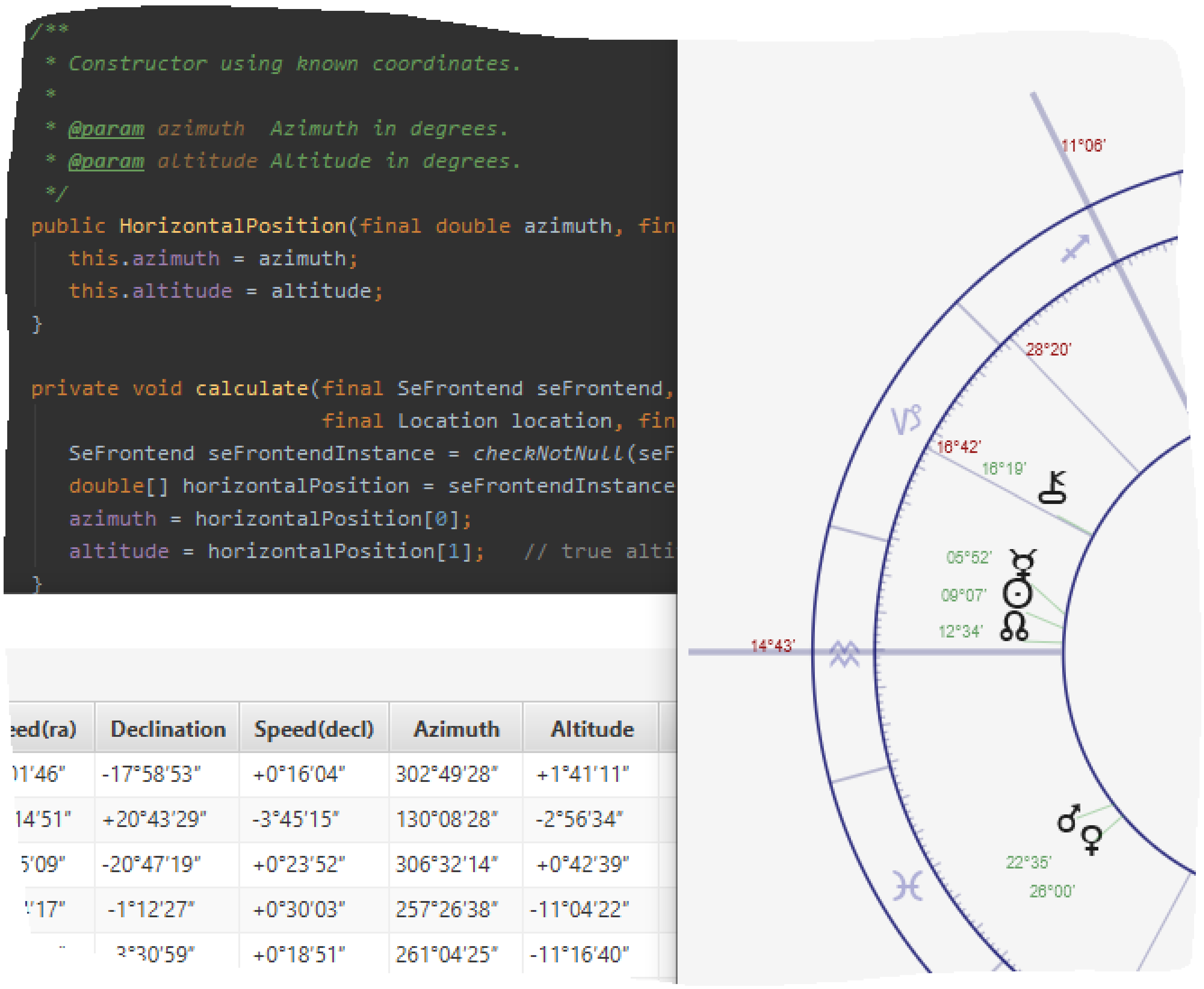
Enigma Programmers Manual

version 2020.1



```
/**
 * Constructor using known coordinates.
 *
 * @param azimuth  Azimuth in degrees.
 * @param altitude Altitude in degrees.
 */
public HorizontalPosition(final double azimuth, fin
   this.azimuth = azimuth;
   this.altitude = altitude;
}

private void calculate(final SeFrontend seFrontend,
                       final Location location, fin
   SeFrontend seFrontendInstance = checkNotNull(seF
   double[] horizontalPosition = seFrontendInstance
   azimuth = horizontalPosition[0];
   altitude = horizontalPosition[1];   // true alti
}
```

eed(ra)	Declination	Speed(decl)	Azimuth	Altitude
)1'46"	-17°58'53"	+0°16'04"	302°49'28"	+1°41'11"
14'51"	+20°43'29"	-3°45'15"	130°08'28"	-2°56'34"
5'09"	-20°47'19"	+0°23'52"	306°32'14"	+0°42'39"
'17"	-1°12'27"	+0°30'03"	257°26'38"	-11°04'22"
[illegible]	3°30'59"	+0°18'51"	261°04'25"	-11°16'40"

Content

Introduction

Enigma is open source. If you want to use (parts of) the code in your own projects, or if you are just curious how Enigma is implemented, this document will come to your aid.

