Bud1%cssdsclboolclbool @ @ @ @E%DSDB` @ @ @

- User Guide
  - Quick Start
    - Unit
    - Acceptance
    - Execution
    - And much more!
  - Structure
    - Presentation
    - Declare examples
      - Styles
        - Acceptance specification
        - Unit specification
      - Results
        - Standard
        - Matchers
        - Expectations
        - Functional
        - Thrown
        - All
          - Short-circuit
      - Pending until fixed
      - Auto-Examples
      - G/W/T
        - Sequencing
        - Extract methods
        - User regexps
        - Factory methods
        - G/W/T sequences
        - Multiple steps
        - ScalaCheck
        - Single step
        - Conversions
        - Unit specification
      - DataTables
    - Links
      - Inclusion
        - Inline
        - Html link
          - Html Link
      - Reference
      - Markdown url
    - Contexts

uide

rary for writing executable software specifications in Scala.

u can write:

ations for simple classes (*unit* specifications)
ations for full features (*acceptance* specifications)

de, you will find:

guide
ture your specification
matchers
ute a specification
philosophy
design

fication User Guide
2 ms
examples, 566 expectations, 0 failure, 0 error

- User Guide
  - Quick Start
    - Unit
    - Acceptance
    - Execution
    - And much more!
  - Structure
    - Presentation
    - Declare examples
    - Styles
      - Acceptance specification
      - Unit specification
    - Results
      - Standard
      - Matchers
    - Expectations
      - Functional
      - Thrown
      - All
        - Short-circuit
    - Pending until fixed
    - Auto-Examples
    - G/W/T
      - Sequencing
      - Extract methods
      - Example
      - User regexps
      - Factory methods
      - G/W/T sequences
      - Multiple steps
      - ScalaCheck
      - Single step
      - Conversions
      - Unit specification
    - DataTables
  - Links
    - Inclusion
      - Inline
      - Html link
        - Html Link
    - Referring
    - Markdown
    - Contexts

ion

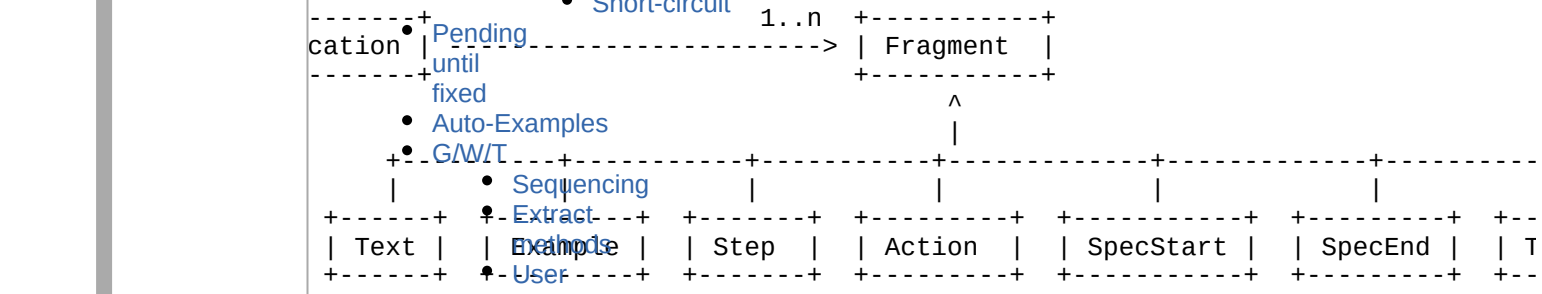
ains the overall design of *specs2*:

cture of a specification
specification is built
specification is executed
reporting works
kages dependencies

ails might not be completely up to date as the code base evolves.

Fragments

f a specification is very simple, it is just a list of `Fragments` provided by the `is` method of the `onStructure` trait



description of all the Fragments:

e text describing the specified system
e: a description and a piece of executable code returning a Result
ction: some action on the system which is only reported if there's an exception
rt / SpecEnd: delimiters for the Specification. They also delimitate included Specifications.
cStart element holds the Arguments used to tune the execution/reporting, the link to an
l/referenced specification
Fragments: those fragments enclose other fragments which can be included or excluded from the
n

cits to create Fragments (found in the `org.specs2.specification.FragmentsBuilder` trait):

=> `Text`, to create a simple Text Fragment
! `Result` => `Example`, to create an Example

