Rocky Enterprise Linux 9.2 Manual Pages on command 'Type::Tiny::Manual::UsingWithMoo.3pm'

C:\>man Type::Tiny::Manual::UsingWithMoo.3pm

Type::Tiny::Manual::UsingWitUser(Contributed Perl DocType::Tiny::Manual::UsingWithMoo(3pm)

NAME

Type::Tiny::Manual::UsingWithMoo - basic use of Type::Tiny with Moo

MANUAL

Type Constraints

Consider the following basic Moo class:

```
package Horse {
  use Moo;
  use namespace::autoclean;
  has name       => ( is => 'ro' );
  has gender     => ( is => 'ro' );
  has age        => ( is => 'rw' );
  has children   => ( is => 'ro', default => sub { [] } );
}
```

Code like this seems simple enough:

```
my $br = Horse->new(name => "Bold Ruler", gender => 'm', age => 16);
push @{ $br->children },
  Horse->new(name => 'Secretariat', gender => 'm', age => 0);
```

However, once you step away from very simple use of the class, things can start to go wrong. When we push a new horse onto "@{ $br->children }", we are assuming that "$br->children" returned an arrayref.

What if the code that created the $br horse had instantiated it like this?

```
  my $br = Horse->new(name => "Bold Ruler", children => 'no');
```

It is for this reason that it's useful for the Horse class to perform some basic sanity-checking on its own attributes.

```
  package Horse {
    use Moo;
    use Types::Standard qw( Str Num ArrayRef );
    use namespace::autoclean;
    has name       => ( is => 'ro', isa => Str );
    has gender     => ( is => 'ro', isa => Str );
    has age        => ( is => 'rw', isa => Num );
    has children   => (
      is      => 'ro',
      isa     => ArrayRef,
      default => sub { return  [] },
    );
  }
```

Now, if you instantiate a horse like this, it will throw an error:

```
  my $br = Horse->new(name => "Bold Ruler", children => 'no');
```

The first type constraint we used here was Str. This is type constraint that requires values to be strings.

Note that although "undef" is not a string, the empty string is still a string and you will often want to check that a string is non-empty. We could have done this:

```
  use Types::Common::String qw( NonEmptyStr );
  has name => ( is => 'ro', isa => NonEmptyStr );
```

While most of the type constraints we will use in this manual are defined in Types::Standard, the Types::Common::String type library also defines many useful type constraints.

We have required the horse's age to be a number. This is also a common, useful type constraint. If we want to make sure it's a whole number, we could use:

```
  use Types::Standard qw( Int );
  has age => ( is => 'rw', isa => Int );
```

Or because negative numbers make little sense as an age:

```
  use Types::Common::Numeric qw( PositiveOrZeroInt );
```

```
  has age => ( is => 'rw', isa => PositiveOrZeroInt );
```

The Types::Common::Numeric library defines many useful subtypes of Int and Num, such as PositiveInt and PositiveOrZeroInt.

The last type constraint we've used in this example is ArrayRef. This requires the value to be a reference to an array.

Types::Standard also provides HashRef and CodeRef type constraints. An example of using the latter:

```
  package Task {
    use Moo;
    use Types::Standard qw( CodeRef Bool );
    has on_success => ( is => 'ro', isa => CodeRef );
    has on_failure => ( is => 'ro', isa => CodeRef );
    has finished   => ( is => 'ro', isa => Bool, default => 0 );
    ...;
  }
  my $task = Task->new(
    on_success => sub { ... },
    on_failure => sub { ... },
    ...,
  );
```

The Bool type constraint accepts "1" as a true value, and "0", "", or undef as false values. No other values are accepted.

There exists an Object type constraint that accepts any blessed object.

```
  package Horse {
    use Moo;
    use Types::Standard qw( Object );
    use namespace::autoclean;
    ...;  # name, gender, age, children
    has father => ( is => 'ro', isa => Object );
    has mother => ( is => 'ro', isa => Object );
  }
```

Finally, another useful type constraint to know about is Any:

```
  use Types::Standard qw( Any );
```

```
has stuff => ( is => 'rw', isa => Any );
```

This type constraint allows any value; it is essentially the same as not doing any type check, but makes your intent clearer. Where possible, Type::Tiny will optimize away this type check, so it should have little (if any) impact on performance.

