The design and implementation of a userfriendly object-oriented markup language

Titouan Vervack

Supervisor: Prof. Marko van Dooren
Counsellor: Benoit Desouter

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Vakgroep Toegepaste Wiskunde, Informatica en Statistiek
Chair: Prof. dr. Willy Govaerts
Faculty of Engineering and Architecture
Academic year 2015-2016

UNIVERSITEIT GENT

The design and implementation of a userfriendly object-oriented markup language

Titouan Vervack

Supervisor: Prof. Marko van Dooren
Counsellor: Benoit Desouter

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Vakgroep Toegepaste Wiskunde, Informatica en Statistiek
Chair: Prof. dr. Willy Govaerts
Faculty of Engineering and Architecture
Academic year 2015-2016

UNIVERSITEIT GENT

Preface and acknowledgement

Permission for usage

Overview

Contents

Chapter 1

Introduction

In this thesis we will be introducing a new markup language that tries to improve upon a few others, namely LaTeX and Markdown, whilst still holding on to their advantages. Before we get into the details of this new language it is necessary to introduce some of the currently most used markup languages and word processors. This introduction will provide the context from which the necessity of this thesis was sparked.

1.1 State of the art

1.1.1 Markdown

One of the most popular markup languages at this point in time is Markdown [10]. It is introduced by the designers of the language as follows: [10]

```
Markdown is a text-to-HTML conversion tool for web writers. Markdown allows you
to write using an easy-to-read, easy-to-write plain text format, then convert it
to structurally valid XHTML (or HTML).
```

Markdown's goal is to be easy to write and read as a plain text document and as such is actually based on plain text emails. Due to this, it attracts a lot of people's attention as it is very easy to create a new simple document, such as a short report. It achieves this by introducing a small and simple syntax [11]. The syntax they chose also feels very natural. For example, to create a title you could underline it with - or = to create an enumeration, you can use *'s instead of bullet points.

By using such a simple syntax compile times for the documents are very short and IDE support is wide. Even without the help of an IDE, Markdown is very readable as it was designed to be. Another advantage of this simple syntax is that it almost entirely removes the possibility of syntax errors to occur.

Due to it's simple nature and limited syntax Markdown can be easily translated into other formats such as PDF or HTML. A tool to convert Markdown to HTML is offered by the designers of the language itself, as stated above. This further increases the appeal of the language, as it allows to very easily create elegant looking documents in a variety of formats.

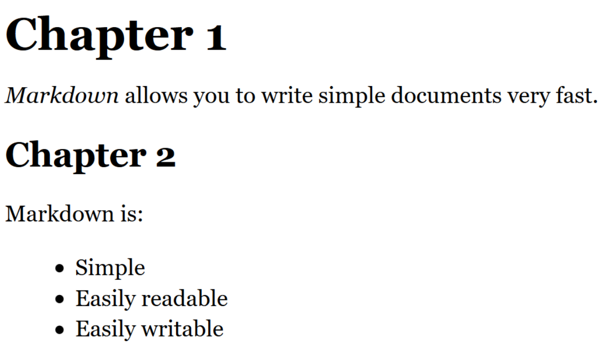
As Markdown files are just plain text files they are robust to file corruption and easy to recover from said corruption. A corrupted byte would show easily and it would likely not destroy the layout of the document as there are so few reserved symbols being used in a Markdown document.

Due to the fact that Markdown uses plain text, it is cross-platform and because it has such a simple syntax, compilers are available for all major operating systems. As a matter of fact, the Perl script offered by the Markdown designers, is cross-platform. Markdown also natively allows for UTF-8 characters to be used in the text thus not requiring weird escape sequences to insert certain symbols. A last advantage of plain text is that it is well suited for Version Control Systems (VCS) such as Git and Mercury.

To illustrate the simplicity of Markdown an example document is shown below:

Listing 1.1: document.md

```
# Chapter 1
*Markdown* allows you to write simple documents very fast.

## Chapter 2
Markdown is:

* Simple
* Easily readable
* Easily writable
```



Chapter 1

Markdown allows you to write simple documents very fast.

Chapter 2

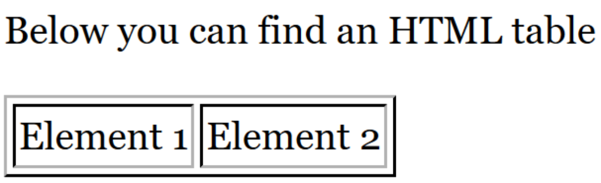
Markdown is:

- Simple
- Easily readable
- Easily writable

Figure 1.1: The document rendered as HTML by the official markdown conversion tool

To allow for some customisability, Markdown allows you to use inline HTML. In case Markdown does not support what you want to do, HTML is what you use. Tables are not a part of default Markdown (though there are variations that have syntax for them, like Github Markdown), so an example of how to create one using HTML is shown below:

```
Below you can find an HTML table
<table border="1">
    <tr>
        <td>
            Element 1
        </td>
        <td>
            Element 2
         </td>
     </tr>
</table>
```



Below you can find an HTML table

Element 1	Element 2

Figure 1.2: The table document rendered as HTML by the official markdown conversion tool

1.1.2 LaTeX

LaTeX provides a, Turing complete, programming model and due to this allows for rich customisability. It also provides a packaging system that allows for libraries, called packages, to be created using this programming model. Over the course of the past 38 years (the initial TeX release was in 1978) a lot of packages have been created for all kinds of different functionality. This is one of the reasons why LaTeX is often used for long and complex documents such as books, articles, scientific papers and syllabi.

Like Markdown lat uses a plain text format that is quite robust to file corruptions and is well suited for VCS. It is however less robust against file corruption as the syntax is a lot more extensive and thus file corruption could lead to your document not compiling any more a lot easier. It also doesn't use UTF-8 making it harder to type more exotic characters such as . Later iterations of TeX, such as XeTex and LuaTex do support UTF-8 though.

In LaTeX the structure and the layout of your document are split by design. LaTeX forces you to create a structured document by requiring you to use `sections, paragraphs,...`. When using a WYSIWYG word processor for example, usually there is a way to structure your documents, using `styles, headers,...` but it is not forced on the user. As a result, in LaTeX consistency is enforced throughout the entire document without too much effort, while in aforementioned WYSIWYG word processors this is often forgotten and less convenient to use. In a WYSIWG word processor, the user thus has to make sure his document is consistent. Once documents get longer, this gets very hard to manage and to spot the inconsistencies.

Next to this customisability, LaTeX's typography is also very sophisticated. It has support for kerning and ligatures and uses an advanced line-breaking algorithm, the Knuth-Plass line-breaking algorithm [8]. The algorithm sees a paragraph as a whole instead of using a more naive approach and seeing each line individually. Kerning places letters closer together or further away depending on character combinations. A ligature is when multiple characters are joined into one glyph e.g fi vs. `fi`.

Another reason why LaTeX is widely used due to its good scientific support. LaTeX and its libraries offer good support for mathematics as well as other scientific areas, e.g. you can create complex good looking graphs using LaTeX. Due to its high customisability, LaTeX can handle about any kind of document you could think of, from sheet music and graphs, to letters and books. It should be noted though that almost all of this functionality is offered through libraries such as PGF/TikZ [17].

LaTeX also works together with BibTeX, a reference management software that is very robust and makes certain all of your references are consistent. It also allows you to create the references outside of your main document, decreasing the amount of clutter in the document. Whilst on the topic of including other files in your document, in LaTeX you can split up your document into multiple smaller documents, allowing for easier management. A common use of this function is to write every chapter, e.g. in a book, in a separate document and than all the chapters are imported into the main document. This allows to easily remove or switch out different parts of a document.

Another very simple feature that LaTeX has, is that is allows you to use vector graphics or PDF as images in your document. This is something not possible in WYSIWIG word processors or lightweight markup languages such as Markdown.

LaTeX is free and open source, which is a huge point of attraction for the open source community. As a result of this free and open source model, combined with its popularity and good scientific support, LaTeX has been integrated in many other applications. A few examples are:

- MediaWiki [12]
- Stack Exchange
- JMathTex [6]

1.1.3 Pandoc

Pandoc [16] is a document converter, but in reality it can do much more than just convert a document from one format to another. Notably, it allows you to write inline LaTeX inside of a Markdown document allowing for a much better experience than just using one of them. However, Pandoc does not really have a programming model, outside of the LaTeX one, it works with so called filters. When a document is issued for conversion, Pandoc will transform the document into an Abstract Syntax Tree. This AST can then be transformed using filters, the AST enters a filter, is transformed and is then past on to the next filter. Filters can be written in a multitude of languages such as Haskell, Python, Perl,... and the filters to be used are passed as a command line argument to the conversion command.

1.1.4 Word processors

Modern word processors such as Microsoft Word and Pages are so called WYSIWYG editors. As you immediately, without compiling, see what your document will look like it is a very popular tool. The fact that you immediately see what you are writing also significantly decreases the complexity of the tool and it does not scare away less technical people the way LaTeX does.

There is also no possibility to use commands as in LaTeX or shorthand syntax as in Markdown, instead buttons in the GUI offer all of the functionality. Structure is not forced onto the user which offers less restrictions allowing the user to do what he wants. It is also very easy to change a font type or change the font size of a certain part of the text. Another feature that is loved in the WYSIWIG community is that you can easily add an image and drag and drop it around.

1.2 Problem statement

We can see that these solutions all have there strong points but there are also quite a few gripes. We will discuss the most common gripes for every solution underneath and afterwards introduce the problem statement.

1.2.1 Markdown

To achieve Markdown's simplicity, a hefty price has to be paid. All of the syntax elements have been hard coded into the language meaning there is no room for customisability. The only way to customise Markdown is by embedding inline HTML. HTML allows users to create a solid structured model but it still does not allow for very much customisability. While Markdown is meant to be easily readable and writeable, HTML is not and using it thus decreases readability and cleanliness of your document significantly. Once you use HTML in an Markdown document, you also require a certain technical proficiency for the source document to be read or edited. HTML also does away with the robustness of Markdown's plain text format, as having a tag being corrupted could lead to very big changes in the rendered file and might not even allow for compilation anymore. HTML also reintroduces the possibility of syntax errors and thus slows down the document creation.

The lack of customisability in Markdown is further enforced due to the fact that it has no programming model. It is one thing that the syntactic shorthands can not be reused, but there is also no other way, next to HTML to create an entirely new entity.

1.2.2 LaTeX

Even though LaTeX offers customisability through a programming model, it is not perfect. The programming model is hard to use and is not easy to read. Due to this programming model it also looks a lot more complex than Markdown, scaring away a lot of potential users. There are a few ways to counter this complex look though, for one there is LyX. This is a GUI around LaTeX offering some of the most common WYSIWG features, while still utilizing LaTeX's strength. Even though the complexity is now hidden under a GUI, it is still there and error messages are as bad as they were before, as is discussed beneath.

Having a cumbersome programming model, LaTeX makes it easy to create syntax errors. The robustness of a plain text document is also diminished, as your document will probably not even compile once an error occurs in one of the documents. Something that is also very common in LaTeX is that the shown error messages do not offer correct information and are often unclear. See the excerpt below for an example of misleading error messages.

```
\documentclass{article}
\begin{document}
\begin{equation}
$a$
\end{equation}
\end{document}
```

```
line 4: Display math should end with $$
    .<to be read again>a $a
line 5: You can't use `\eqno' in math
    mode.\endequation ->\eqno\hbox {\
    @eqnnum }$$\@ignoretrue \end{
    equation}
line 6: Missing $ inserted.<inserted
    text>$ \end{equation}
```

What it is actually trying to tell you is that you are not allowed to use $ inside of an equation, but that is not clear from the error messages.

The programming model used in LaTeX does not use a static type system, which means that we have to compile the document to be able to see if we passed a wrong type to a macro at some point in the document. It also does not work the way we are used to in imperative and functional languages. It does not have functions that receive parameters and that then call more functions. Instead it has macro's and uses macroexpansion. As an example of how this translates into a real world example, a function the implementation of **foreach** in PGF/TikZ is shown below.

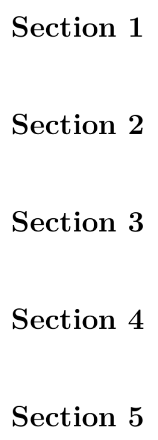
```
\def\pgffor@foreach{%
\pgffor@atbeginforeach%
\let\pgffor@assign@before@code=\pgfutil@empty%
\let\pgffor@assign@after@code=\pgfutil@empty%
\let\pgffor@assign@once@code=\pgfutil@empty%
\let\pgffor@remember@code=\pgfutil@empty%
\let\pgffor@remember@once@code=\pgfutil@empty%
\pgffor@alphabeticsequencefalse%
\pgffor@contextfalse%
%
\let\pgffor@var=\pgfutil@empty
%
\pgffor@vars%
}
```

Not only is this nearly unreadable and very complicated for such a common concept in programming, it also uses a ton of other functions defined in PGF/TikZ just to be able to define a foreach loop.

Now we can include the `pgffor` package and apply this function. Luckily it is somewhat easier to use, than to create, as is shown below.

Listing 1.2: loopfunction.tex

```
\def\loopfunction#1{
  \foreach \index in {1, ..., #1} {
    \section*{Section \index}
  }
}
```


Section 1

Section 2

Section 3

Section 4

Section 5

Figure 1.3: The document produced from calling `loopfunction` with parameter `1`

The macro, called `loopfunction`, loops from 1 to a parameter #1 and prints out a section in every iteration. Calling it using the following code `\loopfunction5` results into the document shown in Figure 1.3.

1.2.3 Pandoc

Transforming an AST afterwards is not always as easy as doing so inline, and it also hides what really happens to the document, which could cause confusion.

1.2.4 Word processors

A first point to note is that the most popular word processors such as Microsoft Word and Pages are not free, but instead cost a significant amount. There are free variants, such as LibreOffice and OpenOffice, but the compatibility between these word processors is not perfect. The free alternatives usually also have less of a focus on the UI, making them look somewhat less sleek.

Another problem with word processors is that their format often changes over the years. This means that your document might not show up the same, or at all, a few years after you wrote it.

Even though some customisability is available, no programming model is available and real customisability is far to be found. You are stuck to what the developers of editor defined and implemented. There is no way to go beyond what the designers specified. Due to this it is not always possible to do exactly what you want either. When you have a few elements in your document and you try to align a new element with it, it will snap to all sorts of guidelines, but chances are you want to place it in between all of the guidelines, which is not possible at that moment. Due to their lack of a programming model, word processors do not allow for even the simplest of computations such as `The result is x + y`.

It is also quite easy to create inconsistencies using word processors as noted above in Subsection 1.1.2. Next to creating inconsistencies, as structure is not enforced on you, small changes could have big side effects on the document. E.g. you placed an image exactly like you wanted, you then later on add a newline somewhere higher up, reflowing the text underneath it an shifting the precisely placed image around. These changes are not always immediately clear, which further contributes to the creation of inconsistencies in the document.

Lastly, word processors often use a proprietary or complex format for there documents, which is not only bigger than a plain text document, but also a lot more prune to corruption. A single bit flip could potentially irreversibly destroy your document. As complex or binary document formats are used, it is hard to use WYSIWIG editors in combination with VCS. This makes it inconvenient to track changes made to the document and go back to an earlier version of the document. It also makes it harder to work on a document as a team, you might be making changes at the same time as someone else, but you have no way of automatically merging both documents creating a possibility of data to get lost.

1.3 Proposed solution: Neio

To counter the problems occurring in the state of the art solutions, we bring forth a new markup language called `Neio` (read as neo). To be able to solve some of the aforementioned problems and still make use of all of their advantages, the following goals were devised for the Neio markup language:

1. User-friendliness: has to be easy to get started with the language
2. Has to use a modern programming paradigm
3. Has to be highly customisable

The first goal supplies us with a simple language that allows to very easily create simple documents. The use of a modern programming paradigm allows us to customise the document and execute easy and complex computations easily in effect achieving the third goal.

To achieve this we chose to use a syntax like Markdown to write our documents in as it is the easiest to read and write. For the programming paradigm we chose for something much like Java as Java is very well known, and working with macro expansion's or filters is not ideal. Lastly, we choose to translate to LaTeX as it has been successively used for decades and has proven by now that it can deliver very clean looking documents. Later on more back ends, such as HTML, could be implemented, but that is outside of scope of this thesis.

1.3.1 Target group

Neio targets anyone that is currently using Markdown but just wants to be able to customise their document a bit more. As we will see later, in Neio it is easy to for example create two images and put them next to each other, something that is not possible in markdown.

It can also be used by people without technical expertise if templates are provided. E.g. given a template for a letter, even a non-technical person is able to quickly write down the contents of the letter and fill in the template variables.

Neio also target package developers and programmers in general as it offers a modern programming model that is a lot well known and much easier than TeX's system. This includes full-time developers as well as students or professors.

Chapter 2

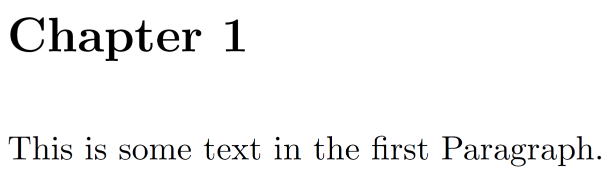
Design of the Neio markup language

In the previous chapter, Chapter 1, the state of the art concerning document creation has been discussed and we notice that some improvement is certainly possible. All of the document creation tools discussed above have their advantages and disadvantages. In this chapter we will present the Neio language and we will explain how our decisions were reached and how they were affected by the state of the art solutions.

