Tutorial

Stage 11
April 27, 2003

Ulrich Hilger
info@lightdev.com
http://www.lightdev.com

About SimplyHTML

SimplyHTML is an application for text processing. It stores documents as HTML files in combination with Cascading Style Sheets (CSS). The application combines text processing features as known from popular word processors with a simple and generic way to store textual information and styles.
SimplyHTML is not intended to be used as an editor for web pages in the first place (although it can be used for this purpose with some limitations too). Especially HTML and CSS produced by other applications might or might not work with SimplyHTML, as other applications as well have trouble with content not produced by themselves.
See also chapter 'What is SimplyHTML? ' in section 'Using SimplyHTML' for additional details.

The SimplyHTML Project

The SimplyHTML project was started to build an application with above features and to document, how this can be done. This is approached by building SimplyHTML in several stages and by documenting all stages thoroughly. Each stage is covering certain functionality making it easier to concentrate on some of the many details such an application is made of. Stages are described in this tutorial, which as well serves as online help. Source codes are documented with Javadoc in addition.

Open Source

All source code is openly available along with the application and can be used to find out, how the application is working. Among serving for above functions, SimplyHTML shall be an example for developers intending to build applications with similar functionality. As well it can serve as basis for other applications.
Please see chapter License for details about terms and conditions for availability and distribution of this product.

Documentation

Sources are documented by Javadoc comments, which are compiled to an API documents collection with Javadoc.
The project and application is documented with this tutorial. The tutorial covers general information about the project and the application, information about installation and requirements, the internal structure and functions of SimplyHTML and finally its usage .
The tutorial can be used as online help for SimplyHTML as well as for reading about how SimplyHTML is built. It is available in formats plain HTML, JavaHelp and PDF. The tutorial was built with application HelpExpert entirely.
Application HelpExpert is available at http://www.calcom.de/eng/product/hlpex.htm

Author

SimplyHTML, this documentation and all contents of the distribution package of SimplyHTML are created and maintained by
Gartenstrasse 15
65830 Kriftel
Germany
Ulrich Hilger
Internet http://www.lightdev.com
Fax +49 721 151 41 09 67
e-mail info@lightdev.com
I would like to hear your comments and suggestions so please don't hesitate to write.

All rights reserved. Please see chapter ' License' for details.

License

Except for parts shown separately below, application SimplyHTML and all of its components such as source codes, documentation and accompanying documents are distributed unter the terms and conditions of the GNU General Public License. To read the full license text, please see file '`gpl.txt`' in the distribution package of this software or refer to its text here.

Parts falling under different license terms

Classes `ExampleFileFilter` , `ElementTreePanel` and their source code is Copyright 2002 Sun Microsystems, Inc. All rights reserved.

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps:
(1) copyright the software, and
(2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and

passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program

subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or amongcountries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.>
Copyright (C) <year> <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU 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 General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) year name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program
`Gnomovision' (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

History of SimplyHTML

April 27, 2003

Stage 11 published with find and replace support.

February 15, 2003

Stage 10 published adding a HTML editor with syntax highlighting and some enhancements in the plug-in architecture.

December 29, 2002

Release 2 of stage 9 published. This release is a major update with enhancements to the way SimplyHTML handles HTML content. Release 2 focuses on a clearer distinction between HTML 3.2 and HTML 4. It lets the user choose the HTML version SimplyHTML should use through the options dialog. Changes were made to the style sheet handling and the plug-in architecture too.

December 13, 2002

Stage 9 published featuring creation and manipulation of links and link anchors. Refines working with paragraph styles and named styles by supporting tag types and multiple styles for given tags in certain controls. Refines cut and paste for nested paragraph tags. Compensates different display of font sizes between Java and Web Browsers. Table rendering was enhanced. Selected dialogs have a button for context sensitive help on the respective dialog. A new section user interface has been added to the tutorial for this purpose. The application 'remembers' the directories of the file that was last opened and saved.

November 8, 2002

Stage 8 published implementing paragraph and named style manipulation. SimplyHTML is enabled for Java Web Start as well and can be started with a single mouse click now from SimplyHTML's homepage at http://www.lightdev.com.

October 20, 2002

Stage 7 published implementing image insertion and manipulation. Fixed a serious bug in class AbstractPlugin.

October 4, 2002

Stage 6 published implementing list formatting.

September 19, 2002

Stage 5 published implementing a plug-in architecture and enhanced handling of resource bundles for dynamic menu creation. Along with the plug-in facility, user settings are implemented to be

persistently stored. An `AttributeMapper` compensates some discrepancies between Java and HTML.

August 15, 2002

Stage 4 published featuring table manipulation

July 11, 2002

Stage 3 published featuring font manipulation and dynamic tool bar creation.

June 29, 2002

Stage 2 published. Features resource bundles, multiple language support, cut and paste, drag and drop, both including HTML styles and dynamic menu creation.

June 21, 2002

First full release of stage 1 published. Inlcudes initial build of the tutorial, API docs, source codes, executable JAR file and a PDF version of the tutorial.

June 16, 2002

published pre-release 1 of stage 1 (source codes and javadoc compilation). Wow, eight weeks went by and still no first release finished. Let's at least share what we have so far. Because the tutorial takes more time to write than the application itself and although the tutorial is almoset done, this pre-release comes without the tutorial.

April 2002

start of the SimplyHTML project.

Parts of the distribution package

The distribution package comes as a single compressed zip file. It contains

- `SimplyHTML.jar` - the executable file for the latest stage of SimplyHTML
- `Help.pdf` - this tutorial as PDF file
- `readme.txt` - essential information about SimplyHTML
- `gpl.txt` - the file containing the license agreement valid for all parts of the distribution package
- `jhall.jar` - JavaHelp runtime extension
- `source` - source code directory
- `doc` - directory with API documentation files

Please refer to chapter 'License' for terms and conditions of using above parts.

Restoring contents of the downloaded ZIP file

Above contents can be restored from the compressed distribution file by using one of the many applications capable to extract ZIP files (WinZIP or Ark for instance). If you do not have such an application, you could use application Extractor available free at http://www.calcom.de/eng/product/xtract.htm

Getting compiled classes

Besides compiling the sources, file `SimplyHTML.jar` has a complete set of classes for the latest stage of SimplyHTML. Java Archive (JAR) files are structured like ZIP files. Their contents can be restored with any application able to extract ZIP archives (see above).

Getting this tutorial in JavaHelp format

Additional to the version in PDF format `SimplyHTML.jar` has a complete set of this tutorial in JavaHelp format in package `com.lightdev.app.shtm.help` too. Opening file `index.htm` in this package lets you open the tutorial with any Java enabled browser as well.

Getting started

Alright, you somehow found this tutorial, made it through lots of preliminary information about SimplyHTML and finally like to use it somehow. Thank you! And here is how it works:

1. find out, if you meet the requirements
2. make sure, download and installation is complete (yes, you have to do it without fancy installers, but hey, relax: it is simple, even I can do it...)
3. and off you go by starting the application

Seriously you should at least be familiar with this part of the documentation before you try to use SimplyHTML.

Requirements

Java

To run SimplyHTML, a Java 2 Standard Edition (J2SE) Runtime Environment (JRE) of version **1.4** or higher is necessary. To work with the source code, a Software Development Kit (SDK) for J2SE version **1.4** or higher is required.
J2SE JRE and SDK are available at no charge at
`http://java.sun.com/j2se/1.4`

JavaHelp

For using this tutorial as online help from out of the application, the JavaHelp runtime extension is needed. The JavaHelp runtime extension is distributed with the SimplyHTML package (file '`jhall.jar`'). To learn more about JavaHelp and for access to sources and API documentation of JavaHelp, please visit
`http://java.sun.com/products/javahelp`

Java Web Start

To install and run SimplyHTML directly from its home page the Java Web Start extension is needed. Java Web Start is installed together with the JRE (see above) automatically. To learn more about Java Web Start technlogy and for access to sources and API documentation, please visit
`http://java.sun.com/products/javawebstart`

Installation

Java Web Start

No installation is necessary with Java Web Start, just go on to 'Starting SimplyHTML' when using Java Web Start.

Traditional installation

Once downloaded the SimplyHTML distribution package zip file

1. create an arbitrary folder such as `C:\Programs\SimplyHTML`
2. extract all contents of the downloaded zip file into that folder

Starting SimplyHTML

Java Web Start

The simplest way to run SimplyHTML is to press button 'Web Start Me Now' at http://www.lightdev.com/dev/sh.htm . Alternately, direct your browser to http://www.lightdev.com/shtm.jnlp to achieve the same.
No file copying, no link or path settings, just one click.

Traditional start

To run the executable JAR file without Java Web Start, use the following command on the command line prompt of your system
(replace \ by / and omit `.exe` on Unix or Linux systems)

```
[JRE]\bin\java.exe -jar [AppDir]\SimplyHTML.jar
```

`[AppDir]` in above command means the directory, you have installed SimplyHTML on your computer. `[JRE]` means the directory, the Java 2 Sandard Edition (J2SE) Runtime Environment (JRE) is stored on your computer.
Note: All paths should not contain blanks. A path such as `C:\Program Files\SimplyHTML` as the <AppDir> will only work if it is put in quotes, such as in `"C:\Program Files\SimplyHTML\SimplyHTML.jar"`

Example 1 (Windows)

If your JRE is in directory
`c:\programs\java\j2re1.4.0_01`
and SimplyHTML is in directory
`c:\programs\SimplyHTML\`
the command to start SimplyHTML would be

```
c:\programs\java\j2re1.4.0_01\bin\java.exe -jar
c:\programs\SimplyHTML\SimplyHTML.jar
```

Example 2 (Linux)

On Linux your JRE might be on
`/usr/lib/java2/j2re1.4.0_01`
and SimplyHTML might be in
`/opt/simplyhtml`
then the command to run SimplyHTML would be

```
/usr/lib/java2/j2re1.4.0_01/bin/java -jar /opt/simplyhtml/SimplyHTML.jar
```

Inside SimplyHTML

This section explains the way application SimplyHTML is built, its structure, design and internal functionality. Refer to section 'Using SimplyHTML' to read about its usage.

Stages sections

SimplyHTML is built in stages. By dividing development into pieces, it is easier to concentrate on a certain detail of the application, paying more attention to the particular design of this part. The application will be more modular making maintenance easier to achieve.
Each stage results in a complete application. The resulting application gets more complex with every stage added while retaining its modular design. This section has chapters directly corresponding to each stage of SimplyHTML.

Source codes

A complete set of source codes is distributed together with this tutorial for each stage.
For stage 2 for instance there is a complete set of sources chapter 'Stage 2' refers to. This set of sources contains all sources of stage 1 and 2 making up one executable which - when compiled - represents stage 2 of the application.
In essence, sources of stage 2 do not contain changes versus stage 1 only, they represent a complete application stage inlcuding previous stages.

Spotlights section

The spotlights section discusses certain topics from a process oriented point of view. Where the stages sections explain the parts of SimplyHTML in the way they are structured (by classes and methods), spotlight topics wrap several parts of the application together to explain one process.

How to use this part of the documentation

This part of the tutorial should be used together with the source codes which have plenty of documentation in addition. Source codes are commented and most of the comments went into the API documentation accompanying the package. Additional (non Javadoc) comments make clear certain parts of code on top of that.
This tutorial does not repeat code. It is structured to lead the way into the very details of the application's source code by adressing certain topics in one block. Chapters usually refer to the source codes by naming certain fields or methods. It is recommended to open the source codes in parallel to reading this tutorial.

Target audience

Basics of Java and programming in general are not covered here, so interested readers should have a basic knowledge about these already. This section concentrates on discussing best practices and how to's in conjunction with a particular part of the application, trying to make a top

down approach in covering the key targets of SimplyHTML.

Planned development stages

The following is an overview of planned stages to be completed in the future. The plan is subject to change but it shall document, what the author intends to add or change in SimplyHTML for upcoming releases:

1. Link target window (clickable links)
2. Extension/consolidation of persistent application settings (Prefs)
3. Refine context sensitive help, popup menus, other GUI enhancements

Stage 1: Documents and files, menus and actions

This stage builds the basis for application SimplyHTML. It produces a basic executable capable to create, open and save simple HTML documents. Stage 1 concentrates on how to build a basic application with a main application frame and menus. As well it shows some concepts on how to work with documents and files.
Following is a short description of classes making up this stage and what they do in general.

AboutBox - A dialog showing information about application SimplyHTML.
App - The class containing the main method. This class constructs application SimplyHTML.
FrmMain - The application's main frame containing all menus and the view of all documents in a tabbed pane.
DocumentPane - GUI representation of a document of SimplyHTML.
ExampleFileFilter - a helper class from Sun Microsystems, Inc. for conveniently applying file filters to JFileChooser components
ElementTreePanel - a panel to show the element tree of a document
SHTMLEditorKit - the editor kit used for controlling documents in SimplyHTML
CSSWriter - a class for writing style sheets to CSS files
LicensePane - subclasses JPanel for displaying SimplyHTML's license
Util - a class with static utility methods

Creating a basic application

For creating a basic application, in Java we need a class having a method called `main` which accepts possible parameters from the standard Java Runtime Environemt (JRE). This basically is done by the class described in `App.java`.
Besides providing the application's main method, `App.class` constructs an instance of the main frame of application SimplyHTML, an instance of the class defined by `FrmMain.java`, and initially displays it.
When application SimplyHTML is started, the Java Runtime Environment opens the main thread for this aplication and executes method main of class `App`. Once all steps such as constructing the application's main frame, control is transferred to the event dispatching thread.

Setting a look and feel

An important feature of Java is to support different system platforms making it necessary to design applications independent from any system specific behaviour. The author of a Java application can not predict, on which system the application actually will be used.
Java provides mechanisms to keep applications away from proprietary look and feels or behaviours with class `UIManager` of package `javax.swing`. `App.class` of SimplyHTML uses methods `setLookAndFeel` and `getSystemLookAndFeelClassName` of class `UIManager` to initially set the look and feel to the one of the system, the application is started on.
By setting the L&F at runtime in the application's main method, an application can be kept independent from system specific behaviour. Class `UIManager` still allows to change L&F settings later in a session with the application if necessary.

Creating a main window and menus

As described in the previous chapter 'Creating a basic application', `App.class` creates and displays the main window of application SimplyHTML. For doing so, it uses another class defining the actual elements and functionality of the main window, class `FrmMain.java`.
Class `FrmMain` mainly has two areas of functionality in stage 1:

- it is the window in which all documents are opened and presented to the user
- it makes available all functions of the application through a menu bar

This section explains class `FrmMain` and its functions.

The class FrmMain

A window typically is represented by class `JFrame` of package `javax.swing.` To create the specific main window of application SimplyHTML, `FrmMain` extends `JFrame` by adding certain fields:

- `jtpDocs` - a `JTabbedPane` for displaying one or more documents
- `APP_NAME` - a string constant for representing the application's human readable name
- Several fields for menus and menu items

The fields are declared private to class `FrmMain` so that they can not be seen or manipulated outside class `FrmMain`.
In addition, class `FrmMain` adds several methods with functionality for

- constructing the window
- ensure window closings do not cause loss of data
- adjusting the windows appearance
- adding a menu bar and menu items

Above functions are described in more detail in the following chapters.

Constructing the main window

In the constructor of class `FrmMain` mainly three steps are done

- preparing the window to watch for events that would close this window
- adjusting the window to requirements special to application SimplyHTML
- adding a menu bar and menus

Once above steps are through as described below, application SimplyHTML's main window is shown. Once shown, the start process of the application has ended and control is transferred to the event dispatching thread which Java created for SimplyHTML automatically.
The event dispatching thread controls the program flow by invoking methods attached to certain events, which usually are fired by user actions.

Preparing for window close events

Closing an ordinary window might or might not need special handling. In the simplest case, respective window just closes which would be done automatically. For an application's main window however, closing usually terminates the application which in turn mostly requires certain cleanup before an application actually can be terminated.
A window can be closed through various actions such as the user selecting 'Exit' from menu 'File' or the user likes to switch off the computer etc. Most of such actions are signaled to a `JFrame`, if it claims to receive respective event notifications.
In class `FrmMain`, method `enableEvents` is called in the constructor of the class for this. Method `enableEvents` is inherited from class `Component` of package `java.awt` and is called, when a subclass of `Component` likes to receive or handle events of a certain type even without a respective event listener in place.
Class `FrmMain` calls for events defined in AWTEvent.WINDOW_EVENT_MASK and causes events of this type being delivered to method `processWindowEvent` of class `FrmMain`. This method handles window closing events for the main window of application SimplyHTML.
Especially see ' Avoiding loss of data in the close process' partly dealing with this topic too.

Handling window close events

Method `processWindowEvent` of class `FrmMain` watches window closing events for application SimplyHTML. All other events are not handled and routed on to `FrmMain's` superclass for possible handling.

Method processWindowEvent

For every `WINDOW_CLOSING` event, method `processWindowEvents` creates an instance of SHTMLFileExitAction and calls its `actionPerformed` method, which actually processes the closing.
For handling window closing events however, it is important that `processWindowEvent` checks whether or not documents are left open upon return from SHTMLFileExitAction. If, yes, this indicates one or more documents could not be closed resulting in refusing to close `FrmMain` thus keeping application SimplyHTML from terminating.
Especially see ' Avoiding loss of data in the close process' partly dealing with this topic too.

Customizing the main window

As mentioned in chapter 'The class FrmMain', this class is extending class `JPanel` by adding some extra fields and methods. To construct these extras as well as to set up the way the window initially is displayed, method `customizeFrame` is used.
After setting the window title and size, method `customizeFrame` gets a reference to the windows content pane, and adds a newly created instance of `BorderLayout` to it. Finally it creates a new `JTabbedPane` and adds it to the center of the content pane. As the `JTabbedPane` is the only

component of the application's main window, it covers all of its visible region except for the menu and title bar.

JTabbedPane jtpDocs

A reference to `FrmMain's` tabbed pane is kept in field `jtpDocs` as a central place for pointing to all documents open in the main window. Whenever a document is created or opened, it is added to `jtpDocs` for display.
Thus, `jtpDocs` is a good place to look for a certain document. If the currently active document shall be adressed for instance, `jtpDocs.getSelectedIndex` points to the currently selected tab in the `JTabbedPane` and `jtpDocs.getComponentAt` fetches the component contained in this tab, which in turn would be the currently active document.

Adding a menu bar and menu items

As class `FrmMain` defines the main window of application SimplyHTML it should present the application's functions to the user. One way of doing this is to maintain a menu bar and menu items. In class `FrmMain` this is done mainly in method `constructMenu` which is called in `FrmMain's` constructor.

Fields referring to menus and menu items

Class `FrmMain` holds a field for each menu and menu item in its menu bar. These fields are visible to all methods within class `FrmMain`. By keeping fields referring to the menu structure it is possible to influence appearance and behaviour of the menu structure later on during the flow of the program.
To build a menu bar using the menu fields of class `FrmMain`, the fields have to be initialized first. This is done at the same time the fields are declared upon construction of a `FrmMain` object. Initialization is done by creating a new instance of `JMenu` and `JMenuItem` respectively on the line of code where that field is declared. Each menu item gets associated with an instance of the action class which is meant for handling selctions of the particular menu item.

Method constructMenu

The previously initialized menu fields of class `FrmMain` are put together on a menu bar by method `constructMenu`. The method first creates a new `JMenuBar` object and then adds all previously initialized menus and menu items to it.
It then adds a new `MenuListener` to the file menu object which takes care of adjusting appearance of the file menu whenever it is selected. Finally the new menu bar is associated with the main window by calling method setJMenuBar, which was inherited from `JFrame`.

Creating and storing documents

From the perspective of application SimplyHTML there is a distinction between documents as represented in package `javax.swing` and documents of application SimplyHTML. Package `javax.swing` provides a very powerful set of classes for working with documents, which relieves applications from having to completely implement their own editor, document model, etc.
How this set of classes is implemented in an application is left to the application developer however, allowing a maximum of flexibility while reducing effort and still retaining a common basis for the particular functionality.

Documents in Java

The package `javax.swing` implements a model-view-controller (MVC) approach to work with documents as shown below:

Document - the model for swing text components
EditorKit - the controller for text components
JTextComponent - the view component

The interface `Document` is a container for text that serves as the model for swing text components. The goal for this interface is to scale from very simple needs (a plain text textfield) to complex needs (an HTML or XML document, for example).
The abstract class `EditorKit` serves as the controller for text components establishing the set of things needed by a text component to be a reasonably functioning editor for some type of text content.
Abstract class `JTextComponent` finally is the view component in the MVC context serving as the base class for swing text components such as `JEditorPane`.

Documents in SimplyHTML

SimplyHTML defines an own class for dealing with the documents used in the application. Class DocumentPane is used to create new documents, load documents from file, save documents, view and edit documents and to define a SimplyHTML document in general.
With DocumentPane there is only one interface to deal with for both GUI and functionality when working with documents.

Style sheets and HTML documents

Before we start looking into how SimplyHTML implements the use of Documents, we should spend some time on reviewing how HTML documents and Cascading Style Sheets (CSS) are related.

What are CSS styles

Styles in CSS syntax are an extension to 'plain' HTML that allow to define attributes describing how HTML content should look. With styles one can define that paragraphs should always have a 6pt margin at the top for instance or that headlines always should be shown in red letters of 18pt size using font Helvetica.

How styles can be used

HTML allows to combine content and styles all in one HTML file, for example by means of adding `style` attributes to HTML tags. Another way is to store a `<style>` tag in the `<head>` tag of the document.
In addition it is possible to store styles in files separated from an HTML file. Storing styles in Cascading Style Sheets (CSS) has the advantage that a set of common styles can be defined at a central location serving as the basis for an arbitrary number of HTML files.

How SimplyHTML uses styles

In SimplyHTML a combination of styles defined separately in style sheets and styles defined as attributes within HTML tags is implemented. Whenever possible SimplyHTML avoids HTML tags such as `<font>` in favour of styles.
See 'Style handling design in SimplyHTML' for additional details.

Style handling design in SimplyHTML

Java implements class `StyleSheet` to define central styles in CSS notation for an `HTMLDocument`. In addition styles in CSS syntax can be included directly in `HTMLDocuments` by storing them as an attribute to a HTML tag.
In all Java versions up to J2SE 1.4 there is no way however, to store styles persistently although working with `HTMLDocuments` in Java would hardly work without styles. Applications creating `HTMLDocuments` have to find a way to save styles on their own. SimplyHTML defines and uses an own class CSSWriter for this matter.

When are styles saved?

In stage 1 of application SimplyHTML styles can not be changed, so a style sheet is only saved, when a newly created document is saved (i.e. the document was not loaded from a file and a style sheet with the same name does not exist at the target location for the document).
This leaves one conflict open:

- a style sheet exists at the location where a new document is to be saved and
- the exisiting style sheet has the same name as the style sheet associated with the new document to be saved and
- the exisiting style sheet is different from the style sheet associated with the new document

In this case, the style sheet is not saved and the existing style sheet is used. A solution for this case is not implemented in SimplyHTML yet. A workaround for the time being is to save newly created documents in a directory which holds only documents sharing the default style sheet of SimplyHTML.
In later stages of SimplyHTML it will be possible to change styles. Then the association of style sheets and documents as well as the handling of saving style sheets will be refined.

The class DocumentPane

`DocumentPane` implements an application centric approach of a document, wrapping the classes of javax.swing together with application and GUI functionality into one class.
`DocumentPane` follows the idea of having a single interface between an application and its documents. It serves as the single point for displaying, editing, opening, saving etc. For this, it combines GUI elements and 'MVC' elements to form one consistent and compact class to work with in applications.

Elements of DocumentPane

The major element of `DocumentPane` is a `JEditorPane`. `DocumentPane` extends `JPanel` and combines a `JScrollPane` and the `JEditorPane` as the only visible components.
In addition it has fields for storing the name of the document, the source where it has been loaded from and a field indicating whether or not the document has changed. Finally constants for a default document name as well as two cursor definitions are contained as fields in the `DocumentPane`.
There are two fields reflecting save operations as well: `saveInProgress` is set to true while a save operation runs and `saveSuccessful` is set by a save operation before (false) and after a save (true, if and only if all went fine) to indicate any errors while saving.
Last but not least, `DocumentPane` implements interface `DocumentListener` to recognize changes.

Interface DocumentListener

In field `textChanged` class `DocumentPane` 'remembers' if there have been changes to it's document since its creation or since the last save to file operation. But the `DocumentPane` needs to be notified about changes for being able to 'remember' them. It implements interface `DocumentListener` for that purpose.
Interface `DocumentListener` defines what a class would have to do to listen to `DocumentEvents` . `DocumentEvents` are fired to registered listeners whenever a document is changed in any way for instance.
Class `DocumentPane` defines methods `insertUpdate`, `changedUpdate` and `removeUpdate` to implement the `DocumentListener` interface. By doing that, class `DocumentPane` is a `DocumentListener` and can register itself with any document to listen for changes.

DocumentPane sets field textChanged to true whenever it is notified of a DocumentEvent.

Constructing a DocumentPane

Simply creating a new instance of an editor pane and throwing it onto a JPanel is not enough to display and use a JEditorPane. Due to the relationship of Document, EditorKit, JEditorPane and StyleSheet it is necessary to initialize all elements properly.
In DocumentPane's basic constructor DocumentPane() a new JEditorPane is created and assigned to field editor. The caret color for the editor pane is set and the typical text cursor is assigned by calling method setEditCursor (see below).
Then a new SHTMLEditorKit is created and assigned to our editor pane. This procedure ensures that a document gets correctly initialized with the style sheet properly attached.
Finally a new JScrollPane is created, the editor pane is added to the scroll pane and both are added to the DocumentPane after defining an appropriate layout.
The editor pane has to reside inside a JScrollPane for a vertical scroll bar automatically being shown as needed. The editor pane automatically wraps words at the right end of the pane so that a horizontal scroll bar is not shown or needed.

Method setEditCursor

Although JEditorPane has a method setCursor inherited from java.awt.Component, setting the cursor with that method does not cause respective cursor to be displayed (probably someone can let me know why sometime). Method setEditCursor therefore adds a MenuListener to our editor, that reacts on mouseEntered and mouseExited events.
When the mouse enters the editor pane, the cursor is set to the text cursor, when the mouse exits the editor pane the cursor is reset to a default cursor.
Method setEditCursor achieves this by getting the glassPane of the DocumentPane's JRootPane, assigning respective cursor and setting the glassPane to visible.

Constructing a DocumentPane with above steps sets up the basic contents of a DocumentPane. Read on to learn how to create a new document for editing or to open an existing one.

The class SHTMLEditorKit

Class HTMLEditorKit in package javax.swing.text.html automatically associates a style sheet with any newly created HTMLDocument. The style sheet used is taken from the Java Runtime Environment and holds default styles for all possible HTML tags.
To use a different set of styles, we can either load another style sheet afterwards and delegate it to the HTMLDocument or we have to override this behaviour at its original location.
Class SHTMLEditorKit overrides all methods in HTMLEditorKit dealing with the default style sheet and uses our own set of styles.

Method getStyleSheet

This method returns the style sheet found in field `defaultStyles` of `SHTMLEditorKit`. If this field points to a `StyleSheet` object, this `StyleSheet` is returned.
If `defaultStyles` is not initialized so far, a new `StyleSheet` object is created. Then the default style sheet, identified by constant `DEFAULT_CSS` , is located by calling method `getResourceAsStream` inherited from class `Class`. `getResourceAsStream` looks for the style sheet file in the class path and returns an `InputStream` for it if found.
A CSS file '`default.css`' is distributed with the classes of application SimplyHTML so that it can be loaded as default in this way. A new `BufferedReader` is created which reads from a new `InputStreamReader` used on the `InputStream`. Method `loadRules` of class `StyleSheet` then reads all styles from the `BufferedReader`.

Creating new documents

To create a new document basically two steps are necessary

1. create a new `DocumentPane` for viewing and editing a new document
2. create a new `HTMLDocument`, initialize it properly and attach it to the editor pane of the `DocumentPane`

With method `DocumentPane(URL url)` class `DocumentPane` has a constructor especially for that purpose. When a `DocumentPane` is created by calling this constructor with `null` as the url parameter, the `DocumentPane` is told to create a new `HTMLDocument` after creating the basic DocumentPane. The constructor calls `this()` to create a basic `DocumentPane` and then it calls method `createNewDocument` for creating the new document.