Parameterized Types

Let's imagine we want to keep track of our horse's race wins:

```
package Horse {
  use Moo;
  use Types::Standard qw( Str Num ArrayRef );
  use namespace::autoclean;
  ...;  # name, gender, age, children
  has wins => (
    is      => 'ro',
    isa     => ArrayRef,
    default => sub { return [] },
  );
}
```

We can create a horse like this:

```
my $br = Horse->new(
  name    => "Bold Ruler",
  gender  => 'm',
  age     => 4,
  wins    => ["Futurity Stakes 1956", "Juvenile Stakes 1956"],
);
```

The list of wins is an arrayref of strings. The ArrayRef type constraint prevents it from being set to a hashref, for example, but it doesn't ensure that everything in the arrayref is a string. To do that, we need to parameterize the type constraint:

```
has wins => (
  is      => 'ro',
  isa     => ArrayRef[Str],
  default => sub { return [] },
);
```

Thanks to the ArrayRef[Str] parameterized type, the constructor will throw an error if the arrayref you pass to it contains anything non-string.

An alternative way of writing this is:

```
has wins => (
  is      => 'ro',
  isa     => ArrayRef->of(Str),
  default => sub { return [] },
);
```

Which way you choose is largely a style preference. TIMTOWTDI!

Note that although the constructor and any setter/accessor method will perform type checks, it is possible to bypass them using:

```
push @{ $br->wins }, $not_a_string;
```

The constructor isn't being called here, and although the accessor is being called, it's being called as a reader, not a writer, so never gets an opportunity to inspect the value being added. (It is possible to use "tie" to solve this, but that will be covered later.)

And of course, if you directly poke at the underlying hashref of the object, all bets are off:

```
$br->{wins} = $not_an_arrayref;
```

So type constraints do have limitations. Careful API design (and not circumventing the proper API) can help.

The HashRef type constraint can also be parameterized:

```
package Design {
  use Moo;
  use Types::Standard qw( HashRef Str );
  has colours => ( is => 'ro', isa => HashRef[Str] );
}
my $eiffel65 = Design->new(
  colours => { house => "blue", little_window => "blue" },
);
```

The HashRef[Str] type constraint ensures the values of the hashref are strings; it doesn't check the keys of the hashref because keys in Perl hashes are always strings!

If you do need to constrain the keys, it is possible to use a parameterized Map constraint:

```
  use Types::Common::String qw( NonEmptyStr );
  use Types::Standard qw( Map );
  has colours => ( is => 'ro', isa => Map[NonEmptyStr, NonEmptyStr] );
```

Map takes two parameters; the first is a type to check keys against and the second is a type to check values against.

Another useful type constraint is the Tuple type constraint.

```
  use Types::Standard qw( ArrayRef Tuple );
  use Types::Common::Numeric qw( PositiveInt );
  use Types::Common::String qw( NonEmptyStr );
  has wins => (
    is      => 'ro',
    isa     => ArrayRef[ Tuple[PositiveInt, NonEmptyStr] ],
    default => sub { return [] },
  );
```

The Tuple[PositiveInt, NonEmptyStr] type constraint checks that a value is a two-element arrayref where the first element is a positive integer and the second element is a non-empty string. For example:

```
  my $br = Horse->new(
    name    => "Bold Ruler",
    wins    => [
      [ 1956, "Futurity Stakes" ],
      [ 1956, "Juvenile Stakes" ],
    ],
  );
```

As you can see, parameterized type constraints may be nested to arbitrary depth, though of course the more detailed your checks become, the slower they will perform.

It is possible to have tuples with variable length. For example, we may wish to include the jockey name in our race wins when it is known.

```
  use Types::Standard qw( ArrayRef Tuple Optional );
  use Types::Common::Numeric qw( PositiveInt );
```

```
  use Types::Common::String qw( NonEmptyStr );
  has wins => (
    is      => 'ro',
    isa     => ArrayRef[
      Tuple[ PositiveInt, NonEmptyStr, Optional[NonEmptyStr] ]
    ],
    default => sub { return [] },
  );
```

The third element will be checked if it is present, but forgiven if it is absent.

