simple-escp Documentation

Table of Contents

Download PDF version[1]

1. Getting Started

simple-escp is a small library for simplyfing ESC/P text-mode printing in Java. Many modern reporting libraries for Java works in graphic mode. While modern dot-matrix and receipt printers can print in graphics mode, printing graphic is slow. It is also cumbersome to print at exact position in graphic mode.

Such problems can be avoided by using text mode printing. In text mode printing, printers will print ASCII characters one by one. They will usually use fonts stored inside their RAM and accept control characters for manipulating printing process.

ESC/P (Espon Standard Code for Printers) is a printer control language mainly used in dot matrix and receipt printers. ESC/P commands can be used to configure page

[1] http://jockihendry.github.io/simple-escp/simple-escp-doc.pdf

length, change font styles, and many more configurations. simple-escp supports ESC/P for 9-pin printers.

To use simple-escp in Gradle, add the following lines to `build.gradle`:

build.gradle

```
repositories {
    maven {
        url "http://dl.bintray.com/jockihendry/maven"
    }
}
dependencies {
    compile group: 'jockihendry', name: 'simple-escp', version: '0.4'
}
```

To include simple-escp manually to a Java project, download simple-escp's binary distribution from https://github.com/JockiHendry/simple-escp/releases.

The first step in using simple-escp is creating a template in JSON format. This template contains page format attributes and placeholders that will be filled later in application. For example, create `template.json` that has the following content:

template.json

```
{
    "pageFormat": {
        "pageWidth": 50,
  ❶
        "pageLength": 13,
  ❷
        "usePageLengthFromPrinter": false
  ❸
    },
    "template": {
        "header": [
            "   ###       Company Name                 Page %{PAGE_NO}",
  ❹
            " #######                                                 ",
            "#########    Invoice No: ${invoiceNo:10}                 "
  ❺
        ],
        "detail": [
            {
                "table": "table_source",
  ❻
                "border": true,
                "columns": [
                    { "source": "code", "width": 9, "caption": "Code" },
  ❼
                    { "source": "name", "width": 34, "caption": "Name" },
                    { "source": "qty",  "width": 6, "caption": "Qty"   }
                ]
            },
            "                                                  ",
            "   (Signature)                  (Signature)       "
        ]
    }
}
```

❶ This report's width is 50 characters.

❷ Every page consists of 13 lines.

❸ Overrides page length settings in printer's ROM so that form feed will work as expected.

❹ `%{PAGE_NO}` is a function that will be replaced by page number.

❺ `${invoiceNo:10}` is a placeholder that will be replaced by value from data source. It will be exactly 10 characters.

❻ This will create a table based on a collection.

❼ This is used to configure every table's column.

The next step is to read the JSON template form Java code, for example:

Main.java

```java
JsonTemplate template = new JsonTemplate(getClass().getResource("/
template.json").toURI();
```

Template need to be filled with values from one or more data sources (unless it doesn't contains any placeholders). Data source can be stored as `Map` or JavaBean object. For example, the following code will define a `Map` data source:

Main.java

```java
Map<String, Object> map = new HashMap<>();
map.put("invoiceNo", "INVC-00001");                                  ❶
List<Map<String, Object>> tables = new ArrayList<>();
for (int i=0; i<5; i++) {
    Map<String, Object> line = new HashMap<>();
    line.put("code", String.format("CODE-%d", i));
    line.put("name", String.format("Product Random %d", i));
    line.put("qty", String.format("%d", i*i));
    tables.add(line);
}
map.put("table_source", tables);                          ❷
```

❶ This value will be used to fill `${invoiceNo:10}`.

❷ This is used by `"table": "table_source"`. Every items in this collection will represent a line in table.

