Android development

HeχTeχ

Dokumentation

Volker Huhn

20. März 2022

Inhaltsverzeichnis

Teil I.

Framework 1.0

1. Basis Elemente

1

[1] zuletzt überarbeitet: 20.03.2022

1.1. Package utils

Das Package *utils* enthält erste Interfaces für allgemeine Status Eigenschaften.

1.1.1. Status-Flags

Objekte wie die `SQLiteDatabase` implementieren die Methode `close()` aus dem Interface *java.lang.Closable*. Um zu Prüfen ob auf einem Objekt sicher zugegriffen werden darf, besitzen diese Objekte ein zusätzliches Statusflag der Form `isOpen: Blooean` oder auch isClose: Blooean.

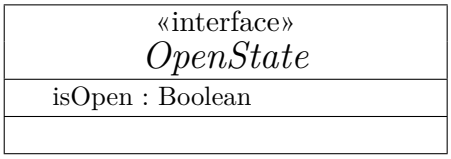


Abbildung 1.1.: Das Interface *OpenState*

Das Interface *OpenState* deklariert für diese abstrakte Attribute `isOpen: Boolean` und gibt somit an ob eine Instanz gerade geöffnet ist oder nicht.

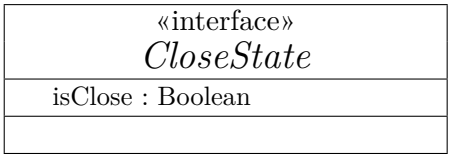


Abbildung 1.2.: Das Interface *CloseState*

Das Interface *CloseState* ist die Komplementär-Instanz zu *OpenState* und deklariert alternativ das abstrakte Attribute `isClose: Boolean` für eine negierte Statusabfrage.

Welche Betrachtung zum Einsatz kommt bleibt hier dem Anwender über lassen. Im Rahmen dieses Frameworks und den daraus resultieren den Projekte wird als Konvetion der *OpenState* in Kombination mit dem Interface *Closable* implementiert.

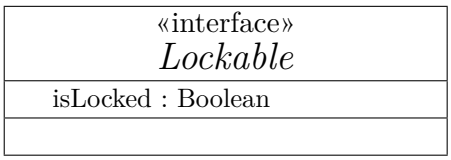


Abbildung 1.3.: Das Interface *Lockable*

Das Interface *Lockable* implementiert das abstrakte Attribute `isLocked: Boolean`. Es ermöglicht das temporäre Sichern von Daten vor unerwünschten Schreib- oder Löschvorgänge.

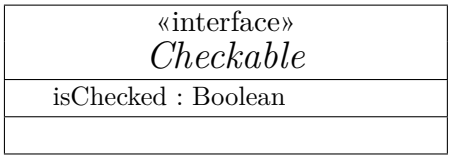


Abbildung 1.4.: Das Interface *Checkable*

Einige View-Elemente ermöglichen das binäre Ein- und Ausschalten von Optionen. Hierzu gehört beispielsweise die Switch oder Check-Views. Diese Elemente haben gemein, dass sie von einem `CompoundButton` ableiten, welches das Attribut `isChecked: Boolean` enthält. Dieses Attribute wird über das Interface *Checkable* repräsentiert und von Elementen für allgemeine `ViewContainer` implementiert.

1.1.2. Builders

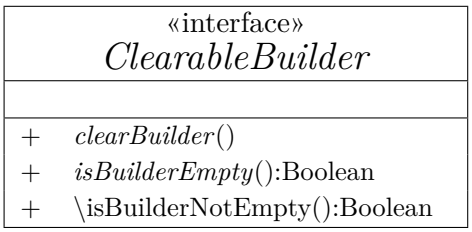


Abbildung 1.5.: Das Interface *ClearableBuilder*

Das Package *framework.maths.sets* kennt das Interface *EmptyQuantible* mit den Methoden `isEmtpy()` und `clear()` für allgemeine Datencontainer. Um möglichen Namenskonflikte vorzubeugen definiert das Unterpaket *utils* das Interface *ClearableBuilder* für Objekte die noch in ihrer Entstehung sind. Die Schnittstelle deklariert die Methoden `clearBuilder()` und `isBuilderEmpty()` und wird häufig von Dialog-Instanzen implementiert, um beispielsweise neue Datenbank Einträge zu Erzeugen.

1.2. Datum und Zeit

Das Package *times* enthält einige Basiselemente für zeitliche Abfolgen. Hierzu gehört unter Anderem das Interface *Time* für Objekte die allgemeine Zeiträume repräsentieren oder aber auch *TimeStamp* für konkrete Zeitpunkte.

1.2.1. Datum- und Zeitangaben in Android

Die gesamt Android-API wurde ursprünglich für die Programmiersprache Java ausgelegt, sodass Daum und Zeitangaben sich primär an die Java-API orientieren. Hierzu gehört unter Anderem, dass Zeiten über einen 64-Bit-Integer (Long) repräsentiert und in Millisekunden gezählt werden. Die *Java-Zeit* ($t = 0$) beginnt hier am 1. Jan. 1970 und kann über

für den aktuellen Wert jeder zeit über den Befehl `System.time.currentTimeMillis()` abgefragt werden.

1.2.2. Schaltjahren (Java)

Ein Jahr dauert etwa 365,25 Tage. Das heißt, dass alle vier Jahre das Jahr um einen weiteren Tag verlängert werden muss, damit es zu keinen Verschiebungen bei den Jahreszeiten kommt. Allgemein ist aber auch die Regel mit alle vier Jahre ein *Schaltjahr* (im eng *leap year*) nicht korrekt, sodass insgesamt zwischen einem vierjährigem hundertjährigen und einem tausendjährigem Rhythmus unterscheiden. Hier gilt jeweils mit steigender Priorität:

1. alle tausend Jahre ist **ein** Schaltjahr;
2. alle hundert Jahre ist **kein** Schaltjahr;
3. alle vier Jahre ist **ein** Schaltjahr;

Das heißt also, dass Jahre ein Jahr welches durch vier Teilbar ist in der Regel auch immer Schaltjahr ist. $\{4, 8, 12, 16, \ldots\}$, alle hundert Jahre jedoch, sodass $\{100, 200, 300 \ldots\}$ keine Schaltjahre sind. Die Höchste Priorität hat der tausendjährige Rhythmus in den dann doch wieder ein Schaltjahr ist mit $\{1000, 2000, 3000, \ldots\}$

Bestimmung der Anzahl der Schalttage

Die Bestimmung eines Schaltjahres kann dann über eine einfache Modulo-Operation in einem `if else`-Zweig erfolgen. Hier müssen die Anzahl der Tage für einen Zyklus zunächst bestimmt sein.

```
public static boolean isLeapYear(int year)
{
   boolean ret;

   if     (0==year%YEARS_BIG_CENTURY_CYCLE){ ret = true; }
   else if(0==year%YEARS_PER_CENTURY_CYCLE){ ret = false; }
   else{  ret = (0==year%YEARS_PER_CYCLE); }

   return ret;
}
```

Die Anzahl der Tage kann durch eine einfache Integer-Division bestimmt werden, mit:

```
public static long countsOfLeapYears(final int year)
{

   long ret = (year/YEARS_BIG_CENTURY_CYCLE);
   ret -= (year/YEARS_PER_CENTURY_CYCLE);
```

```
   ret += (year/YEARS_PER_CYCLE);

   if(year >= 0){ ret++; }

   return ret;
}
```

Während beim tausend- und vierjährige Rhythmus der Wert einfach aufaddiert wird, erfolgt für den im hundertjährigen Rhythmus eine Subtraktion um den vierjährigen zu kompensieren.

Bestimmung der Anzahl von Tagen (Java)

Die Anzahl aller Tage d anhand einer Jahreszahl y kann nun über die Methode `countsOfLeapYear()` bestimmt werden. Hierbei wird der erste Tag in einem Jahr auf 0 gesetzt. Dann gilt:

```
public static long convertYearsToDays(int year)
{

   long ret = 0;

   if(year!=0){
      ret = ((long) year)*DAYS_PER_YEAR;
      ret += Calendar.countsOfLeapYears(year);
      if(isLeapYear(year)&&year>=0) ret--;
   }

   return ret;
}
```

Für Jahresangaben mit $y \geq 0$ erfolgt eine Korrektur über ein Dekrement der Tage. Ansonsten erhielte man immer den zweiten Tag in einem Jahr als Rückgabewert.

Die Rückkonvertierung $d \rightarrow y$ kann dann durch eine schrittweisige Division mit dem Rest erfolgen. Hierfür wird zunächst die Anzahl der Zyklen bestimmt, um diesen als Faktor mit der Anzahl der Tage pro Zyklus zu Multiplizieren. Dabei ist auch hier wieder eine Korrektur über ein Inkrement, bzw. Dekrement nötig.

```
public static int convertDaysToYear(long days)
{
   long year, ret=0;

   //calculate years for big century
   year = (days/DAYS_PER_BIG_CENTURY_CYCLE)*YEARS_BIG_CENTURY_CYCLE;
   days = days%DAYS_PER_BIG_CENTURY_CYCLE;
   ret+=year;

   //calculate years for  century
   if(days>0)days--;

   year = (days/DAYS_PER_CENTURY_CYCLE)*YEARS_PER_CENTURY_CYCLE;
   days = days%DAYS_PER_CENTURY_CYCLE;
   ret+=year;

   if(days>0)days++;

   //calculate years for normal cycle
   year = (days/DAYS_PER_CYCLE)*YEARS_PER_CYCLE;
   days = days%DAYS_PER_CYCLE;
   ret+=year;

   if(days>=DAYS_PER_YEAR)days--;

   //calculate years of rest day
   year = days/DAYS_PER_YEAR;
   days=days%DAYS_PER_YEAR;
   ret+=year;

   if(days<0)
   ret--;

   return (int) ret;
}
```

1.2.3. Time

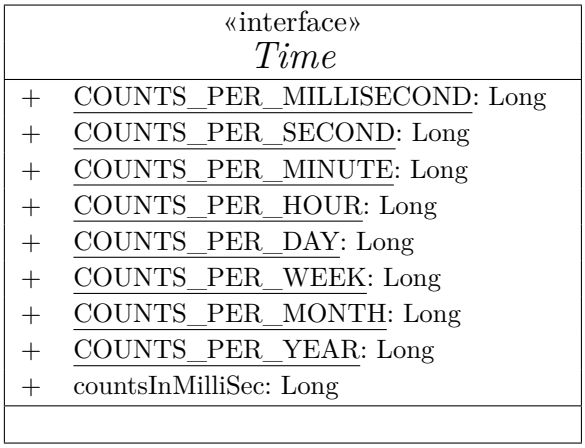


Abbildung 1.6.: Interface: *Time*

Das Interface *Time* definiert das abstrakte Attribute `countsInMilliSec: Long` sowie statische Attribute für Zeiteinheiten (z.B. Tage, Jahre) in Millisekunde.

1.2.4. TimeStamp

Abbildung 1.7.: Interface: *TimeStamp*

Die beiden Interfaces *Time* und *TimeStamp* deklariert das abstrakten Attribute `timeStampInMilliSec: Long`. Während das Interface *TimeStamp* primär für konkrete Zeitpunkte vorgesehen ist – also für Datum und Uhrzeit eines Ereignisses – kann das Interface *Time* auch für Zeiträume wie z.B. für Stoppuhren definieren.

1.2.5. TimeUnit

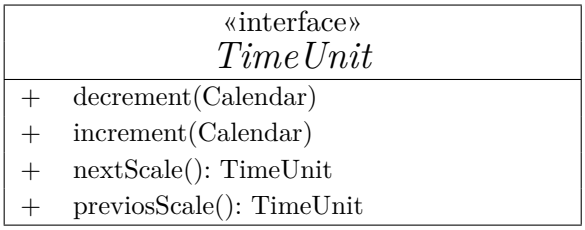


Abbildung 1.8.: Enum-Klasse: *TimeUnit*

Das Enum `TimeUnit` definiert eine Reihen von Attributen, mit denen Zeitintervalle vorgegeben werden können. Die Enum-Attribute werden beispielsweise im Package *diagrams.girds* verwendet um die Achsen eines Diagramms für unregelmäßige Gitterabstände zu skalieren. Zusätzlich definiert die Instanz nützliche Methoden um `Calendar`-Objekte entsprechend der Einheit zu In- bzw. Dekrementieren. Ebenso kann mit `next()`, bzw. `previous()` zwischen den angrenzenden Zeitskalierungen gewechselt werden.

Bezeichner	`countsInMilliSec`
secound	1000 Millisekunden
fiveSec	5-Sekunden
quarterMinute	15-Sekunden
halfMinute	30-Sekunen
minute	60-Sekunden
fiveMinute	5-Minuten-Zyklus
quaterHour	15-Minuten;
halfHour	30-Minuten;
hour	60-Minuten;
oneThirdsDay	3-Stundenzyklus;
quarterDay	6-Stundenzyklus;
day	24-Stunden;
twoDays	2 Tage
week	7-Take pro Zyklus;
halfMoth	1 oder 15 Tag in einem Monat;
month	$\frac{365.25}{12} = 30.4375$ Tage (Rest 10.5 Stunden) pro Zyklus;
quaterYear	3-Monatszyklus;
halfYear	6-Montaszyklus;
year	356,25 Tage pro Zyklus
twoYears	2 Jahreszyklus;
halfDevade	5-Jahreszyklus;
decade	10-Jahreszyklus;
halfCentury	50-Jahreszyklus;
century	100-Jahreszyklus;

Tabelle 1.1.: Attribute des Enum `TimeUnit`

1.3. Textbausteine und Formatierungen

1.3.1. Stringformatierung in der Android-API

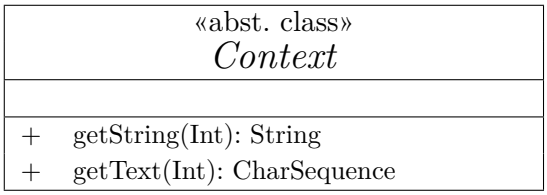


Abbildung 1.9.: Stringformatierung mit der abstrakten Klasse `Context`

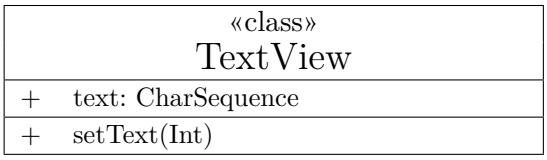


Abbildung 1.10.: Die Klasse `TextView`

1.3.2. Zahlen formatieren

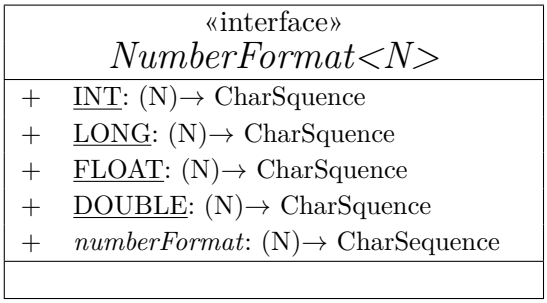


Abbildung 1.11.: Das Interface: *NumberFormat<N>*

Das Unterpakt *texts* implementiert das Interface *Numberformt* für die abstrakte λ-Funktion `numberFormat: (E):→ CharSequence`. Mit ihm können numerische Werte als λ-Funktion definiert und an Methoden als Parameter weitergereicht werden.

1.3.3. Das Unterpakte resoruces und texts

Die beiden Unterpakte *resoruces* und `texts` bilden eine abgestufte Hierarchie für typische Textelemente die allesamt als Interfaces deklariert sind.

Interfaces im Package `resoruces` enthalten allesamt das Postfix -*Reosuce* und können über eine getter-Methode mit einem Context-Parameter gelesen werden. Der Rückgabetyp ist jeweils ein *CharSequence*, sodass deren Wert direkt für die Bildschirmausgabe formatiert sind. Interfaces aus dem Unterpaket *texts* leiten in der Regel von den Resorucen ab und erweitern diese um ein abtraktes Attribute, wobei die getter-Methode per default überschrieben ist.

Datenobjekte die diese Interface implementieren können hierdurch mit Textbausteinen für eine Context-abhängige Bildschirmausgabe versehen werden, alternativ können sie aber auch als String-Resource in Datenbankobjekten gespeichert sein.

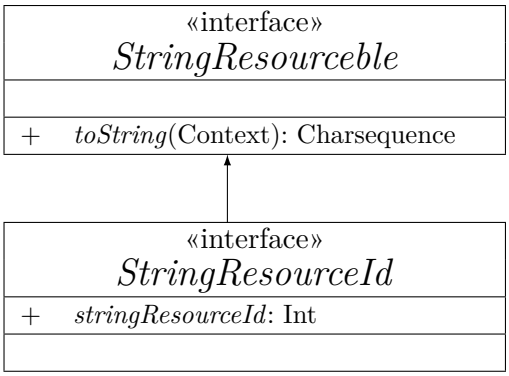


Abbildung 1.12.: Das Interface *StringResourceble* und *StringResourceId*

Das Interface *StringResourceble* überlädt die `toString()`- Methode der kotlin-Klasse für ein `Context` Parameter.

Das Interface *StringResourceId* erweitert die *StringResourceble* um das abstrakte Attribute `stringResourceId: Int` und ermöglicht eine direkte Verknüpfung zum XML-String-File. Die Resource-Id kann als Argument der Methode `setText()` einer `TextView` übergeben werden oder über ein Context-Objekt ausgelesen werden. Per Default wird auch in der *StringResourceId* auch die `toString()`-Methode definiert.

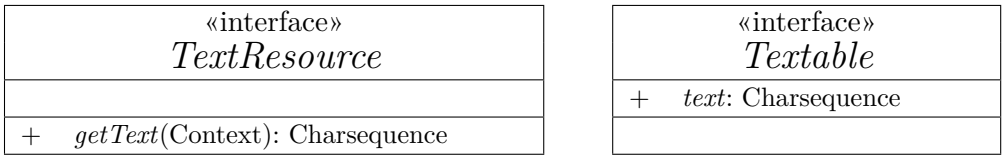


Abbildung 1.13.: Das Interface *TextResoruce* und *Textable*

Enthält ein Objekt mehr als eine Stringresoruce so gerät die `toString()`-Methode schnell an ihre Grenze. Um eine Möglichkeit für differenzierte Textausgaben zu schaffen definiert das Unterpaket *Resources* das Interface *TextResource* mit der Ausgabemethode `getText(Context): Charsequence` und das Unterpaket *texts* die Instanz *Textable* mit dem abstrakten Attribut `text: CharSequence`.

Da das Attribute *text* in einigen Objekte als Variable definiert ist, leitet diese Instanz nicht direkt von *TextResoruce* ab.

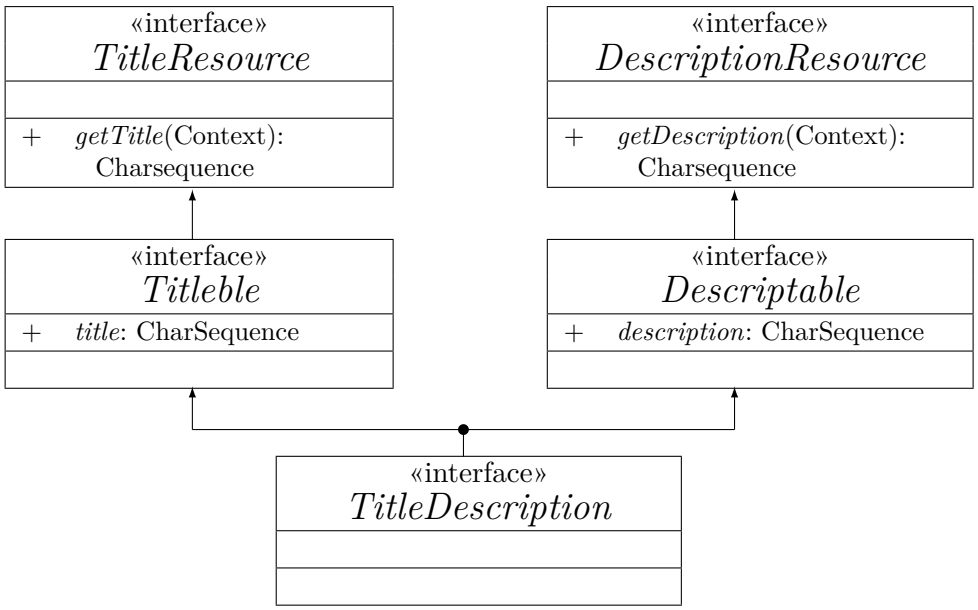


Abbildung 1.14.: Klassenbaum für *TitleDescription*

Viele Objekte enthalten für eine grafische Anwendung einen Name bzw. einen Title sowie eine optionale Beschreibung. Das Interface *TitleDescription* bildet für solche Objekte eine erster Basis und vereint die Instanzen *Titleble* (mit dem abstrakten Attribut `title: CharSeuqunce`) und *Descriptable* (mit dem abstrakten Attribut `description: CharSequence`), wobei diese Instanzen jeweils von *TitleResource* bzw. *DescriptionResoruce* ableiten.

Interface	Attribute
Enumerable	`enumCountToString: CharSequence`
Namable	`name: CharSeuquence`
SQLitePharse	`sqLiteString: String`
Symbol	`charSymbol: Char`
umlHeader	`umlTag: CharSequnce`

Tabelle 1.2.: Interfaces aus dem Paket *framework.texts*

In der Tabelle (1.2) sind weitere Interfaces mit jeweils einem abstrakten Attribut aufgeführt. Das Interface *Nameble* leitet hierbei zunächst von *NameResource* mit der Ausgabemethode `getName(Context): Charsequence` und kann von Objektklassen als alternative zu *titleble* implementiert werden. Von *Nameble* leitet direkt das Interface *EnumInterface* ab (siehe *framework.enums*), welches die Instanz um das abstrakte Attribut `ordinal: Int` erweitert. Das Interface wird hauptsächlich von Enum-Objekten implementiert und ermöglicht hierdurch eine direkte Anbindung für View-Ausgabe und der Selektion über *Listadapter*s.

Das Interface *SQlitePhrase* bildet die Basis für spätere Datenbank Anwendungen. Objekte die dieses Interface implementieren (z.B. das Enum `Relation`) enthalten einen sicheren SQlite-Teilcode, der für Parser verwendet werden kann (Sie Abschnitt 4.2).

Das Interface *Symbol* erlaubt das Kodieren von `Char`-Symbolen im UTF8-Format. Das Interface wird oft von Enum-Instanzen implementiert um beispielsweise Operatoren in View-Elemente zu repräsentieren.

1.4. Enum-Instanzen

Enum-Objekte die nicht unmittelbar einem Unterpaket zuzuordnen sind befinden sich im Unterpakt *enums*. hierzu gehören unter anderem auch Enum-Objekte, die speziell für Datenbank (siehe Abschnitt 4) Anwendungen entwickelt wurden, aber auch für allgemeine mathematische Funktionen verwendet werden können. Die meisten der hier definierten Enum-Klassen implementieren das Interface *EnumInterface*, sodass sie direkt für die Darstellung in Views – z.B. zum Selektieren einer Option – verwendet werden können. Das *EnumInterfaec* leitet von *Namable* ab und erweiter dieses um das abstrakte Attribut `ordinal: Int`.

1.4.1. Enums zum Formatieren

Das Enum `AlphaNumeric` implementiert das Interface *Numberformat* für den generischen Typ Long und ermöglicht das Kodieren einer numerischen Zahl in einer lateinischen alphabetischen Kodierung. Hierbei kann zwischen einer Groß- und Kleinschreibung unterschieden werden.

Das Enum definiert den Operator `get()` für den Rückgabetyp `Char`. Da die alphabetische Kodierung keinen 0 kennt, wird hier hierfür ein allgemeines WITHE-SPACE zurückgeben ansonsten einen Wert in $\{a, \ldots, z\}$ bzw. $\{A, \ldots, Z\}$

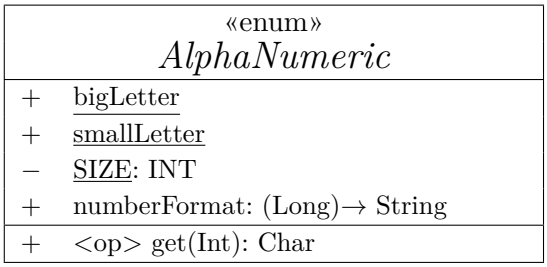


Abbildung 1.15.: Das Enum `AlphaNumeric`

ordinal	name	scale
0	tiny	0,5
1	scriptsize	0,7
2	footnotesize	0,8
3	small	0,9
4	normalsize	1,0
5	large	1,2
6	Large	1,44
7	LARGE	1,728
8	huge	2,074
9	HUGE	2,488

Tabelle 1.3.: Das Enum `TextSizeScale`

Standardmäßig sind Schriftgrößen in `TextView`s über die Layout-Resource bestimmt. Das Framework definiert in der `dimens.xml` ein Setting für Standardschriftgrößen zur Basis `normalTextSize = 20sp`.

Sollen aber die Schriftgrößen eines Textes beispielsweise über ein `Canvas`-Objekt definiert werden, so kann die Skalierung relativ zu einer festen Bezugsgröße über das Enum `TextSizeScale` mit dem Attribute `scale: Float` erfolgen. Die Namensbezeichnung und Scalenwerte sind hierbei dem LaTeX-Standard entlehnt.

1.4.2. SqLite-Enums

Das Unterpaket *enums* unterstützt verschiedene Formen von *SQLite-Schlüsselwörter*. Die sind eingeteilt in `Operator`, `Relation` und `Function`

ordinal	Name	eng	ger	sqLiteString
0	ALTER			*ALTER TABLE*
1	CREATE			*CREATE TABLE*
2	INSERT_INTO			*INSERT INTO*
3	UPDATE			*UPDATE*
4	SELECT			*SELECT*
5	SELECT_COUNT			*SELECT COUNT*
6	ORDER_BY			*ODER BY*
7	DELETE			*DELTE*
8	SET			*SET*
9	WHERE			*WHERE*
10	FROM			*FROM*
11	VALUES			*VALUS*
12	EXIST			*EXIST*
13	MAX			*MIN*
14	MIN			*MAX*

Tabelle 1.4.: Das Enum `Function`

Das Enum `Function` definiert die wichtigsten SQLite-Schlüsselwörter für das Ausführen von Befehlen. Sie enthalten jeweils Teilausdrücke und müssen vom Parser in die richtige Reihenfolge gebracht werden.

ordinal	Name	eng	ger	sqLiteString
0	STRING	string	String	*STRING*
1	INTEGER	integer	Integer	*INTEGER*
2	REAL	real	Real	*REAL*
3	BLOB	blob	Blob	*BLOB*

Tabelle 1.5.: Das Enum `DataPrimeType`

Das Enum `DataPrimeType` definiert die elementaren SQLite Datentypen `STRING`, `INTEGER`, `REAL` und `BLOB`.

ordinal	Name	eng	ger	DataTypeModify
0	none			
1	primaryKey			*PRIMARY KEY*
2	notNull			*NOT NULL*

Tabelle 1.6.: Das Enum `DataTypeModify`

ordinal	Name	eng	ger	sqLiteString
0	AND	and	und	*AND*
1	OR	or	oder	*OR*
2	NOT	not	nicht	*NOT*

Tabelle 1.7.: Das Enum `Operator`

Mit den Schlüsselwörtern aus dem Enum `Operator` können SQLite-String zu logische Verknüpfungen `AND`, `OR` und `NOT` erzeugt werden. Das Enum wird unter anderem für das Parsen von Bedingungsätze benötigt (siehe *SQLiteAttributeParser.Condition* Seite 67).

ordinal	Name	eng	ger	sqLiteString	charSymbol
0	EQUAL	equal	gleich	=	=
1	LESS	less	kleiner	<	>
2	LESS_EQ	less euqal	kleiner gleich	<=	≤
3	GREATER	greater	größer	>	>
4	GREATER_EQ	greater equal	größer gleich	>=	≥
5	NOT_EQ	not equal	ungleich	<>	≠
6	LIKE	like	ungefähr	*LIKE*	≈

Tabelle 1.8.: Das Enum `Relation`

Das Enum `Relation` ermöglicht es einzelne bestimmte werte in eine arithmetische Beziehung zu stellen. Die Instanz wird wie das Enum `Operator` zum erzeugen von Bedingungssätzen benötigt.

ordinal	Name	eng	ger	sqLiteString	charSymbol
0	ASCENDING	ascending	aufseteigend	*ASC*	↓
1	DESCENDING	descending	absteigend	*DESC*	↑

Tabelle 1.9.: Das Enum `OrderType`

Für das Parsen eines Sortieralgoritmus erfolgt mit den Schlüsselwörter ASC bzw. DESC. Die sind im Enum `OrderType` definiert.

1.4.3. Mathematische Enums

ordinal	Name	charSymbol	utf8
0	all	$\forall$	0x2200
1	exist	$\exists$	0x2203
2	nExist	$\nexists$	0x2204

Tabelle 1.10.: Das Enum `Quantor`

Das Enum `Quantor` implementiert lediglich das Interface *Symbol* und codiert die drei Quantoren $\{\forall, \exists, \nexists\}$

2. Package-maths

[1]

Das Package *maths* ist ein Kotlin-Package für allgemeine mathematischen Betrachtungen. Das Gesamtpaket enthält derzeit fünf Unterpakte, wobei *sets* und *arithmetic* vorwiegend Interfaces für allgemeine mathematische Zusammenhänge definiert. Hierzu gehört unter anderem das Gruppen- und Ringtheorem für mathematische Körper oder aber Elemente zur Beschreibung von Mengen und Tupel.

Das Unterpaket *numbers* enthält konkrete Objekte für numerische Operationen. Ebenso sind hier Objekte zum Runden von Werten implementiert oder zum definieren von Intervallsbereiche (*Range*).

Für die Beschreibung von Objekten in einem Raum enthält das Unterpaket *coordinate* ebenfalls eine Reihe von Interfaces, mit denen verschiedene Koordinatensyteme erstellt oder realisiert werden können. Ebenfalls enthalten im Unterpaket *coordinate* sind Basiselemente für Transformationsmatrizen sowie das Drehen oder Projizieren von Objekten auf einer Bildschirmoberfläche.