Or we could just allow tuples to contain an arbitrary list of strings after the year and race name:

```
  use Types::Standard qw( ArrayRef Tuple Str slurpy );
  use Types::Common::Numeric qw( PositiveInt );
  use Types::Common::String qw( NonEmptyStr );
  has wins => (
    is      => 'ro',
    isa     => ArrayRef[
      Tuple[ PositiveInt, NonEmptyStr, slurpy ArrayRef[Str] ]
    ],
    default => sub { return [] },
  );
```

The "slurpy" indicator will "slurp" all the remaining items in the tuple into an arrayref and check it against ArrayRef[Str].

It's even possible to do this:

```
  use Types::Standard qw( ArrayRef Tuple Any slurpy );
  use Types::Common::Numeric qw( PositiveInt );
  use Types::Common::String qw( NonEmptyStr );
  has wins => (
    is      => 'ro',
    isa     => ArrayRef[
      Tuple[ PositiveInt, NonEmptyStr, slurpy Any ]
    ],
    default => sub { return [] },
```

```
);
```

With this type constraint, any elements after the first two will be slurped into an arrayref and we don't check that arrayref at all. (In fact, the implementation of the Tuple type is smart enough to not bother creating the temporary arrayref to check.)

Dict is the equivalent of Tuple for checking values of hashrefs.

```
use Types::Standard qw( ArrayRef Dict Optional );
use Types::Common::Numeric qw( PositiveInt );
use Types::Common::String qw( NonEmptyStr );
has wins => (
  is      => 'ro',
  isa     => ArrayRef[
    Dict[
      year    => PositiveInt,
      race    => NonEmptyStr,
      jockey  => Optional[NonEmptyStr],
    ],
  ],
  default => sub { return [] },
);
```

An example of using it:

```
my $br = Horse->new(
  name    => "Bold Ruler",
  wins    => [
    { year => 1956, race => "Futurity Stakes", jockey => "Eddie" },
    { year => 1956, race => "Juvenile Stakes" },
  ],
);
```

The slurpy indicator does work for Dict too:

```
Dict[
  year    => PositiveInt,
  race    => NonEmptyStr,
  jockey  => Optional[NonEmptyStr],
```

```
    slurpy HashRef[Str],  # other Str values allowed
  ]
```

And "slurpy Any" means what you probably think it means:

```
  Dict[
    year    => PositiveInt,
    race    => NonEmptyStr,
    jockey  => Optional[NonEmptyStr],
    slurpy Any,  # allow hashref to contain absolutely anything else
  ]
```

Going back to our first example, there's an opportunity to refine our ArrayRef constraint:

```
  package Horse {
    use Moo;
    use Types::Standard qw( Str Num ArrayRef );
    use namespace::autoclean;
    has name       => ( is => 'ro', isa => Str );
    has gender     => ( is => 'ro', isa => Str );
    has age        => ( is => 'rw', isa => Num );
    has children   => (
      is      => 'ro',
      isa     => ArrayRef[ InstanceOf["Horse"] ],
      default => sub { return [] },
    );
  }
```

The InstanceOf["Horse"] type constraint checks that a value is a blessed object in the Horse class. So the horse's children should be an arrayref of other Horse objects.

Internally it just checks "$_->isa("Horse")" on each item in the arrayref.

It is sometimes useful to instead check "$_->DOES($role)" or "$_->can($method)" on an object. For example:

```
  package MyAPI::Client {
    use Moo;
    use Types::Standard qw( HasMethods );
```

```
  has ua => (is => 'ro', isa => HasMethods["get", "post"] );
}
```

The ConsumerOf and HasMethods parameterizable types allow you to easily check roles and methods of objects.

The Enum parameterizable type allows you to accept a more limited set of string values. For example:

```
use Types::Standard qw( Enum );
has gender => ( is => 'ro', isa => Enum["m","f"] );
```

Or if you want a little more flexibility, you can use StrMatch which allows you to test strings against a regular expression:

```
use Types::Standard qw( StrMatch );
has gender => ( is => 'ro', isa => StrMatch[qr/^[MF]/i] );
```

Or StrLength to check the maximum and minimum length of a string:

```
use Types::Common::String qw( StrLength );
has name => ( is => 'ro', isa => StrLength[3, 100] );
```

The maximum can be omitted.