Method createNewDocument

As with creating a new editor pane, it is not enough to simply create a new `HTMLDocument` object and send it to the editor pane. We need additional steps inorder to adjust the `HTMLDocument` to the needs of application SimplyHTML which is why it makes sense to put these steps into an own method.
`createNewDocument` first gets the SHTMLEditorKit which was attached to the editor pane upon construction of the `DocumentPane`. A new `HTMLDocument` is created by calling method `createDefaultDocument` of the editor kit. `createDefaultDocument` ensures that we get a new `HTMLDocument` properly initialized and with our default style sheet attached to it.
The `HTMLDocument` then gets inserted a link reference to the style sheet file. Because we create a new `HTMLDocument` , this link refers to the name of SimplyHTML's default style sheet. The link reference is necessary because once we save the `HTMLDocument`, it can be used with other applications too. For an application other than SimplyHTML, the only way to determine which style sheet to use when the `HTMLDocument` is loaded is this particular style sheet reference (see 'Style sheets and HTML documents'). To insert the style sheet reference to the document, method

`insertStyleRef` is used (see below).
The `DocumentPane` is registered as `DocumentListener` with the new document, causing the document to notify its `DocumentPane` about all changes. Finally the new `HTMLDocument` is assigned to the editor pane.

Method insertStyleRef

Method `insertStyleRef` inserts a reference link to the style sheet file to be associated with the HTML document having this reference link. The reference link has a syntax such as `<link rel=stylesheet type="text/css" href="style.css">` and is to be placed in the `<head>` tag of a HTML document (see 'Style sheets and HTML documents').
To insert the reference, method `insertStyleRef` 'walks' through the element structure of a `HTMLDocument` and looks for its `<head>` and `<body>` tags. If a `<head>` tag is found, it is assumed that it already has the appropriate reference and the method does nothing.
Otherwise, it uses method `insertBeforeStart` of class `HTMLDocument` to insert a new `<head>` tag before the start of the `<body>` tag. The `<head>` tag is inserted along with the reference link to the style sheet by passing a respective HTML string to method `insertBeforeStart`.

Saving documents

To persistently store a newly created document or to save changes to an existing one, application SimplyHTML uses an object of class `HTMLWriter` which package `javax.swing` holds especially for `HTMLDocuments`. Again, to simply call `HTMLWriter's write` method is not enough to meet saving requirements of SimplyHTML. Class `DocumentPane` uses methods `saveDocument` and `saveDefaultStyleSheet` to put together its own save routine.

Method saveDocument

In method `saveDocument` class `DocumentPane` uses its field `sourceUrl` to tell `HTMLWriter` where to save a document. Field `sourceUrl` usually has been set previously either by method loadDocument or `saveDocumentAs`. If no `soureUrl` is set for any reason, `saveDocument` does nothing.
When field `sourceUrl` is properly set, `saveDocument` creates a `FileOutputStream` for respective URL, attaches it to a new `OutputStreamWriter` and creates an instance of `HTMLWriter` with these parameters. `HTMLWriter` completely takes care of transforming the model of object `HTMLDocument` to an HTML file when its method `write` is called.
Once the HTML file is saved, its associated style sheet is saved by calling method saveStyleSheet . Finally field `textChanged` is set to `false` to indicate that the document of this `DocumentPane` does not contain changes which need to be saved.

Saving CSS style information

A style sheet is saved by application SimplyHTML with method `saveStyleSheet` in class `DocumentPane`. This method is called by method `saveDocument` whenever a document is saved. To read more about HTML and style sheets in general and about how SimplyHTML uses style sheets, see chapters 'Style sheets and HTML documents' and 'Style handling design in SimplyHTML'.

Method saveStyleSheet

Method `saveStyleSheet` first determines the file name of the style sheet associated with the document to be saved by calling mehtod `getStyleSheetName` (see below). `getStyleSheetName` returns the file name as a URL string. A `URL` object is created with that string and method `getFile` is called on that `URL` object to transform the URL string to a file string. This file string is taken to create a new `File` object for the style sheet to be saved.
With the `File` object it is first ensured that the file does not already exist with the help of method `File.exists`. If it exists, the style sheet is not saved (see 'Style handling design in SimlpyHTML').
If the file does not exist, it is created by calling method `createNewFile` on the `File` object. An `OutputStream` object is opened on the newly created file and an `OutputStreamWriter` is created for that `OutputStream`.
If the document to be saved has been newly created, the `StyleSheet` object of SHTMLEditorKit is taken to be written to file. If the document was loaded from file, the style sheet of the document is taken instead (the second case will not happen in stage 1 of SimplyHTML).
A new CSSWriter object is created with that style sheet and the previously created `Writer.` The style sheet is written to file by calling CSSWriter's `write` method. Once done, `OutputStreamWriter` and `OutputStream` are closed.

Method getStyleSheetName

A style sheet is saved at the location pointed to by the style sheet link reference of its associated document. The style sheet link reference usually is a relative expression containing the file name of the style sheet and an optional path which usually is a relative path.
Method `getStyleName` returns the path and name of the style sheet by combining the document base (the path the document actually is stored at) and the (possibly relative) path and name of the style sheet reference.
First path and name of the style sheet as contained in the document's style sheet reference link is taken by calling method `getStyleRef` (see below). Then the document base is read with method `getBase` of class `HTMLDocument`.
If a style sheet reference link is not found in the document, the default style sheet name is taken from class `SHTMLEditorKit`. Finally a relative path possibly contained in the style sheet

reference is resolved by method `resolveRelativePath` and the resulting name is returned.

Method getStyleRef

In method `getStyleRef` , the first occurrence of a `<link>` tag is searched in the element tree of the document with the help of method `findElement`. If a `<link>` tag is found, the `value` object of its `href` attribute is copied to the local string variable `linkName`.
If no `<link>` tag is found or it does not contain a `href` attribute, `null` is returned.
There is no implementation for the case that there is more than one link reference to style sheets.

The class CSSWriter

As mentioned previously, Java does not provide a class to save information in a `StyleSheet` to a file. To be able to save CSS information, SimplyHTML defines and uses class `CSSWriter`. `CSSWriter` is passed a `Writer` object and a `StyleSheet` object upon construction. The `StyleSheet` object contains the CSS information to be saved and the `Writer` object identifies the destination to store the CSS information at.
In the constructor of `CSSWriter`, the two parameters `Writer` and `StyleSheet` are stored in respective fields `w` and `s` of class `CSSWriter` for later use. To actually write out the styles found in the `StyleSheet s`, method `write` is used.

Method write

In method `write` an `Enumeration` of all rules in the `StyleSheet` is created. An `Enumeration` is a good way to iterate over a collection of elements. It provides two convenience methods `hasMoreElements` and `nextElement for` this purpose. While there are more Elements in the Enumeration of styles, method `write` gets the next element in the `Enumeration` being the name of the next `Style` object.
It then gets the `Style` object identified by that name. `Style` objects are `AttributeSet` objects that can contain attribute objects or other `AttributeSet` objects. The length of the style name is taken to find out how far from the left side the style attributes have to be written to file. This indentation length is stored in field `indentLen` of class `CSSWriter`.
First the style name is written to file followed by the character opening a collection of attributes for a CSS style ('{'). Every style except the one identified by `StyleContext.DEFAULT_STYLE` then is written to file by calling method `writeStyle` on respective `Style` object.

Method writeStyle

In method `writeStyle` again an `Enumeration` is used to go through all attributes found in the style `AttributeSet`. An `AttributeSet` is a pair of objects for the key and the value of a style attribute. For every element in the `Enumeration` the key and value objects are read.
Attribute `StyleConstants.NameAttribute` can be left out from writing because it contains the name of the style, this `Style` object is describing. As well the attribute

`StyleConstants.ResolveAttribute` is not written to file itself, because it contains another `AttributeSet` in its value object. For attributes of type `StyleConstants.ResolveAttribute` method `writeStyle` is called recursively to write out all attributes found in this attribute.
All other attributes are written to file by first writing the key object having the name of the attribute (such as font-size, left-margin, etc.) followed by a colon, the attribute value (such as '12pt' or '20%', etc.) and a semicolon. Every attribute except the first is preceded by a new line and the indentation needed to make the collection of attributes more legible.
Finally the closing character for a CSS style ('}') and a new line is written.

Using the appropriate line separator

At design time it can not be predicted on which operating system application SimplyHTML might be used. As different operating systems use different line separators with their file system, this has to be taken into account for our save routine. `CSSWriter` gets the correct line separator from a global constant defined in Java with command `System.getProperty("line.separator")`. The value returned by this call is appended to every line written to the target style sheet file.

Loading documents from file

When a document is loaded into a `DocumentPane`, the `DocumentPane` and its components need to be initialized properly. This is done in method `loadDocument`.

Method loadDocument

In method `loadDocument` , first the editor kit object is taken from the editor pane of this `DocumentPane` object. With method `createDefaultDocument` of the editor kit a properly initialized empty document object is created with the appropriate style sheet attached. The document base is set from the URL the document is to be loaded from. The document base is necessary so that all relative URLs probably contained in the document are correctly resolved.
The contents of the HTML file are then loaded into the new document object by opening an `InputStream` from the URL and using method `read` of the editor kit. The DocumentPane registers itself as `DocumentListener` with the new document, causing the document to notify its `DocumentPane` about all changes.
Finally the new `HTMLDocument` is assigned to the editor pane and the URL, the document was loaded from is stored in field `sourceUrl` of the `DocumentPane`.

How the style sheet is associated

Application SimplyHTML assumes that every HTML document is associated with a style sheet in a separate CSS file. This style sheet must be referenced by a link in the `<head>` tag of the HTML document as it is generated by method insertStyleRef.
If such a reference link is contained in the `<head>` tag of an `HTMLDocument` and the style sheet file can be found at the referenced location, the `read` method of the editor kit handles the style

sheet reference correctly and the editor pane renders the HTML document with the associated styles.

Connecting GUI and functionality

Functionality of an application - more or less the application kernel - usually is separated from the graphical user interface (GUI) in separate classes. In some cases it makes sense, to combine both in one class or to create inner classes within another class too.
Still there must be a connection between the two, application kernel and GUI, which can be built by using actions.
Actions are explained in general below. In the follwoing chapters some of the actions of FrmMain are explained in greater detail. Additional comments about the actions can be found in the source code.

Actions

Action classes are a design approach to make a central connection between GUI and functionality. With functions being coded as actions, respective functionality is kept in a central place and can be centrally combined with code dealing with availability and behaviour of the respective functions during certain states of the application.

Why actions make sense

Coding certain functionality as actions makes code easier to maintain, because code is stored at exactly one location while the resulting action and its appearance can be connected to several GUI elements such as a menu item and a tool bar button for instance.

What actions do

An action can be constructed with a certain name and icon. Components such as menu items or buttons automatically display an action's name and icon when it is associated with them. As well the action's state (enabled or not enabled) automatically is reflected in the component's display (dimmed or normally shown).
When selecting a component that has an action associated to it, this component automatically fires an action event that calls method actionPerformed of that action. An `ActionEvent` object is created automatically describing the event that led to calling the action.
In essence a big advantage of using actions aside of their easier maintenance is that no additional coding is required to support the mentioned interactions and dependencies.

How actions are designed in class FrmMain

In class `FrmMain`, the actions are designed as inner classes. With that the actions have access to `FrmMain's` fields and methods without `FrmMain` having to pass references to the actions explicitly. The actions defined so far have a lot to do with the documents shown in `FrmMain` so that access to jtpDocs is quite helpful.
In the following chapters we look into some of the actions of FrmMain in detail.

Actions of FrmMain

The actions in FrmMain take use much of class DocumentPane. They call some of the functions for files and documents available in this class. Also other functions are wrapped into actions.
In this section it is explained, how to make existing functions available to the user. The functions of stage 1 themselves are explained in 'Creating and storing documents' and in ' Documenting the application'.

Actions for the file menu

In the file menu, the basic actions on documents and files are selectable. Create new documents, open documents from file, save changes to documents, save new documents or save exisiting documents under a new name and finally exiting the application can be done by the user through this menu.

Actions for the help menu

The help menu makes all kinds of documentation available to the user. With stage 1 of application SimplyHTML the help menu has links to the help file (this tutorial), the API documentation and an 'About this application' dialog.

Action list and short description

SHTMLFileNewAction - create a new document and show it
SHTMLFileOpenAction - open an existing document and show it
SHTMLFileCloseAction - close a currently open document and take care of saving the document before closing if necessary
SHTMLFileCloseAllAction - close all currently open documents and take care of saving the documents before closing if necessary
SHTMLFileExitAction - exit the application and take care of saving open documents before closing if necessary
SHTMLFileSaveAction - save a document
SHTMLFileSaveAsAction- save a document under a new name
SHTMLHelpAppInfoAction - shows SimplyHTML's about dialog
SHTMLHelpShowContentsAction - brings up online help

Dynamic interaction

There is a certain dependability between some of the actions which requires to aviod redundancy of code in the design therefore. Using actions bears an advantage for this because it allows to build a more complex process by combining actions individually.
Especially see 'Avoiding loss of data in the close process' in the Spotlights section partly dealing with this issue too.

SHTMLFileCloseAction

All actions implement method `actionPerformed` which gets called automatically by components bound to that action. Usually all functionality to be performed by an action goes here. `SHTMLFileCloseAction` adds some flexibility to that approach by having an additional method `closeDocument`. This method is called by the action's `actionPerformed` method on the currently active document so that whenever the action is fired, the currently active document is safely closed.

Method closeDocument

With method `closeDocument` an arbitrary document shown in the main window can be closed even if it is not the currently active one. The method is declared public so that it can be called and reused from other places simply by instantiating class `SHTMLFileCloseAction`.
The method's main task is to ensure that a document is only closed, when all of its contents are properly saved. It does this by

1. check, whether or not the particular document needs to be saved (if it was newly created and/or contains unsaved changes, that is)
2. if not, the document is closed
3. if yes, the user is asked if the document shall be saved
4. if the document shall be saved, it is saved by calling the appropriate action class (perform a 'save' or 'save as')
5. if the save was successful or if the user chose not to save changes, the document is closed
6. if the save was not successful or the user wanted to cancel, the document is not closed

In all cases, closing the document is done simply by removing the respective DocumentPane from FrmMain's jtpDocs.

Special case: save thread in progress

As outlined in ' Using threads for lengthy operations', saving a document among other functions is set forth in an own thread. Any close operation has to consider, that a save operation on a particular document could be in progress at the time the user selects to close a document. Method `closeDocument` takes this into account by calling Method `scheduleClose` (see below) in cases where it detects a save operation being in progress or where it caused a save operation itself while attempting to close a document.

Method scheduleClose

When a save operation is in progress on a document that is to be closed, SimplyHTML has to wait for the save operation to finish because a document may only be closed when it was saved successful. Whether or not the save operation was successful can only be determined, once it completely finishes, so in this case, the application has to wait with closing until then.

To block the application from other activities, method `scheduleClose` creates a `Timer` thread and schedules a `TimerTask` with the `Timer` . The `TimerTask` periodically checks whether or not the save operation of the particular document has finished with the help of field `saveInProgress` of class `DocumentPane`. If it has finished, the document is closed and the `Timer` and `TimerTask` are disposed. If there was an error during the save operation, the document remains open.

Design advantage

The advantage of this design is that closing a document safely is implemented only once. Still it can be reused either as action or as method from anywhere in the application as done in SHTMLFileCloseAllAction or `SHTMLFileExitAction` for instance.

Especially see 'Avoiding loss of data in the close process' partly dealing with this topic too.

SHTMLFileCloseAllAction

`SHTMLFileCloseAllAction` uses `SHTMLFileCloseAction` to close all currently open documents. It declares a field for an own private instance of `SHTMLFileCloseAction`. This field gets initialized it with an `SHTMLFileCloseAction` object upon construction of the `SHTMLFileCloseAllAction` object.
In it's `actionPerformed` method `SHTMLFileCloseAllAction` then simply loops through all open documents and calls method closeDocument of `SHTMLFileCloseAction`.

SHTMLFileExitAction

This action will terminate the application and takes care of saving possibly existing unsaved changes before. The application will only be terminated, if all possibly open documents could successfully and safely be closed.
To ensure a safe termination of SimplyHTML, `SHTMLFileExitAction` fires an SHTMLFileCloseAll action to safely close all possibly open documents. It then checks whether or not documents are left open. If yes, one or more documents could not be closed safely and the application will not be terminated.
To properly handle window close events, this action is used in method processWindowEvent of FrmMain too.

Using threads for lengthy operations

Most of the operations we encountered so far are not considerably time consuming. Especially loading or saving documents however, can be a lengthy task depending on the amount of data to be processed. Without any special handling of these tasks, application SimplyHTML could block for the time a particular save or load process would take. Java provides a mechanism to overcome

this potential problem with class `Thread`.

Threads

Usually all activities of an application are done within the event dispatching thread. All lines of code contained in a method called by the event dispatching thread are executed sequentially in the order they are coded. In Java however, this must not be the case always. By opening a new `Thread` object and starting the code placed in its `run` method, this piece of code is executed in parallel or at least asynchonous from the event dispatching thread.

How SimplyHTML uses threads

In SimplyHTML three operations are executed in separate threads so far: saving a document to a file, loading a document from a file and closing one or more documents. All operations are embedded in an inner class of the respectice Actions `SHTMLFileSaveAction`, `SHTMLFileOpenAction`, `SHTMLFileSaveAsAction` and SHTMLFileCloseAction.
The inner classes `FileSaver`, `NewFileSaver`, `FileLoader` are subclasses of class `Thread` and simply wrap the call to saveDocument or loadDocument respectively in the `run` method inherited from the `Thread` class. Once an action is fired, its `actionPerformed` method creates an instance of `FileSaver`, `NewFileSaver` or `FileLoader` and calls its `start` method.
In addition, method scheduleClose in SHTMLFileCloseAction creates a `Timer` thread for each close operation waiting for a save operation to complete.

Documenting the application

Designing and developing an application can be hard work already. But the resulting solution is nothing without proper documentation.
Documentation starts at providing information about the name of the application or license and copyright notices and goes all the way through installation guidelines to a user manual, tutorial and information for developers.
Most of the time, documenting an application is at least as much the work as developing it. This section shall give some hints about how a proper documentation can be accomplished.

Creating an 'About dialog'

An elementary part of an application is its 'about' dialog. An 'about dialog' usually is shown when the user likes to know which application is currently running, by whom it was created, which version the user operates, which terms and conditions are connected to usage, etc.
It is common practice to have an item in the help menu of an application such as 'About SimplyHTML' for this purpose. This menu item in application SimplyHTML creates an instance of its `AboutBox` class.

Class AboutBox

Class `AboutBox` descends from class `JDialog` and hosts a number of information panels. In the upper left part, an image associated with the application is shown (see below). Next to the image on the right a number of `JLabels` have the application name, current release, the author (wow, that's me!) and the application's home page. The name of the application is taken from the constant `APP_NAME` of class `FrmMain`.
Below these components, a license panel shows the full text of the GNU General Public License SimplyHTML is distributed under. Finally a close button is displayed at the bottom of the dialog. `AboutBox` does not have any other function than to construct itself and to be shown for information. Once the close button is pressed, it is disposed properly. This is ensured by overriding method `processWindowEvent` and calling `enableEvents` in the constructor of `AboutBox`.

Reading an image from the class path

A common way to display an image is to place a JLabel somewhere onto a visible component and associate an image to it. `JLabel` has a constructor especially for that, which accepts an `ImageIcon` object as a parameter. The image is taken from the class path of application SimplyHTML, where file '`App.jpg`' is distributed together with the applciation's classes in subdirectory '`image` '.
Method `getResource` in the `Class` object of respective `JLabel` is used to create an `ImageIcon` object for file '`App.jpg`'. Using the command

`this.getClass.getResource("image/App.jpg")` gets the class object of this `JLabel` object and finds out from where it was loaded. Calling `getResource` on this `Class` object resolves the name given as a parameter to `getResource` as a relative path to the path where the class was found.

The class LicensePane

`LicensePane` extens class `JPanel` by adding a `JTextArea` and a `JScrollPane` to it. The `JTextArea` is used to display the full text of the GNU General Public License. By wrapping this functionality into a separate class it is easier for other classes such as AboutBox to show that info text.

Constructing a LicensePane

The constructor takes a `Dimension` object to determine the preferred size of the panel to be constructed. In the constructor a new non-editable `JTextArea` is created with the license text to be displayed as a parameter. The license text is delivered by method `getLicenseText` (see below). A new `JScrollPane` is created and the `JTextArea` is associated with it. The vertical and horizontal scroll bar are set to be displayed as needed.
Finally the `JScrollPane` containing the `JTextArea` is added to the `LicensePane` and the license text is scrolled to the top with method `setCaretPosition`.

Method getLicenseText

The license text is taken from file '`gpl.txt`' delivered in the class path of the distribution of SimplyHTML. Method `getResourceAsStream` is used to locate the file and to open an `InputStream` object to it. An `InputStreamReader` is created on that `InputStream` and the `InputStreamReader` is used to create a `BufferedReader`.
As long as lines are found in '`gpl.txt`' they are read and appended to a `StringBuffer`. Finally the readers are closed properly and the contents of the `StringBuffer` are returned as a `String` .

Adding online help

Once an application is started by the user it is most convenient to provide information and help directly out of that application. With JavaHelp technology the Java language has an extension for online help which any application can use to seamlessly incorporate documentation.

JavaHelp extension

JavaHelp extends the Java Runtime Environment in the way that it provides a specification and platform to display any kind of documentation. All Information presented in JavaHelp is stored in HTML files. Table of contents, index and map are XML files all wrapped together with the HTML topic files into a framework of classes to view and query the information through a common user

interface.
Unfortunately the JavaHelp extension is not a part of the core Java Runtime Environment (JRE) so the extension has to be applied manually by coping the extension file for JavaHelp into the extensions directory of the JRE.

How to use JavaHelp in an application

Once a JavaHelp documentation is set up by creating HTML documents, table of contents, index and map files and the JavaHelp extension is available to the JRE, displaying JavaHelp documentation simply is done by creating a `HelpSet` object for the corresponding help set and display it with an instance of `HelpBroker`.
More about the JavaHelp specification and technology is available at
http://java.sun.com/products/javahelp

How help is created for SimplyHTML

All documentation is stored in a single JavaHelp help set in directory `help` of the SimplyHTML class path (`source/com/lightdev/app/shtm/help`). The help set is the one you are currently reading. It was produced entirely with the Java application HelpExpert. HelpExpert is created by the author of SimplyHTML (hey, that's me again!) and is available at
http://www.calcom.de/eng/product/hlpex.htm

How help is implemented in SimplyHTML

A common practice is to deliver documentation through menu 'Help'. Consequently SimplyHTML has an item 'Help Topics' in the 'Help' menu linking to this help set. In stage 1 the link was performed by `SHTMLHelpShowContentsAction`. This has been refined to now being handled in method `initJavaHelp` of class `FrmMain`.

Method initJavaHelp

In method `initJavaHelp` an instance of a `HelpBroker` is created pointing to the JavaHelp version of this tutorial (included in the Java archive (JAR) file of SimplyHTML). A reference to this `HelpBroker` is kept for later use. Then class `FrmMain's` instance of class `DynamicResource` is used to get the menu item meant for displaying the application's online help topics overview. Class `CSH` (for context sensitive help) of the JavaHelp API is used to create an `ActionListener` responsible to display context sensitive help on occurrence of a given action (selecting the 'Help contents' menu item in this case). This `ActionListener` is registered with the menu item.
Important: A 'readme' document should always be distributed with an application as *plain text file* . The readme file should contain essential information to properly set up and run the application. If all other tools fail, at least the user can be referred to this file to start with.

Creating source code documentation

Source codes usually are documented by comments inserted at positions in the code where the author thought an explanation of what is done there might be needed or otherwise helpful.
If such comments are created following a certain syntax, they can be translated to a set of API documentation files in HTML with the Javadoc tool of the Java Software Development Kit. For details about Javadoc please refer to page 'Javadoc 1.4 Tool' at
http://java.sun.com/j2se/1.4/docs/tooldocs/javadoc/index.html
Generally it is a good idea and common practice to comment Java source codes following Javadoc syntax as almost automatically there will be a very transparent and thorough documentation of the sources openly viewable with nothing more than a browser.
With application SimplyHTML this has been done too, so there is a set of API documentation files in directory `doc` of the SimplyHTML distribution package. By opening file `index.html` the documentation can be viewed with any browser.

Stage 2: Resource bundles and common edit functions

While stage 1 has been a comparably big step, stage 2 will be more concise. Here we have a look on how to add resource bundles to a Java application and how they can be used. As well we will add edit functions common to most applications:

- cascading undo and redo,
- cut and paste and
- drag and drop

Using resource bundles

Resources are simply text files that are accessible to an application. The text files contain information in the format

```
key=value
```

and usually are distributed in the path the application classes are located. Class `java.util.ResourceBundle` makes available the data to an application. Applications read constants or parameters at runtime through methods of class `ResourceBundle`.

Why using resources?

Using resources has the following advantages

Maintenance: text constants and parameters can be maintained outside the source code. Changing a text constant or parameter does not require code changes. Information is stored in a central place. If a change is necessary, respective part has not to be searched in the whole source code. Constants have to be changed in only one place regardless of whether they are used in multiple places in the source code.

Internationalization : Resources can be replaced in one step without code changes making it easy to switch an application to another language.

Control: Parts of the application can be controlled dynamically through parameters in a resource file rather than having to 'hard wire' them.

Presenting SimplyHTML in multiple languages

By using class ResourceBundle for all string constants in SimplyHTML the source code does not contain string constants except for internal ones not appearing on the GUI.
In the place of the former string constants in the source code a call to method `getResourceString()` is put. The actual strings are centrally stored in and retrieved from a resource file instead.
By providing resource files for any language, the application shall be presented in, the application can be switched to those languages simply by switching to the particular resource file.

How a language is picked

In Java class `Locale` is used to represent a specific geographical, political, or cultural region. The Locale of the system an application is running on can be determined by `Locale.getDefault()` .
Class FrmMain has a public and static field `resources` referencing the `ResourceBundle` from which all string constants shall be taken. The `ResourceBundle` is initialized to the `Locale` the System is running on. If a resource file for the Locale can not be found, the default candidate is taken, which would be `SimplyHTML.properties`.
If for instance SimplyHTML is running on a German system and a resource file ending with the appropriate language ("_de") is distributed with the application, this resource file will be taken. If `SimplyHTML_de.properties` can not be found, `SimplyHTML.properties` will be taken as the resource file instead.

Creating a dynamic menu

In stage 1 of SimplyHTML the menu consisting of menu bar, menus and menu items had been hard coded in method constructMenu. This part would always have to be changed when the menu changes or is extended by additional functions. In stage 2 of the application we therefore add functionality to build a menu dynamically controlled by parameters from a resource file.

Connect actions dynamically

Certainly actions triggered by menu selections still have to be coded also because they contain the actual functionality in most cases. But how actions are connected to menu items or other GUI elements does not have to be hard coded and therefore actions are included in the change towards a dynamic menu.

Advantage

By having functions to dynamically construct and control a menu, the code does not have to be changed again once a new menu is to be added or changes in the menu structure occur.
Menus and menu items can be added simply by making an entry in the resource file.

How to automate the menu construction

Each action has a name which we use as unique key, such as `new`, `open`, `save`, etc. In class `FrmMain's` constructor a `commands Hashtable` is created with all actions of SimplyHTML and their action commands (`new`, `save`, etc.). With method `getAction`, an action can be fetched by its command name.

Method createMenuBar

To create the menu bar a menu bar definition string from the resource file is read having the key for each menu delimited by blanks (e.g. file edit help). The keys are in the order as menus shall appear in the menu bar.

Method createMenu

To create menus a menu definition string from the resource file is read having the action key for each menu item delimited by blanks. The keys are in the order as items shall appear in respective menu.

Method createMenuItem

Menu items are created with the key to 'their' action (new, save, etc.) as the action command. The key also serves to get the label for the menu item: In the resource file all menu labels are named `fileLabel`, `newLabel`, `saveLabel`, etc. so they can be read automatically and stored with the menu item.

Consistent state handling over components

Menu items always should reflect if their action is available at a certain point in time during execution of an application. Actions in turn should only be available if it makes sense at that point in time. Action close for instance should only be enabled, if there are documents open, that can be closed.

Listeners for interaction between menus, menu items and actions

A reference between menus, menu items and actions is created in several ways to ensure this behaviour:

- The menu item is connected to a `PropertyChangeListener` and registered with the action belonging to that menu item to automatically update the state of the menu item according to the state of the action.
- The action in turn is registered with the menu item as an `ActionListener` so that the action can execute its `actionPerformed` method whenever the menu item fires the action command.
- Finally with each menu a `MenuListener` is registered that updates the action state of each item in respective menu whenever it is about to be displayed. This ensures that always the actions are in the correct state. (In later stages, this has to be refined by updating controls other than menus if such are connected to actions in addition.)

Typical undo/redo parts

In the Java language undo/redo support is made availabe to all text components through a common set of parts, which can be used almost simliarly between different applications dealing with text processing.

UndoManager

An `UndoManager` 'remembers' undoable edit acionst. It is the central place to store such edit actions for later use in a possible undo/redo action.

UndoableEditListener

For being able to store undoable edit actions a class has to implement the `UndoableEditListener` interface. By implementing this interface, an object can register itself with any `Document` whose undoable edit events it likes to handle.
By listening to undoable edit events, a class acting as an `UndoableEditListener` can as well update GUI elements according to the undoable edit events received.