- Isolation
  - Scope
  - Isolated variables
  - Case classes
  - Context inheritance
  - Before/After
  - In a mutable specification
  - In an acceptance specification
- Around
- Outside
- AroundOutside
- BeforeExample
- Implicit context
- Composition
  - Combinations
  - Composition
- Steps/Actions
  - Steps
  - Actions
- Template
- For fragments
- Execution
- Layout
- Rules
- Formatting fragments
  - Separating groups of examples
  - Reset the levels
  - Changing the indentation level
  - Combinations
  - Turning off the automatic layout
- Unit specification
- Unit specifications
- How to?
- Declare arguments
- Pass arguments

se Fragments can be "linked" with ^, creating a `Fragments` object, containing a `Seq[Fragment]`:

```
gments: Fragments =
 text" ^
elated to this Example" ! success
```

s object is used to hold temporarily a sequence of Fragments as it is built and it makes sure that when done, the Fragments passed for execution will start and end with proper SpecStart and SpecEnd

pecification

pecification there is no visible "link" between Fragments, they're all created and linked through anks to an enhanced version of the `FragmentsBuilder` trait in the `org.specs2.mutable`

```
d an Example and add it to the specFragments variable
xample must succeed" in { success }
hing here" in { success }
```

here is mutation involved here, it's not advised to do anything concurrent at that point.

is triggered by the various reporters and goes through 5 steps:

```
 from the Reporter trait
ntent |> select |> sequence |> execute |> store |> export(spec)
```

n: the Fragments are filtered according to the Arguments object. In that phase all examples but a few iltered if the `only("this example")` option is used for instance. Another way to select fragments is `TaggingFragments` inside the specification.
`olated` argument is true, each example body is replaced with the same body executing in a cloned ation to avoid seeing side-effects on local variables.

cing: the Fragments are sorted in groups so that all the elements of a group can be executed ently. This usually why Steps are used. If my fragments are: `fragments1 ^ step ^ fragments2` fragments1 will be executed,
p, then fragments2

on: for each group, the execution of the fragments is concurrent by default and results are collected in nce of `ExecutingFragments`. We don't wait for the execution of all the Fragments to be finished tarting the reporting.

after an execution we compute the statistics for each specification and store the results in a file (`-reports/specs2.stats`).
ows to do consequent runs based on previous executions: to execute failed specifications only or to he index page with an indicator of previously executed specifications

g: depending on the exporter, the ExecutedFragments are translated to `PrintLines` or `HtmlLines` shed out to the console or in an html file

- Add a title
- Use descriptions
- Enhance failures
- Share examples
- Execute an index
- Tag examples
- In a unit specification
- Skip examples
- Debug statements
- Remove implicits

s start with a sequence of `ExecutingFragments`. A list of `Reducers` is used to collect relevant
and results to display
el" of the text i.e. its indentation. The rules for this are given in the Layout section of the Specification
e page
istics and execution times
licable arguments (where the arguments of an included specification must override the arguments of its
n difficulties in this 'reduction' is the fact that included specifications change the context of what needs
ted. The `reporter.NestedBlocks` trait provides functions to handle this.

gment and associated data (level, statistics, arguments,...) is translated to a display element:

nsole output, `PrintLines`: `PrintSpecStart`, `PrintText`, `PrintResult`,...
ml output, `HtmlLines`: `HtmlSpecStart`, `HtmlText`, `HtmlResult`,...
nit output, a tree of JUnit `Description` objects with the corresponding code to execute (in JUnit the
ions
first, then the examples are executed)

- Matchers
  - Boolean results
  - Standard results
  - Combinators
  - Match results
  - Out of the box
  - Optional
  - Custom
  - With sequences
- ScalaCheck
  - Arbitrary instances
  - With Generators
  - Test properties
- Mock expectations
  - Creation and settings
  - Stubbing
    - Mocking and Stubbing at the same time
  - With matchers
  - Callbacks

ncies

ackage dependencies should be always verified, from low-level packages to high-level ones, where no
ow layer can depend on a package on a higher layer:

```
r
    specification
rm

s reflect  xml html  time json
ion control  io  text  main data
```

ay to represent domain objects or services, and declare expected values in a tabular format. Forms can
ces of specification where complex forms can be built out of simple ones.

by creating `Fields` or `Props` and placing them on rows.
xamples show, by order of complexity, the creation of:

es
Form using properties
Address entity encapsulating the above form
site Customer entity using the Address instance
on table having some related columns
site Order - OrderLine entity (1-n) relationship

samples below you need to extend the `org.specs2.specification.Forms` trait.

