LaTeX for package and class authors
current version

© Copyright 2023–2025, LaTeX Project Team.
All rights reserved.*

2026-06-01

Contents

1 Introduction **2**

2 Writing classes and packages **2**
2.1 Is it a class or a package? . . . 2
2.2 Using 'docstrip' and 'doc' . . . 3
2.3 Policy on standard classes . . . 3
2.4 Command names . . . 4
2.5 Programming support . . . 4
2.6 Box commands and color . . . 5
2.7 General style . . . 5

3 The structure of a class or package **7**
3.1 Identification . . . 8
3.2 Using classes and packages . . . 9
3.3 Declaring options . . . 9
3.4 A minimal class file . . . 11
3.5 Example: a local letter class . . . 12
3.6 Example: a newsletter class . . . 12

4 Commands for class and package writers **13**
4.1 Identification . . . 14
4.2 Loading files . . . 15
4.3 Delaying code . . . 15
4.4 Creating and using keyval options . . . 16
4.5 Passing options around . . . 18
4.6 Useful status tests . . . 20
4.7 Safe file commands . . . 21
4.8 Reporting errors, etc . . . 22

*This file may distributed and/or modified under the conditions of the LaTeX Project Public License, either version 1.3c of this license or (at your option) any later version. See the source `clsguide.tex` for full details.

5 Miscellaneous commands, etc. 23
5.1 Layout parameters . . . 23
5.2 Case changing . . . 23
5.3 Better user-defined math display environments . . . 24
5.4 Normalising spacing . . . 24
5.5 Querying localisation . . . 25
5.6 Extended and expandable references of properties . . . 26
5.7 Preparing link targets . . . 28
5.8 Querying document command argument specifications . . . 29

6 Configuring the Output routine 29

7 Commands superseded for new material 31
7.1 Defining commands . . . 31
7.2 Option declaration . . . 32
7.3 Commands within option code . . . 32
7.4 Option processing . . . 33

1 Introduction

LaTeX 2$_\varepsilon$ was released in 1994 and added a number of then-new concepts to LaTeX. For package and class authors, these are described in the document `clsguide-historic`, which has largely remained unchanged. Since then, the LaTeX team have worked on a number of ideas, firstly a programming language for LaTeX (L3 programming layer) and then a range of tools for authors which build on that language. Here, we describe the current, stable set of tools provided by the LaTeX kernel for package and class developers. We assume familiarity with general LaTeX usage as a document author, and that the material here is read in conjunction with `usrguide`, which provides information for general LaTeX users on up-to-date approaches to creating commands, etc.

2 Writing classes and packages

This section covers some general points concerned with writing LaTeX classes and packages.

2.1 Is it a class or a package?

The first thing to do when you want to put some new LaTeX commands in a file is to decide whether it should be a *document class* or a *package*. The rule of thumb is:

> If the commands could be used with any document class, then make them a package; and if not, then make them a class.

There are two major types of class: those like `article`, `report` or `letter`, which are free-standing; and those which are extensions or variations of other classes—for example, the `proc` document class, which is built on the `article` document class.

Thus, a company might have a local `ownlet` class for printing letters with their own headed note-paper. Such a class would build on top of the existing `letter` class but it cannot be used with any other document class, so we have `ownlet.cls` rather than `ownlet.sty`.

The `graphics` package, in contrast, provides commands for including images into a LaTeX document. Since these commands can be used with any document class, we have `graphics.sty` rather than `graphics.cls`.

2.2 Using 'docstrip' and 'doc'

If you are going to write a large class or package for LaTeX then you should consider using the `doc` software which comes with LaTeX. LaTeX classes and packages written using this can be processed in two ways: they can be run through LaTeX, to produce documentation; and they can be processed with `docstrip`, to produce the `.cls` or `.sty` file.

The `doc` software can automatically generate indexes of definitions, indexes of command use, and change-log lists. It is very useful for maintaining and documenting large TeX sources.

The documented sources of the LaTeX kernel itself, and of the standard classes, etc., are `doc` documents; they are in the `.dtx` files in the distribution. You can, in fact, typeset the source code of the kernel as one long document, complete with index, by running LaTeX on `source2e.tex`. Typesetting these documents uses the class file `ltxdoc.cls`.

For more information on `doc` and `docstrip`, consult the files `docstrip.dtx`, `doc.dtx`, and *The LaTeX Companion*. For examples of its use, look at the `.dtx` files.

2.3 Policy on standard classes

Many of the problem reports we receive concerning the standard classes are not concerned with bugs but are suggesting, more or less politely, that the design decisions embodied in them are 'not optimal' and asking us to modify them.

There are several reasons why we should not make such changes to these files.

- However misguided, the current behavior is clearly what was intended when these classes were designed.
- It is not good practice to change such aspects of 'standard classes' because many people will be relying on them.

We have therefore decided not to even consider making such modifications, nor to spend time justifying that decision. This does not mean that we do not agree that there are many deficiencies in the design of these classes, but we have many tasks with higher priority than continually explaining why the standard classes for LaTeX cannot be changed.

We would, of course, welcome the production of better classes, or of packages that can be used to enhance these classes. So your first thought when you consider such a deficiency will, we hope, be "what can I do to improve this?"

2.4 Command names

Prior to the introduction of the L3 programming layer described in the next section, LaTeX had three types of command.

There are the author commands, such as `\section`, `\emph` and `\times`: most of these have short names, all in lower case.

There are also the class and package writer commands: most of these have long mixed-case names such as the following.

```
\InputIfFileExists  \RequirePackage  \PassOptionsToClass
```

Finally, there are the internal commands used in the LaTeX implementation, such as `\@tempcnta`, `\@ifnextchar` and `\@eha`: most of these commands contain `@` in their name, which means they cannot be used in documents, only in class and package files.

Unfortunately, for historical reasons the distinction between these commands is often blurred. For example, `\hbox` is an internal command which should only be used in the LaTeX kernel, whereas `\m@ne` is the constant -1 and could have been `\MinusOne`.

However, this rule of thumb is still useful: if a command has `@` in its name then it is not part of the supported LaTeX language—and its behavior may change in future releases! If a command is mixed-case, or is described in *LaTeX: A Document Preparation System*, then you can rely on future releases of LaTeX supporting the command.

2.5 Programming support

As noted in the introduction, the LaTeX kernel today loads dedicated support from programming, here referred to as the L3 programming layer but also often called expl3. Details of the general approach taken by the L3 programming layer are given in the document `expl3`, while a reference for all current code interfaces is available as `interface3`. This layer contains two types of command: a documented set of commands making up the API and a large number of private internal commands. The latter all start with two underscores and should not be used outside of the code module which defines them. This more structured approach means that using the L3 programming layer does not suffer from the somewhat fluid situation mentioned above with '`@` commands'.

We do not cover the detail of using the L3 programming layer here. A good introduction to the approach is provided at https://www.alanshawn.com/latex3-tutorial/.

2.6 Box commands and color

Even if you do not intend to use color in your own documents, by taking note of the points in this section you can ensure that your class or package is compatible with the `color` package. This may benefit people using your class or package and wish to use color.

The simplest way to ensure 'color safety' is to always use LaTeX box commands rather than TeX primitives, that is use `\sbox` rather than `\setbox`, `\mbox` rather than `\hbox` and `\parbox` or the `minipage` environment rather than `\vbox`. The LaTeX box commands have new options which mean that they are now as powerful as the TeX primitives.

As an example of what can go wrong, consider that in `{\ttfamily <text>}` the font is restored just *before* the `}`, whereas in the similar looking construction `{\color{green} <text>}` the color is restored just *after* the final `}`. Normally this distinction does not matter at all; but consider a primitive TeX box assignment such as:

```
\setbox0=\hbox{\color{green} <text>}
```

Now the color-restore occurs after the `}` and so is *not* stored in the box. Exactly what bad effects this can have depends on how color is implemented: it can range from getting the wrong colors in the rest of the document, to causing errors in the dvi-driver used to print the document.

