# Mork

## Compiler tool for Java

Michael Hartmeier

# Mork: Compiler tool for Java

by Michael Hartmeier

*This document was generated 2012-08-17 (Mork 1.0.2-SNAPSHOT)*

# Table of Contents

# List of Figures

# List of Examples

# Chapter 1. Introduction

## Preface

**Prerequisites.** This manual assumes knowledge about Java and some understanding of regular expressions and context-free grammars.

**Status.** Mork is developed in my spare time. I consider Mork as beta code. It has been tested for Mork itself (Mork is bootstrapped) and the example applications - and nothing else. The syntax part is stable. The mapping part slowly matures.

**Conventions.** The following font conventions are used throughout this document

- A `typewriter font` indicates example code or shell commands.
- An *italic font* indicates that the given term is defined in the respective sentence. TODO: not completed yet.

**License.** Mork is licensed under the terms of the GNU Lesser General Public License. It is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. For a copy of this license, see the file `license.txt` which is part of the distribution.

**Contact.** Please send your feedback to `<michael.hartmeier@1und1.de>`! Mork has a web page at https://sourceforge.net/apps/trac/beezle/wiki/Mork

## Overview

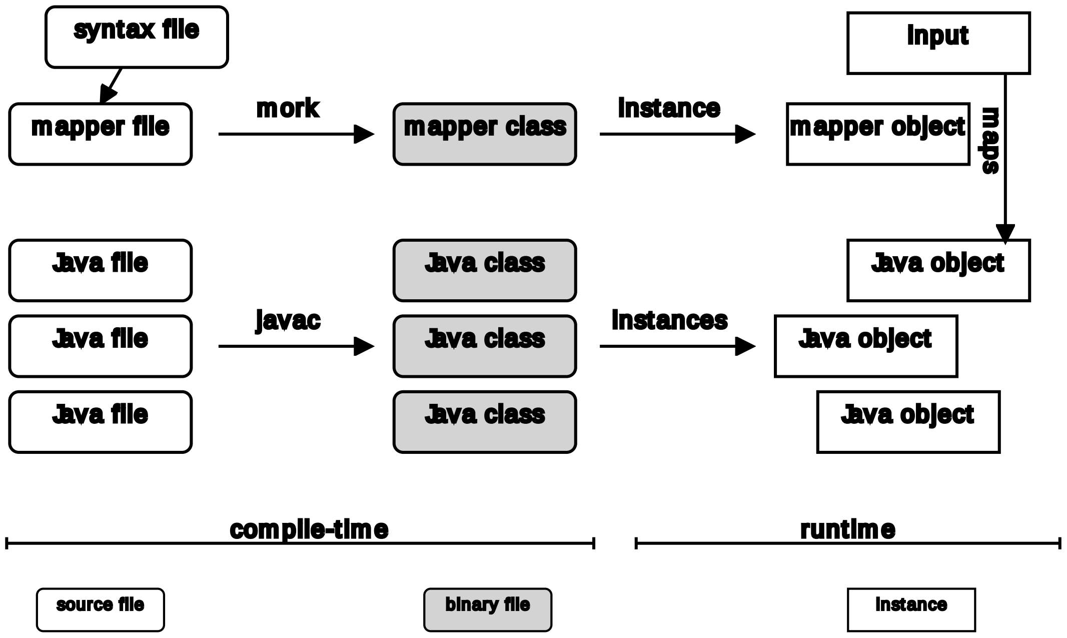
**Applications.** *Mork* is a compiler tool for Java. Its purpose similar to ANTLR, JavaCC (and jjTree), JLex & CUP, or SableCC. Mork can be used for *applications* that process input streams, e.g.

- **calculators.** computing expressions
- **pretty printers.** formatting program sources
- **interpreters.** executing scripts
- **compilers.** translating programs
- **tools like make.** implement some kind of command language
- **application configuration.** reading configuration files (e.g. Tomcat's `server.xml` file or manifest file for `jar`)

**Mapping.** Applications use Mork to map input streams into Java objects. Example 1: In a calculator application, Mork maps the input `17+5*7-10` into a number object `42`. Example 2: In an interpreter application, Mork maps a script file into Java objects like statements and expressions. The underlying statement and expression classes are defined by the application, they usually have methods like `execute` and `eval` that implement the actual execution if the script.

**Development.** Developing an application with Mork centers around the mapper concept . This is outlined in the figure below. The application is defined by three kinds of source files: (1) A syntax file defining the structure of the input stream, (2) a mapper file that defines how to map input streams with this syntax into Java objects, and (3) Java files defining the objects the input stream is mapped to. The application is compiled into Java class files by running a Java compiler on the Java files and my running Mork on the mapper file. At runtime, the application instantiates the mapper class and runs it on input streams in order to map them into Java objects.

**Figure 1.1. Development**



**Terminology.** In this document, the term application denotes an application that uses Mork. The term user refers to the developer who uses Mork to write an application; user does not refer to the person using the application. This manual forms the user documentation for Mork.

**Outline.** The next two chapters form a kind of user's guide, they describe how to install Mork (chapter 2) and how to get going (3). The remaining chapter are written more like a reference. Everything is centered around mappers: how to define the syntax of a mapper (4), how to define the mapper itself (5), how to translate a mapper (6) and how to invoke a mapper from your application (7). The last chapter (8) contains notes about example applications and the implementation of Mork.

## Features

- **Mapper.** Applications use a mapper as a black box: they just run it on some input stream to get their objects instantiated. This is like de-serializing objects from a file with arbitrary syntax
- **Mapping.** An attribute is defined by simply associating symbols with constructors. A set of attributes defines a mapper.
- **Visibility.** Visibility declarations can be used to make attributes available to almost any node in the syntax tree.
- **Scanner.** Unicode support, scanner modes, and the usual regular expression operators.

- **Parser.** LR(k) parsers and regular expression operators for grammar rules: EBNF and more.
- **Attribution.** Ordered attribute grammars.
- **No embedded code.** Clean separation of syntax and semantics: syntax goes into the specification file, semantics into Java files.
- **True compiler.** Mork generates class file, not Java code: no Java-compile step is needed.
- **Class file package.** To generate Java class files. Includes features like automatic stack size computation.
- **Examples.** Mork includes an interpreter example, a compiler example that generates bytecode, and a Java 1.2 parser example.
- **Positional error messages.** Throwing an exception reports the current position and the exception message.

# Chapter 2. Setup

**Prerequisites.** Mork needs Java 6 or higher. JRE is fine, you don't need the JDK.To checkout and run the example, you need Maven 3 [http://maven.apache.org/] and Subversion [http://subversion.apache.org/] installed. Note that the example have no scripts for windows, you have to manually launch them.

**Install.** There's no Mork distribution to download or install. Mork is published to Maven central, you'll just add a dependency to your project (as described later).

# Chapter 3. First steps

**Quick start.** A simple introduction for compiler tools is a calculator to compute expressions like `68/2+8` . Mork provides a calculator example , you can compile and run it as follows:

```
svn export https://beezle.svn.sourceforge.net/svnroot/beezle/mork/trunk/
cd calc
mvn clean package
./target/calc
```

**Example.** This chapter is built around an example called `command` . The above calculator example is not suitable because it demonstrates the most basic features only. `command` is simple enough to serve as an introduction and it is as complex as necessary to demonstrate the important features.

**Overview.** The next section explains how to compile and run `command` without digging into the sources. This is followed by various sections that pick the interesting aspects of `command` and explain them in more detail. Note that this chapter is an introduction, exhaustive coverage follows in the remaining chapters.

## Get going

**`command`.** `command` adds GUI front-ends to command line tools like `javac`, `jar`, or `mork`. To do so, the form of valid command lines is described in a command file. A command file describing Java's `jar` command line tool is shown in the listing below. The description consists of three things:

- a command name: `create jar`
- variable declarations: `jar` for the name of the jar file to create and `dir` for the directory to be zipped.
- a command line: `run "jar cvf " + jar + " " + dir;` defines how to combine variable values into the command line to invoke `jar`

Note: To keep things simple, only a fraction of `jar`'s functionality is provided. Most notably, you can create jar file only, extracting data is not available.

### Figure 3.1. jar command file

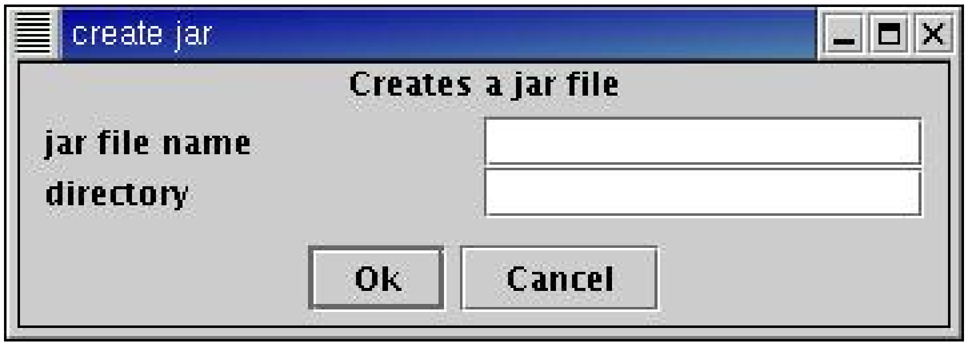
```
command "create jar": "Creates a jar file"
    jar: "jar file name";
    dir: "directory";
{
    run "jar cvf " + jar + " " + dir;
}
```

Get the command example, build and run it:

```
svn export https://beezle.svn.sourceforge.net/svnroot/beezle/mork/trunk/
cd command
mvn clean package
```

```
./target/command src/commands/jar.cf
```

**Figure 3.2. jar command screen shot**



**Front-ends.** Running `command` on a command file opens a window that provides a front-end to the respective command line tool. The front-end has fields for all variables declared in the command file and provides two buttons to start the command or to cancel without running the command. The following figure shows the front-end corresponding to the above command file.

**Source code.** `command`'s source code contains mostly Java files. Most of the Java files will be touched in the course of this chapter. Maybe the two most important Java files are `Main.java` which implements the overall logic and `Command.java` which defines the `command.Command` class. The largest Java files are `Console.java` (implementing a console window for command output) and `Frontend.java` (implementing the initial Swing dialog).

**Figure 3.3. Source files**

```
pom.xml
src/main/Command.syntax
src/main/Command.mapper
src/main/java/Command.java
src/main/java/Console.java
src/main/java/Constant.java
src/main/java/Declarations.java
src/main/java/Expression.java
src/main/java/Failure.java
src/main/java/Frontend.java
src/main/java/Line.java
src/main/java/Main.java
src/main/java/Reference.java
src/main/java/Variable.java
```

Besides Java files, the source directory contains a syntax file `Command.syntax` and a mapper file `Command.mapper`. The syntax file defines the syntax of command files and the mapper file defines how to map a command files into an instance of `command.Command`.

## Overall logic

**Figure 3.4. command.Main**

```
package command;

import net.sf.beezle.mork.mapping.Mapper;

public class Main {
    public static void main(String[] args) {
        Mapper mapper;
        Object[] tmp;
        Command command;

        if (args.length != 1) {
            System.out.println("command: add frontends to command line tools");
            System.out.println("  usage: command.Main <command file>");
        } else {
            mapper = new Mapper("command.Mapper");
            tmp = mapper.run(args[0]);
            if (tmp == null) {
                // runOrMessage has issued an error message
                System.exit(1);
            }
            command = (Command) tmp[0];
            command.run();
        }
        System.exit(0);     // just returning doesn't kill the gui threads
    }
}
```

`Main.java` defines the `main` method which implements the overall logic of `command`. `main` demonstrates the typical steps to use a mapper:

1. Instantiate the mapper class generated by Mork:

   ```
   mapper = new Mapper("command.Mapper");
   ```

2. Use the mapper object to read the command file specified on the command line:

   ```
   tmp = mapper.run(args[0])
   ```

3. Running the mapper turns the command file into an instances of command.Command:

   ```
   command = (Command) tmp[0];
   ```

4. Run the command, i.e. open the front-end window and execute the command line built from the end-user's input.

```
                          command.run();
```

## Syntax files

**Syntax files.** `Command.syntax` defines the syntax of command files. A syntax file has a scanner section and a parser section.

