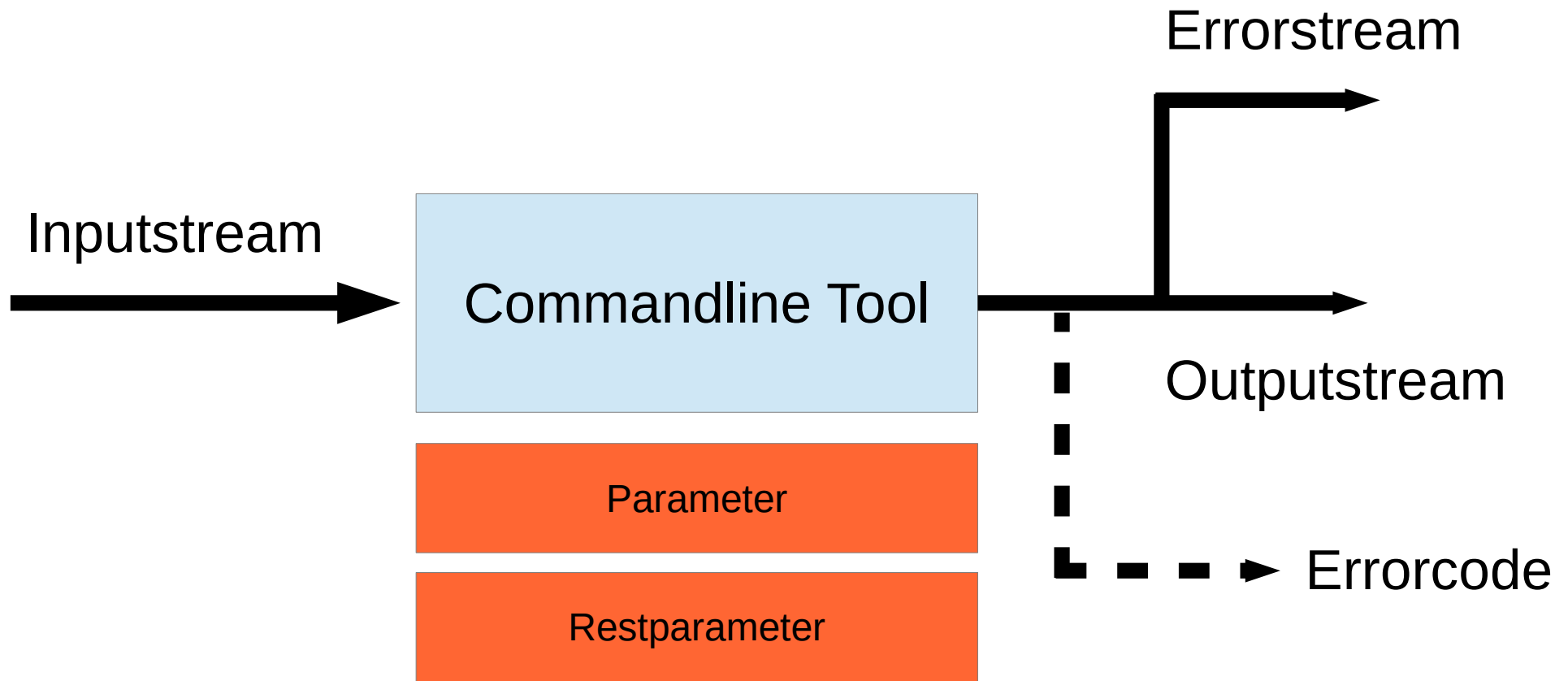


Kommandozeilenprogramme

Markus Fleischauer



Errorcode

- Jedes Programm gibt einen Integer aus, der angibt, ob das Programm mit einem Fehler abgestürzt ist
- Rückgabewert 0 bedeutet: kein Fehler
- Ansonsten kann man jedem Zahlenwert einen Fehler zuordnen
- In Java: `System.exit(int errorcode)`

