## NAME

Params::Util − Simple, compact and correct param−checking functions

## SYNOPSIS

```
# Import some functions
use Params::Util qw{_SCALAR _HASH _INSTANCE};

# If you are lazy, or need a lot of them...
use Params::Util ':ALL';

sub foo {
    my $object  = _INSTANCE(shift, 'Foo') or return undef;
    my $image   = _SCALAR(shift)          or return undef;
    my $options = _HASH(shift)            or return undef;
    # etc...
}
```

## DESCRIPTION

`Params::Util` provides a basic set of importable functions that makes checking parameters a hell of a lot easier

While they can be (and are) used in other contexts, the main point behind this module is that the functions **both** Do What You Mean, and Do The Right Thing, so they are most useful when you are getting params passed into your code from someone and/or somewhere else and you can't really trust the quality.

Thus, `Params::Util` is of most use at the edges of your API, where params and data are coming in from outside your code.

The functions provided by `Params::Util` check in the most strictly correct manner known, are documented as thoroughly as possible so their exact behaviour is clear, and heavily tested so make sure they are not fooled by weird data and Really Bad Things.

To use, simply load the module providing the functions you want to use as arguments (as shown in the SYNOPSIS).

To aid in maintainability, `Params::Util` will **never** export by default.

You must explicitly name the functions you want to export, or use the `:ALL` param to just have it export everything (although this is not recommended if you have any _FOO functions yourself with which future additions to `Params::Util` may clash)

## FUNCTIONS

### _STRING `$string`

The `_STRING` function is intended to be imported into your package, and provides a convenient way to test to see if a value is a normal non-false string of non-zero length.

Note that this will NOT do anything magic to deal with the special `'0'` false negative case, but will return it.

```
# '0' not considered valid data
my $name = _STRING(shift) or die "Bad name";

# '0' is considered valid data
my $string = _STRING($_[0]) ? shift : die "Bad string";
```

Please also note that this function expects a normal string. It does not support overloading or other magic techniques to get a string.

Returns the string as a conveince if it is a valid string, or `undef` if not.

### _IDENTIFIER `$string`

The `_IDENTIFIER` function is intended to be imported into your package, and provides a convenient way to test to see if a value is a string that is a valid Perl identifier.

Returns the string as a convenience if it is a valid identifier, or `undef` if not.

**_CLASS** `$string`

The `_CLASS` function is intended to be imported into your package, and provides a convenient way to test to see if a value is a string that is a valid Perl class.

This function only checks that the format is valid, not that the class is actually loaded. It also assumes ''normalised'' form, and does not accept class names such as `::Foo` or `D'Oh`.

Returns the string as a convenience if it is a valid class name, or `undef` if not.

**_CLASSISA** `$string`**,** `$class`

The `_CLASSISA` function is intended to be imported into your package, and provides a convenient way to test to see if a value is a string that is a particularly class, or a subclass of it.

This function checks that the format is valid and calls the `->isa` method on the class name. It does not check that the class is actually loaded.

It also assumes ''normalised'' form, and does not accept class names such as `::Foo` or `D'Oh`.

Returns the string as a convenience if it is a valid class name, or `undef` if not.

**_CLASSDOES** `$string`**,** `$role`

This routine behaves exactly like `"_CLASSISA"`, but checks with `->DOES` rather than `->isa`. This is probably only a good idea to use on Perl 5.10 or later, when UNIVERSAL::DOES has been implemented.

**_SUBCLASS** `$string`**,** `$class`

The `_SUBCLASS` function is intended to be imported into your package, and provides a convenient way to test to see if a value is a string that is a subclass of a specified class.

This function checks that the format is valid and calls the `->isa` method on the class name. It does not check that the class is actually loaded.

It also assumes ''normalised'' form, and does not accept class names such as `::Foo` or `D'Oh`.

Returns the string as a convenience if it is a valid class name, or `undef` if not.

**_NUMBER** `$scalar`

The `_NUMBER` function is intended to be imported into your package, and provides a convenient way to test to see if a value is a number. That is, it is defined and perl thinks it's a number.