[1] überarbeitet: 24.12.2021

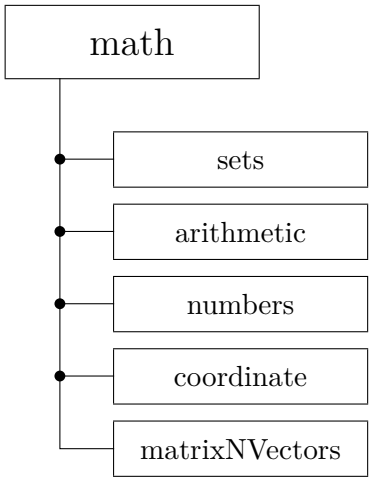


Abbildung 2.1.: Paketbaum des Package maths

Bemerkung: 2.1 *Das gesamte* math-*Package greift ausschließlich auf die Standard-Kotlin-Bibliothek zurück, sodass sie auch unabhängig von Java-Implikationen ist.*

2.1. Extension-Datei

Das Packe *framework* beinhaltet eine Extension-Datei für global verfügbare Methoden. Hierzu gehört unter anderem die Methode `divideProgression()` zur Ausführung eines rekursiven Prozesses nach dem Prinzip »*Teilen und Herrschen*«. Übergabeparameter ist hierfür ein Objekt vom Typ `IntProgression` für das zu behandelnde Intervall sowie zwei Lambda-Objekte für die Feldausgabe bzw. der auszuführenden Operation.

```
public fun <E> divideProgression(
    intProgression: IntProgression,
    field: (Int) -> E,
    op: (a: E, b: E) -> E

): E {

    require(!intProgression.isEmpty()){"int progression is empty"}

    val step = intProgression.step
    val size = 1 + (intProgression.last - intProgression.first) / step

    //if(step!=1) throw IllegalStateException("only step 1 supported")
    if (size <= 0)
```

```
    throw IllegalStateException(
        "illegal progress state for intProgress: $intProgression"
    )

    return when (size) {
        //0 -> zero
        1 -> field(intProgression.first)
        2 -> op(field(intProgression.first), field(intProgression.first + step))
        3 -> {
            val a = op(
                field(intProgression.first),
                field(intProgression.first + step)
            )
            val b = field(intProgression.first + 2 * step)
            op(a, b)
        }
        4 -> {
            val a = op(
                field(intProgression.first),
                field(intProgression.first + step)
            )
            val b = op(
                field(intProgression.first + 2 * step),
                field(intProgression.first + 3 * step)
            )
            op(a, b)
        }
        else -> {
            val splitPoint = intProgression.first + (size / 2) * step

            val aRange: IntProgression = intProgression.first..(splitPoint - 1) step step
            val bRange: IntProgression = (splitPoint)..intProgression.last    step step

            val a = this.divideProgression(aRange, field, op, zero)
            val b = this.divideProgression(bRange, field, op, zero)
            op(a, b)
        }//endElse
    }//endWhen
}//endFun
```

2.2. Statische Lambda-Funktionen

Das Haupt-Package enthält Objekte mit λ-Funktionen für das Ausführen von statischen generischen Operationen. Diese können dann als Argument für Elementaren elementare Datentypen (`Int`, `Long`, `Float`, `Double`) an Funktionen übergeben werden. Hierzu gehören unter anderem die Grundrechenarten, sowie Relationen und ein schnelle generische Typenumwandlung.

2.2.1. Typen-Umwandlung

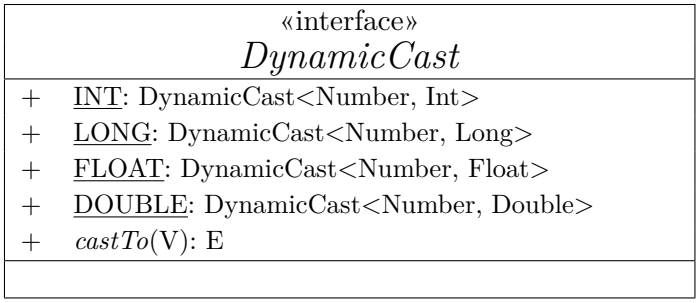


Abbildung 2.2.: Das Interface: *DynamicCast<V,E>*

Mit Hilfe der des Interfaces *DynamicCast*, kann zur Laufzeit eine erste Typenumwandlung vorgenommen werden. Für die elementaren Datentypen gilt als Übergabewert jedes Objekt, welches von *kotlin.*`Number` ableitet. Dieses stellt analog zur *java.lang.Number* die Basisklasse für die Elementartypen dar.

2.2.2. Operatoren

Um mathematische Objekte möglichst allgemein zu halten, sind die Interfaces in der Regel für generische Typen definiert. Für die Umsetzung der generischen Klassen stellt das Hauptpackage zu Beginn eine Reihe einfacher Containerklassen für statische mathematische λ-Operationen bereit.

Numerische Operationen

Das Interface *BaseOp* implementiert die vier Grundrechenarten $+, -, \cdot, /$ als zweiwertige Operatoren, sowie die unäre Negation des jeweiligen Wertes. Die Operationen sind jeweils als Lambda-Ausdruck definiert und bilden die Basis für generische Operation mit primitiven Datentypen. Hiervon abgeleitet ist unter anderem der *NumOp* für primitive Skalare sowie *VecOp* für primitive Arrays (`IntArray`, `LongArray` etc.).

Numerische Operationen

Das Interface *NumOp* erweitert das Interface *BaseOp* für skalare Datentypen und enthält darüber hinaus statische Elemente für die primitiven Typen `Int`, `Long`, `Float` und

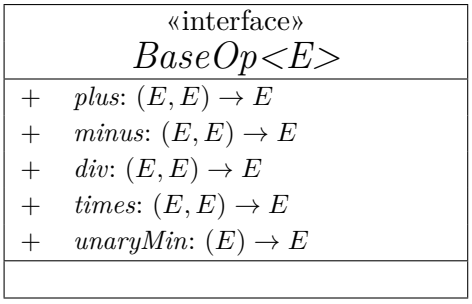


Abbildung 2.3.: Das Interface: *BaseOp*

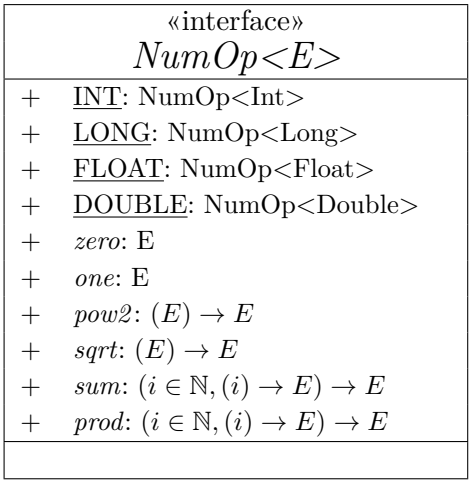


Abbildung 2.4.: Das Interface: *NumOp*<E>

`Double`.

Zu seiner Erweiterung gehören statische Werte für die neutralen Elemente `zero` bzw. `one`, eine einfache Quadrat- bzw. Wurzelfunktion sowie eine Summen und Produktfunktion über ein beliebiges Intervall $\mathbb{I}$ (`IntProgression`). Die Ausführung der beiden Funktionen erfolgt für eine Laufzeitoptimierung über die statische Methode `divideProgression`.

Die Summen- und Produktbildung als λ-Ausdruck erfolgt hier mit Hilfe der Methode `divideProgression`:

```
     override val sum: (IntProgression, (Int) -> Int) -> Int =
{ p: IntProgression, i: (Int) -> Int  ->
    if(p.isEmpty()) this.zero else divideProgression(p, i, plus)
}
```

Relationen

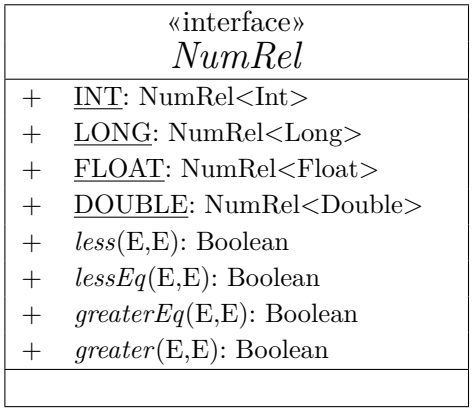


Abbildung 2.5.: Das Interface: *NumRel*

Analog zu den numerischen Operationen definiert das Interfaces *NumRel* zweiwertige Vergleichsoperationen für generische Typen. Die Prüfung auf (un)Gleichheit kann dann wie gewohnt über `equals()` erfolgen.

Funktions-Container

Die Interfaces *DynamicCast*, *NumOp* und *NumRel* ermöglichen das Ausführen numerischer Operationen mit Hilfe von λ-Funktionen. Diese können von allen Objekten implementiert werden, die generische Typen von `kotlin.Number` behandeln. Da die drei Interfaces oft zusammengehören, sind deren Elemente im Datencontainer `Operators` vereint.

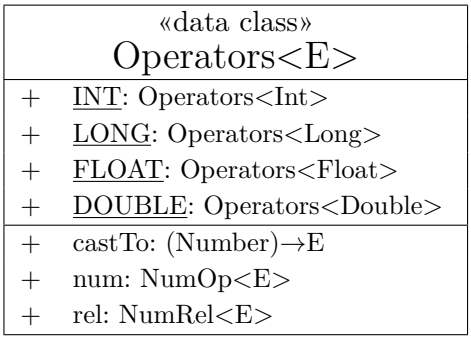


Abbildung 2.6.: Das Interface: *Operators*

2.3. Package sets

Mathematische Objekte sind allgemein über die Mengenlehre (Set) beschrieben. In der Standrad kotlin.Bib sind diese weitgehend über Interfaces wie *Array*, *ArrayList*, *Set* usw. realisiert.

Da die Interfaces der kotlin.Bib eine Vielzahl von Methoden enthalten, definiert das Package *sets* allgemeine Interfaces in denen jeweils Teilaspekte der Mengenlehre enthalten sind. Grundsätzlich gilt hier, das Interfaces in der Regel einen lesenden Zugriff ermöglichen, während konkrete Klassen Änderungen an den Daten erlauben.

Felder und Matrizen

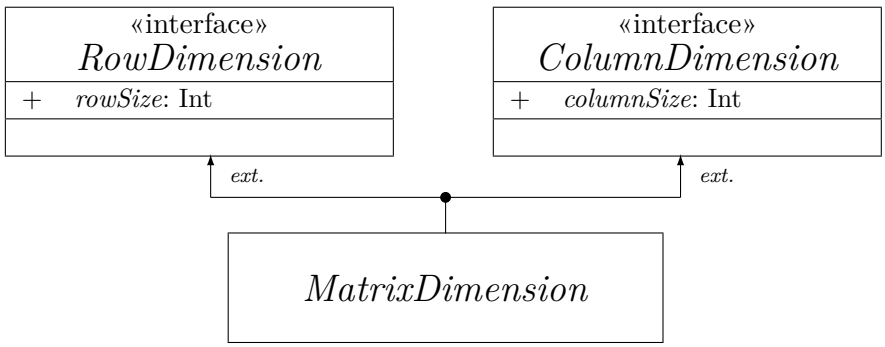


Abbildung 2.7.: Das Interface *MatrixDimension*

Die meisten Daten-Container der kotlin.Bib implementieren das Attribute `size`, während Instanzen der andorid.Bib (z.B. *ListAdapter*, *Cursor*) oft das Attribute `count` deklarieren. Da *hexTex.framework* primär für Matrizen und Tabellenanwendungen ausgelegt ist, definiert *framework.sets* zunächst drei Interfaces in denen die Zeilen (*RowDimemsion* mit `rowSize: Int`) bzw. Spaltenweite (*ColumnDimision* mit `columnSize: Int`) direkt gelesen werden kann, wobei *MatrixDimesion* beide Eigenschaften vereint.

Leermengen

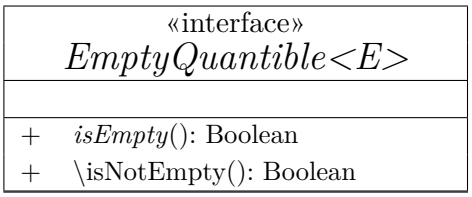


Abbildung 2.8.: Das Interface: *EmptyQuantible<E>*

Die meisten Standard Datencontainer implementieren die Methoden `isEmtpy()` bzw. funIndexisNotEmtpy() für den Rückgabetyp `Boolean`. Diese Eigenschaft von allgemeinen Container-Objekten wird über das Interface *EmptyQuantible* repräsentiert und von zahlreichen Instanzen implementiert wie *Tuple*, *Vector* oder von *framework.widget.ListAdapter*.

Interface *SetProofable*

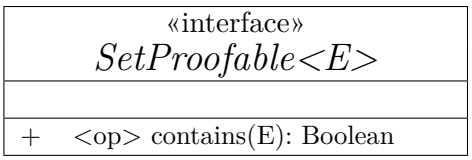


Abbildung 2.9.: Das Interface: *SetProofable<E>*

Das Interface *SetProofable* leitet vom Interfaces *EmptyQuantible* ab und erweitert dieses um den Operator `contains()`. Der Contains-Operator ermöglicht Abfragen für das Schlüsselwort `in` bzw. `!in`.

```
val set;
val element;

if(element !in set)
    set.push(element)
```

Interface *SetClearable*

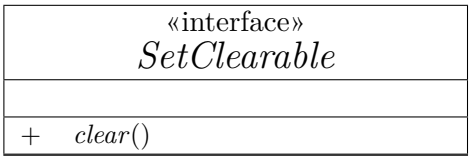


Abbildung 2.10.: Das Interface: *SetClearable*

Das Interface *SetClearable* leitet zunächst von *EmptyQuantible*ab und erweitert dieses um die abstrakte Methode `clear()`.

Vergleichsoperationen

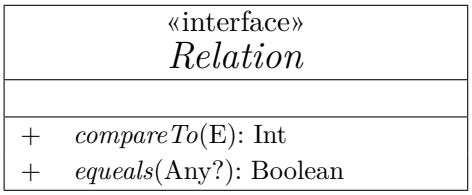


Abbildung 2.11.: Das Interface: *Relation*

Für vergleiche kennt Kotlin den Operator `compareTo()` für den Rückgabetyp Integer. Ein negativer Wert kommt hierbei einer *kleiner als*-Relation gleich, ein Positiver Wert *größer als*-Relation und die 0 einer Gleichheit. Hiermit werden dann auf Prozessorebene alle Relationsoperatoren wie kleiner, kleiner-gleich, größer-gleich und größer gebildet. Bei einfachen numerische Werten kann der Rückgabetyp dann durch einfache Subtraktion gebildet werden mit:

$$c = this - v \tag{2.1}$$

Interface *Tuple*

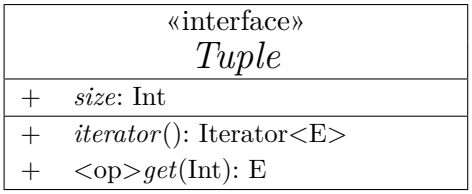


Abbildung 2.12.: Das Interface: *Tuple<E>*

Das Interface *Tuple* für generische Typen ist ein erster Datencontainer für inidizierbare Elemente. Es implementiert den Kotlin-Operator `get()` und verhält sich somit wie ein typisches Array. Zusätzlich wird das abstrakte Attribut `size` für die dazugehörige Feldlänge implementiert. Die Ausgabe eines Elements erfolgt dann wie gewohnt in eckigen Klammern. Neben dem Operator implementiert das Interface zusätzlich *Iterable*, welches bereits definiert ist und einen Default-Iterator zurückgibt.

Beispiel: 2.1 `//definition eines Tuple-Klasse`

```
class MyTuple(privte array: Array<E>): Tuple<E>
{
```

```
   override size get() array.size
   override operator get(index: Int): E
   {
      return array[index]
   }
}

val tuple = MyTuple(Array(...));

val elemenzt = tuple[0];
```

Interface *SetOperators*

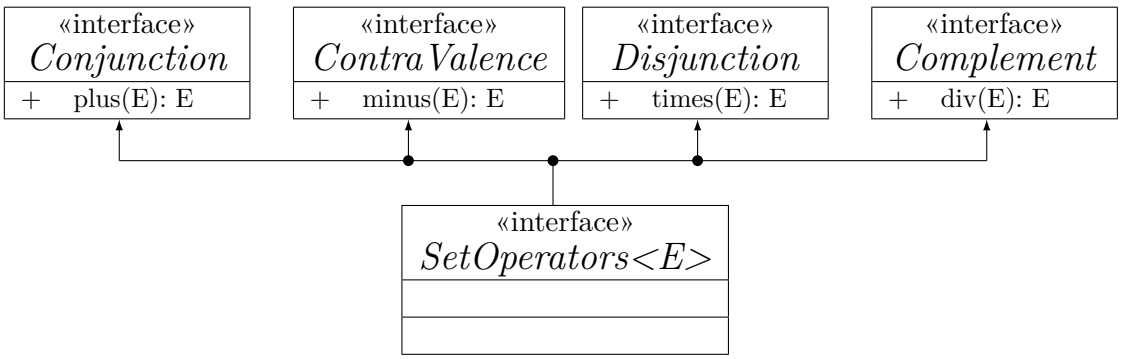


Abbildung 2.13.: Das Interface: *SetOperators*

Das Interface *SetOperators* für generische Typen leitet von den Interfaces *Conjunction* (`*`-Operator), *Disjunction* `+`-Operator, *Complement* `/`-Operator sowie von *ContraValence* (`-`-Operator) ab und ermöglicht die Anwendung von Operatoren auf Mengen.

2.4. Arithmetik

Das Unterpaket *arithmetic* enthält Interfaces für allgemeine mathematische Theoreme und deren Operatoren. Hierzu gehört unter anderem das Additionstheorem, welches im Interface *Group* und *FullAddition* definiert ist.

2.4.1. mathematische Operationen

Die Addition und Multiplikation gehören zu den wichtigsten Grundrechenarten der Mathematik. Hierbei handelt es sich aus mathematische Sicht gesehen um Objekte, die über bestimmte Eigenschaften definiert sind. Hierzu gehört zum Einen eine zweiwertige Verknüpfung () sowie das *neutrale Element* bzw. die *Inverse* einer Operation. Bei der Addition erhält man hieraus eine *Gruppe*, bei der Multiplikation spricht man folglich von einem *Ring*.

Interface *Gruppe* und *FullAddition*

Das *Gruppentheorem* ist definiert durch ein *neutrales Element der Addition* (ZERO), der *inversen* einer Addition (negate), sowie einer zweiwertigen Operation $\oplus$.

$$a \oplus 0 = a \tag{2.2}$$

$$a \oplus (-a) = 0 \tag{2.3}$$

Diese Eigenschaften sind zunächst im Interface *Group* definiert, wobei das Objekt *negate* als Default-Parameter über den `unaryMinus()`-Operator ausführt. Von *Group* leitet

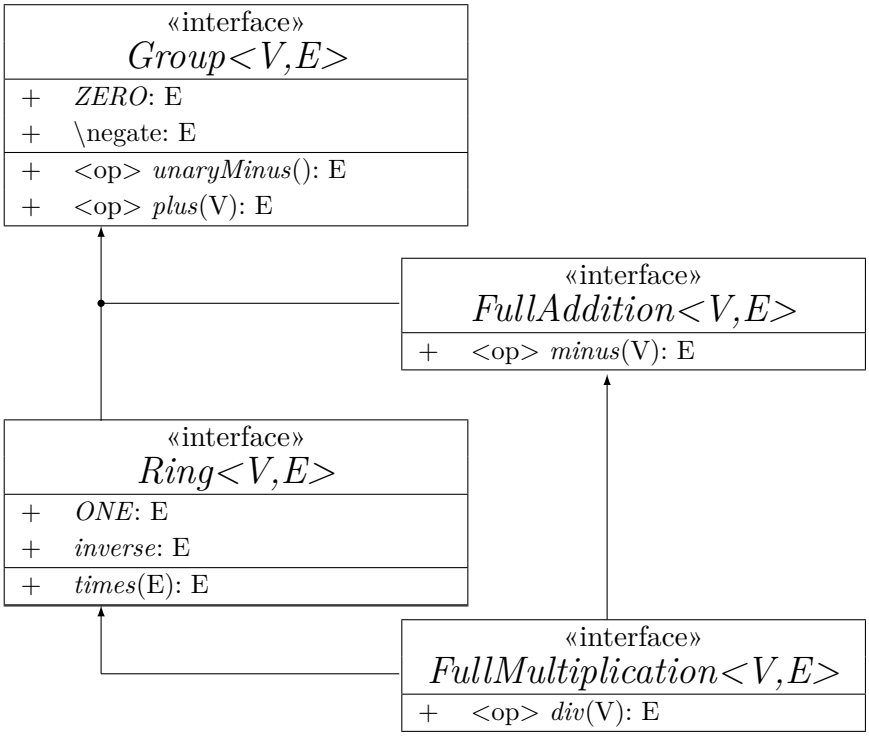


Abbildung 2.14.: Klassen-Diagramm für Addition und Multiplikation

schließlich das Interface *FullAddition* ab, sodass der Minus-Operator optional eingeführt werden kann, falls das konkrete Objekt keinen geschlossene Gruppe darstellt.

Übergabe und Rückgabe Typ können sich hierbei unterschieden, jedoch wird empfohlen hier jeweils den gleichen Typus zu verwenden.

Interface: *Ring* und *FullMultiplication*

Der *Ring* erweitert das Gruppentheorem um die Multiplikation. Auch wird ein neutrales und sowie ein inverses Element definiert und es gilt:

$$a \odot 1 = a \tag{2.4}$$

$$a \odot (1/a) = 1 \tag{2.5}$$

Analog zu *FullAddition*, definiert *FullMultiplication* die Division eines geschlossenen Rings. Das Interface kann somit als Basis für einen mathematischen Körper verwendet werden.

2.4.2. Kurzschlossoperatoren

Mit Hilfe des Präfix *Assign* können Kotlin-Operatoren in Kurzschlussoperatoren umgewandelt werden. Diese Eigenschaften sind in weiteren Interfaces definiert, wie *Sum* (`+=`) und *UpSum* (`-=`)

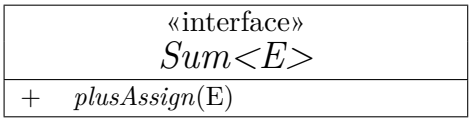


Abbildung 2.15.: Das Interface: *Sum<E>*

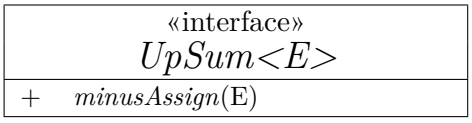


Abbildung 2.16.: Das Interface: *UpSum<E>*

Mit Hilfe der Interfaces *Sum* und *UpSum* können Elemente einer Menge hinzugefügt (also aufaddiert) bzw. entfernt (subtrahiert) werden. Diese Eigenschaften können beispielsweise von Objekten implementiert werden, die ein Integrationsverfahren über einen bestimmten Bereich ausführen sollen.

Für Objekte die gedehnt bzw. gestaucht werden sollen, existieren die beiden Interfaces *Strech* und *Upset*. Diese überladen den Kurzschlussoperator für die Multiplikation, bzw. der Division jeweils für ein `Float`, bzw. `Double`.

«interface» *Stretch*
+ *timesAssign*(Float)
+ *timesAssign*(Double)

Abbildung 2.17.: Das Interface: *Stretch<E>*

«interface» *Upset*
+ *divAssign*(Float)
+ *divAssign*(Double)

Abbildung 2.18.: Das Interface: *Upset*

2.5. Numbers

Das Package *numbers* implementiert Klassen und Interfaces rund um numerische Operationen. Hierzu gehört auch die Definition von Intervallsbereiche bzw. Methoden zum Runden von Werten.

2.5.1. Vorzeichenbetrachtung

Die Java Standardbibliothek kennt für die Klasse `BigDecimal` das Attribute `signum`, welches Auskunft über das Vorzeichen eines Wertes gibt. Diese Eigenschaft wird innerhalb des *framework*-Package von einigen View-Elemente für die Darstellung von Werten genutzt. Daher definiert das Unterpaket *framework.maths.numbers* zur Vereinheitlichung das Interface *Sign* mit seinem abstrakten Attribute `signum: Int` für die Werte: $S : \{-1, 0, 1\}$.

2.5.2. Numerisches Runden

Das Interface *Roundable* ermöglicht ein gezieltes Runden von numerischen Werten und enthält lediglich den Lambda-Ausdruck `round()` für generische Typen. Innerhalb der abstrakten Klasse `NumberRound` ist das Interface für die elementaren Datentypen `Int`, `Long`, `Float` und `Double` realisiert. Das Interface wird beispielsweise von Klassen implementiert um Zahlenwerte für die grafische Darstellung auf einen definierten Ausgabewert zu runden.

number	up	down	celling	floor	half_up	half_down	half_even
5.5	6	5	6	5	5	5	6
2.5	3	2	3	2	2	2	2
1.6	2	1	2	1	2	2	2
1.1	2	1	2	1	1	1	1
1.0	1	1	1	1	1	1	1
-1.0	-1	-1	-1	-1	-1	-1	-1

number	up	down	celling	floor	half_up	half_down	half_even
-1.1	-2	-1	-1	-1	-1	-1	-1
-1.6	-2	-1	-1	-2	-2	-2	-2
-2.5	-3	-2	-2	-3	-3	-2	-1
-5.5	-6	-5	-5	-6	-6	-5	-6

Tabelle 2.1.: Rundungsmodis von `RoundingMode`

Die abstrakte Klasse `NumberRound` implementiert zunächst das Interface *Roundable*. Die abstrakte Klasse enthält zudem ein Enum vom Typ `Roundable.Mode`, welches der Java-Bibliothek entlehnt ist. Das Enum erlaubt somit einen Wert auf bis zu sieben verschiedenen Arten zu runden.

Die abstrakte Klasse `NumberRound` definiert konkrete Objekte über eigene private Klasse, die Mittels statische Methode erzeugt werden können. Als Übergabeparameter wird die Schrittweite (*default*: 1) sowie der Modus (*default*: `halfDown`) erwartet. Darüber hinaus definiert der `NumberRound` auch noch ein Objekt zum logarithmischen Runden von Matissen.

2.5.3. Gleitkommazahlen

Die Klasse `FloatingPoint` ermöglicht das Auslesen von *Matisse* m und *Exponenten* e zu einer beliegen *Basis* b einer beliebigen *Gleitkommazahl* v. Die Instanz wird unter anderem von dem `Log10`-Rounder aus dem vorherigen Abschnitt verwendet und implementiert das Interface *Sign*.

Die einzelnen Werte sind zunächst über

$$v := m \cdot b^e \tag{2.6}$$

verknüpft, wobei das Setzen der Parameter über die Setter-Methode des abstraktes Attributs *value* erfolgt. Standardmäßig werden Matisse und Exponenten über den binären Wert zur Basis 2 ausgelesen. Für die Umwandlung zu einer beliebigen Basis b gilt zunächst allgemeingültigen Zusammenhang:

$$m_a \cdot a^x = m_a b^{x \cdot \log_b(a)}$$

Das Ergebnis im Exponent kann nun auf den nächst ganzzahligen Wert abgerundet werden mit $x \cdot \log_b(a) = x_{rest} + x_{floor}$, sodass nun gilt:

$$m_a \cdot a^x = \underbrace{m_a b^{x_{rest}}}_{m_b} \cdot b^{x_{floor}} \tag{2.7}$$

wobei m_b auch direkt aus v gebildet werden kann mit:

$$m_b = v \cdot \frac{b^{x_{rest}}}{a^x} \tag{2.8}$$

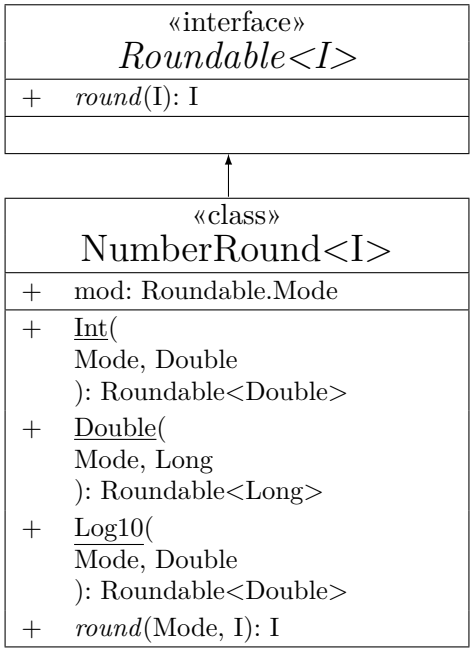


Abbildung 2.19.: Objekte zum numerischen Runden

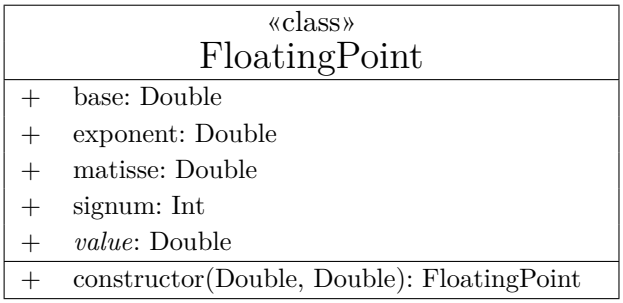


Abbildung 2.20.: Die Klasse: *FloatingPoint*

Da das Java Math-Objekt lediglich den natürlichen Logarithmus ln und den zehner Logarithmus log_{10} kennt, erfolgt der Logarithmus für eine beliebige Basis b über die Identität:

$$\log_b(n) = \frac{\ln(b)}{\ln(n)} \tag{2.9}$$

Somit kann eine Gleitkommazahl für beliebige Exponenten erstellt werden. Mit dem Klassenobjekt `FloatingPoint` kann eine Gleitkommazahl erstellt und über seine `getter`-Methoden ausgelesen werden. Wird ein neues Objekt erzeugt, so kann die Basis und der erster erste Wert beliebig definiert werden. Als Default-Wert gilt dann:

$$b = 10.0$$
$$v = 1.0$$

2.5.4. Numerisches Zählen

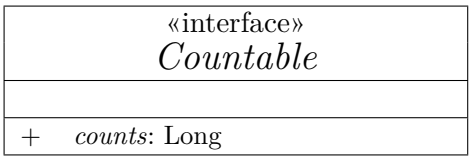


Abbildung 2.21.: Das Interface: *Countable*

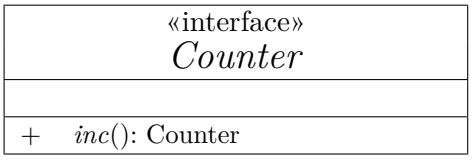


Abbildung 2.22.: Das Interface: *Counter*

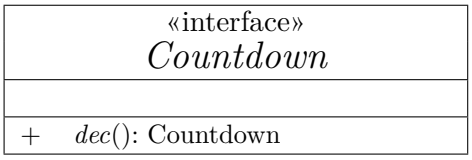


Abbildung 2.23.: Das Interface: *Countdown*

Für zählbare Objekte stellt das Framework das allgemeine Interface *Countable* mit dem abstrakten Attribut `counts` zu Verfügung. Die Instanz kann entweder für das Zählen von Elementen in Kombination mit den Interfaces *Counter* bzw. *Countdown* verwendet werden, oder aber für die Anzahl von Listenelemente, als alternative/Ergänzung zu einem `size`-Element.