The last step is to fill the template and print the result to printer:

Main.java

```
SimpleEscp simpleEscp = new SimpleEscp();
simpleEscp.print(template.parse(), map);  ❶
```

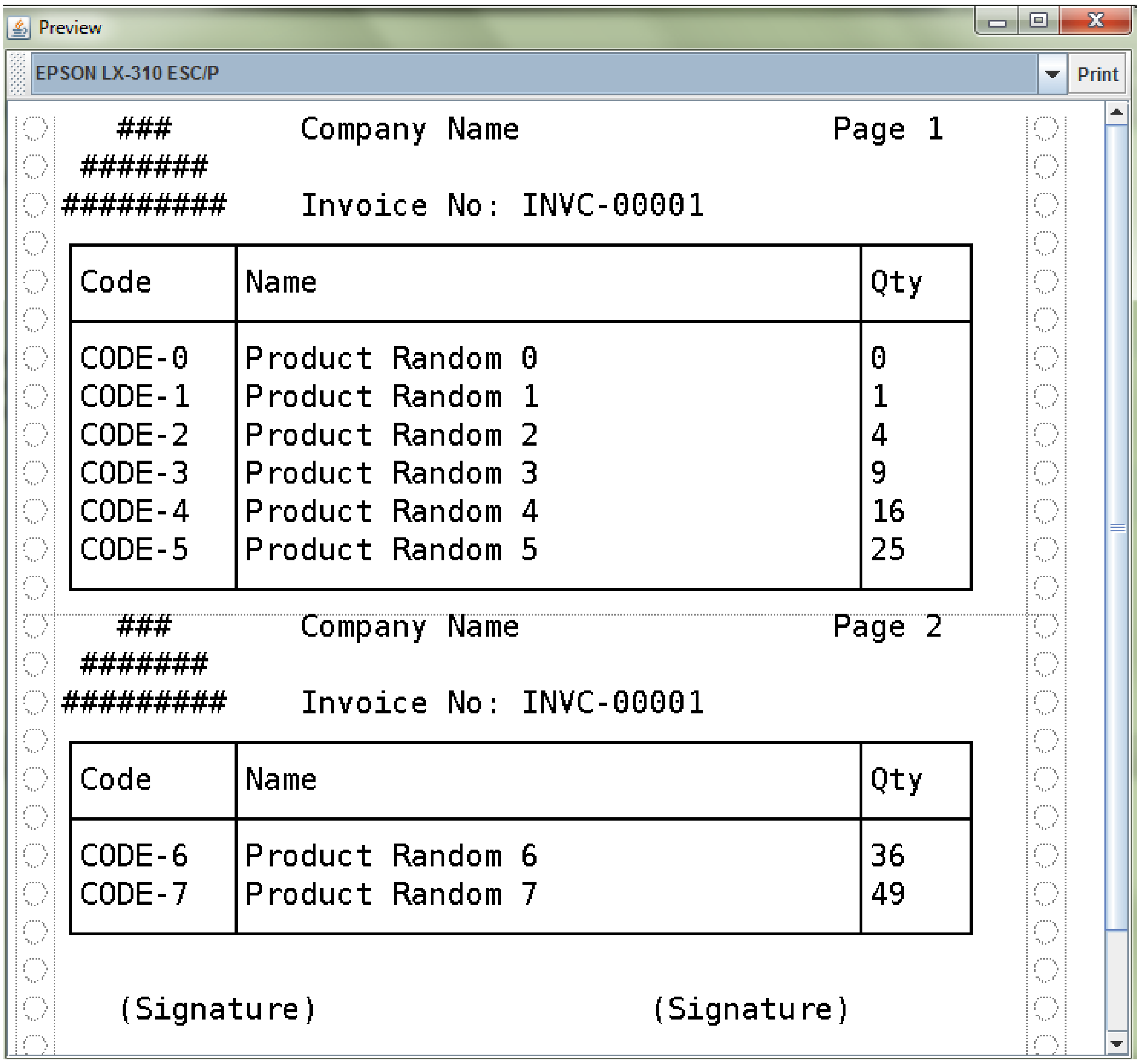
❶ Print directly to default printer.

simple-escp has a preview panel that can be used in Swing application to preview the result. For example, the following is a complete code that will read a JSON template, fill it with data and displays the result:

MainFrame.java

```
import simple.escp.Template;
import simple.escp.json.JsonTemplate;
import javax.swing.JFrame;
import java.awt.BorderLayout;
import java.awt.Dimension;
import java.io.IOException;
import java.net.URISyntaxException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class MainFrameTest extends JFrame {

    public MainFrameTest() throws URISyntaxException, IOException {
        super("Preview");

        Template template = new JsonTemplate(Thread.currentThread().
            getContextClassLoader().getResource("report.json").toURI());
❶

        Map<String, Object> value = new HashMap<>();
❷
        value.put("invoiceNo", "INVC-00001");
        List<Map<String, Object>> tables = new ArrayList<>();
        for (int i=0; i<5; i++) {
            Map<String, Object> line = new HashMap<>();
            line.put("code", String.format("CODE-%d", i));
            line.put("name", String.format("Product Random %d", i));
            line.put("qty", String.format("%d", i*i));
            tables.add(line);
        }
        value.put("table_source", tables);

        PrintPreviewPane printPreview = new PrintPreviewPane(template,
            value, null);
❸

        getContentPane().setLayout(new BorderLayout());
        getContentPane().add(printPreview, BorderLayout.CENTER);
❹


        setPreferredSize(new Dimension(500, 500));
        pack();
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    public static void main (String[] args) {
        try {
            new MainFrameTest();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```

❶ Read the JSON template.
❷ Prepare data.
❸ Create a preview panel.
❹ Add the preview panel to this frame.



```
   ###       Company Name                Page 1
 #######
#########    Invoice No: INVC-00001
```

Code	Name	Qty
CODE-0	Product Random 0	0
CODE-1	Product Random 1	1
CODE-2	Product Random 2	4
CODE-3	Product Random 3	9
CODE-4	Product Random 4	16
CODE-5	Product Random 5	25

```
   ###       Company Name                Page 2
 #######
#########    Invoice No: INVC-00001
```

Code	Name	Qty
CODE-6	Product Random 6	36
CODE-7	Product Random 7	49

```
    (Signature)                  (Signature)
```

Figure 1. The Preview Panel

2. Template

JSON template consists of 2 keys: `pageFormat` and `template`.

2.1. Page Format

The value of `pageFormat` must be a JSON object. The following keys are available for `pageFormat`:

Key	Value Type	Default Value
`"autoFormFeed"`	boolean	`true`
`"autoLineFeed"`	boolean	`false`
`"bottomMargin"`	number	*undefined*

Key	Value Type	Default Value
`"characterPitch"`	number, string	`"10 cpi"`
`"leftMargin"`	number	*undefined*
`"lineSpacing"`	string	`"1/6"`
`"pageLength"`	number	*undefined*
`"pageWidth"`	number	*undefined*
`"rightMargin"`	number	*undefined*
`"typeface"`	string	*undefined*
`"usePageLengthFromPrinter"`	boolean	`true`

Example:

```
{
    "pageFormat": {
        "characterPitch": "5",
        "lineSpacing": "1/8",
        "typeface": "roman"
    }
}
```

Page format can also be created programmatically, for example:

```
PageFormat pageFormat = new PageFormat();
pageFormat.setCharacterPitch("5");
pageFormat.setLineSpacing("1/8");
pageFormat.setTypeface("roman");
```

2.2. Template

The value of `template` can be JSON object or JSON array.

If `template` is a JSON array, every elements of the array is a single line. The first element of the array is the first line, the second element is the second line, and so on. For example, the following `template` consists of three lines:

```
{
    "template": ["Line #1", "Line #2", "Line #3"]
}
```

The template above can also be create programmatically by using the following code:

```
Report report = new Report(pageFormat, null, null);
report.append(new TextLine("Line #1"), false);
report.append(new TextLine("Line #2"), false);
report.append(new TextLine("Line #3"), false);
```

If `template` is a JSON object, every members of the object represents a report section. The following keys are available for `template`:

Key	Description
`"detail"`	Put the content of the report in this section.
`"firstPage"`	This section will be displayed as the first page of the report.
`"footer"`	This section will be added at the bottom of every page in `"detail"` section.
`"header"`	This section will be added at the top of every page in `"detail"` section.
`"lastPage"`	This section will be displayed as the last page of the report.

All sections must have a JSON array as value. Every elements of the array is a single line in that section. Example:

```
{
    "template": {
        "firstPage": ["First page only."],
        "header": ["Header line #1", "Header line #2"],
        "detail": [
            "First line of detail",
            "Second line of detail"
        ],
        "footer": ["The footer."],
        "lastPage": ["Last page only."]
    }
}
```

The template above can also be create programmatically by using the following code:

```
TextLine[] firstPage = new TextLine[] { new TextLine("First page
 only.") };
TextLine[] lastPage = new TextLine[] { new TextLine("Last page only.") };
TextLine[] header = new TextLine[] {
    new TextLine("Header line #1"),
    new TextLine("Header line #2")
};
TextLine[] footer = new TextLine[] { new TextLine("The footer.") };
Report report = new Report(pageFormat, header, footer);
report.appendSinglePage(firstPage, true);
report.append(new TextLine("First line of detail"), false);
report.append(new TextLine("Second line of detail"), false);
report.appendSinglePage(lastPage, true);
```

2.3. Line

`template` and all sections accept an array that represent lines. The elements inside this array should be a string or JSON object. String will be converted to `TextLine`. All `TextLine` may contains placeholders and/or functions. JSON object will be converted to either `TableLine` or `ListLine` depending on their keys.

`detail` section allows mixing multiple `TextLine`, `TableLine` and `ListLine` in any position inside the array.

2.4. Table

Table is a JSON object that contains `table` key. The value for `table` is script placeholder that must be evaluated to `Collection`. All valid keys for table are:

Key	Value Type	Required	Description
`"border"`	boolean		If `true`, simple-escp will add CP437 border to this table. Default value is `false`.
`"columns"`	array		The columns for this table.
`"table"`	string		A script that should return `Collection` as the content of this table.

Every element in `columns` array is a JSON object. The valid keys for this JSON object are:

Key	Value Type	Required	Description
`"caption"`	string		The column name. Default value is the same as value of `"source"`.
`"source"`	string		A script that will be executed for members of Collection to return the value for this column.
`"width"`	number		Width of this column in number of characters.
`"wrap"`	boolean		If `true`, value that exceeds column's width will be advanced to next line. Default value is `false`.

Example of table in JSON template:

```
{
    "pageFormat": {
        "pageLength": 10
    },
    "template": [
        "This is a text line",                                              ❶
        {                                                                   ❷
            "table": "persons",
            "columns": [
                { "source": "firstName", "width": 10, "wrap": true },
                { "source": "lastName", "width": 20 },
                { "source": "nickname", "width": 10 }
            ]
        },
        "This is a text line"                                               ❸
    ]
}
```

❶ A normal text line that will be displayed before the table.
❷ Table is a JSON object that have `"table"` as key.
❸ A normal text line that will be displayed after the table.

If table can't fit in one page, following pages that display the rest of table will have column's name and full border (if border is enabled).

A report can have more than one table.

The `source` key in table’s column can have the following predefined value:

Key	Value Type	Description
`"col"`	number	The column number for current column, starting from `1` for the first column.
`"row"`	number	The row number for current row, starting from `1` for the first row.

For example:

```
{
    "pageFormat": {
        "pageLength": 10
    },
    "template": [
        {
            "table": "persons",
            "columns": [
                { "source": "row", "width": 4, "caption": "No" },        ❶
                { "source": "firstName", "width": 10, "wrap": true }
            ]
        }
    ]
}
```

❶ `"row"` will be evaluated to the current row number.

2.5. List

Like table, list is also used to display collection. The difference is list doesn’t have columnar layout. It can be treated as a collection of similiar lines. All valid keys for list are:

Key	Value Type	Required	Description
`"footer"`	array		Footer that will be displayed in the end of each page if list spans multiple pages.
`"header"`	array		Header that will be displayed in the beginning of each page if list spans multiple pages.
`"line"`	string		Text line that will be used to evaluate every lines of this list.

Key	Value Type	Required	Description
`"list"`	string		A script that should return `Collection` as the content of this list.

Example of list in JSON template:

```
{
    "pageFormat": {
        "pageLength": 10
    },
    "template": [
        "This is a text line",                                              ❶
        {                                                                   ❷
            "list": "persons",
            "line": "${firstName} ${lastname} or ${nickname}",
            "header": [ "List of persons:" ]
        },
        "This is a text line"                                               ❸
    ]
}
```

❶ A normal text line that will be displayed before the list.

❷ List is a JSON object that have `"list"` as key.

❸ A normal text line that will be displayed after the list.

3. Placeholder

All text lines may contain a placeholder in form of `${…}`. Placeholders will be substituted by values from data sources during filling process. simple-escp supports two kinds of placeholder: basic placeholder and script placeholder.

All placeholders supports common configurations. In basic placeholder, the common configurations are separated by `:` such as `${name:10:left}`. In script placeholder, they are separated by `::` such as `{{firstName + " " + lastName::10::left}}`.

The following is list of available configurations for placeholder:

Type	Possible Value	Description
Aggregation	`sum`, `count`	Can be used only in collection that contains number.

Type	Possible Value	Description
Alignment	`left`, `right`, `center`	Determine the alignment of value if number of characters is less than width.
Format	`number`, `integer`, `currency`, `date_full`, `date_long`, `date_medium`, `date_short`	Format the value based on the specified formatter.
Width	Number	The number of characters for value. If number of characters is less than this number, value will be filled by spaces. If number of characters is more than this number, value will be truncated.

The configurations can be in any orders. For example, `${name:20:left}` is equals to `${name:left:20}`.

3.1. Basic Placeholder

Basic placeholder is defined by using the following syntax: `${…}`.

The content of basic placeholder is simply a string that refers to a member of data source. For example, `${name}` will refer to `map.get("name")` if the data source is a map or `object.getName()` if the data source is a JavaBean object.

For JavaBean object data source, basic placeholder supports nested attributes. For example, `${name.firstName}` will refer to `object.getName().getFirstName()`. Basic placeholder also supports method call by prepending the method's name with `@`. For example, `${@fullName}` will refer to `object.fullName()`.

Example:

```
{
    "pageFormat": {
        "pageLength": 10
    },
    "template": [
        "First name  :  ${firstName:20}",
        "Last name   :  ${lastName:20}",
        "Address 1   :  ${address.line1:20}",
        "Address 2   :  ${address.line2:20}",
        "Total       :  ${@total:20:currency}"
    ]
}
```

3.2. Script Placeholder

Script placeholder is defined by using the following syntax: `{{…}}`.

The content of script placeholder will be evaluated by script engine. simple-escp uses JSR 223: Scripting for the Java Platform API to evaluates the content of script placeholder. If Groovy script engine is available, simple-escp will use it. Otherwise simple-escp will use the default JavaScript engine bundled in JDK.

Script can refer to any members of data source by their name. Script may use special variable `bean` to refer to JavaBean data source if it is available.

Example:

```
{
    "template": [
        "First name  :  {{ firstName :: 20 }}",
        "Last name   :  {{ lastName :: 20 }}",
        "Address 1   :  {{ address.getLine1() :: 20 }}",
        "Address 2   :  {{ address.getLine2() :: 20 }}",
        "Total       :  {{ bean.total() :: 20 :: currency}}"
    ]
}
```

User can also add custom variables to script's engine context, for example:

```
FillJob job = new FillJob(report, dataSource);
job.addScriptVariable("prefix", "Mr");            ❶
String result = job.fill();
```

❶ Can be used in script placeholder, for example: `{{prefix + " " + firstName}}`

4. Function

To call function in text lines, use the following syntax: `%{…}`.

4.1. Numbering Functions

The following is list of functions that return number.

Name	Example	Description
GLOBAL_LINE_NO	`%{GLOBAL_LINE_NO}`	Return the current global line number.
INC	`%{INC A}`, `%{INC B}`	Create a global number variable that start from `1` and return its value. The subsequent invocations of this function will increase the variable by `1`.
LINE_NO	`%{LINE_NO}`	Return the current line number. Line number will reset to `1` when encountering a new page.
PAGE_NO	`%{PAGE_NO}`	Return the current page number.

Example:

```
{
    "template": {
        "header": ["Page %{PAGE_NO}"],
        "detail": [
            "%{LINE_NO} This is the content."
        ]
    }
}
```

4.2. Styling Functions

Styling functions are used to generate ESC/P codes to change font style for a portion of text. They are commonly used in form of `%{…} text %{…}`.

The following is list of functions used for changing font style:

Name	Example	Description
BOLD	`%{BOLD}text{%BOLD}`	Bold font style.
DOUBLE	`%{DOUBLE}text{%DOUBLE}`	Double-strike font style.

Name	Example	Description
ITALIC	`%{ITALIC}text{%ITALIC}`	Italic font style.
SUB	`%{SUB}text{%SUB}`	Subscript font style.
SUP	`%{SUP}text{%SUP}`	Superscript font style.
UNDERLINE	`%{UNDERLINE}text%{UNDERLINE}`	Underline font style.

Example:

```
{
    "template": {
        "detail": [
            "%{BOLD}bold text{%BOLD} and %{ITALIC}italic text{%ITALIC}."
        ]
    }
}
```

4.3. Miscellaneous Functions

To generate an ASCII character, call function with ASCII number. For example:

```
{
    "template": {
        "detail": [
            "%{176} %{177} %{178}"
        ]
    }
}
```

ASCII function can also be used to repeat characters. For example:

```
{
    "template": {
        "detail": [
            "%{177 R10}."      ❶
        ]
    }
}
```

❶ Create 10 characters that consist of ASCII character 177.

4.4. Custom Function

User can also create custom functions by extending `Function` class. The following is an example of custom function:

```
public class CustomFunction extends Function {

    public CustomFunction() {
        super("%\\{\\s*(MY_CUSTOM)\\s*\\}");
            ❶
    }

    @Override
    public String process(Matcher matcher, Report report, Page page, Line
 line) {
        return "MyCustomResult";
            ❷
    }

    @Override
    public void reset() {
        // do nothing
    }

}
```

❶ This function will replace all occurences of `%{MY_CUSTOM}`.

❷ It always return `MyCustomResult`.

To make simple-escp recognize this function , it must be added to `FillJob` by using code such as:

```
CustomFunction customFunction = new CustomFunction();
FillJob.addFunction(customFunction);                          ❶
```

❶ `customFunction` will be available globally.

5. Data Source

In simple-escp, a data source is an implementation of `DataSource`. By default, simple-escp shipped with two default implementation: `MapDataSource` to retrieve value from Map and `BeanDataSource` to retrieve value from JavaBean object.

The following code shows how to create data source:

```
MapDataSource mapDataSource = new MapDataSource(map);
BeanDataSource beanDataSource = new BeanDataSource(bean);
FillJob fillWithMap = new FillJob(report, mapDataSource);
FillJob fillWithBean = new FillJob(report, beanDataSource);
FillJob fillMultipleSources = new FillJob(report,
    new DataSource[]{mapDataSource, beanDataSource});
```

User can also directly create `DataSource` from Map or JavaBean object by using `DataSources` factory, for example:

```
FillJob fillWithMap = new FillJob(report, DataSources.from(map));
FillJob fillWithMap = new FillJob(report, DataSources.from(bean));
FillJob fillMultipleSources = new FillJob(report, DataSources.from(map,
 bean));
```

6. Reference

- Source code: https://github.com/JockiHendry/simple-escp
- Javadoc: http://jockihendry.github.io/simple-escp/javadoc/index.html