2.1 Neio document

To illustrate some of the basic concepts we will now present an example. The typical files that a user will write are called Neio documents. Neio documents are heavily based on Markdown as Markdown is so easy to read and write. Since this document is what most of our consumers will be seeing, this was a very important property. Since our consumers will mostly write Neio documents, we are also able to use the widespread knowledge of Markdown to increase the size of our audience and allow users to instantly transition to Neio. The following example is one of the most basic documents you can write using Neio.

```
[Document]

# Chapter 1
This is some text in the first Paragraph.
```



Chapter 1

This is some text in the first Paragraph.

Figure 2.1: The rendered document

Even if you are not familiar with Markdown, you can probably immediately tell what this document represents. There is however a very significant difference with Markdown. In markdown the syntax is hard-coded into the language, a # will always represent a chapter for example. In Neio, on the other hand, the language has no knowledge whatsoever of what # means. In fact, it does not know what anything in a Neio document means. It does not know about these things because everything is a method call, chained together to form an object model that represents the document. The reason for doing so is simple, we want to be able to customise our documents! For example, when you are creating a slide show you might not want # to create a new chapter, instead it would be better and clearer if it would just create a new slide.

Knowing this we can now represent the document as a chain of method calls as follows:

```
new Document()
    .#("Chapter 1")
    .text("This is some text in the first Paragraph");
```

For this to work of course, every one of these method calls has to return an object. In fact, every one of these methods will return an object that is a subtype of the `Content` class, anything in a Neio document that can be shown is a subtype of `Content`. The `Content` class is the base class in Neio documents, like `Object` is the base class in Java.

Now is also a good time to explain a small syntactical difference with Markdown, which you might have already noticed. Every Neio document has to begin with a so-called document class. This is a concept borrowed from LaTeX, and just as in LaTeX it tells us what kind of document the user is trying to build. We say that a Neio document has to start with a document class, but this is not entirely correct. You are allowed to add single- or multi-line Java style comments before the document, or anywhere else in the document for a matter of fact. The following is thus also a valid document.

```
// A single-line comment
[Document]
/*
 * Multi-line comments are also available!
 */
```

To further understand this example we need to introduce the second kind of files available in the Neio markup language, Class files.

2.2 Class files

Class files are very similar to class files in Java. They are the thriving force behind the Neio documents, they define all the object and methods that are used in a Neio document. They are extremely important to the design of the language as they allow us to represent everything in the document with an object structure. The use of this object structure is extensively used in Neio and examples of this will be shown further on.

As means of an introduction we will have a look at the `Document` class that was used as a document class in the previous section.

Listing 2.1: Document.no

```
namespace neio.stdlib;

import neio.lang.Text;

/**
 * Represents the most basic of Documents
 */
class Document extends TextContainer;

Packages packages;
Bibtex bibtex;
String extraPreamble;

/**
 * Creates a Document and adds common LaTeX packages to it
 */
Document() {
    packages = new Packages();
    packages.add("a4wide")
    .add("graphicx")
    .add("amsmath")
    .add("float");

    bibtex = null;
    extraPreamble = "";
}

/**
 * Creates a Chapter and adds it to this
 *
 * @param title The title to use for the new Chapter
 * @return         The newly created Chapter
 */
Chapter #(Text title) {
    Chapter chapter = new Chapter(title, 1);
    addContent(chapter);

    return chapter;
}

/**
 * Returns a list of Package's
 *
 * @return a list of used LaTeX packages
 */
Packages packages() {
    return packages;
}

/**
 * Adds a LaTeX package
 *
 * @param pkg The package to add
 * @return        The list of currently included packages
 */
Packages addPackage(String pkg) {
    return packages.add(pkg);
}

/**
 * Sets a BibTeX file for this Document
 *
 * @param The name of the BibTeX file
 */
void addBibtex(String name) {
    this.bibtex = new Bibtex(name);
}

/**
 * A proxy method for {@code BibTeX}'s {@code cite}
 * Returns a Cite that is linked to {@code key}
 *
 * @param key The key to access the citation
 * @return        The Cite corresponding to {@code key}
 */
Citation cite(String key) {
    if (bibtex != null) {
        return bibtex.cite(key);
    } else {
        return null;
    }
}

/**
 * Creates the LaTeX preamble
 *
 * @return The LaTeX preamble
 */
String preamble() {
    StringBuilder tex = new StringBuilder("\\documentclass{article}\n");
    for (int i = 0; i < packages.size(); i = i + 1) {
        tex.append(packages.get(i).toTex()).append("\n");
    }

    tex.append("\\setlength{\\parindent}{0em}\n")
         .append("\\setlength{\\parskip}{1em}\n")
         .append(extraPreamble);

     return tex.toString();
}

```

```
/**
 * Adds a newpage to the Document
 */
void newpage() {
    addContent(new NewPage());
}

/**
 * Adds a string to the preamble, checking if it is not yet
 * included already.
 *
 * @param s The String to add to the preamble
 */
void addToPreamble(String s) {
    if (!preamble().contains(s)) {
        extraPreamble = extraPreamble + s;
    }
}

/**
 * Creates a TeX representation of this object and its children
 *
 * @return The TeX representation of this object and its children
 */
String toTex() {
    StringBuilder result = new StringBuilder(preamble()).append("\\begin{
        document}\n")
    .append(super.toTex());
    if (bibtex != null) {
        result.append(bibtex.toTex());
    }

    result.append("\n\\end{document}\n");

    return result.toString();
}
```

Except for some variations on the syntax, such as using `namespace` instead of `package`, and the lack of access level modifiers, this is valid Java code. The access level modifiers are automatically set to `private` for members and `public` for methods. They do not have to be written explicitly as the default value is usually what the developer wants and it makes the class file just a little clearer and more concise.

Again though, I stand corrected, this would be valid Java code if not for the # method. In Neio class files, you are allowed to use symbols as a method name. This is a feature that is necessary to allow a Neio document to be represented as a chain of method calls as otherwise you would have to actually write out a document as series of method calls, or you would not be able to use symbols as syntactic shorthand. All the symbols you can use for method names at this time are `#`, `-`, `*`, `_`, `$`, `|`, `=`, `^`, the backtick and the newline. The newline character is recognised by the following regex `\n\r?`. By using all of these symbols we are able to create Domain Specific Languages (DSL) for some things such as tables or structural formulas. This will be illustrated in Chapter 3.

To keep the language simple and to maximize reusability (as well as making the parsing somewhat easier) some functionality from the Java language was dropped. It is for example not possible to create anonymous classes as this goes against the concept of reusability. It is also not possible to create multiple classes in a single file.

2.3 Newlines

Neio is newline sensitive, in the sense that newlines too are methods. The superclass `TextContainer` of `Document` and `Chapter` defines the following method.

```
/**
 * Handles newlines
 *
 * @return returns a new ContainerNLHandler
 */
ContainerNLHandler newline() {
    return new ContainerNLHandler(this);
}
```

The `ContainerNLHandler` will then further handle the flow of the document. The code for `ContainerNLHandler` and `NLHandler` are added below.

Listing 2.2: ContainerNLHandler.no

```
namespace neio.stdlib;

import neio.lang.Content;
import neio.lang.Text;

/**
 * Handles newlines that follow a TextContainer
 */
class ContainerNLHandler extends NLHandler;

/**
 * Creates a ContainerNLHandler and sets the TextContainer that created it as
     parent
 */
ContainerNLHandler(TextContainer parent) {
    super(parent);
}

/**
 * Returns this
 *
 * @return this
 */
ContainerNLHandler newline() {
    return this;
}
```

Listing 2.3: NLHandler.no

```
namespace neio.stdlib;

import neio.lang.Content;
import neio.lang.Text;

/**
 * The default newline handler
 */
class NLHandler;

Content parent;

/**
 * Creates a new NLHandler and keeps a reference to the object that created it
 */
NLHandler(Content parent) {
    this.parent = parent;
}

/**
 * Returns the object that created this
 *
 * @return The object that created this
 */
Content parent() {
    return parent;
}

/**
 * Creates a Paragraph and adds it to the parent.
 *
 * @param text The text for the Paragraph
 * @return         The newly created Paragraph
 */
Paragraph text(Text text) {
    new Paragraph(text) par;
    parent().addContent(par);
    return par;
}

/**
 * Returns this
 *
 * @return this
 */
NLHandler newline() {
    return this;
}
```

Our earlier example now translates to the following:

```
new Document()
    .newline()
    .newline()
    .#("Chapter 1")
    .newline()
    .text("This is some text in the first Paragraph");
```

We needed to do this because a newline in a plain text document has meaning, depending on the context it can mean something else entirely. For example, a single newline in a paragraph should append the next sentence to the paragraph. But on the other hand when we separate two text blocks with an empty line (thus two newlines), the blocks of text are seen as two different paragraphs.

It should also be noted that any text written in a Neio document gets translated to the `text` call as seen in the example above. An implementation of the text method is seen `NLHandler`. The reason for this will become clear later on.

We will see more advanced examples using newline in Chapter 3 that will further expose its benefits. To avoid clutter, the `newline` method will not be shown in the further examples unless it is needed to understand the example.

2.4 Language features

2.4.1 Context types

As said before, everything is a method call, but to be able to just chain any method to any other method, regular methods do not suffice. We want to be able to call methods of different class files whilst constructing our document, but we do not want to specify what object we're calling the method on. This is something that should be clear from the current structure of the file, a previously created object should have a method with the correct signature.

To be able to call the right methods, we need to introduce something called ContextTypes. A ContextType combines an object, the actual type, with its context. We will illustrate this using an example.

```
[Document]
```

As we saw before this translates to the new call `new Document()`. This is actually a ContextType of the newly created `Document` and `null` as there is nothing else yet. We'll call this ContextType `ct1`.

Lets add a chapter to it, when there are two elements, there should be some kind of context.

```
[Document]

#Chapter 1
```

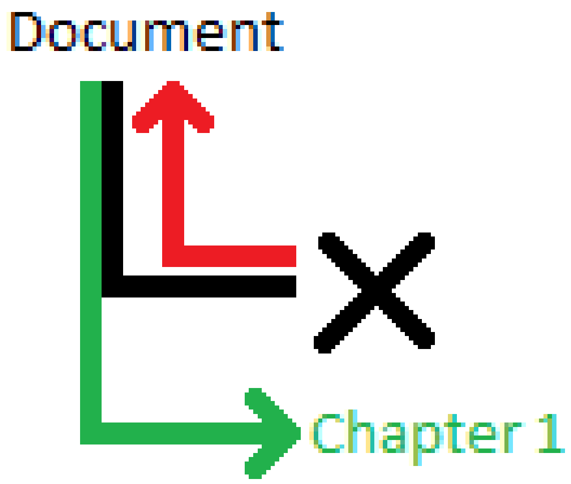


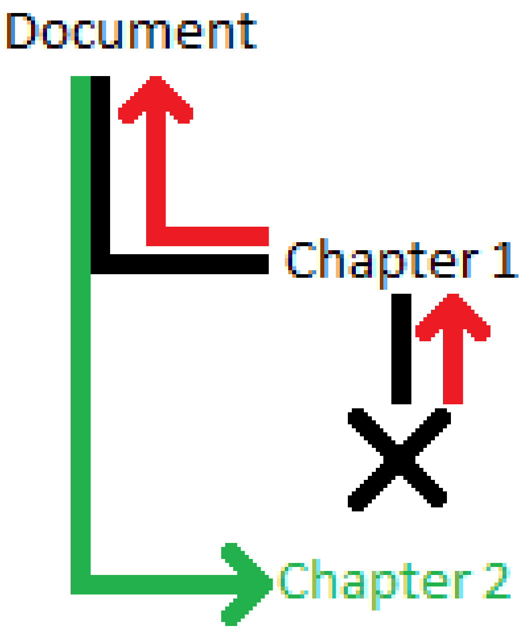
Figure 2.2: In the image above we find ourselves at the **X**. We follow the red arrow upwards and find ourselves in Document. As Document contains the # method, we call # on Document and a Chapter is added to the Document , as represented by the green arrow.

By adding this call our document translates to the following`new Document().#("Chapter 1")`. The addition of a second element allows us to see a ContextType in action.

To know where to find the # method, we will recursively iterate over our context. First we check the actual type of `ct1`, if we could not find the method there, we recursively proceed to check the context of `ct1`. As you can see above in the source code of `Document`, there is indeed a method # in there, so there is no need to look any further. The current document translates to ContextType `ct2`, which contains the newly created Chapter and has `ct1` as context.

We continue by adding one more Chapter.

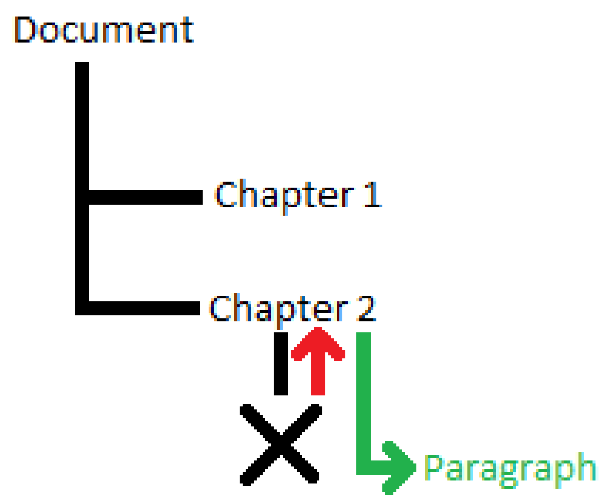
```
[Document]

#Chapter 1
#Chapter 2
```



Now when we look for the # method in `ct2` we first have to check Chapter, but Chapter does not have this method. We thus recursively check the context of `ct2`, which is `ct1`. `ct1`'s actual type is a Document, of which we know that it has a # method. As we have found the method we were looking for, # will be called on the Document in `ct1`. The ContextType that is created for this call is `ct3` that has the newly created Chapter as the actual type, but that has **`ct1`** as context. This is because we have had to skip over `Chapter 1, which shows us that ct2` is not useful anymore for the continued creation of the document.

Lastly we add a paragraph to the document.

```
[Document]

#Chapter 1
#Chapter 2
This is a paragraph.
```



We check the actual type of `ct3` and immediately have a matching method, we thus call the method on `Chapter 2`. The newly created ContextType `ct4`, contains the `Paragraph` as actual type and `ct3` as context.

The following structure is what we are left with.

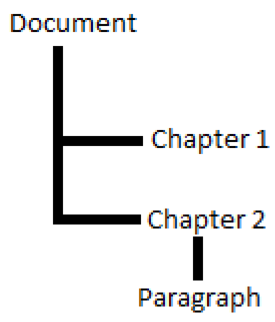


Figure 2.3: The final structure built in the example

2.4.2 Nested methods

Documents often have recursive elements such as sections or enumerations, as in Neio, everything is a method every one of these levels would have to be defined as a separate method. That means creating a method for #, ##, ###,... . This is of course very cumbersome, even impossible at times, and we would like to only define it once as the behaviour should be mostly the same no matter how many symbols we use. Usually only one property, such as font size, is effected by the number of symbols used in such a method call. This is why nested methods were invented. A method can be annotated with a `nested` modifier which means that this method implicitly takes an extra argument, an `Integer` that reflects the depth of this recursive method. The depth of this recursive method is just the number of times the symbol has been used. If we have a look at the `Chapter` class, we see that it has a nested method #.

Listing 2.4: Chapter.no

```
namespace neio.stdlib;

import neio.lang.Content;
import neio.lang.Referable;
import neio.lang.Reference;
import neio.lang.Text;

/**
 * Represents a chapter or a section
 */
class Chapter extends TextContainer implements Referable;

protected Text title;
protected String marker;
protected Integer level;
protected String texName;

/**
 * Creates a new Chapter
 *
 * @param title  The title of the Chapter
 * @param level The level of nesting of the Chapter
 */
Chapter(Text title, Integer level) {
    this.title = title;
    this.level = level;
    this.texName = "section*";
}

/**
 * Creates a new Chapter and adds it to this or
 * or to the nearest other TextContainer above this if
 * the level of the new Chapter and this are is lower or equal
 * to the level of this.
 * This is a nested call thus only ##+ will match
 *
 * @param title  The title of the new Chapter
 * @param level The level of nesting of the new Chapter
 */
nested Chapter #(Text title, Integer level) {
    if (level <= this.level) {
        Chapter c = nearestAncestor(Chapter.class);
        if (c != null) {
            return c.hash(title, level);
        } else {
            return nearestAncestor(Document.class).hash(title);
        }
    }
    Chapter chapter = new Chapter(title, level);
    addContent(chapter);

    return chapter;
}
```

If we would now call `chapter.##("Chapter 1.1")` for example, it would translate to the following call: `chapter.#("Chapter 1.1", 2)`.

An important thing to note is, that the regex matching a nested function, is `..+` where `.` is the symbol that is being used as a method name. This means that the method that contains a single symbol, the method that defines the first level e.g. `#(String name)`, should be defined elsewhere. This has been done because the first level is usually special. In case of an Enumerate for example, when we create the first EnumerateItem, not only an EnumerateItem but also an Enumerate has to be added. Usually the object created by this method should also be added in a different place than the other objects. Let us take the following example:

```
[Document]
# Chapter 1
## Chapter 1.1
# Chapter 2
```

We would want **Chapter 1** and **Chapter 2** to be children of Document, while **Chapter 1.1** should be a child of **Chapter 1**. The ContextType representation of the document is shown in Figure 2.4. If we now say that the nested modifier works on any level, including the first level, then the aforementioned structure would still be achieved, **but** the ContextType representation would be different. The ContextType representation of this document would be as shown in Figure 2.5. This is because `# Chapter 2` is now a method call on `ct1` instead of on `ct0` as the `nested Chapter #(String name, Integer level)` method in Chapter would be the function that resulted from our lookup.