You will find a new version of the documentation with each new release of Enigma.

Copyright

The code of Enigma itself is free to use. However, Enigma use libraries from other parties and of course these parties have their own conditions. For instance, if you want to write your own software and you want to include the Swiss Ephemeris, you will need to check the copyright details of the Swiss Ephemeris. You will find explicit copyright notes at the root of the project. All used libraries support free use in an open source environment but some of them require you to make your own code also open source.

Download

You can download the code from https://github.com/jankampherbeek/enigma

Prerequisites

I assume the reader has up-to-date knowledge of development in Java and basic knowledge of Java FX. If you do not have experience with Java, you could learn something by reading the code but you might have difficulties understanding what is really going on. I am sorry that I cannot elaborate on the use of Java. If you feel the need to learn more, you will easily find enough tutorials on the Internet.

Technical decisions

In Java you will almost always use several libraries and frameworks. If possible, I try to prevent using frameworks as they tend to add complexity and add a dependency on the framework. All frameworks and libraries are defined via pom.xml. An incomplete list of the most important items:

- **Java**, Open JDK version 14. Compiler and runtime environment.
- **Java FX**, version 14. Creates the components for the User Interface.
- **Maven**, version 3.6.1, a build and repository manager.
- **Junit**, version 4.12, for performing unit tests (and integration tests).
- **Mockito**, version 3.2.4, handles mock-objects in unit tests.
- **SwissEph Port** by Thomas Mack, version 2.01.00-02. The Java port of the Swiss Ephemeris.
- **SLF4J**, version 1.7.5. Supports logging.

Development environment

I developed Enigma as a Maven project in the community edition of IntelliJ IDEA. It should be possible to use the code in Eclipse or Netbeans but I will not be able to give the details for those environments.

Java 14

Enigma is built using Java 14 – at this moment the latest version – and it uses new techniques from this version. So make sure you have the JDK for Java 14 installed.

Additional settings

Maven will take care of most of the additional settings. You can check the details in the pom-file. There are some additional settings you have to take care of:

- Importing the Mack port.
- Downloading data files.
- Download JavaFX.
- Setting VM options in the IDE.

Using the Mack port to the Swiss Ephemeris

The Swiss Ephemeris is written in C which means it is not a natural companion for Java. However, Thomas Mack created an excellent Java port to the Swiss Ephemeris. This port is available as source and as a jar. The easiest solution is to import the jar into your project. However it is not available via a Maven repository. Therefore you have to import it yourself to your local Maven repo.
You can download the jar via http://www.th-mack.de/international/download/ . Select the "Ready to use archives". For Enigma I used the standard file (not the ME version).

Make sure that you have Maven installed: https://maven.apache.org/download.cgi

To import the file you need to open a shell (in Windows I used PowerShell) and move to the folder that contains the pom.xml of the Enigma project.
Run the following command:

```
mvn install:install-file -Dfile=C:\dev\enigma\mack\swisseph-2.01.00-02.jar
-DgroupId=de -DartifactId=seport -Dversion='2.01.00-02' -Dpackaging=jar
-DgeneratePom=true
```

- Replace `C:\dev\enigma\mack\swisseph-2.01.00-02.jar` with the full path to the jar file you downloaded.
- Replace `'2.01.00-02'` with the version you actually downloaded. Do not omit the single quotes, otherwise Maven will get confused by the division in the version number.
- Please note that this is a one-line statement.

Installing data files

For the calculations, Enigma uses a large set of data files. These files are supplied by the Swiss Ephemeris. Just install the executable version and you can copy the files from there.
In the current setup Enigma expects the datafiles to be in a folder *\enigma-data\se* on the C-drive.

Download and install Java FX

Surf to https://gluonhq.com/products/javafx/ and download and install the JavaFX Windows x64 SDK 14.01 (or higher) and JavaFX Windows x64 jmods 14.0.1 (or higher).
To install, it is sufficient to unpack the downloads each to a location of your liking.

