NAME

TAP::Parser::ResultFactory - Factory for creating TAP::Parser output objects

SYNOPSIS

```
use TAP::Parser::ResultFactory;
my $token   = {...};
my $factory = TAP::Parser::ResultFactory->new;
my $result  = $factory->make_result( $token );
```

VERSION

Version 3.23

DESCRIPTION

This is a simple factory class which returns a *TAP::Parser::Result* subclass representing the current bit of test data from TAP (usually a single line). It is used primarily by *TAP::Parser::Grammar*. Unless you're subclassing, you probably won't need to use this module directly.

METHODS

Class Methods

new

Creates a new factory class. *Note:* You currently don't need to instantiate a factory in order to use it.

make_result

Returns an instance the appropriate class for the test token passed in.

```
my $result = TAP::Parser::ResultFactory->make_result($token);
```

Can also be called as an instance method.

class_for

Takes one argument: `$type`. Returns the class for this $type, or `croaks` with an error.

register_type

Takes two arguments: `$type`, `$class`

This lets you override an existing type with your own custom type, or register a completely new type, eg:

```
# create a custom result type:
package MyResult;
use strict;
use vars qw(@ISA);
@ISA = 'TAP::Parser::Result';

# register with the factory:
TAP::Parser::ResultFactory->register_type( 'my_type' => __PACKAGE__ );

# use it:
my $r = TAP::Parser::ResultFactory->( { type => 'my_type' } );
```

Your custom type should then be picked up automatically by the *TAP::Parser*.

SUBCLASSING

Please see *"SUBCLASSING" in TAP::Parser* for a subclassing overview.

There are a few things to bear in mind when creating your own `ResultFactory`:

1. The factory itself is never instantiated (this *may* change in the future). This means that `_initialize` is never called.
2. `TAP::Parser::Result->new` is never called, $tokens are reblessed. This *will* change in a future version!
3. *TAP::Parser::Result* subclasses will register themselves with *TAP::Parser::ResultFactory* directly:

   ```
   package MyFooResult;
   TAP::Parser::ResultFactory->register_type( foo => __PACKAGE__ );
   ```

   Of course, it's up to you to decide whether or not to ignore them.

Example

```
package MyResultFactory;

use strict;
use vars '@ISA';

use MyResult;
use TAP::Parser::ResultFactory;

@ISA = qw( TAP::Parser::ResultFactory );

# force all results to be 'MyResult'
sub class_for {
  return 'MyResult';
}

1;
```

SEE ALSO

TAP::Parser, *TAP::Parser::Result*, *TAP::Parser::Grammar*