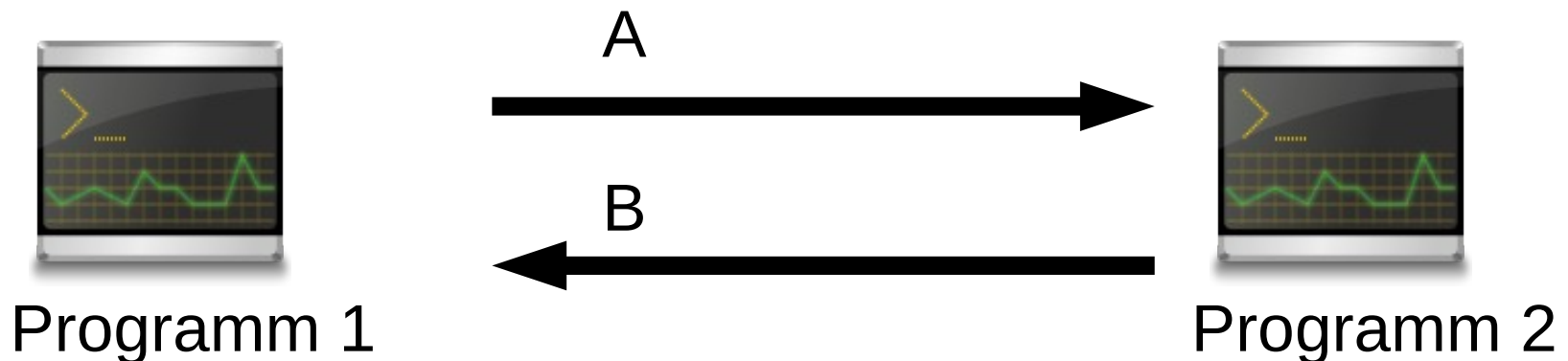
Streams

- “Fluss” von Bytes
- Anders als Array: Normalerweise kein wahlfreier Zugriff
- Stattdessen muss ein Stream kontinuierlich ausgelesen/geschrieben werden
- Potentiell “unendlich”



Streams

- Inputstream:
 - Der Stream, der vom Programm eingelesen wird
- Outputstream
 - Der Stream, in den das Programm hineinschreibt



Streams in Java

- Jedes Programm kennt drei Standardstreams:
 - Inputstream (die Eingabe) **System.in**
 - Outputstream (die Ausgabe) **System.out**
 - Errorstream (Fehlermeldungen und Warnungen) **System.err**
- in Konsolenanwendungen werden Outputstream und Errorstream “vermischt”.
- Eclipse und IntelliJ heben den Errorstream üblicherweise **rot** hervor

Streams und Reader/Writer

- Stream = Folge von Bytes (byte)

System.in

Streams und Reader/Writer

- Meistens will man aber nicht auf Bytes arbeiten, sondern auf Zeichen → *Streams müssen in Reader oder Writer gekapselt werden*
 - Reader/Writer = Folge von Zeichen (char)

```
InputStreamReader r = new InputStreamReader(System.in)
```

- Oft sinnvoll: nicht jedes Zeichen einzeln lesen, sondern mehrere auf einmal: `BufferedReader/BufferedWriter`

```
BufferedReader r = new BufferedReader(  
new InputStreamReader(in));
```


Streams und Dateien

- In Java sind die Klassen **FileInputStream** (Datei lesen) und **FileOutputStream** (Datei schreiben) die Schnittstelle zwischen Stream und Datei
- In der Konsole gibt es die **Redirection** Operatoren **<** und **>**

```
wc -l < eingabe.txt > ausgabe.txt
```

- *liest eingabe.txt ein, zählt die Anzahl an Zeilen, schreibt das Ergebnis in ausgabe.txt*

Streams und Dateien

- `2>` ist der Redirect Operator für den Errorstream
- `>>` bzw. `2>>` fügt den Output ans Ende der Datei an, statt die Datei zu überschreiben

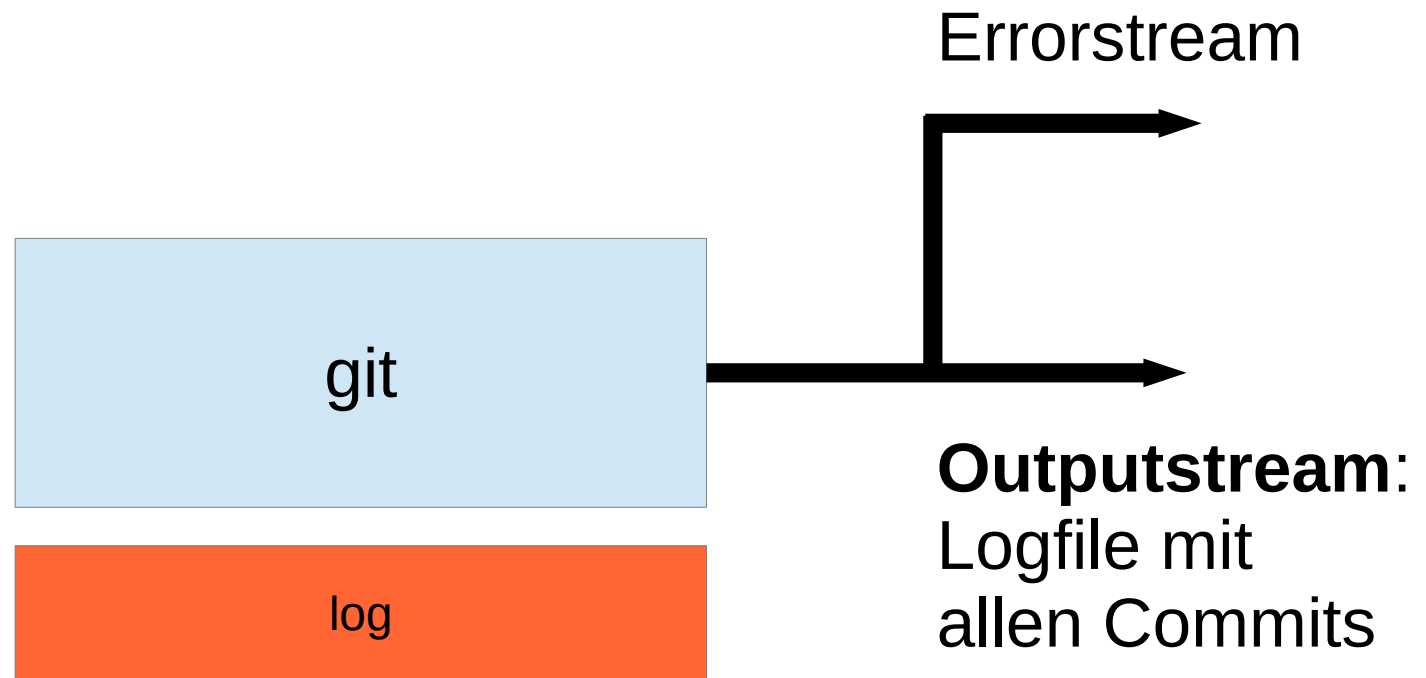
```
someprogram < someinput.txt 2> logfile.txt
```