### Figure 3.5. `Command.syntax`

```
[PARSER]

Command      ::= "command" String ":" String Declarations "{" Line "}" ;
Declarations ::= Variable* ;
Variable     ::= Identifier ":" String ";" ;
Line         ::= "run" Expression ("+" Expression)* ";" ;
Expression   ::= Constant | Reference;
Constant     ::= String ;
String       ::= Literal ;
Reference    ::= Identifier ;

[SCANNER]
    white = Space, Comment ;

Letter       ::= 'A'..'Z' | 'a'..'z';
Digit        ::= '0'..'9';
Space        ::= ('\u0020' | '\b' | '\t' | '\n' | '\f' | '\r' )+;
Comment      ::= '#' '\u0020'..'\u007f'* ('\n'|'\r');
Literal      ::= '"' ( ('\\' 0..65535) | ('\\' | '"' | '\n' | '\r')!)+ '"';
Identifier   ::= Letter (Letter | Digit)* ;
```

**Syntax.** The syntax is specified in terms of symbols. Scanner and parser section give a list of rules. A rule has a left hand side and a right hand side, separated by `::=`. It defines a symbol by specifying how the symbol on the left hand side is composed of symbols on the right hand side. Running the mapper creates a syntax tree according to these rules, where each node represents a symbol. Usually, the leafs of the syntax tree are defined in the scanner section, whereas the inner nodes are define in the parser section. The first rule in the parser section defines the root node of the syntax tree.

## Mapper files

**Mapper files.** `Command.mapper` defines how to turn command files into instances of `command.Command`. Running the mapper reads a command file, checks for the appropriate syntax (defined in the syntax file) and maps the file into an instance of `command.Command`

**Figure 3.6. `Command.mapper`**

```
mapper command.Mapper;

syntax = "Command.syntax";

import command:
    Command, Declarations, Line, Constant, Reference, Variable;
import net.sf.beezle.mork.semantics:
    BuiltIn;

Command         => Command;
Declarations    => Declarations:
                     > \\Command
                     > \\Command//Reference;
Variable        => Variable;
Line            => Line;
Reference       => Reference;
Constant        => Constant;

Identifier      => [text];

String          => BuiltIn.parseString;
Literal         => [text];
```

**Mapping.** A mapper file is comprised of a list of attributes. The basic form of an attribute associates a symbol (on the left hand side) with a constructor (on the right hand side). Whenever the specified symbol is found in some input stream, the associated constructor is triggered. Here are two attributes used in the `command` application.

```
Command         => Command;
String          => BuiltIn.parseString;
```

**Runtime.** At runtime, a mapper first creates a syntax tree of the input stream. If this step is successful (i.e. there are no syntax errors), the mapper walks through the nodes of the syntax tree and invokes the constructors associated for the respective symbol. Example 1: The symbol `Command` triggers invocation of the constructor `Command` which results in an new instance of the class `command.Command`. Example 2: The symbol `Identifier` triggers invocation of the constructor `BuildIn.parseString` which results in a new instance of the class `java.lang.String`. A mapper returns the object(s) created for the root node. In the `command` example, this is a `command.Command` object.

**Constructors.** The right hand side of an attribute specifies a constructor. A constructor may be a Java constructor (e.g. `command.Command`), a Java method (e.g. `net.sf.beezle.mork.semantics.BuiltIn.parseString`), or an internal constructor. Thus, the term constructor is used in a more generalized meaning.

**Internal constructors.** A common requirement for symbols like `Identifier` is to access the actual input matched by the symbol. Example: the symbol `Identifier` might match the input `jar`. For this purpose, Mork provides internal constructors. `command` use the internal constructor `[text]` to access the actual input matched by the symbol. (There are more internal constructors besides `[text]`, but in most cases, `[text]` is all you need).

```
Identifier      => [text];
Literal         => [text];
```

Note: internal constructors are restricted to symbols defined in the scanner section; associating them with symbols defined in the parser section results in an error.

**Import and package.** Import and package declarations in the mapper file are similar to their Java counter-parts. As demonstrated by the attributes for `Command` and `String` symbols, imported classes can be specified without a package name. However, there is one important difference from Java's import declaration: all classes have to be imported explicitly. Example: all classes from `command` are explicitly imported, even though the mapper resides in the same package. Similarly, classes from `java.lang` have to be imported explicitly. Explicitly also means that Mork does not provide something like `import foo.*`.

## Visibility

**Argument passing.** In the previous section, you have seen how to associate symbols with constructors in order to create objects. Constructors take arguments. How do you define the arguments passed to a constructor? The answer is visibility. If you now other compiler tools it might take some time to get used to the visibility idea. But it's the feature that makes mapper files simple and powerful.

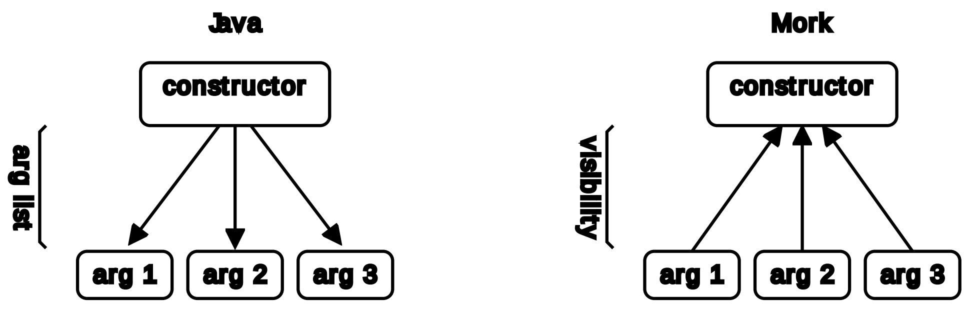
**Java argument passing.** Let's have a look at argument passing in Java. Consider the following Java fragment, where `name`, `decls` and `line` are variables holding appropriate values:

```
new Command(name, decls, line)
```

This expression invokes the constructor of the `command.Command` class. The arguments passed to the constructor are specified by an argument list. In Java, an argument list defines the arguments passed to a constructor. OK, that's obvious. But read on ...

**Mork argument passing.** In Mork, visibility defines the arguments passed to a constructor. In some sense, visibility turns Java's argument lists upside down: argument lists are tied to the constructor and kind of pull arguments into the constructor. In contrast, visibility is tied to an argument and kind of pushes it toward an constructor. This is illustrated in the graphic below.

**Figure 3.7. Argument passing**



**Visibility.** Each attribute has a visibility which defines the constructors the attribute is passed to as an argument. Visibility can be defined explicitly and implicitly. Look at the listing below, it shows two attribute definitions from `command`'s map file. Visibility of the `Declarations` attribute is defined explicitly by a colon `":"` and two paths `\\Command` and `\\Command//Reference`. In contrast, the `Command` attribute has implicit visibility since there is no explicit visibility defined.

```
Command         => Command;
Declarations    => Declarations:
> \\Command
> \\Command//Reference;
```

**Implicit visibility.** An attribute with implicit visibility is passed to the constructor attached to the nearest symbol in the syntax tree above. Example: `command.Command`'s constructor takes three arguments: two Strings, an instance of `command.Declarations` and an instance of `command.Line`. All of the attributes involved (`String`, `Declarations`, and `Line`) have implicit visibility, and the nearest constructor in the syntax tree above is `command.Command`. Consequently, these are the attributes passed `command.Command`.

**Explicit visibility.** Implicit visibility is appropriate in most situations. Explicit visibility enters the scene if an object has to be passed to a constructor other than the constructor in the syntax tree above. Explicit visibility is defined by a list of paths. Attributes with explicit visibility are passed to all constructors defined by the paths. The need for explicit visibility typically arises when resolving identifiers -- which will be studied in the respective section below.

**Argument order.** Visibility declarations do not define the argument order. If two attributes are visible to a given constructor, which attribute is passed first? Basically, Mork computes the argument order by iterating the list of formal arguments, passing the left-most attribute assignable to the current formal arguments. Consider `command.Command`'s constructor: the formal argument list is `(String name, String description, Declarations decls, Line line)`. The list of visible attributes are `String`, `String`, `Declarations` and `Line`. Note that this list is ordered according to the order of attributes in the syntax tree (which mostly resembles the grammar rule `Command ::= "command" String ":" String Declarations "{" Line "}" ;`). `String` argument occurs twice since the `String` symbols has two occurrences in the grammar rule. Argument ordering is performed as follows: The left-most attribute which is assignable to `name` is the first `String` attribute, thus it is assigned to `name`. After this, the second `String` attribute is passed since it is the only remaining attribute which is assignable to the formal argument `String description`. Similarly, the attribute `Declarations` is assigned to the formal argument `decls` and the attribute `Line` is assigned to the formal argument `line`.

Note that the argument order is computed at compile-time, i.e. when translating the map file. An error is reported if there is no valid argument order.

**List arguments.** The attribute `Variable` has default visibility, thus it is added to the argument lists passed to the `command.Declarations` constructor. The type of this argument is `Variable*` where `*` indicates a list argument. `Variable` is a list argument since a given syntax tree might have any number of Variable attributes. A List arguments is assignable to `java.util.List` or arrays of the respective base type. TODO: more about

**Option arguments.** TODO: change command to include an example for optional arguments. TODO: more about optional arguments.

**Argument merging.** Alternative arguments are merged. Example `Constant` and `Reference`. TODO: more about this.

# Error handling

**Errors.** `command`'s `main` method uses `Mapper.run` to read command files. This method reports syntax errors and semantic errors to System.err. A syntax error is reported whenever the syntax of the input stream read by the mapper does not match the syntax specified in the syntax file underlying this mapper. Semantic errors are reported whenever one of the constructors throws a checked exception.

**Semantic errors.** `command` makes sure that variable names are unique. If two variable have the same name, a semantic error is reported. To see this message, just duplicate the declaration for the `jar` variable in `tests/jar.cf` and run `command` on it:

```
examples/command > cat tests/jar.cf
command "create jar"
jar: "jar file";
jar: "jar file";
dir: "directory";
{
run "jar cvf " + jar + " " + dir;
}
examples/command > java command.Main tests/jar.cf
tests/jar.cf:2:5: semantic error: duplicate variable name: jar
examples/command >
```

**Defining.** Defining semantic errors is straight forward: check for the error situation in the appropriate constructor and, if a problem is detected, throw a checked exception. Whenever a mapper encounters a checked exception, it issues a semantic error message, reporting the current source positions and the string obtained from `Exception.getMessage()`. The current source position is the position of syntax tree node, the constructor was invoked for.