Also of interest is the command `\normalcolor`. This is normally just `\relax` (i.e., does nothing) but you can use it rather like `\normalfont` to set regions of the page such as captions or section headings to the 'main document color'.

2.7 General style

LaTeX provides many commands designed to help you produce well-structured class and package files that are both robust and portable. This section outlines some ways to make intelligent use of these.

2.7.1 Loading other files

LaTeX provides these commands:

```
\LoadClass          \LoadClassWithOptions
\RequirePackage     \RequirePackageWithOptions
```

for using classes or packages inside other classes or packages. We recommend strongly that you use them, rather than the primitive `\input` command, for a number of reasons.

Files loaded with `\input <filename>` will not be listed in the `\listfiles` list.

If a package is always loaded with `\RequirePackage...` or `\usepackage` then, even if its loading is requested several times, it will be loaded only once. By contrast, if it is loaded with `\input` then it can be loaded more than once; such an extra loading may waste time and memory and it may produce strange results.

If a package provides option-processing then, again, strange results are possible if the package is `\input` rather than loaded by means of `\usepackage` or `\RequirePackage...`.

If the package `foo.sty` loads the package `baz.sty` by use of `\input baz.sty` then the user will get a warning:

```
LaTeX Warning: You have requested package 'foo',
               but the package provides 'baz'.
```

Thus, for several reasons, using `\input` to load packages is not a good idea.

For example, `article.sty` contains just the following lines:

```
\NeedsTeXFormat{LaTeX2e}
\@obsoletefile{article.cls}{article.sty}
\LoadClass{article}
```

You may wish to do the same or, if you think that it is safe to do so, you may decide to just remove `myclass.sty`.

2.7.2 Make it robust

We consider it good practice, when writing packages and classes, to use LaTeX commands as much as possible.

Thus, instead of using `\def...` we recommend using one of `\newcommand`, `\renewcommand` or `\providecommand` for programming and for defining document interfaces `\NewDocumentCommand`, etc. (see `usrguide` for details of these commands).

When you define an environment, use `\NewDocumentEnvironment`, etc., (or `\newenvironment`, etc., for simple cases) instead of using `\def\foo{...}` and `\def\endfoo{...}`.

If you need to set or change the value of a ⟨*dimen*⟩ or ⟨*skip*⟩ register, use `\setlength`.

To manipulate boxes, use LaTeX commands such as `\sbox`, `\mbox` and `\parbox` rather than `\setbox`, `\hbox` and `\vbox`.

Use `\PackageError`, `\PackageWarning` or `\PackageInfo` (or the equivalent class commands) rather than `\@latexerr`, `\@warning` or `\wlog`.

The advantage of this kind of practice is that your code is more readable and accessible to other experienced LaTeX programmers.

2.7.3 Make it portable

It is also sensible to make your files are as portable as possible. To ensure this, files must not have the same name as a file in the standard LaTeX distribution, however similar its contents may be to one of these files. It is also still lower risk to stick to file names which use only the ASCII range: whilst LaTeX works natively with UTF-8, the same cannot be said with certainty for all tools. For the same reason, avoid spaces in file names.

It is also useful if local classes or packages have a common prefix, for example the University of Nowhere classes might begin with `unw`. This helps to avoid every University having its own thesis class, all called `thesis.cls`.

If you rely on some features of the LaTeX kernel, or on a package, please specify the release-date you need. For example, the keyval options (see Section 4.4) were introduced in the June 2022 release so, if you use them then you should put:

```
\NeedsTeXFormat{LaTeX2e}[2022-06-01]
```

2.7.4 Useful hooks

It is sometimes necessary for a package to arrange for code to be executed at the start or end of the preamble, at the end of the document or at the start of every use of an environment. This can be carried out by using hooks. As a document author, you will likely be familiar with `\AtBeginDocument`, a wrapper around the more powerful command `\AddToHook`. The LaTeX kernel provides a large number of dedicated hooks (applying in a pre-defined location) and generic hooks (applying to arbitrary commands): the interface for using these is described in `lthooks`. There are also hooks to apply to files, described in `ltfilehooks`.

3 The structure of a class or package

The outline of a class or package file is:

Identification The file says that it is a LaTeX 2_ε package or class, and gives a short description of itself.

Preliminary declarations Here the file declares some commands and can also load other files. Usually these commands will be just those needed for the code used in the declared options.

Options The file declares and processes its options.

More declarations This is where the file does most of its work: declaring new variables, commands and fonts; and loading other files.

3.1 Identification

The first thing a class or package file does is identify itself. Package files do this as follows:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{<package>}[<date> <other information>]
```

For example:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{latexsym}[1998-08-17 Standard LaTeX package]
```

Class files do this as follows:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{<class-name>}[<date> <other information>]
```

For example:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{article}[2022-06-01 Standard LaTeX class]
```

The ⟨*date*⟩ should be given in the form 'YYYY-MM-DD' or 'YYYY/MM/DD' and must be present if the optional argument is used. Exactly four digits are required for the year and two each for the month and day. Where necessary, zeros should be added to pad the month and day appropriately. If digits or separators are missing, the date will likely be misinterpreted: the commands expect a valid syntax to speed up the routine usage of the package or class and make no provision for the case there is an error in the date specification.

This date is checked whenever a user specifies a date in their `\documentclass` or `\usepackage` command. For example, if you wrote:

```
\documentclass{article}[2022-06-01]
```

then users at a different location would get a warning that their copy of `article` was out of date.

The description of a class is displayed when the class is used. The description of a package is put into the log file. These descriptions are also displayed by the `\listfiles` command. The phrase `Standard LaTeX` **must not** be used in the identification banner of any file other than those in the standard LaTeX distribution.

3.2 Using classes and packages

A LaTeX package or class can load a package as follows:

```
\RequirePackage[<options>]{<package>}[<date>]
```

For example:

```
\RequirePackage{ifthen}[2022-06-01]
```

This command has the same syntax as the author command `\usepackage`. It allows packages or classes to use features provided by other packages. For example, by loading the `ifthen` package, a package writer can use the 'if... then... else...' commands provided by that package.

A LaTeX class can load one other class as follows:

```
\LoadClass[<options>]{<class-name>}[<date>]
```

For example:

```
\LoadClass[twocolumn]{article}
```

This command has the same syntax as the author command `\documentclass`. It allows classes to be based on the syntax and appearance of another class. For example, by loading the `article` class, a class writer only has to change the bits of `article` they don't like, rather than writing a new class from scratch.

The following commands can be used in the common case that you want to simply load a class or package file with exactly those options that are being used by the current class.

```
\LoadClassWithOptions{<class-name>}[<date>]
\RequirePackageWithOptions{<package>}[<date>]
```

For example:

```
\LoadClassWithOptions{article}
\RequirePackageWithOptions{graphics}[1995/12/01]
```

3.3 Declaring options

Packages and classes can declare options and these can be specified by authors; for example, the `twocolumn` option is declared by the `article` class. Note that the name of an option should contain only those characters allowed in a 'LaTeX name'; in particular it must not contain any control sequences.

LaTeX supports two methods for creating options: a key–value system and a 'simple text' approach. The key–value system is recommended for new classes

and packages, and is more flexible in handling of option classes than the simple text approach. Both option methods use the same basic structure within the LaTeX source: declaration of options first then processing options in a second step. Both also allow options to be passed on to other packages or an underlying class. As the 'classical' simple text approach is conceptually more straightforward to illustrate, it is used here to show the general structure: see Section 4.4 for full details of the key–value approach.