2.4.3 Surround methods

A method could also be annotated with a `surround` modifier. This means that the method name surrounds a piece of text, this can be used to for example create bold text by surrounding it with stars. This is needed to be able to easily annotate text and it also looks very natural.

Examples of surround methods are the ones available in Text.no.

Listing 2.5: Text.no

```
namespace neio.lang;

/**
 * Represents the any kind of text that can be marked up
 */
class Text extends Content;

Text text;
String realText;
// The object to which this Text has been appended
Text appendedTo;

/**
 * Creates an empty Text
 */
Text() {
    appendedTo = null;
    text = null;
    realText = "";
}

/**
 * Creates a Text
 *
 * @param realText The content of the Text
 */
Text(String realText) {
    this();
    this.realText = realText;
}

/**
 * Creates a Text with another Text as content.
 * This allows to add more mark up to {@code text}
 *
 * @param text The Text to use as content
 */
Text(Text text) {
    this();
    this.text = text;
}

/**
 * Append Text to this
 *
 * @param t The Text to append
 * @return The appended Text
 */
Text text(Text t) {
    return appendText(t);
}

/**
 * Returns the Text being used as content
 *
 * @return The Text used as content
 */
Text getText() {
    return text;
}

/**
 * Returns the String used as content
 *
 * @return The String used as content
 */
String realText() {
    return realText;
}

/**
 * Appends the root Text of {@code t} to this
 *
 * @param t The Text to append
 * @return The appended Text
 */
Text appendText(Text t) {
    Text start = t;

    while (start.isAppended()) {
        start = start.appendedTo();
    }
    start.appendTo(this);

    return t;
}

/**
 * Sets the Text to which this is appended
 *
 * @param appendedTo The Text which we are appended to
 */
void appendTo(Text appendedTo) {
    this.appendedTo = appendedTo;
}

/**
 * Returns the Text to which this is appended
 *
 * @return The Text to which this is appended
 */
Text appendedTo() {
    return appendedTo;
}

/**
 * Returns if this Text is appended or not
 *
 * @return If this Text is appended or not
 */
Boolean isAppended() {
    return appendedTo != null;
}

/**
 * Creates the TeX representation for the Text this is appended to
 * Recursively calls {@code toTex} on the Text the current Text is appended to
 *
 * @return the TeX representation for the Text this is appended to
 */
String appendedToToTex() {
    String result = "";
    if (appendedTo != null) {
        result = appendedTo.toTex();
    }

    return result;
}

```

```
/**
* Creates a new {@code BoldText} and appends it to this
*
* @param t The text to make bold
* @return The newly created BoldText
*/
public surround Text *(Text t) {
    return appendText(new BoldText(t));
}

/**
* Creates a new {@code ItalicText} and appends it to this
*
* @param t The text to make italic
* @return The newly created ItalicText
*/
public surround Text _(Text t) {
    return appendText(new ItalicText(t));
}

/**
* Creates a new {@code MonospaceText} and appends it to this
*
* @param t The text to show in a monospace font
* @return The newly created MonospaceText
*/
public surround Text ‘(Text t) {
    return appendText(new MonospaceText(t));
}

/**
* Wraps {@code t} in a new Text and appends it to this
*
* @param t The text to wrap
* @return The wrapped Text
*/
public surround Text $(Text t) {
    return appendText(new Text(t));
}

/**
 * Creates the TeX representation of this Text
 * Printing out {@code realText} or the TeX representation of {@code text}
 * Replaces special characters to create valid TeX
 *
 * @return The TeX representation of this Text
 */
String thisToTex() {
    String me = "";
    if (text == null) {
        // Create valid TeX text
        me = realText.replaceAll("\\\\([^#$%&~\\^_{}\\\\])", "$1");
        me = me.replaceAll("\\\\~", "\\\\textasciitilde{}");
        me = me.replaceAll("\\\\^", "\\\\textasciicircum{}");
        me = me.replaceAll("\\\\\\\\\\\\", "\\\\textbackslash{}");
        me = me.replaceAll("\\\\\\\\", "\\\\textbackslash{}");
    } else {
        me = text.toTex();
    }

    return me;
}

/**
 * Returns the TeX representation of this chain of Texts
 * The chain being the Text this is appended to + this
 *
 * @return The TeX representation of this chain of Texts
 */
String toTex() {
    return appendedToToTex() + thisToTex();
}
```

The text `This is *bold* text` will then be transformed into the following chain `text("This is ").text(*("bold")).text(" text")`.

2.4.4 Code blocks

We've seen that we can use code in Neio class files and customise our document this way. But this doesn't allow us to do everything we want to. We don't want to create a new document class every time we want to do something special. It's also not possible to add anything to a document class that does not exist. Sometimes we might just want to change a property of some content, which is not possible as of yet. For this reason, it is possible to add code blocks in a Neio document. The three different kinds of code blocks are explained below. To avoid confusion, when you are writing text into a Neio document, we say that we are in Text mode. While when we are writing code in a Neio document, we say that we are in Code mode.

Non-scoped code

Now let us say that we want to create a `Document` that uses a special type of content that we have created. For example, we like Fibonacci so much, that we have created a special `Enumerate` that is numbered according to the Fibonacci sequence. The code for this new `Enumerate`, `FibEnumerate` is shown below. We also show the code for the `EnumerateItem` that is used in `FibEnumerate`, `FibItem`.

Listing 2.6: FibEnumerate.no

```
namespace neio.fib;

import neio.stdlib.*;
import neio.lang.Text;
import neio.lang.Content;

class FibEnumerate extends Enumerate;

Integer first = 1;
Integer second = 1;
Integer index = 0;

FibEnumerate() {
    addClassMapping(Enumerate.class, FibEnumerate.class);
    addClassMapping(EnumerateItem.class, FibItem.class);
}

protected String header() {
    String header = "\\begin{description}";
    if ((option != null) && (option.length() > 0)) {
        header = header + "[" + option + "]";
    }

    return header + "\n";
}

// Finalize already exists in java.lang.Object
protected String finalizer() {
    return "\\end{description}\n";
}

Integer getNumber() {
    index = index + 1;
    if (index <= 2) {
        return 1;
    }

    Integer old = first;
    first = calcNext();
    second = old;

    return first;
}

private Integer calcNext() {
    return first + second;
}
```

Listing 2.7: FibItem.no

```
namespace neio.fib;

import neio.stdlib.*;
import neio.lang.Text;
import neio.lang.Content;

class FibItem extends EnumerateItem;

FibItem(Text text, Integer level) {
    super(text, level);
}

FibItem(Text text, Enumerate fib, Integer level) {
    super(text, fib, level);
}

String toTex() {
    return super.toTex(level + "." + enumerate.getNumber());
}
```

The only real thing that changed in this code, compared to their super classes `Enumerate` and

`EnumerateItem`, is the code to translate the object, given in the `toTex()` method. There is however another odd line present in `FibEnumerate`'s constructor. It concerns the following line:

```
addClassMapping(EnumerateItem.class, FibItem.class);
```

For now, it is enough simply to know that this acts mostly as a factory. For this example it means that if `FibEnumerate` or one of it's children (`FibItem`'s for example) were to create an instance of the class `EnumerateItem`, that it will actually create a `FibItem`. The reason we say it, acts as a factory, is because we're asking for a new instance of some object to be created and what we receive is a new instance of that specific class, or one of its subclasses. How this works will be explained further on, in Chapter 4. The bottom line is that this line of code allows us to reuse the `-` method that has been defined in `Enumerate`, to create `FibItem`'s instead of having to redefine this method simply to create an instance of a subclass. It also makes sure that we do not have to write a new factory every time we want to replace a general class by a more specific subclass, as would otherwise have been needed here.

Now to be able to get this `FibEnumerate` into a document all we have to do is open a code block and create a new `FibEnumerate`. A non-scoped code block is represented as a pair of {}, where the { is at the start of a line and the } is followed by a newline. The { and } can not appear on the same line. The reason for this will become clear further on. Also note that a semicolon at the end of the last statement is optional, this is mostly useful in inline code blocks as we will see further on, but even in this case, it makes the document just a little bit easier and less prune to syntax errors.

Listing 2.8: fibDocument.no

```
[Document]
{
    new FibEnumerate()
}
- Item 1
- Item 2
- Item 3
- Item 4
-- Item 5
-- Item 6
--- Item 7
```

1.1 Item 1

1.1 Item 2

1.2 Item 3

1.3 Item 4

  2.1 Item 5

  2.1 Item 6

    3.1 Item 7

Figure 2.6: The rendered Fibonacci document

The example above shows us that this works, but how exactly does this work, you might ask. The reason that this works is because the new object is added to the current document and the following `-` calls will thus be called on this object. An object can only be added in this way through the last statement in a non-scoped code block and the object will only be added if it is a subtype of `Content`. If the last statement does not return an object (a method call with a void return type or an assignment), nothing will be added. It is also possible to explicitly return something from a non-scoped code block by using the `return` keyword.

To further explain this, and to explain how we could possibly call functions on an object we need to introduce a new concept called `this`.

This

A Neio document has a concept of `this` like many programming languages do. `this` will always refer to the last element that was created in Text mode. But if that was the only way to interact with `this` we would not be able to call any methods (as we have nothing to call objects on) and we would not be able to add, modify or remove anything to/in the document either. To show how exactly this works we will take a look at the translated form of the previous example.

```
        new Document();

        FibEnumerate $var0 = new FibEnumerate()
        this.appendContent($var0);

        this.-(new neio.lang.Text(Item 1))
            .-(new neio.lang.Text(Item 2))
            .-(new neio.lang.Text(Item 3))
            ...
```

We see that a separate statement was created for everything above our code block and that the code in our code block was than directly included into the code. The last statement gets assigned to a variable and `appendContent` is called on `this`. This is always what will happen when you use a non-scoped code block, this means that the compiler knows about the `appendContent` method. Another example to further illustrate the use of non-scoped code blocks and `this` will be shown later on. After appending the content, a new statement is started, containing our well known chain of method calls, but the first method is called upon `this`.

In the next stage of translation, variables will be assigned to the objects and this will be filled in with the last object created in Text mode, or the `Content` returned in a non-scoped code block. The next step of translation is shown below.

```
        Document $var0 = new Document();

        FibEnumerate $var1 = new FibEnumerate();
        FibEnumerate $var2 = $var0.appendContent($var1);

        $var2.-(new neio.lang.Text(Item 1))
            .-(new neio.lang.Text(Item 2))
            .-(new neio.lang.Text(Item 3))
            ...
```

To show how code blocks work in combination with this, we will show another example.

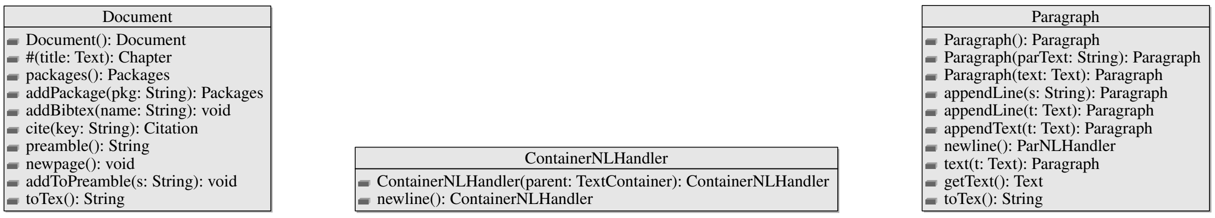
```
[Document]

This is the first paragraph
{
    Chapter c1 = #("Chapter 2");
    c1.*(new Text(c1.title()));
}

This is a paragraph
```

Now to explain the example, in the example a `Document` is created and a `Paragraph` is added to it in Text mode. Then in Code mode we create a new, `Chapter`, called `Chapter 1` using the `#` method. We then create a new `Enumerate` using the `*` method and finally we add a `Paragraph` in Text mode.

The translated version of this example is shown below:

```
new Document().newline().newline().text("This is the first paragraph").newline()
    ;

Chapter c1 = this.#("Chapter 1");
$var0 = c1.*(new Text(c1.title));

this.newline().newline().text("This is a paragraph");
```

In further translation the first `this` will result into the `Document` while the second this will be `c1`. The reason that the first `this` will be the `Document` instead of `Paragraph`, is because of ContextType's. `Paragraph` and `ContainerNLHandler` do not have a `#` method', but `Document` does as is shown in the following UML (only public members are shown to prevent clutter).


Document
Document(): Document
#(title: Text): Chapter
packages(): Packages
addPackage(pkg: String): Packages
addBibtex(name: String): void
cite(key: String): Citation
preamble(): String
newpage(): void
addToPreamble(s: String): void
toTex(): String
ContainerNLHandler
ContainerNLHandler(parent: TextContainer): ContainerNLHandler
newline(): ContainerNLHandler
Paragraph
Paragraph(): Paragraph
Paragraph(parText: String): Paragraph
Paragraph(text: Text): Paragraph
appendLine(s: String): Paragraph
appendLine(t: Text): Paragraph
appendText(t: Text): Paragraph
newline(): ParNLHandler
text(t: Text): Paragraph
getText(): Text
toTex(): String

A last point to note is that since your code is directly inserted into the translation, all the variables you have defined will be available from there on, and will take up their space in the current namespace. This means that you can use your variables in other code blocks, but also means that generic identifiers like `i` should be avoided as you might want to be able to use this identifier again later on.

Scoped code

If all you want to do, is execute some arbitrary code without corrupting your namespace, a block of scoped code is the way to go. The use of a block of scoped code is illustrated below.

```
[Document]
# Chapter 1
{{
List<String> l = new ArrayList<String>();
l.add("upquote");
l.add("pdfpages");
l.add("url");
for (int i = 0; i < l.size(); i = i + 1) {
    addPackage(l.get(i));
}
}}
```

In this example a `Document` is created but to be able to use it for our needs, we need a few more packages that are not included by default in `Document`. We also do not want to think of very descriptive identifiers for our variables as it's just a simple operation, the name `l`should suffice. Normally these variables would be defined in a short function and we would thus not have to worry about shadowing or overriding this name. But as we saw in Subsection 2.4.4, our code is just inserted straight into the final document. The difference is that a block of scoped block is inserted into its own scope. It also does not automatically add anything to the document like non-scoped code does. The previous example is translated to the following Java.

```
Document $var0 = new Document();
ContainerNLHandler $var1 = $var0.newline();
Chapter $var2 = $var0.hash(new neio.lang.Text("Chapter 1"));
ContainerNLHandler $var3 = $var2.newline();
{
    List<String> l = new ArrayList<String>();
    l.add("upquote");
    l.add("pdfpages");
    l.add("pdfpages");
    for (int i = 0; i < l.size(); i = i + 1) {
        $var0.addPackage(l.get(i));
    }
}
```

Another good reason for wanting to use scoped code, is when you only want to call a function that already adds an object to the `Document`. Using non-scoped code, this object would be added to the Document again through appendContent as seen in Subsection 2.4.4. An example is shown below.

```
[Document]

{{
    image("image1")
}}
```

This code creates an Image, but also adds it to the Document, as we don't want to add it twice, or invent an identifier for it, we use non-scoped code. The code for the image function is shown below.

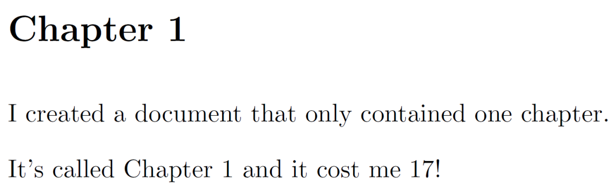
Listing 2.9: TextContainer.no

```
/**
 * Creates a new Image and adds it to this.
 *
 * @param imageName The name of the Image
 * @return The created Image
 */
Image image(String imageName) {
    return image(null, imageName);
}

/**
 * Creates a new Image and adds it to this.
 *
 * @param caption   The caption to use for the Image
 * @param imageName The name of the Image
 * @return The created Image
 */
Image image(Text caption, String imageName) {
    new Image(caption, imageName) neioImage;
    addContent(neioImage);

    return neioImage;
}
```

Inline code

Inline code is the last type of code blocks and is very like non-scoped code. The difference is that it only allows one statement, and the opening and closing brackets should be placed on the same line. As the name implies, this is code that is meant to be used inline. A few examples of this are shown below.

```
[Document]

# Chapter 1
I created a document that only contained one chapter.
{
    Chapter c1 = nearestAncestor(Chapter.class);
    Integer printing = 1;
    Integer driving = 2;
    Integer timeWasted = 14;
}

It is called { c1.title() } and it cost me { printing + driving + timeWasted }!
```



Chapter 1

I created a document that only contained one chapter.

It's called Chapter 1 and it cost me 17!

Figure 2.7: The rendered form of the example to the left

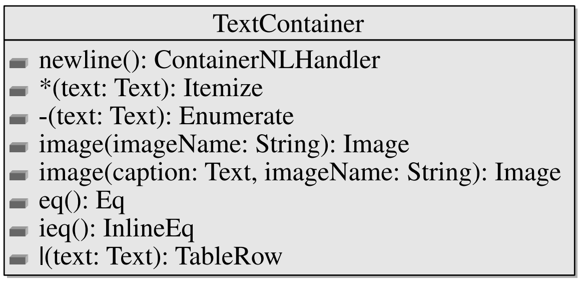
In the example we get the first `Chapter` structurally above us, define some integers, and then later on we get the title of this chapter and do a calculation with our integers. The result of this is returned and passed to the `text` method, as would normally happen with a piece of text. Note that whatever comes out of the inline code block is first transformed into a String using the toString() method.