2.5.5. Intervalle

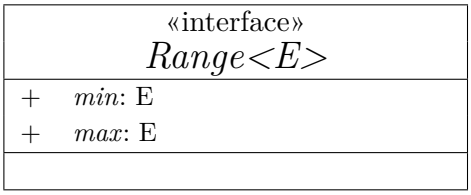


Abbildung 2.24.: Das Interface: *Range<E>*

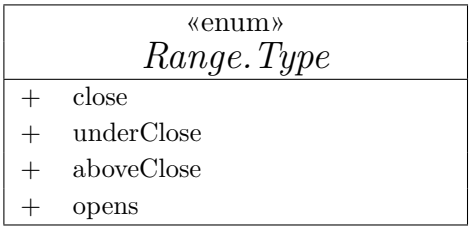


Abbildung 2.25.: Das Enum: *Range.Type*

Intervallsbereiche können über eine Instanze von Type *Range* erzeugt werden. Dieses implementiert zusätzlich ein Enum mit denen die Intervallsgrenzen näher spezifiziert werden können. Gemäß der üblichen Konventionen gilt hier:

$$a \leq x \leq b \qquad [a,b] \qquad \text{close} \tag{2.10}$$

$$a < x \leq b \qquad [a,b) \qquad \text{underClose} \tag{2.11}$$

$$a \leq x < b \qquad (a,b] \qquad \text{aboveClose} \tag{2.12}$$

$$a < x < b \qquad (a,b) \qquad \text{open} \tag{2.13}$$

Bemerkung: 2.2 *Die Bereichsklassen der std. Kotlin Bibliothek repräsentieren hauptsächlich geschlossene Intervalle.*

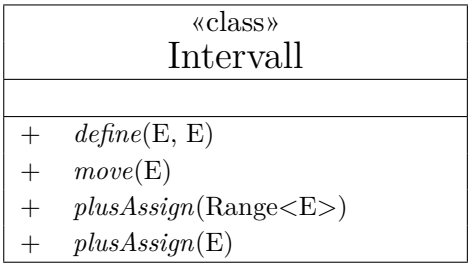


Abbildung 2.26.: Die Klasse: *Intervall*

Die Klasse `Intervall` implementiert und definiert die Interfaces *Range*, *SetOperation* und *SetClerable*. Definitionslücken sind in der gegebenen Version nicht berücksichtigt, sodass lediglich die äußeren Grenzen des Intervalls definiert sind. Zusätzlich sind neben der Implementierung der Interfaces weitere Methoden definiert, mit denen die Werte des Intervall direkt verändert werden können.

2.6. MatrixNVectors

Das Package *matrixNVectors* enthält für die primitiven Typen `Int`, `Long`, `Float` und `Double` jeweils eigene Klassen für Matrizen und Vektoren. Im westlichen sind hierbei die Eigenschaft der primitiven Typen auf mehrdimensionalen dreidimensionalen Raum übertragen worden. Das heißt, dass ein `IntVektor` mit `FloatVektor` verknüpft werden kann, und nur der jeweils höhere Datentyp (`FloatVektor`) wird zurückgegeben.

Für die Realisierung der Matrizen und Vektoren, werden schließlich die entsprechenden Arrays der primitiven Typen (`IntArray`, `LongArray`, etc.) für den internen Speicher verwendet.

2.6.1. Mathematische Grundlagen zu Matrizen

Innerhalb dies Pakets werden lediglich $m \times n$ Matrizen behandelt. Bei Anwendungen höherer Ordnung müssen ggf. eigene Objekte erstellt werden.

Transponierte einer Matrix

Das Transponieren einer Matrix entspricht der Drehung der Matrix entlang seiner Diagonalen. Entsprechend gilt:

$$A^{m\times n} \rightarrow A_T^{n\times m} \tag{2.14}$$

$$A_T = (a_{T,i,j}) = (a_{j,i}) \tag{2.15}$$

Matrix Addition

Bei der Addition bzw. Subtraktion werden die einzelnen Elemente der Matrix $A^{m\times n}$ mit der Matrix $B^{m\times n}$ verknüpft. Als notwendige Bedingung gilt hier, dass beide Matrizen die gleiche Dimension aufweisen müssen. Ist dies nicht der Fall, so wird eine `RuntimeException` ausgelöst, die von der aufrufenden Klasse ggf. behandelt werden muss.

Für die Berechnung gilt dann:

$$C^{m\times n} = A^{m\times n} \pm B^{m\times n} \tag{2.16}$$

$$(c_{i,j}) = (a_{i,j}) \pm (b_{i,j}) \tag{2.17}$$

Partielle Multiplikation

Während bei der Addition/Subtraktion einer Matrix alle Elemente einzeln miteinander verknüpft werden, existiert für die Multiplikation kein entsprechendes Äquivalent. Da bei einigen Anwendung aber genau dies gefordert ist, wird zusätzlich eine partielle Multiplikation definiert mit:

$$C^{m\times n} = A^{m\times n} \odot B^{m\times n} \tag{2.18}$$

$$(c_{i,j}) = (a_{i,j}) \odot (b_{i,j}) \tag{2.19}$$

Hierbei gilt analog, dass nur Matrizen gleicher Dimensionen verknüpft werden können.

Matrix skalieren

Eine Matrix ist im allgemeinen skalierbar. Hierbei wird jedes Element der Matrix mit einem Skalar s multipliziert.

$$B^{m\times n} = A^{m\times n}\dot{s} \tag{2.20}$$

$$(b_{i,j}) = (a_{i,j}) \cdot s \tag{2.21}$$

Matrizen-Multiplikation

Die Matrizenmultiplikation ist eine spezielle Rechenvorschrift und nur mögliche wenn die Anzahl der Zeilen einer Matrix $A^{i\times j}$, gleiche der Anzahl der Spalten einer Matrix $B^{j\times k}$ ist. Als Ergebnis erhält man schließlich eine Matrix der Form $C^{i\times k}$. Für die Rechenvorschrift gilt schließlich:

$$C^{i\times k} = A^{j\times j} \times B^{j\times k} \tag{2.22}$$

$$(c_{i,k}) = \sum_{j} a_{i,j} \cdot b_{j,k} \tag{2.23}$$

Inverse einer Matrix

Die Inverse A^{-1} einer Matrix ist nur für quadratische Matrizen der Form $A^{n\times n}$ definiert und kann durch Umformung aus einer quadratischen Einheitsmatrix E gebildet werden. Allgemein gilt hier der Zusammenhang:

$$E = A \cdot A^{-1} \tag{2.24}$$

2.6.2. Matrizen

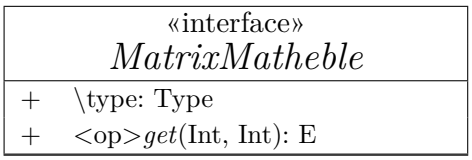


Abbildung 2.27.: Das Interface: *MatrixMatheble*

Das Basiselement für alle Matrizen ähnlichen Objekte ist die *MatrixMatheble* für beliebig generische Typen. Das Interfaces leitet zunächst von *EmptyQuantible* und *MatrixDimension* ab und erweitert dieses um das Attribute `type` und den Operator `get()` für einen zweiwertigen Ausdruck.

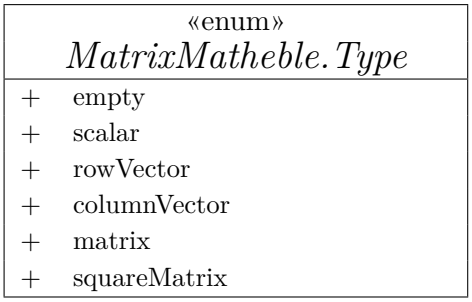


Abbildung 2.28.: Das Enum: *MatrixMatheble*.`Type`

Das Enum *MatrixMatheble*.`Type` besteht hierbei aus sechs Elementen die als Defaultwert anhand der Zeilen- und Spaltenweite ermittelt wird.

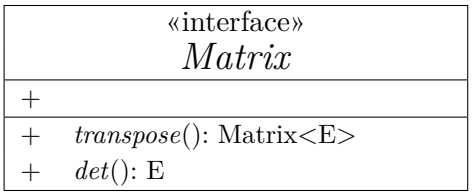


Abbildung 2.29.: Das Interface: *Matrix*

Das Interface *Matrix* leitet von *FullAddition*, *Ring*, sowie *MatrixMatheble* ab, sodass hiermit erste mathematische Operationen für Matrizen zu Verfügung stehen. Darüber hinaus erweitert das Interface die Instanzen um die Methode `transpose()` und `det()`.

Das Interface wird von den Instanzen, `MathMatrix` und `NumMatrix` direkt implementiert, wobei `NumMatrix` eine abstrakte Basisklasse für die primitive Skalare `Int`, `Long`, `Float` und `Double` darstellt und in den dazugehörigen `XXXMatrix`-Objekten realisiert sind.

NumMatrix

«abst. class» *NumMatrix<E>*	
+	*isZERO*: Boolean
+	*isONE*: Boolean
+	*divAssign*(E)
+	*timesAssign*(E)
+	*set*(Int, Int, E)
+	equales($(\mathbb{N}, \mathbb{N}) \rightarrow E$): Boolean

Abbildung 2.30.: abstrakte Klasse: *NumMatrix*

Die abstrakte Instanz `NumMatrix` implementiert die Interfaces `Matrix` sowie *NumberCast* und ist folglich auf Datentypen beschränkt die von *kotlin.*`Number` ableiten. Von ihr leiten die Instanzen `IntMatrix`, `LongMatrix`, `FloatMatrix` sowie `DoubleMatrix` (kurz `XXXMatrix`) ab.

«class» XXXMatrix	
−	zeroMat(Int, Int)→ XXX
−	oneMat(Int, Int)→ XXX
−	field: Array<XXXArray>
+	constructor(Array<XXXArray>)
+	constructor(Int, Int, (Int, Int)→ XXX)
+	constructor(MatrixMathable<Number>)
+	partialMultiply(XXYMatrix): XXZMatritx
+	partialDivision(XXYMatrix): XXZMatritx

Abbildung 2.31.: Die abstrakte Klasse: *NumMatrix*

Die `XXXMatrix` Instanzen definieren die abstrakten Methoden der Vater-Klasse vollständig und implementiert zusätzlich das Interface `Tupel` für den Typ `XXXArray` (z.B. `IntArray`, `FloatArray` etc.) Außerdem sind die Operatoren dahingehend überladen, dass sie auf alle Matrizen für primitive Typen anwendbar sind. Hierbei wird wie bei den mathematischen Operatoren der Primitiven der jeweils höhere Datentyp zurück gegeben.

Als einziges lokales Attribut enthalten die Klassen ein Feld von Typ *Array*<`XXXArray`>, welcher als Zeilenvektor dient und über die Methoden des *Tuple*-Interfaces ausgelesen werden können. Zum Erzeugen einer neuen Instanz stehen insgesamt drei Konstruktoren bereit, wobei der Default-Konstruktor direkt über das Feld erzeugt wird. Hierbei werden zugleich auch die `XXXArray`s auf ihre Spaltenweite geprüft und ggf. eine Exception ausgelöst. Alternativ kann die Matrix auch – analog zum Array – über die Zeilen- und Spaltenbreite mit einem λ-Ausdruck erzeugt werden.

Mit den Methode `partialMultiply()` und `partialDivision()` können zudem die

Elemente von zwei Matrizen multipliziert bzw. dividiert werden. Der Übergabeparameter muss hierbei die gleichen Zeilen- und Spaltendimension vorweisen.

MathMatrix

«class» MathMatrix<E>	
+	isZERO: Boolean
+	isONE: Boolean
+	set(Int, Int, E)

Abbildung 2.32.: Die Klasse: `MathMatrix`

Die Klasse `MathMatrix` ist eine allgemeine Matrizen-Objekt für alle Elemente die vom Interface *FullAddition* und *Ring* ableiten. Somit kann die Instanz alle Objekte verarbeiten wie die von `NumMatrix` ableiten oder auch vom Typ `Complex`. Analog zu `NumMatrix` wird auch dieses Objekt durch die Attribute `isZero` und `isOne` erweitert als auch den setter-Operator.

2.6.3. Vektoren

Die Basis für Vektor-Objekte bildet das Interface *Vector*. Das Interface leitet von *FullAddition*, *MatrixMatheble* sowie von *Tupel* für seine Elemente ab, wobei *MatrixMatheble* per Default einen Spaltenvektor definiert. Er lässt sich somit ohne zusätzlich transponiert werden mit einer Matrize multiplizieren. Analog zu `NumMatrizen` definiert die abstrakte Instanz `NumVector` einen allgemeinen Spaltenvektor für numerische Werte vom Typ `kotlin.Number`.

«abst. class» *NumVektor<E>*	
+	isZERO: Boolean
+	abs(): NumVektor<E>
+	lengthToDouble(): Double
+	lengthToFloat(): Float

Abbildung 2.33.: Die abstrakte Klasse: `NumVektor`

`NumVector` implementiert die Interfaces *Vector* und *NumberCast*. Darüber hinaus ist ein abstraktes ZERO-Flag deklariert, eine Funktion zur Rückgabe eines Vektors für den Absolutbetrag seiner Elemente, sowie die Ausgabe der Vektorlänge für Float- bzw. Double Typen. Somit ist sichergestellt, dass auch für Integer Typen ein hinreichend genauer Betrag ausgegeben wird.

Die Instanzen `NumVektor` sind in den Klassen `IntVector`, `LongVector`, `FloatVector` und `DoubleVector` (kurz `XXXVector`) realisiert. Außerdem definiert das Package *coordi-*

«class» XXXVektor<E>	
−	field: XXXArray
+	constructor(XXXArray)
+	constructor(Int, (Int)→XXX)

Abbildung 2.34.: Die Klasse: `XXXVektor`

nate kartesischen Objekten von Typ `IntVec3`, `LongVec3`, `FloatVec3` und `DoubleVec3`. Diese Objekte verhalten sich analog zu den Matrizen, und können beliebig kombiniert werden.

2.7. Coordinates

«interface» *Coordinate<E>*	
+	zeroInt: Coordinate<Int>
+	zeroLong: Coordinate<Long>
+	zeroFloat: Coordinate<Float>
+	zeroDouble: Coordinate<Double>
+	tupleOf(E,E,E?): Coordinate<E>
+	<op> get(Axis): E

Abbildung 2.35.: Das Interface: *Coordinate<E>*

Das Unterpaket *coordinates* enthält erste Elemente für geometrische Strukturen im Raum. Hierzu gehört das Basisinterface *Coordinate*, welches von Tuple ableitet und den getter-Operator für ein Argument vom Enum `Coordinate.Axis` überlädt. Somit können kartesische Eigenschaften direkt aus dem Objekt gelesen werden. Da nicht jede Achseneigenschaft vom späteren Objekt unterstützt werden muss, leitet *Coordinate* für das Enum `Axis` zusätzlich vom Interface *SetProofeble* ab, um eine sichere Abfrage für den Koordinatenzugriff zu ermöglichen.

Für erste Objekte enthält das Companion-Objekt einige Nullvektoren für nummerische Werte. Zudem erzeugt die statische Methode `tupleOf()` einen Datencontainer für die ersten drei Raumachsen (x, y, z), wobei die z-Komponente ein optionales Attribut darstellt.

Das Enum `Coordinate.Axis` enthält Elemente für die drei kartesischen Raumachsen x, y, z sowie für Polar/ Kugelkorrdinaten r, φ, ϑ als auch den Skalierungsparameter a wie er bei der Behandlung grafischer Anwendungen oft genutzt wird.

Das Interface *Cartesian* leitet von den Interfaces *Vector* und *Coordinatble* ab. Zusätzlich ist das Interface auf nummerische Werte vom Typ `kotlin.Number` beschränkt. Dabei sind die Numerischen Operation jeweils für den Rückgabetyp *Cartesian* überschieben,

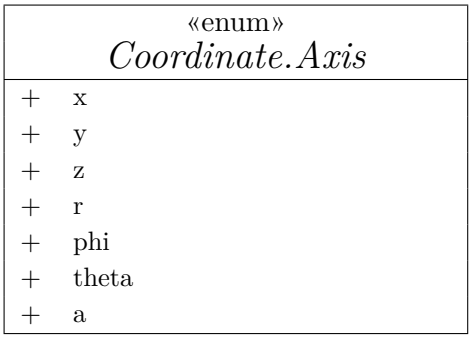


Abbildung 2.36.: Das Enum: *Coordinate.Axis*

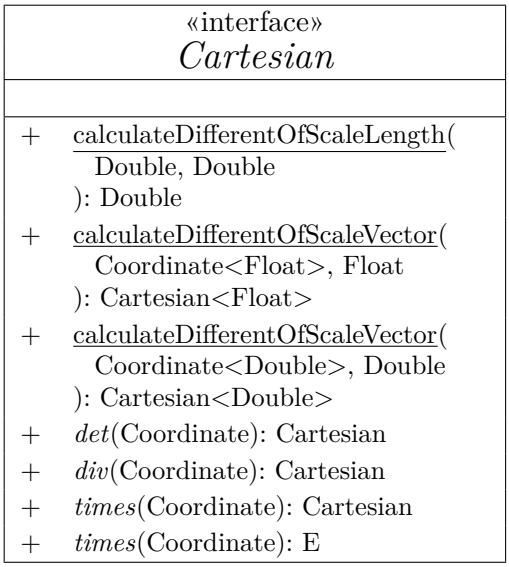


Abbildung 2.37.: Das Interface: *Cartesian*

sowie für den Parameter *Cartesian* überladen. Des weiteren erlaubt der `times`-Operator auch das Skalieren eines Vektors, sowie auch die Skalare Multiplikation zweier Vektoren. Das klassische Kreuzprodukt ist über die Methode `det()` (Determinate) deklariert.

«class» XXXVec3	
+	x: XXX
+	y: XXX
+	z: XXX
+	constructor(XXX, XXX, XXX)
+	constructor(XXX, XXX, XXX, XXX)

Abbildung 2.38.: Die Klasse `XXXVec3`

Das Interface *Cartesian* wird unter anderem von den Klassen `IntVec3`, `LongVec3`, `FloatVec3` und `DoubleVec3` (kurz `XXXVec3`) implementiert und definiert. Die Teilinstanzen definieren jeweils dreidimensionale Vektoren für die öffentlichen Attribute x, y und z. Die Objekte implementieren jeweils zwei Konstruktoren. Mit dem ersten werden die drei Raumachsen direkt definiert, der zweite ermöglicht zusätzlich das Skalieren einer Raumachse für den Parameter a.

3. Package-tables

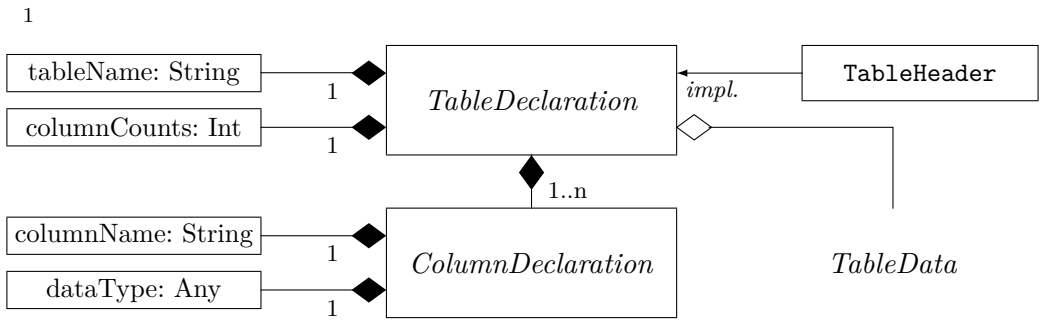


Abbildung 3.1.: Basiselemente einer Tabelle

Das Package *tables* definiert erste allgemeine Objekte zur Deklaration von Tabellen und ihren Daten. Damit möglichst früh zwischen physikalischer und grafischer Tabellendarstellungen differenziert werden kann, unterscheidet das Package bei der Deklaration zwischen `String`- und `Context` abhängigen Ausgabemethoden, wobei letztere Objektklassen am Postfix *Resoucre* erkennbar sind.

[1] überarbeitet: 24.12.2021

3.1. Spaltendeklarationen

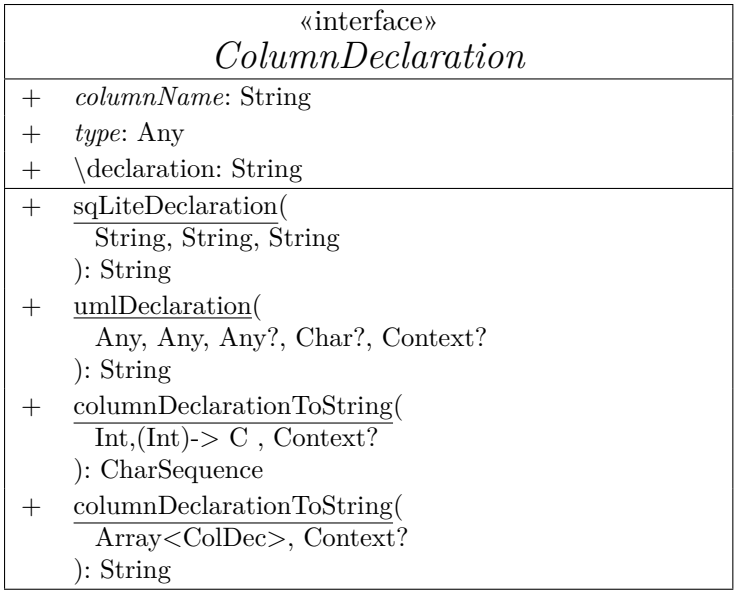


Abbildung 3.2.: Das Interface *ColumnDeclaration*

Das Interface *ColumnDeclaration* definiert einen ersten abstrakten Datencontainer für eine Spaltendeklaration und enthält die Attribute `columnName` und `type`. Während für `columnName` ein String erwartet wird, kann der Typ für jedes beliebige Objekt definiert sein. Für Datenbankanwendungen eignet sich hier das Enum `PrimeType` oder auch `DataType`.

Das Attribute `declaration` gibt hier als Voreinstellung einen String der Form: `columnName type` zurück. Darüber hinaus sind statische Methoden für die Formatierung der Deklaration definiert.

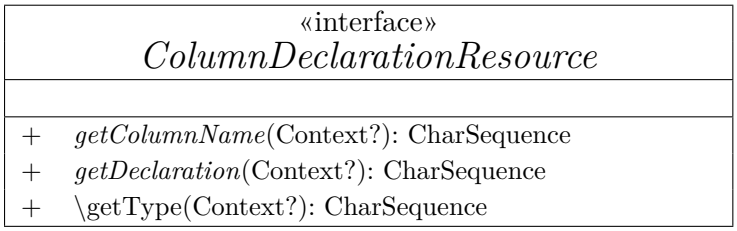


Abbildung 3.3.: Das Interface *ColumnDeclarationResource*

Das Interface *ColumnDeclarationResource* ist analog zu *ColumnDeclaration* aufgebaut, nur dass deren Attribue über eine `getter`-Methode mit einem `Context`-Objekt als Argument ausgelesen werden. Das Interface dient somit ausschließlich zur Darstellung von Deklarationen auf Bildschirmoberflächen.

3.2. Tabellendeklarationen

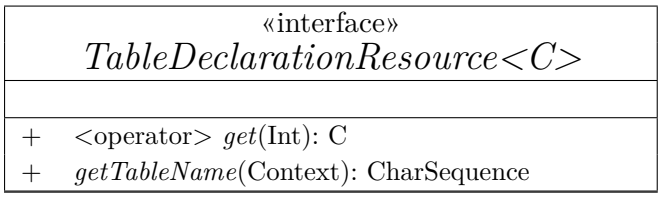


Abbildung 3.4.: Das Interfaces *TableDeclarationResource*

Die beiden Interfaces *TableDeclaration* und *TableDeclarationResource* bilden allgemeine Tabellendeklarationen und leiten zunächst vom Interface *ColumnDimension* aus dem *math*-Package ab. Spaltendeklarationen können hier über den `getter`-Operator als generisches Objekt vom Typ *ColumnDeclaration* bzw. *ColumnDeclarationResource* ausgelesen werden.

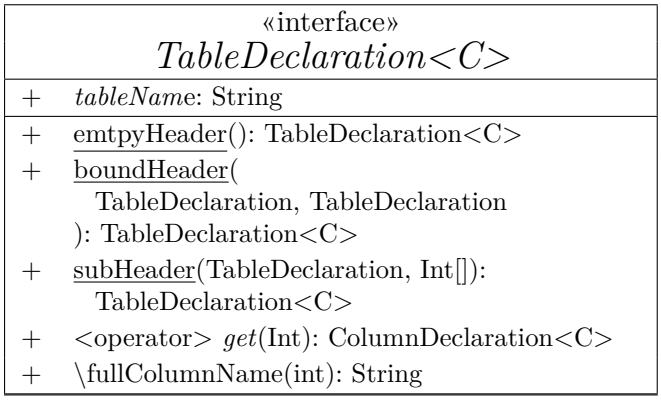


Abbildung 3.5.: Das Interfaces *TableDeclaration*

Die Methode `fullColumnName()` liefert per default eine Stringresource der Form `tabName.colName`.

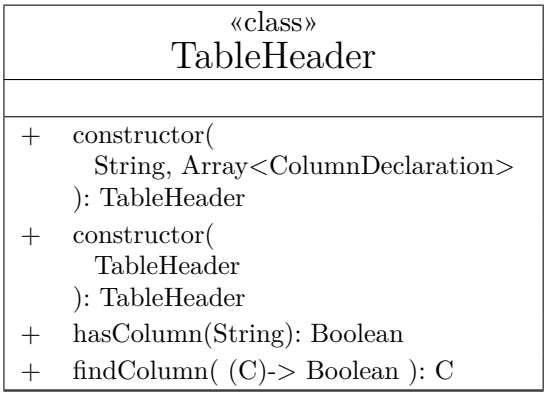


Abbildung 3.6.: Die Klasse `TableHeader`

Die `TableHeader` implementiert das Interface *TableDeclaration* und bildet die Basis für Anbindung an SQLite-Objekte. Als Konstruktor wird der Tabellenname sowie ein Array für die Spaltendeklaration erwartet, alternativ kann das Objekt aber auch über einen bereits bestehenden Header initialisiert werden.

3.3. Datendeklarationen

Für die Darstellung des Datenbereichs definiert das Package *utils* das allgemeine Interface *Id* für konkrete Datenelemente, sowie Ausgabemethoden um Daten aus einer Tabelle zu lesen.

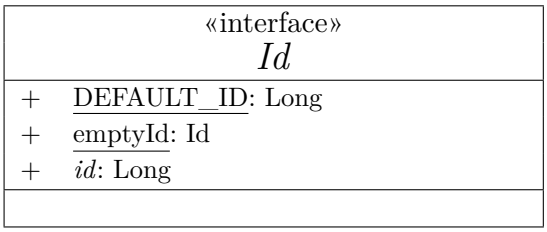


Abbildung 3.7.: Das Interfaces *Id*

Das Interface *Id* enthält das abstrakte Attribut `id: Long` und wird von allen Objekten implementiert, deren Primärschlüssel durch ein Integer-Wert von Typ `Long` definiert ist. Darüber hinaus kann es auch von Objekten implementiert werden, die sich anhand einer Id identifizieren lassen. Als statisches Attribut enthält das Objekt eine Default-Id, die auf 0 gesetzt ist.

Die Erweiterungsdatei `DatabaseExtension` implementiert darüber hinaus die statische Methode `idOf()` für den Rückgabetyp *Id*.

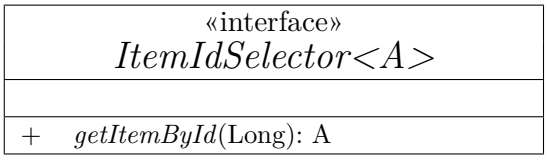


Abbildung 3.8.: Das Interfaces *ItemIdSelector*

Das Interface *ItemIdSelector* definiert die Ausgabemethode `getItemById()` für beliebige generische Typen. Mit ihm können Datenelemente Zeilenweise anhand ihrer Id zurückgegeben werden.

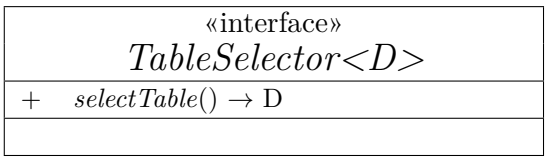


Abbildung 3.9.: Das Interfaces *TableSelector*

Zur Ausgabe eines gesamten Datensatzes eignet sich das Interface *TableSelector*. Als Rückgabetyp kann ein beliebiger Datencontainer gewählt werden wie *Array*s oder auch *Cursor*-Objekte aus dem *database*-Package.

4. Das Package databases

1

4.1. Einleitung

SQLite ist eine Textbasiertes Datenbankanwendung die standardmäßig von Android durch unterstützt wird (siehe auch: *android.database*). Die Schnittstelle zur physikalischen Datenbank-Objekt wird von der Klasse `SQLiteDatabase` repräsentiert und kann über ein `Context`-Objekt geöffnet, erzeugt bzw. gelöscht werden.

[1] überarbeitet: 27.12.2021

Das Package *framework.databases* baut auf das Package *framework.tables* (Abschnitt 3) auf und erweitert dieses – für einen objektorientierte Ansatz – um zusätzliche Text-Parser und Tabellenobjekte.

4.1.1. Paketübersicht

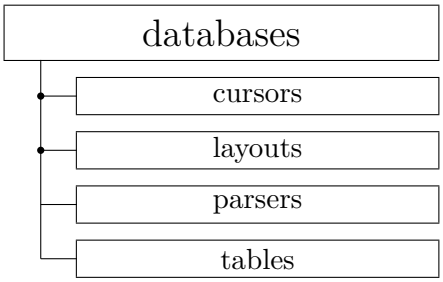


Abbildung 4.1.: Paketübertsicht

Das Hauptpackage *framework.tables* definiert zunächst allgemeine Schnittstellen für den Aufbau von Datenbank Anwendungen. Hierzu gehören z.B. Containerklassen für Tabellen eines Datenbank-Fragments, sowie abstrakte SQLite-Tabellen.

Das Unterpaket *framework.parsers* definiert Schnittstellen zum Erzeugen von SQLite-Befehle, während das Unterpaket *databases.cursors* erste Datencontainer für die Datenausgabe bereit hält. In den Unterpaketen *databases.layouts* und *databases.tables* befinden sich schließlich konkrete Tabellen und Hilfsobjekte für die Konvertierung von Daten.

4.1.2. Android - SQLite - API

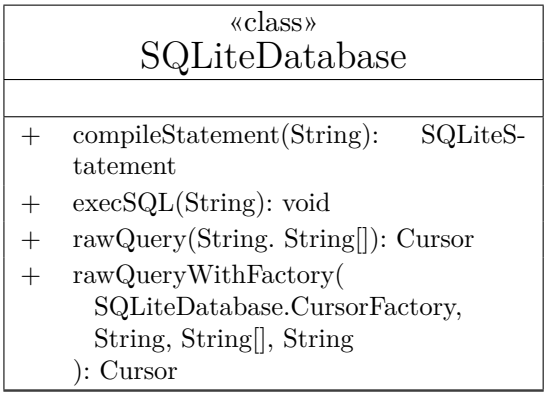


Abbildung 4.2.: Die Klasse *SQLiteDatabase*

Datenbank Operationen werden innerhalb der andorid-API über ein Objekt vom Typ `SQLiteDatabase` ausgeführt. Das Objekt enthält zahlreiche Methoden für das Ausführen von Befehlen, von denen hier nur die nötigsten aufgeführt sind. Bei `select`-Befehle

stehen zahlreiche Varianten, mit einem eigenen Parser für SQLite-Befehle bereit. Diese sind wenig flexible, sodass das Parsen im Rahmen des Framworks über ein Objekt vom Typ *SQLiteCommand* erfolgt.

Für das Auslesen von Daten wird in der Regel der Befehl `rawQuery()` bzw. `rawQueryWithFactory()` verwendet. Für Befehle ohne Rückgabetype kann auch der `execSQL()`- mit dem String als Kommando ausgeführt werden.