**Duplicate Identifier.** `command`'s semantic error `duplicate variable name` is realized in the file `Declarations.java` listed below. The constructor takes an argument with all variables declared in a given command file. After storing these variables in a member variable, `checkDuplicates` is invoked to look for a name that is used more than once. If a duplicate name is found, it throws a `Failure` exception. The `Failure` class extends `java.lang.Exception`, thus it is a checked exception. When reporting this error, the exception message `duplicate variable name` is used to describe the problem. The position reported is the position of the `Declarations` symbol.

```
package command;
```

**Figure 3.8. `Declarations.java`**

```
public class Declarations {
    private Variable[] vars;

    public Declarations(Variable[] vars) throws Failure {
        this.vars = vars;
        checkDuplicates();
    }

    /**
     * Throws Failure if there are multiple variables with the same name.
     */
    private void checkDuplicates() throws Failure {
        int i;
        Variable v;
        String name;

        for (i = 0; i < vars.length; i++) {
            v = vars[i];
            name = v.getName();
            if (lookup(name) != v) {
                throw new Failure("duplicate variable name: " + name);
            }
        }
    }

    public Variable lookup(String name) {
        int i;

        for (i = 0; i < vars.length; i++) {
            if (vars[i].getName().equals(name)) {
                return vars[i];
            }
        }
        return null;
    }

    public boolean runFrontend(String title, String description) {
        Frontend frontend;
        boolean result;
        int i;

        frontend = new Frontend(title, description, vars.length);
        for (i = 0; i < vars.length; i++) {
            frontend.setLabel(i, vars[i].getLabel());
        }
        result = frontend.run();
        for (i = 0; i < vars.length; i++) {
            vars[i].set(frontend.getValue(i));
        }

        return result;
    }
}
```

The next section includes another example for semantic errors.

## Resolving identifiers

**Meta.** Resolving identifiers can be seen as an advanced topic, but it is a common task within applications that use Mork. Almost every compiler or interpreter resolves variable names in one way or another.

**Example.** The `command` example has to resolve identifiers to evaluate the command line specified in a command file. Consider the jar command file listed in the first section. It includes the command line

```
run "jar cvf " + jar + " " + dir;
```

To evaluate this line, `command` has to find the variables referenced by the identifiers `jar` and `dir`. This is called identifier resolution. `command`'s identifier resolution is implemented in the files `Line.java` and `Reference.java`.

**Line objects.** Before digging into identifier resolution, I have to explain the `Line` class. See the listing below. Instances of this class represent command lines like `run "jar cvf " + jar + " " + dir;`. A `Line` is characterized by an array of exceptions. This array is passed to the constructor and stored in the field `exceptions`. The command line `run "jar cvf " + jar + " " + dir;` is an array of four expressions: two constants (`"jar cvf "` and `" "`) and two variable references (`jar` and `dir`). Constants are represented by instances of `command.Constant` and variables are presented by instances of `command.Reference`. Both classes extend `command.Expression` and thus have an `eval` method. The whole purpose of `Line` is to provide another `eval` method; `Line`'s `eval` is used in `Command.execute` to compute the string which is used to launch an operating system process. `Line`'s `eval` method simply evaluates all expressions and concatenates the results.

**Figure 3.9. `Line.java`**

```
package command;

/**
 * A command line is a sequence of expression.
 */

public class Line {
    /**
     * Concatenating these expressions forms the command line
     * to be executed.
     */
    private final Expression[] expressions;

    public Line(Expression[] expressions) {
        this.expressions = expressions;
    }

    public String eval() {
        StringBuffer buffer;
        int i;

        buffer = new StringBuffer();
        for (i = 0; i < expressions.length; i++) {
            buffer.append(expressions[i].eval());
        }
        return buffer.toString();
    }
}
```

**References.** `command.Reference` is shown in the listing below. A reference is nothing but a pointer to the variable it refers to. When asked to evaluate itself, it just returns the current value of the variable. The interesting part of the `Reference` class is its constructor. The constructor expects two variables, a declarations object and an identifier. What happens inside is straight forward: the declarations object is searched for the identifier. If the is a variable with the specified name, it is stored in the `var` field and the identifier resolution has succeeded. Otherwise, a semantic error `unknown identifier` is issued by throwing an exception.

**Figure 3.10. `Reference.java`**

```
package command;

/** Variable reference. */

public class Reference extends Expression {
    /** variable referenced by this expression. */
    private Variable var;

    public Reference(Declarations decls, String identifier) throws Failure {
        var = decls.lookup(identifier);
        if (var == null) {
            throw new Failure("unknown identifier: " + identifier);
        }
    }

    @Override
    public String eval() {
        return var.get();
    }
}
```

**Argument passing.** The constructor of `command.Reference` expects two arguments: a `Declarations` object and a `String` object. As explained in the visibility section above, the arguments passed to a constructor are determined by visibility definitions. The following visibility definitions are relevant here:

```
Declarations    => Declarations:
> \\Command
> \\Command//Reference;
Identifier      => String;
```

The `Identifier` attribute has implicit visibility. Consequently, it is passed to the nearest constructor in the syntax tree above - which is `command.Reference`. The `Declarations` attribute has explicit visibility, defined by two paths. The first path `\\Command` specifies to pass the attribute up to `command.Command`'s constructor - which is not relevant here. The second path `\\Command//Reference` specifies to pass the attribute up to the `Command` symbol and from there down to any `Reference` constructor. Thus, `command.Command` obtains both a `String` and a `Declarations` object and all it's arguments are satisfied.

**Paths.** In general, a path specifies a way through the syntax tree. `\\` passes an attribute up in the syntax tree until the specified symbol is found, and `//` passes an attribute down until the specified symbol is found.

TODO: some words about optional arguments and lists of arguments.

## What's next

This chapter introduced the most important features. There are several ways to go on now:

- **Extend the `command` example.** Interesting stuff is missing in `command`:
    - Beautified dialogs

  - More input elements like browse buttons or check boxes, e.g. to switch between creating a jar file and extracting data from a jar file.
  - Instead of launching an operating system process, a command could invoke static Java methods using reflection.

- **Study more examples.** Have a look at the examples directory. Both the interpreter and the compiler example are more complex than `command`.
- **Start a new application.** This is probably the most challenging way to continue. In particular, visibility declarations are more difficult than you might expect from looking at examples that run out of the box. In addition, the documentation in the remaining chapters is still incomplete. In general, I suggest starting new applications with a small syntax and just few classes you map to. If the small mapper work, continue by adding a single class and growing the syntax/mapping accordingly. Caution: if you first develop the complete syntax, then implement all your classes and then try to define a mapping, its much more difficult to solve visibility problems.
- **Go on reading.** The remaining chapter in this manual provide in depth information about Mork. In my opinion it is more appropriate to just start using Mork an come back later.

# Chapter 4. Syntax files

**About.** This chapter is about syntax files. A syntax file defines the structure of an input stream. Mork reads syntax files when processing a mapper file. The syntax is defined by a list of BNF-style rules. The list of rules is divided into a parser section defining a pushdown automaton and a scanner section defining a finite automaton.

**Naming convention.** My naming convention for syntax files is as follows. The base name of a syntax file is the name of the language specified in the file, capitalized like Java class names. The file name extension is `.syntax`. Examples: `Java.syntax` for a Java syntax.

## Overall structure

**Figure 4.1. Overall structure**

```
Syntax              ::= Parser Scanner ;
Parser              ::= "[" "PARSER" "]" Rule+ ;
Scanner             ::= "[" "SCANNER" "]" Priorities WhiteOpt Rule* ;
Priorities          ::= UsePriorities | NoPriorities;
UsePriorities       ::= ;
NoPriorities        ::= "nopriorities;";
WhiteOpt            ::= ("white" "=" SymbolSet ";")? ;
SymbolSet           ::= ( Symbol ("," Symbol)* )? ;
```

Start symbol of a syntax file is `Syntax`.

**Sections.** A syntax file is divided into a parser section and a scanner section. Sections have a header and a body. Headers adjusts global options that apply to the whole body. The body is comprised of a set of rules. The rules within both sections should proceed from large element to small elements, i.e. the start symbol should be specified first, and terminal symbols should be specified last. For that reason, the parser section precedes the scanner section.

**Parser section.** The parser section defines the context-free structure of the input stream. The section header identifies that parser section, no other options are given. The section body specifies a list of rules, each of them comprised of a symbol and a regular expression. These rules define the context-free grammar that parses the input stream. The symbol of the first rule is the start symbol of the grammar.

**Scanner section.** The scanner section defines the regular structure of the input stream. The section header specifies white symbols, i.e. terminal symbols to be removed from the input stream; these symbols are never returned to the parser. Typical examples for white symbols are white space and comments. The section body is comprised of a list of rules, each of them comprised of a symbol and a regular expression.

**Priorities.** Specifies how to resolve conflicts between two terminals that match the same input. A typical example is a keyword like `begin` and an `identifier` terminal that also matches the input `begin`. Traditionally, scanner generators use priorities to resolve this conflict: if the input `begin` is found, the terminal defined first will be matched. This behavior is useful for programming languages like Java where a keyword is not a valid identifier: just define the keywords before the identifier and you scanner will never match an identifier if there is also a keyword for the respective input.

**No Priorities.** Specify `nopriorities` if you don't want to resolve scanner conflicts based on their priority. In this case, Mork will try to resolve the conflict by choosing appropriate scanner modes. Example: in XML, the `Name` terminal conflicts with the `Nmtoken`. This conflict must not be solved by priorities because the scanner needs to match both terminals. Mork ensures that no parser state can shift both terminals and adds both terminals to different scanner modes. The generated parser automatically switches it's scanner into the mode that matches all terminals currently shiftable.

## Symbols

The syntax of input streams is defined in terms of symbols. Any symbol has a language which is defined by the parser and scanner section. A language is a set of strings. The set might be empty, it might include the empty string, and in may cases, the set in infinite. A symbol is said to match the current input, if the current input starts with an element of its language. Otherwise, the symbol does not match. Example: a symbol `Integer` typically matches `57...`, but not `xyz...`.

**Figure 4.2. Symbols**

```
Symbol              ::= StringSymbol | IdentifierSymbol ;
StringSymbol        ::= StringLiteral ;
IdentifierSymbol    ::= Identifier ;
Rule                ::= Subject "::=" RegExpr ";" ;
Subject             ::= Symbol ;
```

**Symbols.** A symbol is a string symbol or an identifier symbol. The language of a string symbol is defined implicitly by its string, whereas the language of an identifier symbol is defined explicitly by one or more regular expressions. Symbols are case-sensitive.

**String symbol.** A string symbol is specified like a Java string literal. The language of a string symbol is the string itself. For example, a string symbol `"foo"` matches the string `foo`, nothing else. String symbols may contain all escapes known from Java. It is an error to use different escapes to denote the same string. For examples, Mork issues an error if you use both `"foo\n"` and `"foo\x0a"` in a syntax file.

**Identifier symbols.** An identifier symbol is specified like a Java identifier. For any identifier symbol used in the parser section, the must be at least one rule. Let `A ::= w1; A ::= w2; .. A ::= wn;` be the set of all rules with subject `A`. The language of `A` is defined by the regular expression `(w1) | (w2) ... | (wn)`.

**Rules.** Rules define identifier symbols. The symbol on the left side of a rule is called the subject of the rule. The subject must be an identifier symbol. Regular expression in the parser may refer to any symbol, include those defined later in the parser section and to symbols defined in the scanner section. Regular expressions in the scanner section may not refer to symbols in the parser section; they may refer to symbols in the scanner section as long as not recursion is introduced. The same subject may be defined by multiple rules, but all rules must be specified in the same section.

**Inline symbols.** A symbol defined in the scanner section and not referred from the parser section is called an inline symbol. Inline symbols are used to simplify rules in the scanner section (by factoring out common parts), they are not considered terminals. As a consequence, inline symbols cannot be have attributes.

**Conflicts.** A conflict arises if a given input can be matched by two or more symbols. The parser section is tested for conflicts at compile-time. If Mork does not report conflicts, all conflicts, if any, have been resolved.

