der Liste mit einem Faktor.
  // Diese Methode ist exclusive für List<int> definiert.
  // diese Methode ist sinnvoll, weil * Operator nicht auf List<int> definiert
ist.
  List<int> operator *(int factor) {
    return map((e) => e * factor).toList();
```

```
  }
 // Dividiert jedes Element in der Liste durch einen Divisor.
  // Diese Methode ist exclusive für List<int> definiert.
  // diese Methode ist sinnvoll, weil / Operator nicht auf List<int> definiert
ist.
  List<int> operator /(int divisor) {
    return map((e) => e ~/ divisor).toList();
  }
 // Addiert einen Factor zu jedem Element in der Liste.
 // diese Methode ist sinnlose, weil + Operator bereits auf List<int> definiert
ist.
  List<int> operator +(int factor) {
    return map((e) => e + factor).toList();
  }

}
void main(){
  List<int> intList = [2, 6];
  intList = intList * 2;
  print(intList); // [4, 12]

  intList = intList / 4;
  print(intList); // [1, 3]

  intList=intList+100; // Error: The operator '+' kann nicht auf den Typ
'List<int>' und 'int' angewendet werden.
  // Die Überschreibung des Operators '+' ist nicht überschreibbar.

  intList=intList+[100]; // [1, 3, 100]
    print(intList); // [4, 12, 100,100] List hat bereits ein + Operator, der zwei
Listen zusammenfügt.

}
```

Extension Operatoren-Overloading mit verschiedenen Funktionalitäten

Situationsbeschreibung:

Ich möchte eine Extension für die `List<dynamic>`-Klasse erstellen, mit der die `*`- und `/`-Operatoren wie folgt definiert werden:

- Wenn eine Liste mit einem Integer multipliziert wird, wird die Liste entsprechend des Faktors dupliziert. Beispiel:

  ```
  List<dynamic> dynamicList = [1, 'ww', 4, true];
  print(dynamicList * 2) // [1, 'ww', 4, true, 1, 'ww', 4, true];
  ```

Wenn die Liste durch einen Integer geteilt wird, wird die Liste entsprechend des Teilers verkürzt und gibt nur den ersten Teil zurück. Beispiel:

```
List<dynamic> dynamicList = [1, 'ww', 4, true];
print(dynamicList / 2) // [1, 'ww'];
```

Zusätzlich möchte ich eine weitere Extension speziell für List_int erstellen, um die gleichen Operatoren (* und /) exklusiv für Integer-Listen zu überladen:

Wenn eine List-Int mit einem Faktor multipliziert wird, wird jedes Element der Liste mit diesem Faktor multipliziert. Beim Teilen durch einen Integer wird jedes Element der Liste durch diesen Wert geteilt.
Kurzfassung: Ich möchte, dass diese beiden Funktionalitäten je nach Listentyp unterschiedlich funktionieren.
Lösung: Ich kann einfach zwei Extensions erstellen: eine für `List<dynamic>` und eine für `List<int>`. In diesen Extensions kann ich alle benötigten Operatorenmethoden implementieren.

Darüber hinaus könnte ich auch Extensions für andere Listentypen wie `List<String>` oder `List<bool>` erstellen und die gleichen Operatoren wiederverwenden. Das bedeutet, ich könnte die Operatoren `*` und `/` für verschiedene Listentypen jeweils spezifisch definieren und so die Funktionalitäten an den jeweiligen Datentyp anpassen.

Beispiel

```
// Dynamische Liste Operatoren
extension ListAustauschExtension on List {
    List operator /(int divisor) {
    // Berechne die neue Länge der Liste, indem die aktuelle Länge durch den
Divisor geteilt wird.
    final newLength = length ~/ divisor;

    // Erstelle und gib eine Teilmenge der Liste zurück, beginnend bei Index 0
    // und endend bei der berechneten Länge (auf Integer abgerundet).
    return sublist(0, newLength);
  }
  // List verdoppeln je nach dem Faktor.
  List operator *(int factor) {
    List result = [];
    for (var i = 0; i < factor; i++) {
     result. addAll(this);
    }
    return result;
  }
}
// Int Liste Operatoren
extension Operationen on List<int> {
    // Multipliziert jedes Element in der Liste mit einem Faktor.
  // Diese Methode ist exclusive für List<int> definiert.
  // diese Methode ist sinnvoll, weil * Operator nicht auf List<int> definiert
ist.
  List<int> operator *(int factor) {
    return map((e) => e * factor).toList();
  }
 // Dividiert jedes Element in der Liste durch einen Divisor.
```

```dart
  // Diese Methode ist exclusive für List<int> definiert.
  // diese Methode ist sinnvoll, weil / Operator nicht auf List<int> definiert
ist.
  List<int> operator /(int divisor) {
    return map((e) => e ~/ divisor).toList();
  }
}
void main() {
  // Dynamische Liste Operatoren
  List dynamicList = [2, 6, 9, 4];
  dynamicList = dynamicList / 2;
  print(dynamicList); // [2, 6] beim Dynamischen Typen wird die Liste sich selbst
geteilt, und nicht die Werte ihrer Elemente.
  dynamicList = dynamicList * 3;
  print(dynamicList); // [2, 6, 2, 6, 2, 6]. Die Liste wird sich selbst je nach
der Factor verdooppelt.

// Int Liste Operatoren
  List<int> intList = [2, 6];
  intList = intList * 2;
  print(intList); // [4, 12]
  intList = intList / 4;
  print(intList); // [1, 3]
}
```

Wichtiger Hinweis:

- Man kann nicht zwei Operator-Overloadings für denselben Listentyp erstellen, selbst wenn sie in unterschiedlichen Extensions definiert sind. In diesem Fall erhält man zwar keinen direkten Kompilierungsfehler, aber es wird auftreten, sobald man versucht, die betroffenen Operatoren zu nutzen. Das bedeutet, wenn du z. B. den `*`- oder `/`-Operator für `List<int>` in mehreren Extensions überlädst, wird Dart nicht wissen, welche Implementierung verwendet werden soll, was zu einem Fehler zur führt.
- Alle zuvor erwähnten Regeln für Extensions gelten für alle eingebauten Klassen in Dart, nicht nur für die `List`-Klasse.

Aufgabe

Wählen Sie eine eingebaute Klasse in Dart und fügen Sie entweder eine neue Methode oder ein Operator-Overloading dafür hinzu.