Undo and redo actions

To actually undo or redo an undoable edit, actions are used as a common way to make an undo or redo command available on the GUI. These actions call `UndoManager` methods to undo or redo an undoable edit. As well the actions can be used to adapt the GUI according to the current undo/redo state.

How undo and redo work in SimplyHTML

Currently, only the `Document` class offers undo and redo support to text components. For any edit action that can be cancelled and reinstated again in a particular `Document` , the `Document` sends undoable edit events to all UndoableEditListeners registered with that `Document`.

Creating an UndoManager

To store undoable edit actions for later use in a possible undo or redo action, class `FrmMain` creates an instance of class UndoManager. The object is stored in private field `undo` of class `FrmMain` for later use in the undo/redo implementation.

Defining and registering an UndoableEditListener

Class `FrmMain` also defines an inner class `UndoHandler` that implements the UndoableEditListener interface. In its method `undoableEditHappened` it stores undoable edit actions in the previously created UndoManager. As well, `UndoHandler` updates the GUI whenever an undoable edit event is received.
The `UndoHandler` of `FrmMain` is registered with any `Document` created or opened by SimplyHTML in the respective actions SHTMLFileOpenAction and SHTMLFileNewAction. Consequently `UndoHandler` is properly removed from any open `Document` before it is closed to ensure proper cleanup of the disposed `Document` in the garbage collection.

Defining actions for undo and redo

To finally being able to perform an undo or redo command, either through the menu or from other places such as a tool bar, class `FrmMain` defines two actions `UndoAction` and `RedoAction`. In their `actionPerformed` methods `UndoAction` and `RedoAction` call methods `undo` and `redo` of `FrmMain's` `UndoManager` accordingly. In addition they both have a method `update` to bring their GUI representation in line with the current undo state (being enabled or not depending on whether or not undoable edits are present in the `UndoManager`, that is).

Cut and paste mechanism in Java

Cut and paste uses the clipboard to copy a certain part of a document. Data is transferred from one part of a document to the clipboard and later it is transferred from the clipboard to another place of this or another cocument.

Cut, copy and paste actions and methods

In the `javax.swing packages` there are predefined actions one can use to add cut and paste to an application in an 'Edit' menu for instance. `DefaultEditorKit.CutAction`, `DefaultEditorKit.CopyAction` and `DefaultEditorKit.PasteAction` already deliver functionality by calling a target object's `cut` , `copy` and `paste` methods.
Class `JTextComponent` , which is a superclass to `JEditorPane`, in turn implements these methods to perform the actual cut and paste on a `Document` connected to that `JTextComponent` .

How data is transported

To be able to transport data regardless of its type, the Java language has an object `Transferable` which one has to use or design to certain needs. `Transferables` make use of `DataFlavors` to inform about the data types they support. Transport of `Transferables` in cut and paste operations is done through `Clipboard` objects.

Extending the mechanism

Unfortunately this mechanism is not extended by class `JEditorPane` for the transport of HTML during cut and paste operations. `JEditorPane` is limited to plain text in cut and paste operations even when its content type is set to `text/html`. SimplyHTML therefore extends `JEditorPane` with class SHTMLEditorPane overriding its cut, copy and paste methods so that cut and paste *including* styles and HTML specific parts is possible.

How cut and paste work in SimplyHTML

To allow cut and paste for content *including* styles and all HTML specific parts SimplyHTML extends the cut and paste mechanis of Java. It defines two classes to enable the transport of HTML data and extends `JEditorPane` to use SimplyHTML's classes instead of the standard ones.

Class HTMLText

Class `HTMLText` represents a portion of HTML content. It has field `htmlText` for the HTML code representing the content and field `plainText` to represent the content as plain text. With methods `copyHTML` and `pasteHTML` it enables transport of HTML data into and out of `HTMLText` objects.

Transport mechanism

To transport HTML text methods `copyHTML` and `pasteHTML` use `HTMLEditorKit's read` and `write` methods, which allow to read or write a portion of a `Document's` content as HTML code using a `Writer`. By taking a `StringWriter`, data can be transferred into a `String` for temporary storage inside a `HTMLText` object.

Different mechanism within one paragraph

When copying and pasting text within one paragraph, i.e. without paragraph breaks, method HTMLEditorKit.read makes an own paragraph of the pasted text. HTMLText avoids this behaviour by implementing an alternate copy and paste mechanism.
When not copying multiple paragraphs the selection is split into text chunks. For each chunk of text it's attributes. Each chunk of text is inserted togehter with its attributes when it is pasted to another place in a document.

Class HTMLTextSelection

To transfer data in cut and paste operation a `Transferable` object is needed. A `Transferable` object wraps a data object into a common format describing the contained data to transfer operations. Class `HTMLTextSelection` is a `Transferable` for `HTMLText` objects. Whenever `HTMLText` is to be transported it is wrapped into an `HTMLTextSelection` object and passed to any transfer operation such as copy or paste.

Extending JEditorPane

`JEditorPane` is a subclass of `JTextComponent`. `JTextComponent` has methods `cut`, `copy` and `paste` to implement cut and paste operations which are inherited by `JEditorPane`. To actually use `HTMLText` and `HTMLTextSelection` in an editor, `JEditorPane` has to be extended by an own class named `SHTMLEditorPane` in SimplyHTML. `SHTMLEditorPane` overrides methods `cut`, `copy` and `paste` and uses `HTMLText` and `HTMLTextSelection`

accordingly.

Adding an edit menu

With the functions describing cut and paste and undo/redo in previous chapters we finally have what we need to add common edit functionality to SimplyHTML and to make it available on the GUI.
The edit menu is easily built by adding a new menu definition to our resource bundles: Just a few lines of text describing the new menu and its menu items. The entries in the resource bundle look as follows

```
# edit menu definition
edit=undo redo - cut copy paste
editLabel=Edit

# edit menu items
undoLabel=Undo
redoLabel=Redo
cutLabel=Cut
copyLabel=Copy
pasteLabel=Paste
```

Once above text is added to file SimplyHTML.propertiers, construction of the menu as well as proper connection to actions and functionality to enable/disable the menu items is created automatically by the dynamic menu functionality of SimplyHTML.

Implementing drag and drop

Drag and drop more or less is cut and paste without having to use the menu. As SimplyHTML has implemented cut and paste for styled text and HTML text in stage 2, it has the basis for drag and drop too.

Typical drag an drop

A typical drag and drop operation in the editor would act as follows: By selecting text in the editor and by dragging the selection with the mouse, a copy operation is initiated. Once the selection is dropped anywhere else in the editor, the selection is removed from the original location and pasted at the new location.
To implement drag and drop a mechanism has to be implemented to recognize drag and drop activities and to react in the described way.

The Java Tutorial

There is a good example about how to implement drag and drop in The Java Tutorial. The drag and drop example is well explained there already. At this point therefore only some more general aspects are discussed only.
The Java Tutorial is available online at http://java.sun.com/docs/books/tutorial/
Note: In J2SE 1.4 the original drag and drop is simplyfied by wrapping the 'old' implementation into class TransferHandler partly but as SimplyHTML shall be available to users of the 1.3 Runtime, the 'old' mechanism is implemented which in fact does the same as the 'new' one.

Drag and drop in SimplyHTML

Class `SHTMLEditorPane` has been created in this stage to allow for own cut and paste handling. For drag and drop support it has an additional section in the source code. The main parts of the drag and drop implementation are methods

- `initDnd`
- `dragGestureRecognized`
- `drop` and
- `doDrop`

Method initDnd

Method `initDnd` instantiates the objects necessary to control our drag and drop implementation: `DragSource` and `DropTarget`. The `SHTMLEditorPane` registers itself as a `DropTarget` and as a `DropTargetListener`. In `DragSource` `SHTMLEditPane` registers itself as a `DragGestureRecognizer`. As well it registers as a `MouseListener` to keep track of the selection during drag and drop operations.

Method dragGestureRecognized

`dragGestureRecognized` is the method `SHTMLEditorPane` has to implement to be a `DragGestureRecognizer`. The method is invoked by all drag initiating gestures on `SHTMLEditorPane`.
In method `dragGestureRecognized` an `HTMLText` object is created on the currently selected text and an `HTMLTextSelection` transferable is created wrapping the `HTMLText` object. Finally the drag operation is initiated by invoking method `DragSource.startDrag` with the newly created transferable.

Method drop

Once a drop event is recognized, method drop is invoked. In SimplyHTML it calls method `doDrop` if a suitable DataFlavor (`HTMLText` or `String`) is found in the dragged element. Otherwise the drop is rejected.

Method doDrop

Method `doDrop` does the actual cut and paste resulting from the drag and drop operation consisting of adding the dragged element and necessarily removing the dragged element from the original position.

Stage 3: Font manipulation and tool bars

Most text processors have one essential functionality in common. They allow to manipulate fonts in various ways.
Typically fonts can be set through a menu allowing to change all font related settings for a certain portion of text at once. In addition, there mostly is a tool bar to quickly change single font attributes such as size or style.
Stage 3 of SimplyHTML is about adding font manipulation features. It builds a font dialog to change a whole set of font related attributes and it creates tool bars to access most functions of SimplyHTML including new font formatting.

Customizing Java for CSS

Before we can look at how to build GUI and functionality for font manipulation, we need to understand the way, HTML documents are handled in Java a little more.
As shown previously, text and HTML are used in a model-view-controller environment consisting of EditorKit, Document and EditorPane. However, up to Java 2 Standard Edition version 1.4 this environment supports HTML 3.2 only. Especially it does not totally support CSS elements although attributes are stored partly in CSS format already.
SimplyHTML is based on CSS for dealing with styles and font settings belong to styles so consequently, font settings are implemented using CSS as well.

Design approach

In stage 3 of SimplyHTML font manipulation is enabled for a contiguous run of characters. HTML has tag `SPAN` to set font attributes for a contiguous run of characters using CSS . In addition there are tags `FONT`, `B`, `I` etc. allowign the same without using CSS.
In SimplyHTML universal usage of CSS has been chosen because almost any part of a HTML structure can be formatted with CSS regardless of its type.

Solution approach

To show and manipulate font information for documents with SimplyHTML using CSS on character level, support for the `SPAN` tag has to be built into the MVC environment for HTML documents in Java.
SimplyHTML does this by extending classes `HTMLDocument`, `HTMLDocument.HTMLReader`, `HTMLEditorKit` and `HTMLWriter` accordingly, as described in the next chapters.

Extending classes for tag SPAN

SimplyHTML extends classes `HTMLDocument`, `HTMLDocument.HTMLReader`, `HTMLEditorKit` and `HTMLWriter` to support the `SPAN` tag of HTML. In this chapter the overall approach of how this is done is described. Please consult the source code and API documents of the mentioned classes for additional details.

Why extending the mentioned classes?

Class `HTMLDocument` has an inner class `HTMLReader` which is used by `HTMLEditorKit` to read HTML files. `HTMLReader` does not support `SPAN` tags so it is extended by SHTMLReader accordingly. To use `SHTMLReader` in favor of `HTMLReader`, class `HTMLDocument` has to be extended too.
When a HTML document is edited in SimplyHTML, all attributes are stored as CSS attributes which is why it is also necessary to extend class `HTMLWriter` to write out those CSS styles inside `SPAN` tags. To use both our own reader and our own writer, `HTMLEditorKit` needs to be extended as well.

SHTMLWriter

Usually a class is extended simply by overriding one or more of its public methods. Unfortunately class `HTMLWriter` only has one public method `write` which calls other private methods making up the actual write process.
If not a completely new writer is to be created, the only way to extend `HTMLWriter` is to copy its source code completely into a new subclass `SHTMLWriter` and change some of the code to our needs (please do let me know if you can provide a more elegant way to do this...).
To allow to write `SPAN` tags for style attributes, new class `SHTMLWriter` changes method `convertToHTML32`. Whenever a `CONTENT` tag is encountered during write, any CSS attributes are converted to a syntax used in `STYLE` attributes. The resulting style string then is added to a new `SPAN` tag and the `SPAN` tag including the found styles is added to the `CONTENT` tag.

SHTMLDocument

Whenever data is read into an instance of `SHTMLDocument`, method `getReader` returns an instance of the reader special to this type of document, so this method is overridden to provide an instance of `SHTMLReader` (see below).

SHTMLDocument.SHTMLReader

Class `HTMLDocument` contains an inner class `HTMLReader` to read HTML data into an instance of `HTMLDocument`. To support `SPAN` tags `SHTMLDocument` creates a new inner class `SHTMLReader`.

SHTMLReader overrides methods handleStartTag, handleSimpleTag and handleEndTag which are called by the parser for every tag found. SPAN tags are delivered to the reader through method handleSimpleTag. SHTMLReader deviates this tag to be handled by handleStartTag instead.
In method handleStartTag, for any SPAN tag found an instance of SHTMLCharacterAction is invoked. SHTMLCharacterAction is an inner class of SHTMLReader and extends class CharacterAction of class HTMLReader. It does the actual handling of the SPAN tag by removing the SPAN tag and by adding its style attributes to the CONTENT tag the SPAN tag belongs to.
Method handleEndTag properly terminates SHTMLCharacterActions for any SPAN tag in process.

SHTMLEditorKit

Class HTMLEditorKit provides methods to read and write data stored inside an instance of HTMLDocument. It uses method getReader of HTMLDocument in its read an write methods. To support our own set of classes as described above, class SHTMLEditorKit overrides methods read and write accordingly.
SHTMLEditorKit also ensures that a SHTMLDocument is created instead of a HTMLDocument by overriding method createDefaultDocument.

Manipulating fonts and font styles

Documents are modeled by `Elements` which are hierarchically linked according to the content structure in the document. HTML documents for instance define - with certain exceptions - an `Element` object for every HTML tag. As with HTML tags, each `Element` can have one or more attributes (bold, italic, etc.) which are assigned to an `Element` through class `AttributeSet` .

Applicable ranges for AttributeSets

How `AttributeSets` influence the rendering of a document depends on the context they are found in. Attributes for a paragraph for instance can be valid for a whole range of subseuqent `Elements`. In addition, a HTML document has an associated style sheet which defines `AttributeSets` too. If a paragraph element or an element for a range of characters has no attributes in the document itself, attributes from the style sheet might still be relevant for rendering respective content.

Limitation to attributes on character level

In stage 3 of application SimplyHTML only manipulation of fonts and font styles on character level is implemented. There are two methods in SimplyHTML for dealing with attribute changes, one for reading attributes for a given position in a document and one for applying attributes to a given part of a document.

Method getMaxAttributes

All methods in stage 3 of SimplyHTML dealing with fonts and font styles use static method `getMaxAttributes` of class `FrmMain` to determine which attributes are assigned to a certain point inisde a document. Method `getMaxAttributes` combines attributes from the style sheet associated to the document with attributes assigend directly to to a character element inside the document (in later stages this has to be refined to deal with style sheet styles other than <p>, paragraph styles, etc.).
The attribute sets from the style sheet and from the character element are added to a new attribute set which is returned to the calling method.

Method applyAttributes

As described in the follwing chapters all components manipulating fonts and font styles do their changes entirely on the basis of `AttributeSets`. An `AttributeSet` is applied to a document by method `applyAttributes` in class `FrmMain`.
Method `applyAttributes` determines whether or not a range of characters is selected in the given editor pane. If a selection is present, the given `AttributeSet` is applied to that range of text. If no selection is present, the given `AttributeSet` is applied as attributes for subsequent inputs.

To define attributes for subsequent inputs, class `EditorKit` defines a method `getInputAttributes`. When attributes are stored in the AttributeSet returned by this method, these attributes are applied for inputs thereafter.

Creating a GUI for font manipulation

To use the functionality described in the previous chapters, application SimplyHTML needs additional GUI components the first of which is class `FontPanel`. Class `FontPanel` allows to display and change most relevant font and font style attributes at once. By wrapping all related components into a panel, it is easier to use it in different places such as dialogs later.

Setting and getting attributes

`FontPanel` uses methods `getAttributes` and `setAttributes` to exchange font settings with other objects through an AttributeSet.
In the constructor of `FontPanel` all components implementing the `FontComponent` interface (see below) are added to `Vector fontComponents` . This makes it easy for methods `getAttributes` and `setAttributes` to distribute or collect the attributes to and from the various font manipulation components through an `AttributeSet`.
Methods `getAttributes` and `setAttributes` simply go through all objects in `Vector fontComponents` and call methods `getValue` and `setValue` respectively passing the `AttributeSet` containing the actual font settings.

Components of class FontPanel

`FontPanel` uses GUI components defined in different classes to set the various font attributes:

- font family - `FamilyPickList`
- font size - `SizePickList`
- font style - `StylePickList`
- line effects - `EffectPanel`
- colors - `ColorPanel`

`FamilyPickList`, `SizePickList` and `StylePickList` are inner classes of class `FontPanel` and variations of a separate class `TitledPickList` which defines the general behaviour of a pick list typical for font dialogs having a list, a text field and a title label. All mentioned classes are described below shortly. Please consult the sources and API documents for further details.

TitledPickList

Class `TitledPickList` defines a pick list typically being used in font dialogs, consisting of a list title, a text field for the currently selected value and the actual pick list containing all possible values. It implements listeners for the various events produced by user settings inside its controls to synchronize selections in the text field and the pick list at all times. Then it has some getter/setter methods to programmatically get and set a selection.
It also defines a an `EventListener` and `Event` so that external components can be notified of changes in the `TitledPickList` . This mainly is meant to allow `FontPanel` to update the sample text display whenever a selection changes.

FamilyPickList, SizePickList and StylePickList

Classes `FamilyPickList` , `SizePickList` and `StylePickList` all are subclasses of `TitledPickList`. They extend `TitledPickList` by implementing interface `FontComponent`.

Interface FontComponent

Interface `FontComponent` is used to standardize the way attributes are set and retrieved. It defines two generic methods `getValue` and `setValue`. `setValue` is meant for setting a component from an `AttributeSet`, `getValue` should return the setting of a font component in the form of an `AttributeSet`.

Implementing the FontComponent Interface

Each component implementing the `FontComponent` interface can do the implementation special to the attribute or set of attributes it is meant to manipulate. `FamilyPickList` for instance simply reads `CSS.Attribute.FONT_FAMILY`, `StylePickList` acts on a combination of `CSS.Attribute.FONT_WEIGHT` and `CSS.Attribute.FONT_STYLE` and `SizePickList` uses `CSS.Attribute.FONT_SIZE` and adds certain handling for the 'pt' identifier.

EffectPanel

Class `EffectPanel` is a `JPanel` with a `ButtonGroup` of `JRadioButtons` allowing to select, whether or not a text portion should be underlined or striked out. With `CSS.Attribute.TEXT_DECORATION`, attributes `underline` and `line-through` can not be combined which is why `JRadioButtons` are used allowing only one of the possible selections at a time.
`EffectPanel` implements interface `FontComponent` to set and return the component's value in form of an `AttributeSet`.

ColorPanel

Class `ColorPanel` adds a `JLabel`, a `JTextField` and a `JButton` to a `JPanel` and shows a `JColorChooser` dialog when the `JButton` is pressed. Colors selected from the `JColorChooser` are set as the background color of the `JTextField`. The `JTextField` is not editable, it is only used to show the currently selected color as its background color.
`ColorPanel` implements interface `FontComponent` to set and return the component's value in form of an `AttributeSet`. In addition it defines a an `EventListener` and `Event` so that external components can be notified of changes in the `ColorPanel`. This mainly is meant to allow `FontPanel` to update the sample text display when a color is changed.

Using the new font formatting GUI

Previous chapter described `FontPanel` as SimplyHTML's GUI to set most relevant font and font style settings at once. To actually use class `FontPanel` two additional classes are required. First of all, a dialog is needed to present a `FontPanel` to the user. As well an action to invoke respective dialog must be created.

Class FontDialog

Class `FontDialog` simply wraps class `FontPanel` into a `JDialog` and creates all methods necessary to control the dialog such as closing the dialog with 'OK', cancelling the dialog etc.
In its constructor `FontDialog` expects an AttributeSet which is routed on to `FontPanel` for display and manipulation. Once the dialog is closed by pressing the 'OK' button, method `getAttributes` returns the `AttributeSet` from method `getAttributes` of class `FontPanel`. Method `getResult` returns the information whether the dialog was closed with the 'OK' or the 'Cancel' button.

Action FontAction

With class `FontAction` all prevously descibed font functionality is 'plugged' into SimplyHTML's mechanism to make functions avaliable on the GUI. Its method `actionPerformed` creates an instance of class `FontDialog` and applies font changes from `FontDialog` to a document. `FontAction` is added to the commands Hashtable of FrmMain through method `initActions` and its name is reflected in the action name constants of FrmMain ensuring proper usage during dynamic menu creation.
To always reflect proper state to components bound to `FontAction`, it implements interface `SHTMLAction` with method `update`.

Actions and components to switch single font attributes

On top of being able to change most relevant font settings at once using class FontPanel, a couple of actions and components are needed to allow users to toggle or switch single font attributes quickly. In stage 3 of SimplyHTML this is done by adding some inner classes to `FrmMain` implementing respective parts:

- `FontFamilyPicker` - a JComboBox dedicated to font family changes
- `FontSizePicker` - a JComboBox dedicated to font size changes
- `FontFamilyAction` - an action to change the font family
- `FontSizeAction` - an action to change the font size
- `ToggleFontAction` - an action to toggle a single font setting on or off

FontFamilyPicker and FontSizePicker

The easiest way to act on a certain font setting probably would be an action bound to a `JButton` . For font properties family and size however, a possible setting is not just 'on' or 'off', for both attributes there a is a certain list of possible selections instead. For this type of setting a `JComboBox` is the GUI component of choice.

Extending JComboBox

To make such `JComboBoxes` easier to handle, two inner classes `FontFamilyPicker` and `FontSizePicker` extend class `JComboBox` with functions special to the purpose of changing respective font settings.

Customized content and common interface

`FontFamilyPicker` adds all font family names found on the particular system to its combo box using method `getAvailableFontFamilyNames` of class `GraphicsEnvironment` in its constructor. `FontSizePicker` adds a fixed list of point sizes instead. Both classes implement interface FontComponent for standardized access to their selected value.

FontFamilyAction and FontSizeAction

Both actions implement interface `SHTMLAction` so that common handling of setting action properties from our resource bundle and common updating can be used. In their actionPerformed method they apply the attribute represented by their associated picker component (family or size) to the editor.

ToggleFontAction

`ToggleFontAction` allows to switch a single font setting on or off in a generic way. It extends `AbstractAction` by defining some private fields reflecting the font attribute this instance of `ToggleFontAction` represents as well as the value for 'on' and 'off' for that particular font attribute. In the constructor, those fields are initialized from respective arguments passed to the constructor.

Shifting state

Method `actionPerformed` applies the font attribute resulting from the current state (on or off) and then toggles the action's state using method `putValue`. By passing either value `FrmMain.ACTION_SELECTED` or `FrmMain.ACTION_UNSELECTED` with key `FrmMain.ACTION_SELECTED_KEY` to method `putValue`, respective value is stored in the action's poperties table causing a `PropertyChangeEvent` being fired. Any listener to such events can then update its state accordingly.

Interfaces

`ToggleFontAction` implements interface `FontComponent` so that its value can be changed in a standard way from other objects through methods `getValue` and `setValue` . To always reflect proper state to components bound to `FontAction`, it implements interface `SHTMLAction` with method `update`.

Integration to FrmMain

Method `initActions` of class `FrmMain` initializes three instances of `ToggleFontAction` to the central `commands` `Hashtable` , one for `CSS.Attribute.FONT_WEIGHT`, one for `CSS.Attribute.FONT_STYLE` and one for switching `CSS.Attribute.TEXT_DECORATION` between `normal` and `underline` . For each of the three instances a separate action command is created in the constants list of class `FrmMain` for proper handling in dynamic menu and tool bar creation.

Creating a font formatting tool bar

As we have created functionality to manipulate font settings as described in the previous chapters, now it would be handy to have certain font formatting functions availabe in a tool bar as it is done in other text processors too.
Creating a font formatting tool bar for that purpose is easily done though a mechanism we already know from SimplyHTML's dynamic menu creation function. Method `createToolBar` uses the same technique by reading a tool bar definition string and turning it into a tool bar.

Method createToolBar

To create a tool bar a tool bar definition string from the resource file is read having the key for each element in the tool bar delimited by blanks (e.g. fontFamily fontBold fontItalic). The keys are in the order as elements shall appear in the tool bar.

Standard tool bar buttons

The typical case is to add a button on the tool bar for an action defined in the `commands Hashtable` of class `FrmMain`. Class `JToolBar` has a constructor returning a newly created button by passing an action to the constructor. The constructor will do all the connections between the tool bar button and the action automatically.

Combo box elements

Some of the elements in the tool bar however require special handling. `FontFamilyPicker` and `FontSizePicker` for instance are subclasses of `JComboBox`. In their case, `createToolbar` creates an instance of the component and uses method add of `JToolBar`.

Toggle buttons

`FontComponents` other than `FontFamilyPicker` and `FontSizePicker` are instances of `ToggleFontAction`. For `ToggleFontActions` we need a `JToggleButton` instead of a `JButton` in the tool bar and we have to make sure, the `JToggleButton` is property connected to its `ToggleFontAction`.
For each `JToggleButton` in the tool bar a `ToggleActionChangedListener` associated with the corresponding `ToggleFontAction` is created. `ToggleActionChangedListener` implements interface `PropertyChangeListener` and will always adjust the `JToggleButton` according to the action's current state. An `ActionListener` in turn is registered for the `JToggleButton` invoking the action when the button is pressed.

Synchronizing tool bar and document

Font formatting controls in the tool bar not only allow to act on certain font settings in a document, they should also be used to reflect the settings at the current caret position. Class `FrmMain` implements interface `CaretListener` for doing this.

Method caretUpdate

Method `caretUpdate` in class `FrmMain` calls method `updateFormatControls` (see below) whenever the caret changes in the curently active document. `updateFormatControls` is called by `FontAction` too because this action also changes font attributes but the caret position does not change in this case.
`caretUpdate` is registered with every newly opened or created document through method `registerDocument`. Method `unregisterDocument` takes care of removing any listener when a document is closed.

Method updateFormatControls

`updateFormatControls` gets the attributes for the current caret position and calls method `setValue` of any `FontComponents` found in the format tool bar.

Adding a standard tool bar

Already having all functions of SimplyHTML as actions connected to the menu bar and having a method createToolBar with stage 3 of SimplyHTML as described previously, creating additional tool bars is done with almost no additional effort.
To create an additional tool bar for standard actions such as create a new document, open or save a document, for instance an additional tool bar definition in the resource bundle has been prepared.
The additional tool bar definition is read by calling `createToolBar` in method `customizeFrame` of class `FrmMain`. The new standard tool bar then is added to the panel on top of the main frame where our font formatting tool bar is located too.

Stage 4: Tables

Implementing support for tables is a comparably complex task because there are no special objects for a table, table row or table column inside a HTML document. Each table part is represented by elements hierarchically linked, each element having many attributes. Iterating through all cells of a table column for instance needs a special way of handling for this reason.
To complicate things a little, there are only comparably limited ways to manipulate table elements in a document in Java. An additional challenge is to support table borders in Java because up to J2SE 1.4, table cell rendering is not appropriate compared to existing text processors when it comes to borders.
This stage of SimplyHTML implements support for tables trying to solve these limitations. In the follwing chapters is described how this is done in more detail.

Due to the complexity of the topic the documentation does not cover all details of the resulting source code completely. The source code itself should be taken in addition to understand how the implementation is accomplished.

Table manipulation parts to implement

Table manipulation is divided into several parts

- Table creation
- Table content changes and caret movement
- Table and cell attribute changes
- Changes to the table structure

Implementation of these parts is distributed over different parts of application SimplyHTML. The following table is a summary of changes to SimplyHTML to implement table support:

Class	changes
FrmMain	actions needed to interface table manipulation functionality with the GUI
SHTMLEditorPane	table structure manipulation appliance of attribute changes from TableDialog caret movement inside tables keymap and actions for caret movement
SHTMLBoxPainter	new class for table cell rendering
SHTMLWriter	new class with own implementation of an HTML writer (replaces SHTMLWriter of former stages completely)
SHTMLDocument	support for manipulation of element attributes additional support for removing elements
LengthValue	class to represent a CSS length value divided into value and unit
SHTMLBlockView	extension of BlockView to support SHTMLTableView
SHTMLTableView	extension of TableView to support individual rendering of cell borders
TableDialog	Dialog for table attribute changes
DialogShell	new common base class for dialogs of application SimplyHTML
AttributeComponent	Interface to replace interface FontComponent of former stages of SimplyHTML
SHTMLEditorKit	extended ViewFactory for support of SHTMLTableView
BoundariesPanel	Panel to show and manipulate boundaries of a

	rectangular object such as a table cell
SizeSelectorPanel	Panel to show and manipulate a CSS size value
CombinedAttribute	Class to model CSS shorthand properties
BorderPanel	Panel to show and manipulate properties of table cell borders

As seen from above list, many classes are affected by table support in SimplyHTML. The major functionality however is in `SHTMLEditorPane` and `TableDialog`. Details of the implementation are described in the following chapters.