An option is declared as follows:

```
\DeclareOption{<option>}{<code>}
```

For example, the `dvips` option (slightly simplified) to the `graphics` package is implemented as:

```
\DeclareOption{dvips}{\input{dvips.def}}
```

This means that when an author writes `\usepackage[dvips]{graphics}`, the file `dvips.def` is loaded. As another example, the `a4paper` option is declared in the `article` class to set the `\paperheight` and `\paperwidth` lengths:

```
\DeclareOption{a4paper}{%
   \setlength{\paperheight}{297mm}%
   \setlength{\paperwidth}{210mm}%
}
```

Sometimes a user will request an option which the class or package has not explicitly declared. By default this will produce a warning (for classes) or error (for packages); this behavior can be altered as follows:

```
\DeclareOption*{<code>}
```

For example, to make the package `fred` produce a warning rather than an error for unknown options, you could specify:

```
\DeclareOption*{%
   \PackageWarning{fred}{Unknown option `\CurrentOption'}%
}
```

Then, if an author writes `\usepackage[foo]{fred}`, they will get a warning `Package fred Warning: Unknown option 'foo'.` As another example, the `fontenc` package tries to load a file `<ENC>enc.def` whenever the ⟨*ENC*⟩ option is used. This can be done by writing:

```
\DeclareOption*{%
   \input{\CurrentOption enc.def}%
}
```

It is possible to pass options on to another package or class, using the command `\PassOptionsToPackage` or `\PassOptionsToClass` (note that this is a specialized operation that works only for option names): see Section 4.5. For example, to pass every unknown option on to the `article` class, you can use:

```
\DeclareOption*{%
   \PassOptionsToClass{\CurrentOption}{article}%
}
```

If you do this then you should make sure you load the class at some later point, otherwise the options will never be processed!

So far, we have explained only how to declare options, not how to execute them. To process the options with which the file was called, you should use:

```
\ProcessOptions\relax
```

This executes the ⟨*code*⟩ for each option that was both specified and declared (see Section 7.4 for details of how this is done).

For example, if the `jane` package file contains:

```
\DeclareOption{foo}{\typeout{Saw foo.}}
\DeclareOption{baz}{\typeout{Saw baz.}}
\DeclareOption*{\typeout{What's \CurrentOption?}}
\ProcessOptions\relax
```

and an author writes `\usepackage[foo,bar]{jane}`, then they will see the messages `Saw foo.` and `What's bar?`

3.4 A minimal class file

Most of the work of a class or package is in defining new commands, or changing the appearance of documents. This is done in the body of the package, using commands such as `\newcommand` or `\setlength`.

There are four things that every class file *must* contain: these are a definition of `\normalsize`, values for `\textwidth` and `\textheight` and a specification for page-numbering. So a minimal document class file[1] looks like this:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{minimal}[2022-06-01 Standard LaTeX minimal class]
\renewcommand{\normalsize}{\fontsize{10pt}{12pt}\selectfont}
\setlength{\textwidth}{6.5in}
\setlength{\textheight}{8in}
\pagenumbering{arabic}       % needed even though this class will
                             % not show page numbers
```

However, this class file will not support footnotes, marginals, floats, etc., nor will it provide any of the 2-letter font commands such as `\rm`; thus most classes will contain more than this minimum!

[1] This class is now in the standard distribution, as `minimal.cls`.

3.5 Example: a local letter class

A company may have its own letter class, for setting letters in the company style. This section shows a simple implementation of such a class, although a real class would need more structure.

The class begins by announcing itself as `neplet.cls`.

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{neplet}[2022-06-01 NonExistent Press letter class]
```

Then this next bit passes any options on to the `letter` class, which is loaded with the `a4paper` option.

```
\DeclareOption*{\PassOptionsToClass{\CurrentOption}{letter}}
\ProcessOptions\relax
\LoadClass[a4paper]{letter}
```

In order to use the company letter head, it redefines the `firstpage` page style: this is the page style that is used on the first page of letters.

```
\renewcommand{\ps@firstpage}{%
   \renewcommand{\@oddhead}{<letterhead goes here>}%
   \renewcommand{\@oddfoot}{<letterfoot goes here>}%
}
```

And that's it!

3.6 Example: a newsletter class

A simple newsletter can be typeset with LaTeX, using a variant of the `article` class. The class begins by announcing itself as `smplnews.cls`.

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{smplnews}[2022-06-01 The Simple News newsletter class]

\newcommand{\headlinecolor}{\normalcolor}
```

It passes most specified options on to the `article` class: apart from the `onecolumn` option, which is switched off, and the `green` option, which sets the headline in green.

```
\DeclareOption{onecolumn}{\OptionNotUsed}
\DeclareOption{green}{\renewcommand{\headlinecolor}{\color{green}}}

\DeclareOption*{\PassOptionsToClass{\CurrentOption}{article}}

\ProcessOptions\relax
```

It then loads the class `article` with the option `twocolumn`.

```
\LoadClass[twocolumn]{article}
```

Since the newsletter is to be printed in color, it now loads the `color` package. The class does not specify a device driver option since this should be specified by the user of the `smplnews` class.

```
\RequirePackage{color}
```

The class then redefines `\maketitle` to produce the title in 72 pt Helvetica bold oblique, in the appropriate color.

```
\renewcommand{\maketitle}{%
   \twocolumn[%
      \fontsize{72}{80}\fontfamily{phv}\fontseries{b}%
      \fontshape{sl}\selectfont\headlinecolor
      \@title
   ]%
}
```

It redefines `\section` and switches off section numbering.

```
\renewcommand{\section}{%
   \@startsection
      {section}{1}{0pt}{-1.5ex plus -1ex minus -.2ex}%
      {1ex plus .2ex}{\large\sffamily\slshape\headlinecolor}%
}

\setcounter{secnumdepth}{0}
```

It also sets the three essential things.

```
\renewcommand{\normalsize}{\fontsize{9}{10}\selectfont}
\setlength{\textwidth}{17.5cm}
\setlength{\textheight}{25cm}
```

In practice, a class would need more than this: it would provide commands for issue numbers, authors of articles, page styles and so on; but this skeleton gives a start. The `ltnews` class file is not much more complex than this one.

4 Commands for class and package writers

This section describes briefly each of the commands for class and package writers. To find out about other aspects of the system, you should also read *LaTeX: A Document Preparation System*, *The LaTeX Companion* and *LaTeX 2ε for Authors*.

4.1 Identification

The first group of commands discussed here are those used to identify your class or package file.

`\NeedsTeXFormat {`⟨*format-name*⟩`} [`⟨*release-date*⟩`]`

This command tells TEX that this file should be processed using a format with name ⟨*format-name*⟩. You can use the optional argument ⟨*release-date*⟩ to further specify the earliest release date of the format that is needed. When the release date of the format is older than the one specified a warning will be generated. The standard ⟨*format-name*⟩ is `LaTeX2e`. The ⟨*release-date*⟩, if present, must be in the form 'YYYY-MM-DD' or 'YYYY/MM/DD'.

Example:

```
\NeedsTeXFormat{LaTeX2e}[2022-06-01]
```

People often don't know what date to put here. For the kernel, you can find out the right one by consulting `changes.txt` and select the release date of a new feature you are interested in. This is slightly different for packages as they are released throughout the year: you will need to consult their change history.

`\ProvidesClass {`⟨*class-name*⟩`} [`⟨*release-info*⟩`]`
`\ProvidesPackage {`⟨*package-name*⟩`} [`⟨*release-info*⟩`]`

This declares that the current file contains the definitions for the document class ⟨*class-name*⟩ or package ⟨*package-name*⟩.

The optional ⟨*release-info*⟩, if used, must contain:

- the release date of this version of the file, in the form YYYY-MM-DD;
- optionally followed by a space and a short description, possibly including a version number.

The above syntax must be followed exactly so that this information can be used by `\LoadClass` or `\documentclass` (for classes) or `\RequirePackage` or `\usepackage` (for packages) to test that the release is not too old.

The whole of this ⟨*release-info*⟩ information is displayed by `\listfiles` and should therefore not be too long.

Example:

```
\ProvidesClass{article}[2022-06-01 v1.0 Standard LaTeX class]
\ProvidesPackage{ifthen}[2022-06-01 v1.0 Standard LaTeX package]
```

`\ProvidesFile {`⟨*file-name*⟩`} [`⟨*release-info*⟩`]`

This is similar to the two previous commands except that here the full filename, including the extension, must be given. It is used for declaring any files other than main class and package files.

Example:

```
\ProvidesFile{T1enc.def}[2022-06-01 v1.0 Standard LaTeX file]
```

Note that the phrase `Standard LaTeX` **must not** be used in the identification banner of any file other than those in the standard LaTeX distribution.

4.2 Loading files

This group of commands can be used to create your own document class or package by building on existing classes or packages.

`\RequirePackage` [⟨*options-list*⟩] {⟨*package-name*⟩} [⟨*release-info*⟩]
`\RequirePackageWithOptions` {⟨*package-name*⟩} [⟨*release-info*⟩]

Packages and classes should use these commands to load other packages.

The use of `\RequirePackage` is the same as the author command `\usepackage`. Examples:

```
\RequirePackage{ifthen}[2022-06-01]
\RequirePackageWithOptions{graphics}[2022-06-01]
```

`\LoadClass` [⟨*options-list*⟩] {⟨*class-name*⟩} [⟨*release-info*⟩]
`\LoadClassWithOptions` {⟨*class-name*⟩} [⟨*release-info*⟩]

These commands are for use *only* in class files, they cannot be used in packages files; they can be used at most once within a class file.

The use of `\LoadClass` is the same as the use of `\documentclass` to load a class file.

Examples:

```
\LoadClass{article}[2022-06-01]
\LoadClassWithOptions{article}[2022-06-01]
```

The two `WithOptions` versions simply load the class (or package) file with exactly those options that are being used by the current file (class or package). See below, in 4.5, for further discussion of their use.

4.3 Delaying code

As noted earlier, a sophisticated hook system is available and described in `lthooks`. Here, we document a small set of convenient short names for common hooks.

These first two commands are also intended primarily for use within the ⟨*code*⟩ argument of `\DeclareOption` or `\DeclareOption*`.

`\AtEndOfClass {`⟨*code*⟩`}`
`\AtEndOfPackage {`⟨*code*⟩`}`

These commands declare ⟨*code*⟩ that is saved away internally and then executed after processing the whole of the current class or package file.

Repeated use of these commands is permitted: the code in the arguments is stored (and later executed) in the order of their declarations.

`\AtBeginDocument {`⟨*code*⟩`}`
`\AtEndDocument {`⟨*code*⟩`}`

These commands declare ⟨*code*⟩ to be saved internally and executed while LaTeX is executing `\begin{document}` or `\end{document}`.

The ⟨*code*⟩ specified in the argument to `\AtBeginDocument` is executed near the end of the `\begin{document}` code, *after* the font selection tables have been set up. It is therefore a useful place to put code which needs to be executed after everything has been prepared for typesetting and when the normal font for the document is the current font.

The `\AtBeginDocument` hook should not be used for code that does any typesetting since the typeset result would be unpredictable.

The ⟨*code*⟩ specified in the argument to `\AtEndDocument` is executed at the beginning of the `\end{document}` code, *before* the final page is finished and before any leftover floating environments are processed. If some of the ⟨*code*⟩ is to be executed after these two processes, you should include a `\clearpage` at the appropriate point in ⟨*code*⟩.

Repeated use of these commands is permitted: the code in the arguments is stored (and later executed) in the order of their declarations.

4.4 Creating and using keyval options

As with any key–value input, using key–value pairs as package or class options has two parts: creating the key options and setting (using) them. Options created in this way *may* be used after package loading as general key–value settings: this will depend on the nature of the underlying code.

In the following, the *family* for keys is a namespace used in creating the keys. At point of use, multiple families may be examined: see `\ProcessKeyOptions`.

`\DeclareKeys [`⟨*family*⟩`] {`⟨*declarations*⟩`}`

This command creates a series of options from a comma-separated ⟨*declarations*⟩ list. Each entry in this list is a key–value pair, with the ⟨*key*⟩ having one or more ⟨*properties*⟩. A small number of 'basic' ⟨*properties*⟩ are described below. The full range of properties, provided by `l3keys`, can also be used for more powerful processing. See `interface3` for the full details.

The basic properties provided here are

- `.code` — executes arbitrary code; within the code one can refer to the value of the key using `#1`
- `.if` — sets a TEX `\if...` switch
- `.ifnot` — sets an inverted TEX `\if...` switch
- `.pass-to-packages` — for class options, this specifies whether the option should be treated "global" (read by packages from the global list); for package options this property has no effect
- `.store` — stores a value in a macro
- `.usage` – defines whether the option can be given only when loading (`load`), in the preamble (`preamble`) or has no limitation on scope (`general`)

The part of the ⟨*key*⟩ before the ⟨*property*⟩ is the ⟨*name*⟩, with the ⟨*value*⟩ working with the ⟨*property*⟩ to define the behavior of the option.

For example, with

```
\DeclareKeys[mypkg]
  {
    draft.if          = @mypkg@draft       ,
    draft.usage       = preamble           ,
    name.store        = \@mypkg@name       ,
    name.usage        = load               ,
    second-name.store = \@mypkg@other@name ,
    head-color.code   = \renewcommand{\head@color}{\color{#1}} ,
    head-color.usage  = preamble
  }
```

four options would be created. The option `draft` can be given anywhere in the preamble, and will set a switch called `\if@mypkg@draft`. The option `name` can only be given during package loading, and will save whatever value it is given in `\@mypkg@name`. The option `second-name` can be given anywhere, and will save its value in `\@mypkg@other@name`. Finally, `head-color` redefines `\head@color` to use the color given as the key value.

Keys created *before* the use of `\ProcessKeyOptions` act as package options.

`\DeclareUnknownKeyHandler [`⟨*family*⟩`] {`⟨*code*⟩`}`

The command `\DeclareUnknownKeyHandler` may be used to define the behavior when an undefined key is encountered. The ⟨*code*⟩ will receive the unknown key name as `#1` and the value as `#2`. These can then be processed as appropriate, e.g. by forwarding to another package. The entire option is available as `\CurrentOption`, should it be necessary to pass on options which may or may not contain an = sign. For example, this may be used to pass an unknown option on to a non-keyval class such as article:

```
\DeclareUnknownKeyHandler{%
  \PassOptionsToClass{\CurrentOption}{article}
}
```

`\ProcessKeyOptions [`⟨*families*⟩`]`

The `\ProcessKeyOptions` function is used to check the current option list against the keys defined for ⟨*families*⟩ (a comma-separated list). Global (class) options and local (package) options are checked when this function is called in a package. The command will process *all* options given the current package or class: there is no need to also apply `\ProcessOptions`. `\ProcessKeyOptions` should appear *only once* in a class or package file.

`\SetKeys [`⟨*family*⟩`] {`⟨*keyvals*⟩`}`

Sets (applies) the explicit list of ⟨*keyvals*⟩ for the ⟨*family*⟩: if the latter is not given, the value of `\@currname` is used. This command may be used within a package to set options before or after using `\ProcessKeyOptions`.

Here is the example from 3.6, using the key-value approach:

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesClass{smplnews}[2022-06-01 The Simple News newsletter class]

\newcommand{\headlinecolor}{\normalcolor}

\DeclareKeys[samplnews]
  {
    onecolumn .code = {\OptionNotUsed}    ,
    green     .code = {\renewcommand{\headlinecolor}{\color{green}}}  ,
  }

\DeclareUnknownKeyHandler{%
  \PassOptionsToClass{\CurrentOption}{article}}

\ProcessKeyOptions

\LoadClass[twocolumn]{article}
...
```

4.5 Passing options around

These two commands are also very useful within the ⟨*code*⟩ argument of options.