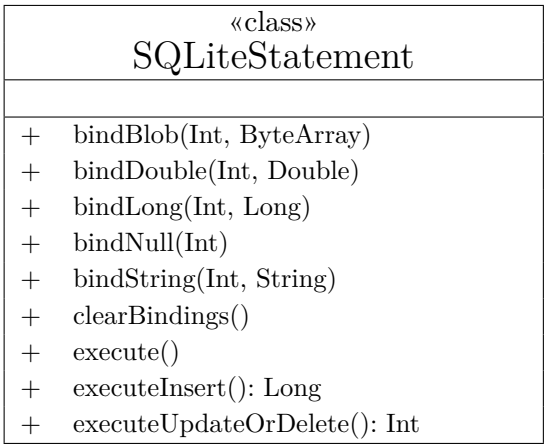


Abbildung 4.3.: Die Klasse *SQLiteDatabase*

Für INSERT- UPDATE und DELTE-Befehl definiert das Package *android.database* die Klasse `SQLiteStatement`. Hierbei handelt es sich um ein Hilfsobjekt, welches über den Methoden `compileStatement()` erzeugt wird. In Ihr können dann vor definierte Stringresourcen mit konkreten Datenelemente verknüpft werden.

Für das Erzeugen bzw. Öffnen einer `SQLiteDatabase` gibt es in der Android-API zwei Möglichkeiten:

1. Öffnen und Erstellen über ein `Context`-Object;

2. Öffnen und Erstellen über ein `SQLiteOpenHelper`;

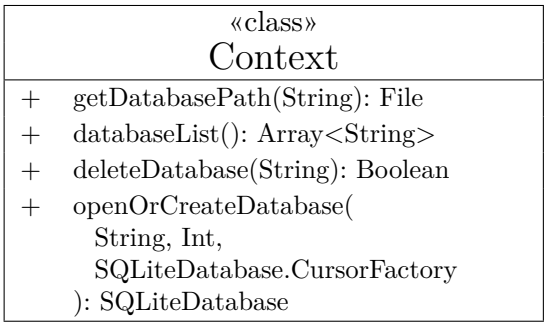


Abbildung 4.4.: SQLite-Methoden der Klasse: `Context`

Das Objekt `Context` enthält bereits einige Methoden mit denen lokale Datenbank-Objekte direkt ermittelt werden können. Neben dem Öffnen/ bzw. Erstellen einer Datenbank gehören hierzu auch das Löschen der Datenbank bzw. das Erfragen nach dem Dateipfad oder der Liste aller bekannten Datenbanken. Auf diese Weise können nun beliebig viele Datenbank-Objekte zur Laufzeit erzeugt und entfernt werden.

«class» SQLiteOpenHelper	
+	writableDatabase: SQLiteDatabase
+	*onCreate*(SQLiteDatabase)
+	*onUpgrade*(SQLiteDatabase, Int, Int)

Abbildung 4.5.: Methoden der Klasse: `SQLiteOpenHelper`

Auch wenn das `Context`-Objekt die wichtigsten Methoden für eine Datenbank bereit stellt, bietet sie jedoch keine direkte Möglichkeit die Datenbank bei der Installation einer App zu Initialisieren. Hierfür stellt die Android-Api den `SQLiteOpenHelper` als abstrakte Klasse bereit. Diese deklariert die beiden abstrakten Methoden `onCreate()` und `onUpgrade()`, sodass die Datenbank leichter gepflegt werden kann.

Wird eine neue Datenbank »`database`« über den `SQLiteOpenHelper` erstellt, so werden standardmäßig zwei physikalische Objekte erzeugt. Das zweite Objekt enthält das Postfix `-journal` und verwaltet die Versionsnummer der Datenbank. Das heißt aber auch zugleich, dass bei manuellen Änderungen über das `Context`-Objekt ggf. auch beide Dateien separat behandelt werden müssen (z.B. beim Löschen). Diese Besonderheit ist bei den Hilfsmethoden der Instanz*SQLiteAdapter* (s.u.) bereits berücksichtigt.

4.1.3. SQLiteAdapter

Die abstrakte Klasse `SQLiteAdapter` deklariert das abstrakte Element `SQLiteDatabase` und definiert Methoden für die Anbindung an das *framework.databases* (Klassendiagramm 4.6).

Der `SQliteAdapter` definiert zunächst eine Reihe von statischen Methoden. Diese benötigen in der Regel ein `Context`-Objekt und vereinfachen den Zugriff bzw. das Verwalten von eigenen `SQLiteDatabase`-Objekten.

Für die Instanzmehtoden des `SQliteAdapter`s werden für Änderungen an Tabellendeklarationen direkt Objekte vom Typ *TableDeclaration* bzw. *ColumnDelcaration* verwendet, für Funktionen im Datenbereich schließlich spezielle Parser-Objekte vom Typ *SQLiteCommand* (Abschnitt **??**).

Statische Methoden des SQLiteAdapters

Bemerkung: 4.1 *Für die statischen Methoden des Interfaces existiert die Prüfklasse `SQLiteHandlerTest`. Innerhalb eines Android-Tests werden verschiedene Datenbanken und Namenskonflikte ausgeführt.*

«abst. class» *SQLiteAdapter*
sqLite: SqliteDatabase
+ clearAllDatabases(Context): Int
+ deleteDatabase(Context: String): Int
+ isDatabaseExistInContext(Context, String): Boolean
+ isTableNameValid(String): Boolean
+ makeNameValid(String): String
+ openOrCreateDatabase(Context, String): SqLiteDatabase
+ renameDatabase(Context, String, String): Int
+ addColumns(String, <vararg> ColumnDeclaration)
+ addColumns(TableDeclaration: <varage> Int)
+ contsOf(SQLiteCommand): Int
+ createTable(TableDeclaration)
+ createTableIfNotExist(TableDeclaration)
+ createTableWithPrimaries(TableDeclaration)
+ delete(SQLiteCommand): Int
+ dropTable(String)
+ getIntOf(SQLiteCommand): Long?
+ getRealOf(SQLiteCommand): Double?
+ getStringOf(SQLiteCommand): String?
+ insert(SQliteCommannd, (SQLiteStatemant)→Unit): Long
+ select(SQLiteCommand): Cursor
+ selectByCursorFactory(SQLiteCommand, SQLiteDatabase.CursorFactory): Cursor
+ selectByIdCursor<i>(SQLiteCommand: IdCursor.Facroy<i>): IdCursor<i>
+ selectByIdCursor<i>(SQLiteCommand: (ColumnCursor)→i, Int): IdCursor<i>
+ selectByItemCursor<i>(SQLiteCommand, ItemCursorFacrory<i>): ItemCursor<i>
+ selectByItemCursor<i>(SQLiteCommand, (ColumnCursor)→i): ItemCursor<i>
+ selectByItemKeyCursor<i>(SQLiteCommand, ItemKeyKusor.Factory<k,i>): ItemKeyCursor<k,i>
+ selectByItemKeyCursor<k,i>(SQLiteCommand, (ColumnCursor)→k, (ColumnCursor)→i): ItemKeyCursor<k,i>

Abbildung 4.6.: Die abstrakte Klasse `SQLiteAdapter`

Die Methode `isDatabaseExistInContext()` besteht lediglich aus zwei Zeilen. Im ersten Schritt wird die Methode `getDatabasePath()` für den Rückgabetyp `File` aufgerufen. Dieser enthält die Methode `exist()` und liefert so Auskunft ob die physikalische Datenbank bereits existiert.

```
fun isDatabaseExistInContext(
   context: Context,
   dbName: String
): Boolean {
   val file = context.getDatabasePath(dbName)
   return file.exists()
}
```

Die Methode `openOrCreateDatabase()` delegiert den Methodenaufruf an das `Context`-Object und ist hier der vollständig halber implementiert. Als `defaultFactory` wird standardmäßig ein `CursorFactory` für den Typ `SQLiteItemCursor` übergeben, wobei das `Context`-Enable: `MODE_ENABLE_WRITE_AHEAD_LOGGING` für eine schreibare Datenbank überreicht wird.

```
fun openOrCreateDatabase(
   context: Context,
   fileName: String,
   factory: SQLiteDatabase.CursorFactory =
   SQLiteItemCursor.defaultFactory
) =
   context.openOrCreateDatabase(
      fileName,
      Context.MODE_ENABLE_WRITE_AHEAD_LOGGING, factory
   )
```

Die `delete()`-Methode für eine Datenbank ist so ausgelegt, dass sie optional auch das dazugehörige Journal-Objekt entfernt. Rückgabetyp ist daher ein Integer-Wert für die Anzahl der gelöschten Datenbanken (0,1 oder 2).

```
fun deleteDatabase(context: Context, databaseName: String): Int {

   val file: File = context.getDatabasePath(databaseName)
   val journal: File = context.getDatabasePath("${databaseName}-journal")

   var counts = 0
   if (file.exists()) {

      if (file.delete()) counts++
      if (journal.exists() && journal.delete()) counts++
   }
   return counts
}
```

Mit `clearAllDatabases()` können alle bekannten Datenbanken in einem Context-Objekt gelöscht werden. Rückgabetyp ist auch hier die Anzahl der gelöschten Objekten.

```
fun clearAllDatabases(context: Context): Int {
   var counts = 0
   val iterator =
   context.databaseList()!!.iterator()

   while (iterator.hasNext())
      if (context.deleteDatabase(iterator.next())) counts++

   return counts
}
```

Über die Methode `File.renameTo()` kann der Name eines Dateipfades direkt umbenannt werden. Für das Umbenennen der Datenbank sollte jedoch geprüft werden, ob der neue Name bereits existiert, bzw. sollte auch das Journal-Objekt bei der Umbenennung berücksichtigt werden. Dies erfolgt im `SQLiteAdapter` durch den Aufruf `renameDatabases`. Rückgabewert ist auch hier die Anzahl der erfolgreich durchgeführten Änderungen (0,1,2).

```
fun renameDatabase(
   context: Context,
   oldName: String,
   newName: String
): Int {

   var counts = 0
   val oldDatabaseFile: File = context.getDatabasePath(oldName)
   val oldDatabaseJournal: File =
   context.getDatabasePath("${oldName}-journal")

   val newDatabaseFile: File = context.getDatabasePath(newName)
   val newDatabaseJournal: File =
   context.getDatabasePath("${newName}-journal")

   if (!oldDatabaseFile.exists())
      throw RuntimeException("no database: $oldName exist!")
   else if (newDatabaseFile.exists())
      throw RuntimeException("new database: $newName is exist already!")

   val success = oldDatabaseFile.renameTo(newDatabaseFile)
   if (success) counts++

   if (success && oldDatabaseJournal.exists()) {
```

```
      if (newDatabaseJournal.exists()) {
         newDatabaseJournal.delete()
      }
      if (oldDatabaseJournal.renameTo(newDatabaseJournal)) counts++
   }
   return counts
}
```

Verwalten von SQLite-Tabellen

Das Erstellen einer neuen Tabelle erfolgt im `SQliteAdapter` mit Hilfe der Funktion `createTable()`. Als Übergabeparameter wird ein beliebiges Objekt vom Typ *Table-Delcaration* erwartet. Als SQLite-REsoruce wird hier ein String mit dem Schlüsselwort *CREATE* der Form:

```
CREATE table.name (colName0 type0, colName1, type1, ... );
```

erzeugt.

Das Entfernen einer Tabelle erfolgt über die Methode `dropTable()` mit dem dem Tabellenname als Argument. Für die SQLite-Resource wird hier das Schlüsselwort *DROP TABLE* verwendet.

```
DROP TABLE tableName;
```

Zur Laufzeit können einer existierenden Tabelle beliebig viele Spalten hinzugefügt. Hierfür stellt der `SQLiteAdapter` die Methode `addColumn()` bereit. Die Methode ist hier doppelt überladen für verschiedene Übergabe Parameter.

```
override final fun addColumns(
   tableName: String,
   vararg columns: ColumnDeclaration)
{...}

final fun addColumns(
   table: TableDeclaration<*>,
   vararg columns: Int
)
{...}
```

In der ersten Variante genügt der Tabellenname sowie eine variable Anzahl an Spaltendeklarationen für die Erweiterung der Tabelle. In der zweiten Variation wird ein Objekt vom Typ *TableDeclaration* erwartet, sowie eine variabel anzahl von Integerwerten mit jeweiligen Spaltenindizes die der Tabelle hinzugefügt werden soll.

Für den SQLite-String wird hier das Schlüsselwort *ALTER TABLE* wobei der der Spaltendeklaration das Wort *ADD* vorangestellt ist.

```
ALTER TABLE tableName ADD colName colType;
```

Der Insert-Command

Das Hinzufügen von neuen Daten erfolgt im `SQLiteAdapter` über mit der Methode `insert(): Long`. Als Übergabe Parameter wird ein Objekt vom Typ *SQLiteCommand* erwartet, sowie ein Optionales λ-Objekt zum Ausführen eines `Statements`-Objekts.

Das Einfügen eines neuen Zeileneintrags erfolgt über die SQLite-Syntax:

```
INSERT INTO  tableName (columns)
VALUES values;
```

Spalteneinträge die hier nicht explizite definiert sind, werden ggf. auf Default-Werte gesetzt. Für den ComandBuilder sind Attribute `columns` und `values` zu definieren.

Die VALUES- Attribute enthalten die Daten direkt als String-Resource. Daten die Innerhalb der SQLite als *STRING*- deklariert sind, werden zwingend mit einem Anführungszeichen »'« formatiert.

Eine sichere Formatierung der Daten kann optional über ein `Statement`-Objekt erfolgen. Hier werden die String zunächst mit einem »`?`« vor definiert und `bindInt`, `bindFloat()` etc. in die richtige Form gebracht.

Der UpdateCommand

Für den Update-Befehl werden die Attribute `sets` und `condition` benötigt. In der SQLite-Syntax folgt dann:

```
UPDATE tableName
SET sets
WHERE condition;
```

Wird keine Bedingung angeführt, so werden alle Elemente auf die neuen Werte definiert. Für die eindeutige Identifikation sollte daher immer ein Primary-Key bei einem Zeilenupdate verwendet werden. Die Kotlin-Code des Parsers ist dann analog zum insert-Befehl aufgebaut.

Der Select-Command

Das Auslesen von Daten erfolgt über einen *SELECT*-Befehl. Der Allgemeine Befehl enthält als Struktur das Schlüsselwort SELECT, sowie einen optionalen Bedingungssatz (eingeleitet mit *WHERE*) und dem Sortieralgorithmus *ORDER BY*.

```
SELECT columns
FROM tableName
WHERE conditions
ORDER BY column_1, column_2;
```

Der Select-Befehl kann bei Bedarf auch als Teil einer *WHERE EXIST*-Bedingung über den Methodenaufruf `addCondition()` des *CondtionBuilder*s genutzt werden. Hierbei wird für den Bedingungssatz ein String der Form:

```
cond0 and cond1 and  EXIST (select CMD) and cond3 ..
```

geparst.

Für die Umsetzung im `SQLiteAdapter` stehen eine Reihe verschiedener `select()` Methoden mit unterschiedlichen Rückgabe zu Verfügung, wobei der allgemeine Rückgabetyp vom Interface *andorid.databases.Cursor* ist.

Für den Speziellen Fall können hier auch Objekte vom Typ *ItemCursor*, *ItemKeyCursor* oder auch *IdCursor* zurückgegeben werden (siehe Abschnitt 4.4).

Der SelectCountCommand

Der SQLite-Befehl *SELECT COUNTS* ist ein spezialisierter SELECT-Befehl der ein einfachen Interger-Wert zurück gibt. Die zu selektierende Spalte wird hier aber nicht über das Schlüsselwort WHERE deklariert, sonder als Argument dem SELECT COUNTS nachgestellt.

Der Select-Count-Befehl ermöglicht eine Abfrage über die Anzahl verschiedener Elemente einer Zeile für eine Bedingung.

```
SELECT COUNTS(columns)
FROM tableName
WHERE conditions;
```

Bei der Umsetzung im `SQliteAdapter` erhält man als Rückgabetyp immer ein *Cursor*-Objekt, der gemäß seiner Matrixeigenschaft auszulesen ist. Damit das Ergebnis der Abfrage immer eindeutig ist, erfolgt bei der Ausführung der `countsOf()`-Methode hier zugleich eine Prüfung der Rückgabewerte und ggf. auch der Auswurf einer Exception.

```
override final fun countsOf(cmd: SQLiteCommand): Int {

   val result: Cursor =
      this.sqLite.rawQuery(cmd.sqLiteString, null)
   if (result.count > 1)
      throw java.lang.RuntimeException("result has more than one entry")

   return if (result.moveToFirst())
         result.getInt(0)
      else 0
  }
}
```

Der ExtremaCommand

Für das Erfragen eines Extremwertes, benötigt die SQLite-Syntax zunächst das Extrema `MIN` bzw. `MAX`, den Spaltenname des Extremwertes, sowie eine Bedingung für die Abfrage.

```
SELECT MAX/MIN(column_name)
FROM table_name
WHERE condition;
```

Wird keine explizite Bedingung angegeben, so wird der Extremwert der entsprechenden Zeile abgefragt.

Der `SQLiteAdapter` unterschiedet bei Einzelabfragen lediglich zwischen `getIntOf()`, `getRealOf()` bzw. der `getStringOf()`-Methode. Die Extremwert-Anfrage selbst ist hierbei über den jeweiligen Übergabeparameter vom Typ `SQLiteCommand` definiert.

Die Methoden selbst arbeiten hier analog zu `countsOf()`, nur dass der Rückgabetyp hier auch `Null` annehmen kann. Dies ist beispielsweise der Fall, wenn eine Extremwertabfrage auf eine leere Tabelle ausgeführt wird.

```
public fun getIntOf(cmd: SQLiteCommand): Long?
{
   val result: Cursor = this.sqLite.rawQuery(cmd.sqLiteString, null)
   if (result.count != 1)
      throw java.lang.RuntimeException("result has more than one entry")

   return if(result.moveToFirst()) result.getLong(0) else null
}
```

Der DeleteCommand

Um Daten aus einer Tabelle zu löschen wird lediglich der Tabellenname und die Löschbedingung benötigt. Die SQLite-Syntax lautet hierfür:

```
DELETE FROM table_name
WHERE condition;
```

Bemerkung: 4.2 *Wird explizit keine Bedingung angeführt, so werden sämtliche Daten einer Tabelle gelöscht!*

4.2. Parser

Das Unterpaket *texts* definiert das Interface *SQLitePhrase* für das abstrakten Attribut `sqLiteString: String`. Dieses Interfaces bildet die Basis für alle Hilfsobjekte zum Erzeugen eines SQLite-Befehls.

4.2.1. SQLite- Schlüsselwörter

Die wichtigsten SQLite-*Schlüsselwörter* befinden sich im Package *enums* und implementieren neben dem Interface *SQLpitePharse* auch das *EnumInterface* für eine textbasierte Bildschirmausgabe. Somit können die Enums über ListAdapter-Klassen selektiert und identifiziert werden.

Die SQLite- fähigen Enum-Klassen gehören:

1. `DataPrimeType` (`STRING`, `INTEGER` `REAL` und `BLOB`);
2. `Function` (`CREATE`, `SELECT`, `INSERT`, `DELTE`, `UPDATE`, etc.);
3. `Operator` (`AND`, `OR`, `NOT` und `NOP`);
4. `Relation` (`EQUAL`, `LESS`, `LESS_EQ`, `GREATER`, `GREATER_EQ`, `NOT_EQ` und `LIKE`);
5. `OrderType` (`ASC` und `DESC`);

Die `DataPrimeType` definiert die vier Grundtypen *INTEGER*, *REAL*, *STRING* und *BLOB*. Höhere Datentypen können alternativ auch über das `DataType`-Enum repräsentiert sein, die Definition der elementaren Datentypen muss dann aber auf Umwegen erfolgen. Das Enum `Function` definiert Elementare SQLite-Funktionen wie `INSERT`, `DELETE` oder ähnliches und stehen meistens zu Beginn eines SQLite-Befehls. Das Enum `Operator` und `Relation` wird vorwiegend für das Erstellen von Bedingungsphrasen bei Select- oder Delete-Commandos benötigt. Mit `OrderType` existieren auch zwei Elemente für auf- und absteigende Sortieralgorithmen.

4.2.2. SQLite-Command-Parser

«abst. class» *SQLiteCommand*	
−	arrayList: ArrayList<SQLitePhrasebleTable>
+	columns: SQLiteAttributeParser.Columns?
+	condtions: SQLiteAttributeParser.Conditions?
+	order: SQLiteAttributeParser.Order?
+	sets: SQLiteAttributeParser.Sets?
+	size: Int
+	values: SQLiteAttributeParser.Values?
+	insertCommand(): SQLiteCommand
+	updateCommand(): SQLiteCommand
+	selectCommand(Boolean): SQLiteCommand
+	selectCountCommand(Boolean): SQLiteCommand
+	extremaCommand(Function): SQLiteCommand
+	deleteCommand(): SQLiteCommand
+	addResoruce(SQLitePhrasebleTable)
+	<Operator> get(Int): SQLitePhrasebleTable

Abbildung 4.7.: Klassendiagramm des Interfaces *CommandBuilder.Parser*

Die abstrakte Klasse `SQLiteCommand` implementiert zunächst das Interface *SQLiteAttributeParser* und erweitert dieses um weitere Attribute für Sub-Interfaces vom *SQLiteAttributeParser*. Außerdem können hier die Tabellenobjekte für die Ausführung des Befehls registriert werden.

Bei Aufruf der Methode `clearBuilder()` (aus dem Interface *ClerableBuilder*) werden alle lokalen Attribute entfernt bzw. auf `NULL` gesetzt. Die Methode `isBuilderEmpty()` gibt den Wert `true` zurück, solange seine ArrayList leer ist.

Das Attribute `sqLiteString` wird erst in den anonymen Unterklassen definiert und können über statische Methoden der `SQLiteCommand` erzeugt werden. Hierbei geben die Strings den SQLite-Befehl jeweils als Ganzes aus.

Bemerkung: 4.3 *Welche Unterobjekte für ein Befehl tatsächlich benötigt werden, ist letztlich vom konkrete Befehl abhängig. So wird beispielsweise das Order-Objekt lediglich als optionales Argument bei einem Select-Befehl eingesetzt und ist für die meisten anderen Befehle redundant.*

4.2.3. SQLiteAttributeParser

Der *SQLiteAttributeParser* ist eine Komposition aus den Interfaces *SQLitePhrase* und *ClearableBuilder* und deklariert darüber hinaus keine weiteren Attribute bzw. Methoden. Es beinhaltet jedoch weitere Sub-Interfaces für das Parsern von SQLite-Argumenten wie Bedingungssätze oder den Sortieralgorithmus. Die konkreten Sub-Parser werden in der Regel über Tabellen Objekte erzeugt die vom Interface *SQLitePhrasebleTable* ableiten und werden als Attribut dem `SQLiteCommand` übergeben.

Columns

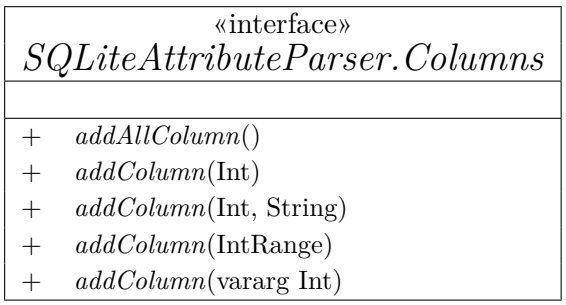


Abbildung 4.8.: Das Interface *SQLiteAttributeParser.Columns*

Das Interface *SQLiteAttributeParser.Columns* erzeugt einen String mit den Spaltennamen für einen Befehl. Üblicherweise kommt diese Instanz bei einem `insert`- oder `select`-Befehl zum Einsatz.

Conditions

Das Interface *SQLiteAttributeParser.Conditions* erzeugt eine Bedingungssatz. Das Builder-Objekt kann hierbei auf die eigene Resorucen der jeweiligen Tabelle zugreifen, oder aber auch auf andere Tabellendeklarationen. Mit den Befehl `addSubCondtion()` können einzelne Ausdrücke zudem geklammert werden.

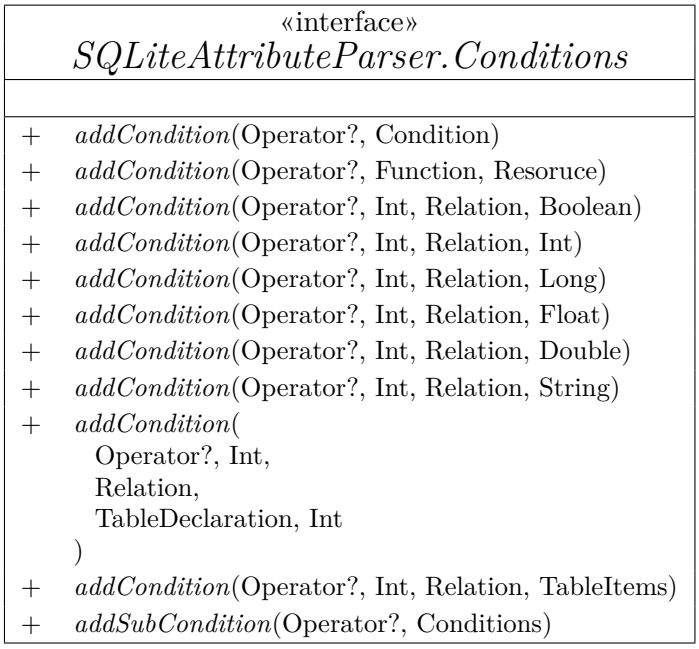


Abbildung 4.9.: Das Interface *SQLiteAttributeParser.Condtions*

Das Binden einer Bedingung erfolgt jeweils nach dem gleichen Schema. Über den Operator (`AND`, `OR`, `NOT`) wird der neue Ausdruck zunächst an seinen Vorgänger gebunden. Handelt es sich hierbei um den ersten Ausdruck im Builder, so wird der Operator ignoriert. Mit dem zweiten Argument erfolgt die Spaltenselektion über den dazugehörigen Index. Im dritten und vierten Argument wird schließlich die Relation und der Wert des Datentyps definiert.

Order

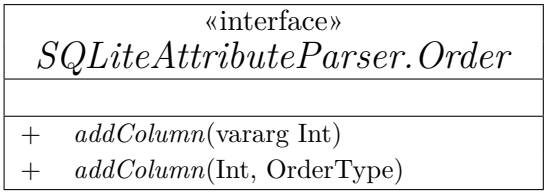


Abbildung 4.10.: Das Interfaces *SQLiteAttributeParser.Order*

Das Interface *SQLiteAttributeParser.Order* definiert die Reihenfolge in der Tabelleneinträge für die Ausgabe sortiert werden sollen. Die Methoden sind hierbei analog zu `Columns` deklariert, nur dass hier das Attribut `OrderType` mit angegeben wird.

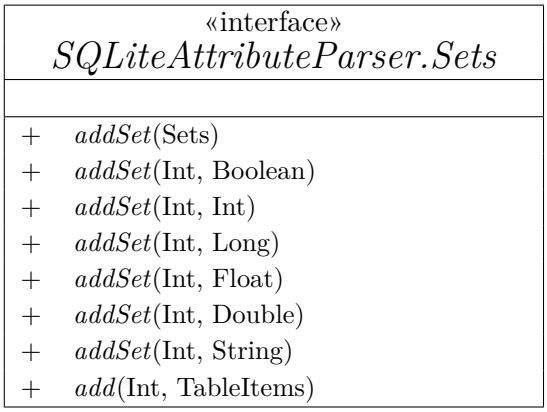


Abbildung 4.11.: Das Interfaces *SQLiteAttributeParser.Sets*

Sets

Das Interface *SQLiteAttributeParserr.Sets* hingegen wird ausschließlich bei der Definition eines SQLite-Update-Befehls benötigt und geniert einen String der Form `colName = colData`.

Values

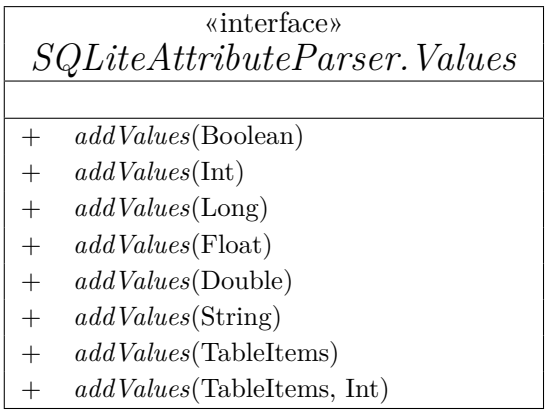


Abbildung 4.12.: Das Interfaces *SQLiteAttributeParser.Values*

Das Interface *SQLiteAttributeParser.Values* ist analog zu *Sets*, jedoch ohne explizite Angabe des Spaltennamens.

4.2.4. Tabellenobjekte

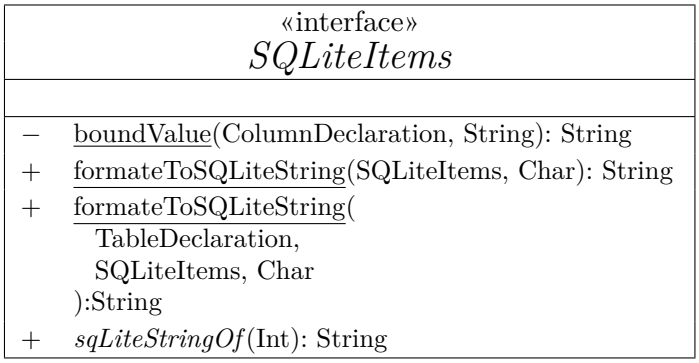


Abbildung 4.13.: Das Interfaces *SQLiteItems*

Das Interface *SQLiteItems* leiten von *ColumnDimension* ab und erweitert dieses um `sqLiteStringOf()`. Hiermit können die Dateninhalte von Tabellendaten in ihren jeweils gültigen SQLite-String formatiert werden.

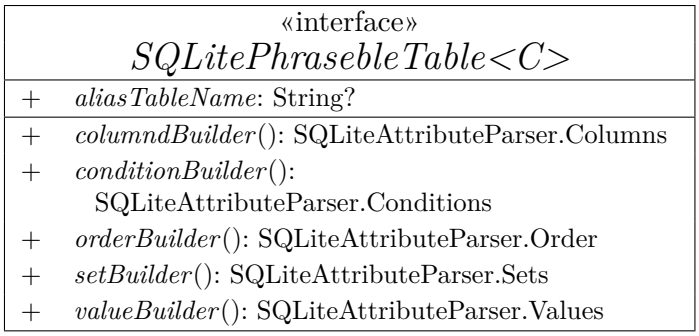


Abbildung 4.14.: Das Interfaces *SQLitePhrasebleTable*

Das Interface *SQLitePhrasebleTable* leitet zunächst von *TableDeclaration* ab und erweiter dieses um Ausgabemethoden für die einzelnen *SQLiteAttributeParser*. Zudem deklariert das Interface das optionale Attribute `aliasTableName`.