Table structure in documents

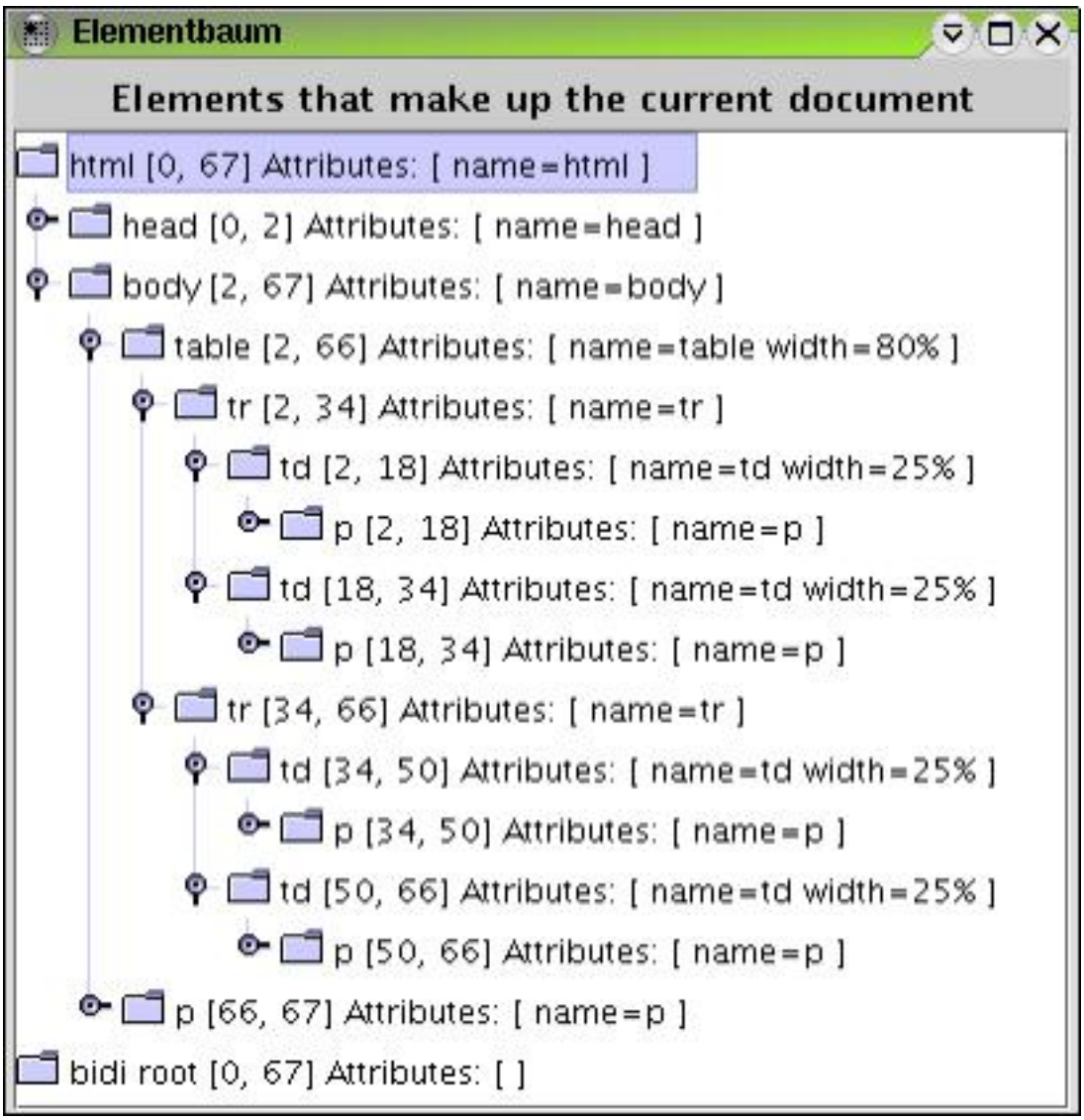
As mentioned previously, Documents are modeled by `Elements` which are hierarchically linked according to the content structure in the document. To manipulate a table structure it is necessary to know how a document models HTML code for a table.
In HTML a table is coded like this

```
<table>
  <tr>
    <td>
      <p>
        row 1, column 1
      </p>
    </td>
    <td>
      <p>
        row 1, column 2
      </p>
    </td>
  </tr>
  <tr>
    <td>
      <p>
        row 2, column 1
      </p>
    </td>
    <td>
      <p>
        row 2, column 2
      </p>
    </td>
  </tr>
</table>
```

Rendered inside a document above HTML code might show as follows (display differs depending on style sheet settings)

row 1, column 1	row 1, column 2
row 2, column 1	row 2, column 2

The element strucutre to be generated inside a document has to be built similar to the HTML code above. Above table viewed with the ElementTree class in SimplyHTML would produce a view such as the following.



To manipulate a table or its parts, an application has to work on that element strucutre and its attributes.

Note: To find out or try how a document's element structure look like, SimplyHTML's ElementTree function is quite helpful. It shows a window as shown with a tree having a node for each element in the element structure of the currently shown document. All element attributes are shown next to each tree node.

Creating a new table

Compared to other table functions, to create a new table and to insert it into a document is a quite simple task. In SimplyHTML this is done with method `insertTable` of class `SHTMLEditorPane` (see below). This method is called by a new action of class `FrmMain` which allows this method to be connected to menus and tool bar buttons, etc.

Method insertTable

Method `insertTable` builds HTML code for an empty table having one row with a given number of cells. The number of cells to create is passed as a parameter so a calling method can implement a function asking the user for the desired number of table columns.
The generated HTML code then is inserted into the document of the `SHTMLEditorPane` by inserting it after the current paragraph element using method `insertAfterEnd` of class `HTMLDocument`.

Generating HTML with class SHTMLWriter

Package `javax.swing.text.html` already provides classes to generate HTML for a given `Document` . Class `HTMLWriter` of this package is meant for this job with the help of classes `AbstractWriter` and `MinimalHTMLWriter`. Unfortunately these classes can not be used in the way it is needed by application SimplyHTML.
In stage 3 of SimplyHTML we already extended HTMLWriter with support to generate `SPAN` tags for character level attributes. To use the writer in the new context described here, finally it has been reimplemented completely so class `SHTMLWriter` now is a completely rewritten class not being a subclass of classes of the Swing package of Java anymore.

Reusing methods of SHTMLWriter

As `SHTMLWriter` writes HTML code to any output writer passed as an argument, we can use it for generating an empty table as well simply by passing a `StringWriter` as the target for writing. Usually `SHTMLWriter` produces HTML based on the element structure of a given document. The methods necessary to do so however can be used to generate HTML not related to a document too.
`SHTMLWriter` provides two methods `startTag` and `endTag` which can be used to generate start and end tags as needed. Method `startTag` accepts a set of attributes too, so start tags can be generated with appropriate HTML and CSS attributes if necessary.

Using SHTMLWriter in method insertTable

To generate HTML for an empty table as described above, `SHTMLWriter` is instanciated to write to a new `StringWriter`. Methods `startTag` and `endTag` are called for the table, row and cell tags accordingly passing a set of attributes having the applicable table and cell widths. The `StringBuffer` of `StringWriter` is converted to a `String` and inserted into the document

finally.

Enabling element and attribute changes

Before we take a closer look on table manipulation in the following chapters, some techniques used in application SimplyHTML for doing element and attribute changes shall be discussed here. Some functions are available through common methods in classes of package `javax.swing.text` and `javax.swing.text.html`, others need to be enabled by own methods.

Adding elements

To add an element such as table, row or cell tags to a document, application SimplyHTML uses methods `insertBeforeStart` and `insertAfterEnd` of class `HTMLDocument`. These methods accept an HTML string to be inserted before or after an existing element of the document.

Removing elements

Almost any removal from an HTML document can be done with method `remove` from class `AbstractDocument`. Method `remove` is passed the start position inside the document and the length of the portion to remove. For some reason, this does not work on the last column of a table (explanations welcome!). An additional method `removeElements` in class `SHTMLDocument` is provided as an additional way to remove elements working for the last table column too. This method does basically the same as `remove`.

Changing attributes

Adding, removing and changing arbitrary attributes all can be done with the help of class `MutableAttributeSet` and its subclasses. A `MutableAttributeSet` is created by getting an `AttributeSet` from an `Element` and casting it to a `MutableAttributeSet`. Changes to that `MutableAttributeSet` then directly affect the `Element` the attributes belong to. `HTMLDocument` however does not provide a way to change attributes in such way. Class `SHTMLDocument` therefore delivers a method `addAttributes` for changing attributes of an `Element` instead.

CSS shorthand properties

CSS 'shorthand properties' allow to store a group of properties in one single property which shortens the way the properties are stored. If for instance a margin should be specified for an object that margin usually applies to a certain side such as top, or left. To store individual margins for all four sides of an object one can specifiy four CSS attributes for each of the four sides or the individual properties of all four sides can be stored in one shorthand property.
To store values in a shorthand property, they have to follow the order top, right, bottom, left. Individual values can be omitted, if some or all values are equal. E.g. if the margin of all sides is the same, only one value needs to be stored in the shorthand property which will be taken for all four sides.
***Example with four equal values**:* `margin:0pt;`
***Example with four different values**:* `margin:0pt 1pt 2pt 3pt;`

Shorthand properties used with HTML tables

For elements of HTML tables modeled by application SimplyHTML the following shorthand properties can be used to shorten attribute expressions inside individual tags:

- `margin`
- `padding`
- `border-width`
- `border-color`

Class CombinedAttribute

To enable usage of shorthand properties, application SimplyHTML provides class `CombinedAttribute` . `CombinedAttribute` models a CSS shorthand property by providing methods to manipulate and store four individual CSS properties in one CSS shorthand property. It is used in classes `SHTMLBoxPainter` to render table cells, `SHTMLEditorPane` to manipulate tables and in `SHTMLWriter` for writing CSS shorthand properties.

Transforming CSS properties to CSS shorthand properties

In the Java languages all CSS shorthand properties are transformed to 'normal' CSS properties when HTML and CSS is modeled (in an `HTMLDocument` for instance). So for any CSS shorthand property four individual CSS attributes are created for an element.
`CombinedAtrribute` is constructed from an `AttributeSet` which may have CSS attributes belonging to a CSS shorthand property or not, so it does not matter whether or not the model uses CSS shorthand properties. When HTML code is to be generated for HTML file creation however, 'normal' CSS properties belonging to a CSS shorthand property need to be transformed from the model to the file accordingly.
Class `SHTMLWriter` does that by initializing a table of CSS properties for which CSS shorthand properties are to be generated. When creating HTML code, method `writeAttributes` filters out

those single CSS atributes, creates CSS shorthand properties for them, and writes out these instead.

Manipulating the table structure

Manipulation of an existing table structure is necessary for following actions

- append row
- append column
- insert row
- insert column
- delete row
- delete column

Each of the above table manipulations is implemented in class `SHTMLEditorPane` with repective methods (`insertTableColumn`, `appendTableCol`, etc.). All table manipulation methods of class `SHTMLEditorPane` have following similarities.

Common logic

Insertions and additions of rows and columns are all done by using methods `insertAfterEnd` and `insertBeforeStart` of class `HTMLDocument` respectively. Deletions of rows and columns are both done by using methods `remove` and `removeElements` of class `HTMLDocument` respectively.
All table manipulation methods assume that they are called while the caret is somewhere inside a table cell. If not, they do nothing. As opposed to attribute changes the table manipulation methods are designed for being called with a single action command ('delete row', 'insert column', etc.).

Adding rows

To add a row, the current (insert) or last (append) row is copied by iterating the row and cell elements and creating an HTML string making up that element structure including attributes but without text content. The resulting HTML code is inserted before the current row element (insert) or inserted after the last row element (append) by use of method `insertAfterEnd` and `insertBeforeStart` of class `HTMLDocument` .

How it works

To accomplish the above functionality method `createNewRow` is shared by methods `insertTableRow` and `appendTableRow` of class `SHTMLEditorPane`. Method `createNewRow` uses `getTableRowHTML` of class `SHTMLEditorPane` to do the actual assembling of HTML code. Method `getTableRowHTML` in turn uses methods `startTag` and `endTag` of class SHTMLWriter to generate HTML.

Entry point for Actions

Methods `insertTableRow` and `appendTableRow` are as well the entry points for respective actions of class `FrmMain` to connect this functionality with GUI elements such as menus and tool bar buttons. They both find out the table row (current or last) by determining the current table cell with the help of method `getCurTableCell`. Method `getCurTableCell` is discussed in more

detail in the chapter about how to implement a customized caret movement and key mapping.

Removing rows

Removing a table row is comparably simple. Because a table row is represented by a single element with child elements belonging to that row only, it is sufficient to just delete this particular element from the document strucutre.
To remove a row method `deleteTableRow` is called. It is as well the method used in `FrmMain's` respective action. Method `deleteTableRow` gets the row the caret currently is in and deletes it by calling method `removeElement`.

Adding columns

As opposed to working with rows, table columns are harder to manipulate because the cells of a column are spread over all row elements. To add a column, the same logic is used as in adding rows except that method `createTableColumn` iterates through all rows of a table working on the particular cell belonging to the column in question in each row.

Retaining table width

Another exception is that SimplyHTML adjusts cell widths by taking half of the width of the current column for the new column. In method creatTableColumn the half width is applied to the column the new column is to be inserted before. Then the new column is created with the same width so that in total the table width did not change.

Removing columns

To remove a column again the same logic is used as with rows but respective method `deleteTableCol` is the most complicated of table manipulation methods.

Retaining table width

`deleteTableCol` first determines which column to increase in width after removal of the current column. By default the column on the left of the current column is taken. If the current column is the first in the table the column right of the current column is taken instead.
The method then gets the width values of both columns and finds out the sum of both widths. The sum is only taken if the unit of both width values is the same (both percent or point). If a sum could be taken, it is added to an attribute set.

Removing cells

`deleteTableCol` then iterates through all rows in the table removing the cell of each row belonging to the column to remove and then adds the new width to its adjacent cell left or right respectively. To remove a cell method `removeElements` of class `SHTMLDocument` is used. For some reason I did not find out up to now why but method `remove` of class `HTMLDocument` does not work when used on the last column in a table.

Enhancing cell border rendering

A prerequisite to manipulation of table and cell attributes as described separately is to provide some enhancement to the way cell borders are rendered by Java.
Up to J2SE 1.4 cell borders are not rendered individually and there is no way to have different colors for borders of different sides of a cell. Either a border is drawn around all sides of a table cell or no border is drawn. There is no way for example to draw a vertical border between two cells only while the other sides of these cells have no borders.

Rendering mechanism

In general `Elements` of an `HTMLDocument` are rendered through the pluggable design construct of `HTMLEditorKit.HTMLFactory`. The idea behind this design is to provide individual `Views` to render `Elements` .

Parts involved in cell border rendering

A table cell is rendered by class `BoxPainter` which is an inner class of class `StyleSheet.BoxPainter` is used in class `BlockView` which is a superclass of class `CellView`. `CellView` in turn is an inner class of class `TableView` (terrible isn't it?).
To change how borders are painted, `StyleSheet.BoxPainter` needs to be replaced by an own class. `TableView` could be subclassed and its `create` method could be reimplemented to provide a replacement of `CellView` replacing `StyleSheet.BoxPainter`.

Enabling for individual border rendering

The constructor of `TableView` is public but unfortunately the class itself is protected so there is no way to simply subclass `TableView` to replace the `ViewFactory` of `TableView` with an own `CellView`. It is dificult to change the rendering while leaving the underlying classes untouched due to `TableView` being protected (I did not want to write a complete new view or view factory only to change a little part - a complete new table view would be hard to write too...).

Solution

`SHTMLBoxPainter` is created allowing to draw borders around a table cell independently from each other. Width and color for each side are drawn independently and borders of adjacent cells are adjusted so that only one border is drawn instead of two when the adjacent cells have no margin..
To enable `SHTMLBoxPainter` in place of `StyleSheet.BoxPainter` the sources of the superclasses `BlockView` and `TableView` are copied unchanged into new ones and only bring in `SHTMLBoxPainter` where appropriate. This is done by classes `SHTMLBlockView` and `SHTMLTableView` respectively. Both classes had to be put into package `javax.swing.text.html` to do so.

Due to class `TableView` being protected admittedly this is an ugly solution so any other and more elegant and effortless one is welcome and highly appreciated!
Highly appreciated also would be an explanation why `TableView` is protected...

Changing table and cell attributes

In the previous chapters basic methods for creating and manipulating a table *structure* are explained in detail. In this chapter it is discussed how to select and apply *attribute* changes to an existing table structure.

Structural changes vs. attribute changes

Structural changes to a table (insert row, delete column, etc.) all can be done in a single step. To add these functions to the GUI of an application, a single menu item or tool bar button is sufficient. A GUI for table attribute changes is achieved not as easy. There are many attributes each talbe element can have and it would be very tedious to change single attributes through single menu items each.
Most of the time, attribute changes are to be applied as a group of changes to a group of elements as one (for instance changing all cells of one column to a certain width and background color). With class `TableDialog` a new dialog for changing table attributes is created therefore.

Introducing TableDialog and DialogShell

It is called through new action `FormatTableAction` of class `FrmMain`. With class `TableDialog` the second formatting dialog is introduced after class FontDialog which is why a new base class `DialogShell` is created too. `DialogShell` has all methods shared by dialogs of application SimplyHTML thus avoiding code redundancies.

Class TableDialog

`TableDialog` wraps all attributes of tables and table cells changeable in SimplyHTML into one dialog. It partly uses components already used in class `FontPanel` and partly introduces additional components.

Common setting and getting of attributes

Class `AttributeSet` in package `javax.swing.text` provides a good way of grouping an arbitrary number of attributes and passing them between elements and components. For this reason application SimplyHTML uses interface `AttributeComponent` (renamend from FontComponent of stage 3) to define a common way of setting and getting attributes to and from GUI components via `AttributeSets`.
All components of `TableDialog` are implementing interface `AttributeComponent`. They are held in two `Vectors`, one for table attributes and one for table cell attributes. Whenever a `TableDialog` is created to reflect a current set of attributes existing for a table and table cell, simply respective attribute sets are passed to methods `setTableAttributes` and `setCellAttributes`.
Both methods then iterate through the mentioned component `Vectors` calling method `setValue` on each of their components. Each component then picks its attribute(s) from the attribute set and

displays them accordingly. Similarly, attributes are returnd by `TableDialog` with methods `getTableAttributes` and `getCellAttributes`. Again these methods iterate through the component `Vectors` to call method `getValue` on each component returning attribute sets with the sum of all changed attributes.

Returning only changed attributes

All components of `TagbleDialog` 'remember' the original attribute value and only return an attribute when it was changed compared to that original value. This mechanism ensures only attributes being applied, that have been set through the dialog although other attributes were shown in the dialog as well. Without this mechanism always all attributes would be returned by the dialog regardless of whetheher they changed, returning only changed attributes avoids redundant storage of attributes.

Applying attributes returned by TableDialog

To apply table attributes method `applyTableAttributes` of class `SHTMLEditorPane` is called. It gets the table element from the current caret position and passes it to method addAttributes of class `SHTMLDocument` along with the attributes to apply.
Basically the same is done for applying cell attributes with the difference that a range of cells is passed in addition. Depending on the users choice to apply attributes to the current cell only, the current column, the current row or all cells of the table, method `applyCellAttributes` of class `SHTMLEditorPane` iterates through the appropriate range of table cells and calls method `addAttributes` of class `SHTMLDocument` accordingly.

Caret movement in tables

As SimplyHTML now has all functionality to create and manipulate tables, it has to provide a way to move the caret inside a table conveniently. Most text processors usually allow to jump to the next or previous cell with the tab key while the caret is inside a table cell. Class `SHTMLEditorPane` therefore has an own section of methods dealing with this kind of caret movement.

Method getCurTableCell

With method `getCurTableCell` the caret position inside a table is determined. It returns the cell the caret currently is in or null, if the caret is not inside a table. This is done by using method `findElementUp` of class `Util` which looks for the next occurrence of a certain element (`TD` in this case) starting at a given element (the character element at the current position in this case). This method is used in almost any table related methods.

Methods getFirstTableCell and getLastTableCell

When the caret shall be moved from one cell of the table to another, it has to be determined if there are cells to move to from the current cell in a certain direction (previous or next). Methods `getFirstTableCell` and `getLastTableCell` return the first and last cell in a table given any cell of that table.

PrevCellAction and NextCellAction

Actions are used to actually move the caret from one cell to the next or previous one. Actions `PrevCellAction` and `NextCellAction` use above methods to determine the next or previous cell to move to and then place the caret into that cell. Both actions are added to the key map of `SHTMLEditorPane` with method `adjustKeyBindings` upon construction of the editor pane. `NextCellAction` is connected to the `TAB` key, `PrevCellAction` is related to `SHIFT TAB` . For the case that the caret is not inside a table, both actions store the original action found in the key map for `TAB` and `SHIFT TAB` respectively. If a table action is invoked by `TAB` or `SHIFT TAB` thereafter and the caret is not inside a table cell, the original action for the associated key is invoked.

Stage 5: Plug-ins, user settings and dynamic resources

While SimplyHTML concentrates completely on text processing for HTML/CSS documents, it shall not be limited to creating and editing such documents. By delivering a solid and powerful editor for single documents it can be the basis for other functions too. Added functionality however is not in the scope of SimplyHTML which is where a plug-in mechanism comes into view.
A plug-in mechanism could allow almost any extension to SimplyHTML while preserving the original scope, leaving additional functions to potential plug-ins. This stage implements such plug-in mechanism along with an enhanced way of working with resource bundles suitable for plug-in usage. To allow users to configure plug-ins individually, a simlpe way of persistently storing user prefernces is implemented too.
The plug-in mechanism is created with the idea of future enhancements according to the needs of plug-in developers. This is best done in an evolutionary process which is why it is started in this stage instead of waiting until additional edititing functions are finalized.

Requirements

A plug-in architecture for SimplyHTML has to meet the following requirements

- allow to incorporate additional functionality without the necessity to change parts of SimplyHTML
- provide a single interface for external objects to plug-in to SimplyHTML
- allow to access functionality of external objects from within SimplyHTML without SimplyHTML 'knowing' about the particular functions
- allow plug-ins to use SimplyHTML's functionality

Read on to find out how the plug-in implementation meets these requirements.

Parts of the plug-in architecture

To meet the previously stated requirements SimplyHTML provides the following new parts

Part	Description
SHTMLPlugin	The interface all plug-ins to SimplyHTML must implement
AbstractPlugin	A base class plug-ins can extend
PluginManager	Class to find and load plug-ins
FrmMain	Extended by an initialization method for plug-ins using the PluginManager
PluginTemplate	Class and properties files forming a basic plug-in for explanatory purposes and as copy template for plug-in creation
ManagePluginsAction	Action to show a PluginManagerDialog
PluginManagerDialog	Dialog for managing plug-ins (activate/deactivate, dock location, etc.)

While application SimplyHTML is distributed in package `com.lightdev.app.shtm`, the above parts are in package `com.lightdev.app.shtm.plugin`. Package `plugin` is also the root package for all plug-ins to be added to SimplyHTML.

Incorporating plug-ins at runtime

To give access to functions of external objects, SimplyHTML adds a plug-in menu to its menu bar. For each plug-in one menu item is added to the plug-in menu. The menu item is to be provided by the plug-in and typically would contain one or more submenus with the functionality delivered by the plug-in.
In the same way SimplyHTML creates a new menu item in the help menu so that the plug-in can provide documentation about the way it is working.
If the plug-in delivers a component to SimplyHTML, it is displayed by SimplyHTML either as a new window or as a panel inside a section of SimplyHTML's main window, whatever is requested by the plug-in.

Plug-in initialization

Upon construction class `FrmMain` uses method `initPlugins` to add all plug-ins present to the application. Method `initPlugins` uses class PluginManager to locate and load plug-ins. In method `initPlugins` a new instance of class PluginManager is created. All plug-ins returned by method `plugins` of PluginManager are iterated and their parts (plug-in menu, help menu and component so far) are added to SimplyHTML as described above.

Class PluginManager

Class `PluginManager` finds and loads plug-ins. Upon creation it calls its method `loadPlugins` which in turn calls method `findPlugins` to locate any plug-ins to be loaded and to create a class loader for them. Method `findPlugins` uses method `readJar` to get the class names from any Java archive (JAR) file found. `readJar` finds out the URLs for respective plug-in classes too. Once all JAR files are searched, the found classes are loaded by method `loadPlugins`. Method `plugins` returns all loaded plug-ins as an Enumeration.

Restrictions for making plug-ins available to SimplyHTML

Plug-in classes and their accompanying classes are to be installed as Java archive (JAR) files. They are to be placed into package `com.lightdev.app.shtm.plugin.installed`. The plug-ins in this package need to be present in the directory where package `com.lightdev.app.shtm` is installed. This restriction makes it easier and faster to locate and load plug-ins.
If the actual plug-in class (the class implementing interface `SHTMLPlugin`, that is) needs additional classes as part of a plug-in package, the additional classes are best placed in sub-packages of package .`plugin.installed` . This makes it faster to load the actual plug-in classes too.

How plug-ins are found

`PluginManager` looks for JAR files in the directory where the class file of `PluginManager` is installed (the application directory of SimplyHTML, that is). `PluginManager` opens any JAR file found and goes through all content of the JAR file. Any class name found in package `com.lightdev.app.shtm.plugin.installed.` is stored along with its URL.
A class loader is created for the found URLs and all found classes are loaded.

Dynamic resources

In stage 2 of SimplyHTML a dynamic way of using resource bundles was implemented. It is capable of providing internationalization support and of dynamic creation of components such as menus and tool bars from parameters from a resource bundle.
This functionality needs to be made available to plug-ins as well which is why a new class `DynamicResource` now provides these features formerly contained in class `FrmMain`.

Class DynamicResource

Class `DynamicResource` provides methods for menu and tool bar creation based on parameters stored in ResourceBundles. As well, it stores and associates actions with components created in such way allowing for rerieval of component by their associated action name.
Class `FrmMain` has been changed to now use class `DynamicResource` for all internationalization and menu creation. It now makes publicly available a static instance of `DynamicResource` referencing components and actions of SimplyHTML. By having one static instance of DynamicResource in FrmMain, any object can use its utility methods without instanciating DynamicResources again.
All classes relying on `FrmMain's` former functionality have been changed accordingly. Please see the source code and SimplyHTML's main `.properties` file for examples of how parameters can be created for automatic component creation.

Using class DynamicResource for plug-ins

Class `AbstractPlugin` is an abstract base class plug-ins can use. By extending class `AbstractPlugin` a plug-in inherits some automatic initialization methods being performed upon construction. If a `.properties` file exists in the plug-in package (e.g. `com.lightdev.app.shtm.plugin.installed.MyPlugIn.properties`), class `AbstractPlugin` automatically creates a ResourceBundle for that `.properties` file. It then uses FrmMain's DynamicResource instance to create menus from the menu definitions found in that .properties file.
If a different approach of plug-in construction is desired, a plug-in class either can be declared not to extend class `AbstractPlugin` or can override some or all methods of class `AbstractPlugin` accordingly.

Creating plug-ins for SimplyHTML

A plug-in can be any Java object implementing interface `SHTMLPlugin`. As specified by interface `SHTMLPlugin`, a plug-in should provide

- a menu delivering access to all of the plug-ins functions (plug-in menu)
- a menu providing documentation about the plug-in as required (help menu)
- a component containing any GUI representation of the plug-in (plug-in component)
- an indicator telling whether the plug-in component shall be 'docked' to SimplyHTML's main window (and where by default) or displayed in a separate window.
- its name as it should be represented on a GUI
- its internal name for possible internal referencing

Above parts need not to be present in all cases, returning `null` in one or more parts is sufficient as well. Apart from above specification, plug-in developers are free to design their plug-in in any way they like. As SimplyHTML is open source (and will remain as such), plug-ins can access all its public parts as described in SimplyHTML's API documentation.

Class AbstractPlugin

To further simplify plug-in creation, there is an abstract base class that can be used to build a new plug-in upon. It basically implements a generic way of creating menus from a resource bundle (if one is delivered with the plug-in). See chapter 'Dynamic resources' and the source code for class `AbstractPlugin` for details.
Plug-ins are not obliged to extend `AbstractPlugin` or they can override some or all of its classes to adjust the plug-in accordingly. When extending AbstractPlugin unchanged, a `.properties` file should be provided with the new plug-in having menu definitions, etc.