The paragraph thus translates into the following chain.

```
this.text("It is called ").text(c1.title().toString()).text(" and it cost me ").
    text("" + (printing + driving + timeWasted)).text("!");
```

2.4.5 Text

Something that has not been noted yet, is that most of the time, we don't actually use Java String's. What is used most of the time is Neio Text. Neio Text (in Code mode, not in class files) is created like Java String's, by enclosing some text with a pair of doubles quotes (`"`). A Java String is still sometimes needed and can be created using a pair of triple, single-quotes (`'''`). We use three quotes, as a pair of single quotes still represents a Java Character, while two quotes would be very confusing when used in combination with double quotes in a non-monospace font. The inspiration of using three quotes came from the multi-line String literal in Python. The difference is that Neio text can contain anything you would type in Text mode, which includes code blocks, you can thus effectively endlessly nest Text and Code mode. The reason we use Neio text for most things instead of Java String's (just have a look at a few of the methods in the UML shown below), is because Neio text can be marked up, by using surround methods for example.



```
TextContainer
newline(): ContainerNLHandler
*(text: Text): Itemize
-(text: Text): Enumerate
image(imageName: String): Image
image(caption: Text, imageName: String): Image
eq(): Eq
ieq(): InlineEq
|(text: Text): TableRow
```

We could for example create a bold Chapter using the following code `new Chapter(*Bold chapter*)`.

2.5 Considerations

Whilst constructing the language a few other things were considered but not implemented, they will be discussed below.

Static typing

The language has been designed using static typing. The reason for this is, that while we are writing a Neio document in Code mode, or we are writing a Neio class, we can be notified of possible type mismatches. This decreases the time spent debugging and spend waiting on the compiler as you weed out one type mismatch after another. It also allows you to create a safer program as type errors might be caught at compile time when using static typing. On the other hand when using dynamic typing, type errors will only show up at runtime. With the help of an IDE, we also don't really have that much more typing to do than when we would be using dynamic typing. The language has been designed with static typing in mind, and types have been incorporated into the compiler, but the necessary checks are not yet in place. At this point in time, when a type error appears in the code, we will get a `LookupException` when we're looking for that method or variable. In the future, checks should be placed and appropriate errors should be thrown that make clear where exactly the error occurs.

Security

Another issue that we have considered is security. As it is possible to directly execute Java code, security is an issue. However, security is also an issue in LaTeX as has been shown in several articles [3] [4]. Since our language is more readable than LaTeX it might be easier to spot for people who understand code, that the document or library contains malicious code, than it is to spot malicious code in a LaTeX document. Adding a layer of security on top of Neio would require a lot of effort and research and is not in scope of this thesis. Thus we conclude by saying that we known there is a security risk, but dismissing it should be seen as future work.

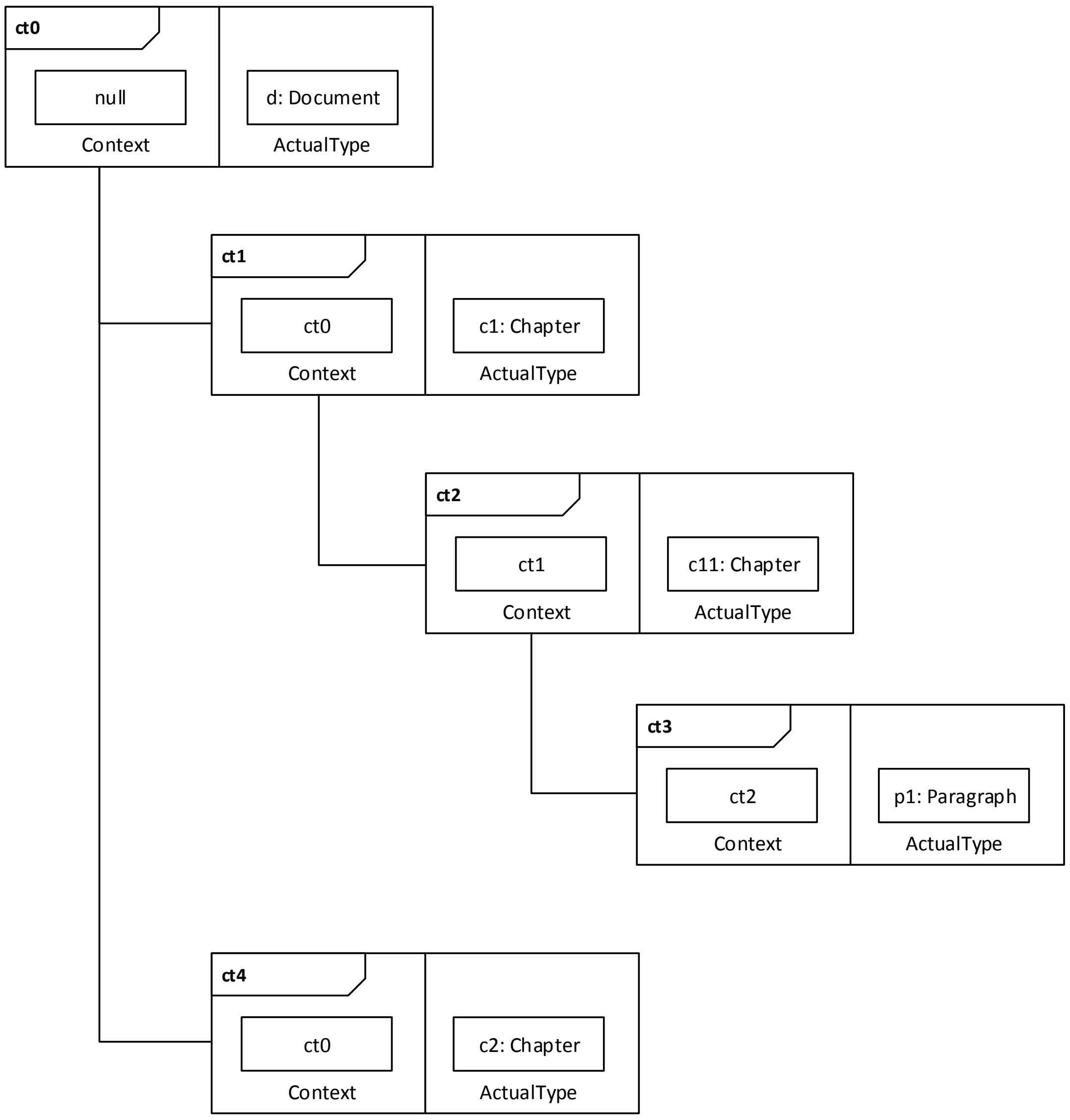


Figure 2.4: The correct ContextType representation

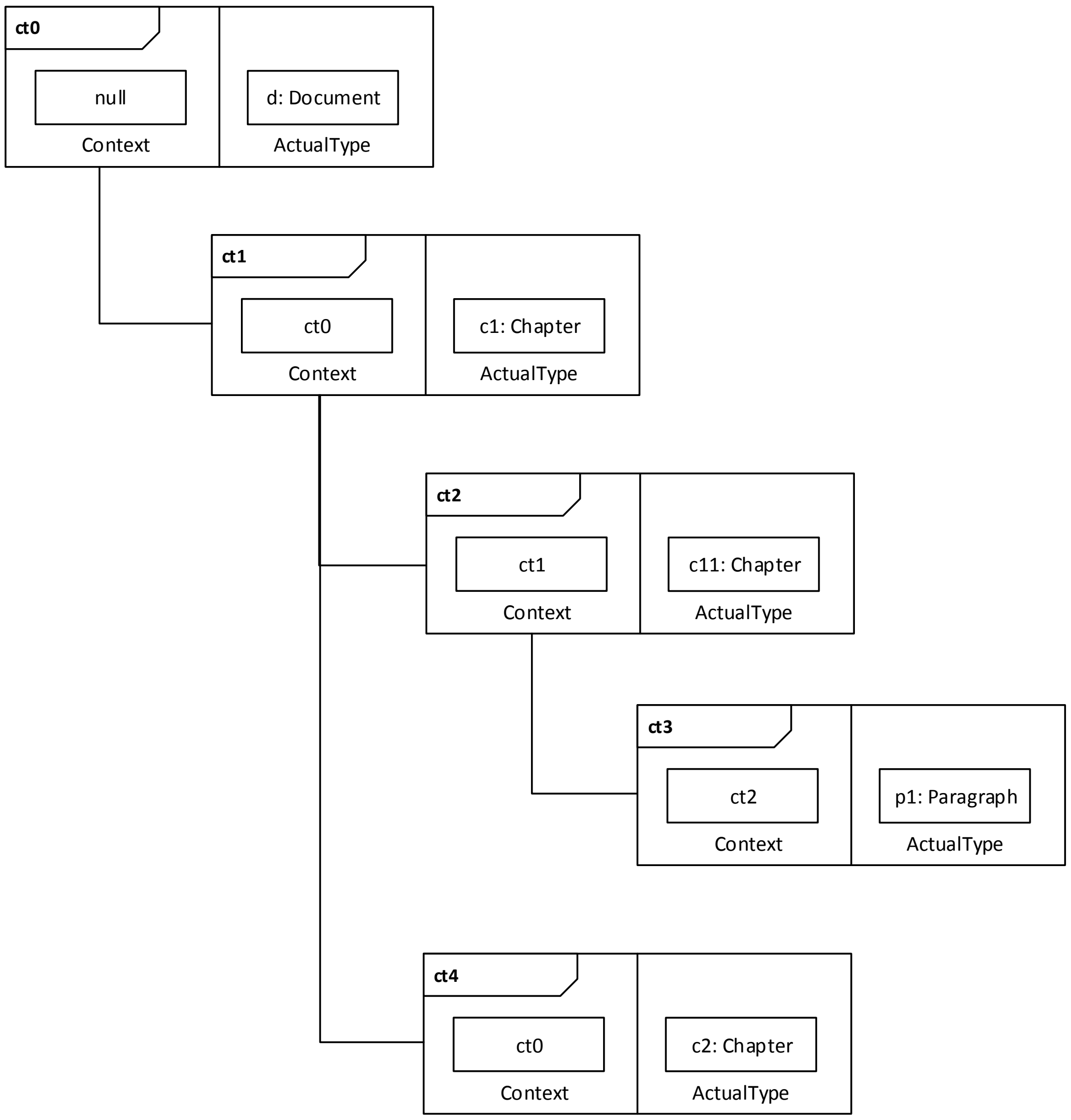


Figure 2.5: The wrong ContextType representation

Chapter 3

Supported document types and libraries

This chapter will go into some more details about what document types can be handled by Neio, and how to specifically build these kind of documents. We will also discuss which libraries have been recreated in Neio to allow for a wide use of the language.

3.1 Document classes

It is important to note that to create complete document classes would require a lot more work than was available for this thesis. If we just have a look at the user guides of Memoir [13] or KOMA-Script [9], we see that they consist of 609 and 419 pages respectively. This is not something we could hope to reproduce in the scope of this thesis.

The simplest of documents can be created using the `Document` class. It provides syntax to create anything that Markdown can create, and on top of that, it allows you to create tables, LaTeX math, uml,... It also allows you to use the code blocks defined in Subsection 2.4.4, to customise your document. For example, it is possible to place two `Content`'s next to each other by using the `leftOf` and `rightOf` methods that are defined in the `Content` class. The code of these methods is shown below. This code takes the parent from one of the two `Content`'s, the base, and links the other Content to the base. The implementation creates a root LaTeX `Minipage`

[15] if these are the first two objects in a row to be placed next to each other, otherwise it just wraps the two `Content`'s in a `minipage`. The root `minipage` takes up the total width, while the `minipage`'s that wrap the content take up 1/children of the width.

Listing 3.1: Content.no

```
 * Sets {@code c} right of this Content and sets the parent of
 * {@code c} if we have a parent or sets this parent if {@code c}
 * has a parent. One of the Contents should have a parent!
 *
 * @param c The Content to our right
 * @return The Content to our right
 */
<T extends Content> T leftOf(T c) {
    Minipage mp = prepareParent(this, c);
    Integer siblings = ((Minipage) (mp.content(0))).siblings();
    Minipage mp2 = new Minipage(siblings);
    mp2.addContent(c);
    mp.addContent(mp2);

    return c;
}

/**
 * Sets {@code c} right of this Content and sets the parent of
 * {@code c} if we have a parent or sets this parent if {@code c}
 * has a parent. One of the Contents should have a parent!
 *
 * @param c The Content to our left
 * @return The Content to our left
 */
<T extends Content> T rightOf(T c) {
    c.leftOf(this);
    return c;
}


/**
 * Prepares the parent of {@code left} or {@code right} to have
 * horizontally aligned children (by using Minipage).
 *
 * @param left  The left element whose parent has to be prepared
 * @param right The right element whose parent has to be prepared
 * @return The root Minipage
 */
private Minipage prepareParent(Content left, Content right) {
    Content child = null;
    if (left.parent() == null) {
        child = right;
    } else {
        child = left;
    }

    if (Minipage.class.isInstance(child.parent())) {
        List<Minipage> descendants = child.parent().directDescendants(Minipage.
            class);
        for (int i = 0; i < descendants.size(); i = i + 1) {
            Minipage m = descendants.get(i);
            m.incrementSiblings();
        }
        return (Minipage) (child.parent().parent());
    }

    Minipage rootPage = new Minipage(0);
    child.replaceWith(rootPage);

    Minipage mpl = new Minipage(1);
    mpl.addContent(left);
    rootPage.addContent(mpl);

    return rootPage;
}
```

Another of implementing this, that would probably have worked better was through `TikZ` picture's. This would have been better because it allows you to easily set any two things next to each other or above/beneath each other. `Minipage`'s require a lot more manual work, they might also split at a page-end which is not what we really want.

The `Document` document class can thus be used to write simple documents, such as small reports on various tasks.

3.1.1 Book

A prime subject of writing a book, is this document itself. The entirety of this book has been created in a single Neio document. The way we do this is by creating a new `Document` class that represents our LaTeX template. In this case the template is implemented in the `Thesis` class given below.

Listing 3.2: Thesis.no

```
namespace neio.thesis;

import java.util.ArrayList;
import java.util.List;

import neio.stdlib.BlankPage;
import neio.stdlib.Chapter;
import neio.stdlib.Document;
import neio.stdlib.Packages;
import neio.stdlib.Pdf;
import neio.stdlib.Toc;

/**
 * A document class to be used in for a Thesis
 */
class Thesis extends Document;

private Packages packages;

/**
 * Intialises the document class, adding the required packages and adding a ToC
 */
Thesis() {
    packages = new Packages();
    packages.add("graphicx")
    .add("fancyhdr")
    .add("amsmath")
    .add("float")
    .add("listings")
    .add("tocloft")
    .add("color")
    .add("pdfpages")
    .add("hyperref");

    List<String> options = new ArrayList<String>();
    options.add("english");
    packages.add(options, "babel");

    addClassMapping(Chapter.class, ThesisChapter.class);
    addContent(new Pdf("TitlePage"));
    addContent(new BlankPage());
    addContent(new Pdf("TitlePage"));
    addContent(new FrontMatter());
}

/**
 * Sets up the main matter
 */
void mainMatter() {
    addContent(new Toc());
    addContent(new MainMatter());
}

/**
 * Builds the preamble for this document class
 *
 * @return The preamble for this document class
 */
String preamble() {
    StringBuilder preamble = new StringBuilder("\\documentclass[11pt,a4paper,
        oneside,notitlepage]{book}\n")
    .append("\\setlength{\\topmargin}{2cm}\n")
    .append("\\setlength{\\headheight}{0.5cm}\n")
    .append("\\setlength{\\headsep}{1cm}\n")
    .append("\\setlength{\\oddsidemargin}{3.5cm}\n")
    .append("\\setlength{\\evensidemargin}{3.5cm}\n")
    .append("\\setlength{\\textwidth}{16cm}\n")
    .append("\\setlength{\\textheight}{23.3cm}\n")
    .append("\\setlength{\\footskip}{1.5cm}\n")
    .append(packages.toTex())
    .append("\\DeclareGraphicsRule{*}{mps}{*}{}")
    .append("\\pagestyle{fancy}\n")
    .append("\\renewcommand{\\chaptermark}[1]{\\markright{\\MakeUppercase{#1}}}\
        n")
    .append("\\renewcommand{\\sectionmark}[1]{\\markright{\\thesection~#1}}\n")
    .append("\\newcommand{\\headerfmt}[1]{\\textsl{\\textsf{#1}}}\n")
    .append("\\newcommand{\\headerfmtpage}[1]{\\textsf{#1}}\n")
    .append("\\fancyhf{}\n")
    .append("\\fancyhead[LE,RO]{\\headerfmtpage{\\thepage}}\n")
    .append("\\fancyhead[LO]{\\headerfmt{\\rightmark}}\n")
    .append("\\fancyhead[RE]{\\headerfmt{\\leftmark}}\n")
    .append("\\renewcommand{\\headrulewidth}{0.5pt}\n")
    .append("\\renewcommand{\\footrulewidth}{0pt}\n")
    .append("\\fancypagestyle{plain}{\n")
    .append("\\fancyhf{}\n")
    .append("\\fancyhead[LE,RO]{\\headerfmtpage{\\thepage}}\n")
    .append("\\fancyhead[LO]{\\headerfmt{\\rightmark}}\n")
    .append("\\fancyhead[RE]{\\headerfmt{\\leftmark}}\n")
    .append("\\renewcommand{\\headrulewidth}{0.5pt}\n")
    .append("\\renewcommand{\\footrulewidth}{0pt}\n")
    .append("}\n")
    .append("\\renewcommand{\\baselinestretch}{1.5}\n")
    .append("\\hyphenation{ditmagnooitgesplitstworden dit-woord-splitst-hier}\n
       ")
    .append("\\setlength{\\parindent}{0em}\n")
    .append("\\setlength{\\parskip}{1em}\n")
    .append("\\definecolor{comments}{RGB}{98, 151, 85}\n")
    .append("\\lstset{\n")
    .append("belowcaptionskip=1\\baselineskip,\n")
    .append("breaklines=true,\n")
    .append("frame=l,\n")
    .append("comment=[l]{//},\n")
    .append("morecomment=[s]{/**}{*/},\n")
    .append("commentstyle=\\color{comments},\n")
    .append("xleftmargin=\\parindent,\n")
    .append("showstringspaces=false,\n")
    .append("basicstyle=\\footnotesize\\ttfamily,\n")
    .append("keywordstyle=\\bfseries,\n")
    .append("stringstyle=\\ttfamily,\n")
    .append("numbers=left,\n")
    .append("numbersep=7pt,\n")
    .append("numberstyle=\\tiny,\n")
    .append("lineskip={-1.5pt}\n")
    .append("}\n");

    return preamble.toString();
}
```

It also says that any `Chapter` that is automatically created, typically by the `#` method in `Document` or `Chapter`, should be an instance of `ThesisChapter`. This is done through the following line.

```
addClassMapping(Chapter.class, ThesisChapter.class);
```

A `ThesisChapter` just redefines a chapter so that it translates into the LaTeX `chapter` macro when it is level 1 instead of directly calling the LaTeX section macro.