`\PassOptionsToPackage {`⟨*options-list*⟩`} {`⟨*package-name*⟩`}`
`\PassOptionsToClass {`⟨*options-list*⟩`} {`⟨*class-name*⟩`}`

The command `\PassOptionsToPackage` passes the option names in ⟨*options-list*⟩ to package ⟨*package-name*⟩. This means that it adds the ⟨*option-list*⟩ to the list of options used by any future `\RequirePackage` or `\usepackage` command for package ⟨*package-name*⟩.

Example:

```
\PassOptionsToPackage{foo,bar}{fred}
\RequirePackage[baz]{fred}
```

is the same as:

```
\RequirePackage[foo,bar,baz]{fred}
```

Similarly, `\PassOptionsToClass` may be used in a class file to pass options to another class to be loaded with `\LoadClass`.

The effects and use of these two commands should be contrasted with those of the following two (documented above, in 4.2):

```
\LoadClassWithOptions
\RequirePackageWithOptions
```

The command `\RequirePackageWithOptions` is similar to `\RequirePackage`, but it always loads the required package with exactly the same option list as that being used by the current class or package, rather than with any option explicitly supplied or passed on by `\PassOptionsToPackage`.

The main purpose of `\LoadClassWithOptions` is to allow one class to simply build on another, for example:

```
\LoadClassWithOptions{article}
```

This should be compared with the slightly different construction

```
\DeclareOption*{\PassOptionsToClass{\CurrentOption}{article}}
\ProcessOptions\relax
\LoadClass{article}
```

or

```
\DeclareUnknownKeyHandler{\PassOptionsToClass{\CurrentOption}{article}}
\ProcessKeyOptions
\LoadClass{article}
```

As used above, the effects are more or less the same, but the first is a lot less to type; also the `\LoadClassWithOptions` method runs slightly quicker.

If, however, the class declares options of its own then the two constructions are different. Compare, for example:

```
\DeclareOption{landscape}{\@landscapetrue}
\ProcessOptions\relax
\LoadClassWithOptions{article}
```

or

```
\DeclareKeys[mypkg]
  {
    landscape.if    = @mypkg@landscape ,
    landscape.usage = load             ,
  }
\ProcessKeyOptions
\LoadClassWithOptions{article}
```

with:

```
\DeclareOption{landscape}{\@landscapetrue}
\DeclareOption*{\PassOptionsToClass{\CurrentOption}{article}}
\ProcessOptions\relax
\LoadClass{article}
```

or

```
\DeclareKeys[mypkg]
  {
    landscape.if    = @mypkg@landscape ,
    landscape.usage = load             ,
  }
\DeclareUnknownKeyHandler{\PassOptionsToClass{\CurrentOption}{article}}
\ProcessKeyOptions
\LoadClass{article}
```

In the first two, the article class will be loaded with option `landscape` precisely when the current class is called with this option. By contrast, in the second two it will never be called with option `landscape` as in that case article is passed options only by the default option handler, but this handler is not used for `landscape` because that option is explicitly declared.

4.6 Useful status tests

A number of status tests are available which can be used by package and class authors to query the presence and date of other code, the current kernel release and other related ideas. Three forms of each test are provided, one requiring arguments for both ⟨*true*⟩ and ⟨*false*⟩ branches, one requiring only a ⟨*true*⟩ branch and one requiring only a ⟨*false*⟩ branch. These are indicated in their names as `TF`, `T` and `F`, respectively. Here, we document only the `TF` versions: the other forms are however also available.

`\IfPackageLoadedTF {`⟨*package-name*⟩`} {`⟨*true code*⟩`} {`⟨*false code*⟩`}`
`\IfClassLoadedTF {`⟨*class-name*⟩`} {`⟨*true code*⟩`} {`⟨*false code*⟩`}`
`\IfFileLoadedTF {`⟨*file-name*⟩`} {`⟨*true code*⟩`} {`⟨*false code*⟩`}`

These commands test whether the named package, class or file has been loaded: this is done by using information that must be contained in an appropriate

`\Provides...` line, as described in Section 3.1. In the case of `\IfFileLoadedTF`, the full ⟨*file-name*⟩ must be provided; in contrast, no extension should be given if testing for a package or class.

`\IfPackageLoadedWithOptionsTF {`⟨*package-name*⟩`} {`⟨*options*⟩`} {`⟨*true code*⟩`} {`⟨*false code*⟩`}`
`\IfClassLoadedWithOptionsTF {`⟨*class-name*⟩`} {`⟨*options*⟩`} {`⟨*true code*⟩`} {`⟨*false code*⟩`}`

These commands test whether the named package or class has been loaded with *exactly* the ⟨*options*⟩ specified. In order to take the ⟨*true*⟩ branch, the package or class must be loaded (giving a ⟨*true*⟩ result for `\If...LoadedT`) and the option list used when loading it must be identical to the ⟨*options*⟩.

`\IfPackageAtLeastTF {`⟨*package-name*⟩`} {`⟨*date*⟩`} {`⟨*true code*⟩`} {`⟨*false code*⟩`}`
`\IfClassAtLeastTF {`⟨*class-name*⟩`} {`⟨*date*⟩`} {`⟨*true code*⟩`} {`⟨*false code*⟩`}`
`\IfFileAtLeastTF {`⟨*file-name*⟩`} {`⟨*date*⟩`} {`⟨*true code*⟩`} {`⟨*false code*⟩`}`

These tests are used to check whether the date information given in the `\Provides...` line of a package, class or file is no earlier than the given ⟨*date*⟩. The ⟨*date*⟩ is compared with that in the optional argument to `\Provides...` (as described in Section 3.1); if `\Provides...` was missing or had no optional argument, it is treated as `0000/00/00` (i.e. earlier than any other date). As for `\Provides...`, the ⟨*date*⟩ should be given in the form 'YYYY-MM-DD' or 'YYYY/MM/DD'. If the package, class or file is not loaded, the ⟨*false*⟩ branch will be taken; in contrast, if the ⟨*date*⟩ is not given in the required form, the behavior is formally undefined.

`\IfFormatAtLeastTF {`⟨*date*⟩`} {`⟨*true code*⟩`} {`⟨*false code*⟩`}`

Tests the release ⟨*date*⟩ of the LaTeX format and selects the appropriate branch. The date used by the format is adjusted to incorporate any roll forward or back that has been applied to it, so that the apparent date of the format will be that after roll forward or back.

4.7 Safe file commands

These commands deal with file input; they ensure that the non-existence of a requested file can be handled in a user-friendly way.

`\IfFileExists {`⟨*file-name*⟩`} {`⟨*true*⟩`} {`⟨*false*⟩`}`

If the file exists then the code specified in ⟨*true*⟩ is executed.

If the file does not exist then the code specified in ⟨*false*⟩ is executed.

This command does *not* input the file.

`\InputIfFileExists {`⟨*file-name*⟩`} {`⟨*true*⟩`} {`⟨*false*⟩`}`

This inputs the file ⟨*file-name*⟩ if it exists and, immediately before the input, the code specified in ⟨*true*⟩ is executed.

If the file does not exist then the code specified in ⟨*false*⟩ is executed.

It is implemented using `\IfFileExists`.

4.8 Reporting errors, etc

These commands should be used by third party classes and packages to report errors, or to provide information to authors.

`\ClassError {`⟨*class-name*⟩`} {`⟨*error-text*⟩`} {`⟨*help-text*⟩`}`
`\PackageError {`⟨*package-name*⟩`} {`⟨*error-text*⟩`} {`⟨*help-text*⟩`}`

These produce an error message. The ⟨*error-text*⟩ is displayed and the `?` error prompt is shown. If the user types `h`, they will be shown the ⟨*help-text*⟩.

Within the ⟨*error-text*⟩ and ⟨*help-text*⟩: `\protect` can be used to stop a command from expanding; `\MessageBreak` causes a line-break; and `\space` prints a space.