- der Pipe-Operator `|` ermöglicht das Verbinden von Programmen:

```
program1 | program2
```

Diese kleine “Pipeline” zählt wie viele Commits ihr in eurem Repository gemacht habt und schreibt diese Zahl in eine Datei

```
git log | grep -e commit | wc -l >> gitcommits.txt
```



```
commit 0aad1e012ad2db5b298b1b680f7fcb6b97b5d49f
Author: Kai Dührkop <kai.duehrkop@uni-jena.de>
Date: Tue Nov 26 14:16:32 2013 +0100

profiles

commit 4e86dc28cd411daafab992ec8982ae547e0c02ac
Merge: 02bb05a b4c38ae
Author: Kai Dührkop <kai.duehrkop@uni-jena.de>
Date: Mon Nov 4 11:43:09 2013 +0100

Merge branch 'hotfix_dynamicilp'
Now, DP solver is used if ilp solver is not provided

commit 02bb05a48cb4c7ba6e25be7c2585fdd2cad1a1c7
Merge: 130c803 f66e34e
Author: Kai Dührkop <kai.duehrkop@uni-jena.de>
Date: Mon Nov 4 11:42:52 2013 +0100

uptodate

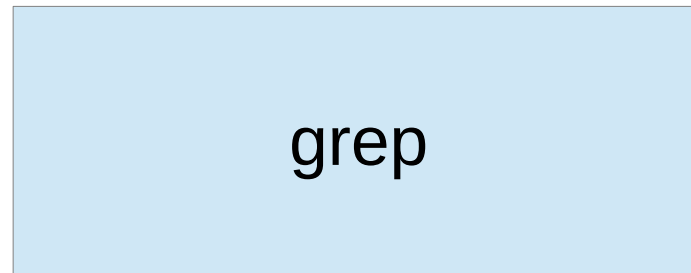
commit b4c38ae3ab11d49e3596bef613a86365eae6fd6d
Merge: 1c9505c f66e34e
```

Errorstream

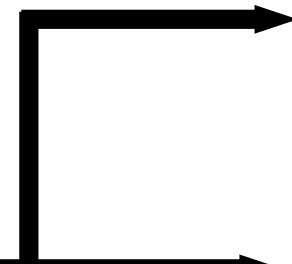
Inputstream:



ein beliebiger
Text



grep



Outputstream:

alle Zeilen im
Text, die den
Suchbegriff
beinhalten

-e commit

der Suchbegriff

```
commit 0aad1e012ad2db5b298b1b680f7fcb6b97b5d49f
Author: Kai Dührkop <kai.duehrkop@uni-jena.de>
Date: Tue Nov 26 14:16:32 2013 +0100

profiles

commit 4e86dc28cd411daafab992ec8982ae547e0c02ac
Merge: 02bb05a b4c38ae
Author: Kai Dührkop <kai.duehrkop@uni-jena.de>
Date: Mon Nov 4 11:43:09 2013 +0100

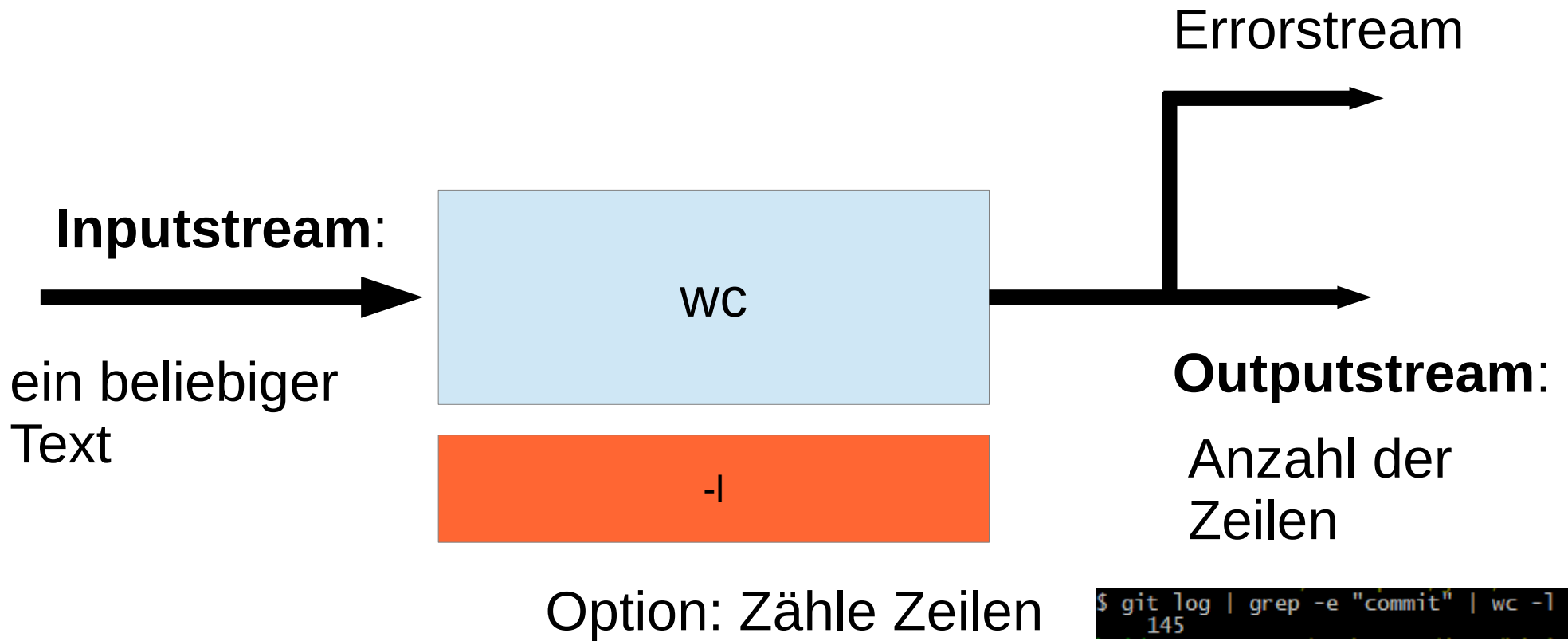
Merge branch 'hotfix_dynamicilp'
Now, DP solver is used if ilp solver is not provided

commit 02bb05a48cb4c7ba6e25be7c2585fdd2cad1a1c7
Merge: 130c803 f66e34e
Author: Kai Dührkop <kai.duehrkop@uni-jena.de>
Date: Mon Nov 4 11:42:52 2013 +0100

uptodate

commit b4c38ae3ab11d49e3596bef613a86365eae6fd6d
Merge: 1c9505c f66e34e
```

```
$ git log | grep -e "commit"
commit e4fa07501d954b013c045eaadbee362f0433a5a9
commit b1cdebddd966e29c61f205b42837a6f56ce3024a
commit 678d99cd4de9ab31524ff7f53cd59170855f7771
commit 61aad190ed75c3b2b4bbd1f15178adc464ebd200
commit 31d8582a1267385ebed435651ba635b39879d643
commit 91982466600e28ebcb5c4bfd24288145943ca1fe
commit f3268ed6c830be582580eaafd7dc3b393091ce22
commit 0aad1e012ad2db5b298b1b680f7fcb6b97b5d49f
commit 4e86dc28cd411daafab992ec8982ae547e0c02ac
```

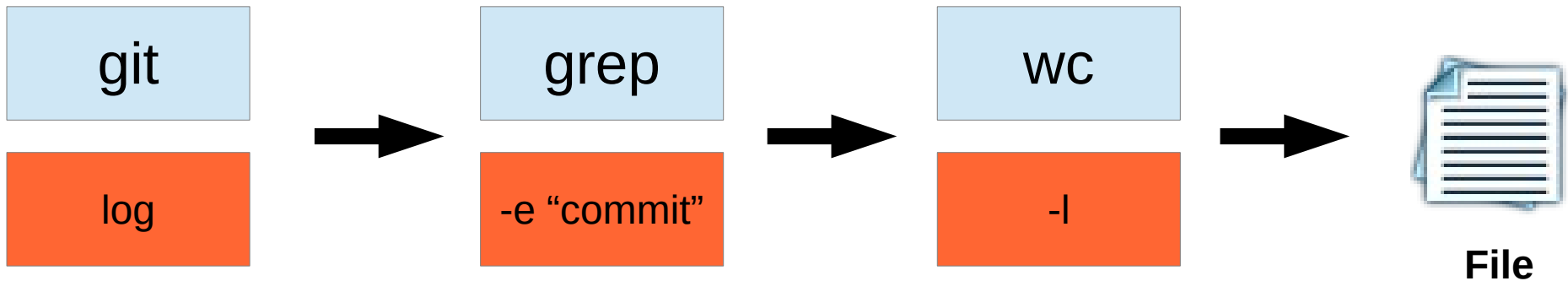


```
$ git log | grep -e "commit"  
commit e4fa07501d954b013c045eaadbee362f0433a5a9  
commit b1cdebddd966e29c61f205b42837a6f56ce3024a  
commit 678d99cd4de9ab31524ff7f53cd59170855f7771  
commit 61aad190ed75c3b2b4bbd1f15178adc464ebd200  
commit 31d8582a1267385ebed435651ba635b39879d643  
commit 91982466600e28ebcb5c4bfd24288145943ca1fe  
commit f3268ed6c830be582580eaafd7dc3b393091ce22  
commit 0aad1e012ad2db5b298b1b680f7fcb6b97b5d49f
```

```
$ git log | grep -e "commit" | wc -l  
145
```