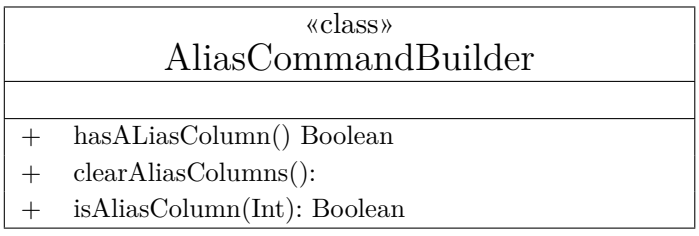


Abbildung 4.15.: Die Klasse `AliasCommandBuilder`

Die Klasse `AliasCommandBuilder` leitet von der Instanz `TableHeader` ab und imple-

mentiert das Interface *SQLitePhrasebleTable*.

Alias Spaltennamen werden innerhalb dieses Objekt zunächst über den Column-Builder definiert. Diese sind temporär in einer *HashMap* gespeichert und können bei Bedarf über `clearAliasColumns()` auch wieder entfernt werden.

4.3. Datenbank-Objekte

Der `SQLiteAdapter` bildet die Schnittstelle zwischen der physikalischen und der virtuellen Datenbank und weist im wesentliche prozedurale Eigenschaften auf. Um auch dem objektorientiert Ansatz gerecht zu werden enthält *framework.databases* Hilfsobjekte um geschlossene Datenbankobjekte zu realisieren.

4.3.1. Datenbank-Manager

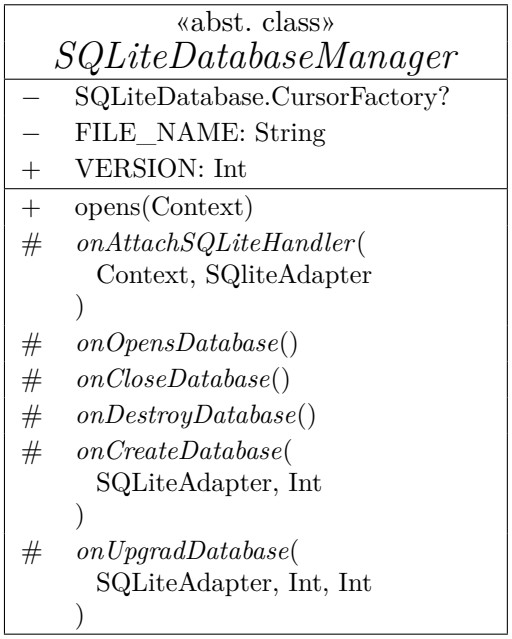


Abbildung 4.16.: Die abstrakte Klasse `SQLiteDatabaseManager`

Die abstrakte Klasse `SQLiteDatabaseManager` bildet innerhalb das Frameworks das Wurzelelement für Datenbanken. Hiervon abgeleitete Klassen sind in der Regel als `object` definiert, sodass alle in ihr enthaltenen Elemente global zugänglich sind und nicht in jeder `Activity` neu erzeugt bzw. geladen werden müssen.

Der Datenbank Manager wird über die Methode `opens()` initialisiert und über `close()` (aus dem Interface *Closable*) wieder sicher geschlossen. Hierbei durchläuft der Manager einen eigenen Lebenszyklus um Tabellenobjekte und ihre Daten sicher zu laden bzw. wieder zu schließen.

4.3.2. Datenbank Fragmente

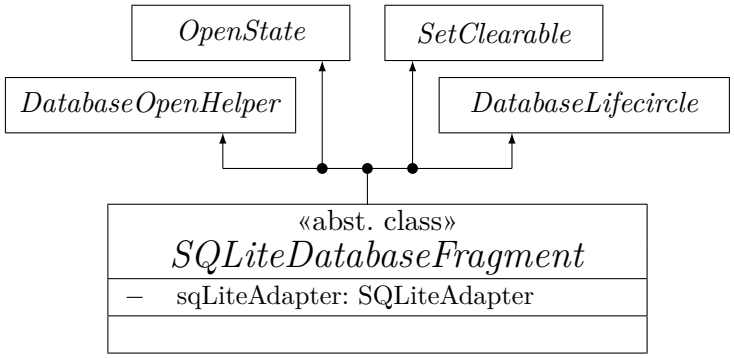


Abbildung 4.17.: Die abstrakte Klasse `SQLiteDatabaseFragment`

Mache Datenbanken können für eine bessere Wartbarkeit in Fragmente zerlegt werden. Hierfür stellt die *framework.databases* die abstrakte Instanz `SQLiteDatabaseFragment` bereit. Ein `SQLiteDatabaseFragment` wird in der Regel von einem `DatabaseManager` erzeugt und initialisiert.

Die abstrakte Klasse implementiert die Interfaces *DatabasOpenHelper*, *DatabaseLifecircle* (siehe unten), *OpenState* und *SetClerable*. Darüber hinaus enthält es einen geöffneten `SQLiteAdapter` als Attribut.

Die beiden Methoden des Interfaces *DatabaseLifecircle* (`opensDatabase()` und `closeDatabase()`) definieren innerhalb der Instanz das lokale Attribute `isOpen` aus *openState*. Zu Beginn des Methodenaufrufs wird jeweils geprüft, ob auch der `SQLiteAdapter` geöffnet ist. Andernfalls wird eine `RuntimeException` ausgelöst. Ebenso sind beide Methoden mit der Annotation `@CallSuper` deklariert, sodass in den abgeleiteten Klassen auch immer der Methodenaufruf der Superklasse erfolgen muss.

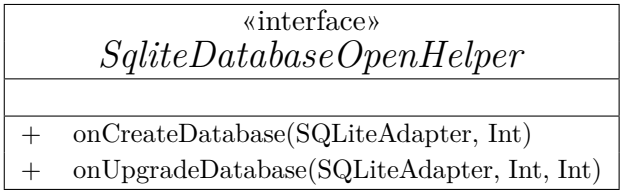


Abbildung 4.18.: Das Interface *SqliteDatabaseOpenHelper*

Das Interface *SQLiteDatabaseOpenHelper* ist analog zur abstrakten Instanz *android.databases.sqLite.*`SQLiteOpenHelper` und deklariert die Methoden `onCreateDatabase()` bzw. `onUpgradeDatabase()`. Die Methoden unterscheiden sich jedoch durch den Übergabeparameter `SQLiteAdapter` anstelle der `SQLiteDatabase`.

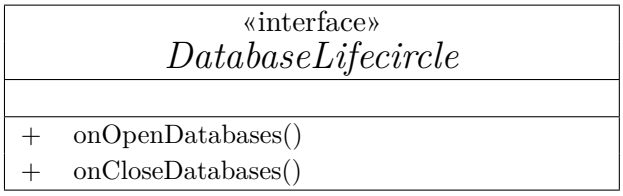


Abbildung 4.19.: Das Interface *DatabaseLifecircle*

Das Interface *DatabaseLifecircle* bildet einen rudimentären Lebenszyklus für die Datenbank nach und deklariert die beiden Methoden `onOpensDatabase()` bzw. `onCloseDatabase()`.

4.4. Cursor

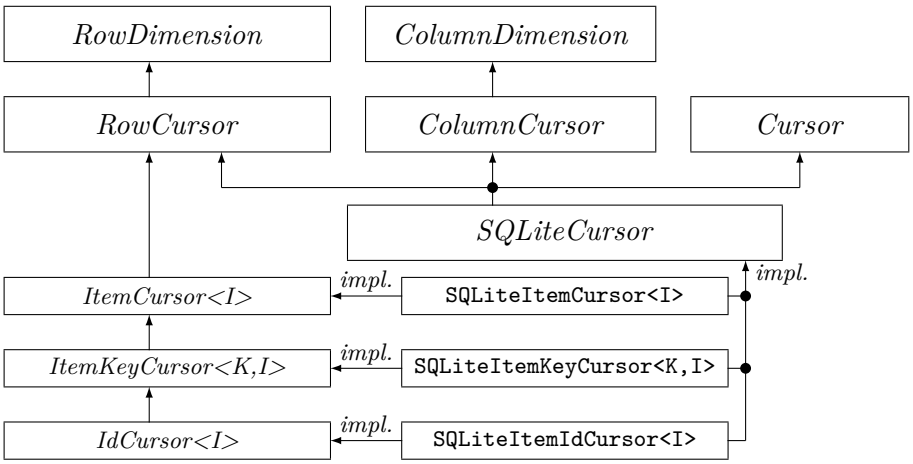


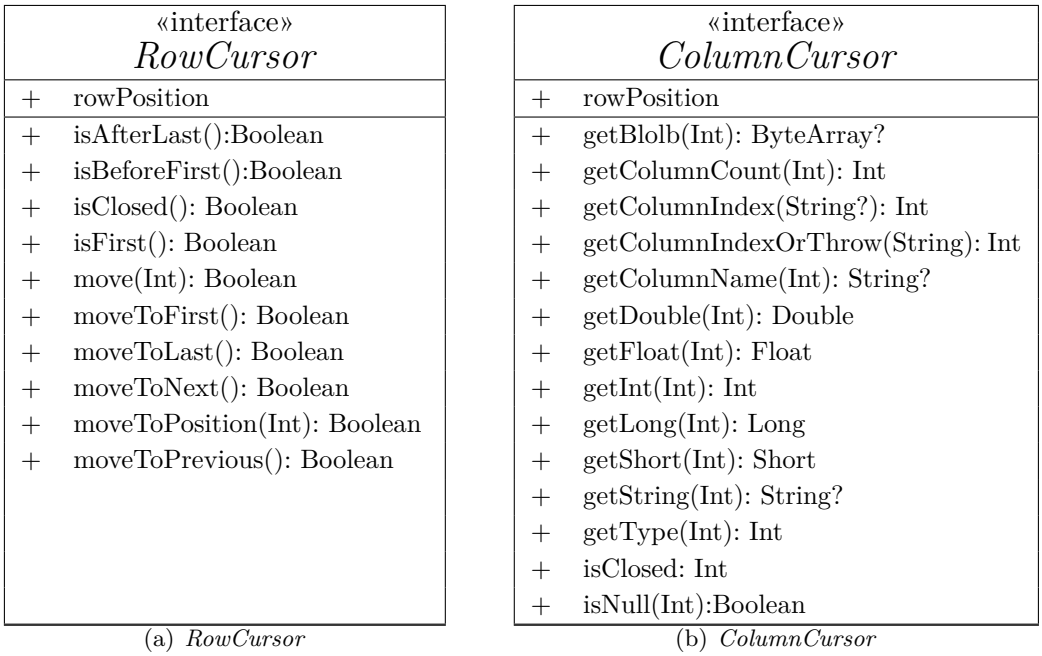
Abbildung 4.20.: *SQLiteCursor*-Objekte

Die android-Standard-Bibliothek kennt für das Lesen der Daten aus einem Datenbank-Objekte den Rückgabewert *android.database.Cursor*. Dieser enthält Methoden mit denen Daten entlang ihrer Zeilen- und Spalteneigenschaften ausgelesen werden.

Für das Unterpaket *cursors* wurde der *Cursor* zunächst in die beiden Teilinterfaces *Row-* und *ColumnCursor* aufgeteilt. Der *RowCursor* ermöglicht zunächst das navigieren entlang der Zeilen. Die Navigation erfolgt hierbei entweder relativ zur aktuellen `rowPosition`, oder absolut. Der Rückgabewert ist jeweils vom Typ `Boolean` und gibt an, ob sich an der neuen Position auch Daten befinden.

Der *ColumnCursor* enthält allgemeine Ausgabemethoden innerhalb einer gesetzten Zeile. Der Übergabetyp `Int` repräsentiert jeweils den Spaltenindex. Neben den Daten können hier auch Spaltenname und Datentyp ermittelt werden.

Der *SQLiteCursor* leitet von dem *Cursor*-Interface der andorid.lib sowie dem *Row-* und `ColumnCursor` ab und bildet hierdurch eine gemeinsame Schnittstelle für konkreten Klassen-Objekte.



(a) *RowCursor* (b) *ColumnCursor*

Abbildung 4.21.: Die Interfaces *Row-* und *ColumnCursor*

Ein erstes Objekt welches den *RowCursor* implementiert und definiert ist die abstrakte Instanz `BaseRowCursor`. Von ihr leiteten meist anonyme Klassen ab, die über Ausgabemethoden wie `emtpyCursor()` in den jeweiligen Interfaces definiert sind. Die konkreten Cursor-Objekte leiten zunächst von der Klasse *android.database.sqlite.*`SQLiteCursor` ab und implementieren zudem den *framework.databases.SQLiteCursor*. Über spezielle Factory-Objekte können diese im Rahmen eines Selekt-Befehls über den `SQLiteDatabase` erzeugt werden.

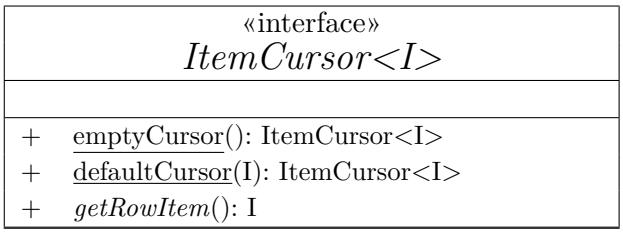


Abbildung 4.22.: Das Interfaces *ItemCursor*

Vom *RowCursor* leitet zunächst das Interface *ItemCursor* ab, welches die abstrakte Methode `getItem()` für einen beliebigen Rückgabetypen deklariert.

Der *ItemKeyCursor* erweitert den *ItemCursor* um das generische Attribut `rowKey: K` (*Primärschlüssel*). Der Zeilenschlüssen ist für eine Tabelle ein optionales Spaltenelement, welches in der Spaltendeklaration mit dem SQLite-Modifyer `PRIME KEY` versehen ist. In

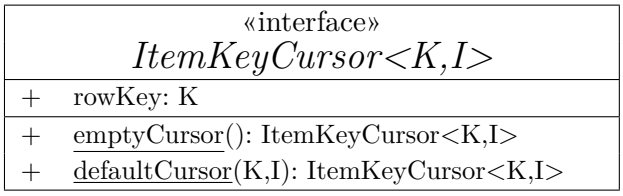


Abbildung 4.23.: Das Interfaces *ItemKeyCursor*

den meisten Anwendungen wird für den Primärschlüssel die Zeilen-Id von Typ `Long` verwendet, welcher durch den *IdCursor* direkt unterstützt wird.

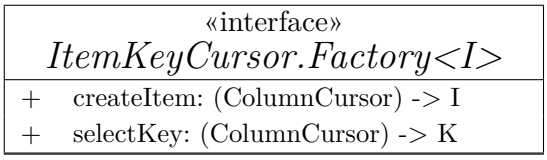


Abbildung 4.24.: Das Interfaces *ItemKeyCursor.Factory<I>*

Für das Erzeugen eines neuen Items implementieren die einzelnen Cursor-Objekte zusätzliche Factory-Objekte mit abstrakten λ-Objekten als Attribut. Mit diesen können dann die Zeilenobjekte bzw. der Primärschlüssel über einen *ColumnCursor* erzeugt werden.

4.5. SQLite-Tabellen

Das *framework.databases*-Paket kennt die beiden Unterpaket *layouts* und *tables*. Während *laoyuts* in erster Linie Interfaces für Tabellen-Items und den dazugehörigen Tabellenobjekten deklariert, definiert das Paket `tables` konkrete Daten- und Tabellenobjekte.

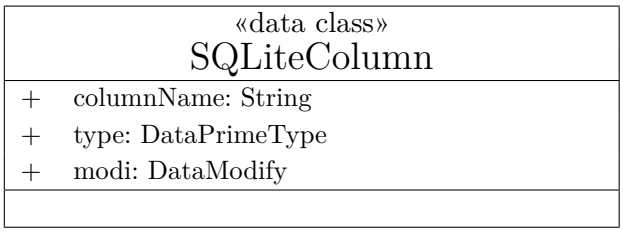


Abbildung 4.25.: Die Datenklasse `SQLiteColumn`

Die Datenklasse `SQLiteColumn` implementiert das Interface *ColumnDeclaration* wobei der Datentyp für vom Enum `DataPrimeType` definiert ist. Zusätzlich implementiert die Klasse ein Attribute vom Enum `DataModify` (defaut: `non`) um die Spalte näher zu spezifizieren.

Das Basisinterface für funktionale Tabellen ist die *TableSupport* für generische Typen C (*ColumnDeclaration*), D (Datencontainer z.B. Cursor) und I (*SQLiteItems*). Das

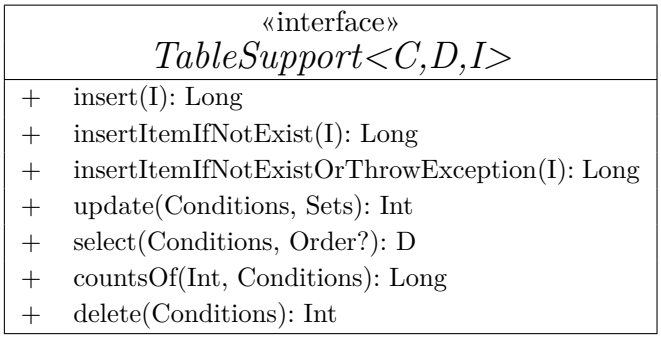


Abbildung 4.26.: Das Interfaces *TableSupport<D>*

Interface leitet zunächst von *SQLitePhrasebleTable* aus dem Unterpaket *framework.databases.parsers* (Abschnitt **??**) ab und ermöglicht über deren Ausgabemethoden das Parsen von Teilargumenten für *Condition*- oder *Order*-Sätze.

Als Rückgabetyp für den Datenbereich deklariert Das Interface die beiden generischen Typen `I` und `D`. Die Zeilenelemente `I` müssen hierbei vom Interface *SQLiteItems* ableiten, während die Datencontainer beliebig gewählt werden können, in der Regel aber von einem *ItemCursor* ableiten (siehe *framework.databases.cursors*).

Darüber hinaus implementiert das Interface noch die Instanzen *framework.tables.TableSelector* sowie *MatrixDimension* und *SetClearable* aus dem Package *framework.maths.sets*. Somit kann auch die Zeilenweite direkt gelesen und auch der Tabelleninhalt ohne Umwege gelöscht werden.

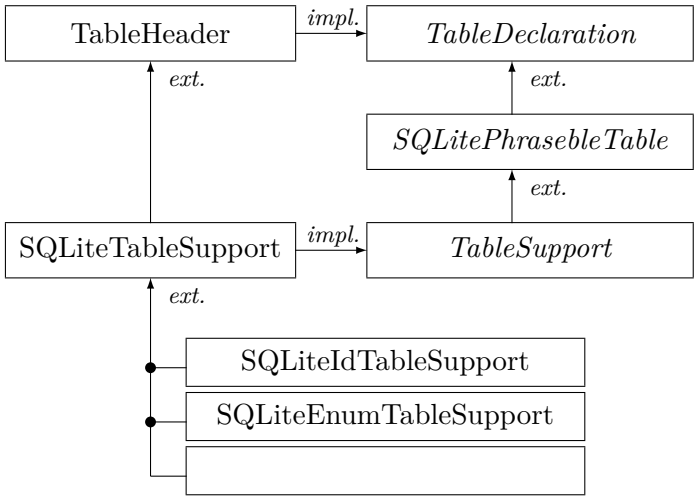


Abbildung 4.27.: Übersicht zu SQLite-Tabellen

Das Basselement für SQlite-Kompatible Tabellen ist die abstrakte Instanz `SQLiteTableSupporter` aus dem Unterpaket *framework.database.table*.

Für die Spaltendeklaration `C` wird für gewöhnlich ein Objekt vom Typ `SQLiteColumn`

genutzt, wobei im optionalen Attribut der Primärschlüssel als `modify` definiert werden kann.

4.5.1. Standardtabellen

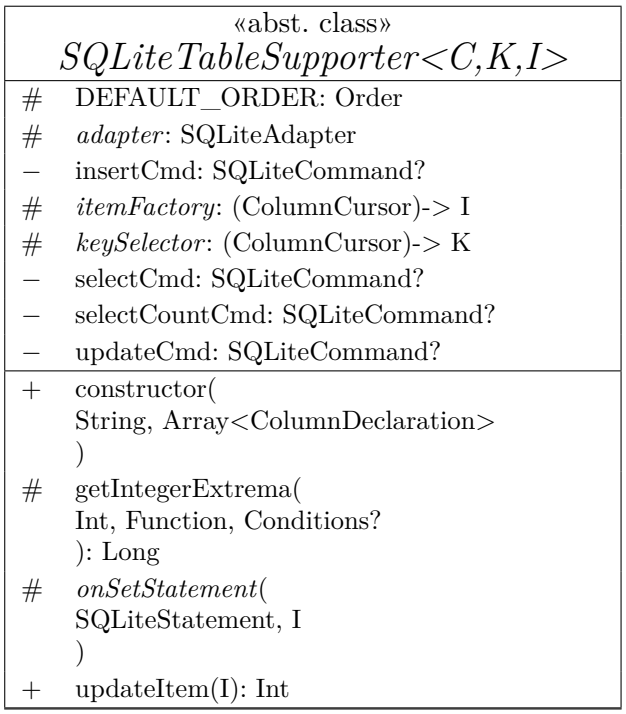


Abbildung 4.28.: Die abstrakte Klasse *SQLiteTableSupporter*

Die abstrakte Klasse `SQLiteTableSupporter` leitet von `TableHeader` ab und implementiert das Interface *TableSupport* für generische Typen C (*ColumnDeclartion*) einem Key (K) für beliebige Typen und I (*TableItems*). Hiermit ist eine erste Basisklasse für eine SQLite-Kompatible Tabellenobjekt geschaffen, wobei für D zunächst ein Objekt vom Typ *ItemKeyCursor* gesetzt ist.

Für die SQLite-Anbindung implementiert die Instanz ein abstraktes Attribut SQLiteAdapter, sowie Factory-Objekte für das Auslesen des Primärschlüssels und den Zeilenitems. Während die Update- und Delete-Methoden bereits vollständig implementiert sind, sind die Insert- und Select-Befehle weiterhin abstrakt.

Damit auch Blob-Elemente fehlerfrei eingelesen werden können existiert für den `insert`- und `update`-Befehle zusätzlich die abstrakte Methode `onSetStatement()`. Prinzipiell kann hier jeder primitive Datentyp eingelesen werden, sofern im insert-Kommando, der pirmitive Wert durch den Platzhalter »`?`« ersetzt wird.

4.5.2. Id-Tabellen

Die meisten Anwendungstabellen leiten von der abstrakten Instanz `SQLiteIdTable-Support` ab und implementieren die Spalte `_ID: Long` als Primary-Key. Die Id kann genutzt werden um Tabelleneinträge mit anderen Elemente zu verknüpfen. Der Rückgabe Typ `Long` ist hierbei der Standard Rückgabetyp für Insert-Befehle.

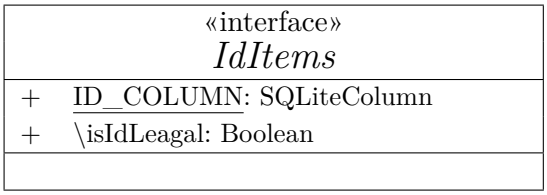


Abbildung 4.29.: Das Interface *IdItems*

Die Zeileneinträge für eine SQLite-Id-Tabelle leitet implementiert mindestens das Interface *IdItems*, welches die beiden Interfaces *SQLiteItems* und *Id* vereint.

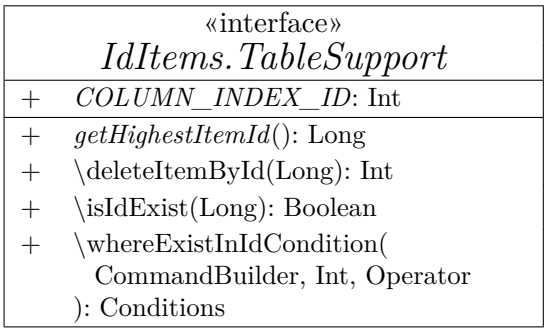


Abbildung 4.30.: Das Interface *IdItems.TableSupport*

Das Interface *IdItems.TableSupport* erweitert schließlich den *SQLiteTableSupport* und wird unter anderem von der `SQLiteTableSupport` implementiert.

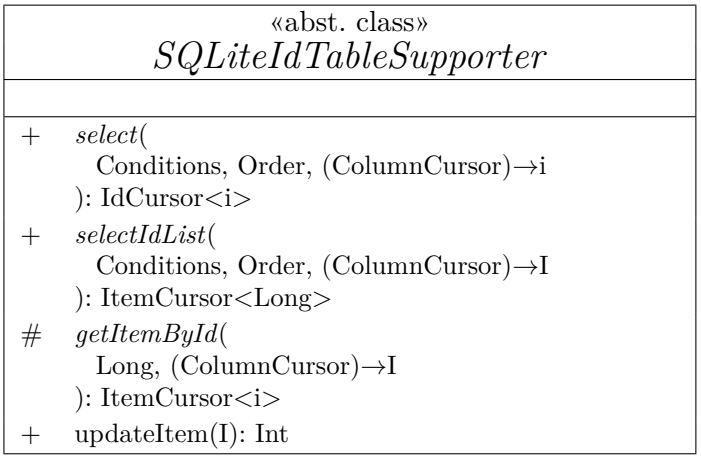


Abbildung 4.31.: Die abstrakte Klasse `SQLiteIdTableSupporter`

Von dem `SQLiteIdTableSupport` leiten derzeit die konkreten Objekte `Database-Info.Sup-port`, `Protcol.Support` sowie der `PrimaryItem.Support` ab, die jeweils als Unterklasse der dazugehörigen Item-Klassen definiert sind.

4.5.3. Enum-Tabellen

Enum-Klassen implementieren sowohl einen Bezeichner (`name`) als auch eine Ordnungszahl (`order`) und können daher auf zwei Arten als Primary Key in einer Datenbank hinterlegt werden. Um die Abfrage-Rountine möglichst einfach zu halten, wird als Primary-Key für die physikalische Tabelle die Ordnungszahl bevorzugt. Da Enums im Allgemeinen statische Objekte repräsentieren, sollten die Tabelle bei ihrer Erstellung für alle Enum-Attribute vorinitialisiert werden, sodass hier lediglich Update-Befehl bei Änderungen zum Einsatz kommen.

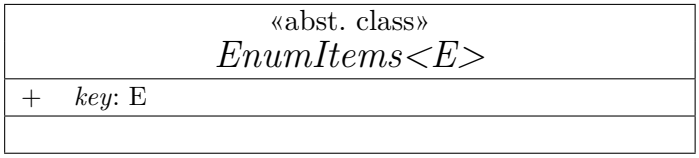


Abbildung 4.32.: Das Interface `EnumItems`

Das Interface *EnumItems* leitet vom Interface *SQLiteItems* ab und erweitert dieses um das Attribute `key: E` für beliebige generische Objekte vom Typ `E: Enum<E>`.

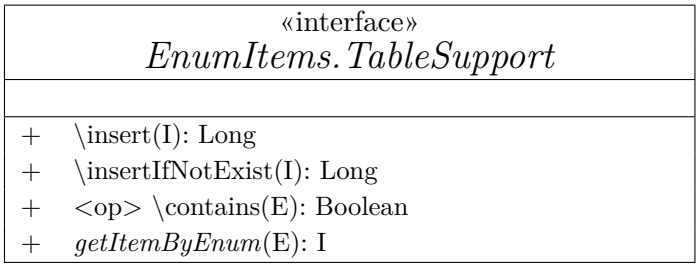


Abbildung 4.33.: Das Interface `EnumItems.TableSupport`

Das Interface *EnumItems.TableSupport* leitet von *TableSupport* ab und erweitert dieses um den abstrakten Operator `contains()`.

Ebenso erhalten die Methoden `insert(I)` und `insertIfNotExist()` eine erste Default-Definition, in dem sie standardmäßig die Methode `insertIfNotExist()` aufrufen. Hierdurch ist sichergestellt, dass immer nur Zeilen eingefügt werden können, die noch nicht existieren.

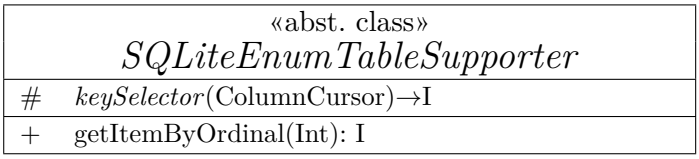


Abbildung 4.34.: Die abstrakte Klasse `SQLiteEnumTableSupporter`

Die Instanz `SQLiteEnumTableSupporter` definiert einer erstes abstraktes Tabellenobjekt und erweitert den `SQLiteTableSupporter` um die abstrakte λ-Funktion `keySelector: ()→ E` sowie der Methode `getItemByOrdinal()`. Als Konvention wird davon ausgegangen, dass sich Enum-Objekt in der `SQLiteEnumTableSupporter` ersten Spalte befinden und über ihre Ordnungszahl identifiziert werden können.

Eine konkrete Enum-Tabelle ist mit `Permission.Support` realisiert. Hierbei handelt es sich um eine einfache Tabellenanwendung, für das Verwalten von Berechtigungen, die nicht über die Android-API gemanagt werden. Hierzu gehören z.B. die Einwilligung zur Datenschutzerklärung oder auch der Allgemeinen Nutzungsbedingungen einer App. Neben dem Primary-Key enthält die Tabelle eine weitere Spalten für die aktuelle Version der Berechtigung, sowie ein Status-Enum mit den Attributen `accepted`, `notAccepted` und `neverAskAgain`.

4.5.4. SQlite-Master

Jede SQLite-Datenbank enthält eine *Master-Tabelle* in der sämtliche Tabellen einer Datenbank registriert sind. Im der `SQLiteDatabase` ist diese unter dem Namen `sqlite_master` zu finden und enthält insgesamt 5 Spalten.

Im Unterpaket *framwork.databases.tables* sind die Daten im Datenobjekt `TableInfo` und das dazugehörige Tabellenobjekt innere Klasse vom Typ `TableInfo.SQLiteMaster` realisiert.

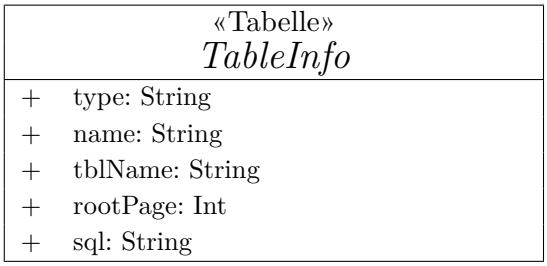


Abbildung 4.35.: Die Tabelle `TableInfo`

Die `TableInfo.SQLiteMaster` leitet direkt von dem `SQLiteTableSupporter` ab, und ist so ausgelegt, dass auf ihr nur lesend zugegriffen werden kann, während sämtliche Schreibbefehle (`INSERT`, `UPDATE`, `DELTE`) Exeptions auswerfen.