Note that the ⟨*error-text*⟩ will have a full stop added to it, so do not put one into the argument.

For example:

```
\newcommand{\foo}{FOO}
\PackageError{ethel}{%
   Your hovercraft is full of eels,\MessageBreak
   and \protect\foo\space is \foo
}{%
   Oh dear! Something's gone wrong.\MessageBreak
   \space \space Try typing \space «return»
   \space to proceed, ignoring \protect\foo.
}
```

produces this display:

```
! Package ethel Error: Your hovercraft is full of eels,
(ethel)                and \foo is FOO.

See the ethel package documentation for explanation.
```

If the user types `h`, this will be shown:

```
Oh dear! Something's gone wrong.
  Try typing  «return»  to proceed, ignoring \foo.
```

```
\ClassWarning {⟨class-name⟩} {⟨warning-text⟩}
\PackageWarning {⟨package-name⟩} {⟨warning-text⟩}
\ClassWarningNoLine {⟨class-name⟩} {⟨warning-text⟩}
\PackageWarningNoLine {⟨package-name⟩} {⟨warning-text⟩}
\ClassInfo {⟨class-name⟩} {⟨info-text⟩}
\PackageInfo {⟨package-name⟩} {⟨info-text⟩}
```

The four `Warning` commands are similar to the error commands, except that they produce only a warning on the screen, with no error prompt.

The first two, `Warning` versions, also show the line number where the warning occurred, whilst the second two, `WarningNoLine` versions, do not.

The two `Info` commands are similar except that they log the information only in the transcript file, including the line number. There are no `NoLine` versions of these two.

Within the ⟨*warning-text*⟩ and ⟨*info-text*⟩: `\protect` can be used to stop a command from expanding; `\MessageBreak` causes a line-break; and `\space` prints a space. Also, these should not end with a full stop as one is automatically added.

5 Miscellaneous commands, etc.

5.1 Layout parameters

```
\paperheight
\paperwidth
```

These two parameters are usually set by the class to be the size of the paper being used. This should be actual paper size, unlike `\textwidth` and `\textheight` which are the size of the main text body within the margins.

5.2 Case changing

```
\MakeUppercase [⟨keyvals⟩] {⟨text⟩}
\MakeLowercase [⟨keyvals⟩] {⟨text⟩}
\MakeTitlecase [⟨keyvals⟩] {⟨text⟩}
```

As described in `usrguide`, case changing for text should be carried out using the commands `\MakeUppercase`, `\MakeLowercase` and `\MakeTitlecase`. If you need to change the case of programmatic material, the team strongly suggest using the L3 programming layer commands in the `l3str` module. If you do not wish to do this, you should use the TEX `\uppercase` and `\lowercase` primitives *in this situation only.*

5.3 Better user-defined math display environments

`\ignorespacesafterend`

Suppose that you want to define an environment for displaying text that is numbered as an equation. A straightforward way to do this is as follows:

```
\newenvironment{texteqn}{%
  \begin{equation}%
    \begin{minipage}{0.9\linewidth}%
}{%
    \end{minipage}%
  \end{equation}%
}
```

However, if you have tried this then you will probably have noticed that it does not work perfectly when used in the middle of a paragraph because an inter-word space appears at the beginning of the first line after the environment.

You can avoid this problem using `\ignorespacesafterend`; it should be inserted as shown here:

```
\newenvironment{texteqn}{%
  \begin{equation}%
    \begin{minipage}{0.9\linewidth}%
}{%
    \end{minipage}%
  \end{equation}%
  \ignorespacesafterend
}
```

This command may also have other uses.

5.4 Normalising spacing

`\normalsfcodes`

This command should be used to restore the normal settings of the parameters that affect spacing between words, sentences, etc.

An important use of this feature is to correct a problem, reported by Donald Arseneau, that punctuation in page headers has always (in all known TEX formats) been potentially wrong whenever a page break happens while a local setting of the space codes is in effect. These space codes are changed by, for example, the command `\frenchspacing`) and the verbatim environment.

It is normally given the correct definition automatically in `\begin{document}` and so need not be explicitly set; however, if it is explicitly made non-empty in a class file then automatic default setting will be over-ridden.

5.5 Querying localisation

Localisation information is needed to customise a range of outputs. The LaTeX kernel does not itself manage localisation, which is well-served by the bundles babel and polyglossia. To allow the kernel and other packages to access the current localisation information provided by babel or polyglossia, the command `\BCPdata` is defined by the kernel. The initial kernel definition expands to tag parts for `en-US`, as the kernel does not track localisation but does start out with a broadly US English setup. However, if babel or polyglossia are loaded, it is redefined expand to the BCP-47 information from the appropriate package. The supported arguments are the BCP-47 tag breakdowns:

- `tag` The full BCP-47 tag (e.g. `en-US`)
- `language` (e.g., `de`)
- `region` (e.g., `AT`)
- `script` (e.g., `Latn`)
- `variant` (e.g., `1901`)
- `extension.t` (transformation, e.g., `en-t-ja`)
- `extension.u` (additional locale information, e.g., `ar-u-nu-latn`)
- `extension.x` (private use area, e.g., `la-x-classic`)

The information for the *main* language for a document is be provided if these are prefixed by `main.`, e.g. `main.language` will expand to the main language even if another language is currently active.

In addition to the tag breakdown, the following semantic arguments are supported

- `casing` The tag for case changing, e.g. `el-x-iota` could be selected rather than `el` to select a capital adscript iota on uppercasing an *ypogegrammeni*

For example, the case changing command `\MakeUppercase` is (conceptually) defined as

```
\ExpandArgs{e}\MakeUppercaseAux{\BCPdata{casing}}{#1}
```

where `#1` is the user input and the first argument to `\MakeUppercaseAux` takes two arguments, the locale and input text.

5.6 Extended and expandable references of properties

A property is something that LaTeX can track while processing the document, such as a page number, a heading number, other counter values, a heading title, a position on the page, etc. The current value of such properties can be labeled and written to the `aux`-file. It can then be referenced in the next compilation, similar to the way the standard `\label`/`\ref` commands work (they record/reference a fixed set of properties: label, page, title, and target).

`\RecordProperties{`⟨*label*⟩`}{`⟨*list of properties*⟩`}`

This command writes the value(s) of the ⟨*list of properties*⟩ `aux`-file labeled by ⟨*label*⟩. Recorded are either the values current when `\RecordProperties` is called or the value current when the next shipout happens—which depends on the declaration for each property. The arguments ⟨*label*⟩ and ⟨*list of properties*⟩ can contain commands that are expanded. ⟨*label*⟩ can expand to an arbitrary string (as long as it can safely be written to the `aux`-file) but note that the label names of `\label` and `\RecordProperties` share a single namespace. This means that you get a `Label 'A' multiply defined` warning with the following code:

```
\label{A}\RecordProperties{A}{abspage}
```

`\RefProperty{`⟨*label*⟩`}{`⟨*property*⟩`}`

This command allows to reference the value of the property ⟨*property*⟩ recorded in the previous run and labeled by ⟨*label*⟩. Differently to the standard `\ref` command the command is expandable and the value can for example—if it is a number—be used in an assignment.[2]

```
\section{A section}
\RecordProperties{mylabel}{pagenum,counter}
\RefProperty{mylabel}{counter} % outputs section
\setcounter{mycounter}{\RefProperty{mylabel}{pagenum}}
```

As `\RefProperty` is expandable it can not issue a rerun warning if a label is not found. If needed such a warning can be forced by the following command:

`\RefUndefinedWarn{`⟨*label*⟩`}{`⟨*property*⟩`}`

LaTeX predefines a set of properties, this set contains also the properties stored by the standard `\label` command. In the list below "default" indicates the value returned when the value is not yet known (i.e., if it wasn't recorded in the previous run and "at shipout" means that this property is not recorded immediately when `\RecordProperties` is used but during the next `\shipout`.