**Conflicts on nonterminals.** Mork issues an error message for all conflicts on nonterminal symbols. The user must change the rules in the parser section to resolve the conflict. There is currently no way to resolve conflicts by additional declarations, e.g. operator precedence.

**Conflicts on terminals.** In contrast to nonterminals, all conflicts on terminal symbols are resolved automatically. This conflict resolution follows the scheme used most frequently by scanner tools: (1) if conflicting matches have different lengths, the symbol with the longest match is applied; (2) otherwise, the symbol defined first is applied. String symbols are considered to be defined before identifier symbols.

**Example.** The input `if` can typically be matched by the symbol `Identifier` and the string symbol `"if"`. Mork matches the `"if"` symbols because its definition proceeds the definition of the `Identifier` symbol.

# Regular expression

Regular expressions define languages.

### Figure 4.3. Regular expression syntax

```
RegExpr              ::= Choice ;
Choice               ::= Choice "|" Sequence | Sequence ;
Sequence             ::= (Factor | Restriction)* ;
Restriction          ::= Factor "-" Factor;
Factor               ::= Operation | Reference | Range | "(" RegExpr ")"
Operation            ::= Star | Plus | Option | Times | Not ;

Star                 ::= Factor "*" ;
Plus                 ::= Factor "+" ;
Option               ::= Factor "?" ;
Times                ::= Factor ":" IntegerLiteral ;
Not                  ::= Factor "!" ;

Reference            ::= Symbol ;

Range                ::= Atom (".." Atom)? ;
Atom                 ::= CharacterLiteral | Code ;
Code                 ::= IntegerLiteral ;
```

**Regular Expression.** A regular expression defines a language, which is a set of strings. In the parser section, regular expressions define context-free languages; in the scanner section, regular expressions define regular expressions. A regular expression is said to match a string if the string is an element of the language.

**Choice.** A choice is a list of one or more sequences. A choice matches a string if one of its sequences matches the string. Formally, the language of an choice is the union of the languages of its sequences. Example: `A ::= "a" | "b";` matches `a` or `b`.

**Sequence.** A sequence is a list of zero or more factors or restrictions. A sequence matches any string that is the concatenation of strings matched by its factors. Formally, the language of a sequence is the set of concatenations of element, one element taken from any of its factors. An empty sequence matches the empty string. Example: `A ::= "a" "b";` matches `ab`.

**Restrictions.** A restriction matches a string if the left factor matches the string and the right factory does not match the string. This is useful, for example, to specify block comments: `comment ::= "/*" any* - (any* "*/" any*) "*/" ;`. Note that the syntax for restrictions does not permit `A - B - C` because the precedence is not obvious. Use `A - (B - C)` or `(A - B) - C` instead.

**Operation.** Operations are specified in postfix notation, giving priority to the inner-most operator. The factor left of an operator is called its operand. The following operations are available:

- **Plus.** A loop. Matches any string that is the concatenation of at least one string matched by the operand. Example: `A ::= "a"+;` matches `a`, `aa`, `aaa` etc.
- **Star.** Optional loop; matches any string that is the concatenation of zero or more strings matched by the operand. Example: `A ::= "a"*;` matches empty string, `a`, `aa`, etc.
- **Option.** Matches the empty string and any string matched by the operand. Example: `A ::= "a"?;` matches the empty string and `a`.
- **Times.** Matches any string that is the concatenation of the given number of strings matched by the operand. Example: `A ::= "a":4;` matches `aaaa`.
- **Not.** matches the inverted character set of the operand. It is an error to apply this operator to an operand which is not a character set. A character set is a regular expression with only ranges and choices. Example: `A ::= 'a'..'z'!;` matches any character which is not a lower case letter. Advanced note: not inverts ranges, not languages; otherwise, `'a'!` would match any string except `a`, e.g. `aa`, `aaa`, etc. To invert languages use restrictions with an unmatchable left factory.

**Reference.** Matches any string matched by the specified symbol. It is an error to reference parser symbols from the scanner section. References to scanner symbols from the scanner section are valid as long as no recursion is introduced; these symbols are automatically inlined, i.e. the symbol is replaced by it's definition.

**Ranges.** A range `a..z` matches any character `x` with `a <= x <= z`. A single boundary range `a` abbreviates the range `a..a`. An empty range is a range where the upper bound is smaller than its lower bound. Empty ranges are legal, they match nothing.

**Atom.** An atom matches the Unicode character it denotes. The character can be specified by a character literal or by a Unicode number.

**Translation into BNF.** The following substitutions are used by Mork to turn regular expression rules into plain context-free grammar rules. This is used to translate the parser section into a context-free grammar. Substitutions have been chosen to avoid nonterminal conflicts. In particular, the substitution of empty loops `Lst ::= Item*` into `Lst::= ; Lst::= Item Lst;` is not used, even though the rules are less complicated.

```
A ::= B | C    ->  A ::= B;   A ::= C;
A ::= B C? D; ->  A ::= B D; A ::= B C D;
A ::= B+;      ->  A ::= X;   X ::= X B;   X ::= B;
A ::= B*;      ->  A ::= B*?;
->  A ::= X;   A ::= ;      X ::= X B; X ::= B;
```

Notes: (1) The plus operator introduces new internal symbols. (2) Negation is not mentioned here, because it is illegal in the parser section.

**Example.** The rule `New ::= "new" Name "(" (Object ("," Object)*)? ")";` is internally turned to:

```
New ::= "new" Name "(" Object ")" ;
New ::= "new" Name "(" Object "," Object ")" ;
New ::= "new" Name "(" Object Lst ")" ;
Lst ::= Lst "," Object ;
Lst ::= "," Object ;
```

## Lexical structure

Syntax files essentially follow the lexical conventions of Java. The only major difference is the `EndOfLineComment`.

### Figure 4.4. Lexical structure

```
WhiteSpace         ::= ( ' ' | '\t' | '\n' | '\f' | '\r' )+ ;
EndOfLineComment   ::= '/' '/' ('\n' | '\r')!* ('\n' | '\r') ;
TraditionalComment ::= '/' '*' ('*'! | '*' '/'!)* '*' '/' ;


IntegerLiteral     ::= '0'
| '1'..'9' '0'..'9'*
| '0' ('x'|'X')
('0'..'9' | 'a'..'f' | 'A'..'F')+
| '0' ('0'..'7')+ ;

StringLiteral      ::= '"' (('\\' 0..65535)
| ('\\' | '"' | '\n' | '\r')!)+ '"' ;

CharacterLiteral   ::= '\'' (('\\' 0..65535) |
('\\' |'\'' |'\n' | '\r')!)+ '\'' ;

Identifier ::=
( 0x0024..0x0024 | 0x0041..0x005a | 0x005f..0x005f |
0x0061..0x007a | 0x00a2..0x00a5 | 0x00aa..0x00aa |
0x00b5..0x00b5 | 0x00ba..0x00ba | 0x00c0..0x00d6 |
0x00d8..0x00f6 | 0x00f8..0x00ff )
( 0x0000..0x0008 | 0x000e..0x001b | 0x0024..0x0024 |
0x0030..0x0039 | 0x0041..0x005a | 0x005f..0x005f |
0x0061..0x007a | 0x007f..0x009f | 0x00a2..0x00a5 |
0x00aa..0x00aa | 0x00b5..0x00b5 | 0x00ba..0x00ba |
0x00c0..0x00d6 | 0x00d8..0x00f6 | 0x00f8..0x00ff )* ;
```

Differences from Java:

- **`EndOfLineComment`.** `EndOfLineComment` start with the hash character #. Double slashes `//` known from Java `EndOfLineComment`s have a different meaning (they are used in paths).
- **Documentation comments.** Documentation comments are not distinguished from block comments.
- **literals.** Floating-point and boolean literals are not available; type suffixes in `IntegerLiterals` are not available.
- **keywords.** The set of keywords is different from Java. For examples, `start` is a keyword within specification files, whereas `for` is an identifier, not a keyword.

- **identifier.** Identifiers are restricted to Unicode characters with a character code <= 255.
- **Unicode preprocessing.** Prior to scanning, no Unicode preprocessing is performed. Thus, a token in general may not contain Unicode escapes. However, the definition of character and string literals has been extended to support Unicode escapes.

## Scanner modes

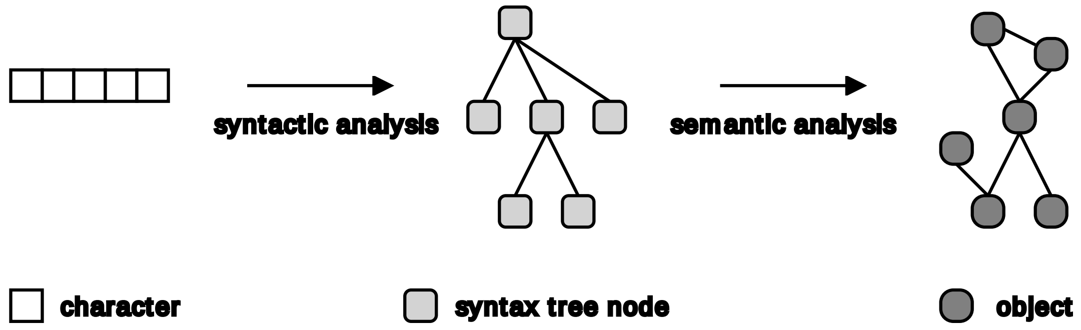
A scanner mode is a set of terminals. Whenever matching a string, the scanner matches only terminals in the current mode. Scanner modes are completely transparent, usually, there is no need to care about them: at compile-time, scanner modes are computed automatically, there is no need to manually declare scanner modes. At runtime, the parser automatically switches to the appropriate scanner mode, there is no need to manually set the scanner mode. For most syntax files, the resulting scanner has exactly one mode (containing all terminals).

TODO: more

# Chapter 5. Mapper files

**Mapping.** Turning input streams into objects is called *mapping* . Mapping turns data from a character representation into an object representation. Initially, data is represented as a sequence of characters without structure. This representation is appropriate to edit, store or transfer data. In contrast, the object representation the input stream is mapped to is appropriate to process the data. For example, numbers are represented by `int`s and list are represented by `List` objects. The purpose of mapping is to turn input streams into the representation most appropriate to process the data. Note: the object representation is similar to abstract syntax trees generated by some compiler tools, but the classes underlying individual objects can be freely chosen and objects are not restricted to trees.

**Figure 5.1. Mapping**



**Steps.** Mapping is comprised of two steps. In step one, the input stream is syntactically analyzed. This includes scanning and parsing and results in a syntax tree. This step turns that input stream from a character representation into a tree representation. The tree representation is used because it's most appropriate to check the syntax of the input stream. Step two takes the syntax tree and maps it into objects. This step computed attributes and includes semantic analysis like identifier resolution and type checking.

**Mapper.** Mapping is performed by an mapper object. A mapper is invoked for an input stream, performs scanning, parsing and semantic analysis and returns the objects resulting from the last step. A mapper is similar to an `ObjectInputStream`, they both kind of de-serializes objects and they are both objects themselves. However, a mapper de-serializes from a human-readable stream, whereas an `ObjectInputStream` de-serialize from an machine-readable stream.

**Mapper files.** A mapper file is a text file that defines a mapper. It defines the syntax of an input stream by referencing a syntax file and it defines how to map syntax trees into objects. Mork translates mapper files into Java classes.

**Naming conventions.** Mork does not restrict the name of mapper files. My naming convention for mapper files is as follows. The base name is the same as the base name of the syntax file. The file name extension is `.mapper`. Example: Mork itself has two mappers: `Syntax.mapper` and `Mapper.mapper`.