Listing 3.3: ThesisChapter.no

```
namespace neio.thesis;

import neio.lang.Text;
import neio.stdlib.Chapter;

class ThesisChapter extends Chapter;

ThesisChapter(Text title, Integer level) {
    super(title, level);
}

String subs(Integer lvl) {
    String subs = "";
    for (int i = 2; i < lvl; i = i + 1) {
        subs = subs + "sub";
    }

    if (lvl < 2) {
        texName = "chapter";
    } else {
        texName = "section";
    }

    return subs;
}
```

It extends the Document class and adds its own packages and creates its own LaTeX preamble. It also provides a method that will initialise the main matter, which contains a Table of Contents (ToC) using the following line. The code for a ToC is shown below.

Listing 3.4: Toc.no

```
namespace neio.stdlib;

import neio.lang.Content;

class Toc extends Content;

String toTex() {
    return "\\newpage\n\\pagestyle{fancy}\n\\tableofcontents\n";
}
```

We can also add things to a document class ourselves which might allow us to reuse other document classes For example, to add an abstract to the Thesis document class, we just have to create an Abstract class that extends Content and add it to the document at the right time. This allows us to add more functionality to a document class, without having to modify the document class itself, or without having to create a new one, e.g. Thesis and ThesisWithAbstract.

To add an abstract , we could initialise an Abstract, this Abstract might then overwrite the newline function to create an AbstractNLHandler. The AbsttractNLHandler can then override the text method to initialize itself, as an abstract is just a block of text. Overriding the newline method in the AbstractNLHandler allows us to escape this ”Abstract mode” by typing an empty line. The code to for Abstract and AbstractNLHandler are shown below, an example of what the output looks like can be found at the beginning of this document.

3.1.2 Other document classes

To be able to create an article, a letter, or anything else, the only thing we would have to do is create a new Neio class for this document class. This Neio class will then implement its own

LaTeX preamble and decide what packages to include by default and it might have to redefine what types of objects should be used using `addClassMapping`. It might also add methods to be filled in, using a code block, or by overwriting methods and making use of the newline handlers.

For example, you might create a `LetterHeader` class that represents the header of a formal letter. This header requires some information about the writer, the name, the addressee,.... One way to fill these in is just by passing them to the constructor of `LetterHeader` or by using setters defined in `LetterHeader`. Another way to do this, is by overwriting some methods in `LetterHeader`. We could overwrite the `#` method to represent the author's name, and `-` to represent the addressee for example. Another possibility is to used nested methods for this, `#` represents the author, `##` represents the addressee and so on.

Yet another way would be to create objects for the parameters we need (grouping some together if that makes sense) such as `Author` and `Addressee`. We then implement a method, let us take `#` once more, in `LetterHeader` that creates a new `Author` and returns it. We then override a method, the same or a different one, in `Author` that creates the `Addressee` and returns it. The advantage of this last step is that the structure of the letter is well split up and is enforced on the user. Forgetting to define the `Author` should be immediately noticeable as the method to create the `Addressee` should not be known yet without creating the `Author` first. On the other hand this is also more prune to error as people tend to forget things all the time.

3.1.3 Slides

3.2 Libraries

To show what the language is capable of a few libraries were created or reimplemented. All of the currently available libraries in Neio will be explained below. Note that none of these libraries are complete and would require a lot more effort to reach a state of completion. Just having a look at the manuals of a few popular libraries such as TikZ[17] we can see that more time is needed to write such a library than there is available for this thesis. The libraries that were created, are meant for illustrative purpose, to show what Neio is capable of.

3.2.1 BibTeX

The library created for BibTeX is one of the easiest ones created for Neio. The implementation is just a binding to LaTeX and does not really allow for a lot of customisation as we do not have an object model of the BibTeX file. We chose to implement BibTeX this way to illustrate that binding to LaTeX is not very difficult, and because we would have had to create an entire parser for BibTeX to be able to create the object structure needed to have full control over the BibTeX files. As the BibTeX format is quite complicated we felt that our time was better spent elsewhere. This is mainly because the only thing we require of BibTeX for this thesis, is to allow us to easily quote articles, websites and so on. The code for our implementation of BibTeX can be found below.

Listing 3.5: Bibtex.no

```
namespace neio.stdlib;

import neio.lang.Content;

/**
 * Implements a binding to the LaTeX's BibTeX
 */
class Bibtex extends Content;

String name;

/**
 * Initializes the BibTeX
 *
 * @param name The path to the BibTeX file
 */
Bibtex(String name) {
    this.name = name;
}

/**
 * Cites something defined in this BibTeX
 *
 * @param cname The keyword of the citation
 * @return A Citation object representing the citation of {@code cname}
 */
Citation cite(String cname) {
    return new Citation(cname);
}

/**
 * Creates a TeX representation of this BibTeX, without citations
 *
 * @return The TeX representation of this
 */
String toTex() {
    StringBuilder result = new StringBuilder("\n\\bibliographystyle{plain}\n");
    result.append("\\bibliography{").append(name).append("}\n");

    return result.toString();
}
```

A BibTeX can now be added through the `addBibtex` method in `Document` and we can cite an entry by using the proxy method, `cite`, in `Document`.

Listing 3.6: Document.no

```
/**
 * Sets a BibTeX file for this Document
 *
 * @param The name of the BibTeX file
 */
void addBibtex(String name) {
    this.bibtex = new Bibtex(name);
}

/**
 * A proxy method for {@code BibTeX}'s {@code cite}
 * Returns a Cite that is linked to {@code key}
 *
 * @param key The key to access the citation
 * @return        The Cite corresponding to {@code key}
 */
Citation cite(String key) {
    if (bibtex != null) {
        return bibtex.cite(key);
    } else {
        return null;
    }
}
```

Lastly we show the code for the `Citation` class.

Listing 3.7: Citation.no

```
namespace neio.stdlib;

import neio.lang.Text;

/**
 * Represents something that has been cited.
 * It is a subclass of Text to allow for it to be used in inline code blocks.
 */
class Citation extends Text;

String name;

/**
 * Initialises a Citation
 *
 * @param name The name of the Citation
 */
Citation(String name) {
    this.name = name;
}

/**
 * Creates a TeX representation of this
 *
 * @return The TeX representation of this
 */
String thisToTex() {
    return "\\cite{" + name + "}";
}
```

The `Citation` class is a subclass of `Text` as we call it through an inline code block. All the cites in this thesis have been provided through these classes and are called as below. `key` is a Java String as it is used to do a lookup on the BibTeX entries, it should not be allowed to be marked up.

```
[Document]
This is how we cite something {cite('''key''')}.
```

3.2.2 References

The way references have been implemented is quite straightforward. Any class that implements the `Referable` interface, such as `Chapter` or `Image`, can be referenced by calling the `ref` method. The way we typically go about a reference is as follows.

```
[Document]
# Chapter 1
{Chapter chap = (Chapter) parent()}

In {chap.ref()} we will explain references.
```

What we use here is a little trick to be able to get the `Chapter` into a variable. What happens is as follows, a `Chapter` is created using the `#` method and then the `newline` method is called upon it (this method has been inherited from `TextContainer`). A `NewlineHandler` always takes the object that created it as a parameter in its constructor so that it can refer to it later on. The `parent` call in the inline code block is thus called on the NewlineHandler, more specifically the `ContainerNLHandler`. As the parent of a `ContainerNLHandler` is a `TextContainer`, a cast to `Chapter` is needed. The code block is allowed to be an inline code block instead of a non-scoped code block because the statement in it is an assignment and thus returns nothing. To reference an object LaTeX needs a unique label, we use the hashcode of an object for this as it is easy to retrieve at any time and it is unique. The newline method from `TextContainer` as well as the `ref` method in `Chapter` and the `Referable` interface are shown below.

Listing 3.8: TextContainer.no

```
/**
```

```
 * Handles newlines
 *
 * @return Returns a new ContainerNLHandler
 */
ContainerNLHandler newline() {
    return new ContainerNLHandler(this);
}
```

Listing 3.9: Chapter.no

```
/**
 * Creates a reference to this Chapter and adds a correct prefix to it
 * such as Chapter, Section or Subsection.
 *
 * @return A Reference to this Chapter
 */
Reference ref() {
    String prefix = "";
    if (level < 2) {
        prefix = "Chapter";
    } else if (level == 2) {
         prefix = "Section";
    } else {
        prefix = "Subsection";
    }

    return new Reference(prefix, "" + hashCode());
}
```

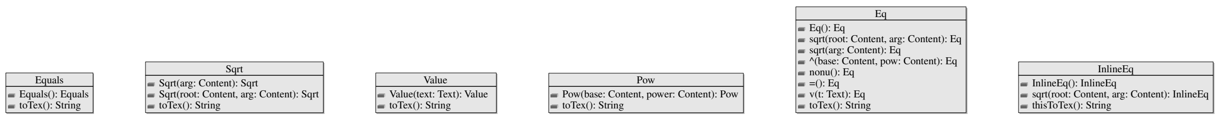
Listing 3.10: Referable.no

```
namespace neio.lang;

/**
 * Declares that a class can be referred to
 */
interface Referable;

/**
 * References an object
 *
 * @return A reference to this object
 */
Reference ref();
```

As we need a variable to reference something, this means that the object we want to refer to has to be available when we want to refer to it. This means that we can not refer to things that are defined later on in the project, which LaTeX can do. LaTeX is able to do this because it compiles more than once and is able to gather the information needed to create a reference in one of the earlier runs. For Neio to be able to do this, we would have to do something similar as we can not simply let the compiler create the object we want to reference in advance. This is not possible because the creation of an object is dependent on its context.

3.2.3 LaTeX math and amsmath

As mentioned in Subsection 1.1.2, LaTeX is very good at typesetting mathematical formulas. As such it is something that we could not forget about. Two ways to make use of the LaTeX math and the amsmath package have been created. The UML below shows the currently implemented set of objects in the Neio math library.



To create an inline math formula, in LaTeX this is done by surrounding some text with a pair of 's, we create an instance of the `InlineEq` class. The only method that has been implemented for it thus far is `sqrt` so we will show how we can create an inline square root.

```
{
    Integer base = 2;
    Integer arg = 10;
}
${ieq().sqrt("{base}", "{arg}")}$
```

As you can see we also make use of the `$` operator, in our case it is a surround method defined in `Text`. What we do next is a bit tricky, inside the surround method, we create an inline code block, and in it we create an `InlineEq` on which we call the `sqrt` method. The reason we have to open this code block is because the contents of a surround method has to be a text and in normal text we can not call functions. The arguments that are passed to the `sqrt` method are also special, each of the arguments is an inline code block wrapped in a Neio text. Once again, this has been done because we need to be able to pass `Text` to the `sqrt` method, not just an integer. A solution for this is proposed in Section 5.3.

Continuing the example, it is also possible to explicitly create an equation, in LaTeX this is done using the `equation` environment.

```
{{
    eq().nonu().sqrt("{base}", "{arg}");
    eq().^("{base}", "{arg}").=().v("{
        Math.pow(base, arg)}");
}}
```

$$\sqrt[2]{10}$$

$$2^{10} = 1024.0 \tag{1}$$

Figure 3.1: The rendered form of the example to the left

The code above creates two `Equation`, on the first one it disable numbering through the `nonu` method and calls sqrt on the equation, as shown before. In the second equation we use the `^` method to create an exponentiation and use the `=` method to create an equation from it. The `v` method creates a `Value` and calculates the actual value of what has been typeset before it. To be clear, everything except for the `Math.pow()` typesets something, but computes nothing. This can be clearly seen in Figure 3.1. As most of the code in this library is straightforward, does nothing new and is a direct translation to LaTeX, we only show the code for `Eq` class.

Listing 3.11: Eq.no

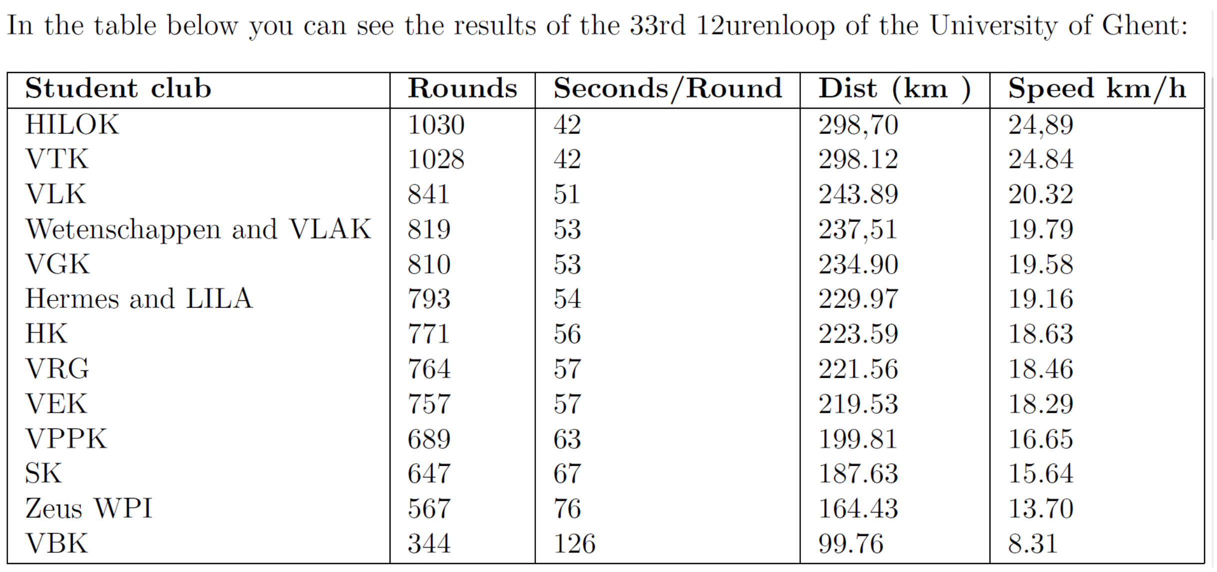
```
namespace neio.stdlib.math;

import neio.lang.Content;
import neio.lang.Text;

/**
 * Creates an explicit equation
 */
class Eq extends Content;

String texname;

/**
 * Initialises the equation
 */
Eq() {
    texname = "equation";
}

/**
 * Creates a square root and adds it to this equation, returning this to allow
     further chaining
 *
 * @param root The base of the square root
 * @param arg  The value to take the square root of
 * @return This equation
 */
Eq sqrt(Content root, Content arg) {
    addContent(new Sqrt(root, arg));
    return this;
}

/**
 * Creates a square root with no explicit base
 *
 * @param arg The value to take the square root of
 * @return This equation
 */
Eq sqrt(Content arg) {
    return sqrt(null, arg);
}

/**
 * Creates an exponentiation and appends it to this equation
 *
 * @param base The base of the exponentiation
 * @param pow The exponent of the exponentiation
 * @return This equation
 */
Eq ^(Content base, Content pow) {
    addContent(new Pow(base, pow));
    return this;
}

/**
 * Disables numbers for this equation
 *
 * @return This equation
 */
Eq nonu() {
    texname = texname + "*";
    return this;
}

/**
 * Appends an equals sign to this equation
 *
 * @return This equation
 */
Eq =() {
    addContent(new Equals());
    return this;
}

/**
 * Appends a value to this equation
 *
 * @return This equation
 */
Eq v(Text t) {
    addContent(new Value(t));
    return this;
}

/**
 * Creates the TeX representation of this equation
 *
 * @return The TeX representation of this equation
 */
String toTex() {
    return "\\begin{" + texname + "}" + super.toTex() + "\n\\end{" + texname +
        "}";
}
```

3.2.4 LaTeX tables

Tables are one of the main things missing in Markdown and is one of the things people have the most problems with in LaTeX. As such we tried to make a table that is simple to create, that is readable in source code and that allows you to easily edit values in the table later on, making it so we don't have to inline all of the changes, making it hard to read how the table. The table is created like you would create an ASCII table, with pipes, spaces, dashes and values for the elements of the table. An example is shown below.

```
[Document]
```

```
In the table below you can see the results of the 33rd 12urenloop of the
    University of Ghent:

|       Student club        | Rounds | Seconds/Round | Dist (km) | Speed km/h |
------------------------------------------------------------------------------
| HILOK                     |  1030  |      42       |   298,70  |   24,89    |
| VTK                       |  1028  |      42       |   298.12  |   24.84    |
| VLK                       |   841  |      51       |   243.89  |   20.32    |
| Wetenschappen and VLAK    |   819  |      53       |   237,51  |   19.79    |
| VGK                       |   810  |      53       |   234.90  |   19.58    |
| Hermes and LILA           |   793  |      54       |   229.97  |   19.16    |
| HK                        |   771  |      56       |   223.59  |   18.63    |
| VRG                       |   764  |      57       |   221.56  |   18.46    |
| VEK                       |   757  |      57       |   219.53  |   18.29    |
| VPPK                      |   689  |      63       |   199.81  |   16.65    |
| SK                        |   647  |      67       |   187.63  |   15.64    |
| Zeus WPI                  |   567  |      76       |   164.43  |   13.70    |
| VBK                       |   344  |     126       |    99.76  |    8.31    |
```



In the table below you can see the results of the 33rd 12urenloop of the University of Ghent:

Student club	Rounds	Seconds/Round	Dist (km)	Speed km/h
HILOK	1030	42	298,70	24,89
VTK	1028	42	298.12	24.84
VLK	841	51	243.89	20.32
Wetenschappen and VLAK	819	53	237,51	19.79
VGK	810	53	234.90	19.58
Hermes and LILA	793	54	229.97	19.16
HK	771	56	223.59	18.63
VRG	764	57	221.56	18.46
VEK	757	57	219.53	18.29
VPPK	689	63	199.81	16.65
SK	647	67	187.63	15.64
Zeus WPI	567	76	164.43	13.70
VBK	344	126	99.76	8.31

Figure 3.2: The rendered form of the table given above

First of all, we note that it does not matter how many spaces are placed in side the elements of the table. The way this table is build is by mixing the use of normal method calls of `|`, a nested call of `-` and by making use of a NewlineHandler. The table is created by the `|` method in `TextContainer`, so that it van be used in `Chapter`'s as well as `Document`'s (they are both subclasses of `TextContainer`).

