# Musicdroid

The intention of this document is to give a better understanding of some of the concepts we use in the project.

Created: 29.12.2014
Last change: 06.01.2015

# How to start

We recommend these steps when starting with the project:

- Read the book “Clean Coder” by Robert C. Martin: https://www.ufm.edu/images/0/04/Clean_Code.pdf
- Read this document for some basic knowledge
- Install the tools (Git, Android Studio etc.)
- Checkout the project (GitHub)
- Get a current issue from Jira and create a new branch on GitHub. In the best case scenario someone will work together with you.
    - Have a look at some classes and their tests to get a better understanding while working on the issue.
    - Implement your solution.
    - **ALWAYS CREATE TESTS ☺**
    - Create a pull request on GitHub after you are done. Someone else will have a look at your code. After this the branch will either be merged into the develop branch or the reviewer will leave a note on what needs to be improved before a merge can be done.

*Don’t be afraid to ask questions!*

**Naming Conventions**

- **GitHub**
    - Creating a new branch: The name of the branch starts with the Jira issue (MUS-<Number>_) lead with some key words about the task (without a space).
      **Examples:**
        - MUS-12_DrawBreak
        - MUS-20_HitBoxAlgorithm
        - MUS-30_RefactoringSymbol
    - Creating a commit: The commit title should start with the Jira task name number, followed by a short description of what was done. For more details the commit message can be used.
      **Examples:**
        - MUS-12 Added BreakSymbol
        - MUS-20 Refactored algorithm
        - MUS-30 Added test cases for refactored Symbol
- **Android Studio**
    - We are using the coding convention that is common to Java development

# How music is created

This chapter explains the underlying basic and more complex concepts which are needed to create music in Musicdroid.

**Package:** org.catrobat.musicdroid.pocketmusic.note

## Basics

*A basic understanding of music and musical notation is very helpful in understanding the following concepts.*

In this section we are going to cover some basic vocabulary:

- **MIDI (Musical Instrument Digital Interface):** Standard for the exchange of musical information. It can be used to notate for example when to press which key on a piano (and in which way). Note though that a MIDI file is much simpler than a MP3 file as it is not a file consisting of sound waves but of control information. Time (knowing when an event like a piano key press starts or ends) is described through ticks.
- **Beats per minute:** Indicator for the amount of quarter notes per minute. 60 beats per minute = 60 quarter notes per minute = 30 half notes per minute etc.
- **Tick:** A tick is used to describe the length of an event (like a sound event) in the MIDI format. Ticks can be converted into seconds.
  - *Ticks = seconds * beats per minute * 8*
  - Playing an instrument at 60 beats per minute means that a single quarter note lasts for 1 second or 480 ticks ($1 \text{ second} * 60 \text{ beats per minute} * 8 = 480 \text{ ticks}$).
- **C4 or other musical notes:** We are using the scientific pitch notation in which all notes are written in capital letters. Most German people have learned the Helmholtz pitch notation in school:
  - A C4 is written as c1 or c' in Helmholtz notation.
  - A C5 is written as c2 or c'' in Helmholtz notation. It is also known as "tenor c" or in German "das hohe c".

This is a short list of enums to describe the most basic musical concepts we use:

- `MusicalInstrument:` An instrument like a piano or a guitar.
- `MusicalKey:` Violin and bass. Used to map the staff (set of five horizontal lines) to musical notes.
- `NoteLength:` Describes the length of a musical note (quarter, half, whole and so on).
- `NoteName:` the name of a musical note (C4, D5 etc.).
- `NoteFlag:` Shows which kind of flag a musical note has. A quarter note for example has no flag, while a sixteenth note has two).

## NoteEvent

A `NoteEvent` is a simple structure which indicates the beginning and the end of a certain note:

```
NoteEvent(NoteName noteName, boolean noteOn)
```

**Example:**

```
NoteEvent(NoteName.C4, true) → start of C4
NoteEvent(NoteName.C4, false) → end of C4
```

## Track

A `Track` is the recording of a single instrument. It consists of a map of ticks and `NoteEvents`.