**Outline.** This chapter is about mapper files. Its sections are structured along the main entities of mapper files. A section typically starts with a formal syntax specification, followed by an informal description of the respective entity. Sections try to be complete, even if forward references are necessary. Note: the full syntax specification for mapper files is given in the appendix, the formal syntax specification at the start of a section are taken from this appendix.

**Lexical structure.** Syntax files and mapper files have the same lexical structure. A description was given in the previous chapter, it's not repeated here.

**Meta.** In some sense, syntax files as described in the previous chapter are simple because there is nothing special compared to other compiler tools. Defining mapper files should also be simple because it is basically a simple association. However, this chapter probably turns out to be more difficult because (1) the mapping concept differs from other tools and thus needs more explanations and (2) this chapter of the manual is incomplete.

## Overall structure

**Figure 5.2. Overall structure**

```
Mapper             ::= MapperName SyntaxFile Imports Definitions ;
MapperName         ::= "mapper" Name ";" ;
SyntaxFile         ::= "syntax" "=" StringLiteral ";" ;
Imports            ::= Import* ;
Import             ::= "import" PackageName ":" Class ("," Class)* ";";
Class              ::= Identifier ("->" Identifier)? ;
PackageName        ::= Name ;
Name               ::= Identifier ("." Identifier)* ;
```

**Overview.** Start symbol of the mapper file grammar is `Mapper`. `Definitions` form the core of mapper files, they associates symbols defined in `SyntaxFile` with Java classes declared in `Imports`.

## Mapper name

**Purpose.** The mapper name specifies a fully qualified name for the mapper. The name has to follow Java's rules for fully qualified class names. When generating a mapper, the name is used as a prefix for the various classes resulting from the generation. When running an application, the mapper name is used to locate and load these classes. The name is fully qualified, i.e. the mapper package is always included. Thus, the mapper name is like Java package declaration combined with a class definition combined into a single statement.

**File names vs. class names.** Java distinguishes class names and file names (for class files): A class name is not necessarily the name of the class file with the byte code for this class. For example, you can define a (non-public) Java class `Foo` in a Java file `Bar.java`. The Java compiler will generate a file `Bar.class` which defines a class `Foo`. Mork implements the same behavior for mappers: Defining a mapper `Foo` in a mapper file `Bar.mapper` results in a Java class file `Foo.class` defining a class `Bar`.

## Syntax file

**Syntax File.** `SyntaxFile` refers to the file defining the mapper's syntax. `StringLiteral` is the file name. A relative file name is interpreted relative to the location of the mapper file, not relative to the current directory. The character separating directories in the file name is `/`, regardless of the platform you run Mork on. Thus, it is illegal to use backslashes on Windows. Rationale: make mapper file platform independent.

## Import declarations

**Purpose.** `Imports` resemble Java import declarations: Import declarations specify classes to be referenced by simple identifiers, without the package that contains the class. The purpose of both Java and Mork import declarations is to simplify references to classes with long qualified names. Qualified names can get quite long, especially because Java naming conventions suggest a unique vendor name as part of the package name.

**Imported classes.** An import declaration `import p:X -> Y;` (where `p` is a package name and `X` is a simple identifier) declares the imported class `Y`. The imported class `Y` is a reference to the class `p.X`. `import p:X;` is equivalent to `import p:X -> X;`.

**Restrictions.** To import a class, the following conditions have to be met:

- There is exactly one imported class `Y`.
- `p.X` has been properly compiled and is available on the `CLASSPATH`.
- `p.X` is a class. It is an error to import primitive types or interface types.
- `p.X` is a public; it must not have private or default (i.e. package) visibility.
- Classes in unnamed packages are not supported. To import a class, it has to be member of a package.
- Nested classes are not supported, only top-level classes may be imported.

**Document dependencies.** The purpose of Mork import declarations goes beyond Java's import declarations: they are meant to document all dependencies of a mapper file. To realize this, mapper files allows to reference classes by simple identifiers only, qualified names are not allowed. Thus it is impossible to reference a class without mentioning the class in an import declaration.

**No import on demand.** In contrast to Java, `import *` - aka import on demand - is not supported. E.g. Mork rejects `import java.util.*`). Rationale for this restriction: document dependencies, every class used in a mapper file has to be declared explicitly by an import declaration. To reduce the typing resulting from this strategy, you can import multiple classes from the same package without repeating the package name.

**No implicit imports.** Mork has nothing like Java's implicit `import java.lang.*`. Thus, Mork requires an `import java.lang: Integer` before the `Integer` class can be referenced. In addition, classes from the mapper's package have to be explicitly imported. Example: `import foo: Bar;` is required to reference `foo.Bar` even if the mapper package is `foo`. Once again, the rationale is to document dependencies.

## Constructors

**Figure 5.3. Constructors**

```
Constructor         ::= ClassRef | MemberRef | Internal | Copy;
ClassRef            ::= Identifier ;
MemberRef           ::= Identifier "." Identifier ;
Internal            ::= "[" Identifier "]" ;
Copy                ::= "(" Identifier ")" ;
```

**Definition.** A constructor creates an object or throws an exception. A constructor is a generalization of a Java constructor. It is either a normal constructor or a special constructor. A normal constructor is a Java constructor or a Java member (i.e. methods or a fields). Thus, most Java classes define a set of constructors. A Java constructor is referenced by its class, a member constructor is referenced that a class and an identifier. Note that classes are referenced by simple identifiers, qualified class names are not supported. (See the section about import declarations for more details). Besides normal constructors, Mork provides two kind of special constructors: internal constructors and copy constructors.

**Unused constructors.** A constructors is invoked even if the result is never used. This is useful if the constructors has side-effects (maybe side-effects on arguments), and the return type is void or the result is not used.

A constructor is characterized by its:

- name
- argument type list
- result type
- exceptions

**Static typing.** Argument and return types are known at compile time. This type information is used by Mork to perform static type checking. At runtime, all values passed to or returned from a constructor are guarantied to be assignable to/from the specified type. It is impossible to get runtime type mismatches here because of an illegal constructor invocation.

## Normal constructors

**Java constructors.**

- **name.** A public Java constructor of an imported class `C` defines a constructor with the name `C`.
- **arguments.** `C` takes the same arguments as the Java constructor.
- **result.** `C` returns the object created by the Java constructor.
- **exceptions.** `C` throws all exceptions (both checked and unchecked) thrown by the Java constructor.

**Java methods.**

- **name.** A public Java method `m` of an imported class `C` defines a constructor with the name `C.m`.
- **arguments.** `C.m` takes the same arguments as the Java method; non-static methods take an additional argument of type C as the first argument.
- **result.** `C.m` returns the result of the Java method call. For non-static methods the constructor invokes the Java method on the object passed as first argument.
- **exceptions.** `C.m` throws all exceptions (both checked and unchecked) thrown by `C.m`.

**Java fields.**

- **name.** A public Java field `F` of a imported class `C` defines a constructor `C.F`.
- **arguments.** If `C.F` is a static field, `C.F` takes no arguments. Otherwise, `C.F` takes an argument of type `C`.
- **result.** If `C.F` is a static field, `C.F` returns the value of the static field. Otherwise, `C.F` returns the value of the field of the argument object.
- **exceptions.** C.F throws a `NullPointerException` is a non-static field is invoked with a null argument.

**Example.** TODO

## Special constructors

**Internal constructors.** Internal constructors provide access to internal variables maintained by the scanner. Only terminal symbols can use internal constructors, it is an error to trigger an internal constructor for a non-terminal symbol. Internal constructors are specified within array brackets. The following list shows the available internal constructors, what the constructor returns and the return type. None of these constructors takes arguments and none of these constructors throws exceptions.

- **text.** Returns a string object with the characters of the current terminal symbol. For example, `42` might be the text returned for an `Integer` terminal symbol. Return type is String.
- **`ofs`.** Returns an Integer value with the current source offset. The first offset is 0. Return type is Integer.
- **line.** Returns an Integer value with the current source line. The first line is 1. Return type is Integer.
- **column.** Returns an Integer value with the current source column. The first column is 1. Return type is Integer.

**Copy constructors.** A copy constructor is specified by a class name within brackets. The specified name must be the simple name of a class imported in the section header. This notation resembles the Java cast operator. The copy constructor does nothing, in particular, no constructor is invoked. Copy constructors are useful to work around some argument passing problems. Copy constructors should become obsolete by solving these problems.

## Exceptions

**Purpose.** Constructors may throw checked and unchecked exceptions. An unchecked exception is an exception derived from `java.lang.RuntimeException` or `java.lang.Error`). Otherwise, it is a checked exception. Technically, a constructor may throw exceptions of any type, whenever it sees fit; there are no restrictions imposed by Mork. However, I suggest using exceptions as outlines in the section.

### Unchecked exceptions

Unchecked exceptions should be thrown if a constructor detects an internal error. An internal error is something that "should not happen", an inconsistent/unexpected state. An internal error can be avoided by the programmer. If not, it's a bug. A `NullPointerException` usually indicate an internal error.

**Example.** `java.lang.Integer.parseInt` throws the unchecked exception `NumberFormatException`. If the mapper syntax imposes the correct number syntax on the strings passed to `parseInt`, the method can be used as a constructor. If a `NumberFormatException` occurs, this is a bug in the application, because the string passed to `parseInt` is assumed to be a valid number.

### Checked exceptions

**Usage.** A constructor should use checked exceptions to impose constraints on the input. Whenever the input (i.e. the arguments passed to the constructors) violates a constraint, a checked exception should be thrown. You are free to check whatever constraint you wish. A mapper catches checked exceptions and turns the exception's message into an error message. Typically, this message is show to the end-user.

**Semantic errors.** Compiler text books use the term semantic error if some constraint on the input stream has been violated. The classic semantic errors are "undefined identifiers" and "type mismatches".

**Example 1.** `java.lang.Class.forName` throws the checked exception `ClassNotFoundException`. When used as a constructor, this is reported as a semantic error.

**Example 2.** Consider a calculator and an end-user entering the term `1/0`. The `div` method below issues the appropriate error message by throwing a `GenericException`. (`GenericException` is a general-purpose checked exception provided by Mork.) Without testing for 0 in the method body, the `1/0` would result in an internal error of the calculator because `left/right` would triggers the (unchecked!) exception `ArithmeticException` if `right == 0`.

```
package foo;

import net.sf.beezle.mork.util.GenericException;

public class bar {
public static int div(int left, int right) throws GenericException
if (right == 0) {
throw new GenericException("division by zero");
}
return left/right;
}
}
```

**Note.** Imposing constrains on constructor arguments sounds like Java's `IllegalArgumentExceptions`. This exception usually indicates an internal error: some Java code has invoked a constructor with illegal arguments. In contrast, a semantic error is a checked exception, it is not considered an internal error. The difference to an `IllegalArgumentException` is, that applications cannot prevent semantic errors: arguments are attributes, and attributes stem (directly or indirectly) from end-user's input. Unlike Java code, end-user input cannot be controlled by the application.

## Attributes

**Overview.** An attribute associates a symbols with a constructor. A simple attribute is defined by `Foo => Bar`, where `Foo` is a symbol and `Bar` is the name of a constructor. Basically, a mapper file is a list of attribute definitions, attributes form the core of the file,

**Figure 5.4. Attribute syntax**

```
Definitions         ::= Group* ;
Group               ::= Symbol Attribute+ ;
Symbol              ::= StringSymbol | IdentifierSymbol ;
StringSymbol        ::= StringLiteral ;
IdentifierSymbol    ::= Identifier ;
Attribute           ::= AttributeName "=>" Constructor Visibility;
AttributeName       ::= (":" Identifier)? ;
```

## Definition

**Attribute.** An attribute is defined by (1) a symbol, (2) an attribute name, (3) a constructor name and (4) visibility. The attribute is said to be attached to its symbol. Symbol and constructor name are most important here, they define the actual mapping. In contrast, it's common not to specify an `AttributeName` and a `Visibility`. The constructor name (`Constructor`) may be overloaded. For example, a constructor name `foo` might refer to a constructor `foo(String)` and `foo(List)`.