Similarly, the maximum and minimum values for a numeric type can be expressed using IntRange and NumRange:

```
use Types::Common::Numeric qw( IntRange );
# values over 200 are probably an input error
has age => ( is => 'ro', isa => IntRange[0, 200] );
```

Parameterized type constraints are one of the most powerful features of Type::Tiny, allowing a small set of constraints to be combined in useful ways.

Type Coercions

It is often good practice to be liberal in what you accept.

```
package Horse {
  use Moo;
  use Types::Standard qw( Str Num ArrayRef Bool );
  use namespace::autoclean;
  ...;  # name, gender, age, children, wins
  has is_alive => ( is => 'rw', isa => Bool, coerce => 1 );
}
```

The "coerce" option indicates that if a value is given which does not pass the Bool

type constraint, then it should be coerced (converted) into something that does.

The definition of Bool says that to convert a non-boolean to a bool, you just do "!! $non_bool". So all of the following will be living horses:

```
  Horse->new(is_alive => 42)
  Horse->new(is_alive => [])
  Horse->new(is_alive => "false") # in Perl, string "false" is true!
```

Bool is the only type constraint in Types::Standard that has a coercion defined for it. The NumericCode, UpperCaseStr, LowerCaseStr, UpperCaseSimpleStr, and LowerCaseSimpleStr types from Types::Common::String also have conversions defined.

The other built-in constraints do not define any coercions because it would be hard to agree on what it means to coerce from, say, a HashRef to an ArrayRef. Do we keep the keys? The values? Both?

But it is pretty simple to add your own coercions!

```
  use Types::Standard qw( ArrayRef HashRef Str );
  has things => (
    is     => 'rw',
    isa    => ArrayRef->plus_coercions(
      HashRef,     sub { [ values %$_ ] },
      Str,         sub { [ split /;/, $_ ] },
    ),
    coerce => 1,
  );
```

(Don't ever forget the "coerce => 1"!)

If a hashref is provided, the values will be used, and if a string is provided, it will be split on the semicolon. Of course, if an arrayref if provided, it already passes the type constraint, so no conversion is necessary.

The coercions should be pairs of "from types" and code to coerce the value. The code can be a coderef (as above) or just string of Perl code (as below). Strings of Perl code can usually be optimized better by Type::Tiny's internals, so are generally preferred. Thanks to Perl's "q{...}" operator, they can look just as clean and pretty as coderefs.

```
  use Types::Standard qw( ArrayRef HashRef Str );
  has things => (
```

```
    is     => 'rw',
    isa    => ArrayRef->plus_coercions(
      HashRef,     q{ values %$_ },
      Str,         q{ [ split /;/, $_ ] },
    ),
    coerce => 1,
  );
```

Coercions are deeply applied automatically, so the following will do what you expect.

```
  has inputs => (
    is     => 'ro',
    isa    => ArrayRef->of(Bool),
    coerce => 1
  );
```

I am, of course, assuming you expect something like:

```
  my $coerced = [ map { !!$_ } @$orig ];
```

If you were assuming that, congratulations! We are on the same wavelength.

And of course you can still add more coercions to the inherited ones...

```
  has inputs => (
    is     => 'ro',
    isa    => ArrayRef->of(Bool)->plus_coercions(Str, sub {...}),
    coerce => 1
  );
```

Method Parameters

So far we have just concentrated on the definition of object attributes, but type constraints are also useful to validate method parameters.

Let's remember our attribute for keeping track of a horse's race wins:

```
  use Types::Standard qw( ArrayRef Tuple Optional );
  use Types::Common::Numeric qw( PositiveInt );
  use Types::Common::String qw( NonEmptyStr );
  has wins => (
    is      => 'ro',
    isa     => ArrayRef[
```

```
      Tuple[ PositiveInt, NonEmptyStr, Optional[NonEmptyStr] ]
    ],
    default => sub { return [] },
  );
```

Because we don't trust outside code to push new entries onto this array, let's define a method in our class to do it.

```
  package Horse {
    ...;
    sub add_win {
      my $self = shift;
      my ($year, $race, $jockey) = @_;
      my $win = [
        $year,
        $race,
        $jockey ? $jockey : (),
      ];
      push @{ $self->wins }, $win;
      return $self;
    }
  }
```