4.5.5. Datenbank-Tabelle

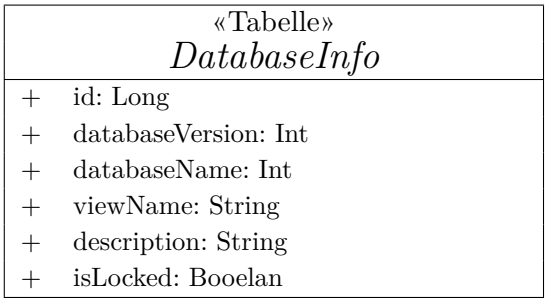


Abbildung 4.36.: Die Tabelle `DatabaseInfo`

Allgemeine Datenbankobjekte können wie Abschnitt (4.1.2) das `Context`-Objekt erzeugt, geöffnet und wieder geschlossen bzw. entfernt werden. Sollen Datenbanken darüber hinaus in einem geschlossenem System verwaltet werden, so bietet das Framework die Klasse `DatabaseInfo` mit dem dazugehörigen Tabellenobjekt `DatabaseInfo.Support` an.

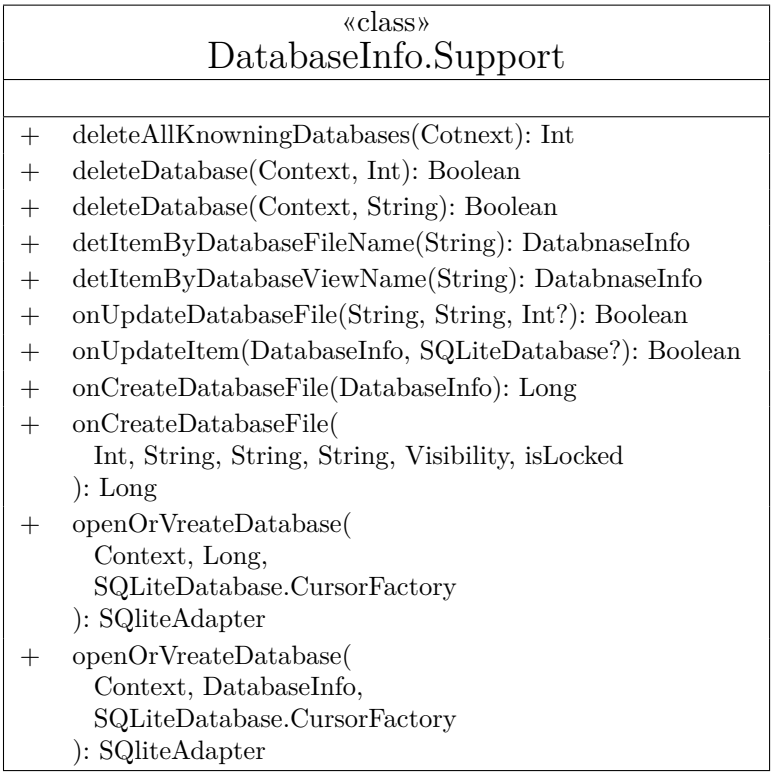


Abbildung 4.37.: Die Klasse `DatabaseInfo.Support`

Die `DatabaseInfo.Support` leitet von der abstrakten Instanz `SQLiteIdSupport` ab und implementiert zudem das Interface *Closable*, deren Methoden an den `SQLiteAdapter` delegiert sind. Datenbankobjekte die hier registriert erhalten auch immer eine Id und können auch anhand dieser geladen bzw. geöffnet werden.

4.5.6. Protokolle

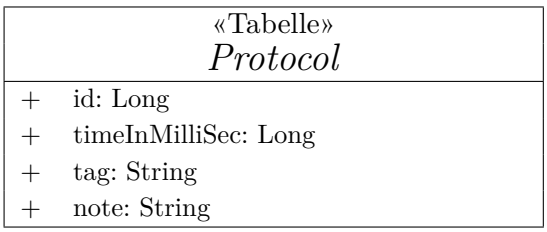


Abbildung 4.38.: Die Tabelle `Protocol`

Die Datenklasse `Protocol` ermöglicht das Führung eines Datenbankbasierten Protokolls innerhalb einer Datenbankanwendung und implementiert die Interfaces *IdItem* sowie *Timestamp*. Analog zu Klasse *android.util*.`Log` bestehen Tabelleneinträge hier aus

einem Tag und der Notiz.

Das SQLite-Tabellenobjekt ist wie bei anderen Klassen auch hier als innere Instanz über den `Protocol.Support` realisiert.

4.5.7. Typen Umwandeln

Das Unterpaket *tableLayouts* enthält einige vorgefertigte Interfaces, für häufig vorkommende Tabelleneigenschaften. Die Vorlagen erweitern zunächst den Datentype *TableItems* und beinhalten zusätzliche innere Interfaces für eine Erweiterung des *TableSupporter*. Ebenso sind hier Hilfsobjekte (`...Converter`) definiert, mit denen abgeleitete Datentypen in ihre Roh-Form transformiert werden können.

blob

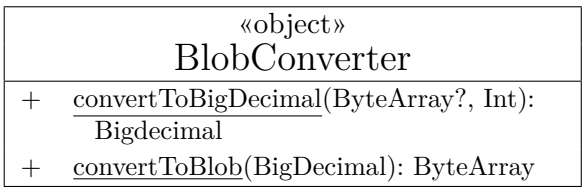


Abbildung 4.39.: Das Objekt `BlobConverter`

Mit Hilfe des BlobConverter können Byte-Ressourcen in ihre Datenobjekte umgewandelt werden.

integer

Integer-Spalten können eine Vielzahl verschiedener Informationen repräsentieren.

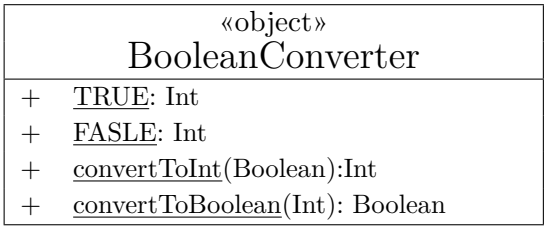


Abbildung 4.40.: Das Objekt: `BooleanConverter`

Da die SQLite standardmäßig keine boolschen Werte unterstützt, müssen diese mit Hilfe eines Integer ersetzt werden. Für die Umwandlung kommt hierfür der BooleanConverter zum Einsatz.

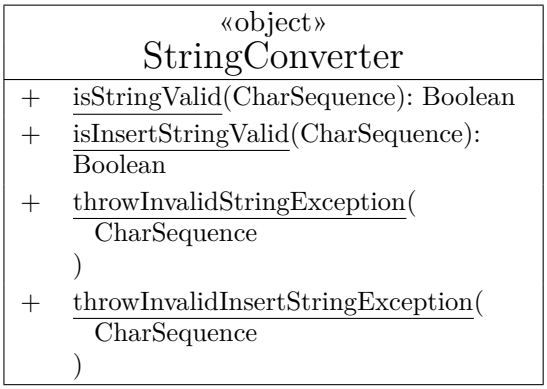


Abbildung 4.41.: Das Objekt: `StringConverter`

string

Um String-Resourcen als solche kenntlich zu machen, werden diese innerhalb der SQLite-Datenbank Anwendung in Anführungszeichen gesetzt. Hierdurch wird der String vom eigentlichen Quellcode unterscheidbar, sodass die SQLite-Abfragen geschützt sind. Allerdings dürfen die verwendeten Zeichen ihrerseits nicht mehr Teil der Eingabewerte sein. Um die hieraus resultierende Fehler früh zu erkennen bietet der StringConverter zwei Methoden um String-Attribute zu testen. Die Methode `isStringValid()` wird aufgerufen, bevor die Anführungszeichen gesetzt werden, während `isInsertStringValid()`, den String auf die Gültigkeit nach dem Setzen der Anführungszeichen prüft. Der String enthält dann genau zwei Anführungszeichen zu zwar Beginn und am Ende.

Mit den beiden Methoden `throwInvalidStringException()` bzw. `throwInvalidInsertStringException()` wird bei einer Verneinung der Abfrage direkt eine Ausnahme ausgelöst.

4.5.8. Tabellen Grundtypen

Im Rahmen dieses Packages können Tabellentypen unterschieden werden:

1. PrimaryTables (Id und String-Eintrag);
2. IntergerTables (Id, Integer-Feld);
3. LinkedIdTabels (Id, Integer-Feld mit linkedIds);
4. PrimaryIdTabels (Id, String-Eintrag und Integer-Feld mit linkedIds);

Jede Tabelle ist so aufgebaut, dass immer eine Id zur eindeutigen Identifizierung einer einzelnen Zeile möglich ist. Für die Umsetzung implementiert jeder Tabellentyp ein entsprechendes Interface für den Tabellenkopf (*..Support*) und deren Tabelleneinträge (*..Items*).

Bei der *PrimaryTable* handelt es sich um eine einfache Tabelle, die neben der Id noch ein weiteres Feld für einen String-Eintrag (entry) enthält. Hier wird das Interface *RootSupport* für den TabellenHeader implementiert, bzw. *RootItems* für die Tabelleneinträge. Die Tabelle kann genutzt werden, um einzelne Wörter zu einem bestimmten Themengebiet zusammenzufassen (z.B. alle vorkommenden Namen in einer Datenbank).

Definition: 4.1 *Eine* PrimaryTable *ist eine zweispaltige Tabelle mit einer eindeutigen Id und einem String-Eintrag (entry).*

Die Integer-Tabelle implementieren das Interface *IntegerSelectSupport* bzw. IntergerItems. Dieses Tabellenformat dient hierbei als Basis für abgeleitete Integer-Tabellen.

Definition: 4.2 *Eine* IntegerTable *ist eine Tabelle mit n Spalten, bestehend aus einer eindeutigen Id, sowie maximal* $n-1$ *integer Felder als Daten-Objekt.*

Mit Hilfe einer *IdTable* können verschiedene id-Adressen aus anderen Tabellen mit einander verknüpft werden. Hierdurch entsteht ein Kontext, für deren Auswertung die Inhalte der verlinkten Tabellen benötigt wird.

Definition: 4.3 *Eine* LinkedIdTable *ist eine Tabelle mit n Spalten, bestehend aus einer eindeutigen Id, mit maximal* $n-1$ *verlinkte Ids zu anderen Tabelleninhalten.*

Die Linked-Id-Table ist direkt von der IntegerTabel abgeleitet. Dies gilt sowohl für die abstrakte Supporter-Klasse, das Item-Objekt sowie auch deren implementierten Interfaces.

Bei einigen Anwendungen hat sich gezeigt, dass ein einzelner String mit genau einem Eintrag in einer LinkedId-Tabelle verknüpft ist (1 zu 1 Verknüpfung). Damit String und Id-Tabellen nicht aufwendig in separaten Tabellen geführt werden müssen, existiert mit der *PrimaryIdTable* ein Tabellenformat, welche sowohl die Eigenschaften einer RootTable, als auch einer LinkedId-Table vereint.

Definition: 4.4 *Eine* PrimaryIdTable *ist eine Tabelle aus n Spalten mit einer eindeutigen Id, mindestens einem String-Eintrag und maximal* $n-2$ *linkedId-Spalten, die auf ein Element in derselben oder einer anderen Tabelle zeigt.*

5. Views und Dialogs

1

[1] zuletzt bearbeitet: 02.01.2022

5.1. Views

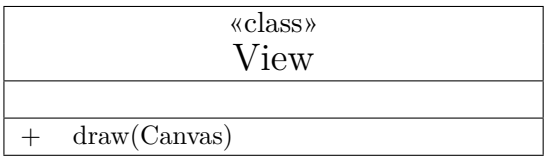


Abbildung 5.1.: Die Klasse `View`

Das Basis für eine Android Display-Element ist die Klasse *andorid.view*.`View`.

5.1.1. Text-Views

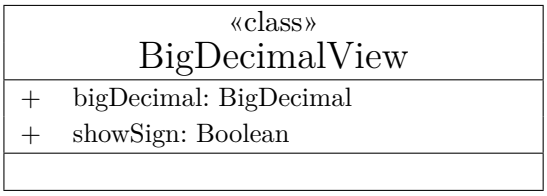


Abbildung 5.2.: Die Klasse `BigDecimalView`

Die Klasse `BigDecimalView` leitet direkt von der `TextView` ab und bildet einen allgemeinen Datencontainer für eine `BigDecimal`-Instanz. Die Klasse wird häufig in Zusammenhang mit Währungen eingesetzt. Über das Flag `showSign` kann zudem angegeben werden, ob das Vorzeichen bei der Textformatierung mit ausgegeben werden soll.

5.1.2. Hintergrund Grafiken

In der Regel sind Hintergrundgrafiken als *Drawable* in *res*-Order *drawable* bzw. *mipmap* hinterlegt und können über die `ResId` direkt in einer View oder einem Layout als Hintergrund definiert werden. Für gewöhnlich geschieht die Definition bereits im den XML-Layoutdateien. Bei Dialoge müssen der Hintergrund ggf. manuell definiert werden. Das Interface *BackgroundResourceId* deklariert das abstrakte Attribute `backgroundRe-`

Abbildung 5.3.: Das Interface *BackgroundResourceId*

`sourceId: Int` und ermöglicht somit das statische Referenzieren einer Drawable-ResId für den Hintergrund. Um Hintergründe auch manuell setzen zu können deklariert die *BackgroundDrawable* darüber hinaus passende setter-Methoden.

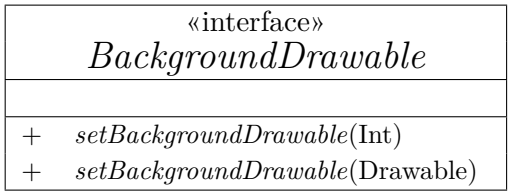


Abbildung 5.4.: Das Interface *BackgroundDrawable*

5.1.3. Grafische View-Objekte

Für grafische Anwendungen kennt das Unterpaket *framework.views* die beiden abstrakten Objektklassen `GraphicView` und `GraphicSurfaceView`. Beide Klassen besitzen hierbei den gleich Aufbau und unterschieden sich lediglich von ihrer Vater-Klasse. Während die `GraphicView` direkt von einer `View` ableitet, ist die `GraphicSurfaceView` von der `SurfaceView` abgleitet.

Die beiden Instanzen überschieben im wesentlichen die Methode`onTouchEvent()` um und ermöglichen hierdurch einen ausdifferenzierten *Click-* *Move-* und *Scale*-Event.

Touch und Move Event

```
protected abstract fun onActionDown(
   r: Coordinate<Double>, eventTime: Long
): Boolean
```

Die Methode `onActionUp()` wird aufgerufen, sobald ein einfacher TouchEvent-Registriert wird. Als Parameter wird die Event-Koordiante $\vec{r}$ überreicht sowie die aktuelle Event-Zeit. Der Rückgabewert vom Typ Boolean gibt schließlich an, ob das Event im nächsten Zyklus über den `onActionMove()` weiter ausgeführt werden soll.

```
protected abstract fun onActionMove(
   r: Coordinate<Double>,
   dr: Coordinate<Double>,
   eventTime: Long
): Boolean
```

Die Methode `onActionMove()` wird ausgeführt, wenn ein einfacher Touch-Event als Bewegung registriert worden ist. Die Methode enthält neben der Event-Koordinate $\vec{r}$ und der Event-Zeit noch die Wegdifferenz $\Delta\vec{r}$.

Nach Beendigung einer Bewegung wird nach Möglichkeit die Methode `onActionUp()` abgerufen, welche die gleichen Übergabeparameter wie die Down-Routine enthält.

Scale-Event

Ein Scale-Event ermöglicht das hinein, bzw. hinaus zoomen in eine Grafik. Das Event wird gestartet, sobald zwei Finger das Touchpad berühren und beendet, wenn weniger als zwei Finger vom Display registriert sind.

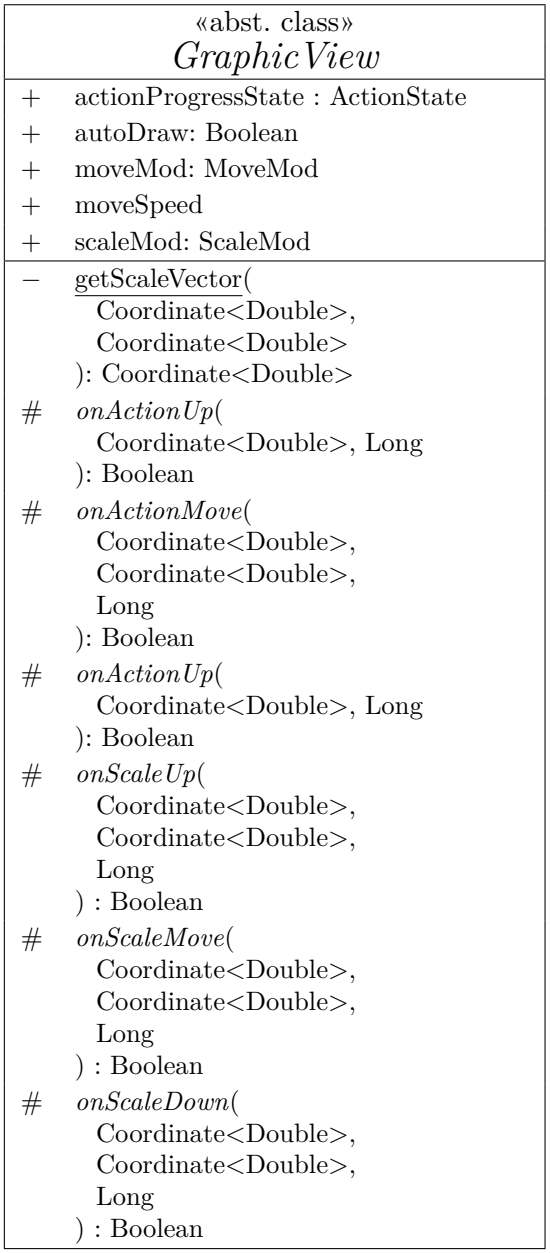


Abbildung 5.5.: Die abstrakte Klasse `GraphicView`

```
protected abstract fun onScaleDown(
   rCenter: Coordinate<Double>,
   scaleLength: Coordinate<Double>,
   eventTime: Long
): Boolean
```

Zu Beginn eines Zoom-Evens wird die Methode `onScaleDown()` aufgerufen. Als Übergabeparameter erhält man einen Vektor für den zentrierten Zoom-Ort (`rCenter` $\vec{r}_c$) sowie einen Vektor für den Wegunterschied (`scaleLength` $\vec{r}_{|\Delta|}$) zwischen den beiden Fingerkoordinaten $\vec{r}_1, \vec{r}_2$ als Betrag.

$$\texttt{rCenter} = \vec{r}_c = \vec{r}_1 + \frac{1}{2}(\vec{r}_2 - \vec{r}_1) \tag{5.1}$$

$$\texttt{scaleLength} = \vec{r}_{|\Delta|} = \{\vec{r}_2 - \vec{r}_1\}_{|\Delta|} \tag{5.2}$$

Mit Hilfe der Wegdifferenz $\vec{r}_{|\Delta|}$ kann zu Beginn eines Zoom-Vorgangs eine Differenzierung nach Koordinatenachsen vorgenommen werden. So ermöglicht z.B. die `DiagrammView` den Zoom auf der x oder der y Achse zu beginnen, sodass jeweils nur auf eine der beiden Dimensionen der Event ausgeführt wird.

```
protected abstract fun onScaleMove(
   rCenter: Coordinate<Double>,
   dScale: Coordinate<Double>,
   eventTime: Long
): Boolean
```

Folgt bei Aufruf der Methode `onScaleDown()` der Rückgabewert `TRUE` so wird die Methode `onScaleMove()` solange ausgeführt, wie auch hier der Rückgabewert `TRUE` folgt oder sich zwei Finger auf dem Display befinden. Der Übergabeparameter $\vec{r}_c$ gibt auch hier wieder den zentrierten Zoomort wieder, während $d\vec{s} : \vec{e}_x, \vec{e}_y, \vec{e}_r$ denn aktuellen Scaliervektor $(0 \ldots \infty)$ bereitstellt.

5.1.4. TableView

Für die Anzeige von Tabelleninhalten kennt die Android-API die `ListView`, mit dem widgets-Interface *ListAdapter*. Hierin können Daten in Form einer Liste für das GUI aufbereitet werden. Für Tabellen kennt die Andorid-API zwar das Layout-Objekt `TableLayout`, allerdings existiert ist keine äquivalente `TableView` die über einem *TableAdapter* gefüllt werden kann.

Die `TableView` aus dem Unterpaket *views* bietet hierfür eine Schnittstelle um Tabellen analog zum `ListView` Datenobjekt über ein *RowLayout* zu füllen.

Das Hauptelement zum Setzen der Tabellendaten erfolgt über den *TableAdapter* aus dem Unterpaket *framework.widgets* (siehe Abschnitt (5.4.4)). Darüber hinaus besteht die Möglichkeit über eine `TableRow` einen Tabellenkopf zu definieren.

«class» TableView	
−	datasetObsever: DatasetObsever
+	tableAdapter: TableAdapter
+	tableHeader: TableRow?
−	addRow(Int)
−	addRow(TableRow, Int)
+	setOnCellClickListener(onCellClickListener)
+	setOnCellLongClickListener(onCellLongClickListener)
+	setOnColumnClickListener(onColumnClickListener)
+	setOnColumnLongClickListener(onColumnLongClickListener)
+	setOnRowClickListener(onRowClickListener)
+	setOnRowLongClickListener(onRowLongClickListener)
−	setRows()
+	setTableHeader(Int, (Int, Context)→ CharSequence)
+	setTableHeader(TableDeclaration)

Abbildung 5.6.: Die Klasse `TableView`

Click-Listener

Analog zu den *OnItemClickListener* unterstützt die `TableView` eine Reihe von Click-Listener um auf Events reagieren zu können. Hierbei wird zwischen Zellen- (*Cell*), Zeilen- (*Row*) und Spalten-Events (*Column*) unterschieden, wobei auch hier eine Differenzierung zwischen einem einfachen und einem Long-CLick möglich sind. Das *Table-*

«interface» *TableView.OnCellClickListener*	
+	*onCellClick*(TableView, TableRow, View, Int, Int, Long)

Abbildung 5.7.: Das Interface *TableView.OnCellClickListener*

View.OnCellClickListener erlaubt Events die einzelne Zelle von Tabellendaten. Als Parameter wird hier die `TableView`, die Zeilen-View (`TableRow`), sowie die Spalten-View (`View`) überreicht. Hinzu kommen die Indizes für die Zeilen und Spalten (jeweils `Int`), sowie die *Id* (`Long`) des Zeilenobjekts.

Der *TableView.OnCellLongClickListener* enthält die abstrakte Methode `onCellLongClick()` für die gleichen Parameter wie der *OnCellClickListener*. Allerdings mit dem Rückgabewert `Boolean`, für eine erfolgreiche Behandlung des Events.

Das Selektieren von Zeilen erfolgt über den *TableView.OnRowClickListener* bzw. den

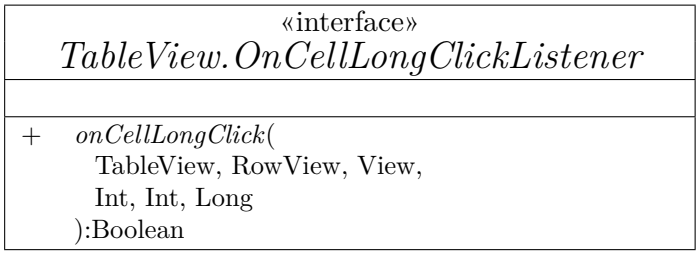


Abbildung 5.8.: Das Interface *TableView.OnCellLongClickListener*

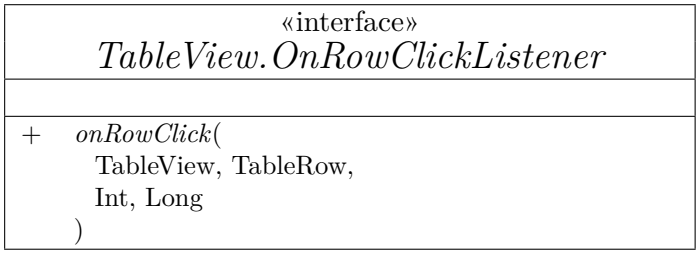


Abbildung 5.9.: Das Interface *TableView.OnRowClickListener*

TableView.OnRowLongClickListener. Als Übergabeparameter wird hier die *TableView* und die `TableRow` erwartet sowie der Zeilenindex und die *Id*. Die Listener Instanzen für einen Zeilen-Click-Event werden über die setter-Methode `setOnRowClickListener()` über eine anonyme Instanz an eine *OnCellClickListener* delegiert. Hierbei fällt dann jeweils der Spalte Index bzw. die Spalten-View heraus.

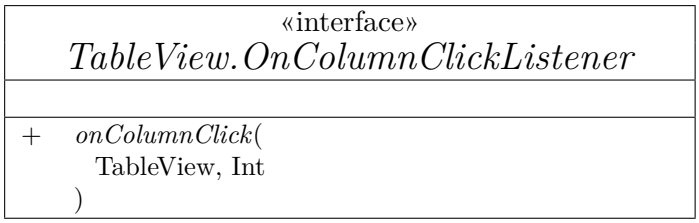


Abbildung 5.10.: Das Interface *TableView.OnColumnClickListener*

Der *TableView.OnColumnClickListener* bzw. *TableView.OnColumnLongClickListener* deklariert die Methode `onColumnClick()` bzw,. `onColumnLongClick()` für den Parameter `TableView` und den Spalten-Index (`Int`).

Während die Click-Events der *OnCellClickListener* bzw. der *OnRowClickListener* für den Datenbereich ausgeworfen werden, wird der *OnColumnClickListener* auch im Tabellenkopf (`tableHeader: TableRow`) ausgelöst. Hierdurch erlaubt die `TableView` neben der Selektion der Datenelemente auch die Selektion der Spaltendeklaration für eine nähere Beschreibung.

Setzen der Tabellendaten

Da die TableView nicht wie der `ListAdapter` von der `AdapterView` ableitet, ist die View auch nicht imstande einzelne Zeilen-Layouts zu recyclen. Das heißt, das im Rahmen dieser Anwendung alle Zeilenelemente gesetzt werden müssen, was bei großen Tabellen gewisse Nachteile mit sich bringt.

Um eine Tabelle zu aktualisieren definiert die `TableView` die Methode `setRows()`. Diese wird immer aufgerufen, sobald ein neuer *TableAdapter* definiert wird oder der `DatasetObserver` ausgelöst wird. Bei Aufruf von `setRows()` werden zunächst all Layout-Resoruce entfernt (`removeAllViewsInLayout()`). Anschließend wird die Header-Resource gesetzt und die Daten aus dem *TableAdapter* gelesen.

DatasetObsever

Um die Table-View bei Änderungen von Tabellendaten zu aktualisieren implementiert die Instanz ein Attribute vom Typ `DatasetObserver`, während der *TableAdapter* die Methoden `registerDataSetObserver()` bzw. `unregisterDataSetObserver()` aus dem Interface *DataSetObserverRegisteble* bereit stellt.

Um Laufzeitfehler zu vermeiden wird der Observer nur dann registriert, wenn die View im Fenster sichtbar wird. Hierfür definiert der lokale `DatasetObsever` das Attribute `isRegistable: Boolean`, welches beim Überschrieben der Methoden `onAttachedToWindow()` bzw. `onDetachedFromWindow()` definiert wird.

```
override fun onAttachedToWindow() {

   super.onAttachedToWindow()

   this.dataSetObserver.isRegistable = true
   this._tableAdapter?.registerDataSetObserver(this.dataSetObserver)

   this.setRows()
}
```

Die Methode `onAttachedToWindow()` wird aufgerufen, sobald die `View` im Fenster sichtbar ist. Hierbei wird zunächst das `isRegistable` Flag gesetzt und im Anschluss der Observer am Adapter registriert. Zu Letzt wird noch die Methode `setRows()` aufgerufen um die View-Elemente in einen definierten zustand zu überführen.

```
override fun onDetachedFromWindow() {

  this.dataSetObserver.isRegistable=false
  this._tableAdapter?.unregisterDataSetObserver(this.dataSetObserver)

  super.onDetachedFromWindow()

}
```

Beim entfernen der View von Fenster erfolgt der umgekehrte Weg. Zunächst wird das `isRegistable`-Flag gelöscht und der Observer aus dem Adapter entfernt. Das Entfernen der View-Elemente er dann über die Super-Methode durch Aufruf `removeAllViewsInLayout()`.

5.2. View-Container

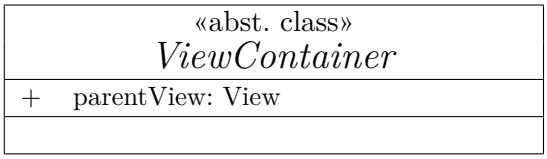


Abbildung 5.11.: Die abstrakte Klasse `ViewContainer`

`ViewContainer`s aus dem Unterpaket *viewContainers* sind abstrakte Datencontainer, die Datenobjekte als View-Elemente definieren und somit eine direkte Schnittstelle zwischen der grafischen Oberfläche und ihren Daten bildet. Die Instanz implementiert das Interface *SetClearable* sodass zu jederzeit geprüft werden kann, ob mögliche Daten gesetzt bzw. auch gültig sind.

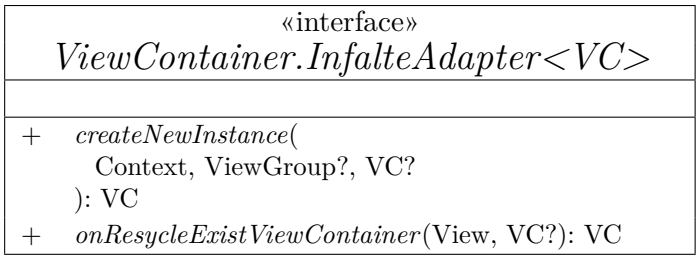


Abbildung 5.12.: Die Interface *ViewContainer.InflateAdapter*

Das Interface *ViewContainer.InflateAdapter* deklariert die Methoden `createNewInstance()` und `onResycleExistViewContainer()` jeweils für ein generisches Objekt `VC` welches von einem `ViewContainer` ableitet. Der *InfalteAdapter* bietet somit die Möglichkeit View-Elemente in *ListAdapter* neu zu erzeugen bzw. auch wieder zu verwenden.

5.2.1. Text-View-Container

Die Klasse `TextViewContainer` leitet von dem `ViewContainer` ab und implementiert darüber hinaus das Interface *Textable* mit dem Attribute `text: CharSequence` als Variable.

Das Objekt TextViewContainer.DefaultInflater implementiert den *InflateAdapter* für ein einfaches Text-Layout der Schriftgröße 20 sp (`normalTextSize`) und wird unter anderem von der `TextListAdapter` als Default-Inflater verwendet.