`abspage` **(default: `0`, at shipout)** The absolute value of the current page: starts at 1 and increases monotonically at each shipout.

[2] For this to work the default value for the property would need to be a number too, because recorded values aren't known in the first LaTeX run.

`page` **(default: `0`, at shipout)** The current page as given by `\thepage`: this may or may not be a numerical value, depending on the current style. Contrast with `abspage`. You get this value also with the standard `\label`/`\pageref`.

`pagenum` **(default: `0`, at shipout)** The current page as arabic number. This is suitable for integer operations and comparisons.

`label` **(default: `??`)** The content of `\@currentlabel`. This is the value that you get also with the standard `\label`/`\ref`.

`title` **(default: `\textbf{??}`)** The content of `\@currentlabelname`. This command is filled beside others by the nameref package and some classes (e.g. memoir) and typically gives the title defined in the document by some sectioning command

`target` **(default: ⟨*empty*⟩)** The content of `\@currentHref`. This command is normally filled by hyperref and holds the name of the last destination it created.

`pagetarget` **(default: ⟨*empty*⟩, at shipout)** The content of `\@currentHpage`. This command is filled by hyperref (version v7.01c or newer) and holds the name of the last page anchor it created.

`counter` **(default: ⟨*empty*⟩)** The content of `\@currentcounter`. This command contains after a `\refstepcounter` the name of the counter.

`xpos`, `ypos` **(default: `0`, at shipout)** These properties records the x and y coordinates of a point previously stored with `\pdfsavepos`/`\savepos`. E.g. (if bidi is used it can be necessary to save the position before and after the label):

```
\pdfsavepos
\RecordProperties{myposition}{xpos,ypos}%
\pdfsavepos
```

Class and package authors can define more properties to store other values they are interested in.

`\NewProperty{`⟨*name*⟩`}{`⟨*setpoint*⟩`}{`⟨*default*⟩`}{`⟨*code*⟩`}`
`\SetProperty{`⟨*name*⟩`}{`⟨*setpoint*⟩`}{`⟨*default*⟩`}{`⟨*code*⟩`}`

These commands declare or change a property ⟨*name*⟩[3]. If a new property is declared within a package it is suggested that its name is always structured as follows: ⟨*package-name*⟩/⟨*property-name*⟩. ⟨*setpoint*⟩ is either `now` or `shipout` and decides if the value is written directly or at the next shipout. ⟨*default*⟩ is used if the property is referenced but not yet known, e.g., in the first run. ⟨*code*⟩ is the code executed when storing the value. For example, the `pagenum` property is declared as

[3]Only change properties that you have declared. The declarations of standard properties of LaTeX and properties of other packages should never be altered!

```
\NewProperty{pagenum}{shipout}{0}{\the\value{page}}
```

The commands related to properties are offered as a set of CamelCase commands for traditional LaTeX 2ε packages (and for use in the document preamble if needed) as well as `expl3` commands for modern packages, that use the L3 programming layer of LaTeX. The `expl3` commands and more details can be found in `ltproperties-doc.pdf`.

5.6.1 Templates (prototype document commands)

Templates as defined by LaTeX are a mechanism to cleanly separate the three layers needed for writing a document

1. authoring of the text with mark-up;
2. document layout design;
3. implementation (with TeX programming) of the design.

They allow document authors to modify design without altering code, and allow programmers to make portable changes to classes.

Implementing this mechanism requires a number of steps and a family of commands which allow variation in outcomes. A typical use of templates will make use of most or all of `\NewTemplateType`, `\DeclareTemplateInterface`, `\DeclareTemplateCode`, `\DeclareInstance` and `\UseInstance`, plus potentially some more specialized commands. These are described in `lttemplates-doc` in full detail.

5.7 Preparing link targets

Active links in a document need targets to which they can jump to. Such targets are often created automatically (if the package hyperref is loaded) by the `\refstepcounter` command but there are also cases where class or package authors need to add a target manually, for example, in unnumbered sectioning commands or in environments. For this LaTeX provides the following commands. *Without* hyperref they do nothing or insert only a whatsits (to ensure that spacing doesn't change when hyperref is loaded), *with* hyperref they add the necessary targets. Details about the behavior and the arguments of the following commands can by found in the hyperref package in `hyperref-linktarget.pdf`.

```
\MakeLinkTarget[⟨prefix⟩]{⟨counter⟩}
\MakeLinkTarget[⟨prefix⟩]{}
\MakeLinkTarget*{⟨target name⟩}
```

This command prepares the creations of targets.

`\LinkTargetOn`
`\LinkTargetOff`

These commands allow to enable and disable locally the creation of targets. This can be useful to suppress targets otherwise created automatically by `\refstepcounter`.

`\NextLinkTarget{`⟨*target name*⟩`}`

This changes the name of the next target that will be created.

5.8 Querying document command argument specifications

There are some specialized cases in which a package author may need to query the argument specification of a command. When commands are created using `\NewDocumentCommand`, etc., this is possible using the *code* level function `\\detokenize {cmd_arg_spec:N}`. The function will *expand* to the argument specification if available, or `X` if this is not available. The latter case is true for undefined commands and those created using `\newcommand`, `\DeclareRobustCommand` or `\def`.

6 Configuring the Output routine

For nearly 40 years LaTeX's output routine (the mechanism to paginate the document and attach footnotes, floats and headers & footers) was a largely hardwired algorithm with a limited number of configuration possibilities. Packages or classes that attempted to alter one or the other aspect of the process had to overwrite the internals with the usual problems: incompatibilities and out of date code whenever something was changed in LaTeX.

To improve this situation and to support the production of accessible PDF documents we started in 2024 to refactor the output routine and have added a number of hooks and sockets, so that packages that want to adjust the output routine can do so safely without the dangers associated with that in the past.

For packages, we implemented the following hooks:

`build/page/before`, `build/page/after` These two hooks enable packages to prepend or append code to the page processing in the output routine. They are implemented as mirrored hooks.

Technically, they are executed at the start and the end of the internal LaTeX 2_ε `\@outputpage` command, respectively. A number of packages alter that command to place code in exactly these two places—they can now simply add their code to the hooks instead.

`build/page/reset` Packages that set up special conventions for text in the main galley (such as catcode changes, etc.) can use this hook to undo these changes within the output routine, so that they aren't applied to unrelated material, e.g., the text for running header or footers.

`build/column/before`, `build/column/after` These two hooks enable packages to prepend or append code to the column processing in the output routine. They are implemented as mirrored hooks.

Technically, they are executed at the start and the end of the internal LaTeX 2_ε `\@makecol` command, respectively. A number of packages alter `\@makecol` to place code in exactly these two places—they can now simply add their code to the hooks instead.

We also added a number of sockets for configuring the algorithm and to support tagging. One socket that is of interest for class files but also for user in the document preamble is `build/column/outputbox`. It defines how the column text, the column floats (top and bottom) and the footnotes are combined, i.e. their order and spacing. To change the layout all one has to do is to assign a different predefined plug to the socket with

`\AssignSocketPlug{build/column/outputbox}{`⟨*plug-name*⟩`}`

The predeclared plugs are the following:

`space-footnotes-floats` After the galley text there is a vertical `\vfill` followed by the footnotes, followed by the bottom floats, if any.

`footnotes-space-floats` As before but the `\vfill` is between footnotes and floats.

`floats-space-footnotes` Floats are directly after the text, then a space and then footnotes at the bottom.

`space-floats-footnotes` Both floats and footnotes are pushed to the bottom with footnotes coming last.[4]

`floats-footnotes` All excess space is distributed across the existing glue on the page, e.g., within the text galley, the separation between blocks, etc. The order is text, floats, footnotes.

`footnotes-floats` As the previous one but floats and footnotes are swapped. This is the LaTeX default for newer documents, i.e., this plug is assigned to the socket when `\DocumentMetadata` is used.