This function is basically a Params::Util-style wrapper around the Scalar::Util `looks_like_number` function.

Returns the value as a convience, or `undef` if the value is not a number.

**_POSINT** `$integer`

The `_POSINT` function is intended to be imported into your package, and provides a convenient way to test to see if a value is a positive integer (of any length).

Returns the value as a convience, or `undef` if the value is not a positive integer.

The name itself is derived from the XML schema constraint of the same name.

**_NONNEGINT** `$integer`

The `_NONNEGINT` function is intended to be imported into your package, and provides a convenient way to test to see if a value is a non-negative integer (of any length). That is, a positive integer, or zero.

Returns the value as a convience, or `undef` if the value is not a non-negative integer.

As with other tests that may return false values, care should be taken to test via ''defined'' in boolean validy contexts.

```
unless ( defined _NONNEGINT($value) ) {
   die "Invalid value";
}
```

The name itself is derived from the XML schema constraint of the same name.

**_SCALAR \$scalar**

The _SCALAR function is intended to be imported into your package, and provides a convenient way to test for a raw and unblessed SCALAR reference, with content of non-zero length.

For a version that allows zero length SCALAR references, see the _SCALAR0 function.

Returns the SCALAR reference itself as a convenience, or undef if the value provided is not a SCALAR reference.

**_SCALAR0 \$scalar**

The _SCALAR0 function is intended to be imported into your package, and provides a convenient way to test for a raw and unblessed SCALAR0 reference, allowing content of zero-length.

For a simpler ''give me some content'' version that requires non-zero length, _SCALAR function.

Returns the SCALAR reference itself as a convenience, or undef if the value provided is not a SCALAR reference.

**_ARRAY** $value

The _ARRAY function is intended to be imported into your package, and provides a convenient way to test for a raw and unblessed ARRAY reference containing **at least** one element of any kind.

For a more basic form that allows zero length ARRAY references, see the _ARRAY0 function.

Returns the ARRAY reference itself as a convenience, or undef if the value provided is not an ARRAY reference.

**_ARRAY0** $value

The _ARRAY0 function is intended to be imported into your package, and provides a convenient way to test for a raw and unblessed ARRAY reference, allowing ARRAY references that contain no elements.

For a more basic ''An array of something'' form that also requires at least one element, see the _ARRAY function.

Returns the ARRAY reference itself as a convenience, or undef if the value provided is not an ARRAY reference.

**_ARRAYLIKE** $value

The _ARRAYLIKE function tests whether a given scalar value can respond to array dereferencing. If it can, the value is returned. If it cannot, _ARRAYLIKE returns undef.

**_HASH** $value

The _HASH function is intended to be imported into your package, and provides a convenient way to test for a raw and unblessed HASH reference with at least one entry.

For a version of this function that allows the HASH to be empty, see the _HASH0 function.

Returns the HASH reference itself as a convenience, or undef if the value provided is not an HASH reference.

**_HASH0** $value

The _HASH0 function is intended to be imported into your package, and provides a convenient way to test for a raw and unblessed HASH reference, regardless of the HASH content.

For a simpler ''A hash of something'' version that requires at least one element, see the _HASH function.

Returns the HASH reference itself as a convenience, or undef if the value provided is not an HASH reference.

**_HASHLIKE** $value

The _HASHLIKE function tests whether a given scalar value can respond to hash dereferencing. If it can, the value is returned. If it cannot, _HASHLIKE returns undef.

**_CODE** $value

The _CODE function is intended to be imported into your package, and provides a convenient way to test for a raw and unblessed CODE reference.

Returns the CODE reference itself as a convenience, or undef if the value provided is not an CODE reference.

**_CODELIKE** $value

The _CODELIKE is the more generic version of _CODE. Unlike _CODE, which checks for an explicit CODE reference, the _CODELIKE function also includes things that act like them, such as blessed objects that overload '&{}'.

Please note that in the case of objects overloaded with '&{}', you will almost always end up also testing it in 'bool' context at some stage.