Listing 3.12: TextContainer.no

```
/**
 * Creates a new Table and adds a new TableRow to it.
 * The Table is added to this.
 *
 * @param text The text to use in the first element of the TableRow
 * @return The created TableRow
 */
TableRow |(Text text) {
    new TableRow(text) row;
    new Table(row) table;
    addContent(table);

    return row;
}
```

The `TableRow` that is returned form the the `|` method is `TableRow` that is stil under construction. We continue building it by repeatedly calling the `|` method on it, this time this method is located in `TableRow`. The code for the `TableRow` and `TableElement` are shown below.

Listing 3.13: TableRow.no

```
namespace neio.stdlib;

import java.util.List;
import java.util.ArrayList;

import neio.lang.Content;
import neio.lang.Text;

class TableRow extends Content;

TableElement fte = null;
List<TableElement> elements = new ArrayList<TableElement>();
boolean bold = false;

TableRow() {
}

TableRow(Text text) {
    elements.add(new TableElement(text));
}

void setBold() {
    bold = true;
}

TableRow |(Text text) {
    elements.add(new TableElement(text));
    return this;
}

TableRow |() {
    fte = new FinalTableElement();
    return this;
}

Table parent() {
    return (Table) (super.parent());
}

TableRow appendRow(TableRow row) {
    return parent().appendRow(row);
}

Hline appendHline() {
    return parent().appendHline();
}

TableNLHandler newline() {
    return new TableNLHandler(this);
}

String header() {
    StringBuilder header = new StringBuilder();
    for (int i = 0; i < elements.size(); i = i + 1) {
        TableElement te = elements.get(i);
        header.append(" ").append(te.header()).append(" | ");
    }

    return header.toString();
}

String toTex() {
    if (elements.isEmpty()) {
        return "";
    }

    if (bold) {
        elements.get(0).setBold();
    }
    StringBuilder result = new StringBuilder(elements.get(0).toTex());
    for (int i = 1; i < elements.size(); i = i + 1) {
        TableElement element = elements.get(i);
        if (bold) {
            element.setBold();
        }
        result.append(" & ").append(element.toTex());
    }

    if (fte != null) {
        result.append(fte.toTex());
    }

    return result.toString();
}
```

Listing 3.14: TableElement.no

```
namespace neio.stdlib;

import java.util.List;
import java.util.ArrayList;

import neio.lang.BoldText;
import neio.lang.Content;
import neio.lang.Text;

/**
 * Represents an element in a Table
 */
class TableElement extends Content;

protected String header;
Text text;

/*
 * Initialises the TableElement and alligns it to the left
 */
TableElement() {
    header = "l";
}

/*
 * Initialises the TableElement with a given value
 *
 * @param text The text to be used as value of this element
 */
TableElement(Text text) {
    this();
    this.text = text;
}

/**
 * Enables the bold property, if this is set the element will be
 * printed in bold
 */
void setBold() {
    text = new BoldText(text);
}

/**
 * Returns this element should be aligned
 *
 * @return A string representing how this element should be aligned
 */
String header() {
    return header;
}

/**
 * Build the TeX representation of this
 *
 * @return The TeX representation of this
 */
String toTex() {
    return text.toTex();
}
```

As you can see in the `TableRow` class, in the DSL we created, a `|` method without any arguments, denotes the end of a `TableRow`. To continue building the table, we use a `TableNLHandler` that has been created in the `newline` method of the `TableRow`. We use the nested '-' method in this handler to be able to insert newlines. In our DSL the length of the nested method is ignored, but it could be used to measure the width of a table for example. The handler also offers the `|` method which appends a new `TableRow` to the `Table`.

Finally we can write an empty newline to get out of the DSL for creating a `Table` because of the `newline` method in the handler.

The following libraries that were implemented are more domain specific and less generally usable. However this shows that we can use Neio to typeset a wide variety of different documents.

3.2.5 Red black trees

Red black trees are trees that are constantly update according to a defined algorithm. When we want to show a red black tree in LaTeX, we can typeset an instance of such a tree using `TikZ`. Or we could create it in a third party program and include an image of an instance of such tree in our document, this is a solution that would also work Markdown.

However, since an algorithm is used, and we have a programming model, couldn't we just build the tree and then display it? It turns out that we can. All that we have to do is take some code that can generate red black trees and add a display method to it. We show an example of the red black trees in use.

Listing 3.15: rbtDocument.no

```
[Document]

{
    List<Integer> tree = new ArrayList<Integer>();
    tree.add(33);
    tree.add(15);
    tree.add(10);
    tree.add(5);
    tree.add(20);
    tree.add(18);
    tree.add(47);
    tree.add(38);
    tree.add(36);
    tree.add(39);

    String numbers = '''''' + tree.get(0);
    for (int i = 1; i < tree.size(); i = i + 1) {
        numbers = numbers + ''', ''' + tree.get(i);
    }
    Integer a = 5;
}

Given the red black tree of {numbers}
{
    RedBlackTree rbt = new RedBlackTree().insert(tree);
    return rbt;
}

Add 49
{
    RedBlackTree rbt2 = rbt.insert(49);
    return rbt2;
}

Add 51
{
    RedBlackTree rbt3 = rbt2.insert(51);
    return rbt3;
}
```

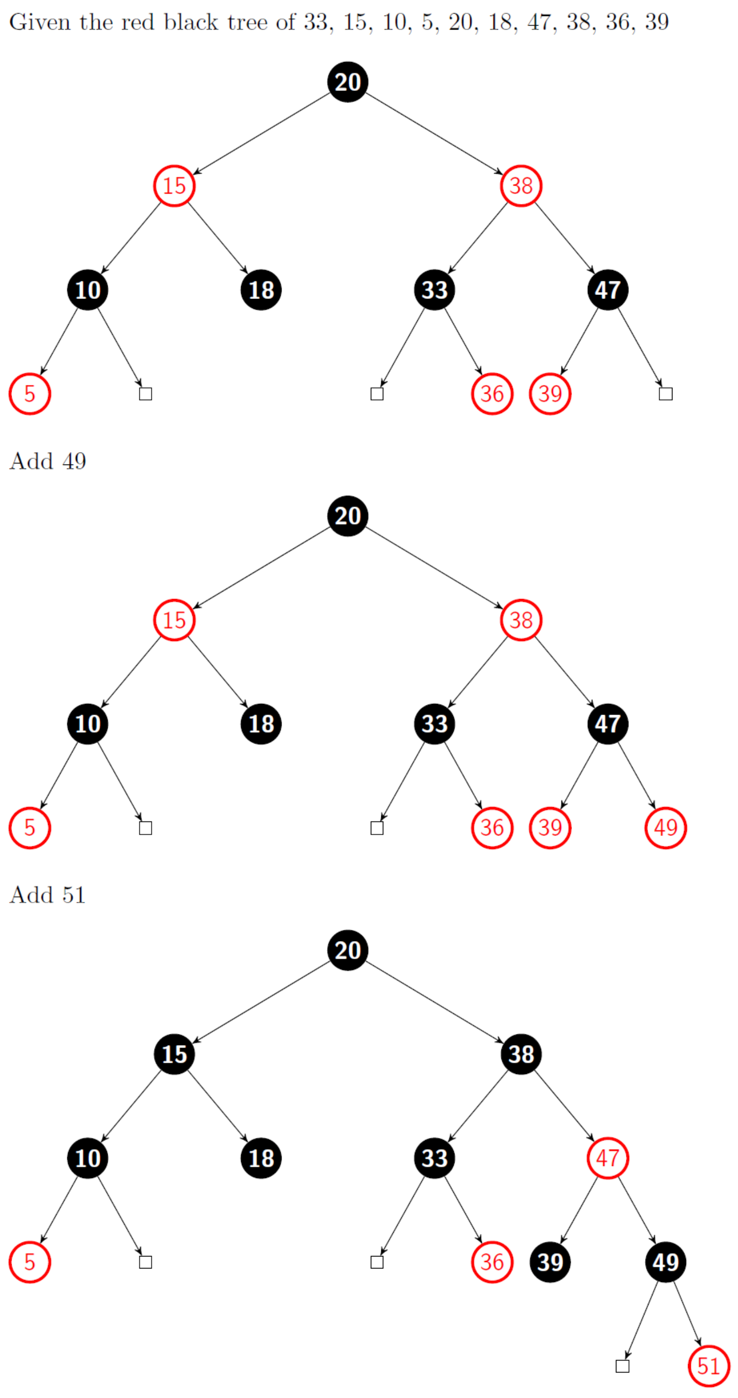


Figure 3.3: The rendered form of the above example

The rendered document is shown in Figure 3.3. As we can see it is very easy to create and show a red black tree. We could do this for any tree or graph as long as we have code for it and write a method that can display it. As showing the algorithm to create red black trees is not the purpose of this example, we only show the code to display it and to append something to it.

Listing 3.16: RedBlackTree.no

```
/**
 * Clones the current red black tree and inserts a new element into items
 *
 * @param value The value to insert
 * @return The newly created red black tree
 */
RedBlackTree insert(Integer value) {
    if (last == null) {
        last = new RedBlackNode(RedBlackNode.BLACK, value);
        return this;
    }
    RedBlackTree newTree = new RedBlackTree(root());
    newTree.insert(value, newTree.root());

    return newTree;
}
/**
 * Makes sure the TikZ library is used and that everything is prepared to
     typeset a red black tree in LaTeX
 */
void preTex() {
    Document d = nearestAncestor(Document.class);
    d.addPackage("tikz");

    StringBuilder header = new StringBuilder("\\usetikzlibrary{arrows}")
    .append("\\tikzset{treenode/.style = {align=center, inner sep=0pt, text
        centered, font=\\sffamily},\n")
    .append("arn_n/.style = {treenode, circle, white, font=\\sffamily\\bfseries,
         draw=black,\n")
    .append("fill=black, text width=1.5em},% arbre rouge noir, noeud noir\n")
    .append("arn_r/.style = {treenode, circle, red, draw=red,\n")
    .append("text width=1.5em, very thick},% arbre rouge noir, noeud rouge\n")
    .append("arn_x/.style = {treenode, rectangle, draw=black,\n")
    .append("minimum width=0.5em, minimum height=0.5em}% arbre rouge noir, nil\n
        ")
    .append("}\n");

    d.addToPreamble(header.toString());

    super.preTex();
}

/**
 * Creates the TeX representation this red black tree
 *
 * @return The TeX representation of this tree
 */
String toTex() {
    if (last == null) {
        return "";
    }

    StringBuilder result = new StringBuilder("\n\\begin{tikzpicture}[->,>=
        stealth',level/.style={sibling distance = 5cm/#1,");
    result.append("level distance = 1.5cm}]\n");
    result.append(root().toTex());
    result.append("\\end{tikzpicture}");

    return result.toString();
}
```

Listing 3.17: RedBlackNode.no

```
/**
 * Recursively creates the Text representation of the tree.
 *
 * @return The TeX representation of the tree
 */
String toTex() {
    StringBuilder result = new StringBuilder();
    if (isRoot()) {
        result.append("\\node [arn_");
    } else {
        result.append("child { node [arn_");
    }

    if (isBlack()) {
        result.append("n]");
    } else {
        result.append("r]");
    }

    result.append(" {").append(value.toString()).append("}\n");
    // If we don't have two empty leaves
    if (!(!hasLeft() && !hasRight())) {
        if (!hasLeft()){
            result.append("child { node [arn_x] {}}\n");
        } else {
            result.append(left.toTex());
        }
        if (!hasRight()) {
            result.append("child { node [arn_x] {}}\n");
        } else {
            result.append(right.toTex());
        }
    }

    if (!isRoot()) {
        result.append("\n}\n");
    } else {
        result.append(";\n");
    }

    // Remove empty newline, it breaks the tikzpicture
    return result.toString().replaceAll("(\n)\\1+","$1");
}
```

We also show the LaTeX code that is generated for the first of the three trees. This code is not that easy to read, hard to write and thus prune to syntax errors. This code is also newline sensitive, placing empty newlines between the entries will make it so that your document no longer compiles. The code for the next tree is very similar, but it is hard to reuse this code.

Listing 3.18: rbtDocument.tex

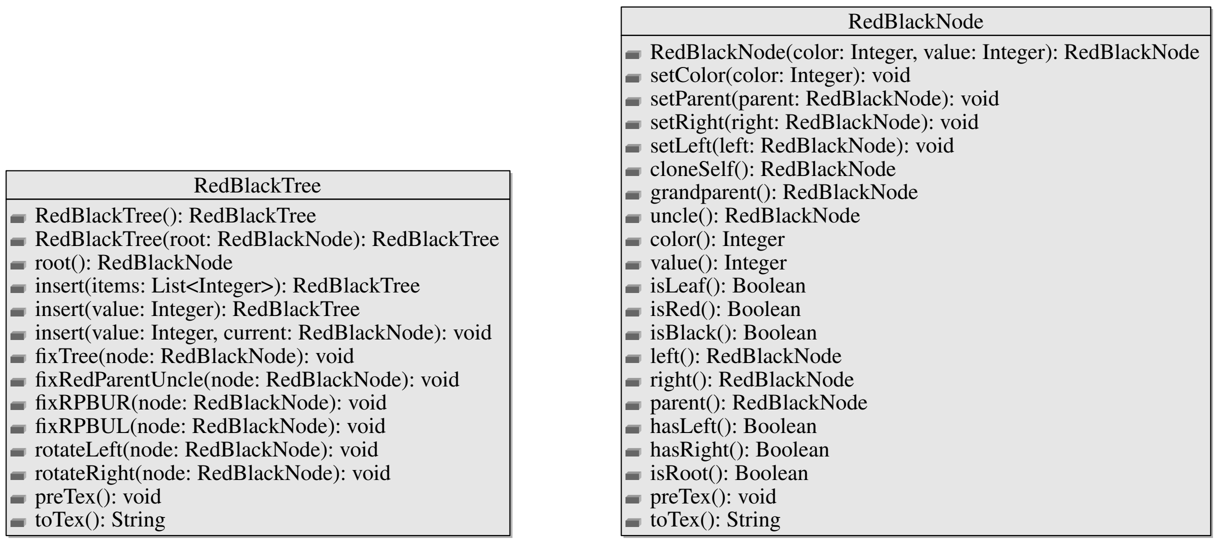
```
\begin{tikzpicture}[->,>=stealth',level/.style={sibling distance = 5cm/#1,level
    distance = 1.5cm}]
\node [arn_n] {20}
child { node [arn_r] {15}
child { node [arn_n] {10}
child { node [arn_r] {5}
}
child { node [arn_x] {}}
}
child { node [arn_n] {18}
}
}
child { node [arn_r] {38}
child { node [arn_n] {33}
child { node [arn_x] {}}
child { node [arn_r] {36}
}
}
child { node [arn_n] {47}
child { node [arn_r] {39}
}
child { node [arn_x] {}}
}
}
;
\end{tikzpicture}
```

Note that this tree is not optimally implemented, in our case when we append something to the tree, we completely clone it and then add a new node to this new instance. The red black tree in this example also only works for integers. The reason we clone the tree every time we insert a new value is because otherwise all three of the trees would all be shown exactly the same and they would all show the last version of the three.