**Attribute groups.** Attributes attached to the same symbol may be grouped. This is for your convenience, the effect is the same as to isolated attribute definitions.

**Attribute name.** `AttributeName` defines a name that can be used to refer to the attribute in visibility declarations. Every attribute has a name, if it is not supplied in the attribute definition, the name defaults to the name of the symbol the attribute is attached to. Attribute names have to be unique throughout a mapper file.

**Main attributes.** A main attribute is an attribute whose name equals the name of the symbol it is attached to. Since attribute names have to be unique, a given symbol has at most one main attribute. Main attributes play a key role in defining implicit visibility.

**Visibility.** `Visibility` defines constructors where the attribute is passed as an argument.

**Argument passing.** The argument list passed to a constructor is defined by the visibility of all attributes defined in a mapper file. This differs from the Java mechanism where arguments are specified explicitly by an argument list.

## Compile-time

At compile-time, Mork computes various values for each attribute: the type, the argument list, and the constructor.

**Type.** Objects are guaranteed to be of the attribute type. Attribute type is a Java type. The type of an attribute is used to perform static type checking and to resolve overloaded constructor names. In this sense, the attribute type is similar to the type of a local variable in Java. However, the attribute type is not explicitly declared: The type `t` of an attribute with constructor name `N` is computed as follows:

1. Determine the list `C` of all constructors with name `N`. In may cases, `C` has a single element only. Otherwise, `N` is overloaded.
2. Determine the list `T` of all return types of the constructors in `C`.
3. Replace all primitive types in `T` by the corresponding wrapper type. For example, replaced `int` by `Integer`.
4. The attribute type `t` is the most special Java class type that is assignable from each element in `T`.

Notes: (1) The attribute type is always a reference type, the above steps never result in a primitive type. (2) The attribute type is always a class type, the above steps never result in an interface type. Interface types had to be excluded because they would introduce ambiguities to the algorithm.

**Example.** Consider the attribute `X => a;` and two constructors `String a(String)` and `int a(int)`. The attribute type is `Object`, because Object is the common supertype of `String` and `Integer`.

**Argument list.** The argument list is a list of attributes with cardinality. The visibility section describes how the argument list is computed. TODO

**Attribute constructor.** An attribute's constructor is obtained by using the argument list to resolve overloaded constructor names: A constructor name is resolved to a constructor `F` if the argument list can be converted to the constructor's argument list. It is an error if there is no such `F` or if `F` is not unique. This is the same mechanism used in Java, the argument type decides, if the constructor name is ambiguous. Note that overloading is resolved at compile-time, using static type information. It's not possible to choose the constructor at runtime, using the actual argument type(s).

**Argument list conversion.** Argument list conversion adjusts an argument list to match the formal argument list of the attribute constructor. Conversion re-orders and unwrap arguments if necessary. The conversion algorithm takes a list of actual arguments and transforms it to a list of converted arguments:

- Choose the first formal argument F of the constructor.
- Choose the leftmost argument A from the actual argument list such that A can be converted to F.
- If no such A is found, the actual argument list cannot be converted. Otherwise, remove A from the actual argument list an append the converted type for A to the converted argument list.
- Repeat the previous steps for all remaining formal arguments.
- Conversion is succeeds, if all actual arguments have been moved to the converted arguments.

Argument conversion is explained in the visibility section.

**Argument order.** A given symbol may have any number of attributes. The ordering of objects resulting at runtime follows the ordering of attribute definitions in the mapper file. This implies that the array returned from running a mapper is ordered according the attribute sequence for the start symbol.

## Runtime

**Runtime vs compile-time.** Symbols and attributes are compile-time entities. They define how to create their respective runtime entity: syntax tree nodes and objects. Each syntax tree node has a defining symbol, and each object has a defining constructor.

**Runtime effect.** A syntax tree node for a given symbol `S` is mapped into objects by invoking the constructors of all attributes attached to `S`. Example: an attribute `Foo => Bar;` defines that the symbol `Foo` is mapped to the object returned by the constructor `Bar`. Whenever a mapper identifies a `Foo` syntax tree node, the `Bar` constructor is invoked. The constructor returns an object which is passes to other constructors as defined by the attributes visibility.

**Runtime effect.** At runtime, syntax tree nodes are mapped into objects. Consider a symbol `S`. At runtime, each syntax tree node of `S` is augmented with the objects that result from invoking all constructors triggered for symbol.

**Wrapper objects.** If a constructor returns a primitive value, this is automatically wrapped by the appropriate wrapper object. This wrapping is reflected in the above rules to compute the attribute type. For example, `int` values are wrapped by `Integer` objects. Automatic wrapping corresponds with automatic unwrapping for constructor arguments. In effect, wrapping and unwrapping is transparent for constructor definitions. Within Java code, you can freely choose to work with primitive values or wrapper values.

# Visibility

Mork uses visibilities rules because most of the argument passing is trivial and known implicitly. Visibility rules just specify where trivial argument passing does not apply.

TODO: This section is a collection of fragments. Visibility is very experimental and subject to change.

**Figure 5.5. Visibility syntax**

```
Visibility          ::= Implicit | Explicit ;
Implicit            ::= ";" ;
Explicit            ::= ":" (">" Path)* ";" ;
```

Visibility defines to what constructor an object is passed as an argument. It is specified by modifiers and a list of views. Most argument passing is specified by modifiers, paths deals with special situations.

**Empty visibility.** The constructor of an attribute is invoked, even if it has an empty explicit visibility. Empty explicit visibility is useful if the constructor has side-effects on its arguments.

**Argument types and argument values.** An argument type is not the same as a Java type, because an argument value is not (always) the same as a Java value! In fact, any number of Java values might make up a single argument value. In many cases, an argument value is comprised of exactly one Java value. However, argument values can consist of a list of objects. In order to encode this, the argument type is comprised of two elements: a component type and a cardinality.

**Component type.** The component type is a Java reference type, for example String or Object. Each individual Java value within an argument value is assignable to the component type. Note that the component type can not be a Java primitive type like int or boolean. This is not a restriction because Mork automatically uses wrapper classes. Note also, that reference types include both arrays of primitive types and arrays of reference types. The Java language defines arrays to be reference types.

**Cardinality.** The cardinality determines how many Java values make up an argument value.

- **value.** Exactly one Java value. In this manual, C:1 denotes an argument type with component type C and value cardinality.
- **option.** None or one Java value. In this manual, C:? denotes an argument type with component type C and option cardinality.
- **sequence.** Any number of Java values, including zero. In this manual, C:* denotes an argument type with component type C and sequence cardinality.

**Conversion.** Mappers pass argument values around, but constructors are defined by Java programs and operate on Java objects. Constructors take Java values as arguments and they return Java values as results. Thus, a mapper has to convert between attribute values and Java values when transferring values to and from constructors. The rules for these conversions are described in the constructors section.

**Argument conversion.** Arguments are passed to constructors, which includes argument conversion. At runtime, argument conversion might unwrap objects and create arrays or lists. An argument A can be converted to a formal argument F if:

- A has value or option cardinality and F is assignable from the component type of A.
- A has value or option cardinality and F is a primitive type and the wrapper type for F is assignable from the component type of A.
- A has sequence cardinality and F is a assignable from a Java List or F is assignable from the array type for the component type.

## Paths

TODO: this section is a collection of fragments. Paths are still evolving and subject to change.

**Purpose.** Paths declare non-local visibility. They are special in two ways: (1) a single path can select multiple values, and (2) paths can access remote values.

**Figure 5.6. Path syntax**

```
Path                ::= ImplicitPath | LocalPath | NormalPath ;
ImplicitPath        ::= "\\\\*" ;
LocalPath           ::= Identifier ;
NormalPath          ::= Step+ ;
Step                ::= Move Identifier ;
Move                ::= Ups | Up | Downs | Down ;
Ups                 ::= "\\\\" ;
Up                  ::= "\\" ;
Downs               ::= "//" ;
Down                ::= "/" ;
```

**Paths.** A path is a sequence of steps, where a step is comprised of a move and an identifier. The last identifier within a path is specifies an attribute, all other identifiers specify symbols. The context of a path is the attribute symbol, the path is used in.

**Move.** Move specify how to move in the syntax tree. Up and down moves (`\` and `/`) specify a single step, ups and downs moves (`\\` and `//`) specify repeated steps.

**Dead-end paths.** A dead-end path is a path that does not reach any attributes, i.e. that does not actually contribute to the visibility.  It is an error to specify dead-end paths. Example: `/X` if a dead-end path when applied to a symbol `Y ::= Z Z;` . TODO: what about the start symbols?

# Chapter 6. Compiling

**Overview.** Mork generates mappers. Technically, this is done by translating ("compiling") mapper files into class files. The mapper file is supplied by the user, it specifies the mapping of input streams into Java objects as needed by the respective application. The generated class files are Java class files, they follow the same rules as class files generated by the Java compiler. Applications need these class files in their `CLASSPATH` to load and run mappers.

## Invocation

**Prerequisites.** Before running Mork, make sure to compile you Java source files. Mork checks classes referenced in mapper files to perform type checking. These classes have to be properly compile before running Mork: run `javac` on them.

**Invocation.** `"mork" option* file*`

Mork is similar to `javac`: the user specifies a list of options followed by a list of files to compile. Mork compiles the file in the given order.

**Example.** `mork Foo.mapper` compiles the mapper file, loading it from the current working directory. A successful compilation results in a class file `Foo.class` (and some other class files `Foo*.class`).

**`CLASSPATH`.** When invoking Mork on a file `Foo.mapper`, the `CLASSPATH` has to contain all classes imported in the mapper file. For example, if `Foo.mapper` imports a class `mypkg.Bar`, this class has to be available via the `CLASSPATH` variable. Otherwise, Mork reports an error, indicating that the class `mypkg.Bar` is not defined.

**Memory consumption.** If Java runs out of memory (i.e. throws an `OutOfMemoryError`), use the Java `-Xmx` option in the launch script to increase the memory available to the virtual machine. Example `java -Xmx128M net.sf.beezle.mork.compiler.Main veryComplex.mork`.

**Command-line arguments vs. mapper files.** Mork is controlled by both mapper files command line options, but what setting is specified where? Logical settings go into the mapper files, i.e. setting affecting what input is actually mapped into which object. The same mapper file always results in the same mapping, no matter what options have been specified on the command line. Physical settings go into the command line: where the generated code is stored, or what additional output it generated. Or even - in future version - optimization settings.

## Options

Options start with "-" followed by the option name. Options are case-sensitive. Currently available options:

- **`-help`.** Prints help about invoking Mork.
- **`-stat`.** Prints statistics about mappers to standard output. Use this option to see the size of the generated finite automatons and pushdown automatons.
- **`-k <n>`.** Specifies the lookahead, the k in lr(k). Default is 1.
- **`-lst`.** Generates listing files. If this option is given, Mork generates a listing of each compiled file. The listing includes a list of symbols, the mapper's grammar and the generated automatons and can get very long. The listing for a file `Foo.mapper` is written to the file `Foo.lst`. Use this option to find grammar conflicts.

- **`-d` directory.** Sets the destination directory for generated class files. If a mapper is part of a package, Mork puts the class files in a sub directory reflecting the package name, creating directories as needed.
- **`-verbose`.** Issues overall progress information to standard output. If compiling is slow, this option can be used to locate which part of the mapper generation is slow.