Class PluginTemplate

Another help for plug-in developers is class `PluginTemplate`. It constructs a working - though functionless - plug-in for SimplyHTML. Plug-in developers can use it as a copy template for own plug-ins or for explanatory purposes.
With class PluginTemplate two .properties files are provided too. They have the necessary menu definitions and texts for english and german language. Use these .properties files as an example for how to define appropriate .properties files for you plug-ins.

Defining actions for plug-ins

A plug-in typically adds functionality to SimplyHTML which can best be provided to SimplyHTML through respective action classes. To integrate action classes of plug-ins with SimplyHTML, they should be connected to plug-in menu items.
Actions from plug-ins are added to SimplyHTML by using FrmMain's DynamicResource instance. Call method addAction of class DynamicResource passing it an instance of a plug-in action along with its command name. The action command should has to be the same expression as it was

used to identify the menu item in the ResourceBundle. See method `initActions` of class `FrmMain` for an example.

Making plug-ins available to SimplyHTML

Plug-ins need to reside in Java archive (JAR) files. Inside the JAR file a plug-in must be in package `com.lightdev.app.shtm.plugin.installed`. If a plug-in is accompanied by `.properties` files for internationalization or dynamic menu creation, the `.properties` files have to reside in package `com.lighdev.app.shtm.plugin.installed` too.
Should there be additional classes distributed along with the actual plug-in class (the class implementing interface `SHTMLPlugin`, that is), these additional classes should be placed into sub-packages such as `com.lightdev.app.shtm.plugin.installed.mypluginaddons` inside the JAR file.
Any JAR file containing a plug-in must be placed into the application directory of SimplyHTML. The application directory of SimplyHTML is

1. the directory in the file system, where the JAR file of SimplyHTML is installed or
2. the root package directory of file `PluginManager.class`, if SimplyHTML is not operated out of a JAR file

Removing plug-ins

To remove a plug-in from SimplyHTML, remove its JAR file from the application directory and restart SimplyHTML.

Examples for plug-in installation

In the following examples it is assumed that a plug-in is to be added to SimplyHTML from a file MyPlugIn.jar

Example 1

SimplyHTML is operated out of file `SimplyHTML.jar` and `SimplyHTML.jar` is in directory `C.\Programs\SimplyHTML\`
Installation: File `MyPlugIn.jar` must be placed into directory `C:\Programs\SimplyHTML\`

Example 2

SimplyHTML is operated as an uncompressed class file not residing in a JAR file, the SimplyHTML package is installed in `C:\Programs\SimplyHTML\classes\`, i.e. file `PluginManager.class` is installed in `C:\Programs\SimplyHTML\classes\com\lightdev\app\shtm\plugin\`
Installation: File `MyPlugIn.jar` must be placed into directory `C:\Programs\SimplyHTML\classes\`

Example: Making a new plug-in

To create a new plug-in for SimplyHTML,

1. copy file `PluginTemplate.java` into your plug-in project's source directoy
2. rename `PluginTemplate.java` to a name you would like to give your new plug-in class
3. change the paramteters of call `super` in `PluginTemplate` 's constructor according to names you choose (refer to class `AbstractPlugin` for a definition of these parameters)
4. create one or more `.properties` files as needed for your new plug-in (you can use file `PluginTemplate.properties` a an example)
5. adjust parameter `"pluginTemplateLabel"` in method `getGUIName` to an ID String referring to respective entry in your `.properties` file(s)
6. add methods and fields to implement the functionality desired for your new plug-in
7. compile the new plug-in's source file (`.java`) to a `.class` file
8. place `.class` file and `.properties` file(s) into a Java archive (JAR) file, package `com.lightdev.app.shtm.plugin.installed`
9. place the JAR file into the directory `SimplyHTML.jar` is installed in

When starting SimplyHTML for the next time, your new plug-in should be listed in the 'Manage Plugins...' dialog.

Changing plug-in settings individually

In addition to its installation each plug-in can be configured individually per user. With class `PluginManagerDialog` all loaded plug-ins are displayed and can be configured. `PluginManagerDialog` is shown with the help of class `ManagePluginsAction` which is registered with respective menu item of SimplyHTML's plug-in menu.
`ManagePluginDialog` is the GUI representation of plug-in manipulation methods actually provided by class `AbstractPlugin`. It uses the methods each plug-in class has to provide through implementing interface `SHTMLPlugin` to display and change settings such as whether or not the plug-in is active or where it shall dock. Class `AbstractPlugin` provides an implementation of interface `SHTMLPlugin` which persistently stores the settings made in `MangagePluginDialog` automatically. To do so, an additional class `Prefs` is used, which is introduced in this stage 5 of SimplyHTML (see below).

Class Prefs

Class `Prefs` provides a simple approach to store user settings persistently. It maintains a `Hastable` of key/value pairs through a set of getter/setter methods. Whenever the `Hashtable` is changed, it is serialized to a file. Upon construction of class `Prefs` the serialized `Hashtable` is read from disk. If none is found a new and empty one is created. The preferences file created by class `Prefs` is named `SimplyHTML.prf` and is stored in the directory pointed to by expression `System.getProperty("user.home")` which usually references the home directory of the user currently logged in.
By using the home directory of the user preferences can be stored individually per user.

Stage 6: Lists

The Java language already includes some standard actions to implement list formatting with documents in a JEditorPane. These actions however only provide a very basic approach to working with lists. On top of just starting a new empty list and to enter text into this new list, for application SimplyHTML the requirement is to offer a simple way of switching list formatting on or off for an arbitrary portion of existing text paragraphs. As well there should be a way to change formatting of an existing list partly or completely through a separate dialog.
This stage implements list handling as described above covering the following topics
Lists in HTML documents
Implementing list formatting in SimplyHTML
Switching list formatting on or off
Creating a list format dialog
Adding actions and GUI elements

Lists in HTML documents

In HTML documents content is embedded into tags such as <p> or <td> etc. To apply list formatting to a set of paragraphs, they have to be enclosed into list item tags (
) which in turn are enclosed by list tags for ordered or unordered lists (and).
For example, a list coded as

```
<ul>
<li>
<p>
Item 1
</p>
</li>
<li>
<p>
Item 2
</p>
</li>
</ul>
```

would be rendered as

- Item 1
- Item 2

Certain attributes can be applied to above HTML code by either storing them directly with a tag inside the HTML document or by defining styles for respective tag in a CSS style sheet.
Read on to see how above list formatting is applied with application SimplyHTML.

Implementing list formatting in SimplyHTML

As described in the introduction of stage 6, to apply list formatting in SimplyHTML the requirement is to offer a simple way to switch list formatting on or off for an arbitrary portion of existing text paragraphs. As well there should be a way to change formatting of an existing list partly or completely through a separate dialog.
While parts of previous stages of SimplyHTML can be re-used to achieve list formatting thorugh a dialog, a certain difficulty is to change an arbitrary portion of existing text to list formatting because the element structure of the existing document content has to be changed.

Changing the element structure

The only way to change the element structure of an HTML document which is publicly available in the Java classes is to insert HTML code replacing a given part of respective document.
SimplyHTML uses class `SHTMLWriter` to synthesize HTML code as already done in table formatting. The process is described in more detail in the following chapters. Please see the source code of stage 6 for additional details.

Switching list formatting on or off

When an HTML document initially is filled with content by typing text, the content is not formatted as list. To reach list formatting for parts of a document the user could

1. start list formatting and then type content in the form of list items as needed or
2. type in content and then switch on list formatting for the recently typed paragraphs

While the first case is comparably easy to achieve, both cases require a simple toggle functionality to switch list formatting on or off.

Basic approach

Such list toggle functionality basically this is achieved by

1. detecting whether or not the required list formatting is present for a given text portion
2. if list formatting is present, switch list formatting **off**
3. if list formatting is not present (or not present as required), switch list formatting **on**

Method toggleList

Above scheme is implemented with method `toggleList` in class `SHTMLEditorPane`. Method `toggleList` finds out the parent element of a selected text portion. It then uses method `switchOn` to determine whether or not list formatting is already present in the selection inside the parent element. If method `switchOn` 'decides' to switch on list formatting (returns `true` that is), method `toggleList` uses method `listOn` to switch on list formatting. Otherwise list formatting is turned off by calling method `listOff`.

Difficulties

Up to here the solution sounds rather simple. In detail however, some difficulty is contained in the way how existing list formatting is to be changed. There are two cases we need to look at in more detail:

1. list formatting is to be switched off for only parts of an existing list
2. list formatting is to be switched on for one or more lists having mixed list formatting

Splitting lists

The first case above requires to split an existing list into up to three sections: The list remaining at the beginning of the selection, the actual selection for which list formatting is to be switched off and the list part possibly following the selection.
Method `listOff` splits a list by iterating through all list items in three steps. In the first step, it generates HTML for the list portion remaining unchanged and creates a new list end tag at the start of the selection to mark the start of the list split.
Secondly it continues iterating through the list items belonging to the selection generating HTML code with `<li>` tags removed.
Finally a new list start tag is created to mark the end of the split and iteration is continued over the remaining portion of the original list generating HTML code for the remaining list elements.

The resulting HTML code is inserted into the document replacing the 'old' part.

Merging lists

The second case above requires to merge different list formattings to one new list formatting. In addition, possible list formatting preceding or following the selection has to be split as described in the previous paragraph.
Method `listOn` merges lists by iterating through all list items and adjusts list start and end tags, merging and splitting lists as needed. `<li>` start and end tags are inserted if necessary.
The resulting HTML code is inserted into the document replacing the 'old' part.

Creating a list format dialog

Switching lists on and off as described in the previous chapter formats lists in their default formatting as defined in the style sheet of respective document (i.e. applies tags `<ul>` and `<ol>` without additional attributes). To adjust list formatting, additional functionality is needed.
To change list formatting, a dialog is created acting on both list attributes and list elements. When changing an existing ordered list from number to letter ordering for instance, attributes of the list are to be changed. When switching from an existing list ordered by numbers to an unordered bullet list with square bullet symbol, the list element itself *and* its attributes are to be changed in one step.

Re-use of existing parts

List formatting functionality in part is similar to what has been implemented for table formatting already. Consequently, some of the existing functionality of SimplyHTML can be re-used: Classes `DialogShell`, `AttributeComboBox` and `BoundariesPanel` which share common classes to work with attributes and attribute sets themselves. In stage 6 class `AttributeComboBox` has been turned into an own class from the former inner class in class `TableDialog`.

New parts to implement

To create the new list format dialog, class `ListDialog` is created extending class `DialogShell`. Class `ListDialog` is a container for the class showing the actual list attributes, new class `ListPanel`. Class `ListPanel` in turn uses classes `AttributeComboBox` and `BoundariesPanel` to make available respective list attributes.
How the list attributes actually are shown and changed is implemented exactly the same as in class TableDialog.
To apply list attributes as set with `ListDialog`, a new method `applyListAttributes` is added to class `SHTMLEditorPane` which is again similar to what applyTableAttributes does.

Adding actions and GUI elements

To finally use the new list functionality it has to be made available to the user by adding it to SimplyHTML's GUI. Three new actions are created in class `FrmMain`

- `ToggleBulletsAction`
- `ToggleNumbersAction`
- `FormatListAction`

They are added to `FrmMain's` `initActions` method and inlcuded into menu and tool bar definitions of the resource bundle of application SimplyHTML. Through SimyplHTML's dynamic resource mechanism, new menu items and tool bar buttons are created for above actions automatically.

Class `SHTMLEditorPane` has an additional action `NewListItemAction` which is registered with the key map of the editor pane (see also the explanations in the tables section). This action is used to create a new list item when the user presses the [Enter] key while the caret is inside a list.

Stage 7: Images

This stage is enabling application SimplyHTML to add images to documents and to adjust the display of images in documents. It implements an image repository and dialog for all image manipulation inside SimplyHTML.
The functionality for image support in SimplyHTML is described in the following chapters:

Image references in HTML

In HTML images are separated from documents. Documents contain references to image files in the place where images should appear inside a document. An image reference describes the location and name of respective image file as well as how it is to be rendered. The actual images are stored separately in image files and loaded dynamically when documents are displayed.
An image reference in HTML is expressed by an `img` tag having the reference attribute and additional attributes such as in

```
<img src="images/picture.jpg" border=0 width=100 height=200>
```

The image reference attribute (attribute `src`) can be either an absolute or relative path and file name expression. The other attributes specify information about how the image is to be displayed such as image width and height or how much space between text and image is to be rendered.

Absolute references

An absolute reference is an expression containig the full path and name of an image file such as in

```
file:/C:/data/documents/myDoc/images/picture.jpg
```

Absolute image references should be avoided as they need to be changed whenever the image file is moved to another location.

Relative references

A relative reference has path and file name information relative to the location a HTML document is stored. An expression such as

```
images/picture.jpg
```

inside a document stored as

```
c:\data\document\myDoc\doc.htm
```

would mean the same as the absolute reference expression from previous paragraph. The main difference however is that whenever the document and image file are moved to a new location together, the relative reference can remain unchanged.

Resolving relative references

In SimplyHTML's implementation of image support only relative references are used. To resolve relative image references, class `HTMLDocument` allows to specify a base location with method `setBase`. When a document is loaded, its source path is passed to method `setBase` as the the base directory. When a new document is created, a temporary directory is set with method `setBase`.
In both cases all relative references are resolved against the base directory set with method `setBase`.

General concept for image support

In HTML images are separated from documents. Documents contain references to image files in the place where images should appear inside a document. An image reference describes the location and name of respective image file as well as how it is to be rendered. The actual images are stored separately in image files and loaded dynamically when documents are displayed. SimplyHTML supports image references by maintaining an image repository for each document. An image repository in this context is a directory containing all images referenced by a HTML document.

Restrictions

To keep image maintenance simple, the following restrictions are used in SimplyHTML

- image files referenced from HTML documents are automatically placed into directory `images`
- directory `images` is created automatically inside the directory, a HTML file is saved

SimplyHTML has no support for creation or manipulation of image files as in image editing software . Image files need to be present already to be added to documents created or maintained with SimplyHTML.

Supported image formats

SimplyHTML supports the following image file formats

- Graphics Interchange Format (GIF)
- Joint Photographic Expert Group (JPEG) format

Temporary storage

To allow image processing for newly created documents (i.e. documents not having been saved at the time images are added) a temporary directory is maintained. The temporary storage is maintained automatically by SimplyHTML in directory
`[user home]/SimplyHTML/temp/`.
`[user home]` in this context is the directory returned by the Java expression `System.getProperty("user.home")`. It is usually the directory where a user logged in to a given system has all rights and where no other user except for system administrators has access rights unless explicitly granted by the owner or system administrator.
Using directory `[user home]` has the effect that every user has an own temporary storage area.

Images in new documents

For each newly created document a directory is created inside the temporary directory named after the document,
e.g. `[user home]/SimplyHTML/temp/Untitled 1/`.

If images are added to a document which has not been saved so far, directory images is created inside the temporary directory,
e.g. `[user home]/SimplyHTML/temp/Untitled 1/images/`.
Once a new document is saved, the image directory is copied from the temporary storage to the directory the new document has been saved.

Images in existing documents

If images are added to an existing document, respective image files are stored in directory `images` inside the directory the document was loaded from. Directory `images` is created in the directory the document was loaded from by application SimplyHTML when it is not already present an images are added to that document.

Implementing image storage

To enable image reference manipulation functionality as described in the next chapter, maintenance of an image repository is required. The following parts have been created to support image repository maintenance:

Class, Method, Field	Description
FrmMain.appTempDir	new field referencing the temporary directory of application SimplyHTML
FrmMain.initAppTempDir()	method to initialize the temporary directory of application SimplyHTML
FrmMain.getAppTempDir()	method to get the directory for temporary storage of application SimplyHTML
DocumentPane.getImageDir()	method to get the image directory for a particular document open in SimplyHTML
DocumentPane.saveImages()	method to save images at a new location
Util.copyFile()	method to copy files

Generally speaking, an image repository always is kept with the document it belongs to. Whenever a document is saved, the image directory of the document is saved at the same location, necessarily copying image files as described below.

Methods initAppTempDir and getAppTempDir

With method `initAppTempDir` of class `FrmMain` new field `appTempDir` is initialized. The method creates a file object referencing a directory named `[user home]/SimplyHTML/temp`. If the directory does not exist it is created by method `initAppTempDir`.
Method `getAppTempDir` publicly makes available field `appTempDir` for read only access.

Method saveImages

In class `DocumentPane` documents are being saved with method `saveDocument`. With the new image support of stage 7 of SImplyHTML document storage has to be extended by a method to save any image files referenced in a particular document.
New method `saveImages` of class `DocumentPane` is called by method `saveDocument` for this task. It uses new method `getImageDir` (see below) to find out the source location of any image

files. It then copies all image files to the location, the document is being saved to using method `copyFile` of class `Util` (see below).

Method getImageDir

Method `getImageDir` finds out the source location of image files for a document to be saved. The method checks whether images are currently stored in a temporary directory for a given document. If the document was not newly created, `getImageDir` finds out if the document is about to be saved at a new location (save as) or if it is being saved at the location it was loaded from (save) in which cases the source locations are to be taken from different locations in class `DocumentPane` .
The source image directory is returned to the calling method.

Method copyFile

Method `copyFile` in class `Util` is a simple way to copy a file from one location to another. It accepts two file objects as parameters for the source and destination file to be copied. It opens `RandomAccessFile` objects for the two files and creates the destination file if necessary. It then reads blocks of content bytes from the source file and writes them to the destination file.
If the destination file already exists, `copyFile` does nothing.

Creating a GUI to manipulate image references

As with table and list support, to create and manipulate image references a graphical user interface (GUI) is needed. Class `ImageDialog` is created for this purpose.

Class ImageDialog

Class `ImageDialog` is used to insert image references and to set all relevant attributes for these references. As well it provides a repository from which images can be selected, added and removed. An `ImageDialog` is divided into three panels from left to right. The left panel has a list which shows all files present in the image directory of the given document as well as buttons to add and remove images from the list. The middle panel is used as a preview region for any image selected from the image list. In the right panel all attributes of a selected image are shown and can be set.

Image list maintenance

When an `ImageDialog` is created, it is passed a directory which is to be used as the image repository. The image list is filled with the names of all files found in this directory by calls to method `updateFileList` (the method has only one line setting the `JList` content to the the current result of a directory listing of the directory referenced by `imgDir`).

Method handleAddImage

When button 'Add' is pressed, a file chooser is opened to allow selection of an image file to be placed into the repository. If a file is picked in the file chooser it is copied to the image repository with the help of method Util.copyFile and method `updateFileList` is called to reflect the new file in the file list.

Method handleDeleteImage

When button 'Delete' is pressed, an option dialog asks the user whether or not to delete the image file currently selected in the image list (if any). If the user chooses to really delete the selected file, it is deleted and the image list and preview are updated accordingly.

Image attribute manipulation

Once an image is selected from the list of images, all attributes of the selected image are displayed in the panel on the right of an `ImageDialog`. From there all attributes of the image can be set accordingly. Changes to image attributes such as size or scale are reflected in the preview immediately. In addition, attributes such as border width or distance to the surrounding text can be set and will be effective on the image in the document once applied (see below).
In class `ImageDialog` a set of listeners is used to synchronize all parts of the dialog to user changes of particular attributes. Section 'event handling' in the source code of class `ImageDialog` has the mentioned listeners which are applied to respective components in the

constructor of the dialog. Each listener calls helper methods such as `applyPreviewHeight`, `applyPreviewWidth` or `applyPreviewScale` in case an event occurs which a listener is bound to.

Returning image reference and image attributes

Once an image is selected and all attributes settings meet the desired display in the document, method `getImageHTML` returns the HTML code representing an image reference with all attributes according to the selection in the `ImageDialog`. Method `getImageHTML` uses class `SHTMLWriter` to create an image tag and attributes from the settings on the `ImageDialog`.
The components on the `ImageDialog` used for setting image attributes are implementing the `AttributeComponent` interface so each of them returns its value in an `AttributeSet` object. All such settings are brought together in an instance of `SimpleAttributeSet` and passed to method `startTag` of class `SHTMLWriter` along with the actual image reference returned by method `getImageSrc` .

Setting an initial image reference and attributes

Besides creating new image references class `ImageDialog` can be used to display and manipulate an existing image reference too. The same functionality is used as described above after the `ImageDialog` has been set to an existing image reference with method `setImageAttributes`.
Method `setImageAttributes` iterates through the `Vector` of `AttributeComponents` and applies attributes from a given `AttributeSet` to the components. As well it sets the `ImagePreview` to the image reference found in the `AttributeSet`.

Image preview

When an image is selected from the list of images or when attributes af a selected image are changed, the resulting image as it would appear in the document is shown in the preview section of class `ImageDialog`. The preview is produced by class `ImagePreview` which is an extension to class `JComponent`.
Class `ImagePreview` takes care of displaying any image and has methods to apply a given scale percentage to that image. It implements the `Scrollable` interface so it can be embedded in a `JScrollPane` for cases where an image is to be viewed in a region being smaller than respective image.
You can refer to the source code of `ImagePreview` for more details about how the preview of images is achieved.

Making the GUI available to the user

As functionality and GUI for manipulation of image references as described in previous chapters is present, a way to use it is needed in addition.

Actions InsertImage and FormatImage

Similar to the procedure used in previous stages, two actions are added to class `FrmMain` as inner classes. The new actions are added to the `DynamicResource` instance of class `FrmMain` with method `initActions` (see the documentation of stage 2 and stage 5 for a detailed description of actions and dynamic resources).
Actions `InsertImageAction` and `FormatImageAction` both create an instance of class ImageDialog. `FormatImageAction` shows the dialog reflecting settings for an image currently selected in the editor to allow for attribute adjustments or to change the image file.
`InsertImageAction` brings up the dialog to select a file from the image repository and adjust attributes.

How image settings are applied

When a selection is made in class `ImageDialog` `InsertImageAction` applies the settings with the help of method `insertBeforeStart` of class `HTMLDocument`. The chosen image from class `ImageDialog` is taken as HTML code got from method `getImageHTML` and passed to method `insertBeforeStart`.
When an existing image reference is changed with `FormatImageAction`, method `getImageHTML` is used to get the image settings as HTML code again. The HTML code is passed to method `setOuterHTML` of class `HTMLDocument` in this case, replacing the changed image reference.

Stage 8: Paragraph styles and named styles

Paragraph styles

In stage 3 of SimplyHTML font manipulation was added to the application. Font settings are applied to individual parts of a document down to single characters with this functionality. With paragraph styles such settings can be applied to one or more paragraphs in one step too. As well paragraph styles allow to manipulate additional attributes such as alignment or margins.

Named styles

Named styles in turn are an elegant way to define styles that are frequently used and store them in a separate style sheet. Usage of style sheets was already part of application SimplyHTML since the first stage however this stage finally adds functionality to use style sheets to their original purpose.

Contents of this stage

Implementation of both paragraph styles and named styles is the subject of this stage and explained in detail in the follwoing chapters

- Styles in HTML and CSS
- Parts needed to implement style manipulation
- Approach to work with paragraph and named styles
- Class ParaStyleDialog
- Class StyleSelector
- Interaction between style components and style sheet
- Adding the new style components to the GUI
- Style sheet storage

Styles in HTML and CSS

Character attributes vs. paragraph attributes

The simplest way to apply a certain format to a portion of a HTML document is to store HTML format attributes such as `b`, `i` or `align` with any tag to be formatted in the particular way. While this approach is most flexible in terms of combination of such attributes, plain HTML attributes allow only limited formatting compared to CSS styles. As well this method adds a lot of formatting information to the plain content of a document with much rendundancy in most cases.

CSS attributes

By using the `style` attribute, CSS attributes such as `margin-top`, `padding-right`, `text-align`, etc. can be added to HTML tags instead. This method opens additional formatting settings but it still requires attributes to be stored with each tag having the same impact as HTML attributes.

Paragraph styles

To reduce the formatting overhead HTML and CSS attributes can be applied to paragraph tags so that they are valid on any tag contained in such paragraph.

Style sheets with named styles

The most flexibility and power however is reached with usage of Cascading Style Sheets (CSS) in combination with a given HTML document. By defining styles in a separate style sheet groups of format attributes can be held independent from HTML documents resulting in significant advantages

- styles are defined only once avoiding rendundancies and increasing maintainability
- styles can be shared over many documents again reducing redundancy and maintenance efforts
- documents only need references to styles reducing storage space
- a predefined group of attributes can be applied in a single making formatting faster

Parts needed to implement style manipulation

While many existing functions of SimplyHTML and the Java classes can be used to build the new style setting functionality, some additional parts are needed too. The following table gives an overview of all new or changed items

Class	Purpose, Changes
AttributePanel	Panel to set a group of attributes, base class for other classes such as margin or style panel
CSSWriter	Enhanced method structure for writing individual styles
DocumentPane	additional methods for style sheet storage and merging style sheets
FrmMain	new actions for paragraph and named style formatting as well as new tool bar component for setting named styles, some methods and inner classes consolidated to avoid redundancies
MarginPanel	new class to set margins and paddings, made stand alone class from former inner class to share functionality between table and paragraph dialog
ParaStyleDialog	dialog for setting either paragraph styles or named styles
StylePanel	new class to set paragraph attributes, made stand alone class from former inner class to share functionality between table and paragraph dialog
StyleSelector	component to apply named styles through the tool bar
Util	utility methods for working with internationalized option panes, resolving nested attribute sets and style sheets

Mostly GUI changes

Functionality to read, modify and apply attributes has already been created in previous stages and can be re-used in this stage unchanged. Working with named styles and style sheets is covered by class `StyleSheet` of the Java Swing package in addition.
Therefore above parts almost all are GUI elements. Some 'non-GUI' methods and changes had to be added in this stage mainly to classes `CSSWriter` and `Util` and the only other 'non-GUI' method `saveStyleAs` in class `ParaStyleDialog` was too small to create an extension to class

`StyleSheet` for it.
In essence this stage mainly adds GUI extensions and relies on exisiting functionality of previous stages and the Java classes to implement style manipulation.

Much interaction

Nevertheless a lot of interaction between the mentioned parts is necessary so that an emphasis in this stage of the tutorial lies on explaining these interactions and their implementation as well.

Approach to work with paragraph and named styles

As pointed out, style manipulation functionality is already exisiting in the Java classes and the previous stages of application SimplyHTML. What is needed in addition is a GUI to visualize the existing styles and to let the user add new, change existing or delete styles.

Dialog to manipulate styles

Class ParaStyleDialog is added to SimplyHTML as a new component to achieve this. Class `ParaStyleDialog` has two major functions:

- to manipulate any given paragraph style and
- to maintain all paragraph styles in a given style sheet.

Because both tasks require the same elements (components to reflect settings of paragraph attributes), they are combined inside class `ParaStyleDialog` and made available as two separate operation modes of the component.

Components to select named styles and alignment

To make available existing named styles for paragraphs as defined in the style sheet, class StyleSelector is created. It shows all available named paragraph styles and lets the user apply a given style to the currently selected paragraph(s).
A paragraph attribute which is used quite often is the text alignment setting (left, center or right). For setting text alignmen in one step, inner class `ToggleFontAction` has been changed to class `ToggleAction` and can now can be used as a generic action to toggle certain character and paragraph attributes including text aligment. The new action is used by SimplyHTML's dynamic resource mechanism to add respective toggle button components to the tool bar.

Style sheet storage

As stage 8 allows to change contents of a given style sheet, the way a style sheet is saved has to be revised. When a document is saved, SimplyHTML now recogniszes whether or not a style sheet with the same name exists in the location where a document is to be saved.
Is a style sheet present already, it is merged with the style sheet to be saved.

Class ParaStyleDialog

Class `ParaStyleDialog` has two major functions:

- to manipulate any given paragraph style and
- to maintain all paragraph styles in a given style sheet.

The two tasks are available as two separate operation modes of the component, `MODE_PARAGRAPH_STYLE` and `MODE_NAMED_STYLES`.
In `MODE_PARAGRAPH_STYLE` class `ParaStyleDialog` is used to directly manipulate a given set of paragraph style attributes. In `MODE_NAMED_STYLES`, the dialog is used to manipulate styles in a style sheet which does not affect formats of the underlying document directly (only indirect through style sheet changes).

Setting the operation mode

The operation mode is derived from the constructor used to create a `ParaStyleDialog`. When constructed to operate with a certain `Document`, `MODE_NAMED_STYLES` is assumed and the `Document's` style sheet is taken to be operated upon.
If no `Document` is passed to the constructor of `ParaStyleDialog`, it is constructed in `MODE_PARAGRAPH_STYLE`, i.e. not using a style sheet.

Passing initial dialog settings

Class `ParaStyleDialog` implements interface AttributeComponent (introduced as FontComponent initially in stage 3) so that its contents can be set or read through a set of attributes in an `AttributeSet` object. When in `MODE_PARAGRAPH_STYLE`, initial dialog contents need to be set by a call to method `setValue` passing an `AttributeSet` object having all initial paragraph styles to be manipulated.
When in `MODE_NAMED_STYLES` , a list of existing named paragraph styles is read from the style sheet of the `Document` passed in the constructor. Whenever a style is picked from those, the dialog is set to show the attributes of this style.