`git log | grep -e "commit" | wc -l >> gitcommits.txt`

```
$ git log | grep -e "commit"
commit e4fa07501d954b013c045eaadbee362f0433a5a9
commit b1cdebddd966e29c61f205b42837a6f56ce3024a
commit 678d99cd4de9ab31524ff7f53cd59170855f7771
commit 61aad190ed75c3b2b4bbd1f15178adc464ebd200
commit 31d8582a1267385ebed435651ba635b39879d643
commit 91982466600e28ebcb5c4bfd24288145943ca1fe
commit f3268ed6c830be582580eaaafd7dc3b393091ce22
commit 0aad1e012ad2db5b298b1b680f7fcb6b97b5d49f
```



```
commit 0aad1e012ad2db5b298b1b680f7fcb6b97b5d49f
Author: Kai Dührkop <kai.duehrkop@uni-jena.de>
Date: Tue Nov 26 14:16:32 2013 +0100

profiles

commit 4e86dc28cd411daafab992ec8982ae547e0c02ac
Merge: 02bb05a b4c38ae
Author: Kai Dührkop <kai.duehrkop@uni-jena.de>
Date: Mon Nov 4 11:43:09 2013 +0100

Merge branch 'hotfix_dynamicilp'
Now, DP solver is used if ilp solver is not provided

commit 02bb05a48cb4c7ba6e25be7c2585fdd2cad1a1c7
Merge: 130c803 f66e34e
Author: Kai Dührkop <kai.duehrkop@uni-jena.de>
Date: Mon Nov 4 11:42:52 2013 +0100

uptodate

commit b4c38ae3ab11d49e3596bef613a86365eae6fd6d
Merge: 1c9505c f66e34e
```

```
$ git log | grep -e "commit" | wc -l
145
```

Kurzes Fazit

- Kommandozeilenprogramme ermöglichen das Verknüpfen verschiedener Programme über deren Input- und Outputstreams (solche verknüpften Programme nennt man **Pipelines**)
- komplexe Aufgaben lassen sich automatisieren, in dem man für jede Einzelaufgabe ein eigenes, kleines Kommandozeilenprogramm nutzt und diese dann miteinander verknüpft

Parameter

- Nachteil von Streams: Wir haben nur 1 Eingabe und 2 Ausgabestreams
- Das reicht uns nicht!
- viele Programme brauchen mehrere Eingaben
- daher nehmen sie Parameter entgegen

Parameter

- Jedes Programm erhält einen Array aus Strings als Parameter
- auf der Konsole werden diese Parameter nach dem Programmnamen eingegeben und mit Leerzeichen voneinander getrennt
- braucht man das Leerzeichen als Zeichen in einem Parameter, so muss man dieses escapen (mit \) oder den Parameter in "Anführungszeichen" schreiben

Parameter

```
myprogram parameter1 parameter2 parameter3
```

```
myprogram "parameter 1"
```

```
myprogram parameter\ 1
```

Arten von Parametern

- darüber hinaus gibt es Konventionen, wie Parameter für ein Programm auszusehen haben
- dies erleichtert die Bedienung des Programms für Außenstehende!
- verschiedene Programme sind auf die gleiche Weise zu bedienen

```
cp -r /paht
```

```
rm -r /path
```

– In beiden Fällen bedeutet das -r “rekursiv”

Boolscher Parameter

```
search --count
```

Es ist nur entscheidend, ob der Parameter gesetzt ist (**true**) oder nicht (**false**)

```
search -c
```

Viele Parameter erlauben “Kurzformen”. Die “Langform” fängt in der Regel mit zwei Bindestrichen an, die Kurzform besteht nur aus einem Buchstaben und fängt mit einem Bindestrich an

Boolscher Parameter

```
search --count --ignore --measure
```

```
search -c -i -m
```

```
search -cim
```

Manche Programme erlauben das Zusammenfassen von boolschen Parametern.

Value Parameter

```
search --algorithm=KMP --frame=3
```

```
search --algorithm KMP --frame 3
```

```
search --algorithmKMP --frame3
```

Letztere Form wird nicht von allen Programmen unterstützt.