## Re-compiling

When do you have to re-run Mork? Nothing surprising here, the rules are similar to Java files and the Java compiler. A compiled mapper depends on its mapper file and the signature of the referenced constructors. Thus, re-compile a mapper if the mapper file (or the referenced syntax file) or the signature of at least on of the referenced constructors has changed. Note that changing a constructor without affecting the signature does not require a re-compile.

Class file analyzer tools see constructor references as normal Java call, no reflection is used. Thus, it is possible to use a to find class file dependencies.

## Maven

You'll usually use Maven to build your application. Here's the approriate configuration to invoke Mork on the file `src/main/Foo.mapper`.

```
<plugin>
        <artifactId>maven-antrun-plugin</artifactId>
        <version>1.7</version>
        <executions>
          <execution>
            <id>mork</id>
            <phase>process-classes</phase>
            <configuration>
              <target>
                <java fork="true" dir="${basedir}" classname="net.sf.beezle.mork.c
                  <classpath>
                    <pathelement path="${basedir}/target/classes" />
                    <path refid="maven.compile.classpath" />
                  </classpath>
                  <arg value="-d" />
                  <arg path="${basedir}/target/classes" />
                  <arg path="${basedir}/src/main/Foo.mapper" />
                </java>
              </target>
            </configuration>
            <goals>
              <goal>run</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
```

# Chapter 7. Usage

This chapter explains how to work with mapper objects in your applications. The first section outlines the most important steps. This is followed by a kind method reference of the mapper class. Finally, the last section describes the runtime environment required to use a mapper.

## Overview

Using a mapper in an application involves creation and running. Suppose the mapper name was `foo.Mapper`. The following code fragment create a `foo.Mapper` instance and runs it on the file `xy`.

**Example 7.1. Mapper usage**

```
import net.sf.beezle.mork.mapping.Mapper;

Mapper mapper;
Object[] result;
int i;

mapper = new Mapper("foo.Mapper");
result = mapper.run("xy");
if (result != null) {
System.out.println("success");
for (i = 0; i < result.length; i++) {
System.out.println("\t" + result[i]);
}
} else {
System.out.println("error(s), aborted");
}
```

If the file `xy` is mapped without errors, the returned object array is printed. Otherwise, `run` prints an error message and returns `null`. Except for Java IO problems, all error messages include the current source position.

## The `Mapper` class

The class `net.sf.beezle.mork.mapping.Mapper` forms the main API used by applications. The following describes the most important constructors and methods. This description is sufficient for normal use. See the API documentation for more details.

**`Mapper` vs. Generated classes.** Note that `Mapper` is not a generated classes. How does this class know the specific mapping? And what about the classes actually generated by Mork? Simple - `Mapper` loads the generated classes. `Mapper`'s purpose is to wrap the generated classes with a fancy API. And to reduce duplicated code since major parts of the mapper code do not depend on the specific mapping. The generated classes encapsulate the application-specific parts of the code, but the `Mapper` class hides them from the application.

**Advanced note.** Note that the `Mapper` class solves a subtle circular dependency problem. Suppose some class `A` in some application directly uses a class `G` generated by running Mork on a mapper file `M`. To compile `A`, `G` has to exist, and thus, `M` must have been compiled. But run Mork on `M`, all Java classes

`C` that define constructors referenced from `M` must have been compiled. This is a circular dependency if `A` defines constructors or if there is only a single Java-compile step to compile all Java classes. The `Mapper` class works around this problem by using Java reflection to refer to generated classes. As far as I know, there is no performance drawback.

**Threading.** Mapper objects are not thread-safe. If your code uses multiple threads you have to synchronize them to make sure that at most one thread at a time executes within a given mapper object.

## Constructor

```
Mapper(String name)
```

Instantiates the mapper with the specified name. The `name` argument specifies the mapper name as defined in the mapper file. You can create any number of mapper objects, even instances with the same mapper name. However, most applications will use a single mapper object since a given object may be run any number of times.

**`CLASSPATH`.** The specified `name` is used to locate the mapper classes generated by Mork. Class loading is performed using the class loader that loaded the `Mapper` class. In most cases, this is the system class loader which searches the class along the application's `CLASSPATH`. As a result, the same rules for setting the `CLASSPATH` and placing class files apply to class files generated by Mork and class files generated by the Java compiler.

**Class loading.** Generated classes a loaded on demand, they are not loaded unless actually used. In most cases, they are loaded as part of one of the various `run` methods. The purpose is to allow applications to instantiate all `Mapper` objects they might need without wasting time to load classes not actually needed. Time for class loading is an issue since generated classes might get quite large.

**`IllegalStateException`.** An `IllegalStateException` is thrown if the generated classes cannot be loaded (e.g. because of an internal `ClassNotFoundException`). Usually, this indicates a typo in the specified mapper name or a `CLASSPATH` problem. If you get an `IllegalStateException`, check your `CLASSPATH` and make sure that the mapper was compiled properly with the appropriate version of Mork. Note: an unchecked exception is thrown because the caller is not expected to catch and recover from this problem. I can see only good reason for catching the exceptions: to report an internal application error and quit.

## `setErrorHander`

```
setErrorHandler(net.sf.beezle.mork.mapping.ErrorHandler)
```

Defines the error handler used by `run` to report errors. The `ErrorHandler` documentation below explains the various types of errors.

**Default error handler.** If `setErrorHandler` is not used to explicitly define an error handler, `run` uses a default error handler. The default error handler prints all errors to `System.err`. All error messages includes the position where the problem occurred. This behavior is sufficient for may applications. Especially if you start with a new application, it is save to ignore `setErrorHandler` an rely on the default error handler.

## `setEnvironment`

```
setEnvironment(Object)
```

Defines the mapper's environment object. Use `YourSymbol : [env];` to access this object from your mapper file.

## `run`

`run(Object context, Reader src)`

Runs the mapper on the specified stream. `run` performs syntactic analysis (i.e. scanning and parsing) and mapping into Java objects. If `run` does not detect errors, it returns the Java objects of the start symbol. The context argument specifies a name for the reader, `context.toString()` is used to report error positions and is typically a file name. The `src` argument is internally wrapped by a buffer, the is no need to pass some kind of buffered reader.

**Error handling.** If `run` detects an error (e.g. a syntax errors), this problem is reported to the errors handler defined via `setErrorHandler`. After reporting the error, `run` is terminated, returning `null` to the caller. Note that the first errors terminates `run`, there is currently no way to recover, mapping result in one error at most.

**Advanced issues.** (1) A mapper can be run more than once, it is not necessary to create a new mapper to run it on a second stream. (2) Run is not thread-save, the caller has to make sure that only one thread enters the method at a given time.

**Unchecked exceptions.** `run` does not catch unchecked exceptions thrown in constructors. Thus, any unchecked exception thrown by a constructor causes the Java virtual machine to terminate `run` abruptly and propagates the exception o the caller. In my opinion, unchecked exceptions should not be catched. Usually, there is no way for an application to recover from an unchecked exception because it is either some virtual machine problem (e.g. out of memory) or an internal error in the application. In both bases the application should be terminated.

**Cyclic dependencies.** `run` throws an unchecked exception `CyclicDependencies` if the attribute grammar used internally has a cyclic dependency. This indicates a problem with the mapper specification, not with the stream processed by the mapper. To correct this problem, the underlying map file has to be fixed. `CyclicDependencies` is an unchecked exception because the application has to be fixed, applications are not expected to recover from this problem.

## `run`

`public Object[] run(String fileName)`

Convenience method to map files. Opens `fileName` as a `FileReader` and invokes `run(Object context, Reader src)`, passing `fileName` and the `FileReader` object as argument. As usual, a relative `fileName` is interpreted relative to the current working directory. The `FileReader` object will always be closed when returning from the method, even if the mapper throws an unchecked exception.

# The `ErrorHandler` interface

`net.sf.beezle.mork.mapping.ErrorHandler`

Error handling is work in progress. If you have a choice, please rely on the default error handling. Avoid implementing you own error handler since the interface is expected to change. Similarly, this documentation is preliminary. It concentrates on the various error types (which I consider more stables), it does not explain individual methods of the interface. If you have to implement an error handler, the source code of `net.sf.beezle.mork.mapping.PrintStreamErrorHandler` might provide some help.

**Exceptions.** `ErrorHandler` distinguishes various error types, where each type of error reported by a different exception. Different exceptions resemble the various modules involved in mapping. Every module reports errors to the `run` method by throwing his individual exception. `run` catches these exceptions and reports them to the registered error handler. The following exceptions are distinguished:

- **`IOException`.** Indicates an IO problem in Java's IO classes. These classes feed characters to the scanner.
- **`IllegalToken`.** Indicates a scanner problem, aka a lexical error.
- **`SyntaxError`.** Indicates a parser problem, aka a syntax error.
- **`SemanticError`.** Indicates a problem while mapping into Java objects, aka a semantic error.

Note: The terms lexical error, syntax error and semantic error resemble compiler construction terminology. These terms are kept even though they conflicts with Java terminology, where errors usually indicate a problem in the virtual machine. Both `SyntaxError` and `SemanticError` are derived from Java's `java.lang.Exception` class.

**Position.** `IllegalToken`, `SyntaxError` and `SemanticError` have a position field that stores the position in the input stream where the problem occurred. The source code for `PrintStreamErrorHandler` demonstrates how to use this field to report problems to the user.

**Semantic errors.** A `SemanticError` exception is thrown if a constructor invoked by the mapper throws a checked exception. `SemanticError` has a field with the original exception thrown by the constructor and the position of the symbol that triggered the constructor. The term semantic error is compiler construction terminology. A typical semantic error is an undefined identifier or a type mismatch. Constructors are expected to check for semantic errors like undefined identifiers and throw a checked exception to indicate problems.

## Deployment

As long as you applications on your development system, you can skip this section: all Mork classes needed to run the application have been set up by installing Mork on the development system. This section applies if you want to (manually or automatically) deploy applications on a system where Mork is not installed.

**Runtime environment.** Two things have to be installed on a system to run an application: a Java runtime environment, version 1.2 or higher, and the Mork classes in `lib/mork.jar`. If the application is executed, Mork classes have to be available in its `CLASSPATH`. This can be done by globally adding the jar file to the `CLASSPATH` in some initialization script or by providing an application launch script that defines the `CLASSPATH` for the application only.

**Optimizations.** `lib/mork.jar` includes quite a few classes that are need by Mork only, their are not necessary to run applications. If you want to dig into re-compiling Mork, here are some tips how to reduce the size of the jar file.

- Omit classes from the library that are not used by a mapper. Most notably, you don't need Mork's `compiler` and `classfile` packages. If possible, use a packaging tool to automatically remove classes/methods not needed by your application.
- Re-compile Mork without debug and line number information (`javac`: omit the -g option, add the -O option).
- Use a better compiler. For example, Jikes 0.53 produces significantly smaller class files than Blackdown JDK 1.2 pre 2 `javac`.