Through the map of ticks and `NoteEvents` we know when to play which musical note for how long. This structure is very similar to the MIDI format.

**Note:** Even though the structure of a `Track` is similar to the MIDI format, we do not have all the features that this format offers.

```
Track(MusicalKey key, MusicalInstrument instrument, int
beatsPerMinute)
```

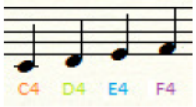
**Example:**

```
Track(MusicalKey.VIOLIN, MusicalInstrument.ACOUSTIC_GRAND_PIANO, 60)
```

Here are two examples of short pieces of music written down in musical notes and the corresponding representation as a `Track`:

**Example #1:**

60 beats per minute, one quarter note equals 480 ticks

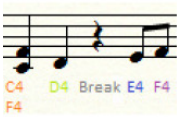



| Tick: | 0 | 480 | 960 | 1440 | 1920 |
|---|---|---|---|---|---|
| NoteEvents: | C4 start | C4 end | | | |
| | | D4 start | D4 end | | |
| | | | E4 start | E4 end | |
| | | | | F4 start | F4 end |

**Example #1** shows a very simple `Track` object. Four quarter notes are played in a row without a break. The tick numbers are markers in the “timeline”. This means that we have one event at tick 0, two events at 480, two events at 960, two events at 1440 and one event at 1920. Through subtraction we can determine the length of a musical note: F4 starts at 1440 and ends at 1920. F4 had a length of 480 ticks ($1920 - 1440 = 480$).

**Example #2:**

60 beats per minute, one quarter note equals 480 ticks




**Break between 960 and 1440**
**E4 starts at 1440 because of this**
**(There is no "BreakEvent")**

| Tick: | 0 | 480 | 960 | 1440 | 1680 | 1920 |
|---|---|---|---|---|---|---|
| NoteEvents: | C4 start | C4 end | | | | |
| | F4 start | F4 end | | | | |
| | | D4 start | D4 end | | | |
| | | | | E4 start | E4 end | |
| | | | | | F4 start | F4 end |

**Example #2** shows a more complicated `Track` object with a break and simultaneous played musical notes. As the break describes a part of no sound in the track, there is no such thing as a "BreakEvent". A break only changes the start of the next occurring `NoteEvent`. Compare the green and the blue events of both tracks to see the difference. Also notice the shorter length of E4 ($1680 - 1440 = 240$) and F4 ($1920 - 1680 = 240$).

## Project

A `Project` is a container of played music. For this it consists of zero to many `Tracks`.

```
Project(int beatsPerMinute) [0-n tracks]
```

**Example:**

```
Project(60) [empty]
Project(120) [piano track]
Project(120) [piano track, guitar track]
```

A `Project` object can be saved as a MIDI file. We will have a closer look at MIDI (and how music is played) in the next chapter.

# How music is stored and played

An object of class `Project` holds all necessary information to play music. We are using the MIDI format to save and load a project. Also such a MIDI file can be played easily by most media players.

**Package:** org.catrobat.musicdroid.pocketmusic.note.midi

## ProjectToMidiConverter

This class is used to save a `Project` object as a MIDI file. We are using an external library to do this.

### MIDI file

Similar to the `Project` class a MIDI file also consists of several tracks. Besides the given musical information we can extract from our own `Track` objects, a MIDI track can also hold meta information like beats per minute or simple text. Because of this we are using the first track in a MIDI file to mark it as a `Project`.

**Example:**

`Project` object with 60 beats per minute:

- Piano `Track` object
- Guitar `Track` object

Will be converted into

MIDI file:

- META track (60 beats per minute, Musicdroid Midi File)
- piano midi track
- guitar midi track

## MidiToProjectConverter

This class is used to save a `Project` to a MIDI file. To do this the file must be a Musicdroid MIDI file (see above).