`footnotes-floats-legacy` As the previous one but LaTeX's bottom skip bug is not corrected, i.e., in ragged bottom designs where footnotes are supposed to be directly attached to the text, they suddenly appear at the bottom of the page when the page is ended with `\newpage` or `\clearpage`. While this is clearly a bug, it was the case since the days of LaTeX 2.09; thus for compatibility we continue to support this behavior.

There are a few more configuration possibilities but they are so specialized that we have only documented them in the LaTeX source in chapter 54 (which can be viewed with `texdoc source2e`).

[4] There are two more permutations, but neither of them has ever been requested so they aren't set up by default — doing that in a class would be trivial though.

7 Commands superseded for new material

A small number of commands were introduced as part of LaTeX 2ε in the mid-1990s, are widely used but have been superseded by more modern methods. These are covered here as they are likely to be encountered routinely in existing classes and packages.

7.1 Defining commands

The *-forms of these commands should be used to define commands that are not, in TeX terms, long. This can be useful for error-trapping with commands whose arguments are not intended to contain whole paragraphs of text.

`\DeclareRobustCommand` {⟨*cmd*⟩} [⟨*num*⟩] [⟨*default*⟩] {⟨*definition*⟩}
`\DeclareRobustCommand*` {⟨*cmd*⟩} [⟨*num*⟩] [⟨*default*⟩] {⟨*definition*⟩}

This command takes the same arguments as `\newcommand` but it declares a robust command, even if some code within the ⟨*definition*⟩ is fragile. You can use this command to define new robust commands, or to redefine existing commands and make them robust. A log is put into the transcript file if a command is redefined.

For example, if `\seq` is defined as follows:

```
\DeclareRobustCommand{\seq}[2][n]{%
  \ifmmode
    #1_{1}\ldots#1_{#2}%
  \else
    \PackageWarning{fred}{You can't use \protect\seq\space in text}%
  \fi
}
```

Then the command `\seq` can be used in moving arguments, even though `\ifmmode` cannot, for example:

```
\section{Stuff about sequences $\seq{x}$}
```

Note also that there is no need to put a `\relax` before the `\ifmmode` at the beginning of the definition; this is because the protection given by this `\relax` against expansion at the wrong time will be provided internally.

`\CheckCommand` {⟨*cmd*⟩} [⟨*num*⟩] [⟨*default*⟩] {⟨*definition*⟩}
`\CheckCommand*` {⟨*cmd*⟩} [⟨*num*⟩] [⟨*default*⟩] {⟨*definition*⟩}

This takes the same arguments as `\newcommand` but, rather than define ⟨*cmd*⟩, it just checks that the current definition of ⟨*cmd*⟩ is exactly as given by ⟨*definition*⟩. An error is raised if these definitions differ.

This command is useful for checking the state of the system before your package starts altering the definitions of commands. It allows you to check, in particular, that no other package has redefined the same command.

7.2 Option declaration

The following commands deal with the declaration and handling of options to document classes and packages using a classical 'simple text' approach. Every option name must be a 'LaTeX name'.

There are some commands designed especially for use within the ⟨*code*⟩ argument of these commands (see below).

`\DeclareOption {`⟨*option-name*⟩`} {`⟨*code*⟩`}`

This makes ⟨*option-name*⟩ a 'declared option' of the class or package in which it is put.

The ⟨*code*⟩ argument contains the code to be executed if that option is specified for the class or package; it can contain any valid LaTeX 2_ε construct.

Example:

```
\DeclareOption{twoside}{\@twosidetrue}
```

`\DeclareOption* {`⟨*code*⟩`}`

This declares the ⟨*code*⟩ to be executed for every option which is specified for, but otherwise not explicitly declared by, the class or package; this code is called the 'default option code' and it can contain any valid LaTeX 2_ε construct.

If a class file contains no `\DeclareOption*` then, by default, all specified but undeclared options for that class will be silently passed to all packages (as will the specified and declared options for that class).

If a package file contains no `\DeclareOption*` then, by default, each specified but undeclared option for that package will produce an error.

7.3 Commands within option code

These two commands can be used only within the ⟨*code*⟩ argument of either `\DeclareOption` or `\DeclareOption*`. Other commands commonly used within these arguments can be found in the next few subsections.

`\CurrentOption`

This expands to the name of the current option.

`\OptionNotUsed`

This causes the current option to be added to the list of 'unused options'.

7.4 Option processing

`\ProcessOptions`

This command executes the ⟨*code*⟩ for each selected option.

We shall first describe how `\ProcessOptions` works in a package file, and then how this differs in a class file.

To understand in detail what `\ProcessOptions` does in a package file, you have to know the difference between *local* and *global* options.

- **Local options** are those which have been explicitly specified for this particular package in the ⟨*options*⟩ argument of any of these:

  ```
  \PassOptionsToPackage{<options>} \usepackage[<options>]
  \RequirePackage[<options>]
  ```

- **Global options** are any other options that are specified by the author in the ⟨*options*⟩ argument of `\documentclass[<options>]`.

For example, suppose that a document begins:

```
\documentclass[german,twocolumn]{article}
\usepackage{gerhardt}
```

whilst package `gerhardt` calls package `fred` with:

```
\PassOptionsToPackage{german,dvips,a4paper}{fred}
\RequirePackage[errorshow]{fred}
```

then:

- `fred`'s local options are `german`, `dvips`, `a4paper` and `errorshow`;
- `fred`'s only global option is `twocolumn`.

When `\ProcessOptions` is called, the following happen.

- *First*, for each option so far declared in `fred.sty` by `\DeclareOption`, it looks to see if that option is either a global or a local option for `fred`: if it is then the corresponding code is executed.

  This is done in the order in which these options were declared in `fred.sty`.

- *Then*, for each remaining *local* option, the command `\ds@<option>` is executed if it has been defined somewhere (other than by a `\DeclareOption`); otherwise, the 'default option code' is executed. If no default option code has been declared then an error message is produced.

  This is done in the order in which these options were specified.

Throughout this process, the system ensures that the code declared for an option is executed at most once.

Returning to the example, if `fred.sty` contains:

```
\DeclareOption{dvips}{\typeout{DVIPS}}
\DeclareOption{german}{\typeout{GERMAN}}
\DeclareOption{french}{\typeout{FRENCH}}
\DeclareOption*{\PackageWarning{fred}{Unknown `\CurrentOption'}}
\ProcessOptions\relax
```

then the result of processing this document will be:

```
DVIPS
GERMAN
Package fred Warning: Unknown `a4paper'.
Package fred Warning: Unknown `errorshow'.
```

Note the following:

- the code for the `dvips` option is executed before that for the `german` option, because that is the order in which they are declared in `fred.sty`;
- the code for the `german` option is executed only once, when the declared options are being processed;
- the `a4paper` and `errorshow` options produce the warning from the code declared by `\DeclareOption*` (in the order in which they were specified), whilst the `twocolumn` option does not: this is because `twocolumn` is a global option.

In a class file, `\ProcessOptions` works in the same way, except that: *all* options are local; and the default value for `\DeclareOption*` is `\OptionNotUsed` rather than an error.

Note that, because `\ProcessOptions` has a `*`-form, it is wise to follow the non-star form with `\relax`, as in the previous examples, since this prevents unnecessary look ahead and possibly misleading error messages being issued.

`\ProcessOptions*`

This is like `\ProcessOptions` but it executes the options in the order specified in the calling commands, rather than in the order of declaration in the class or package. For a package this means that the global options are processed first.

`\ExecuteOptions {`⟨*options-list*⟩`}`

It can be used to provide a 'default option list' just before `\ProcessOptions`. For example, suppose that in a class file you want to set up the default design to be: two-sided printing; 11pt fonts; in two columns. Then it could specify:

```
\ExecuteOptions{11pt,twoside,twocolumn}
```