For example:

```
sub foo {
    my $code1 = _CODELIKE(shift) or die "No code param provided";
    my $code2 = _CODELIKE(shift);
    if ( $code2 ) {
        print "Got optional second code param";
    }
}
```

As such, you will most likely always want to make sure your class has at least the following to allow it to evaluate to true in boolean context.

```
# Always evaluate to true in boolean context
use overload 'bool' => sub () { 1 };
```

Returns the callable value as a convenience, or undef if the value provided is not callable.

Note − This function was formerly known as _CALLABLE but has been renamed for greater symmetry with the other _XXXXLIKE functions.

The use of _CALLABLE has been deprecated. It will continue to work, but with a warning, until end−2006, then will be removed.

I apologise for any inconvenience caused.

**_INVOCANT** $value

This routine tests whether the given value is a valid method invocant. This can be either an instance of an object, or a class name.

If so, the value itself is returned. Otherwise, _INVOCANT returns undef.

**_INSTANCE** $object, $class

The _INSTANCE function is intended to be imported into your package, and provides a convenient way to test for an object of a particular class in a strictly correct manner.

Returns the object itself as a convenience, or undef if the value provided is not an object of that type.

**_INSTANCEDOES** $object, $role

This routine behaves exactly like "_INSTANCE", but checks with −>DOES rather than −>isa. This is probably only a good idea to use on Perl 5.10 or later, when UNIVERSAL::DOES has been implemented.

**_REGEX** $value

The _REGEX function is intended to be imported into your package, and provides a convenient way to test for a regular expression.

Returns the value itself as a convenience, or undef if the value provided is not a regular expression.

**_SET \@array,** $class

The _SET function is intended to be imported into your package, and provides a convenient way to test for set of at least one object of a particular class in a strictly correct manner.

The set is provided as a reference to an ARRAY of objects of the class provided.

For an alternative function that allows zero-length sets, see the _SET0 function.

Returns the ARRAY reference itself as a convenience, or undef if the value provided is not a set of that class.

**_SET0 \@array,** $class

The _SET0 function is intended to be imported into your package, and provides a convenient way to test for a set of objects of a particular class in a strictly correct manner, allowing for zero objects.

The set is provided as a reference to an ARRAY of objects of the class provided.

For an alternative function that requires at least one object, see the _SET function.

Returns the ARRAY reference itself as a convenience, or undef if the value provided is not a set of that class.

**_HANDLE**

The _HANDLE function is intended to be imported into your package, and provides a convenient way to test whether or not a single scalar value is a file handle.

Unfortunately, in Perl the definition of a file handle can be a little bit fuzzy, so this function is likely to be somewhat imperfect (at first anyway).

That said, it is implement as well or better than the other file handle detectors in existence (and we stole from the best of them).

**_DRIVER** $string

```
sub foo {
  my $class = _DRIVER(shift, 'My::Driver::Base') or die "Bad driver";
  ...
}
```

The _DRIVER function is intended to be imported into your package, and provides a convenient way to load and validate a driver class.

The most common pattern when taking a driver class as a parameter is to check that the name is a class (i.e. check against _CLASS) and then to load the class (if it exists) and then ensure that the class returns true for the isa method on some base driver name.

Return the value as a convenience, or undef if the value is not a class name, the module does not exist, the module does not load, or the class fails the isa test.

## TO DO

– Add _CAN to help resolve the UNIVERSAL::can debacle

– Would be even nicer if someone would demonstrate how the hell to build a Module::Install dist of the ::Util dual Perl/XS type. :/

– Implement an assertion-like version of this module, that dies on error.

– Implement a Test:: version of this module, for use in testing

## SUPPORT

Bugs should be reported via the CPAN bug tracker at

<http://rt.cpan.org/NoAuth/ReportBug.html?Queue=Params-Util>

For other issues, contact the author.

## AUTHOR

Adam Kennedy <adamk@cpan.org>

## SEE ALSO

Params::Validate

## COPYRIGHT

The full text of the license can be found in the LICENSE file included with this module.