Reading dialog settings

As an `AttributeComponent` class `ParaStyleDialog` returns its currents attribute settings in an `AttributeSet` object through method `getValue`. In `MODE_PARAGRAPH_STYLE` method `getValue` can be used to get the set of attrbutes to be applied.
In `MODE_NAMED_STYLES` class `ParaStyleDialog` is not meant to deliver a set of attribute settings, although method `setValue` certainly can be used too. Instead, the dialog only makes available all named paragraph styles found in a given style sheet.
All changes to a given set of paragraph attributes can be saved to that style sheet using class `ParaStyleDialog`. By changing attribute settings of an existing named paragraph style and storing them back to the style sheet, format of all paragraphs using respective named style is

changed implicitly, immediately and automatically in the underlying document.
Thus no direct reading of attribute settings is necessary in `MODE_NAMED_STYLES`.

Style sheet manipulation

In `MODE_NAMED_STYLES` class `ParaStyleDialog` offers to

- save settings to an existing named style
- create a new named style and
- to remove an exsting named style from the style sheet

Saving attributes to an existing style and creation of a new style both is done using method `addRule` of class `StyleSheet`. This method expects a style to be passed in the form of a CSS declaration string,
e.g. `p.myStyle { text-align:center; }`.
To transform attribute settings from class `ParaStyleDialog` in to this format, method `writeRule` of class `CSSWriter` is used.To remove an existing style from the style sheet class `ParaStyleDialog` uses method `removeStyle` of class `StyleSheet`.
Class `ParaStyleDialog` adds methods necessary to interact with the user upon style changes accordingly, e.g. by asking whether or not to really delete a particular style or by checking whether or not a style shall be overwritten having the same name as a name entered by the user.
With stage 8 class `Util` has some new methods combining a generic `JOptionPane` with calls to SimplyHTML's class DynamicResource for support of messages in other languages. These methods are applied to other usages of `JOptionPane` in SimplyHTML as well.

Class StyleSelector

Class `StyleSelector` is a component used to make available a list of existing named paragraph styles and to apply a named style to the currently selected parapraph(s). It extends class `JComboBox` by implementing interfaces `AttributeComponent` and `ChangeListener`.
Added to a tool bar its method `getValue` allows access to the currently set style while method `setValue` can be used to reflect the style of the paragraph the caret is currently in.
As desribed in more detail in the next chapter, class `StyleSelector` listens to changes in the `JTabbedPane` of class `FrmMain` and to changes of a given style sheet. Whenever the `JTabbedPane` or the style sheet changes (i.e. another document is active or the styles have changed), the list of named paragraph styles in the `StyleSelector` is updated.

Interaction between style components and style sheet

As described previously , several components are involved in paragraph and named styles manipulation:

- the style sheet of the document currently edited,
- the `JTabbedPane` of class `FrmMain` ,
- the `StyleSelector` in the tool bar,
- the list of named styles in class `ParaStyleDialog`.

Components reflecting named styles have to be updated accordingly when

- a new document is created,
- an existing document is opened,
- another document in the group of currently open documents is activated or
- the style sheet of the currently edited document changes.

Listeners to watch for changes

Instead of implementing hard wired relations between objects to handle style related events, application SimplyHTML implements listeners with these objects.

StyleSelector

Class `StyleSelector` implements the `ChangeListener` interface to handle `ChangeEvents`. The `StyleSelector` object in the tool bar is registered as a `ChangeListener` with the style sheet of any document with method `registerDocument` of class `FrmMain`. With that the `StyleSelector` object is notified whenever a style sheet changes. When a document is closed, class `StyleSelector` is removed as a `ChangeListener` in method `unregisterDocument` of class `FrmMain`.
In method `createToolBar` of class `FrmMain` class `StyleSelector` is registered with `FrmMain's JTabbedPane` as `ChangeListener` too. Whenever another document is activated in the JTabbedPane, the `StyleSelector` object in the tool bar is notified.

ParaStyleDialog

Class `ParaStyleDialog` is also implementing the `ChangeListener` interface. It registers itself as a `ChangeListener` with the style sheet of the currently active document. Whenever class `ParaStyleDialog` is used in `MODE_NAMED_STYLE` and a style is saved to the style sheet, the respective change event triggers an update of class `ParaStyleDialog's` list of named styles. Class `ParaStyleDialog` overrides method `dispose` to remove itself from the list of `ChangeListeners` of the underlying style sheet.

Adding the new style components to the GUI

As in previous stages actions are used to connect the new functionality to the GUI elements such as menus or the tool bar. There are three new actions and one changed action in class FrmMain

- `FormatParaAction` - action to set paragraph styles
- `EditNamedStyleAction` - action to manipulate named styles
- `SetStyleAction` - action to apply a given named style to the current selection
- `ToggleAction` - adaption of former ToggleFontAction to support paragraph alignment too

The actions are initialized in method `initActions` of class `FrmMain` and are added to `FrmMain's` dynamic resource.

FormatParaAction and EditNamedStyleAction

`FormatParaAction` and `EditNamedStyleAction` both create an instance of ParaStyleDialog . `FormatParaAction` uses the set of attributes returned by `ParaStyleDialog` and applies them to the currently selected paragraph(s). `EditNamedStyleAction` does nothing on return of `ParastyleDialog`, the dialog does all the work for manipulation of named styles.

SetStyleAction

SetStyleAction takes the class attribute returned by the StyleSelector component in the tool bar and applies it to the currently selected paragraph(s).

ToggleAction

ToggleAction applies the attribute returned by its getValue method to the current selection. Depending on the attribute key the action object represents attriubtes are applied to either on paragraph or character level. The action then sets a state indicator to allow a bound component such as a JToggleButton to reflect the state accordingly (selected or unselected).

New components

Besides the new `StyleSelector` component, in method `createToolBar` of class `FrmMain`, additional toggle buttons are created to toggle between different paragrpah alignments (left, center, right). A new tool bar button for setting a paragraph style is added too. In menu 'Format', new menu items are created for setting paragraph style, style sheet manipulation and paragraph alignment.
All new components are added by including their associated action names in the properties file of SimplyHTML as described in stage 2 and especially 'Adding an edit menu' as well as 'Dynamic resources' from stage 5.

Style sheet storage

Style sheets are part of SimplyHTML since stage 1 of the application. Since then they were only saved along with a document with a set of static styles. With stage 8 of SimplyHTML manipulation of named styles is supported so that the original style sheet handling needs to be extended.

Loading style sheets

Class DocumentPane now has two ways of creating a document with respect to style sheets. A new document is created with the underlying EditorKit creating a default style sheet from the resources package of SimplyHTML (as done in previous stages).
This is not longer done when an existing document is loaded. In such case the underlying EditorKit creates a default document without a default style sheet. Class DocumentPane instead looks for the style sheet reference inside this document and loads this style sheet for the particular document instead.
The EditorKit not longer shares a single style sheet among different documents, each document has associated its own style sheet.

Saving style sheets

When a style sheet is saved, four cases are now handled

1. no styles are present at save location, create new style sheet
2. the style sheet was loaded from somewhere else and now is being saved at a new location where a style sheet exists havig the same name
3. the style sheet is saved at the same location where it was loaded from
4. the style sheet was newly created and now is being saved at a location where a style sheet exists havig the same name

In case 2 and 4 above, the style sheets are merged overwriting existing styles in the found style sheet with styles from the saved style sheet. Styles from the found style sheet not existing in the saved style sheet are kept in the newly saved style sheet.
In case 3 above the existing style sheet is overwritten by the new version.

Tradeoffs

While above save strategy does not require user interaction other than to choose a save loaction and name for the respective document (as before) it still leaves the problem that an existing style sheet with the same name could have styles with the same name as altered ones in the saved style sheet. Overwriting such styles could cause unwanted styles to appear in other documents sharing the particular style sheet.
Therefore the user is obliged to either

1. not save documents in the same directory when they do not share the same set of named styles or
2. use different style names for different styles over all documents sharing the same style sheet

Stage 9: Links

In this stage creation and manipulation of links is implemented. While a link seems to be a rather trivial element in an HTML document, working with links without knowing how to to type in HTML code requires a quite complex GUI. This is because links can have many different forms with several exceptional handling of certain link types.
To allow for links referencing certain parts inside a given document, a way to edit so called link anchors is needed in addition.
Besides links and link anchors, stage 9 has some refined handling of paragraph tags and named styles from previous stage. And last but not least it compensates a somehow ugly effect of Java showing font sizes smaller than any web browser.
Read all about above topics in more detail in the following chapters

- Links in HTML
- New parts in this stage
- How to apply links
- Creating a GUI to define links
- Creating a GUI to define link anchors
- Using LinkDialog and AnchorDialog

Links in HTML

A link in HMTL is a reference to another location a user can jump to. The common syntax for a HTML link is

```
<a href="URI">link text</a>
```

Link types

Link types are reflected by the type of uniform resource identifier (URI) of attribute `href` . The following formats for an URI are possible

1. anchor in the same document (e.g. `#anchorname`)
2. other document (e.g. `myDoc.htm`)
3. other document in other directory (e.g. `../directory/myDoc.htm`)
4. anchor in other document (e.g. `myDoc.htm#anchorname`)
5. anchor in other document in other directoy (e.g. `../directory/myDoc.htm#anchorname`)
6. WWW address (`http://...`)
7. address to local document (`file://...`)
8. Gopher address (`gopher://...`)
9. FTP address (`ftp://...`)
10. Telnet address (`telnet://...`)
11. Newsgroup address (`news:...`)
12. E-Mail address (`mailto:name@domain.xy`)

Link types 1 to 5 above can be summarized as links with relative addresses (i.e. relative to the location the document containing the link is stored), all others work with absolute link addresses. Instead of a link text an image can be specified as well, for instance

```
<a href="home.htm">
   <img src="anImageFile.jpg" width="160" height="34" border="0">
</a>
```

SimplyHTML models all types of links and supports links of type 1 to 7 above.

Link anchors

As shown in link types 1, 4 and 5 above, links not only refer to other files, they can point to specific locations inside a given file too. To enable a link to point to a certain location inside a file, so called *link anchors* have to be defined as target locations in respective target files. Link anchors are defined by inserting HTML code such as

```
<a name="anchorname">link anchor text</a>
```

Link anchors can be defined with or without an anchor text, however SimplyHTML only allows to create link anchors with text.

New parts in this stage

As with previous stages several adjustments to existing classes as well as some new classes are needed to build the new link functionality

Class	Purpose, Changes
LinkDialog	Dialog to create and edit links
AnchorDialog	Dialog to create and edit anchor links
FrmMain	new actions for link creation and formatting
SHTMLEditorPane	new methods to apply and change links and anchor links
Util	new methods to build relative paths and to locate link elements

Addtional to working with links, stage 9 has some refined features for working with paragraph tags and named styles as well as for rendering HTML:

Class	Purpose, Changes
SHTMLEditorKit	support for additional views
SHTMLInlineView	new view compensating font size differences between Java and web browsers
SHTMLParagraphView	new view compensating font size differences between Java and web browsers
TagSelector	new component to select paragraph tag types from the tool bar
ParaStyleDialog	additional tag type selector to set named styles for tags other than paragraph
SHTMLEditorPane	new method to apply tag types to paragraph tags

Classes `SHTMLTableView` and `SHTMLBlockView` have been changed and moved to package com.lightdev.app.shtm. Class `LengthValue` has been abandoned and removed from the project.

How to apply links

In Java links inside HTML documents are represented different to other HTML tags. As previously described a link usually is denoted by a tag such as

```
<a href="URI">link text</a>
```

But instead of being a branch element of type `HTML.Tag.A` inside a `HTMLDocument`, a link is represented as an `AttributeSet` for a given content element. To complicate things a little, images are represented as an `AttributeSet` for a content element too, so image links in turn are represented as two `AttributeSets` each nested inside the `AttributeSet` of a given content element.

Applying links

To apply a text or image link, class `SHTMLEditorPane` has two new methods both called `setLink`. The two methods only differ in the parameters they expect. One of the methods just wraps the other one into a more convenient call with fewer parameters for text links. Method `setLink` determines, whether or not the selection currently is inside a link with the help of method `findLinkElementUp` of class `Util` (see below).

Text links and image links

If inside a link, this link is replaced by the new link. If the selection is not inside a link, the new link just is created at the current caret position. Possibly selected text is replaced by the text of the new link in this case.
After it has determined, whether the selection is inside a link, method `setLink` splits into calls to methods `setTextLink` and `setImageLink` respectivley, whatever applies from the parameters received from the calling method. If no image file is passed (`linkImage` is `null`), a text link is assumed and vice versa (`linkText` is `null` instead).

Method setTextLink

Method `setTextLink` takes the link reference and stores it as `HTML.Attribute.HREF` in a new `AttributeSet`. If a style name was passed, it is stored as `HTML.Attribute.CLASS` in the `AttributeSet` for the link. The new link then is applied to the selection depending on what is inside the selection.
If the selection contains a link, this link is replaced by the new link. Otherwise, the selected text is replaced by the new link text and the link attributes are applied to this new text.

Method setImageLink

Method `setImageLink` works similar to method `setTextLink`. The only difference is that it creates an additional AttributeSet for representing the image (file, width and height). This AttributeSet is applied instead of link text to the selection replacing any existing link or other text along with the new link attributes.

Method findLinkElementUp

To find a link element from the position of a given element upwards in the element strucutre of a document, the attribute sets of elements have to be inspected (not the element names). Method `findLinkElementUp` does this by iterating through parent elements of a given element and looking for an attribute with key `HTML.Tag.A`. If such an attribute is found, this attribute represents a nested `AttributeSet` . Method `findLinkElementUp` then looks for an attribute with key `HTML.Attribute.HREF` inside this nested `AttributeSet`. If `HTML.Attribute.HREF` is found, a content element with a link attached has been found and this element is returned.

Creating a GUI to define links

With methods to apply links as explained in previous chapter, a new dialog is required to allow for convenient link creation and manipulation. New class LinkDialog is the central place to allow for link entries of all kind.

Class LinkDialog

Class `LinkDialog` has a collection of components for all relevant link attributes. It can be used to set a new link or to view and manipulate an exisiting link.
As several dialogs of application SimplyHTML have been explained in earlier stages already, we only look at some additional 'specialties' of class `LinkDialog` here. `LinkDialog` establishes a central `ActionListener` to handle changes to any of its components. If the link type combo box changes for instance, the buttons to select a local file or link anchor are enabled or disabled accordingly.

Switching of relative and absolute paths

If the link type is set to 'relative' or to 'local', the link address is switched between absolute (`file:/C:/Data/aFile.htm`) or relative (`../../aDir/aFile.htm`) notation with the help of methods `resolveRelativePath` and `getRelativePath` of class `Util`.

Selection of local files and link anchors

A link type of '`local` ' allows to use a `JFileChooser` dialog to browse for a local file. Link types ' `local`' and '`relative` ' as well allow to define a link anchor with respective browse button.
Link address and link anchor can be typed directly into respective text fields too.

Image selection

By selecting 'show link as image', the dialog switches to display an image panel with a browse button to set an image from the repository. In the `LinkDialog` any selected image is only shown with the width and height selected in the `ImageDialog`. These settings can only be changed by using the `ImageDialog` through respective browse button.

Returning link settings

Once all link attributes are set with class `LinkDialog`, methods `getLinkText`, `getHref`, `getStyleName`, `getLinkImage` and `getLinkImageSize` can be used to find out the user settings.

Creating a GUI to define link anchors

Link anchors denote certain positions inside a document making it possible to link to that particular position directly. Link anchors are identified by a link anchor name which is internally stored in the document at the position it denotes. Link anchors typically are not visible in the document, they are present only in the HTML code making up such document.
to enable a user to create and manipulate link anchors requires some extra work because SimplyHTML hides HTML code with the intention to let an author concentrate on content rather than HTML coding.

Class AnchorDialog

With class AnchorDialog a GUI is built for making link anchors of a document visible and for creating or deleting link anchors. As several dialogs of application SimplyHTML have been explained in earlier stages already, we only look at some additional 'specialties' of class `AnchorDialog` here.

Document as object or file

`AnchorDialog` can be constructed either with a `Document` object or with a URL leading to a document. When constructed with a URL, `AnchorDialog` loads the `Document` found at that location. When link anchors are added to or removed from a document loaded from a URL, the changes are saved to the document before `AnchorDialog` is destroyed.

Method getAnchors

Class `AnchorDialog` displays a list of link anchors existing in the document associated to the dialog with the help of method `getAnchors`. Method `getAnchors` iterates through all elements of the document and looks for `HTML.Tag.A` in the attribute set of each element. If such attribute is found, `getAnchors` looks for `HTML.Attribute.NAME`. Any element having `HTML.Attribute.NAME` inside `HTML.Tag.A` is listed as a link anchor.
Whenever method `getAnchors` is called to find all anchors in a document, a `Hashtable anchorTable` is filled with the anchors found. `Hashtable anchorTable` references elements with anchor names. The `Hashtable` is used to fill the list of available anchors too.

Making anchors visible

As mentioned previously, anchors are only visible in the HTML code of a document. To make a link anchor visible a `Highlighter` is used in class `AnchorDialog`. A `ListSelectionListener` is implemented by class `AnchorDialog` and registered with the list of anchors. Whenever an item is selected in the list, the element the anchor name refers to is taken from the `anchorTable Hashtable`. The element is used to determine the anchor position in the document and the position is passed to the Highlighter to be shown.

Using LinkDialog and AnchorDialog

The two dialogs making up the new link functionality in this stage are made available through new actions in FrmMain.

- `InsertLinkAction` - action to create a new link
- `EditLinkAction` - action to edit an exisiting link
- `EditAnchorsAction` - action to create or remove anchor links

`InsertLinkAction` and `EditLinkAction` both create an instance of class `LinkDialog` to let the user create or work on a link. When the user has not cancelled upon return of the dialog, method `setLink` of class `SHTMLEditorPane` is called to apply the link settings from the `LinkDialog` object.
`EditAnchorsAction` is used similar to the above, the only difference is usage of method `insertAnchor` of class `SHTMLEditorPane` instead.

Usage of AnchorDialog through LinkDialog

Additional to using `AnchorDialog` directly on a document currently edited, the dialog can be used from out of class `LinkDialog`. The idea is that either

1. an anchor link is applied to a document in one step and a link is created to reference this anchor link in a second step later or
2. an anchor link is created in the course of creating a link both in one step.

To create a link to a newly created anchor link in one step, as in case 2. above, the anchor link needs to be created directly out of class `LinkDialog`. To do so, `LinkDialog` has a browse button next to the text field for typing in a link anchor name. When the browse button is pressed class `LinkDialog` creates an instance of `AnchorDialog` passing the URL currently entered as the link address.
`AnchorDialog` opens the document referenced by the URL received from `LinkDialog` and shows existing anchors of this document. The user now can choose an existing anchor link or create a new one which then is chosen.
Once a link anchor is selected, its name is taken back to the calling `LinkDialog` and the anchor name is included with the new link currently worked on in the `LinkDialog` object.

Stage 10: HTML code editor and syntax highlighting

SimplyHTML focuses on creation and manipulation of text documents. The fact that the documents are stored as HTML files along with cascading style sheets (CSS) was hidden from the user interface of SimplyHTML intentionally. The autor should not be forced to know or work with HTML code to write a text document.
On the other hand, for an experienced user being familiar with HTML it sometimes is quicker to manipulate a certain portion of HTML directly instead of having to wade through GUI elements. For this reason this stage of SimplyHTML implements a way to work on the HTML representation of any given text document.
The following parts are covered in this stage:

- HTML code editor: a simple approach
- how to add syntax highlighting
- how to integrate the new component

HTML code editor: a simple approach

Although SimplyHTML is mainly meant for text processing, sometimes it is useful to do a change directly in the HTML representation of a document. For this purpose a component to display and edit HTML code is required. The editor shall not replace a powerful web page HTML editor, it only adds basic HTML manipulation functions.
To implement such an editor an ordinary `JEditorPane` is used. Setting the content type to "text/plain" and adding the HTML code of a given document as content for the JEditorPane leads to have a fully working editor.

Obtaining the HTML code for a given text document

To get the HTML code for a given document which can be shown in above mentioned editor pane, class `HTMLWriter` (or `SHTMLWriter`, depending on the user selection) is used. The writer creates HTML code for any given instance of class `Document` and its subclasses. By using method `getEditorKit` of an `EditorPane` the `EditorKit` for a displayed document is taken. Method write of class `EditorKit` uses `HTMLWriter` implicitly.
See new method setHTMLView of class `DocumentPane` about how this approach is used.

Simple but not enough

While the above would already be enough to edit HTML for any given text document it is comparably hard to work with HTML in a plain text display. In plain text the structural elements of HTML are not visually separated from content elements. Thus, the next chapter explains how syntax highlighting is added to our new simple HTML editor for improved legibility.

Adding syntax highlighting

The previous chapter describes how a simple HTML code editor can be built. But with a plain text view structure and content of a HTML file is not visually separated. To improve legibility, syntax highlighting can be used: By displaying certain parts such as tags or attributes in a color or style different to the one used for content the reader can easily find certain parts of the document. There are different approaches possible to implement syntax highlighting. For SimplyHTML regular expressions are used for their simple way of defining patterns in a single expression.

Class SyntaxPane

A new class `SyntaxPane` is created as a subclass of `JEditorPane`. In the constructor of `SyntaxPane` method `setupPatterns` is called, which defines the patterns for HTML tags, attributes and attribute content. Method `setMarks` (see below) is used to apply syntax highlighting to a given part of the document in the `SyntaxPane`.
The `SyntaxPane` registers itself as a `CaretListener` and uses method `caretUpdate` to keep the syntax highlighting up to date for any changed text. When a document is shown initially, `setMarks` is called for the entire content (making it a lengthier process for bigger documents to display the highlighting initially). During changes only the highlighting of the current line is updated so that typing text is not slowed down too much.
A tradeoff with above approach is that multiline formats such as multiline comments are not handled with it.

Method setupPatterns

Method `setupPatterns` uses regular expressions to define a pattern for each element to be shown different from normal content. A HTML tag for instance is enclosed in < and > and can have letters and numbers with or without a slash inside those markers. An attribute ends with =, etc. For each `Pattern` an `AttributeSet` is created having the style to apply for that particular `Pattern` .
In method `setupPatterns` a `Vector` is used to hold pairs of one `Pattern` and one `AttributeSet` wrapped into inner class `RegExStyle`.

Inner class RegExStyle

Inner class `RegExStyle` is used as a convenience class to bundle a `Pattern` with a set of attributes. It simply has two class fields for the `Pattern` and the `AttributeSet` and respective getters and setters. All defined `RegExStyles` are stored in `Vector patterns` of class `SyntaxPane` .

Method setMarks

Method `setMarks` is the public member of `SyntaxPane` which is used to apply syntax highlighting to a given portion of the current document. Method `setMarks` creates an instance of

inner class `StyleUpdater` (see below) and calls `invokeLater` of class `SwingUtilities` to have styles updated without conflicts in the event dispatch thread.

Inner class StyleUpdater

Class `StyleUpdater` implements the `Runnable` interface by wrapping its functionality in a public method named `run`. Its main task is to apply styles associated with regular expression patterns to a given portion of the document which is currently edited.
This is done by iterating through `Vector patterns` of class `SyntaxPane` . For each `Pattern` found a `Matcher` is created. To all instances of the the `Pattern` found by the `Matcher` the style associated to the `Pattern` is applied.

Method caretUpdate

Method `caretUpdate` finds out the start and end position of the line the caret currently is in and calls method `setMarks` for this portion of text each time the caret position changes.

Recommended readings

'Regular Expressions and the JavaTM Programming Language' at
http://developer.java.sun.com/developer/technicalArticles/releases/1.4regex/
and
presentation slides 'Rich Clients for Web Services' from JavaOne 2002 at
http://servlet.java.sun.com/javaone/resources/content/sf2002/conf/sessions/pdfs/2274.pdf

Integrating the new component

Class `DocumentPane` is used as the GUI representation of a document in application SimplyHTML. To let the user switch between layout view and HTML view in stage 10 class `DocumentPane` has some additional parts:

- an editor pane to show and edit HTML code additional to the one used to show and edit the text and layout
- a `JTabbedPane` to hold two editor panes and to switch between the two
- a method to track the state of the new `JTabbedPane` and to react on state changes
- methods that handle transfer of content between the two editor panes

Initializing the two views

The `JTabbedPane` is initialized in the constructor of `DocumentPane` and a reference is kept in new class field `tpView`. The `JTabbedPane` is added to the center area of the content pane of class `DocumentPane`.

Adding two editor panes

A new class field `htmlEditor` of class `DocumentPane` references the new `SyntaxPane`. The field is initialized in the constructor of class `DocumentPane` with a new instance of class `SyntaxPane`.
The `SHTMLEditorPane` in class field `editor` and the `SyntaxPane` in class field `htmlEditor` are added to the `JTabbedPane` in the constructor of class `DocumentPane`. Now the two resulting tabs in the `DocumentPane` can be used to toggle display between layout view and HTML view.

Tracking tab clicks

Class `DocumentPane` implements interface `ChangeListener` by adding new method `stateChanged`. Class `DocumentPane` is added to the `JTabbedPane` as a `ChangeListener`. Method `stateChanged` is called by the `JTabbedPane` whenever its state changes (another tab has been clicked, that is).
Method `stateChanged` of class `DocumentPane` checks if the state of the `JTabbedPane` of class `DocumentPane` has changed. Every time the state of the `JTabbedPane` chages, the view associated to the clicked tab is opened through methods `setLayoutView` and `setHTMLView`.

Method setLayoutView

In method `setLayoutView` the current content of the `SyntaxPane` is taken (the HTML code) and transferred over to the `SHTMLEditorPane`. Because method `setLayoutView` is used when the HTML display is hidden and the layout display is shown, it removes the instance of `DocumentPane` as a `DocumentListener` from the `SyntaxPane` and adds it to the `SHTMLEditorPane` so that changes are tracked by `DocumentPane` accordingly.

Method setHTMLView

Method `setHTMLView` works the same as `setLayoutView` in the way that it removes and adds class `DocumentPane` as a `DocumentListener` accordingly. It takes contents of `SHTMLEditorPane` and adds them to the `SyntaxPane` too.
To see the HTML code instead of the textual representation of the document however, method `setHTMLView` transforms the document content, before storing the resulting HTML code in the `SyntaxPane`. It uses the `HTMLWriter` or `SHTMLWriter` of the editor kit of class `SHTMLEditorPane` to generate HTML code for the particular document. This HTML code then is set to be the initial content of the `SyntaxPane`.
Finally it calls method `setMarks` to apply syntax highlighting initially.

Stage 11: Find and replace

Java[tm] technology already offers generic functionality to find portions of text within a given string. This stage of SimplyHTML uses such functions to build a user interface suitable for most find and replace operations. In addition the new find and replace logic implements a way of replacing text over an arbitrary number of separate documents.

This chapter does not go into every detail of how to build respective user interface. It concentrates on aspects in conjunction with find and replace as shown in the following topics

Find and replace basics

Search direction and start position

The actual finding of a given text phrase is achieved with usage of methods `indexOf` and `lastIndexOf` of class `String`. Both methods return the position a given text phrase is found at inside another string. `indexOf` and `lastIndexOf` accept an optional position to start the search from so that the variations of these methods already can be taken to implement searching from start or from end of a document in either upward or downward direction.

Whole word search

Methods `indexOf` and `lastIndexOf` find any occurrence of a given search phrase as either part of a word or whole word. To restrict matches to whole words a character array of word separators `WORD_SEPARATORS` is used. Method `isSeparator` is used in method `doFind` of class FindReplaceDialog to determine whether or not a found occurrence is a full word.

Case sensitive search

Methods `indexOf` and `lastIndexOf` are case sensitive. They return a found occurence only when capitalization of letters matches the given search phrase. To do a case insensitive search method `toLowerCase` of class `String` is applied to both the search phrase and the text to search in before a find operation is initiated.
Read on in the next topics to find ouit more about how these basics are applied in the user interface and logic of SimplyHTML.

Find and replace user interface

To build the find and replace user interface in SimplyHTML the following set of classes is used that already existed from another project:

Class	Purpose
`FindReplaceDialog`	User interface and main functionality
`FindReplaceListener`	Listener to achieve search and replace over multiple documents
`FindReplaceEvent`	Event being thrown when a document search is finalized and another document would be needed for multi document find and replace

Above classes are taken from package `de.calcom.cclib.text` and encapsulate the complete logic and user interface needed to implement find and replace in a typical dialog. `FindReplaceDialog` only has been extended with internationalization supprt so that it can be used language independent inside SimplyHTML.