Value Parameter

```
search --algorithm="Knuth Morris Pratt"
```

Wenn Parameter Leerzeichen enthält, muss dieser escaped oder in Anführungszeichen geschrieben werden

Rest-Parameter / Argumente

```
search -p "ACAG" somedir/*.fasta
```

- Viele Programme akzeptieren am Ende ihrer Parameterliste eine Folge von Files (oder beliebigen anderen Strings)
- diese können entweder nacheinander eingetippt, oder über Bash-Wildcards erzeugt werden: *.fasta
- jedes File ist ein Parameter in der Parameterliste

Subcommands

- Widersprechen eigentlich dem Prinzip, Kommandozeilenprogramme klein und einfach zu halten

`git log`

`git mv`

`git clone`

`git rm`

- Uebliche kommandos koennen wieder verwendet werden

Subcommands

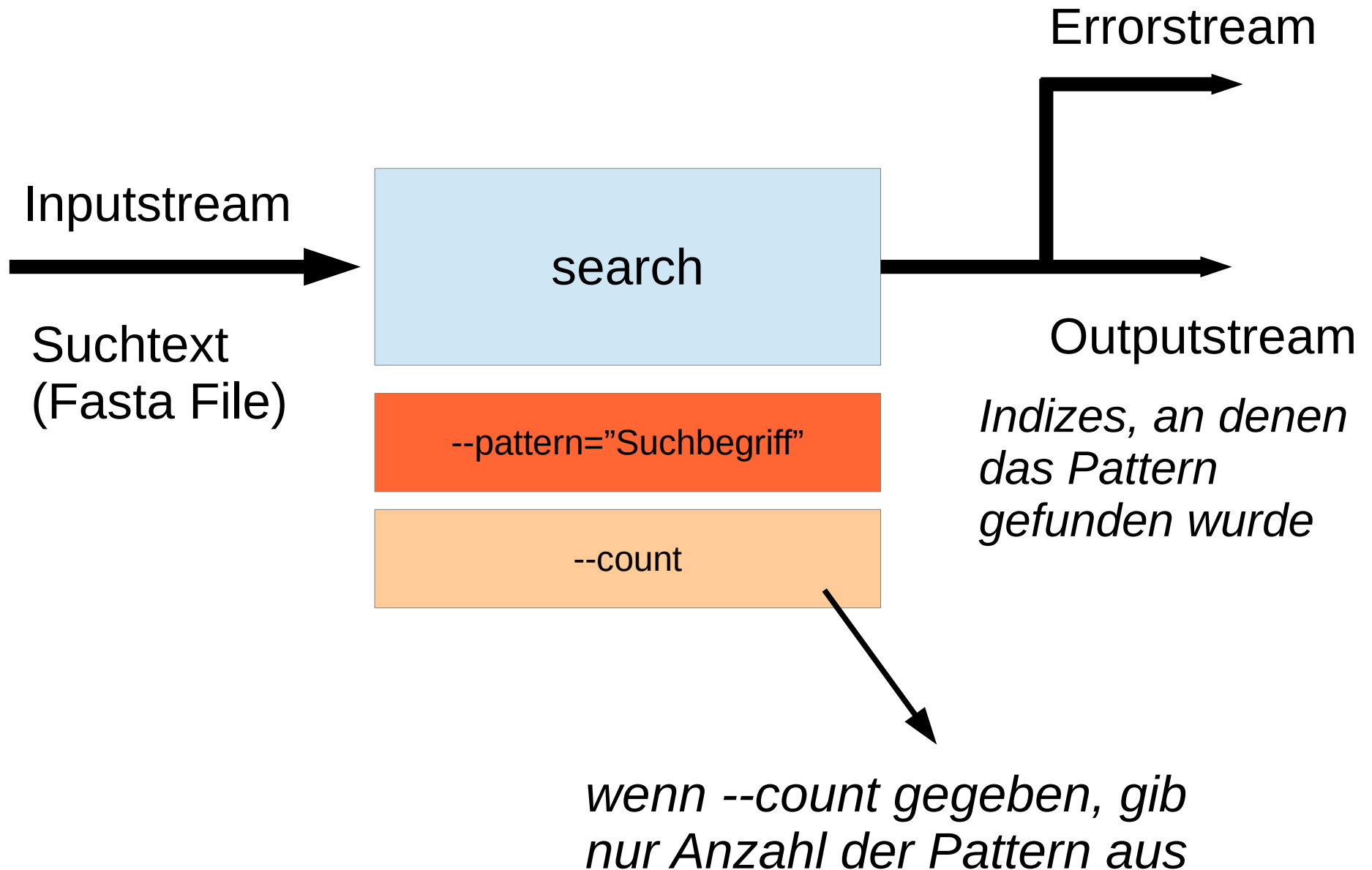
- Interessant für „toolboxen“

```
seq-tools --workingDir=/path/ search --algo=KMP
```

- Parameter können sinnvoll geordnet und so als allgemeine Parameter und Subcommand Parameter unterteilt werden (“Parameter von Parametern”)

- Das alles sind nur Konventionen!
- Java-Tools (javac, java, jar, ...) nutzen z.B. immer nur ein Bindestrich für ihre Parameter
- Übliche (allgemeine) Parameter:
 - -v und --verbose für: Informative Ausgaben
 - -h und --help für: Gib Dokumentation zu Kommandozeilenprogramm aus
 - --version für: Gib Programmversion aus

Beispiel: Exakte Suche



Beispiel: Exakte Suche

```
search < sequence.fasta -p ACCGACTA -c
```

```
search < sequence.fasta -p ACCGACTA > indices.txt
```

Umsetzung in Java

- Einfache Kommandozeilentools kann man leicht selbst schreiben
- Alternativ gibt es viele Bibliotheken, die einem diese Arbeit erleichtern.
- Für java: simpleopts, jewelcli, args4j, JSAP, **picocli**,...