3.2.6 MetaUML

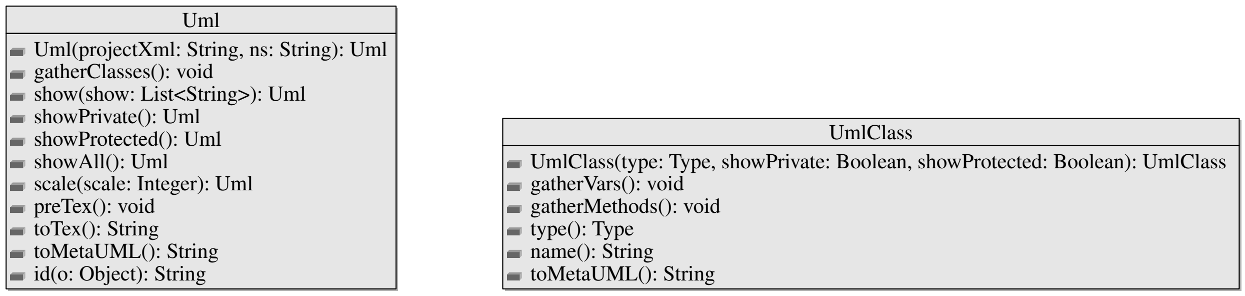
The next example has already been used a few times in this document. Every UML diagram up to now was created using a UML library implemented in Neio. It is also a little different from the other libraries we have seen thus far as it does not directly translate into LaTeX. Instead we're making use of MetaUML [14]. A high-level overview of how we create UML diagrams will be given here, but a lower level overview will be given in Chapter 4. We'll start out with an example.

```
[Document]
{
    new Uml('''./project.xml''', '''neio.stdlib.graph''')
}
```

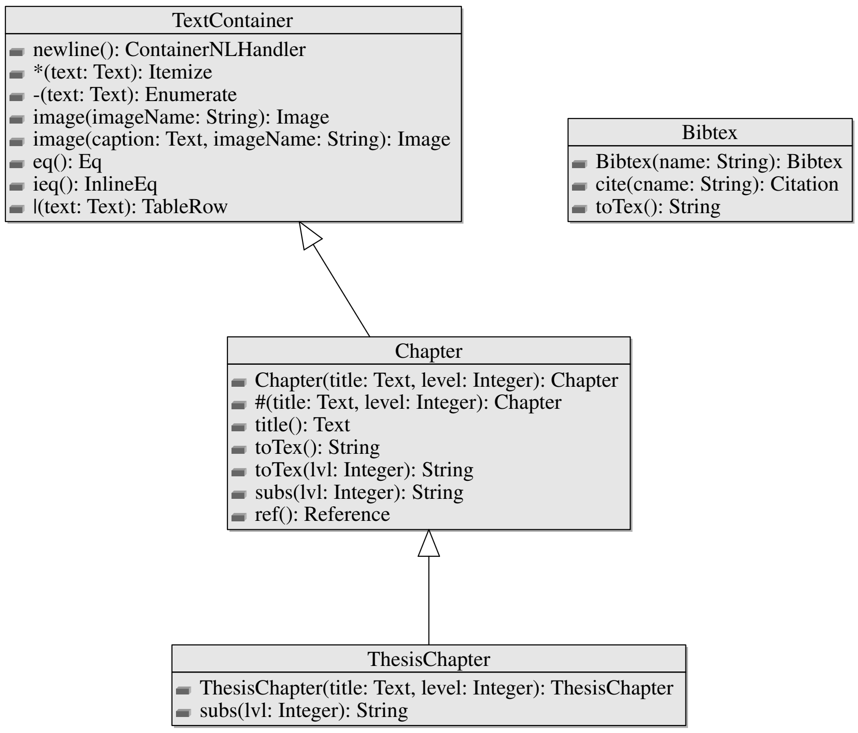

RedBlackTree
RedBlackTree(): RedBlackTree
RedBlackTree(root: RedBlackNode): RedBlackTree
root(): RedBlackNode
insert(items: List<Integer>): RedBlackTree
insert(value: Integer): RedBlackTree
insert(value: Integer, current: RedBlackNode): void
fixTree(node: RedBlackNode): void
fixRedParentUncle(node: RedBlackNode): void
fixRPBUR(node: RedBlackNode): void
fixRPBUL(node: RedBlackNode): void
rotateLeft(node: RedBlackNode): void
rotateRight(node: RedBlackNode): void
preTex(): void
toTex(): String
RedBlackNode
RedBlackNode(color: Integer, value: Integer): RedBlackNode
setColor(color: Integer): void
setParent(parent: RedBlackNode): void
setRight(right: RedBlackNode): void
setLeft(left: RedBlackNode): void
cloneSelf(): RedBlackNode
grandparent(): RedBlackNode
uncle(): RedBlackNode
color(): Integer
value(): Integer
isLeaf(): Boolean
isRed(): Boolean
isBlack(): Boolean
left(): RedBlackNode
right(): RedBlackNode
parent(): RedBlackNode
hasLeft(): Boolean
hasRight(): Boolean
isRoot(): Boolean
preTex(): void
toTex(): String

This code reads the `project.xml` file, that tells it what project we want to create UML for. The second parameter is the namespace the uml should be created for. It is also possible to only show a few classes, or to also show private and protected members. This is shown in the example beneath

```
[Document]
{
    List<String> list = new ArrayList<String>();
    list.add('''neio.stdlib.uml.Uml''');
    list.add('''neio.stdlib.uml.UmlClass''');
    new Uml('''./project.xml''', '''neio.stdlib.uml''').scale(100).show(list).
        showPrivate();
}
```



As the code for this library is over 600 lines long, we will not show all of the code. To get an image out of this, we output our object structure to the MetaUML format, instead of LaTeX. And then run it through the `mpost` command line program which generates TeX code for the UML, which we can then include. The problem with the MetaUML tool is that positioning is not done automatically. The way it was implemented in our case is a bit like a tree. We build a tree of levels, the first level contains all the classes that have no super class to be shown as well as the highest level classes. Subclasses are shown on the level beneath their super class. For example, `ThesisChapter` is a subclass of `Chapter`, which is a subclass of `TextContainer`. If we now show these three classes, they will be displayed one under the other because we built a tree with three levels. To visualize how the levels are being used, we show this example below but also add the Bibtex class to the UML , as it has no connection to the other classes.


TextContainer
newline(): ContainerNLHandler
*(text: Text): Itemize
-(text: Text): Enumerate
image(imageName: String): Image
image(caption: Text, imageName: String): Image
eq(): Eq
ieq(): InlineEq
|(text: Text): TableRow
Bibtex
Bibtex(name: String): Bibtex
cite(cname: String): Citation
toTex(): String
Chapter
Chapter(title: Text, level: Integer): Chapter
#(title: Text, level: Integer): Chapter
title(): Text
toTex(): String
toTex(lvl: Integer): String
subs(lvl: Integer): String
ref(): Reference
ThesisChapter
ThesisChapter(title: Text, level: Integer): ThesisChapter
subs(lvl: Integer): String

Note that our implementation of UML diagrams only shows what it is asked to show (hence there are no uses arrows) and only draws inheritance arrows for what is shown. This is done to avoid clutter.

3.2.7 Chemfig

3.2.8 Lilypond

Chapter 4

Implementation

Next to designing a language, a compiler to be able to actually use the language had to be build. How we implemented the compiler, as well as some lower level concepts used in Neio classes, will be explained in this chapter.

4.1 Used libraries and frameworks

To be able to build the compiler, we made few of a few open-source libraries and frameworks. They will be explained in this section. The number of libraries or framework that we used is rather limited, only three of them were used.

4.1.1 ANTLR4

The ANTLR4 [1] library is an open-source parser generator released under the BSD license. It is used to parse all of the Neio files, the Neio documents and the Neio classes. We used ANTLR4 in the following way. First we created two grammars, consisting of a parser and a lexer, for the Neio documents and classes, thus four files in total. These are written in ANTLR4's own DSL. These grammars are then fed to the `antlr4` command line tool, this tool generates Java classes using the grammars you defined. The ANTLR4 Java classes provide us with an interface that makes use of the Visitor pattern. Using the visitor pattern, we visit every rule and terminal for a given file. Whilst visiting these, we build an Abstract Syntax Tree (AST) of a given file which we can then manipulate further on. The ANTLR4 grammars and visitor classes thus form the front end of our compiler. The objects that are used in this AST, are objects defined in Jnome and Chameleon.

4.1.2 Chameleon and Jnome

Chameleon and Jnome are both projects that were developed by professor van Dooren and released under the MIT license. Chameleon [2] is a framework that allows you to model a programming language, or as we have shown in this Thesis, a markup language. It does so by providing a lot of objects that represent concepts that are commonly used in programming languages. A few examples of such objects are the following: `Expression, VariableDeclaration, Type, MethodInvocation, InfixOperatorInvocation,....`

Jnome [7] uses Chameleon and implements the front end and backend of a compiler specifically for the Java language. It can read Java and build an AST from it (using ANTLR3), thus it is a front end for the Java language. To be able to do this it extends the object model available in Chameleon with Java specific objects. It is also able to write out Java code, given an AST built from Jnome and Chameleon objects. This part is the backend of the compiler.

Jnome can read entire Java projects at once, in our compiler for example, it loads the entire Java library. This is needed because our objects use Java objects, such as String's. It will read all of the source files and libraries that are specified in a file called `project.xml`. This file was mentioned earlier in Subsection 3.2.6. This same file is used to read Neio projects in our compiler. The `project.xml` for the Neio compiler is shown below.

Listing 4.1: project.xml

```xml
<project name="Thesis">
    <language name="neio" />
    <baselibraries>
        <library language="Neio" load="true" />
    </baselibraries>
    <sourcepath>
        <source root="../../examples/0.8/lib" />
    </sourcepath>
    <binarypath>
        <jar file="../build/libs/rejuse.jar" />
        <jar file="../build/libs/chameleon.jar" />
        <jar file="../build/libs/ChameleonSupport.jar" />
        <jar file="../build/libs/jnome.jar" />
    </binarypath>
</project>
```

Having obtained an AST in Subsection 4.1.1, we now manipulate it in the middle end of our compiler. After the manipulations have been completed, we write it out using the Jnome back-end.

4.2 Compile flow

Now that all parts of the compiler have been introduced, we show a diagram that depicts the flow a source file from Neio source up to to the final LaTeX file.

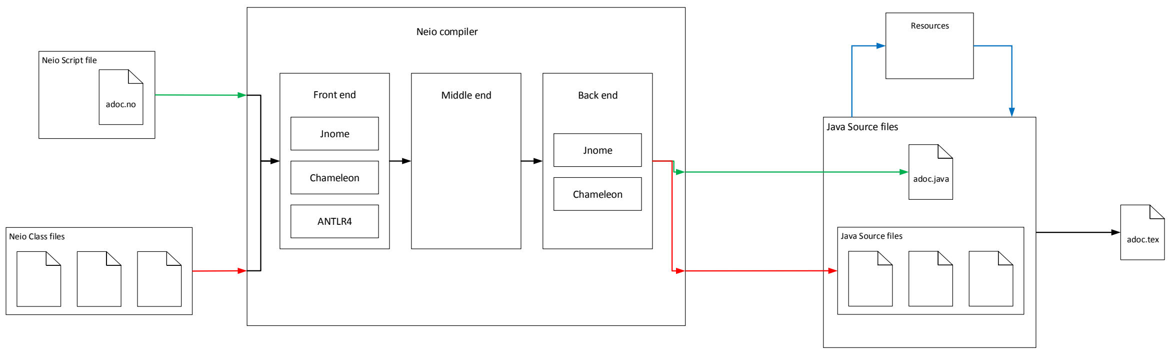


Figure 4.1: Illustration of how a Neio document is compiled. Neio class files translate to Java classes, Neio script files to a Java file with a main function. This then gets output to Tex using Neio's standard library

First of the Java and Neio library (the Neio class files) are read by the front end of the Neio compiler and translated into an AST. Then Neio documents are read by the front end and it is also transformed in an AST. The latter AST is past to the middle end of the compiler and is transformed to an object model that can be output to valid Java. Then the transformed AST and the AST's for the Neio library are passed to the compiler back end which will generate the final Java code. This final Java code will then be compiled and executed again. During the execution, resources might be created, if so, they can be used in the next step. This next step is the building of the document to a final representation. The way it always happens at is the time is as follows. The `TexFileWriter` is invoked with the root of the AST of the Neio document, this will call `preTex` and `toTex` on this root. `preTex` allows a document to prepare itself for being printed to LaTeX. A common preparation is to add a package that is uses to the root document. `toTex` does what is says and just creates a LaTeX representation of the object. Once that is completed, it will write out this TeX string to a file named the same as the Neio document but with the TeX extension. Finally the path to this new TeX file is passed to the `TexToPDFBuilder` which will compile the LaTeX code. The code for these two files is shown below.

Listing 4.2: TexFileWriter.no

```
namespace neio.io;

import java.nio.file.*;
import java.util.Arrays;
import neio.lang.Content;

/**
 * Recursively converts Content into a TeX string and prints it to a file
 */
class TexFileWriter implements Writer;

Content content;

/**
 * Initializes the writer with a root content
 *
 * @param content The root Content
 */
TexFileWriter(Content content) {
    this.content = content;
}

/**
 * Creates the TeX string
 *
 * @param name The name of the file to print to (without extension)
 * @return The path to the created file
 */
String write(String name) {
    content.preTex();
    String toWrite = "% Name: " + name + "\n" + content.toTex();
    Path path = Paths.get(name + ".tex");
    Files.write(path, Arrays.asList(toWrite.split("\n")));
    System.out.println("Wrote " + path.toAbsolutePath());

    return path.toAbsolutePath().toString();
}
```

Listing 4.3: TexToPDFBuilder.no

```
namespace neio.io;

import java.io.File;
import java.util.List;
import java.util.ArrayList;

/**
 * @author Titouan Vervack
 */
class TexToPDFBuilder implements Builder;
```

```

/**
 * Compiles a TeX file
 *
 * @param name The name of the file to compile (without extension)
 */
void build(String name) {
    if (name != null) {
        List<String> command = new ArrayList<String>();
        command.add("latexmk");
        command.add("-pdf");
        command.add("-dvi-");
        command.add("-bibtex");
        command.add("-lualatex");
        // Make sure the delimiters are correct
        command.add(new File(name).getAbsolutePath());

        ProcessBuilder pb = new ProcessBuilder(command);
        pb.redirectOutput(ProcessBuilder.Redirect.INHERIT);
        pb.redirectError(ProcessBuilder.Redirect.INHERIT);
        pb.start().waitFor();

        String texlessName = name.substring(0, name.length() - 4);
        name = texlessName + ".pdf";
        System.out.println("Wrote " + name);

        command.clear();
        command.add("latexmk");
        command.add("-c");
        command.add(name);

        ProcessBuilder pb2 = new ProcessBuilder(command);
        pb2.redirectOutput(ProcessBuilder.Redirect.INHERIT);
        pb2.redirectError(ProcessBuilder.Redirect.INHERIT);
        pb2.start();
        System.out.println("Cleaned files for " + texlessName);
    }
}
```

4.3 Translation

4.3.1 Reasons for choosing Java

The reason we chose for Java was multi-fold. First and foremost, we already had a front- and back end available for Java thus this saved us a lot of time. A second reason is because Java is platform independent which does not restrict us to a single operating system. Java is also very well known in the current world of computer science, which is why we chose to base our programming model on it. Since the Neio semantics now resemble the Java semantics so well, the reasons for going with Java are enforced.

4.3.2 Fluent interface

We tried to make use of fluent interfaces as much as possible, in the design of the language, by using method chains. But also in the libraries we implemented in Section 3.2. We chose to this as it imposes less boilerplate code (no constant variable declarations for example) and because it is clearer to read what we are doing. Instead of passing a ton of arguments to one method, we split it up in multiple methods and build up one object.

4.3.3 Outputting

Neio makes use of Context Types, which are not something that exists in other languages as of yet. This means we had to create a custom translation for it. The way we choose to translate it, is by breaking up the method chains created in the Text mode of a Neio document at every method call. A variable is then assigned to the output of this method call.

To know what to call our method on, `ContextType`'s were implemented as an extension of `RegularJavaType`'s in Jnome. A `RegularJavaType` is what we know as a Java class. This `ContextType` adds a virtual inheritance relation between itself and the actual type of the method, and one between itself and the ContextType of this method. The latter ContextType represents the Context at that point as we saw in Subsection 2.4.1.

When a lookup reaches a `ContextType` and asks us what our inherited members are (the members you normally receive from super classes), we return the members of all our inheritance relations. We only have two inheritance relations, the actual type and the ContextType. If what we are looking for was somewhere we could see, we will now have returned it and we are able to make use of it. Lookups are provided by Chameleon and explaining how exactly they work is out of the scope of this Thesis.

We also hold a Stack of all the variables that have already been declared. When a lookup for a method, described above, succeeds, we check what type is returned by this method and look for the first variable in our stack of that type, popping everything on the way there. We then know what to call our method on.

Lastly, we also have to translate `this`. The translation is analogue to the previous step, where we searched what to call the method on. To have a starting point, we just replace `this` with the last defined variable and let the `ContextType`'s do their job. A translation for the very first example we looked at in this document is shown below.

Listing 4.4: testInput.no

```
[Document]

# Chapter 1
This is some text in the first Paragraph
```

```
new Document()
    .newline()
    .newline()
    .#("Chapter 1")
    .newline()
    .text("This is some text in the
       first Paragraph");
```

Listing 4.5: testInput.java

```
package testInput;

import neio.fib.*;
import neio.io.*;
import neio.lang.*;
import neio.stdlib.*;
import neio.thesis.*;
import neio.stdlib.chem.*;
import neio.stdlib.graph.*;
import neio.stdlib.math.*;
import neio.stdlib.music.*;
import neio.stdlib.uml.*;
import java.util.*;

public class testInput {
    public static void main(String[] args) {
        Document input = new Document();
        createDocument(input);
        finishDocument(input);
    }

    public static void createDocument(Document input) {
        ContainerNLHandler $var0 = input.newline();
        ContainerNLHandler $var1 = $var0.newline();
        Chapter $var2 = input.hash(new neio.lang.Text("Chapter 1"));
        ContainerNLHandler $var3 = $var2.newline();
        Paragraph $var4 = $var3.text(new neio.lang.Text("This is some text in
           the first Paragraph"));
        ParNLHandler $var5 = $var4.newline();
    }

    public static void finishDocument(Document input) {
        java.lang.String $var0 = new TexFileWriter(input).write("testInput");
        new TexToPDFBuilder().build($var0);
    }

}
```

Note that the middle end has created three functions. The first one is the main function, it allows the document to be executed as a whole. It initialises the document class and then calls `createDocument` which actually creates an object model that will represent the document. Finally `finishDocument` will compile the document to TeX and create a PDF of of it.