# Chapter 8. Source Code

## Overview

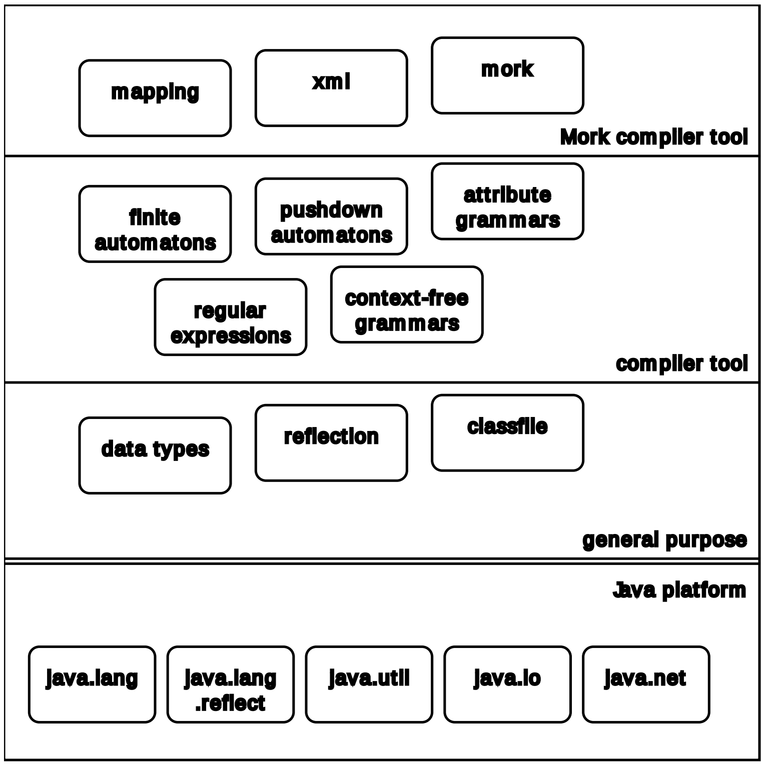
**Version numbering.** The version numbering scheme is `major.minor`. The minor number is bumped for normal releases with normal, incremental improvements. The major number is bump for milestones.

**Figure 8.1. Directory structure**

```
mork
|- src/main/java/net/sf/beezle/mork
|   |- compiler              compiler
|   |- mapping               map symbols to objects
|   |- semantics             attribute grammars
|   |- parser                push-down automatons
|   |- scanner               finite automatons
|   |- grammar               context-free grammars
|   |- regexpr               regular expressions
|   |- classfile             Java class files
|   |- reflect               reflection
|    - util                  data types, misc
|- src/test/java             tests
|- src/site/manual           documentation
 - examples                  example applications
```

**Architecture.** Basically, Mork is a Java class library together with some documentation, examples, a build system (python) and some launch scripts. The packages in the class library a structured into three layers, with lower layers having no dependencies on upper layers. The lowest layer is general-purpose, it is not tied to compiler tools. The compiler tool layer is by far the biggest part. Basically, it implements typical data types described in compiler construction text book. This layer is not tied to Mork, you could use it to realize other compiler tools. The Mork layer is the top-most layer, adding Mork-specific stuff.

**Figure 8.2. Architecture**



## Example applications

Mork includes a number of example applications. Every example is implemented in a package of the example's name. Java's `main` method is always placed in a class called Main. The following steps compile and run the example application `foo`.

- Optional: build `javadoc`:

```
cd examples
mkdir foo/javadoc
javadoc -d foo/javadoc -sourcepath . foo
```

- Compile the Java files:

```
cd examples/foo
javac *.java
```

- Compile the mapper:

```
mork Mapper.mapper
```

- Run it:

  ```
  java foo.Main
  ```

  If the example needs arguments, a usage message will be printed.

**calc.** The classic example for compiler tools: calculate simple arithmetic expression.

**command.** The application described in the first steps chapter of this manual.

**interpreter.** Interpreter for a simple language. The implementation instantiates classes that resemble a kind of abstract syntax. The methods of these classes implement the interpreter. Note that no symbol table is necessary, and that variable references point directly to the referenced variable.

**compiler.** Compiler for a simple language. The implementation instantiates classes similar to those in the interpreter example. But instead of interpretation methods, these classes have methods to generate Java byte code.

To compile and run a program `Foo.prog`, process as follows.

- `java compiler.Main Foo.prog`
- `java Foo`

**`jp`.** A parser for Java 2, version 1.2. For example, you can run `jp.Main examples/*/*.java` to parse all examples.

## Building Mork

To build Mork, simply invoke "mvn clean install site"

## Miscellaneous

**Source conventions.**

- maximum line length: 100 characters
- indentation: 4 spaces, not tabs
- newline: Unix convention
- a raw `RuntimeException` is thrown if an inconsistent state, i.e. a bug, has been detected.

# Appendix A. Mork syntax files

Mork is bootstrapped, the file listed in here are written in their own language.

`Syntax.syntax` defines the syntax of syntax files.

```
#
# Copyright 1&1 Internet AG, http://www.1and1.org
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU Lesser General Public License as published by
# the Free Software Foundation; either version 2 of the License,
# or (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
# See the GNU Lesser General Public License for more details.
#
# You should have received a copy of the GNU Lesser General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.
#

[PARSER]

#
# overall structure

Syntax              ::= Parser Scanner ;
Parser              ::= "[" "PARSER" "]" Rule+ ;
Scanner             ::= "[" "SCANNER" "]" Priorities WhiteOpt Rule* ;
Priorities          ::= UsePriorities | NoPriorities;
UsePriorities       ::= ;
NoPriorities        ::= "nopriorities;";
WhiteOpt            ::= ("white" "=" SymbolSet ";")? ;
SymbolSet           ::= ( Symbol ("," Symbol)* )? ;


#
# symbols

Symbol              ::= StringSymbol | IdentifierSymbol ;
StringSymbol        ::= StringLiteral ;
IdentifierSymbol    ::= Identifier ;
Rule                ::= Subject "::=" RegExpr ";" ;
Subject             ::= Symbol ;

#
# regular expressions

RegExpr             ::= Choice ;
Choice              ::= Choice "|" Sequence | Sequence ;
Sequence            ::= (Factor | Restriction) * ;
```

```
Restriction          ::= Factor "-" Factor;
Factor               ::= Operation | Reference | Range | "(" RegExpr ")" ;
Operation            ::= Star | Plus | Option | Times | Not  ;

Star                 ::= Factor "*" ;
Plus                 ::= Factor "+" ;
Option               ::= Factor "?" ;
Times                ::= Factor ":" IntegerLiteral ;
Not                  ::= Factor "!" ;

Reference            ::= Symbol ;

Range                ::= Atom (".." Atom)? ;
Atom                 ::= CharacterLiteral | Code ;
Code                 ::= IntegerLiteral ;

[SCANNER]
    white = WhiteSpace, EndOfLineComment, TraditionalComment ;


IntegerLiteral      ::= '0'
                      | '1'..'9' '0'..'9'*
                      | '0' ('x'|'X')
                            ('0'..'9' | 'a'..'f' | 'A'..'F')+
                      | '0' ('0'..'7')+ ;

StringLiteral       ::= '"' (('\\' 0..65535)
                         | ('\\' | '"' | '\n' | '\r')!)+ '"' ;

CharacterLiteral    ::= '\'' (('\\' 0..65535) |
                            ('\\' |'\'' |'\n' | '\r')!)+ '\'' ;

Identifier ::=
   ( 0x0024..0x0024 | 0x0041..0x005a | 0x005f..0x005f |
     0x0061..0x007a | 0x00a2..0x00a5 | 0x00aa..0x00aa |
     0x00b5..0x00b5 | 0x00ba..0x00ba | 0x00c0..0x00d6 |
     0x00d8..0x00f6 | 0x00f8..0x00ff )
   ( 0x0000..0x0008 | 0x000e..0x001b | 0x0024..0x0024 |
     0x0030..0x0039 | 0x0041..0x005a | 0x005f..0x005f |
     0x0061..0x007a | 0x007f..0x009f | 0x00a2..0x00a5 |
     0x00aa..0x00aa | 0x00b5..0x00b5 | 0x00ba..0x00ba |
     0x00c0..0x00d6 | 0x00d8..0x00f6 | 0x00f8..0x00ff )* ;

#
# white space

WhiteSpace          ::= ( ' ' | '\t' | '\n' | '\f' | '\r' )+ ;
EndOfLineComment    ::= '#' ('\n' | '\r')!* ('\n' | '\r') ;
TraditionalComment ::=
  "/*"
        ( ('*'! | '*'+ ('*' | '/')!)*
        | '*'!* '*' ('*' | ('*' | '/')! '*'!* '*')*
        )
  "*/" ;
```

`Mapper.syntax` defines the syntax of mapper files.

```
#
# Copyright 1&1 Internet AG, http://www.1and1.org
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU Lesser General Public License as published by
# the Free Software Foundation; either version 2 of the License,
# or (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
# See the GNU Lesser General Public License for more details.
#
# You should have received a copy of the GNU Lesser General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.
#

[PARSER]

#
# overall structure

Mapper              ::= MapperName SyntaxFile Imports Definitions ;
MapperName          ::= "mapper" Name ";" ;
SyntaxFile          ::= Grammar ;
Grammar             ::= "syntax" "=" StringLiteral ";" ;
Imports             ::= Import* ;
Import              ::= "import" PackageName ":" Class ("," Class)* ";";
Class               ::= Identifier ("->" Identifier)? ;
PackageName         ::= Name ;
Name                ::= Identifier ("." Identifier)* ;

#
# Constructors

Constructor         ::= ClassRef | MemberRef | Internal | Copy;
ClassRef            ::= Identifier ;
MemberRef           ::= Identifier "." Identifier ;
Internal            ::= "[" Identifier "]" ;
Copy                ::= "(" Identifier ")" ;

#
# Attributes

Definitions         ::= Group* ;
Group               ::= Symbol Attribute+ ;
Symbol              ::= StringSymbol | IdentifierSymbol ;
StringSymbol        ::= StringLiteral ;
IdentifierSymbol    ::= Identifier ;
Attribute           ::= AttributeName "=>" Constructor Visibility ;
AttributeName       ::= (":" Identifier)? ;
```

```
#
# visibility

Visibility          ::= Implicit | Explicit ;
Implicit            ::= ";" ;
Explicit            ::= ":" (">" Path)* ";" ;
Path                ::= ImplicitPath | LocalPath | NormalPath ;
ImplicitPath        ::= "\\\\*" ;
LocalPath           ::= Identifier ;
NormalPath          ::= Step+ ;
Step                ::= Move Identifier ;
Move                ::= Ups | Up | Downs | Down ;
Ups                 ::= "\\\\" ;
Up                  ::= "\\" ;
Downs               ::= "//" ;
Down                ::= "/" ;


[SCANNER]
    white = WhiteSpace, EndOfLineComment, TraditionalComment ;


StringLiteral      ::= '"' (('\\' 0..65535)
                         | ('\\' | '"' | '\n' | '\r')!)+ '"' ;

Identifier ::=
   ( 0x0024..0x0024 | 0x0041..0x005a | 0x005f..0x005f |
     0x0061..0x007a | 0x00a2..0x00a5 | 0x00aa..0x00aa |
     0x00b5..0x00b5 | 0x00ba..0x00ba | 0x00c0..0x00d6 |
     0x00d8..0x00f6 | 0x00f8..0x00ff )
   ( 0x0000..0x0008 | 0x000e..0x001b | 0x0024..0x0024 |
     0x0030..0x0039 | 0x0041..0x005a | 0x005f..0x005f |
     0x0061..0x007a | 0x007f..0x009f | 0x00a2..0x00a5 |
     0x00aa..0x00aa | 0x00b5..0x00b5 | 0x00ba..0x00ba |
     0x00c0..0x00d6 | 0x00d8..0x00f6 | 0x00f8..0x00ff )* ;

#
# white space

WhiteSpace         ::= ( ' ' | '\t' | '\n' | '\f' | '\r' )+ ;
EndOfLineComment   ::= '#' ('\n' | '\r')!* ('\n' | '\r') ;
TraditionalComment ::=
  '/' '*'
         ( ('*'! | '*'+ ('*' | '/')!)*
         | '*'!* '*' ('*' | ('*' | '/')! '*'!* '*')*
         )
  '*' '/' ;
```

# Index