Ohne Bibliothek

```
enum Algorithm {NAIVE, KMP, BM}

public static void main(String[] args) {

    Algorithm algorithm = Algorithm.NAIVE;
    boolean count = false;

    for (int i = 0; i < args.length; ++i) {
        String opt = args[i];
        if (opt.equals("-h") || opt.equals("--help")) {
            printHelp();
        } else if (opt.equals("--algorithm") || opt.equals("-a")) {
            String value = args[++i];
            try {
                algorithm = Algorithm.valueOf(value.toUpperCase());
            } catch (IllegalArgumentException e) {
                System.err.println("Unknown algorithm " + value);
                System.err.println("Use one of NAIVE, KMP or BM");
            }
        } else if (opt.equals("-c") || opt.equals("--count")) {
            count = true;
        } else if ...
    }
}
```


Mit Bibliothek: picocli

- Doku: <https://picocli.info/>
 - enthält Tutorial mit Beispielen
- GitHub: <https://github.com/remkop/picocli/>
- Gradle: `compile 'info.picocli:picocli:4.0.4'`

- Man definiert seine Kommandozeilenoptionen als Methoden oder Felder mit **Annotationen**
- Strings werden automatisch in Datentypen geparsed
- Eine „Hilfe“ mit Parameter Übersicht und Beispielen werden automatisch generiert

picocli

- Ihr definiert eine Klasse mit Annotierten Methoden oder Feldern und erzeugt ein Instanz dieser Klasse.
- Zur Laufzeit wird einer Parser Klasse (Commandline.java) das args[] array und die o.g. Instanz übergeben und geparsed.
- Das Ergebnis wird an die entsprechenden Felder geschrieben bzw. annotierte Methoden ausgeführt

Klasse mit Parameter Annotationen

```
1 package de.unijena.bioinf.ms;
2
3 import picocli.CommandLine;
4
5 import java.io.File;
6
7 public class TarOptions {
8     @CommandLine.Option(names = "-c", description = "create a new archive")
9     public boolean create;
10
11     @CommandLine.Option(names = { "-f", "--file" }, paramLabel = "ARCHIVE", description = "the archive file")
12     public File archive;
13
14     @CommandLine.Parameters(paramLabel = "FILE", description = "one ore more files to archive")
15     public File[] files;
16
17     @CommandLine.Option(names = { "-h", "--help" }, usageHelp = true, description = "display a help message")
18     private boolean helpRequested = false;
19 }
```