The reason we split up this process is so that we can call the creation of the document separately. This allows us to include a document into another document by just calling the `createDocument` method with an appropriate argument.

The middle end also imported every namespace in the Neio library as well as the `java.util` namespace as it is possible that those will be needed. The name of the Java class that is generated is the same as the name of the Neio document.

It is important to note that we only split up the methods chains created in Text mode. The code written in code mode is kept as is, only occurrences of `this` are replaced.

4.3.4 Reflection

In Subsection 2.4.4 we touched on `addClassMapping` and said that the class mapping was comparable to a factory. This is true and the need for it appeared when we wanted to create drop in replacements for objects that are created in Neio Text mode. For example, if we want to use a special kind of `Chapter` in the `Document` document class, we would have to create a new document class that overrides the `#` method, we also have to override the nested `#` method in `Chapter` because otherwise only the first `Chapter` would be an instance of the special `Chapter`.

This is a problem that is normally solved by using factories. When you know your object will create instances of other objects, in our example a `Document` creates `Chapter`'s, that might be changed with more specialized forms of that object, you create a factory. This factory however would have to be generic enough to be able to be used almost anywhere. Even if we are able to do this, the writer of the specialized object would still have to create a new factory. Factories usually are also very similar to each other, yet the code can not be reused.

For this reason we chose to swap out new calls with calls that use reflection to initialize the object. This is a process that is executed in the middle end of the compiler. Only the new calls of objects extending `Content` have been replaced as those should be the only thing a user wants to replace. Every new call gets replaced by the `getInstance` call in the `Content` class. The code needed to understand this process, as well as an example call is given below.

Listing 4.6: Content.no

```
// Determines as what class new Content instances will be instantiated
private final Map<Class, Class> _classMapping;
/**
 * Returns the Class as which a Class should be instantiated
 *
 * @param klass The key of which we want to get a mapping
 * @return The mapping found for key {@code klass}
 */
final Class classMapping(Class klass) {
    return _classMapping.get(klass);
}

/**
 * Overrides default behaviour and instantiates new instances of {@code oldClass
    } as instances of {@code newClass}
 *
 * @param oldClass The old class object
 * @param newClass The new class object
 */
public final <T> void addClassMapping(Class<T> oldClass, Class<? extends T>
    newClass) {
    _classMapping.put(oldClass, newClass);
}

/**
 * Used instead of a new call. This allows to create more specific objects than
    originally
 * specified. e.g. this allows to create SpecialChapter's instead of Chapter's
 * <p>
 * It will create an instance of {@code klass} in case the class mapping has not
     been overridden.
 * If the mapping was overridden, an instance of the overriding class will be
    created.
 *
 * @param klass      The class to instantiate
 * @param paramTypes The types of the parameters
 * @param params     The parameters to instantiate {@code klass}
 * @return The instantiated object
 */
public final <T> T getInstance(Class<T> klass, Class[] paramTypes, Object[]
    params) {
    // Have I been overridden?
    Class result = classMapping(klass);
    Content current = parent();
    // Has any of the parents been overridden?
    while ((current != null) && (result == null)) {
        result = current.classMapping(klass);
        current = current.parent();
    }

    // No parents have been overridden
    if (result == null) {
        return instantiate(klass, paramTypes, params);
    }
    // A parent was overridden, use that type
    else {
        return (T) instantiate(result, paramTypes, params);
    }
}

/**
 * Does the actual instantiation of an object using Reflection
 *
 * @param klass      The class to instantiate
 * @param paramTypes The types of the parameters
 * @param params     The parameters to instantiate {@code klass}
 * @return The instantiated object
 */
private <T> T instantiate(Class<T> klass, Class[] paramTypes, Object[] params) {
    return klass.getConstructor(paramTypes).newInstance(params);
}
```

```
Chapter chapter = getInstance(Chapter.class, new Class[]{Text.class, Integer.
    class}, new Object[]{title, 1});
```

If you already understood the code from the comments, this paragraph can be skipped. Every instance of content has a mapping from `Class` to `Class`, by default this mapping is empty. When we want to use a specialized form of an object we call `addClassMapping`, as we did in Subsection 2.4.4. Let's say we have the following example. We added a mapping from `Enumerate.class` to `FibEnumerate.class` in the root `Document`. When we now call `getInstance(Enumerate.class, ...), this will be called on Chapter, not on Document`. But fear not, class mappings are looked up recursively. In the `getInstance` method we first check if we have a mapping for the class we are trying to instantiate. If we do the value of the mapping is used, if not we recursively check the mappings of our parents. If no mapping has been found after this, we just instantiate the class that has been given as argument to `getInstance`.

```
[Document]
# Chapter 1
{
    nearestAncestor(Document.class).addClassMapping(Enumerate.class,
        FibEnumerate.class);
}
```

4.3.5 Escaping

A Neio document is parsed and converted a number of times. First, it is parsed by the Neio compiler, then it is transformed into Java code that is then converted into LaTeX code (in most cases). This means that we have to handle escaping of characters in three different languages. Escaping in Neio is easy, just add a \ before the character you want to escape and it is escaped. Of course \\`[ntr]` hold on to their special meaning. However \\`[bfu]` have not been included in Neio as they are not yet needed. In the future it might prove useful to include these special characters to.

Moving on, we can't just translate these characters straight to Java as we have no special meaning for \\`[bfu]`. For this reason the middle end transforms every backslash into two backslashes, and transforms a double backslash (used to denote an escaped single backslash in Neio) to four backslashes in Java as this is how you represent an escaped backslash in a Java String. We then do some further translation in the Text class to assure that we are able to create valid latex. LaTeX only has ten special characters thus replacing is quite easy. The code to do the replacing is shown below.

Listing 4.7: Text.no

```
/**
 * Creates the TeX representation of this Text
 * Printing out {@code realText} or the TeX representation of {@code text}
 * Replaces special characters to create valid TeX
 *
 * @return The TeX representation of this Text
 */
String thisToTex() {
    String me = "";
    if (text == null) {
        // Create valid TeX text
        me = realText.replaceAll("\\\\([^#$%&~\\^_{}\\\\])", "$1");
        me = me.replaceAll("\\\\~", "\\\\textasciitilde{}");
        me = me.replaceAll("\\\\^", "\\\\textasciicircum{}");
        me = me.replaceAll("\\\\\\\\\\\\", "\\\\textbackslash{}");
        me = me.replaceAll("\\\\\\\\", "\\\\textbackslash{}");
    } else {
        me = text.toTex();
    }

    return me;
}
```

4.4 Resource usage and creation

4.5 Limitations

Whilst development of the compiler a few limitations were discovered, they are discussed in this section.

4.5.1 Windows

In Section 4.4 we saw that command line commands are sometimes executed. Executing commands using the Java Process class, however came with some not immediately visible issues. When developing on Linux, executing shell commands from Java worked perfectly fine and gave no problems whatsoever. However, on Windows some of the processes would hang, some would always hang, some would hang only once in a while. The reason for this is that Windows offers only a limited buffer size for the input and output streams of a process, the program deadlocks if the input is not written or read. After closer inspection of the Javadoc [5] this became apparent but due to the API for the Process class being so simple, it was a surprise nevertheless. The fact that we have to worry about different platforms on a cross-platform language as Java is also something that is not too common.

4.5.2 Back end

About every back end has some limitations on how we can translate our Neio document. This is due to the fact that Neio can use symbols as methods. In Java methods can only contain letters, numbers, $ or _. This means we have to translate the symbol function to valid Java identifiers. The way we do this is very straight forward. We use the following mapping and just replace any occurrence of a key in a method or method invocation with the value.

```
"#" -> "hash"
"*" -> "star"
"=" -> "equalSign"
"^" -> "caret"
"-" -> "dash"
"_" -> "underscore"
"`" -> "backquote"
"$" -> "dollar"
"|" -> "pipe"
```

With other back ends we would do the same, some languages allow you to overwrite operators but doing this might lead to unexpected side effects, thus just translating it this way seemed like the safest way to go.

Chapter 5

Future work

As mentioned before in Section 2.5 there are still a few things that could be done in the future. There are however a lot more things that could be done and we will discuss them in the rest of this chapter.

5.1 Automatisation

Automatic StringBuilder

Since we are constantly translating to TeX and other formats, we use `StringBuilder` very often. An example is almost every `toTex` method, we show the one defined in `Content` as an example.

Listing 5.1: Content.no

```
/**
 * Returns a String representing this Content and its children as TeX.
 * The string is build by recursively calling {@code toTex} on all of children
 * in this Content.
 *
 * @return The TeX string
 */
String toTex() {
    StringBuilder tex = new StringBuilder();
    for (int i = 0; i < content.size(); i = i + 1) {
        tex.append("\n").append(content.get(i).toTex());
    }

    return tex.toString();
}
```

In fact we use it so many that in the future it might be better to just replace String's by `StringBuilder`'s at all times. This would make it easier for people to write packages. The performance gain, or loss of this would have to be investigated though.

Automatic return of the object itself

Another pattern that is often encountered is that we often return the same object even when we haven't changed it. This is to be able to make use of fluent interfaces, as said before. An example from the `InlineEq` class is shown below.

Listing 5.2: InlineEq.no

```
/**
 * Creates a square root and adds it to this equation, returning this to allow
     further chaining
 *
 * @param root The base of the square root
 * @param arg  The value to take the square root of
 * @return This equation
 */
InlineEq sqrt(Content root, Content arg) {
    content().add(new Sqrt(root, arg));
    return this;
}
```

We actually already have a solution for this, constructors. Constructors do not tell us in their signature what they are going to return, but we do know what they are going to return. No explicit return is needed either. Implementing this for other methods would allow us to lower the amount of boilerplate code that has to be written.

Automatic Text conversion

The final automation is needed to counter what we saw with the math library. There we had to open an inline code block just to open Text mode because that creates a `Text`. A way to prevent this would be to automatically convert `Content`, or maybe even `Object` in a generic way. In Java this is done through the `toString` method while in python the `repr` and `str` methods are used for this. However if we were to implement it as naively as this, we would lose any kind of static typing when a Text is expected as an argument. This does not seem like a good thing, especially when you consider that not every object might have a real textual representation.

This means that automatically transforming an object to text might in a lot of cases not even create a sensible result.

A better way might be to define an interface, `Representable`, that has a `repr` method. Any class that implements this class, and the native types such as String and Integer, could than be transformed into a Text using this `repr` method.

5.2 Moving content

When we assign content to a variable, and apply small changes to it later on, it is possible that the content moves or that both the original and the new content are changed. This is because the language does not enforce every placed content to be static and in turn this is the responsibility of the library developer and the user of those libraries. We showed that we had to make sure that we cloned our objects when making changes to them before in Subsection 3.2.5 and Subsection 3.2.8. It would be a very nice feature if Neio could enforce this behaviour by itself and take care of the implementation of clone by itself. A way that this might be implemented is by copying objects that are going to be changed through reflection.

5.3 Use

5.4 Packages

Package support

No package system was provided in this thesis, but in the future however it would be beneficial to do so. It would allow users without a lot of technical expertise to import new functionality easily. The way packaging is handled right now is straight through LaTeX packages and Java packages. We allow you to add LaTeX packages through the `addPackage` method in `Document` and we allow you to import Java classes through the regular `import namespace` statements. To actually include a Neio library in the document, right now the source code for it has to be put into a folder, next to the rest of the standard Neio library. In the future we should be able to download (manually or automatically) packaged libraries and have them be stored somewhere else.

Implementation of packages

As we said before, non of the document classes or libraries that were created in this thesis are complete, and we also did not create that many of them, in comparison to what Java and LaTeX offer that is. In the future a lot of work should be put into writing complete libraries and porting libraries from other available languages such as LaTeX. This can however be done by the community for a big part, as is what happens for Java and LaTeX.

5.5 Compiler improvement

Efficiency

The compiler is fast enough for small documents but for a large document like this thesis, the compile time starts to ramp up. The thesis takes about 30 seconds to a minute to compile, depending on the compute power of your computer. This is on top of the `latexmk` that has to run afterwards and that also takes quite a while. However no attention was paid to the efficiency of the compiler in this thesis, as thus it is likely that there are a lot of optimizations that can be performed.

Other front- and back ends

Another compiler improvement that could be created in the future, is that we could create different front and back ends. We don't have to output to Java, if someone were to write a back end for python for example, we could make use of that back end. The same thing is possible for the front end. The syntax of Neio documents and class files could be changed, as long as the fundamental design decisions of the language are kept the same, such as `ContextType`'s and nested methods, another syntax could be designed. As long as a front end is developed that can parse a Neio document with a different syntax that can still parse the document into the same AST as is used at this moment, it is possible to change the currently used syntax..

5.6 Tool improvement

To effectively be able to use Neio, we need some improvements to the tools available right now. Chameleon provides support for the Eclipse IDE and in the future this could be used to improve our tooling.

Syntax highlighting

One of the base requirements to be able to work with a language is syntax highlighting. This is one of the features that Chameleon provides through its Eclipse integration. However it was not made for a language such as Neio where we represent a document as a series of method chains. At this moment the Eclipse integration treats the entire method chain as a whole instead of every method call separately. This would cause problems as everything between codeblocks would be colored the same and we would not be able to have different colouring for special methods, such as the symbol methods.

It should however work out of the box for Code mode and the class files.

Auto completion

A second feature that Chameleon offers and that is very useful when writing in a certain language, is auto completion. This should again work well in Code mode and class files, however it would be a bit harder in Text mode. This is because it would have to know when to auto complete and when it should just let us write. It might be best to hide the auto complete behind a hotkey while in Text mode. This still allows the user to get a list of methods he could use at a certain time, but would remove the annoyance of having a pop-up for auto completion come up at every word that is written.

IDE with preview

The last thing that would be a very nice to have for a markup language like this, would be an IDE that previews the document. This is not something that is supported by Chameleon and would thus require quite a bit more work. The compiler in its current state is also not ready for this, as said before it gets quite slow on large documents. As it would have to be constantly running in the background, it would use a lot of energy (which is problematic for laptops) and would be too slow to actually produce results in time.

Chapter 6

Conclusion and reflection

Reflection

After having used the language for a year, written a Thesis in it and written a complete compiler for it, we found that it might have been better to just use almost pure java syntax as a front end. We changed a few things, such as adding syntactic sugar for new calls and calling packages, namespaces. The reason for the latter was because in our eyes namespace was a more general time that better explained what this keyword did.

But by choosing different keywords, we ended up with name conflicts. At some point in development we wanted to make use of a class in a package called `namespace`. Of course this did not work out well as the parser did not know how to parse this. By changing the syntax as much as we did, it also became impossible to run Javadoc on our class files without making major changes to it.

Disallowing a substantial part of Java (anonymous classes, shorthand operators such as `++`, `+=`, ...) and so on also hurts the adoption rate of the language as it is a bit harder to convert a Java file into a Neio file.

We think that it would have been better to have just used pure Java syntax and add the `nested` and `surround` modifiers to it, as well as allowing for symbols in the method names. In the future we might then also add the proposition made in Subsection 5.1.

Conclusion

In the end we think we showed that is possible to create a healthy mix between user friendlyness and a powerful programming model. We showed that the language can be as easy to read, write and learn as Markdown. While it can also be as powerful and diverse as LaTeX whilst remaining quite legible. A lot of work can still be done, and should still be done to commercially use Neio, as shown in Chapter 5, but we think that good basis has been provided. The completion of this book should be enough proof of this.

Bibliography

[1] Antlr4. `http://www.antlr.org/`. Accessed: 18-05-2016.

[2] The chameleon framework. `https://github.com/markovandooren/chameleon`. Accessed: 18-05-2016.

[3] Stephen Checkoway, Hovav Shacham, and Eric Rescorla. Are text-only data formats safe? or, use this latex class file to pwn your computer. In *LEET*, 2010.

[4] Steve Checkoway, Hovav Shacham, and Eric Rescorla. Dont take latex files from strangers. 2011.

[5] Javadoc for process. `https://docs.oracle.com/javase/8/docs/api/java/lang/Process.html`. Accessed: 09-05-2016.

[6] Jmathtex: Tex in java. `http://jmathtex.sourceforge.net/`. Accessed: 08-05-2016.

[7] The jnome framework. `https://github.com/markovandooren/jnome`. Accessed: 18-05-2016.

[8] Donald E. Knuth and Michael F. Plass. Breaking Paragraphs into Lines. *Software - Practice and Experience*, 11:1119–1184, 1981.

[9] Koma-script guide. `http://texdoc.net/texmf-dist/doc/latex/koma-script/scrguien.pdf`. Accessed: 18-05-2016.

[10] Markdown homepage. `https://daringfireball.net/projects/markdown/`. Accessed: 07-05-2016.

[11] Markdown syntax. `https://daringfireball.net/projects/markdown/syntax`. Accessed: 08-05-2016.

[12] Mediawiki latex extension. `https://www.mediawiki.org/wiki/Extension:Math`. Accessed: 08-05-2016.

[13] Memoir user guide. `http://tug.ctan.org/tex-archive/macros/latex/contrib/memoir/memman.pdf`. Accessed: 18-05-2016.

[14] Metauml. `https://github.com/ogheorghies/MetaUML`. Accessed: 18-05-2016.

[15] Latex minipage. `http://www.sascha-frank.com/latex-minipage.html`. Accessed: 09-05-2016.

[16] Pandoc. `http://pandoc.org/`. Accessed: 08-05-2016.

[17] Till Tantau. *The TikZ and PGF Packages*.