# How music is displayed

Drawing an object of class `Track` is a two part process:

1. We convert the object into a list of symbols (this can be seen as the reverse operation to the two picture examples in the chapter about the `Track` class)
   **Package:** org.catrobat.musicdroid.pocketmusic.symbol
2. We draw each symbol in the list
   **Package:** org.catrobat.musicdroid.pocketmusic.draw

## Converting symbols

`Symbol` is the basic interface which is used for all symbols. Currently there are two types: `BreakSymbol` and `NoteSymbol`.

The `BreakSymbol` is a very simple class which only holds a `NoteLength` object, while a `NoteSymbol` object can consist of several musical notes. Because of this we are going to have a closer look at it in the next section.

### NoteSymbol

```
NoteSymbol() [0-n musical notes]
```

For adding a single musical note to a `NoteSymbol` object two parameters are needed: `NoteName` and `NoteLength`.

This class also provides wrapper methods which describe the overall shape of the symbol:

- `hasStem()`
- `isStemUp()`
- `getFlag()`

Also it can provide a list of all `NoteName` objects it holds in a sorted format.

**Example:**

```
NoteSymbol() { NoteName.F3, NoteLength.EIGHT }
NoteSymbol() { [NoteName.C4, NoteLength.QUARTER], [NoteName.E5,
NoteLength.QUARTER] }
```

### TrackToSymbolsConverter / SymbolsToTrackConverter

These two classes are used to convert a track to a list of symbols and vice versa.

## Drawing symbols

After creating the list of symbols we can start drawing them. Android's basic class for drawing is `Canvas`. We are using a façade class to get access to `Canvas`.

### NoteSheetCanvas

This is a simple façade class which gives access to all basic drawing operations like:

- `getWidth()`
- `getHeight()`
- `drawRect(Rect rect, Paint paint)`
- `drawBitmap(Resources resources, int bitmapId, int bitmapHeight, int xPosition, int yPosition)`

### Drawer classes

These classes use the `NoteSheetCanvas` to draw parts of different symbols or the note sheet / staff (consisting of 5 lines and some other elements). Here are some examples:

- `NoteSheetDrawer:` draws the five lines, the musical key and bars at the start and at the end of the five lines.
- `NoteBodyDrawer:` draws the oval parts of a `NoteSymbol`.
- `BreakDrawer:` draws `BreakSymbol`.
- `NoteSheetDrawPosition:` is used in all drawer classes to know the end of the canvas as well as the next draw position for a symbol.

### Drawer tests

To test all drawer classes in a short and easy way we had to create a small framework. `CanvasMock` extends the `Canvas` class and is used for all drawing while in a testing environment. All used methods are overridden by the `CanvasMock`. Instead of real drawing all methods log their actions in a queue of strings. This way testing drawn elements comes down to checking the content of the queue.

The `AbstractDrawerTest` class was created to make testing all drawer classes as simple as possible. It offers some important features:

- `setUp()` to initialize basic components a drawer needs (**Note:** always call `super.setUp()` if you override this method).
- `tearDown()` to check if the queue of drawn elements is empty after a test (postcondition). This way we can be sure that nothing else gets drawn without it being taken care of in a test (**Note:** always call `super.tearDown()` if you override this method).
- A custom assert method which checks the size of the string queue
- Several custom assert methods which check the first element in the queue. **Example**:
    - `assertCanvasElementQueueBitmap(int bitmapId, int bitmapHeight, int xPosition, int yPosition)` checks if the top of

the queue is a bitmap drawing with the given parameters like position or height. It will also remove the top of the queue.

**Example:**

Let's say we have a drawer which paints two rectangles at the screen at different x and y coordinates:

- startX: 50, startY: 100, stopX: 150, stopY: 200
- startX: 350, startY: 400, stopX: 450, stopY: 500

In a testing environment the `CanvasMock` string queue would look like this:

- "DRAW_RECT 50 100 150 200"
- "DRAW_RECT 350 400 450 500"

A test case for such a drawer would boil down to reproduce these strings by calculating the x and y positions as well:

```
assertCanvasElementQueueRect(50, 100, 150, 200)
assertCanvasElementQueueRect(350, 400, 450, 500)
assertCanvasElementQueueSize(0) // done by teardown() as a postcondition
```