This works pretty well, but we're still not actually checking the values of $year, $race, and $jockey. Let's use Type::Params for that:

```
  package Horse {
    use Types::Common::Numeric qw( PositiveInt );
    use Types::Common::String qw( NonEmptyStr );
    use Type::Params qw( compile );
    ...;
    sub add_win {
      state $check = compile(
        PositiveInt,
        NonEmptyStr,
        NonEmptyStr, { optional => 1 },
      );
```

```
    my $self = shift;
    my ($year, $race, $jockey) = $check->(@_);
    my $win = [
      $year,
      $race,
      $jockey ? $jockey : (),
    ];
    push @{ $self->wins }, $win;
    return $self;
  }
}
```

The first time this method is called, it will compile a coderef called $check. Then every time it is run, $check will be called to check the method's parameters. It will throw an exception if they fail. $check will also perform coercions if types have them (and you don't even need to remember "coerce => 1"; it's always automatic) and can even add in defaults:

```
state $check = compile(
  PositiveInt,
  NonEmptyStr,
  NonEmptyStr, { default => sub { "Eddie" } },
);
```

On older versions of Perl (prior to 5.10), "state" variables are not available. A workaround is to replace this:

```
sub foo {
  state $x = bar();
  ...;
}
```

With this:

```
{       # outer braces prevent other subs seeing $x
  my $x;  # declare $x before sub foo()
  sub foo {
    $x = bar();
    ...;
```

```
  }
}
```

(While we're having a general Perl syntax lesson, I'll note that &$check with an ampersand and no parentheses is a shortcut for "$check->(@_)" and actually runs slightly faster because it reuses the @_ array for the called coderef. A lot of people dislike calling subs with an ampersand, so we will stick to the "$check->(@_)" syntax in these examples. But do consider using the shortcut!)

The generalized syntax for "compile" is:

```
state $check = compile(
  \%general_options,
  TypeForFirstParam,  \%options_for_first_param,
  TypeForSecondParam, \%options_for_second_param,
  ...,
);
```

As a shortcut for the "{ optional => 1 }}" option, you can just use Optional like in Tuple.

```
state $check = compile(
  PositiveInt,
  NonEmptyStr,
  Optional[NonEmptyStr],
);
```

You can also use 0 and 1 as shortcuts for Optional[Any] and Any. The following checks that the first parameter is a positive integer, the second parameter is required (but doesn't care what value it is) and the third parameter is allowed but not required.

```
state $check = compile(PositiveInt, 1, 0);
```

It is possible to accept a variable number of values using "slurpy":

```
package Horse {
  use Types::Common::Numeric qw( PositiveInt );
  use Types::Common::String qw( NonEmptyStr );
  use Types::Standard qw( ArrayRef slurpy );
  use Type::Params qw( compile );
  ...;
```

```
  sub add_wins_for_year {
    state $check = compile(
      PositiveInt,
      slurpy ArrayRef[NonEmptyStr],
    );
    my $self = shift;
    my ($year, $races) = $check->(@_);
    for my $race (@$races) {
      push @{ $self->wins }, [$year, $win];
    }
    return $self;
  }
}
```

It would be called like this:

```
$bold_ruler->add_wins_for_year(
  1956,
  "Futurity Stakes",
  "Juvenile Stakes",
);
```

The additional parameters are slurped into an arrayref and checked against ArrayRef[NonEmptyStr].

Optional parameters are only allowed after required parameters, and slurpy parameters are only allowed at the end. (And there can only be a at most one slurpy parameter!)

For methods that accept more than one or two parameters, it is often a good idea to provide them as a hash. For example:

```
$horse->add_win(
  year    => 1956,
  race    => "Futurity Stakes",
  jockey  => "Eddie",
);
```

This can make your code more readable.

To accept named parameters, use "compile_named" instead of "compile".

```
package Horse {
  use Types::Common::Numeric qw( PositiveInt );
  use Types::Common::String qw( NonEmptyStr );
  use Type::Params qw( compile_named );
  ...;
  sub add_win {
    state $check = compile_named(
      year    => PositiveInt,
      race    => NonEmptyStr,
      jockey  => NonEmptyStr, { optional => 1 },
    );
    my $self = shift;
    my $args = $check->(@_);
    my $win = [
      $args->{year},
      $args->{race},
      exists($args->{jockey}) ? $args->{jockey} : (),
    ];
    push @{ $self->wins }, $win;
    return $self;
  }
}
```

"compile" and "compile_named" work pretty much the same, except the latter accepts named parameters instead of positional, and returns a hashref.