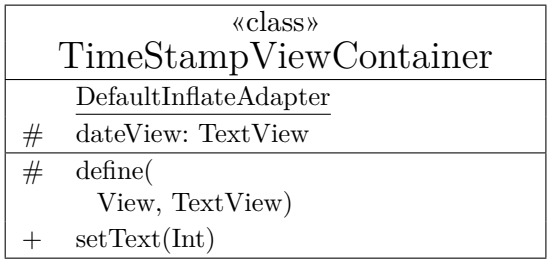


Abbildung 5.13.: Die Klasse `TiemStampViewContainer`

5.2.2. TimeStamp-Container

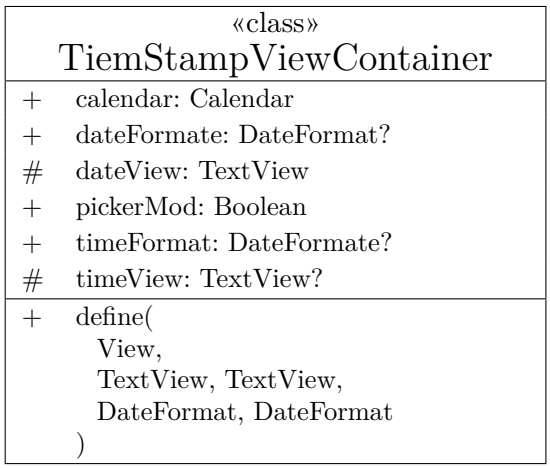


Abbildung 5.14.: Die Klasse `TiemStampViewContainer`

Die Klasse `TimeStampViewContainer` implementiert das Interface *TimeStamp* und definiert die beiden Text-View-Attribute `dateView` und `timneView` sowie zwei Objekte vom Type *DateFormat* für die Textformatierung. Das Attribute `timeStampInMilliSec: Long` ist als Variable die jedezeit neu gesetzt und definiert werden kann. Außerdem können über das lokale Attribute `pickerMod: Boolean` ein Listener-Flag für die `TextViews` gesetzt werden, welches die Definition der Zeit über den `DatePickerDialog` bzw. `TimePickerDialog` ermöglicht.

5.3. Dialoge

5.3.1. Android-API für Dialoge

DialogInterface

Dialog-Klasse

AlertDialog

AlertDialog-Builder

TimeStamp

5.3.2. Dialog Adapter

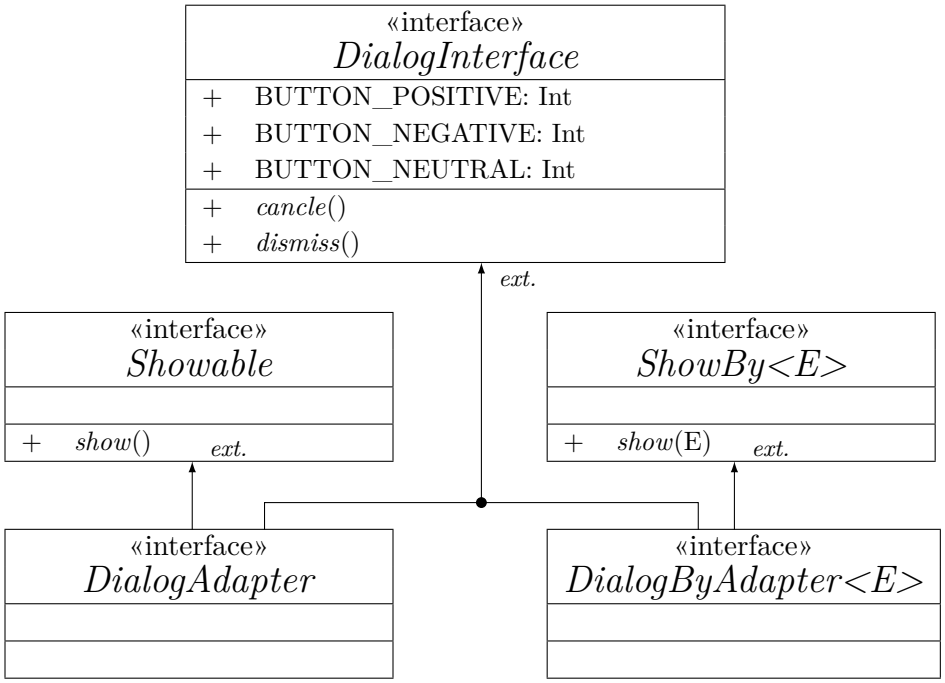


Abbildung 5.15.: Klassenbaum Dialoge

Das Unterpaket *dialogs* definiert eine Reihe von Interfaces und abstrakten Klassen für ein vereinheitlichtes User-Interface über Dialog. Das Basis-Element wird hier vom Interface *Showable* bereit gestellt, mit der abstrakte Methode `show()`. Objekte die dieses Inteface implementieren können somit ein oder mehrere Dialog-Objekt enthalten oder aber auch ein `Toast`. In nächster Instanz definiert der *DialogAdapter* einen abstrakten Dialog der zusätzlich den *DialogInterface* implementiert.

Soll beim Start des Dialogs ein Parameter übergeben werden, so kann die Instanz alternativ zum *Showable* das Interface *ShowBy* implementieren bzw. den hiervon abgeleiteten *DialogByAdapter*.

5.3.3. Dialog-Listener

Das Interface *DialogSupport* leitet von den Interface *DialogInterface* und *BackgroundDrawable* ab und definiert darüber hinaus weitere innere Interface-Listener um auf typische Dialog Ereignisse zu reagieren. Die Interfaces werden in der Regel von Listener-Instanzen implementiert, die zentral auf verschiedene wohl definierte Aktionen reagieren.

Das Interface *DialogSupport.OnCreateByDialogListener* implementiert die abstrakte Methode `onCreate()` für einen beliebigen generischen Typen. Ausgelöst wird das Event von Objekten die das Erzeugen neuer Instanzen über einen Dialog ermöglicht. Objekt die Hierüber nicht erzeugt werden konnten geben als Parameter eine `NULL`-Referenz zurück.

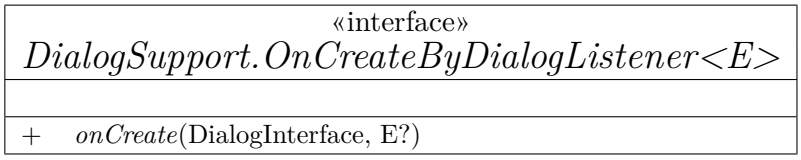


Abbildung 5.16.: Das Interface *DialogSupport.OnCreateByDialogListener*

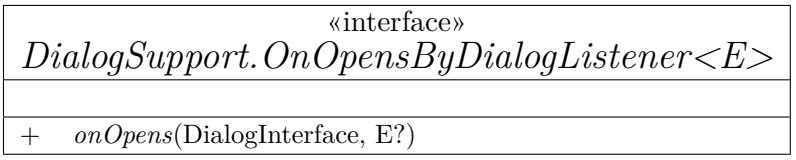


Abbildung 5.17.: Das Interface *DialogSupport.OnOpensByDialogListener*

Statische Objekte oder auch Dateien müssen in einer Anwendung ggf. geöffnet bzw. auch wieder geschlossen werden. Das Interface *DialogSupport.OnOpensByDialogListener* deklariert hierfür die abstrakte Methode `onOpens()` für ein beliebiges generisches Objekt. Konnte eine Instanz nicht geladen bzw. geöffnet werden, so wird analog zu `onCreate()` auch hier eine `Null`-Referenz zurückgegeben.

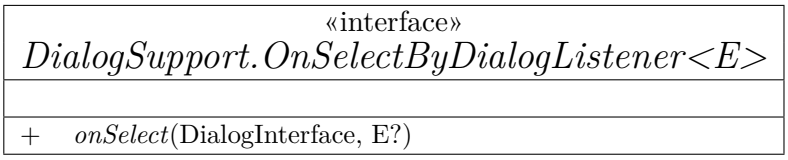


Abbildung 5.18.: Das Interface *DialogSupport.OnSelectByDialogListener*

Das Öffnen bzw. die Auswahl eines Objekt ist meist äquivalent zueinander. In einigen Fällen muss hier ggf. weiter differenziert werden. Für eine Genaue Differenzierung deklariert das Interface *DialogSupport.OnSelectByDialogListener* die Methode `onSelect()`.

Ein weitere wichtige Funktion im Lebenszyklus eines Objektes ist das Ändern ihrer Eigenschaften. Dies kann beispielsweise über das Interface *DialogSupport.OnChangeByDialogListener* mit der abstrakten Methode *onChange()* erfolgen. Der `Boolean`-Parameter gibt hierbei an, ob am Objekt Änderungen vorgenommen worden sind.

Das Interface *DialogSupport.OnStateChangeByDialogListener* ist eine Erweiterung der *OnChangeByDialogListener* und ermöglichst das Ändern konkreter Zustände. Als Übergabe Parameter wird ein beliebiger Schlüssel (`Key`) erwartet sowie der neue generische Objektzustand (textttValue). Für beide Parameter werden typischer weise Enums definiert.

Objekte die über einen Dialog geöffnet wurden, müssen ggf. auch wieder sicher geschlossen werden. Hierzu bietet sich das Interface *DialogSupport.OnCloseByDialogListener* mit der abstrakten Methode `onCLose()` für beliebige generischen Typen an. Der Boolean-Parameter gibt an ob das Objekt erfolgreich geschlossen wurde.

Das Interface *DialogSupport.OnRemoveByDialogListener* ermöglicht außerdem das si-

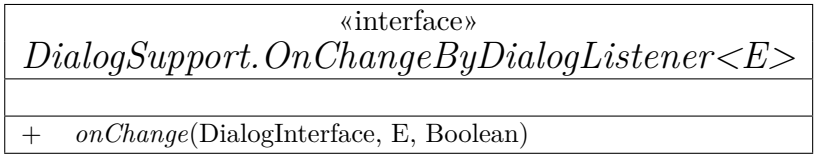


Abbildung 5.19.: Das Interface *DialogSupport.OnChangeByDialogListener*

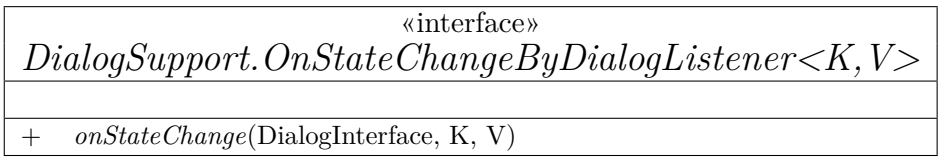


Abbildung 5.20.: Das Interface *DialogSupport.OnStateChangeByDialogListener*

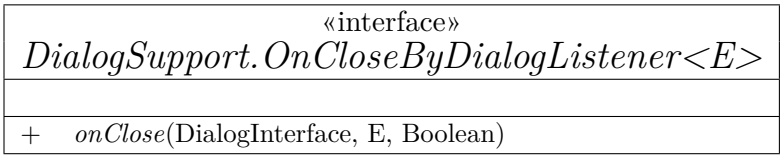


Abbildung 5.21.: Das Interface *DialogSupport.OnCloseByDialogListener*

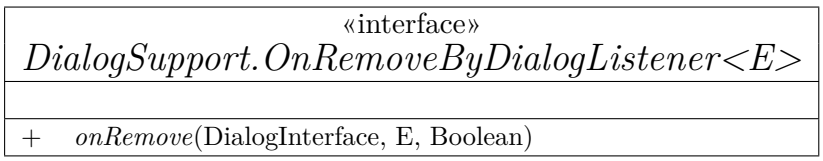


Abbildung 5.22.: Das Interface *DialogSupport.OnRemoveByDialogListener*

chere Entfernen eines Objekts über einen Dialog und deklariert die Methode `onRemove()` für beliebige generische Objekte.

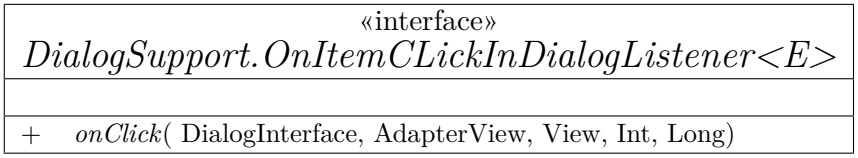


Abbildung 5.23.: Das Interface *DialogSupport.OnItemClickInDialogListener*

Das Interface *DialogSupport.OnItemClickInDialogListener* ist wie der *DialogSupport.OnItemLongClickInDialogListener* ein relativ unspezialisiertes Objekt ermöglicht jedoch das delegieren eines *OnItemClick*-Event einer `ListView`. Entsprechend deklariert es die gleichen Parameter zuzüglich des Dialogs.

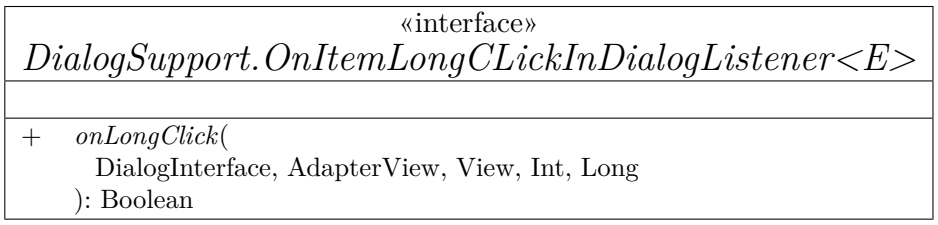


Abbildung 5.24.: Das Interface *DialogSupport.OnItemLongClickInDialogListener*

Der Rückgabeparameter der *DialogSupport.OnItemLongClickInDialogListener* gibt zudem an, ob das Event während seines Durchlaufs abgearbeitet wurde.

5.3.4. Dialog-Containers

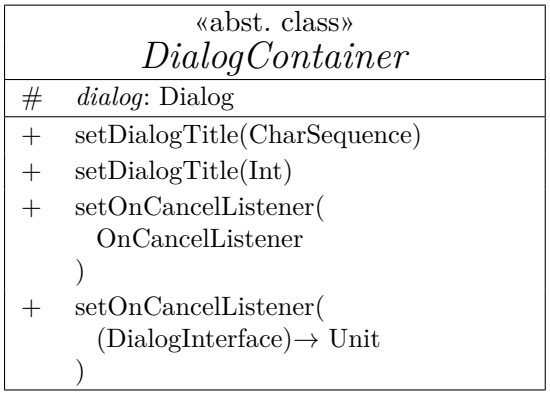


Abbildung 5.25.: Die abstrakte Klasse `DialogContainer`

Der `DialogContainer` bildet eine erste abstrakte Instanz Dialog basierte Interaktionen. Die Klasse implementiert die Interfaces *DialogAdapter* und *DialogSupport* und

delegiert deren Methoden an das abstrakte Attribute `dialog: Dialog`. Darüber hinaus definiert die Instanz setter-Methoden für einen *OnCancelListener* und das Setzen des Dialog-Titels.

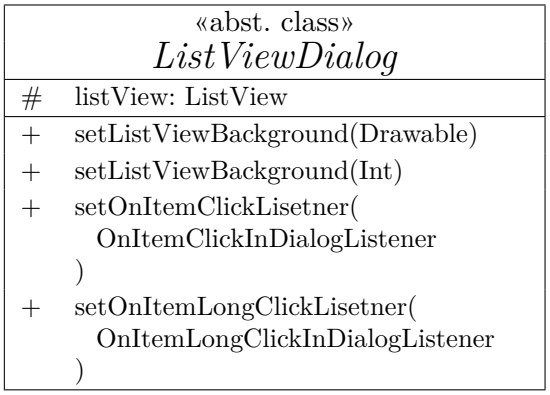


Abbildung 5.26.: Die abstrakte Klasse `ListViewDialog`

Vom `DialogContainer` leitet die ebenfalls abstrakte Klasse `ListViewDialog` ab und erweitert diese um das ebenfalls abstrakte Attribute `listView: ListView`. Analog zum `DialogContainer` werden auch hier einzelne setter-Methoden an die `ListView` delegiert. Darunter gehört auch das Setzen eines *OnItemClickInDialoglistener*.

5.4. Widgets

5.4.1. Abstrakte Datensammlung

Datensammlungen sind bei Android-Anwendungen oft vom Typ *Array* oder alternativ von *Cursor*. Die Darstellung der Sammlung erfolgt in der Regel über eine `AdapterView` mit Hilfe eines *ListAdapter*s. Das Package *widgets* stellt hierfür eine Reihe von Unterobjekten in Form von Interfaces bereit, bei denen der eigentliche *ListAdapter* zunächst in seine Teile zerlegt sind.

In Abbildung (5.27) ist die Klassen Hierarchie für den Datenbereich des *ListAdapter*s dargestellt. Das Interface *ItemSelector* enthält lediglich die abstrakte Methode `getItem(Int): E` für ein beliebiges generisches Objekt `E` als Rückgabetyp. Das Interface *ItemAdapter* leitet hiervon direkt ab und implementiert darüber hinaus die Interfaces *RowDimension* und *EmptyQuantible* aus dem Package *framework.maths.sets*.

Die nächste Erweiterung erfolgt durch das Interface *ItemIdAdapter* mit der abstrakten Methode `getItemId(Int): Long`, sodass hierdurch sämtliche Elemente abgedeckt sind, die den Datenbereich des *ListAdapter*s abbildet.

Vom *ItemIdAdapter* leiten noch die beiden Interfaces *ItemKeyAdapter* bzw. *ItemStringAdapter* ab. Der *ItemKeyAdapter* deklariert die Methode `getItemKey(Int): K` für ein beliebiges generisches Objekt `K` (Key). Als Key wird häufig ein Enum verwendet, zudem das *EnumInteface* implementiert und definiert.

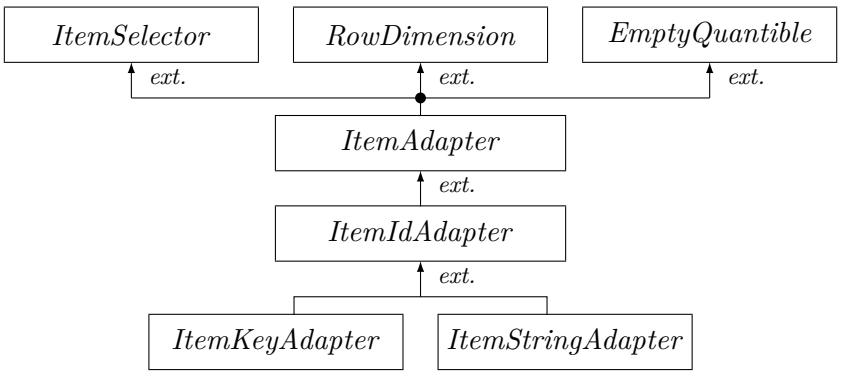


Abbildung 5.27.: Das Interface *ItemAdapter*

Der *ItemStringAdapter* deklariert zudem die Methode `getItemToString(Int, Context): CharSequence` und ermöglicht eine direkte String-Format für das Datenelement.

5.4.2. Item-Adapter

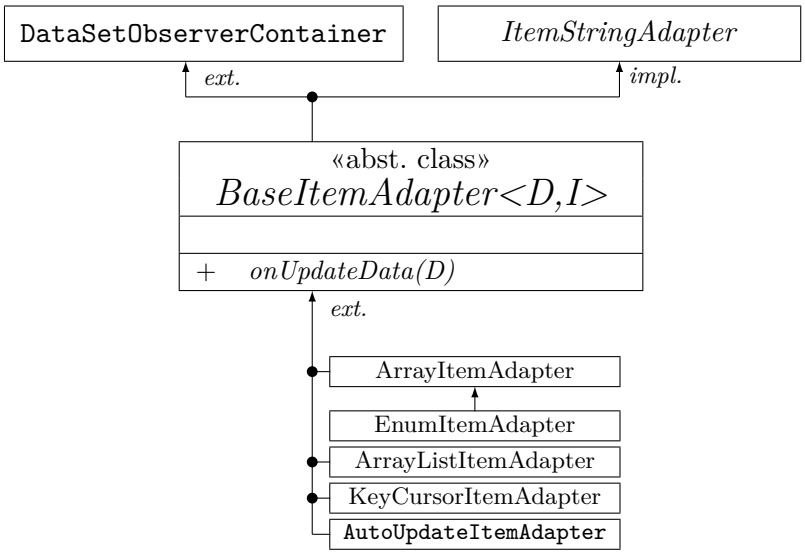


Abbildung 5.28.: Die abstrakte Klasse: `BaseItemIdAdapter`

Das Unterpaket *widgets.itemAdapters* definiert den abstrakten Datencontainer *ItemIdAdapter*. Dieser leitet zunächst von der Klasse `DataSetObserverContainer` aus dem Unterpaket *framework.listeners* ab, implementiert das Interface *ItemStringAdapter* und deklariert die abstrakte Methode `onUpdateData(D)` für ein beliebigen Datencontainer `D` (z.B. *Array*, *Cursor* etc.). Mit Hilfe der Methode können neue Daten gesetzt werden, **ohne** jedoch die Methoden `onInvalidate()` bzw. `onChanged()` aus der Vaterklasse auszuführen.

Die Instanzen `ArrayItemAdapter`, `ArrayListItemAdapter` und `KeyCursorItemAdapter` definieren den Datencotnainer `D` jeweils für den Typ *Array*, *ArrayList* bzw. *KeyCursor* und implementieren ihren Container als Variable. Dabei wird beim Setzen eines neuen Containers die Methoden des `DataSetObserver` automatisch ausgeführt.

Der `EnumItemAdapter` ist ein spezialisierter `ArrayItemAdapter` für Enum-Instanzen. Als Id wird hierbei die Ordnung (`ordinal: Int`) des Enums ausgegeben in ein `Long` konvertiert.

5.4.3. List-Adapter

Abbildung 5.29.: Das Interface *ListAdapter*

5.4.4. TableAdapter

6. Money

[1]

Das Paket *money* beinhaltet Interfaces und konkrete Klassen für den Umgang mit Währungen und deren Komponenten. Um den Speicherverbrauch für Layout-Resourcen möglichst gering zu halten, existiert für Dialoge und View-Container das Erweiterungspaket *moneylayouts*.

6.1. Währungen

Währungen werden allgemein über das Java-Objekt `Currency` abgebildet. Ein konkrete Instanz wird hierbei über statische Methoden erzeugt.

Das Interface *Funds* stellt die Basis für alle Objekte dar, die ein Währungszeichen implementiert. Die spätere Rechenmethoden auch immer ein exaktes Ergebnis liefert, werden Beträge und Summe durch den Datentyp `BigDecimal` repräsentiert. Hierfür stellt *Funds* einige Basismethoden bereit, mit denen die Werte dann besser normiert werden können. Das statische Attribute `scale` gibt hierbei die Anzahl der Nachkommastellen an und ist Standardmäßig auf 2 definiert.

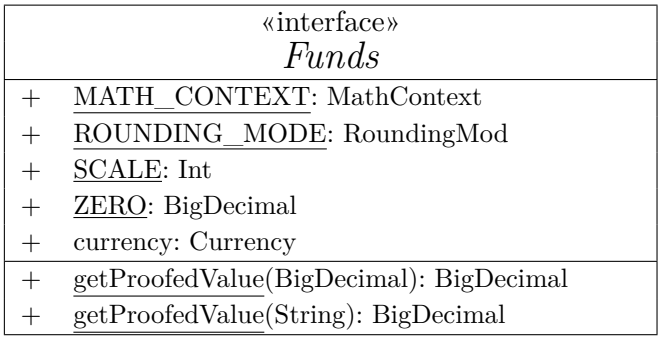


Abbildung 6.1.: Interface: *Funds*

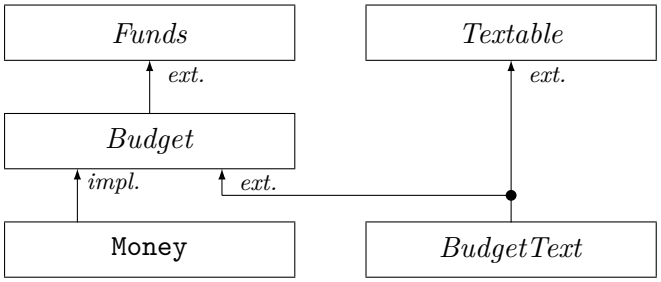


Abbildung 6.2.: Klassenbaum für Budgets

6.1.1. Budgets

Das Interface *Budget* leitet direkt von *Funds* ab und erweitert dieses durch das abstrakte Attribute `value: BigDecimal`. *BudgetText* leitet zusätzlich noch vom Interface *Textable* ab erweitert das Budget für das abstrakte Atrtibute `text: CharSequenze`.

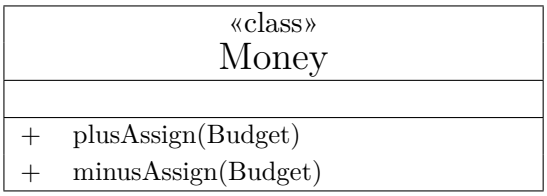


Abbildung 6.3.: Die Klasse `Money`

Die Klasse `Money` leitet von der abstrakten Instanz *java.lang.*`Number` ab und implementiert das Interface *Budget* sowie *Sign* und *FullAddition* aus dem Unterpaket *framework.maths* (Abschnitt 2).

Die Abstrakten Methoden der Klasse `Number` sind hierbei an den aktuellen Betrag `value` vom Typ `BigDecimal` delegiert, sowie auch Element `signum` aus dem *Sign*-Interface.

Die Operatoren der *FullAddition* können für Objekte vom Typ *Budget* als Parameter

[1] zuletzt überarbeitet: 31.01.2021

angewandt werden und liefern jeweils das höherwertige `Money` zurück. Enthält das *Budget* des Parameters eine andere Landeswährung, so wird hier umgehend eine *Exception* ausgelöst.

6.1.2. Handelsaktionen

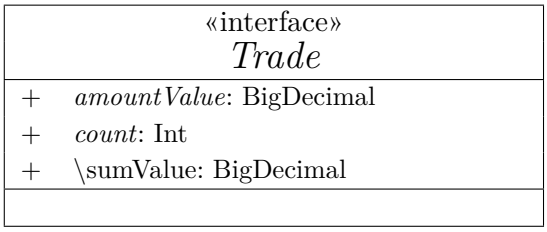


Abbildung 6.4.: Die Klasse `Money`

Für einfache Handelsumsätze definiert das Framework die Interfaces *Trade* bzw. die um das Attribute `text` erweiterte *TradeText* als abstrakten Datencontainer. Die *Trade* leitet hierfür zunächst von *Funds* und *TimeStamp* ab und erweiter diese um die Attribute *amountValue*, *count* und das vordefinierte *sumValue*.

6.1.3. Bilanzen

Bilanzen sind Budgets die sich aus der Summe von Ertrag (`yield: Bidgedimal`) und Aufwand (`expensions: BigDecimal`) zusammensetzen lässt. Hierfür definiert das Framework die Interfaces *BalanceSheet* bzw. *BalanceSheetText* für kommentierte Bilanzen, wobei der Betrag (`value: BigDecimal`) über das Summe von Aufwand und Ertrag vordefiniert ist.

6.1.4. Währungsumrechnung

Das Interface *CurrencyExchange* ist ein abstrakter Datencontainer für das Umrechnen von Länderspezifischen Währungen und enthält die Attribute `exchangeBudget`, `exchangeBudget` und `exchangeBudget`. Darüber hinaus definiert das Interface statische Elemente für die direkte Umrechnung der Elemente. Hierbei wird der `exchangeRate` üblicher Weise mit vier Nachkommastellen definiert (`EXCHANGE_SCALE: Int`).

6.2. Darstellung von Währungen

Das Unterpaket *money.styles* enthält Enum Instanzen für die Text- und Farbformatierung von Währungsangaben für die Bildschirmausgabe. Die können dann im späteren Verlauf von View und View-Container genutzt werden um die Darstellung von Währungen auf der Bildschirmoberfläche zu definieren.

Da die meisten Objekte sowohl die Währung als auch den Betrag anzeigen, sind beide Enum-Klassen als abstraktes Attribut im Interfaces *CurrencyStyle* zusammengefasst.

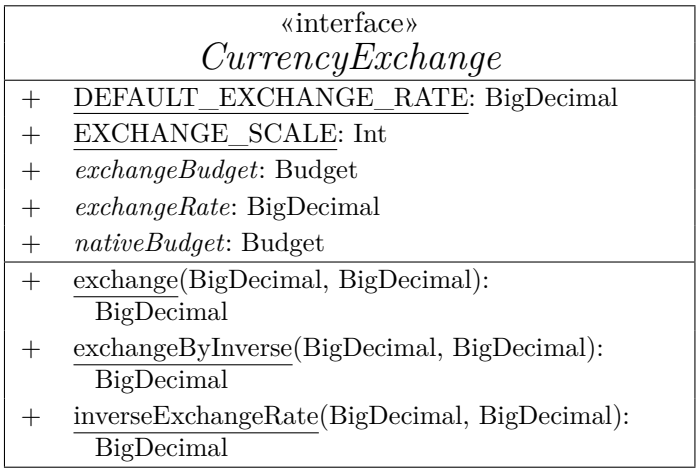


Abbildung 6.5.: Das Interface *CurrencyExchange*

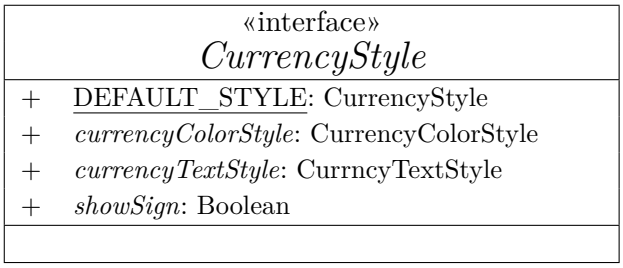


Abbildung 6.6.: Das Enum `CurrencyStyle`

Darüber hinaus enthält die Instanz noch das abstrakte Attribute `showSign: Boolean`, um anzugeben ob auch das Vorzeichen bei negativen Werten angezeigt werden soll oder nicht.

Das statische Attribute `DEFAULT_STYLE: CurrencyStyle` definiert eine Stil-Voralge deren Parameter in der nachfolgenden Tabelle abgebildet sind.

Value	Wert
currencyColorStyle	greenRed
currencyTextStyle	iso4217code
showSign	true

Tabelle 6.1.: Wertezuordnung des Objekts: `DEFAULT_STYLE`

6.2.1. Text-Formatierung

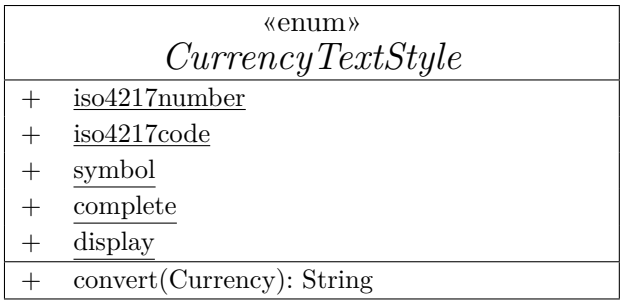


Abbildung 6.7.: Das Enum `CurrencyTextStyle`

Das Enum `CurrencyTextStyle` enthält fünf Attribute für die Textformatierung derer Währung in einer View. Über die Methode `convert()` kann dann ein Objekt von Type `Currency` in die gewünschte String-Konstante überführt werden.

- `iso4217number`: konvertiert eine Währung in die länderspezifische nummerischen Kennziffer;
- `iso4217code`: konvertiert eine Währung in die länderspezifische alphanumerische Kennziffer (z.B. EUR, USD);
- `symbol`: konvertiert die Währung in das länderspezifische Symbol (z.B. $, €);
- `complete`
- `display`

6.2.2. Farbcodierung

Das Enum `CurrencyColor` definiert die drei Grundfarben (Schwarz, Rot, Grün) für eine typische Farbcodierung von Beträgen. Das Enum enthält hierfür die beiden loka-

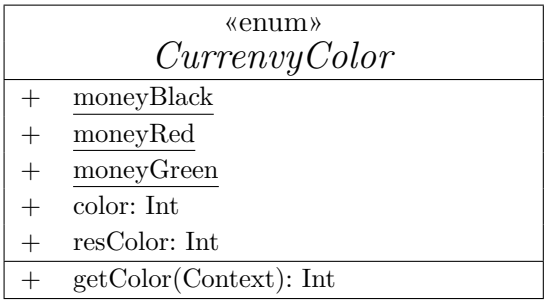


Abbildung 6.8.: Das Enum `CurrencyColor`

len Attribute `color: Int` bzw. `resClor: Int` um den Farbwert alternative über ein Kontext-Objekt auslesen zu können.