Usage of the dialog is very simple. Once added to SimplyHTML (or any other project) it is instanciated with a `JEditorPane` as a parameter. The `JEditorPane` is expected to have the document to perform search and replace upon. `FindReplaceDialog` then does all find and replace operations including document manipulation, user messages, state handling and optional multi document processing.

To control the state of the dialog `FindReplaceDialog` uses a flag to indicate the current operation in process. It can be one of

- OP_NONE
- OP_FIND and
- OP_REPLACE

See the following topics to learn more about how find and replace logic is implemented in these classes.

Find logic

In a design that separates functionality from user interface another class would have been needed for the logic such as `FindReplaceLogic` for instance. Instead the following methods of `FindReplaceDialog` have the main logic

Method	Purpose
`findNext`	find the next occurrence of a given phrase from start or end of a given document either in upwards or downwards direction.
`findWholeWords`	Find the next whole word occurrence of the searched phrase from a given position.
`isSeparator`	determine whether or not a character is a word separator with the help of character array `WORD_SEPARATORS`.

In addition methods `initFind`, `doFind` and `find` are used on top of the above methods to

- initiate a find process (`initFind`)
- manage multi document processing, if applicable (`find`) and
- handle results of `findNext` (`doFind`)

Above methods are called by `FindReplaceDialog` when either the 'find next' button is pressed or when the next occurrence of a phrase to be replaced is searched during a replace operation.

See the next topic to find out more about how the replace logic works.

Replace logic

Similar to the find logic described in previous topic the following methods of `FindReplaceDialog` have the main logic for replacing occurrences of a given phrase:

Method	Purpose
`replace`	Initiate a replace operation. If no (more) hits are found, a message is displayed and the dialog is unlocked for a new search operation.
`replaceOne`	Replace the currently selected occurrence of the search phrase.

By pressing button `jbtnReplace` a find operation is initiated with a call to method `initFind` and above methods are called with an initial replace option of `RO_YES`. Subsequent iterations through the replace process are driven by the user through a selection in method `getReplaceChoice` which is called each time an instance of the search phrase is found and which can be one of

- `RO_YES` - replace and find next
- `RO_NO` - do not replace and find next
- `RO_ALL` - replace all occurrences
- `RO_DONE` - exit replace process

Supporting find and replace over multiple documents

With SimplyHTML more than one document can be open at the same time. The find and replace logic introduced in stage 11 offers a way to apply find and replace to all open documents optionally. The way to apply find and replace to multiple documents can be customized in addition to account for different purposes such as applying find and replace to a set of documents inside a kind of 'project' as delivered by a possible SimplyHTML plug-in.

FindReplaceListener and FindReplaceEvent

To support find and replace over multiple documents interface `FindReplaceListener` and class `FindReplaceEvent` are used. An instance of class `FindReplaceListener` can be passed as a parameter during construction of a `FindReplaceDialog` . Passing a `FindReplaceListener` signals `FindReplaceDialog` that find and replace is to be performed over more than one document. `FindReplaceDialog` fires `FindReplaceEvents` to the given `FindReplaceListener` to signal that it is through with searching a particular document and that another document is required to continue.

Feedback during a multiple document process

The object registered as `FindReplaceListener` has to give feedback to `FindReplaceDialog` during multiple document operations. Methods `getFirstDocument` and `getNextDocument` have to call either `FindReplaceDialog.resumeOperation` or `FindReplaceDialog.terminateOperation` at their end, depending on whether or not there are documents left to process.

Using FindReplaceDialog in SimplyHTML

Application SimplyHTML uses a new action in class `FrmMain` to invoke `FindReplaceDialog`. Inner class `FindReplaceAction` instanciates a `FindReplaceDialog` in its `actionPerformed` method. Class `FindReplaceAction` implements interface `FindReplaceListener`. Whenever more than one document is open inside SimplyHTML `FindReplaceAction` passes itself as `FindReplaceListener` to the newly instanciated `FindReplaceDialog`. Methods `getFirstDocument`, `getNextDocument` and `findReplaceTerminated` in class `FindReplaceAction` hold functionality to implement find and replace for all documents currently open in SimplyHTML.

Spotlights

While most functions of SimplyHTML are explained from the perspective of the structure in the source code by explaining certain methods or classes, for some functionality it makes sense to view it from a rather process oriented perspective.
In this section, such cases are explained in the process context, wrapping together several functions located at different places but belonging to a particular process. As well topics are reflected which do belong to certain part or class of SimplyHTML but rather should be explained in more general context.

Avoiding loss of data in the close process

The process of closing one or more documents technically is easy to achieve. However, making sure that changes to a document are not lost when it is closed is a more complex task.

Step 1: Intercept all close actions

Because a close operation can be caused by different actions, it is important to take into account all occasions that would cause a document to close. So the first step to take is to ensure that each and every possible close action is intercepted by a check wheter or not it is ok to close that document or what requirements are bound to closing it.
In the design of SimplyHTML proper handling of close requests is ensured by having all methods to call the same action for closing a document: SHTMLFileCloseAction. Having all related functionality in a central place and having all other related methods to call that central functionality makes it easier to implement exactly the correct functionality and makes sure it is implemented only once.
The close actions to intercept are

- closing the main frame (method processWindowEvent)
- selecting 'Exit' from menu 'File' (SHTMLFileExitAction)
- selecting 'Close' from menu 'File' (SHTMLFileCloseAction)
- selecting 'Close all' from menu 'File' (SHTMLFileCloseAllAction)

Step 2: Ensure documents are closed only when conditions allow it

The second step is to ensure that a document is only closed when conditions allow to close it. Before it closes a document, SHTMLFileCloseAction tests in a central place, if changes are to be saved for that document first or if a save process currently is going on which finalization has to be waited for.
In this functionality the logic is placed to notify the user, that he is about to close a document which contains unsaved changes and to ask the user for a decision whether or not the changes should be saved before closing.

Step 3: Testing the result of the close action

In cases where an action has to follow the close action, such as when the application shall be terminated, the exit action needs to test if a document has been actually closed after it requested to close it. Otherwise, an application would terminate even if the user opted to cancel the operation during the close action (e.g. when asked to save the document first).

Using layouts for proper alignment of visible components

The Java language holds a powerful mechanism to properly align and size GUI elements within a frame with the `Layout` concept. Other than by stating absolute coordinates at design time of GUI elements, layouts define a display model relative to certain rules.

Examples in Class AboutBox

Class AboutBox of SimplyHTML is an example of how to apply the layout concept. GUI elements such as labels or images are placed onto panels. For each panel exactly one layout scheme is associated by which the panel controls positions and sizes of it's contents.
The panel `textPane` for instance uses a `GridLayout` with one column and six rows to arrange the labels contained in the panel one below the other with a gap of 5pt between each other. The `contentPane` of AboutBox uses a `BorderLayout` to align all other panels with a border of the contentPane. `buttonPane` sticks to the bottom edge of the `contentPane`, `northPane` to the top edge and so on.

Conclusion

By using layouts, the GUI elements are sized automatically to fit the resulting scheme. Most important, they all are resized according to the rules of respective layout when the container is resized. By defining layouts, the developer does not have to worry about how the components need to be sized and resized. Only their positions relative to each other and relative to their container need to be taken into account.
So the layout model rather follows the original intention of the developer rather than forcing him to transform the design intentions into coding models over and over again.

Using interfaces

Interfaces are a good way to define rules by which objects interact. If one object likes to communicate with another it has to have a way to determine whether or not that other object 'understands'. If an object can determine from which class another object was instantiated, it can expect or not expect certain methods being present.
If an object is to implement an interface it has to implement all methods the particular interface defines. How the methods are implemented, i.e. which code they actually hold, is up to the implementing object. A single object can implement many interfaces.

Example: SHTMLAction

In application SimplyHTML this is demonstrated by interface `SHTMLAction`.
In the process of dynamic menu creation method createMenu adds an `SHTMLMenuListener` to each menu. `SHTMLMenuListener` is used to update all actions to reflect the up to date enabled state prior to selection of a menu. To be able to do this, `SHTMLMenuListener` must determine, whether or not an action that is to be updated, actually *has* a method to update its enabled state. Interface `SHTMLAction` is defined so that `SHTMLMenuListener` can do that.
`SHTMLMenuListener` checks if an action is of instance `SHTMLAction` which it only would be if it implements interface `SHTMLAction`. Only if an action is an instance of `SHTMLAction`, its update method is called, because otherwise `SHTMLMenuListener` can not be sure if there is a method `update` in the particular action object.
Another advantage of the interface methodology is that objects of any class can implement interface `SHTMLAction` so that an object instantiated from class `SHTMLUndoAction` can be an instance of `SHTMLUndoAction` and an instance of `SHTMLAction` too even if `SHTMLAction` is not a superclass according to the class hierarchy of `SHTMLUndoAction` .

Using listeners

Listeners are referred in many places of this documentation probably already giving an idea about how and why they are used. Anyway the listener concept should be explained in more detail here.

Example: Font manipulation

In stage 3 of SimplyHTML many classes dealing with font manipulation had been added. These classes mostly are GUI elements or are related to GUI elements in some way. The interaction between objects during font manipulation demonstrates the importance of proper design in handling such a rather complex topic.
In class `FontPanel` for instance several objects allow changes to font attributes while all attributes are reflected in another object, the sample view.

Listener interface instead of hard coding object relations

Instead of hard coding a relationship between the font attribute selectors and the sample view component, each attribute selector *defines* a listener interface. Whenever an attribute is changed, a change event is fired in the format that interface defines.
The sample view component in turn *implements* the interface by having a method `valueChanged` which is defined in the listener interface of the font selectors. The sample view component is then registered as a listener with the component defining the interface.
Having functions to handle calls to method `valueChanged`, the functions need to be coded only once and only at the place they belong to - the sample view component in our case.

Conclusion

The listnener concept is an elegant way of letting an arbitrary number of objects dynamically interact without having to hard code relationships between objects. By defining interfaces and listening to events a clear separation according to object boundaries is accomplished.
By implementing code reacting on events, redundancies are avoided and objects do not need to 'know' about how other objects have to be changed by own actions.

Discrepancies in HTML and CSS rendering

SimplyHTML tries to implement HTML and CSS usage as close to the specified standard as possible. Still there are discrepancies for rendering of the resulting documents when viewed in different environments.
This chapter lists known discrepancies, why they seem to occur and how SimplyHTML tries to compensate the effects, if possible. Any additional hints and ideas to the author are appreciated.
Results have been tested in the following environments so far:

- Netscape 6.2.1 (SuSE Linux 8.0)
- Opera 6.0 B 1 (SuSE Linux 8.0)
- Internet Explorer 5.5 (Windows Me)
- Java J2SE 1.4 (SuSE Linux 8.0, Windows Me, Windows NT 3.51)

Following is a list of known discrepancies.

Font names

Fonts are locally bound to the machine SimplyHTML is running on. When formatting text to display font '`Palatino`' for instance it is not possible to predict if respective document will display similarly in any given environment. To make it easier to exchange similar font settings over different system environments, some standardized font names can be used. Common font names for that purpose are

- `Sans-Serif`
- `Serif`
- `Monospace`

Unfortunately, the Java language has the name `SansSerif` for the font that most other applications know as `Sans-Serif`. As well Java uses name `Monospaced`, while other applications partly use `Monospace`.

Solution

This effect is fixed by mapping between the possible values mentioned above with class `AttributeMapper`. Class `AttributeMapper` is used in class `SHTMLWriter` to map from Java to HTML and in class `SHTMLDocument.SHTMLReader.SHTMLCharacterAction` to map from HTML to Java.
In the defualt style sheet of SimplyHTML this is solved by having several font family names with the one relevant for Java as the first, e.g. `p { font-family:SansSerif, Sans-Serif; }`. For some reason, however, this does not work with Java on Linux, i.e. having more than one font family name in the style sheet causes Java to not recognize the font stlye name at all under Linux.

Font sizes

Due to a bug in the `javax.swing` package, font sizes are rendered approximately 1.3 times smaller in `JEditorPane` than in any browser (bug id 4765271, see

`http://developer.java.sun.com/developer/bugParade/bugs/4765271.html`).

Solution

SimplyHTML compensates this bug by providing customized views in class SHTMLEditorKit.SHTMLFactory. The views adjust font sizes before they are rendered so inside SimplyHTML fonts are displayed similar to as they are displayed in web browsers.
Unfortunately this does not fix the bug for cases where HTML is being displayed through Java APIs such as JavaHelp. So a bug fix from Sun to become available soon would still be highly welcome.

Table cell borders

Up to J2SE 1.4 cell borders are not rendered individually and there is no way to have different colors for borders of different sides of a cell. Either a border is drawn around all sides of a table cell or no border is drawn. There is no way for example to draw a vertical border between two cells only while the other sides of these cells have no borders.

Solution

SimplyHTML uses customized views to establish individual border rendering for table cells. Unfortunately this does not apply for cases where HTML is being displayed through Java APIs such as JavaHelp. A fix from Sun to become available soon would still be highly welcome.

Table cell margins

The CSS specification describes CSS attribute `margin` and its variations `margin-top`, `margin-bottom`, etc. as a way to set the distance between two block elements such as two paragraphs to each other but also for elements such as a table cell. However, a setting of `margin-left:2pt` for an arbitrary table cell is not being rendered up to now in any of the tested browsers.
Instead, only HTML attribute `cellspacing` is rendered so far, which is applicable only in the `table` tag (i.e. affecting all cells of respective table). Therefore specification and rendering of distances between individual table cells or for individual sides of a table cell is done correctly in SimplyHTML but it will not be shown in a web browser as it is shown in SimplyHTML.
Because SimplyHTML is built around formatting through CSS attributes, the `cellspacing` attribute can not be set for a given table in SimplyHTML. Attribute `cellspacing` is rendered in SimplyHTML, when contained in an exisitng HTML file.

Solution

There is no solution for this effect up to now.

Using Java Web Start to launch SimplyHTML

Java Web Start - a technology for simplifying deployment of Java applications - gives users the power to launch full-featured applications with a single click from a Web browser. Introduced in version 1.4 of the Java 2 Standard Edition (J2SE) Java Web Start allows to download and launch applications, such as SimplyHTML, without going through complicated installation procedures.

Benefits

The following benefits as listed in the Java Network Launching Protocol (JNLP) specification result from using Java Web Start:

- **No installation phase**: A JNLP Client simply needs to download and cache the application's resources. The user does not need to be prompted about install directories and the like.
- **Transparent update**: A JNLP Client can check the currently cached resources against the versions hosted on the Web Server and transparently download newer versions.
- **Incremental update**: The JNLP Client only needs to download the resources that have been changed when an application is updated. If only a few of the application's resources have been modified, this can significantly reduce the amount of data that needs to be downloaded when upgrading to a new version of an application. Furthermore, incremental update of individual JAR files is also supported.
- **Incremental download**: A JNLP Client does not need to download an entire application before it is launched. For example, for a spreadsheet application the downloading of the graphing module could be postponed until first use. JNLP supports this model by allowing the developer to specify what resources are needed before an application is launched (eager), and what resources can be downloaded later (lazy). Furthermore, JNLP provides an API so the developer can check if a resource is local or not (e.g., need to be downloaded or not), and to request non-local resources to be downloaded.
- **Offline support**: A JNLP Client can launch an application offline if a sufficient set of resources are cached locally. However, most applications deployed using JNLP are expected to be Web-centric, i.e., they will typically connect back to a Web server or database to retrieve their state. Hence, many applications will only work online. The application developer specifies if offline operation is supported, and what resources are needed locally to launch the application offline.

How Java Web Start works for SimplyHTML

With stage 8 of SimplyHTML, the application home page at `http://www.lightdev.com/dev/sh.htm` holds a link to a `.jnlp` file which in turn specifies all details of application SimplyHTML (required files, descriptions, etc.). When the link is clicked, Java Web Start is invoked on the client and the application is loaded down to the client. Once loaded, SimplyHTML is launched and the application can be used immediately.

No manual installation, no copying of files, no command line scripting or desktop links, no compatibility checking, nothing.
Users can choose to always start SimplyHTML through the web or to download it to the client permanently and work with the application offline.

How it is done

To achieve a Java Web Start for SimplyHTML a `.jnlp` file is created as follows (with `codebase` below having an example entry)