It will automatically allow for a hashref to be provided instead of a full hash.

The following both work, but the $args variable will always be given a hashref.

```
$horse->add_win({
  year    => 1956,
  race    => "Juvenile Stakes",
});
$horse->add_win(
  year    => 1956,
  race    => "Futurity Stakes",
```

```
    jockey  => "Eddie",
  );
```

Well... I say "always" but you can tell "compile_named" to accept named parameters but return a positional list of parameters:

```
  package Horse {
    use Types::Common::Numeric qw( PositiveInt );
    use Types::Common::String qw( NonEmptyStr );
    use Type::Params qw( compile_named );
    ...;
    sub add_win {
      state $check = compile_named(
        { named_to_list => 1 },
        year    => PositiveInt,
        race    => NonEmptyStr,
        jockey  => NonEmptyStr, { optional => 1 },
      );
      my $self = shift;
      my ($year, $race, $jockey) = $check->(@_);
      my $win = [
        $year,
        $race,
        $jockey ? $jockey : (),
      ];
      push @{ $self->wins }, $win;
      return $self;
    }
  }
```

Optional and slurpy named parameters are supported as you'd expect.

With named parameters, it can be easy to misspell keys in your method definition.

For example:

```
  my $win = [
    $args->{year},
    $args->{race},
```

```
    exists($args->{jockee}) ? $args->{jockey} : (),
  ];
```

Note "jockee"! This can lead to hard-to-find bugs. There's a "compile_named_oo" function which may help and can lead to cleaner code.

```
  package Horse {
    use Types::Common::Numeric qw( PositiveInt );
    use Types::Common::String qw( NonEmptyStr );
    use Type::Params qw( compile_named_oo );
    ...;
    sub add_win {
      state $check = compile_named_oo(
        year    => PositiveInt,
        race    => NonEmptyStr,
        jockey  => NonEmptyStr, { optional => 1 },
      );
      my $self = shift;
      my $args = $check->(@_);
      my $win = [
        $args->year,
        $args->race,
        $args->has_jockey ? $args->jockey : (),
      ];
      push @{ $self->wins }, $win;
      return $self;
    }
  }
```

Now $args is a blessed object that you can call methods on. There is of course a performance penalty for this, but it's surprisingly small.

For more information on Type::Params, and third-party alternatives, see Type::Tiny::Manual::Params.

NEXT STEPS

Congratulations! I know this was probably a lot to take in, but you've covered all of the essentials.

You can now set type constraints and coercions for attributes and method parameters in Moo! You are familiar with a lot of the most important and useful type constraints and understand parameterization and how it can be used to build more specific type constraints.

(And I'll let you in on a secret. Using Type::Tiny with Moose or Mouse instead of Moo is exactly the same. You can just replace "use Moo" with "use Moose" in any of these examples and they should work fine!)

Here's your next step:

```
?   Type::Tiny::Manual::UsingWithMoo2
    Advanced use of Type::Tiny with Moo, including unions and intersections,
    "stringifies_to", "numifies_to", "with_attribute_values", and "where".
```

NOTES

On very old versions of Moo "coerce => 1" is not supported. Instead you will need to provide a coderef or object overloading "&{}" to coerce. Type::Tiny can provide you with an overloaded object.

```
 package Horse {
   use Moo;
   use Types::Standard qw( Str Num ArrayRef Bool );
   use namespace::autoclean;
   ...;  # name, gender, age, children, wins
   has is_alive => (
     is      => 'rw',
     isa     => Bool,
     coerce  => Bool->coercion,  # overloaded object
   );
 }
```

If you have a very old version of Moo, please upgrade to at least Moo 1.006000 which was the version that added support for "coerce => 1".

AUTHOR

Toby Inkster <tobyink@cpan.org>.

COPYRIGHT AND LICENCE

This software is copyright (c) 2013-2014, 2017-2019 by Toby Inkster.

This is free software; you can redistribute it and/or modify it under the same

terms as the Perl 5 programming language system itself.

DISCLAIMER OF WARRANTIES

THIS PACKAGE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.