Parsing Beispiel

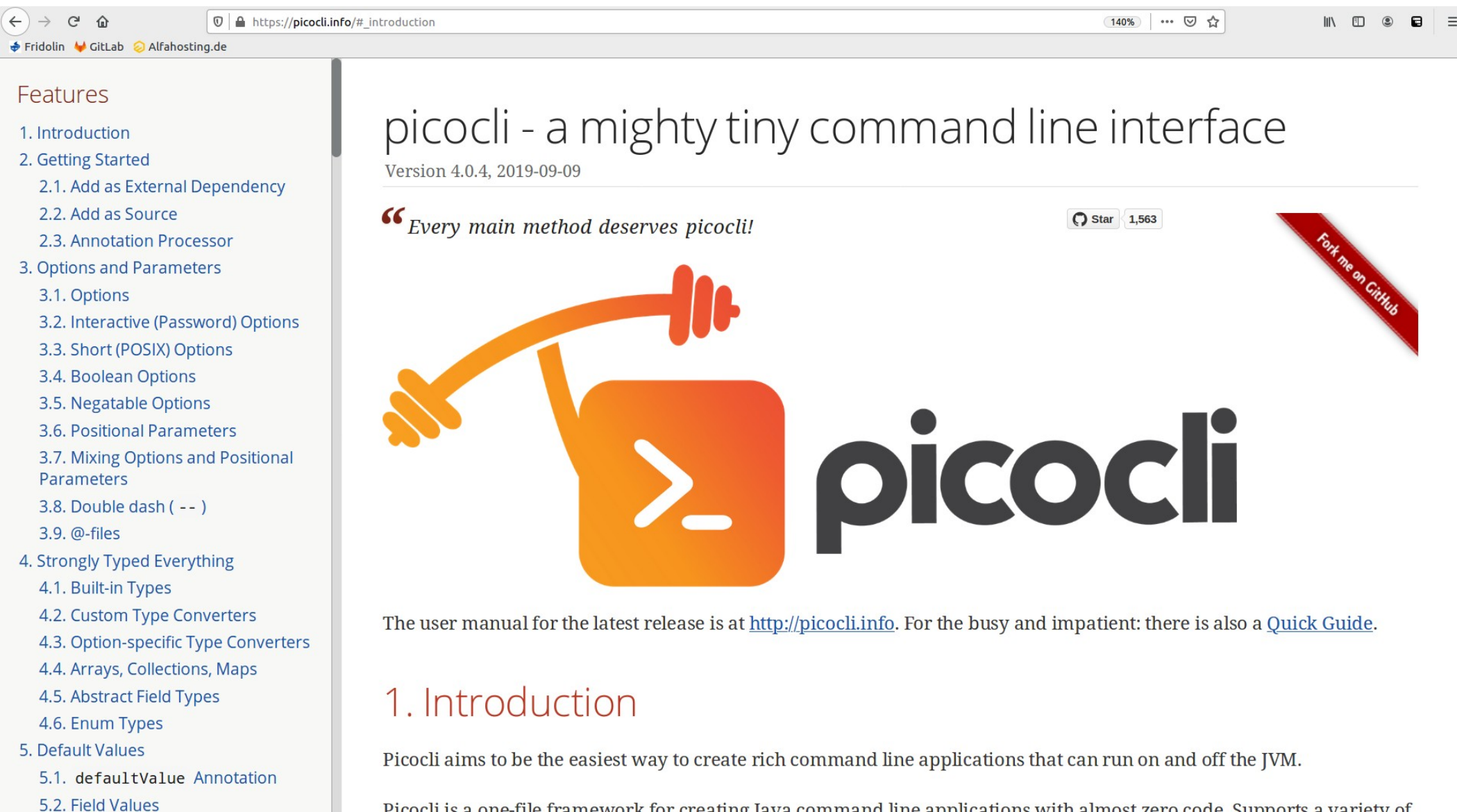
```
1 package de.unijena.bioinf.ms;
2
3 import picocli.CommandLine;
4
5 ▶ public class MyTarApp {
6 ▶   public static void main (String[] args) {
7       TarOptions options = new TarOptions();
8       CommandLine cli = new CommandLine(options);
9       CommandLine.ParseResult result = cli.parseArgs(args);
10
11       if (result.isUsageHelpRequested())
12           cli.usage(System.err);
13       else {
14           //todo do some useful stuff
15       }
16   }
17 }
```

Hilfe!

```
myTarApp -h
```

```
Usage: <main class> [-ch] [-f=ARCHIVE] [FILE...]
      [FILE...]      one ore more files to archive
-c                create a new archive
-f, --file=ARCHIVE  the archive file
-h, --help         display a help message
```

Fuer alles weitere...



The screenshot shows a web browser window displaying the introduction page for picocli. The browser's address bar shows the URL https://picocli.info/#_introduction. The page features a navigation sidebar on the left with a 'Features' section containing a list of topics from 'Introduction' to 'Field Values'. The main content area has a title 'picocli - a mighty tiny command line interface' and a version string 'Version 4.0.4, 2019-09-09'. A quote reads 'Every main method deserves picocli!' with a 'Star 1,563' button and a 'Fork me on GitHub' ribbon. The picocli logo, an orange hand holding a terminal icon, is displayed next to the word 'picocli'. Below the logo, the text states that the user manual is at <http://picocli.info> and mentions a 'Quick Guide'. The first section, '1. Introduction', begins by stating that Picocli aims to be the easiest way to create rich command line applications that can run on and off the JVM. The final sentence of the introduction is partially visible: 'Picocli is a one-file framework for creating Java command line applications with almost zero code. Supports a variety of'.

Features

1. Introduction
2. Getting Started
 - 2.1. Add as External Dependency
 - 2.2. Add as Source
 - 2.3. Annotation Processor
3. Options and Parameters
 - 3.1. Options
 - 3.2. Interactive (Password) Options
 - 3.3. Short (POSIX) Options
 - 3.4. Boolean Options
 - 3.5. Negatable Options
 - 3.6. Positional Parameters
 - 3.7. Mixing Options and Positional Parameters
 - 3.8. Double dash (--)
 - 3.9. @-files
4. Strongly Typed Everything
 - 4.1. Built-in Types
 - 4.2. Custom Type Converters
 - 4.3. Option-specific Type Converters
 - 4.4. Arrays, Collections, Maps
 - 4.5. Abstract Field Types
 - 4.6. Enum Types
5. Default Values
 - 5.1. `defaultValue` Annotation
 - 5.2. Field Values


picocli - a mighty tiny command line interface

Version 4.0.4, 2019-09-09

“Every main method deserves picocli!”

Star 1,563

Fork me on GitHub



picocli

The user manual for the latest release is at <http://picocli.info>. For the busy and impatient: there is also a [Quick Guide](#).

1. Introduction

Picocli aims to be the easiest way to create rich command line applications that can run on and off the JVM.

Picocli is a one-file framework for creating Java command line applications with almost zero code. Supports a variety of

<https://picocli.info/>