NAME

UNIVERSAL - base class for ALL classes (blessed references)

SYNOPSIS

```
    $is_io    = $fd->isa("IO::Handle");
    $is_io    = Class->isa("IO::Handle");

    $does_log = $obj->DOES("Logger");
    $does_log = Class->DOES("Logger");

    $sub      = $obj->can("print");
    $sub      = Class->can("print");

    $sub      = eval { $ref->can("fandango") };
    $ver      = $obj->VERSION;

    # but never do this!
    $is_io    = UNIVERSAL::isa($fd, "IO::Handle");
    $sub      = UNIVERSAL::can($obj, "print");
```

DESCRIPTION

`UNIVERSAL` is the base class from which all blessed references inherit. See *perlobj*.

`UNIVERSAL` provides the following methods:

`$obj->isa( TYPE )`

`CLASS->isa( TYPE )`

`eval { VAL->isa( TYPE ) }`

Where

`TYPE`

is a package name

`$obj`

is a blessed reference or a package name

`CLASS`

is a package name

`VAL`

is any of the above or an unblessed reference

When used as an instance or class method (`$obj->isa( TYPE )`), `isa` returns *true* if $obj is blessed into package `TYPE` or inherits from package `TYPE`.

When used as a class method (`CLASS->isa( TYPE )`, sometimes referred to as a static method), `isa` returns *true* if `CLASS` inherits from (or is itself) the name of the package `TYPE` or inherits from package `TYPE`.

If you're not sure what you have (the `VAL` case), wrap the method call in an `eval` block to catch the exception if `VAL` is undefined.

If you want to be sure that you're calling `isa` as a method, not a class, check the invocand with `blessed` from *Scalar::Util* first:

```
  use Scalar::Util 'blessed';
```

```
        if ( blessed( $obj ) && $obj->isa("Some::Class") ) {
            ...
        }
```

`$obj->DOES( ROLE )`

`CLASS->DOES( ROLE )`

> `DOES` checks if the object or class performs the role `ROLE`. A role is a named group of specific behavior (often methods of particular names and signatures), similar to a class, but not necessarily a complete class by itself. For example, logging or serialization may be roles.
>
> `DOES` and `isa` are similar, in that if either is true, you know that the object or class on which you call the method can perform specific behavior. However, `DOES` is different from `isa` in that it does not care *how* the invocand performs the operations, merely that it does. (`isa` of course mandates an inheritance relationship. Other relationships include aggregation, delegation, and mocking.)
>
> By default, classes in Perl only perform the `UNIVERSAL` role, as well as the role of all classes in their inheritance. In other words, by default `DOES` responds identically to `isa`.
>
> There is a relationship between roles and classes, as each class implies the existence of a role of the same name. There is also a relationship between inheritance and roles, in that a subclass that inherits from an ancestor class implicitly performs any roles its parent performs. Thus you can use `DOES` in place of `isa` safely, as it will return true in all places where `isa` will return true (provided that any overridden `DOES` *and* `isa` methods behave appropriately).

`$obj->can( METHOD )`

`CLASS->can( METHOD )`

`eval { VAL->can( METHOD ) }`

> `can` checks if the object or class has a method called `METHOD`. If it does, then it returns a reference to the sub. If it does not, then it returns *undef*. This includes methods inherited or imported by `$obj`, `CLASS`, or `VAL`.
>
> `can` cannot know whether an object will be able to provide a method through AUTOLOAD (unless the object's class has overridden `can` appropriately), so a return value of *undef* does not necessarily mean the object will not be able to handle the method call. To get around this some module authors use a forward declaration (see *perlsub*) for methods they will handle via AUTOLOAD. For such 'dummy' subs, `can` will still return a code reference, which, when called, will fall through to the AUTOLOAD. If no suitable AUTOLOAD is provided, calling the coderef will cause an error.
>
> You may call `can` as a class (static) method or an object method.
>
> Again, the same rule about having a valid invocand applies -- use an `eval` block or `blessed` if you need to be extra paranoid.

`VERSION ( [ REQUIRE ] )`

> `VERSION` will return the value of the variable `$VERSION` in the package the object is blessed into. If `REQUIRE` is given then it will do a comparison and die if the package version is not greater than or equal to `REQUIRE`, or if either `$VERSION` or `REQUIRE` is not a "lax" version number (as defined by the *version* module).
>
> The return from `VERSION` will actually be the stringified version object using the package `$VERSION` scalar, which is guaranteed to be equivalent but may not be precisely the contents of the `$VERSION` scalar. If you want the actual contents of `$VERSION`, use `$CLASS::VERSION` instead.
>
> `VERSION` can be called as either a class (static) method or an object method.

WARNINGS

NOTE: `can` directly uses Perl's internal code for method lookup, and `isa` uses a very similar method and cache-ing strategy. This may cause strange effects if the Perl code dynamically changes @ISA in any package.

You may add other methods to the UNIVERSAL class via Perl or XS code. You do not need to `use UNIVERSAL` to make these methods available to your program (and you should not do so).

EXPORTS

None by default.

You may request the import of three functions (`isa`, `can`, and `VERSION`), **but this feature is deprecated and will be removed**. Please don't do this in new code.

For example, previous versions of this documentation suggested using `isa` as a function to determine the type of a reference:

```
use UNIVERSAL 'isa';

$yes = isa $h, "HASH";
$yes = isa "Foo", "Bar";
```

The problem is that this code will *never* call an overridden `isa` method in any class. Instead, use `reftype` from *Scalar::Util* for the first case:

```
use Scalar::Util 'reftype';

$yes = reftype( $h ) eq "HASH";
```

and the method form of `isa` for the second:

```
$yes = Foo->isa("Bar");
```