NAME

MooseX::Declare::Context − Per−keyword declaration context

VERSION

version 0.43

DESCRIPTION

This is not a subclass of Devel::Declare::Context::Simple, but it will delegate all default methods and extend it with some attributes and methods of its own.

A context object will be instantiated for every keyword that is handled by MooseX::Declare. If handlers want to communicate with other handlers (for example handlers that will only be setup inside a namespace block) it must do this via the generated code.

In addition to all the methods documented here, all methods from Devel::Declare::Context::Simple are available and will be delegated to an internally stored instance of it.

ATTRIBUTES

caller_file

A required `Str` containing the file the keyword was encountered in.

preamble_code_parts

An `ArrayRef` of ‘‘CodePart’’s that will be used as preamble. A preamble in this context means the beginning of the generated code.

scope_code_parts

These parts will come before the actual body and after the ‘‘preamble_code_parts’’. It is an `ArrayRef` of ‘‘CodePart’’s.

cleanup_code_parts

An `ArrayRef` of ‘‘CodePart’’s that will not be directly inserted into the code, but instead be installed in a handler that will run at the end of the scope so you can do namespace cleanups and such.

stack

An `ArrayRef` that contains the stack of handlers. A keyword that was only setup inside a scoped block will have the blockhandler be put in the stack.

METHODS

add_preamble_code_parts(CodePart `@parts`)

```
Object->add_preamble_code_parts (CodeRef @parts)
```

See ‘‘add_cleanup_code_parts’’.

add_scope_code_parts(CodePart `@parts`)

```
Object->add_scope_code_parts    (CodeRef @parts)
```

See ‘‘add_cleanup_code_parts’’.

add_cleanup_code_parts(CodePart `@parts`)

```
Object->add_cleanup_code_parts  (CodeRef @parts)
```

For these three methods please look at the corresponding `*_code_parts` attribute in the list above. These methods are merely convenience methods that allow adding entries to the code part containers.

inject_code_parts_here

```
True Object->inject_code_parts_here (CodePart @parts)
```

Will inject the passed ‘‘CodePart’’s at the current position in the code.

peek_next_char

```
Str Object->peek_next_char ()
```

Will return the next char without stripping it from the stream.

inject_code_parts

```
Object->inject_code_parts (
    Bool    :$inject_cleanup_code_parts,
    CodeRef :$missing_block_handler
)
```

This will inject the code parts from the attributes above at the current position. The preamble and scope code parts will be inserted first. Then then call to the cleanup code will be injected, unless the options contain a key named `inject_cleanup_code_parts` with a false value.

The `inject_if_block` method will be called if the next char is a `{` indicating a following block.

If it is not a block, but a semi-colon is found and the options contained a `missing_block_handler` key was passed, it will be called as method on the context object with the code to inject and the options as arguments. All options that are not recognized are passed through to the `missing_block_handler`. You are well advised to prefix option names in your extensions.

TYPES

BlockCodePart

An `ArrayRef` with at least one element that stringifies to either `BEGIN` or `END`. The other parts will be stringified and used as the body for the generated block. An example would be this compiletime role composition:

```
['BEGIN', 'with q{ MyRole }']
```

CodePart

A part of code represented by either a `Str` or a ‘‘BlockCodePart’’.

SEE ALSO

- MooseX::Declare
- Devel::Declare
- Devel::Declare::Context::Simple

AUTHOR

Florian Ragwitz <rafl AT debian DOT org>

COPYRIGHT AND LICENSE