VM options variables in the IDE

Make sure you add the following line to the VM options of the Run/Debug configuration. In Enigma this is the configuration to start the class ‘App’.

```
--module-path "C:\Program Files\java\javafx-sdk-14.0.1\lib"
--add-modules javafx.controls,javafx.fxml,javafx.web,javafx.graphics
```

- Replace `"C:\Program Files\java\javafx-sdk-14.0.1\lib"` with the location for the Java FX sdk. The double quotes are only required if you have a space in your path.
- Please note that this is a one-line statement.

Deployment

Enigma uses the tool *jPackage* from JDK 14 to create an installable msi package for Windows. This msi installs a Windows executable including a JRE; the user does not have to install a separate Java environment.
The approach I use is a slightly adapted copy of the *JPackageScriptFX* approach advocated by Dirk Lemmermann: https://github.com/dlemmermann/JPackageScriptFX

Maven

In pom.xml, a profile with the id build-windows, is used which only works in a Windows environment. If you work on Linux or Mac you have to change this. A batch file *build_projectE.bat* is called and takes over after the preparation by Maven. If you do intend to use the code onLinux or a Apple, you need to replace the hardcoded path to c:\enigma-data in the source. (This path will become a property in the next release).

Batchfile

The batchfile creates the msi file in the folder target/installer . The batchfile itself is well documented by Dirk Lemmermann. Rename the created file to *Enigma-2020.1.0.msi* .

Additional

Enigma requires several additional items. I use a batchfile *install.bat* that prepares these items and then starts the msi installer.

The batchfile creates the following directory structure and content:
c:\enigma-data

\db	Location for the database files.
\log	Location where log-files will be placed.
\se	Datafiles for the Swiss Ephemeris.

Font

You need to install the font EnigmaAstrology manually. Make sure that you install it for all users. See also the *User Manual*.

Architecture

Mixed approaches

I have been experimenting with different approaches and you will see the effect of this in the code: it is not as consistent as it should be. However, I believe a general approach is now pretty much clear to me. In the following releases I will refactor the code. Part of it will already be done in release 2020.2 but I expect to need more releases before all code is in the final form. Final for now, that is :-) .
I intend to use the following approach:

- Dependency injection. Currently sparsely done with some factories. I will probably use Dagger 2 or alternatively implement the DIY-DI approach as proposed by Chad Parry. See: https://blacksheep.parry.org/wp-content/uploads/2010/03/DIY-DI.pdf . In both approaches, DI is realized at run time.
- Defensive coding. I will use Pre and Post Conditions and enforce them with checks from Guava. Currently this is only done for null-checks.
- Increase test coverage. Partly due to the experimenting the coverage needs improving. It should be above the 80% threshold. The use of a simple DI approach as described above will be helpful.
- The UI does not use fxml (and therefore no SceneBuilder), everything is written in Java. It might be a matter of personal preferences but I think using plain Java gives the best overview and the best control. This is already realised in the current release but several classes still need to be refactored to a smaller main class and several additional classes.

Setup

Enigma is divided in three main parts.

1. **UI**: In the package *com.radixpro.enigma.ui* you will find anything related to the User Interface. All JavaFX related code and several additional classes.
2. **Backend**: The package *com.radixpro.enigma.be* contains all classes for calculations, comparisons etc.
3. **Exchange**: In *com.radixpro.enigma.xchg* everything is located that is used by the UI ànd by the Backend. Mainly the api that makes interconnections possible and the domain which is shared between UI and Backend.

Additionally, there is a package *com.radixpro.enigma.shared* that mainly contains utilities and helper classes that can be used by any class.
The UI and Backend are logically separated . There should be no direct connection between these two parts, all sharing should be done via the Exchange.
This separation is not technically forced but is maintained in the code.

Dependency injection

Only constructor injection is used. I have been experimenting with several solutions and you will find some factories in the code that have been used to realize this. For release 2020.2 I hope to have a definitive solution. The possibilities I consider are Dagger 2 and DIY/DI.

Design by contract

I used Google Guava to check for null-values and I will extend the checks on incoming parameters in following releases. The basic idea is using pre- and post-conditions.

Persistence

For saving data, like calculated charts and configurations, Enigma uses simple csv-files. This has been a last-minute change. I initially used Nitrite, a JSON based document database. It appeared that the database could not constintently handle the change of executables, or that I did not understand how to use Nitrite. Anyhow, it raised doubt in how far it would be usable for updates in the database and application. Therefore I re-implemented the DAO’s based on data in csv-files. Probably I will switch back to JSON but that will also be file based.

Testing

Unit testing

Standard unit testing is realized with jUnit 4 and, when required, Mockito.