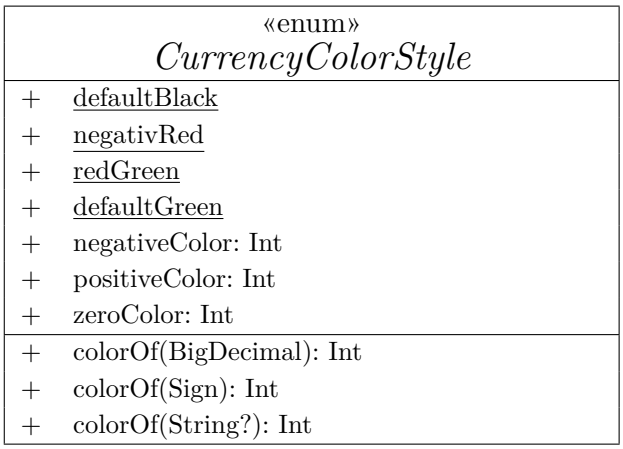


Abbildung 6.9.: Das Enum `CurrencyColorStyle`

Mit Hilfe des Enums `CurrencyColorStyle` kann der Betrag eines Wertes in Abhängigkeit seines Vorzeichens ermittelt und eingefärbt werden. Hierfür stellt das Enum drei drei Konstanten für *positve*, *negative* und *zero*-Beträge und kann durch Aufruf der Methode `colorOf()` mit einem Parameter vom Typ `BigDecimal` ausgelesen werden.

- `defaultBlack`: Alle Beträge werden in schwarz dargestellt;
- `negativeRed`: Negative Werte werden in rot dargestellt, 0 und positive Werte in schwarz;
- `greenRed`: Negative Werte werden in rot, 0-Werte in schwarz und positive Werte in grün dargestellt;
- `defaultRed`: alle Werte werden sind in rot dargestellt;
- `defaultGreen`: alle Werte werden sind in grün dargestellt;

6.3. Budget-View

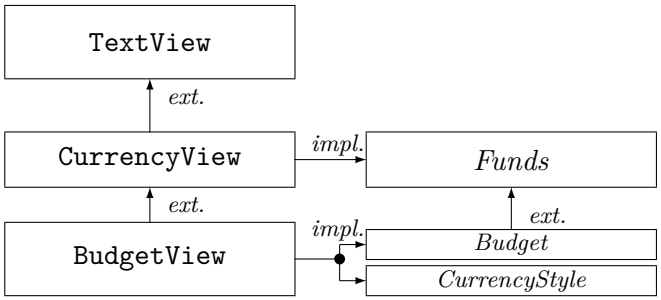


Abbildung 6.10.: Klassendiagramm der `BudgetView`

Das Unterpaket *money.views* enthält die beiden Instanzen `CurrencyView` und `BudgetView`. Die `CurrencyView` leitet von einer gewöhnlichen `TextView` ab und implementiert das Interface *Funds* sowie das Attribute `currencyTextStyle`. Die `BudgetView` leitet ihrerseits von der `currencyView` ab und implementiert darüber hinaus das Interface *Budget* und *CurrencyStyle*.

Die abstrakten Attribute aus den Interfaces sind jeweils als Variable realisiert, sodass sie jederzeit geändert werden können. Die Formatierung der TextView erfolgt schließlich durch Aufruf der geschützten Methode `onChangeText()` sobald ein Parameter der View geändert wird.

6.4. View-Container

«interface» *CurrencyViewContainer*	
+	*parentView*: View
+	*currencyView*: View
+	*setCurrencyStyle*(CurrencyStyle)

Abbildung 6.11.: Das Interface: *CurrencyViewContainer*

Das Interface *CurrencyViewContainer* ist eine abstrakte Adapter-Instanz für View-Containers, die Währungen oder vergleichbares abbilden. Der *vCurrencyViewContainer* leitet von den Interfaces *Funds* und *CurrencyStyle* ab und erweiterte diese um die abstrakten Attribute `parentView` und `currencyView` sowie die abstrakte Methode `setCurrencyStyle()`.

Konkrete Klassen die dieses Interface implementieren leiten in der Regel von einem `TextViewContainer` ab, mindestens aber von seiner Vaterinstanz dem `ViewContainer`.

Der *CurrencyViewContainer* definiert analog zum `ViewContainer` die Adapter Instanz *CurrencyViewContainer.InflateAdapter*, welcher von der *ViewContainer.InflateAdapter*

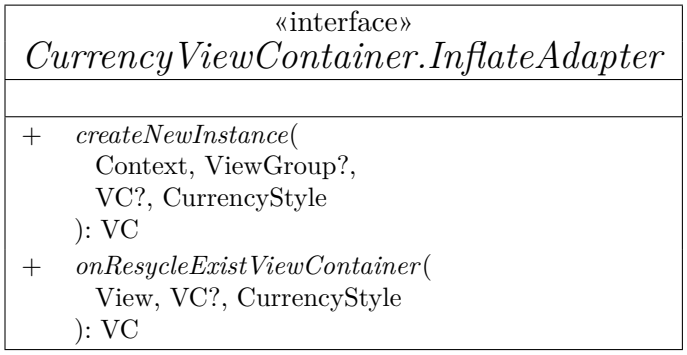


Abbildung 6.12.: Das Interface: *CurrencyViewContainer.InflateAdapter*

und einem *CurrencyStyle* ableitet und deren Methoden mit für dem zusätzlichen Parameter `CurrencyStyle` überlädt. Der Rückgabe Typ `VC` ist hierbei ein generischer Typ, der sowohl vom `ViewContainer` ableitet als auch den *CurrencyViewContainer* implementiert. Konkrete Inflate-Adapter sind Teil der Paketerweiterung in *moneylayouts.layoutInflater*.

6.4.1. Money-View-Container

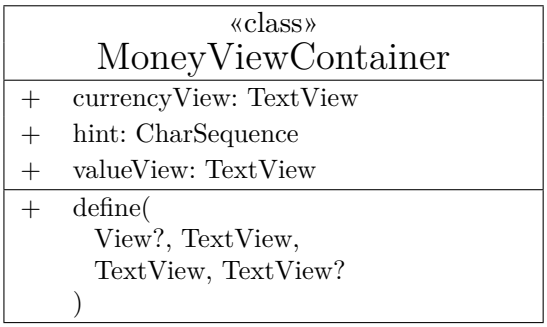


Abbildung 6.13.: Die Klasse `MoneyViewContainer`

Die Klasse `MoneyViewContainer` leitet von der Instanz `TextViewContainer` ab und implementiert die Interfaces *BudgetText* bzw. *CurrencyViewContainer*. Von der Vaterinstanz erbt die Klasse bereits die beiden Attribute `parentView` und `textView`, sodass die Klasse lediglich um die beiden `TextViews` `currencyView` und `valueView` erweitert wird. Mit Hilfe der Methode `define()` können sämtlich View-Elemente gesetzt werden.

Die Abbildung des Interfaces *Budget* erfolgt innerhalb des Containers über zwei separate Text-View-Elemente, die beim Setzen der Beträge und Währungen synchronisiert werden.

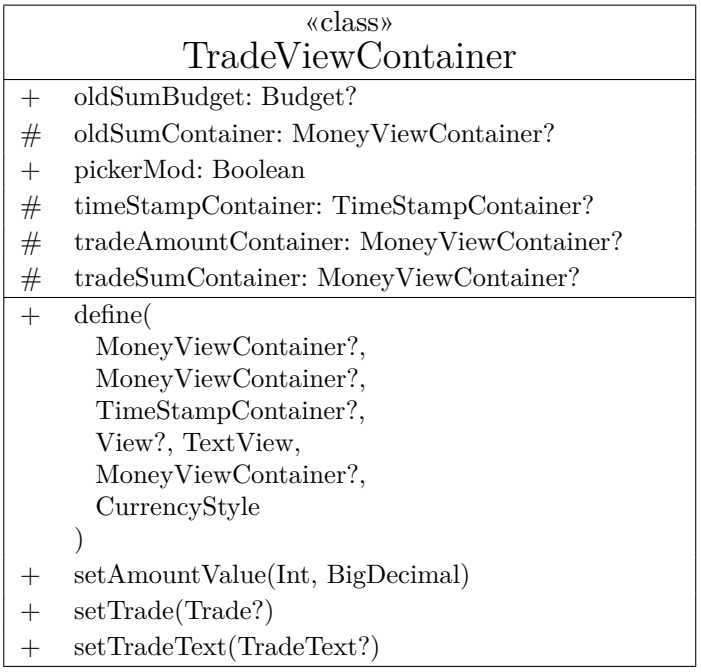


Abbildung 6.14.: Die Klasse `TradeViewContainer`

6.4.2. Trade-View-Container

Die Klasse `TradeViewContainer` leitet von der Instanz `TextViewContainer` ab und implementiert die Interfaces *TradeText* und *CurrencyViewContainer*. Für den Zeitstempel implementiert die Klasse das Attribute `timeStampContainer`, deren PcikerMod über die lokale Instanz gesetzt werden kann. Die Darstellung der einzelnen Beträge (`amountValue, sumValue`) erfolgt dann jeweils über eine `MoneyView`. Darüber hinaus stellt die Instanz das optionale Attribute `oldSumBudget` bereit um ggf. den akutellen Betrag einer Summe anzuzeigen.

6.4.3. Currency-Exchange-Container

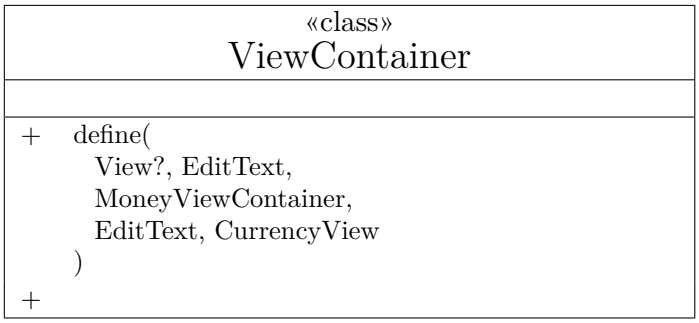


Abbildung 6.15.: Die Klasse `ViewContainer`

Teil II.

Erweiterungen

7. Erweiterungspaket Permission-Handler

[1]

Die Android-API gibt die Möglichkeit auf Hardwarekomponenten des Gerätes zuzugreifen. Hierfür wird vom Benützer zuvor eine *Berechtigung* (*Permission*) eingefordert, welche vom Betriebssystem verwaltet und gesteuert wird. Neben dem Hardwarezugriff sollte jede App darüber hinaus auch *Nutzungsbedingung* und Datenschutzerklärung enthalten, die im Rahmen der Erweiterung *permissionhandler* verwaltet werden kann.

Neben dem dem Verwalten von Berechtigungen enthält dieses Modul auch eine kleine Erweiterung für das Impressum. Dieses kann dann zusammen mit den einzelnen Berechtigungen beispielsweise in einem Dreipunktmenü eingesehen werden.

7.1. Impressum

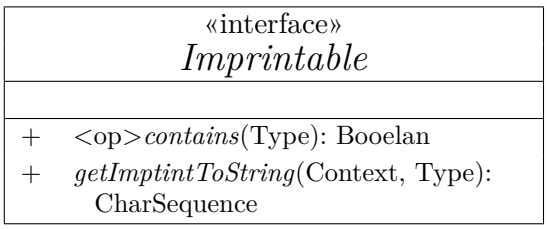


Abbildung 7.1.: Das Interface *Imprintable*

Für das *Impressum* (engl. *Imprint*) stellt das Package das *permissionHandler* das Interface *Imprintable* bereit. Diese besteht aus dem Operator `contains(): Boolean` für den Enum-Parameter `Imprintable.Type`.

[1] zuletzt überarbeitet: 04.02.2022

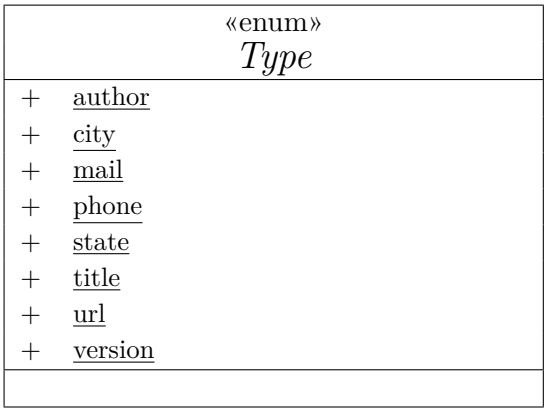


Abbildung 7.2.: Das Enum `Imprintable.Type`

Darüber hinaus deklariert das Interface die Methode `getImprintToString(): CharSequence` um die Werte der einzelnen Impressum-Typen auszulesen. Die einzelnen Enum Typen sind im hierfür im Klassendiagramm (7.2) aufgeführt.

Das Unterpaket *adapters* definiert mit der abstrakten Instanz `ImprintAdapter` einen `ArrayItemAdapter` für die spätere Darstellung in einem *ListAdapter*. Der `ImprintDialog` aus dem Unterpakt *dialogs* zeigt das Impressum schließlich als Dialog.

7.2. Permission Enum

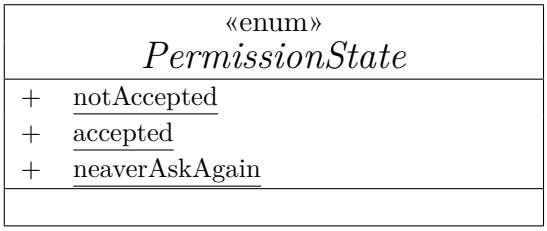


Abbildung 7.3.: Das Interface *PermissionState*

Die Enum Klasse `PermissionState` definiert die drei Elemente `accepted`, `notAccepted` und `neverAskAgain` und adaptiert somit die Android spezifischen Statusmöglichkeiten für Berechtigungen.

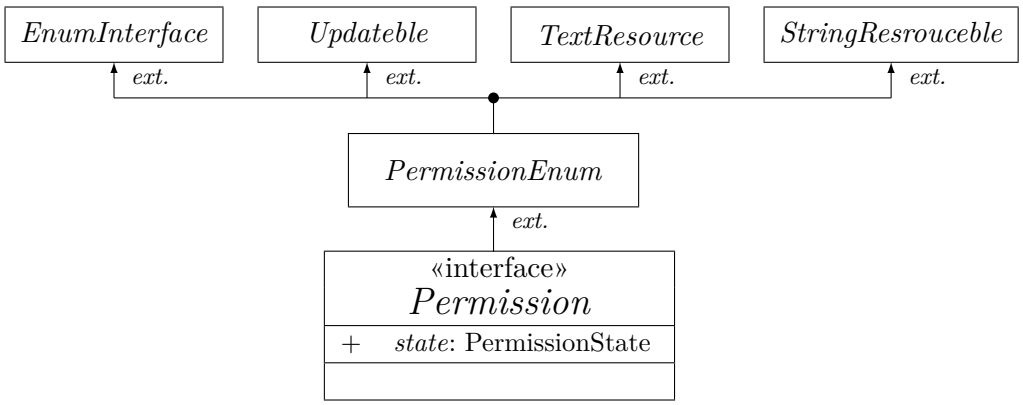


Abbildung 7.4.: Das Interface *Permission*

Abbildung (7.4) zeigt den Klassenaufbau für Berichtigungen. Das Interface *PermissionEnum* definiert eine erste abstrakte Enum-Instanz welche von *EnumInterface*, *TextResouce StringResoruce* und *Updateble* ableitet. Das Interface *Permission* erweitert das *PermissionEnum* und das abstrakte Attribute `sate: PermissionState`.

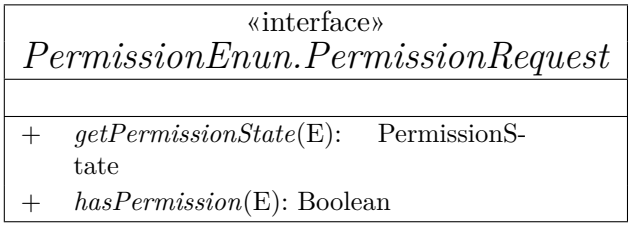


Abbildung 7.5.: Das Interface *PermissionEnun.PermissionRequest*

Die *PermissionEnum*-Instanz enthält darüber hinaus weitere Sub-Interfaces, für die

statische Verwaltung von Berechtigungen. Das Sub-Interface *PermissionEnun.PermissionRequest* erlaubt zunächst das Auslesen des Statusflags.

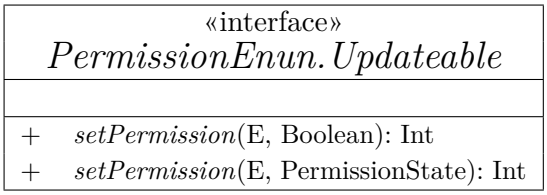


Abbildung 7.6.: Das Interface *PermissionEnun.Updateable*

Das Setzen des Stauts erfolgt über das Sub-Interface *PermissionEnun.Updateable* mit seinen setter-Methoden.

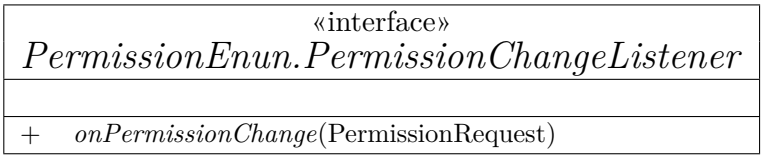


Abbildung 7.7.: Das Interface *PermissionEnun.PermissionChangeListener*

Das Interface *PermissionEnun.PermissionChangeListener* enthält die Methode `onPermissionChange()` mit einem *PermissionRequest* als Übergabeparameter.

7.3. Verwaltung von Berechtigungen

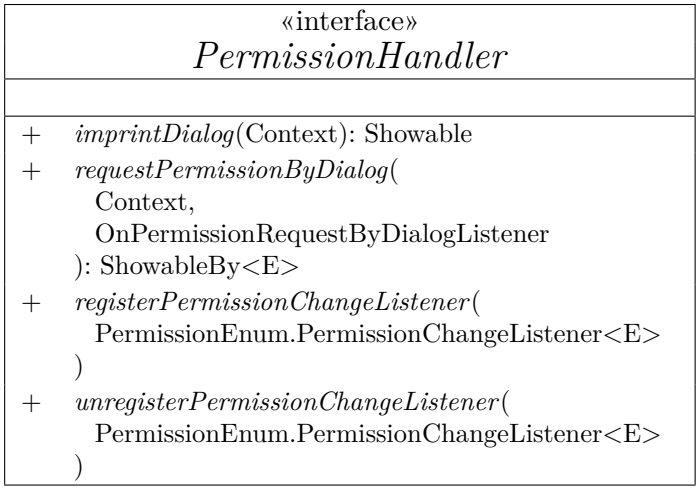


Abbildung 7.8.: Das Interface *PermissionHandler*

8. Erweiterungspaket InputLayouts

[1]

Das Erweiterungspakt *inputLayouts* wurde für das Projekt *Datamining* entwickelt, um die Dateneingabe für beliebige Tabellenattribute zu ermöglichen. Es implementiert vorwiegend `ViewContainer`s mit einheitlichen Layout-Resorucen für die Eingabe von Daten. Dem Eingeschlossen gehört auch ein *ListAdapter* an, der die Dateneingabe über eine ListzView ermöglicht. Das Erweiterungspakt ist abgestimmt auf das Unterpakt *framework.tables*, sodass deren Interfaces hier teilweise implementiert sind.

8.1. ViewContainers

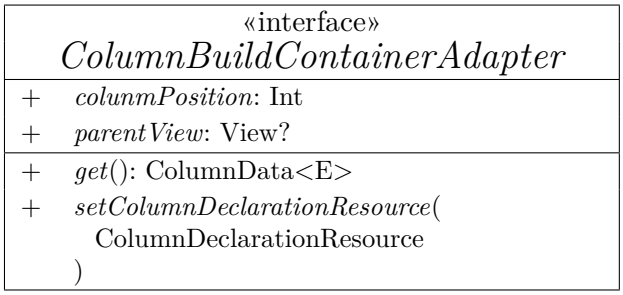


Abbildung 8.1.: Das Interface *ColumnBuildContainerAdapter*

Das Interface *ColumnBuildContainerAdapter* definiert einen ersten abstrakten Datencontainer für Eingabefelder. Dieser leitet von den Interfaces *ColumnData* bzw. *SetClearable* ab und erweitert dieses um das abstrakte Attribute *columnPoisition*, sowie um `getter`- und `setter`-Methoden für die einzelnen Elemente.

[1] zuletzt überarbeitet: 14.02.2022

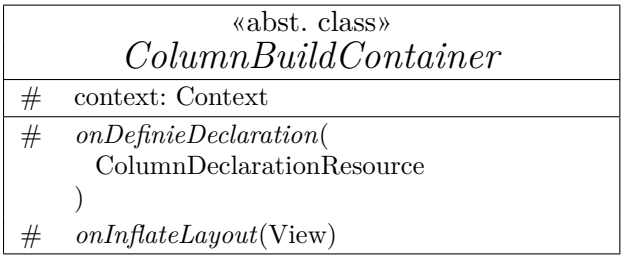


Abbildung 8.2.: Das Interface *ColumnBuildContainerAdapter*

Die abstrakten Instanz `ColumnBuildContainer` implementiert Interface *ColumnBuildContainerAdapter*, leitet von einem `ViewContainer` ab und bildet eine erste Basis für Input-Container und welcher das implementiert und fast vollständig definiert. Mit Ausnahme des `TimeColumnBuildContainer`s leitet alle Build-Containers von `ColumnBuildContainer` ab.

Die einzelnen View-Elemente sind schließlich in den konkreten Klassen realisiert und enthalten mindestens eine `TextView` für die Spaltendeklaration sowie ein Eingabefeld (`EditText`) für die Daten.

- `BigDecimalColumnBuildContainer`;
- `BooleanColumnBuildContainer`;
- `IntegerColumnBuildContainer`;
- `LinkedIdColumnBuildContainer`;
- `MoneyColumnBuildContainer`;
- `RealColumnBuildContainer`;
- `StringColumnBuildContainer`;
- `TimeColumnBuildContainer`;

8.1.1. Eingabefelder für Zahlenwerte

8.1.2. Eingabefeld für den Datentyp Boolean

8.1.3. Eingabefeld für verlinkte Datensegmente

8.1.4. Eingabefeld für Zeitstempel

8.2. Der Data-Input-Adapter

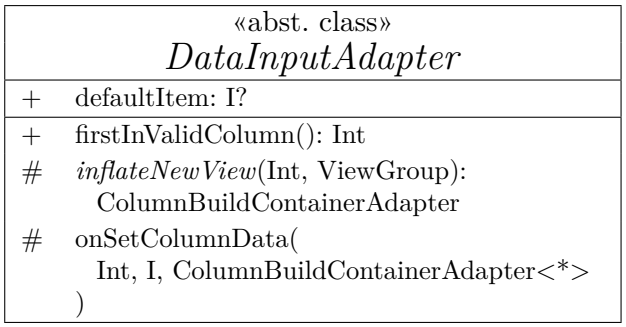


Abbildung 8.3.: Das Interface *ColumnBuildContainerAdapter*

Die abstrakte Instanz `ColumnBuildContainerAdapter` implementiert die Interfaces *ItemListAdapter* sowie *ClearableBuilder*. Initialisiert wird das Objekt über eine *TableDeclarationResource* und einem `defaultItem: I?`, zur zur Initialisierung der View-Elemente. Um die View-Elemente zu entfalten enthält der Adapter ein Array für die jeweiligen Container-Objekte. Da der `ListAdapter` nur genau ein Datensegment repräsentiert, sollte er auch nur in genau einer `ListView` zur Anzeige gebracht werden.

8.2.1. Handling im Dialog

Üblicherweise erfolgt der Focus für die Eingabefelder über das Betriebssystem automatisch. Werden die Item jedoch über den Adapter definiert, so muss der Fokus ggf. manuell gesetzt werden. Für eine Anwendung im Dialog muss dieser zunächst über die `show()`-Methode initialisiert sein, anschließend können die entsprechenden Flags über ein `Window`-Objekt definiert werden.

```
this.dialog = builder.show()

val window = this.dialog.window!!

window.clearFlags(
   WindowManager.LayoutParams.FLAG_NOT_FOCUSABLE or
   WindowManager.LayoutParams.FLAG_ALT_FOCUSABLE_IM
)
window.setSoftInputMode(
   WindowManager.LayoutParams.SOFT_INPUT_STATE_ALWAYS_VISIBLE
)

this.dialog.cancel()
```

8.2.2. Fehlerhinweise

Der `DataInputAdapter` darf von seinem Design nur einmal pro `ListView` verwendet werden. Das heißt zugleich dass auch nur ein `DataSetObserver` registriert ist. Die Prüfung erfolgt hierbei über die Methoden der `registerDataSetObserver()` bzw. `unregisterDataSetObserver()`.

```
override fun registerDataSetObserver(dataSetObserver: DataSetObserver) {

     require(
        this.dataSetObserver==null || this.dataSetObserver===dataSetObserver
     ){"dataSetObserver is set already"}
     this.dataSetObserver=dataSetObserver

    }

override fun unregisterDataSetObserver(dataSetObserver: DataSetObserver) {

     require(this.dataSetObserver===dataSetObserver)
        {"cannot clear this datasetObserver"}

     //this.clearContainers()
     this.dataSetObserver=null

 }
```

Für einen sauberen Ablauf wurde um löschen des `DataSetObserver` zunächst sämtliche Build-Layout-Resourcen aus dem Array auf `null` gesetzt. Es zeigte sich jedoch, dass insbesondere beim Handling in Kombination mit einem Dialog, der `ClickListener` aus dem `LinkedIdColumnBuildContainer` nicht mehr funktioniert. Daher ist die Methode vorerst mit `Deprecpted` deklariert.

```
 @Deprecated("")
 private fun clearContainers()
 {
     for(i in 0 until  builderLayouts.size) {
            builderLayouts[i] = null
     }
 }
```

9. Erweiterungspaket Contact-Layouts

[1]

Das Erweiterungspakt *contactLayouts* definiert Interfaces und View-Containers für das Erstellen von *Kontaktdaten*.

9.1. Name

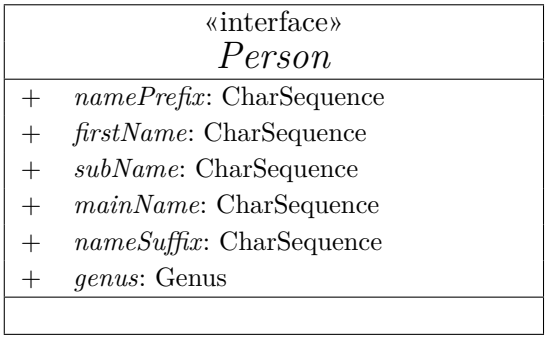


Abbildung 9.1.: Das Interface *Person*

Das Interface *Person* und ist ein abstrakter Datencontainer für Personen und enthält die gleichen Attribute wie sie auch aus den Kontaktfeldern der Android-API bekannt sind. Das Interface leitet außerdem von *Nameble* ab und implementiert eine default-definiert für das abstrakten Attribute *name: Charsequence*.

Das Geschlecht einer Person ist über das Enum `Person.Genus` mit seinen vier Attributen bestimmt. Das Enum implementiert zudem die Interfaces *EnumInterfaces*, *StringResourceble* und *Symbol*.

Das Geschlecht `divers` übernimmt hier zugleich den Platzhalter für »*unbestimmt*« und ist mit der Ordnungszahl 0 daher als erstes gesetzt. Mit Hilfe des Typs `comtpany` können zudem auch Unternehmen als juristische Personen kodiert werden.

[1] zuletzt überarbeitet: 14.02.2022

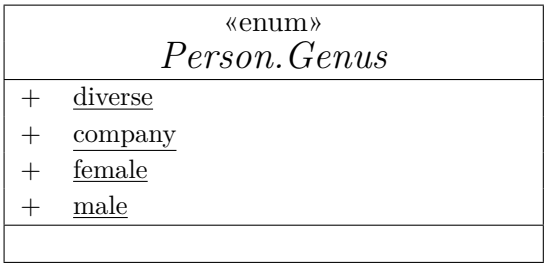


Abbildung 9.2.: Das Enum `Person.Genus`

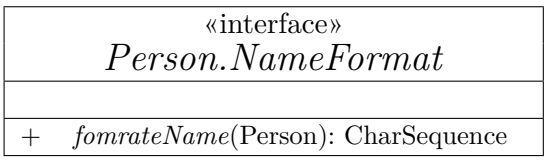


Abbildung 9.3.: Das Interface *Person.NameFormat*

9.2. Adresse

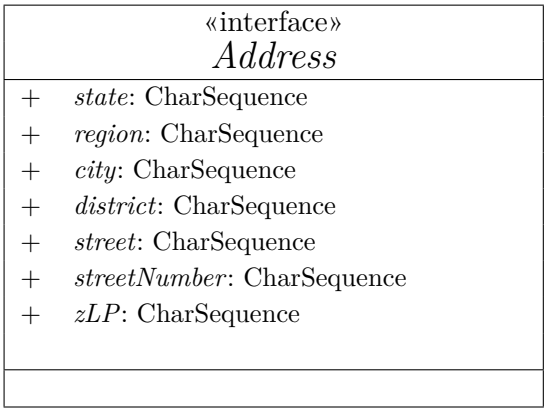


Abbildung 9.4.: Das Interface *Address*

Das Interface *Address* leitet vom Interface *Adressable* ab definiert abstrakte Attribute für Staat, Stadt, Straße für das Setzen von Adressen.

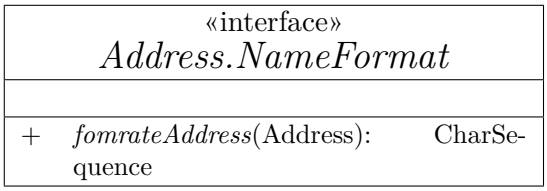


Abbildung 9.5.: Das Interface *Address.NameFormat*

10. Erweiterungspaket Money-Layouts

[1]

Das Erweiterungspaket *moneyLayouts* basiert Hauptsächlich auf das Paket *framework.money* und definiert vorwiegend Layoutresourcen für die Interfaces *BudgetText*, *TradeText* und *BalanceSheetText*. Des weiteren enthält diese Unterpaket auch vorgefertigte Dialoge, mit den Objekte vom Typ *CurrencyStyle* erzeugt werden können.

10.1. Layout-Resourcen

10.2. Currency-Style-Picker

[1] zuletzt überarbeitet: 14.02.2022

A. Anhang

Abbildungsverzeichnis

Tabellenverzeichnis

Index