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec="1.0+" codebase="http://www.lightdev.com/dev/">
  <information>
    <title>SimplyHTML</title>
    <vendor>Light Development</vendor>
    <homepage href="http://www.lightdev.com/dev/sh.htm" />
    <description>SimplyHTML text processor for HTML and CSS</description>
    <offline-allowed/>
  </information>
  <security>
    <all-permissions/>
  </security>
  <resources>
    <j2se version="1.4+" />
    <jar href="SimplyHTML.jar"/>
    <extension name="Java Help" href="javahelp.jnlp">
    </extension>
  </resources>
  <application-desc main-class="com.lightdev.app.shtm.App" />
</jnlp>
```

A similar `.jnlp` file is created to deploy the JavaHelp runtime extension (file `jhall.jar`). The `.jnlp` files are copied onto the web server along with the signed application `.jar` file. Once the application home page has the mentioned link to the `.jnlp` file, it is ready to be 'Web Started'.

References

A very good article about how to 'Web Start' an application can be found at
http://developer.java.sun.com/developer/technicalArticles/Programming/jnlp/

A perfect explanation of how to obtain a certificate from a Certificate Authority and how to sign own code with such certificate can be found at
http://www.dallaway.com/acad/webstart/

The official Java Web Start product page is at
http://java.sun.com/products/javawebstart/

Using SimplyHTML

Once application SimplyHTML is set up as described in chapter 'Getting started', it is ready to be used. The chapters in this section describe *usage* of application SimplyHTML. For information about how the functionality mentioned in this section is achieved, please consult chapter 'Inside SimplyHTML'. This section is divided into the following chapters

- What is SimplyHTML?
- Creating new documents
- Opening existing documents
- Saving documents
- Editing documents
- Closing documents
- Using plug-ins

Users might want to start with 'What is SimplyHTML' to see whether or not the application fits their needs. Besides general information about document usage in SimplyHTML the part about editing documents mentioned above takes the most room as it is the main functionality of SimplyHTML.

What is SimplyHTML?

As shortly described in chapter 'About the SimplyHTML project ', SimplyHTML is an application for text processing on the basis of HTML documents formatted with CSS. It combines functionality of a word processor with the standards of HTML and CSS.
Usage of the HTML and CSS standard for documents created with SimplyHTML opens a wide variety of usage possibilities because many other applications 'understand' and use HTML and CSS such as

- web browsers
- other text processors
- presentation and graphics software

Features

SimplyHTML features the following functionality

- opens, maintains and saves documents in HTML and CSS format
- any number of documents can be opened at the same time and are displayed in a tabbed pane
- formatting of paragraph styles
- creation of own named styles for paragraphs
- formatting of font attributes on character level
- rich table formatting
- list formatting
- insertion and formatting of JPEG and GIF images
- creation and manipulation of links and link anchors
- drag and drop in the editor pane
- cut and paste for styled HTML text
- cascading undo/redo
- plug-in facility for extension of SimplyHTML
- support of Java Web Start for easy deployment
- editor for HTML code with syntax highlighting

See 'Planned development stages' for additional features to be present in future.

Differences of HTML and CSS in various environments

SimplyHTML tries to consequently adhere to HTML and CSS standards. However, there are discrepancies over different environments, which sometimes can be compensated, sometimes only partly or not at all. Please see chapter ' Discrepancies in HTML and CSS rendering' on that subject too.

Creating new documents

To create a new empty document

- select 'New' from menu 'File'

A new tab in the main frame of SimplyHTML will be opened with a new empty and untitled document to work with. All other open documents remain open which is indicated by respective tabs identifying other open documents. The display switches to the newly created document with its tab on top.

The document is now ready to be edited .

Using an existing style sheet for new documents

Usually a reference between a document and a style sheet containing named styles is only created when named styles are added to a document explicitly. In cases where a new document shall use an existing style sheet which was previously created for another document, option 'Link new documents with default style sheet' has to be set in the options dialog.

Opening existing documents

To open an existing document

- select 'Open...' from menu 'File',
- choose a document file to open from the selection frame possibly by navigating to a different location than initially shown
- press 'Open'

A new tab in the main frame of SimplyHTML will be opened with the chosen document opened into it. All other open documents remain open which is indicated by respective tabs identifying other open documents. The display switches to the opened document with its tab on top.
The document is now ready to be edited .
Note: Documents are opened using the HTML version set in the options dialog. Unless set otherwise, HTML 3.2 is used as the default.

Saving documents

To save a document, it has to be visible as the currently edited document. To save this document

- select 'Save' from menu 'File'

The document will be saved at the location it was openend from. If respective document was newly created and thus not saved so far, you will be prompted for a location and file name for the document as with saving a document under a different name (see below).
Important: You have to enter a file name including extension (.htm or .html). The extension will not be completed by the application if it was omitted.
Note: Documents are opened using the HTML version set in the options dialog. Unless set otherwise, HTML 3.2 is used as the default.

Image directory

Along with the document file, an image directory is created with any image file referenced in the saved document.

Style sheet maintenance

If named styles were created or changed for a document, a style sheet will be created during the save process if none is already present. If a style sheet with the same name is found at the target location, it is merged with the style sheet to be saved. Instead of defining named styles for a document and merging them with an existing style sheet, a new document can also use an existing style sheet right away. See 'Creating new documents' and chapter 'Options dialog' for an explanation about this option.
Caution: An existing style sheet with the same name could have styles with the same name as altered ones in the saved style sheet. Overwriting such styles could cause unwanted styles to appear in other documents sharing the particular style sheet.
Therefore you should consider to either

1. not save documents in the same directory when they do not share the same set of named styles or
2. use different style names for different styles over all documents sharing the same style sheet

Saving a document under a different name

To save a document under a different name or at a different location than where it has been openend from

- select 'Save as...' from menu 'File'
- choose a new location and/or new file name and
- press 'Save'

Respective document again has to be visible as the currently edited document.
The document will be saved at the chosen location under the given file name. The original document remains intact at the original location but SimplyHTML switches the current document to

be the newly saved one so subsequent save operations go to that file unless explicitly changed again.

Editing documents

The main functionality of SimplyHTML is to create and manipulate text so editing an existing text is a main part of this documentation as well. In this section all editing functions are explained in detail. They are divided into the follwing topics.

Common edit functions

SimplyHTML implements the common edit functions such as cut, copy and paste as usualy with any text processing application. They are available through menu 'Edit'.

To cut, copy or paste contents of SimplyHTML documents

- select a text portion and
- choose an option from menu 'Edit'

Cut, copy and paste are performed including styles. In addition, the editor of SimplyHTML has drag and drop capabilities, by which the same functionality can be reached simply by selecting, clicking and dragging certain text portions.

Changing font settings

Stage 3 of application SimplyHTML adds font manipulation functionality in two ways. Font settings can be changed either through a font dialog or through respective controls in the tool bar.

Changing several font settings at once

To change all font settings for a text portion at once

- select the part of text to change font settings for
- select 'Font' from menu 'Format' or press respective button in the tool bar
- select font settings from the dialog, that appears and
- press 'OK'

Changing single font settings

A quick way to change single font settings is through the tool bar

- select the part of text to change a single font setting for and
- press respective button in the tool bar

This way the tool bar allows to switch font family and size as well as to toggle between bold/normal, italic/normal and underlined/not underlined each in one step.

Creating and manipulating tables

To create a table,

- move the caret to the location where the table shall be inserted
- select 'Table...' from menu 'Insert'
- choose the number of columns the new table initially shall have and
- press 'OK'

A new table with a single row and the chosen number of columns will be inserted at the caret postion. The caret is placed into the first cell of the new table.

Caret movement inside tables

By pressing the TAB key, the caret is moved into the next cell. Pressing SHIFT TAB moves the caret to the previous cell. Pressing the TAB key while the caret is in the last cell of the table will append a new row before moving to the next cell.

Changing the table structure

Structural changes include inserting or appending rows and columns and deleting rows and columns. To change the table strucutre

- move the caret into the table and
- select an option from menu 'Table' accordingly

Rows and columns always are inserted before the current caret position. Column widths are adjusted to maintain the table width.

Changing the table format

Table format includes all attributes of the table and its cells such as table width, cell width, background color of cells, text alignment inside cells, margins and borders, etc. To change the table format

- move the caret into a table,
- select 'Table...' from menu 'Format',
- set table and cell attributes in the table format dialog and
- press 'OK'

The table dialog initially shows all attributes currently set for that table. All changes made are applied to the underlying table at once when button 'OK' is pressed. Cell attributes can be applied to the current cell, the current row, the current column or to all cells.

Creating and manipulating lists

Lists are a consecutive amount of paragraphs being displayed with a symbol separating each paragraph at the beginning. There can be ordered lists, which follow an order such as `1., 2., 3., 4.` or `a., b., c., d.` and unordered lists following no specific order such as lists with a bullet symbol at the beginning of each paragraph.
Lists can be created in two ways

1. start list formatting and then type content in the form of list items as needed or
2. type in content and then switch on list formatting for recently typed paragraphs

Turning list formatting on or off

To turn list formatting on or off

- place the caret to the position list formatting shall be switched on or off or
- select a text portion for which list formatting shall be switched on or off and
- select 'Bulleted list on/off' or 'Numbered list on/off' from menu 'Format'

Alternately, respective toggle buttons in the tool bar can be used as well.

Caret movement inside lists

While typing text, the caret moves similar to a region not being formatted as list. Pressing [Enter] while the caret is inside a list will create a new list item after the item the caret is currently in.

Changing the list formatting

List format can be changed individually through a list format dialog too. The list format dialog can be used to change the list type (bulleted, numbered, bullet symbol and order criteria) as well as the indentation of a list. To change list formatting individually

- select the the text portion to change list formatting for
- select 'List...' from menu 'Format'
- make settings in the list format dialog and
- press 'OK'

Working with images

In line with the HTML specification for image references, images can be added to any document of application SimplyHTML by inserting references to separate image files. To insert an image

- place the caret to the point in the document where an image shall appear
- select 'Image...' from menu 'Insert'
- in the image dialog press button 'Add'
- locate the file containing the image in the file chooser
- press 'OK' in the file chooser
- adjust attributes in the image dialog as needed and
- press 'OK' in the image dialog

The selected image is inserted by placing a reference to the selected image file into the document. The image appears at the chosen location as long as the associated image file is present in the image repository. The image repository is created and maintained by application SimplyHTML automatically. When adding image files as described above, they are copied to and kept in the image repository. Once an image file is present in the image repository of a document, it does not have to be located in the file system again.

Important: If a document containing image references is moved to another storage location by hand, the associated directory with image files (the image repository, a directory named `images`) has to be moved too.

Making changes to images in documents

To change settings of an image

- click on the image
- select 'Image...' from menu 'Format'
- adjust attributes in the image dialog as needed and
- press 'OK' in the image dialog

To remove an image and its file reference from a document

- click on the image
- press the [Backspace] or [Delete] key

To remove an image file from the image repository

- select 'Image' from menu 'Insert'
- select the image file to remove from the list of files in the image dialog
- press button 'Delete'
- press 'Yes' in the option pane that appears
- press button 'Cancel' in the image dialog

This will permanently delete the image file from the image directory of the currently open document.

Important: When deleting an image file from the image repository, all references to that image file inside the document remain intact but the image is rendered as 'broken' link (broken icon picture).

Changing paragraph styles

As opposed to styles applied to single characters, paragraph styles are a way to define a group of formatting settings for one or more paragraphs at once.

To set the style for one or more paragraphs

- select the paragraph(s) to be formatted in the editor
- choose 'Paragraph...' in menu 'Format'
- set formatting options accordingly and
- press 'OK'

All contents of the selected paragraph(s) will assume the settings formatted in the Paragraph Style Dialog accordingly, unless other individual styles have been set for single characters inside the respective paragraph(s).

Read on to find out how to define and store a set of predefined formats as named styles.

Creating and manipulating named styles

The previous chapter describes how a certain format can be applied to one or more paragraphs at once. However, in many cases similar formats are applied to the content over and over again. To reduce maintenance effort and storage space, named styles can be defined for such formats. Named styles are not applied directly to a document when created. Instead they are saved to the style sheet of a document. To use named styles two steps are necessary:

1. define a named style by setting all its formatting attributes and saving it to the style sheet
2. select a portion of the document in the editor and pick a named style for it from the style selector in the tool bar

Once a named style is saved in the style sheet, it will appear in the style selector in the tool bar of SimplyHTML. It is necessary to define named styles only once for a combination of document and style sheet. The style sheet for a document is maintained automatically by SimplyHTML and saved whenever a document is saved.

Creating a named style

To define and save a named style

- select 'Named style...' from menu 'Format'
- set formatting attributes in the paragraph style dialog accordingly
- select an element type for the new style (selector located above list of named styles)
- press button 'Save as...'
- enter a name for the style and press 'OK'

Important: Styles are created only for the element type that is currently selected (paragraph, link, heading, etc.). Only changed attributes are saved to the style sheet.

Changing a name style

To change settings for an existing named style

- select 'Named style...' from menu 'Format'
- select the element type (selector located above list of named styles)
- select the named style to change settings for in the list of existing named styles
- change formatting attributes in the paragraph style dialog accordingly
- switch back to the 'Paragraph' tab if you are on tab 'Font'
- press button 'Save'
- press button 'OK'

The named style for the selected element type (paragraph, link, heading, etc.) will be overwritten with the new settings. At the same location named styles can be deleted from the style sheet too.

Deleting a named style

To delete a named style

- select 'Named style...' from menu 'Format'
- select the named style to change settings for in the list of existing named styles
- press button 'Delete'

Caution: Deleting a named style will cause content portions formatted with that style to be rendered in an unpredictable style. Be sure to delete only unused styles. Keep in mind that a named style could be used in another document sharing styles with the currently edited one.

Creating and manipulating links

A link in HMTL is a reference to another location and/or document a user can jump to. See chapter ' Links in HTML' for a technical description of links.

To create a link

- select a portion of text to format as link in the editor
- select 'Link...' from menu 'Format'
- make link settings as appropriate and
- press 'OK'

To change settings for an existing link

- move the caret into text formatted as link
- select 'Link...' in menu 'Format'
- make link changes as appropriate and
- press 'OK'

With both selections above, a link dialog is brought up to set link attributes accordingly. Once the link dialog is completed, the previously selected text portion is formatted as link accordingly.

Note: A link is only shown as link. It does not work as a link in the way that the link target is shown when the link is clicked. This probably will be added in a future stage.

See also

Links in HTML
Creating and manipulating link anchors
Link dialog
Link anchor dialog

Creating and deleting link anchors

Link anchors are a way to designate certain positions inside a document. By using link anchors a link can directly reference and jump to a certain position inside a document. To make this possible, a link anchor has to be defined in that document before any link can reference to that position.

To define a link anchor inside a document that is to be targeted by a link

- open the target document
- select 'Anchors...' from menu 'Format'
- in the link anchor dialog select the text portion inside the document to be used as a link target
- click button 'Add'
- type in a name for the new link anchor
- press 'OK'
- select 'save' from menu 'File' to save the document

A link to the newly defined link anchor can now be set in the link dialog. As an alternative to above steps a link and link anchor can be set in one step using the link dialog too.

To delete a link anchor

- open the target document
- select 'Anchors...' from menu 'Format'
- in the link anchor dialog select the name of the link anchor to delete
- click button 'Delete'
- press 'OK'
- select 'save' from menu 'File' to save the document

Important: Above steps only delete the selected link anchor. Links referencing this link anchor remain the same thus leading to a 'dead' location.

As an alternative to above steps a link anchor can be deleted through the link dialog too.

See also

Links in HTML
Creating and manipulating links
Link dialog
Link anchor dialog

Setting the element type

A specialty of HTML is to have different element types for content of a document separate from the content formatting. Element types could be paragraph, link or a certain kind of heading. One can think of element types being a way to distinguish certain structures in a document while named styles are a way to differentiate between certain formats both independent from each other.

Element types are applied to one or more paragraphs always. By default a paragraph is of element type paragraph as well.

To set another element type,

- select one or more paragraphs in the editor pane
- choose an element type from the element type combo box in the tool bar

Currently the following element types can be set

- paragraph
- heading 1
- heading 2
- heading 3
- heading 4
- heading 5
- heading 6

Working with HTML code

For an experienced user being familiar with HTML it sometimes is quicker to manipulate a certain portion of HTML directly instead of having to wade through GUI elements. For this reason in stage 10 of application SimplyHTML a way to work on the HTML representation of any given text document is implemented.

To see and edit the HTML representation of a given text document

1. open the document
2. click on the tab `HTML code view` at the bottom of the editor pane

The display will switch from layout view to HTML code view. In the HTML code view the structural information for a particular document is shown along with its content. Tags and attributes are mixed with plain text.

The HTML code view does not show how the plain text will look. Instead it highlights the structural portions of HTML and allows to work on the HTML code.

All changes to the HTML code are transformed to the resulting layout automatically when switching back to layout view (clicking on tab `Layout view` at the bottom of the editor pane, that is).

Important: To work on HTML code should only be done by users being experienced in HTML use. Changing HTML code can destroy the structure of the document leading to unpredictable results in the layout of respective document and even loss of content.

Note: With bigger documents it can take a short while to display the HTML code initially.

Using find and replace

In development stage 11 of application SimplyHTML functionality to find and replace text phrases inside of documents was added. Find and replace is implemented as a dialog combining all relevant functions. To invoke find and replace

1. open menu `Edit`
2. select `Find and replace`

A dialog will appear allowing to set all relevant options and to initiate a find or replace operation. When more than one document is open while find and replace is invoked option 'Search whole project' allows to perform find and replace over all open documents.

Closing documents

To close the document, that is currently shown in the editor

- select 'Close' from menu 'File'

To close a currently open document, that is not shown in the editor

- click on the tab representing the document and
- select 'Close' from menu 'File'

To close all currently open documents

- select 'Close all' from menu 'File'

Closing a document will attempt to save pending changes before closing. If this fails, an error message will be shown the document will remain open.

When exiting the application while documents are open, as well attempts are made to save pending changes for all open documents. If this fails, an error message will be shown for each document, where a save could not commence, respective document(s) stay open and the application is not terminated.

Using plug-ins

SimplyHTML provides a mechanism to extend the main editing functionalities of the application by external objects called 'plug-ins'. Plug-ins are installed by copying one or more plug-in files into the directory SimplyHTML is installed in. Once installed, a plug-in is loaded and activated inside SimplyHTML upon start of the application by default.

Accessing plug-in functions

Each plug-in provides an own menu in menu 'Plugin' of SimplyHTML where all plug-in functionality is available. In addition there usually is a menu item in the 'Help' menu having documentation on the plug-in.

To access the functions of a plug-in

- open menu 'Plugin'
- open the sub-menu for respective plug-in

To access help information for a given plug-in

- open menu 'Help'
- select the menu item referring to help for a given plug-in

Activating and de-activating plug-ins

Loaded plug-ins can be activated or deactivated for each user individually. As well the location where a plug-in is docked inside SimplyHTML's main window can be changed. To change the settings of any loaded plug-in

- select 'Manage plug-ins...' from menu 'Plugin'
- choose a plug-in from the list of loaded plug-ins
- adjust settings accordingly and
- press 'Close'

The settings are applied to respective plug-ins and are stored individually for the user who made the changes. See respective chapter in section 'Inside SimplyHTML' if you would like to know more about plug-ins.

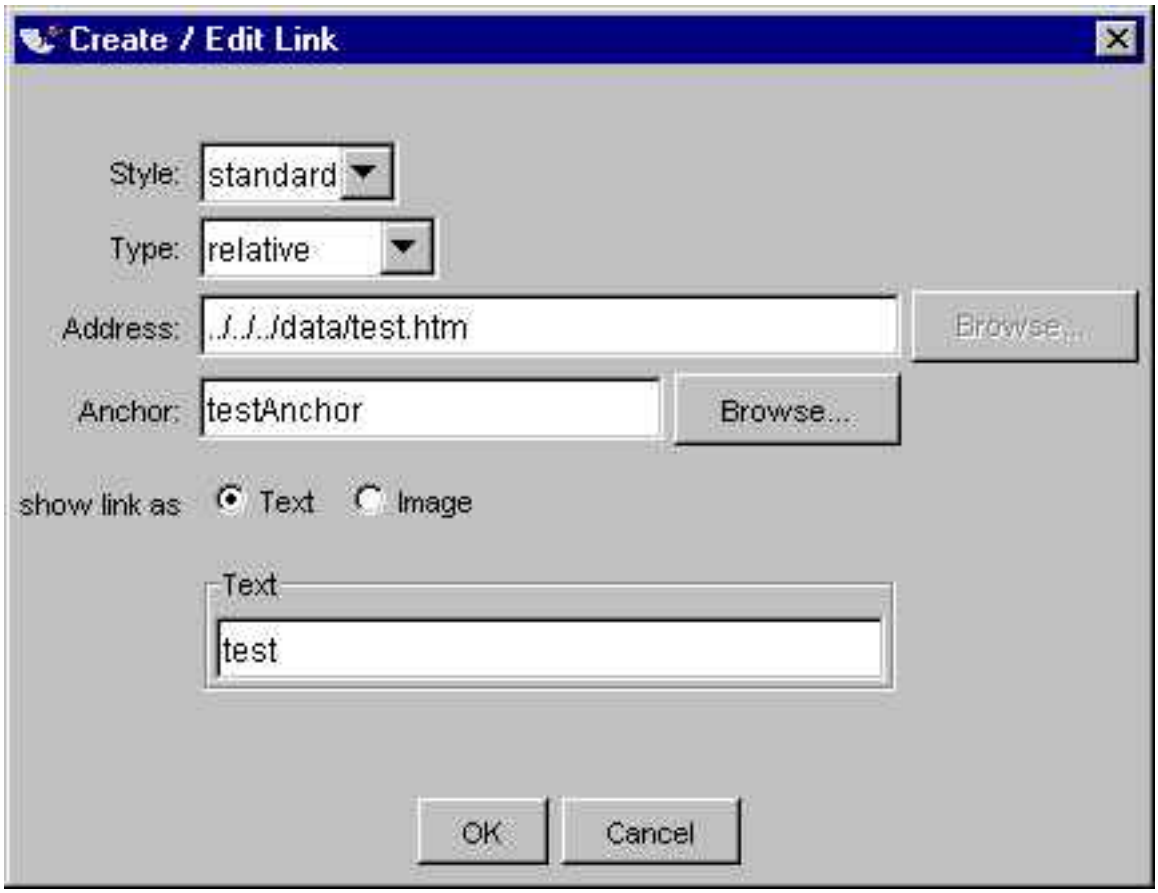
User interface

This chapter explains parts of the user interface of application SimplyHTML. The chapter describes from the perspective of certain parts of the GUI (e.g. a dialog) as opposed to previous chapters explaining parts of the application from a functional perspective (e.g. 'how to save a file'). This section covers of the following GUI parts of application SimplyHTML

- Link dialog
- Link anchor dialog
- Image dialog
- Options dialog

Link dialog

In the link dialog all link attributes are shown and can be changed (see also 'Creating and manipulating links').



the link dialog

Style

Combo box `Style` allows to set by which named style the link shall be formatted in the editor. Any existing named style from the associated style sheet is listed in this combo box.

Type

With combo box `Type` a link type can be set such as 'local file', 'relative address', 'mailto', etc. Other elements of the link dialog are enabled or disabled according to selections in combo box `Type`.
When switching between types 'local' and 'relative' or vice versa, contents of text field Address is automatically changed accordingly (relative or absolute address snytax),

Address

In text field `Address` the address of the link target is shown. An address can be typed into this field at any time. No protocol such as `telnet:/` or `file:/` needs to be typed into field `Address`. The protocol information is automatically generated from the selection in combo box `Type`.

Browse (Address)

With button `Browse` next to text field `Address` a local file can be selected through a file chooser dialog. This button is enabled, if `'local'` is selected in combo box `Type`.

Anchor

In text field `Anchor` the name of a link anchor can be entered (see also 'Creating and deleting anchor links'). The entered anchor name refers to an anchor link inside the target referred to by the entry in text field `Address`. The separator character `(#)` for link anchor names does not need to be entered, it is completed automatically by the link dialog.

Browse (Anchor)

With button `Browse` next to text field `Anchor` an existing link anchor can be selected or a new link anchor can be defined throgh the link anchor dialog. The link anchor dialog is shown for the link targed specified in text field `Address`.

Show link as Text

With button `Show link as Text` is selected that the link specified in this dialog is to be entered as a text link into the associated document. When this button is selected, panel `Text` will be shown.

Text panel

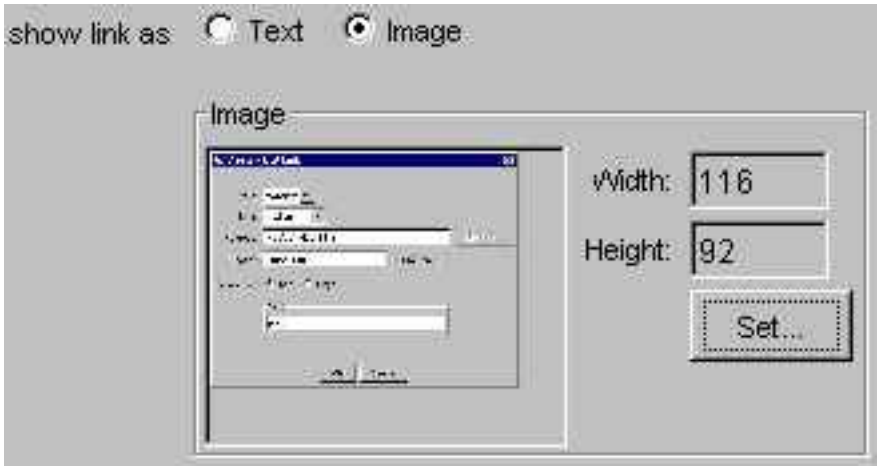
In panel `Text` a text can be entered that is to be formatted as link. Panel `Text` is only visible, if button Show link as Text has been selected.

Show link as Image

With button `Show link as Image` is selected that the link specified in this dialog is to be entered as an image link into the associated document.

Image Panel

In panel `Image` an image and image size can be selected. The selected image will be shown in the associated document as link. All image settings in panel `Image` are only *shown*. They can be set or changed through button `Set...`.



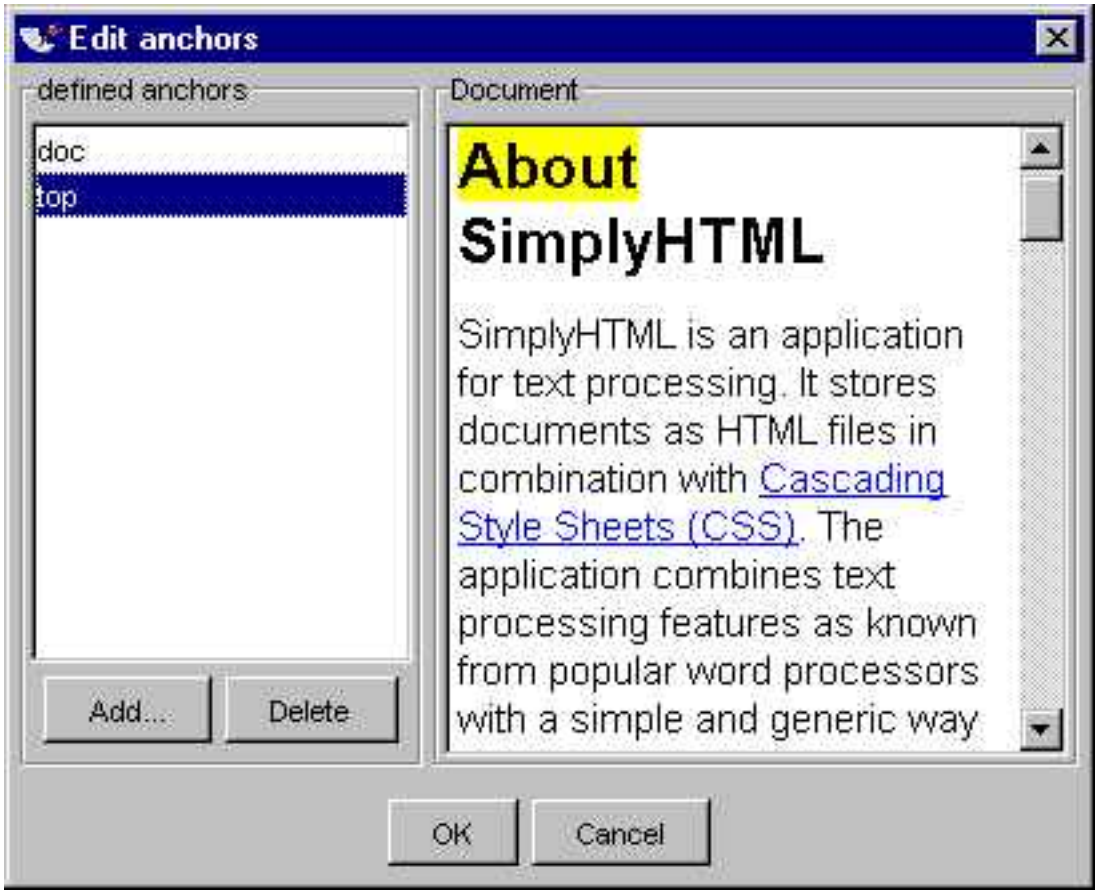
the link image panel

Button Set...

All image properties in panel `Image` are set through button `Set...` which will bring up an image dialog. In the image dialog an image file and its size can be selected accordingly.

Link anchor dialog

The link anchor dialog allows to create and remove link anchors for a document. It is shown with either the document that is currently edited (when opened through menu 'Format') or with the document targeted in a link (through the link dialog).



the link anchor dialog

Defined anchors panel

In panel `defined anchors` all existing link anchors for the shown document are listed. Clicking one link anchor name in panel `defined anchors` highlights the position of that link anchor in panel `Document`.

Add

Clicking button `Add` will show an input dialog to enter a name for a new link anchor to create. Button `Add` is only enabled when a portion of text is selected in panel `Document`. The new link anchor will be created for the position selected in panel `Document`.

Delete

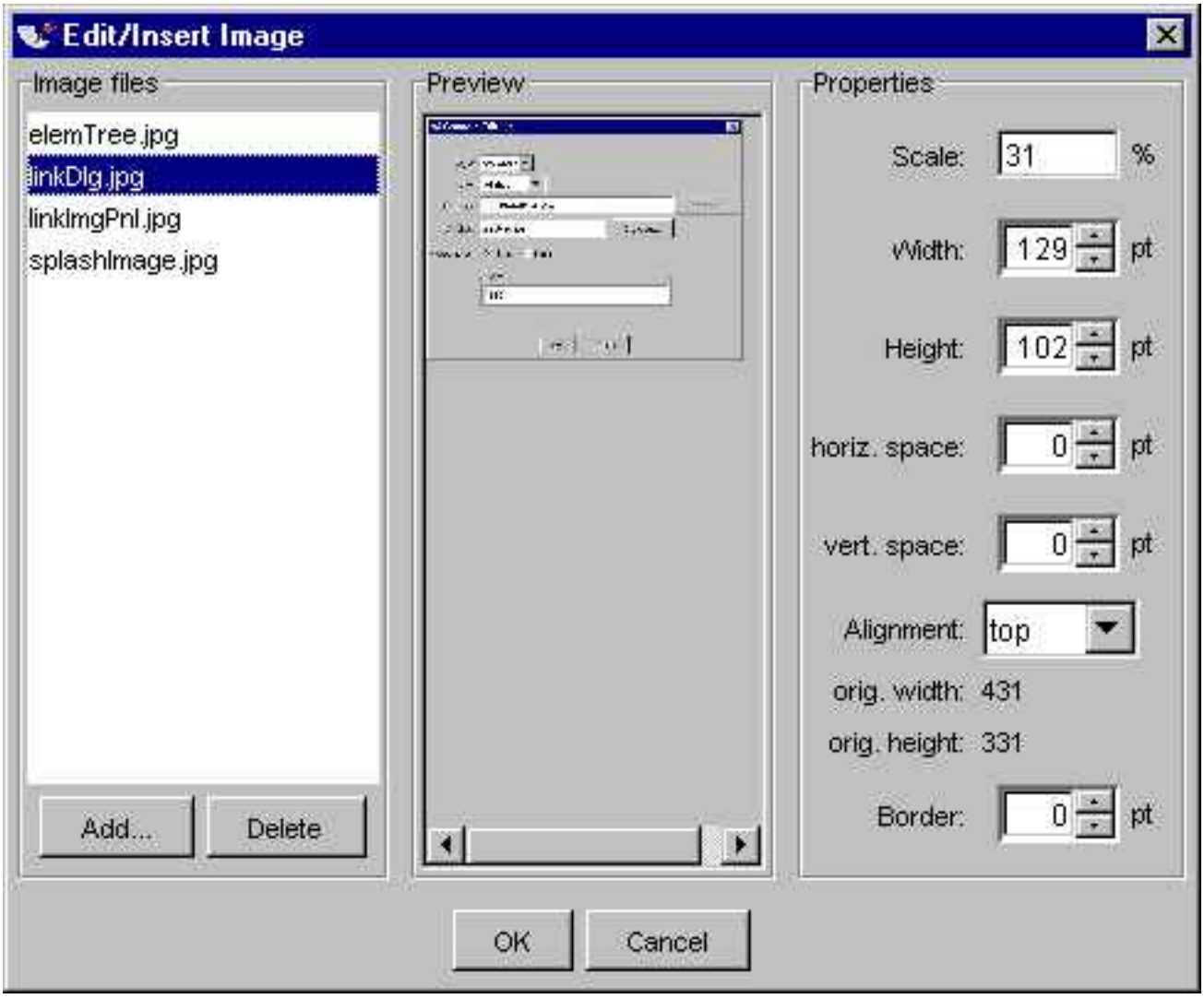
When button `Delete` is clicked, the selected link anchor is removed from the assocated document. Button `Delete` is only enabled when a link anchor is selected in the `defined anchors` panel.

Document panel

Panel `Document` shows the document this link anchor dialog allows to set link anchors for. In the shown document, the currently selected link anchor is highlighted. If a new link anchor is to be created, panel `Document` is used to select the text to formatted as a link anchor first.

Image dialog

The image dialog is used to add images to a document and change attribute settings for such image. It also allows to maintain the image repository a document is associated to. See also chapter 'Working with images' about how images are applied to documents.



the image dialog

Image files panel

In panel image files all files are shown that are stored in the image repository that is associated to the currently edited document. Clicking on one image file name listed in panel Image files shows the image in panel Preview and lists image attributes in panel Properties.

Add

Press button Add to add a new image file to the repository. A file chooser dialog will appear to select a file from. The selected file is copied into the image repository associated with the document and displayed in pnale Image Files.

Delete

Button Delete is used to remove a file from the image repository. Respective file is permanently deleted from the directory used as an image repository.
Note: All references to the deleted image in any document associated to this image repository will be 'broken' when deleting an image from the repository.

Preview panel

Panel Preview displays an image selected in panel Image files. An image is scaled to the current size of the preview area when initially displayed. When image attributes are changed in panel Properties, the image size changes accordingly. The size of the preview area remains unchanged and scroll bars are shown when an image does not fit into the preview area. The size of the preview area can be changed by resizing the dialog window. In this case the image size remains the same and only the size of the preview area changes along with the window size.

Properties panel

In panel Properties attributes of the image currently displayed in panel Preview are shown. Changes to image attributes in panel Properties directly affect the image shown in panel Preview, i.e. the image changes in size accordingly.

Scale

Use this control to enter a value by which the size of the image is to be scaled. Size is scaled proportionally, i.e. the relation of width and height is always kept.

Width

Use this control to set a width for the image. Height and scale are adjusted accordingly.

Height

Use this control to set a height for the image. Width and scale are adjusted accordingly.

Horiz. Space

This field can be used to set a value for space between the image and objects on the left and right of the image.

Vert. Space

This field can be used to set a value for space between the image and objects on top and bottom of the image.

Alignment

Alignment controls how the image is to be aligned relative to other objects, i.e. an alignment setting of left aligns the image left of any other objects. (This setting is modeled but not rendered in SimplyHTML)

Border

Set the width of a border around the image. A setting of 0 means no border, any other value means a border around the image in the specified width.

Options dialog

The options dialog allows to set general preferences for application SimplyHTML. All settings are persistently stored for each individual user and reinstated when SimplyHTML is launched.

the options dialog

Look and feel

A feature of the Java[tm] platform is its 'pluggable look and feel' concept. With this setting the user can select a look and feel from those installed on the machine SimplyHTML is running on.

Write HTML files as HTML 3.2

Setting this option causes SimplyHTML to create document files in HTML 3.2. This means that some features from later HTML versions are not available such as certain CSS formats and HTML tags. Setting this option is useful for environments that require HTML 3.2 such as JavaHelp.

Write HTML files as HTML 4

Setting this option causes SimplyHTML to create documents in HTML 4. This enables certain formatting features such as individual table borders or background coloring for individual parts of text.

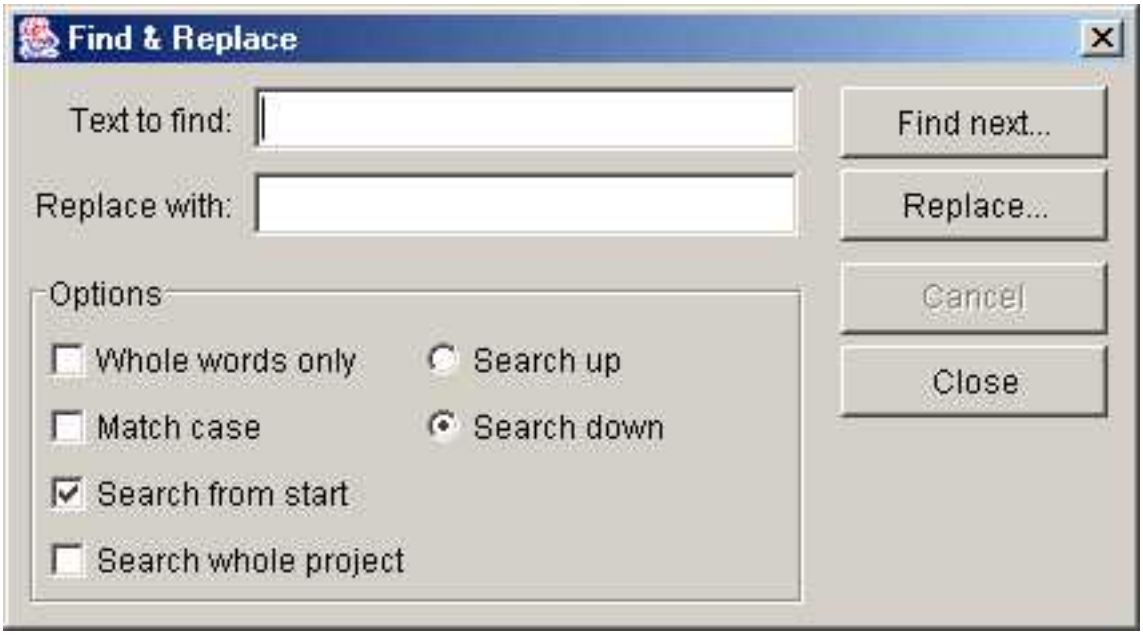
Link new documents with default style sheet

Usually a reference between a document and a style sheet containing named styles is only created when named styles are added to a document explicitly. In cases where a new document shall use an existing style sheet which was previously created for another document, this option can be used.

Selecting this option causes creation of a style sheet reference in newly created documents linking to the default style sheet ('`style.css`'). File '`style.css`' always is created by SimplyHTML, when named styles are added to a document.
When a new document is created with this option selected, the style sheet reference to '`style.css`' is included in the new document upon creation even without named styles being defined for the new document so far. The new document has then to be saved to the directory where respective file '`style.css`' is already located for the styles from '`style.css`' to be in effect in the new document.

Find and replace dialog

Find and replace is done in a dialog window where all relevant options can be set:



the find and replace dialog

Text to find

In field `Text to find` an arbitrary text phrase can be entered. This text phrase will be searched during a find or replace operation.

Replace with

In field `Replace with` a text phrase is entered that is to replace the text phrase entered in field Text to find.

Find next...

Button `Find next...` is pressed to initiate a find operation with the settings currently selected in the find and replace dialog.

Replace...

By pressing button `Replace...` a replace operation is initiated. The find and replace dialog is hidden and the next occurrence of the phrase in field `Text to find` is located. When found, an option pane lets the user choose whether or not the found occurrence shall be replaced by the phrase in field `Replace with`. Alternately the user can choose to replace all occurrences or to terminate the replace operation. Once done, the find and replace dialog appears again.

Cancel

Button `Cancel` is used to terminate a find operation. It is only enabled (not dimmed) when a find operation is in progress (button `Find next...` has been pressed). With pressing button `Cancel` the current find operation is terminated and the find and replace dialog can be used to start a new find and replace operation with different settings.

Close

With button `Close` the find and replace dialog is closed.

Whole words only

Option `Whole words only` is used to restrict matches to whole words. When not selected all occurrences of the phrase in field `Text to find` are located even when they are found as part of a word.

Match case

By selecting option `Match case` a find operation only locates occurence of the phrase in field `Text to find` when they match the exact case.

Search from start

Selection of option `Search from start` causes a find operation to begin either at the first or last position of the particular document regardless of the current caret position, depending on setting `Search up` and `Search down` respectively.

Search whole project

Option `Search whole project` is only visible when more than one document is open at the moment. With selection of option `Search whole project` all documents are searched that are currently open.

Search up

Option `Search up` causes the search to be performed from bottom to top of any given document.

Search down

Option `Search down` causes the search to be performed from top to bottom of any given document.