---

ply a label and a value. It is used in forms to display regular information. You can create a `Field` with
:

`value)`: creates a field for a value, where the label is empty
`label, value)`: creates a field with a label and a value
`label, field1, field2, ...)`: creates a field with a label and values coming from other fields

is displayed, here's how the fields are displayed:

code	is displayed as	
ue")	value	
el", "value")	label	value
el", field("value1"), field("value2"))	label	value1/value2

cution, the value is only evaluated when the `Field` is executed (when executing the parent Form for
is thrown during that evaluation, the exception message will be displayed in place of the value.

---

almost like a `Field` but it never shows its value, unless there's an exception when executed. The value
supposed to have some kind of side-effect, like clicking on a webpage, and only the effect label will be
here is an exception in that case the exception message is added). You can create an `Effect` with
:

(value): creates an effect with no label
(label, value): creates an effect with a label and a value that will be evaluated when the Effect
ted
(effect1, effect2, ...): creates an effect with all the effects labels and a side-effect
cing all side-effects

Field, it has a label. But you can give it 2 values, an "actual" one and an "expected" one. When
roperty, both values are compared to get a result. You can create a Prop with the following functions:

alue): a property with no label
abel, actual): a property with a label and an actual value
abel, actual, expected): a property with a label, an actual value and an expected one

abel, actual, constraint): a property with a label, an actual value and a function taking the
alue,
cted one and returning a Result

```
("label", "actual", (a: String, b: String) => (a === b).toResult)
```

abel, actual, constraint): a property with a label, an actual value and a function
e expected value,
g a Matcher that will be applied to the actual one

```
("label", "expected", (s: String) => beEqualTo(s))
```

abel, actual, matcher): a property with a label, an actual value and a matcher to apply to that

```
("label", Some(1), beSome)
```

tcher is muted then no message will be displayed in case of a failure.

value is not provided when building the property, it can be given with the apply method:

```
y "sets" the expected value
ply("expected")

xpected")
```

ew examples:

code	is displayed as		
cted")("expected")	expected		
", "expected", "expected")	label	expected	
", "expected")("expected")	label	expected	
", "actual")("expected")	label	expected	'actual' is not equal to 'expected'
", { error("but got an error"); "actual" ed")	label	expected	but got an error
", "actual", (a: String, b: String) => (a ===)("expected")	label	expected	
", "actual", (s: String) =>))("expected")	label	expected	
			'actual' is not equal

", "actual", beEqualTo("expected"))	label	actual	to 'expected'
", "actual", beEqualTo("expected").mute)	label	actual	

cells to see the stacktraces]

e, the display of Fields and Properties can be left as it is but sometimes you want to style the output alues. You can do this by using `decorateWith` and `styleWith` methods, or some equivalent

code	is displayed as	
el", "value").decorateWith(f: Any => {f})	*label*	*value*
el", "value").bold	**label**	**value**
el", "value").boldLabel	**label**	value
el", "value").boldValue	label	**value**
el", "value").italics	*label*	*value*
el", "value").italics.bold	***label***	***value***
st_== 1").code	`1 must_== 1`	
el", "value").styleWith("color"->"#FF1493")	label	value
el", "value").color("#FF1493")	label	value
el", "value").bkColor("#FF1493")	label	value
el", "value").green	label	value
el", "value").bkGreen	label	value

s above, when named `xxx` are available as `xxxLabel` and `xxxValue` to do the formatting for the label

The available colors are:

e	color
FF	
FF	
99	
CC	
99	
EE	

m

ow how to create Fields and Properties, creating a `Form` is as easy as putting them on separate lines:

```
org.specs2.form._
FormsBuilder._

Address").
rop("street", actualStreet(123), "Oxford St")).
rop("number", actualNumber(123), 20))
```

- Add a title
- Use descriptions
- Enhance failures
- Share examples
- Create an index
- Tag examples
  - In a unit specification
  - Skip examples
  - Debug statements
  - Remove implicits
- Matchers
  - Boolean results
  - Standard results
  - Combinators
  - Match results
  - Out of the box Optional
  - Custom
  - With sequences
  - ScalaCheck
  - Arbitrary instances
  - With Generators
  - Test properties
- Mock expectations
  - Creation and settings
  - Stubbing
  - Mocking and Stubbing at the same time
  - With matchers
  - Callbacks

- Parameters for the answers function
- Verification Order of calls ignoring stubs
- Spies
- Functions/PartialFunctions
- Auto-boxing
- Byname
- DataTables
- Forms
- Forms
- Fields
- Effects
- Properties
- Styles
- Simple form
- Adding several rows at once
- Nesting a Form into another Form
- Nesting a Form into an Effect or a Prop
- Using tabs
- Aggregating forms
- Lazy cells
- Xml cells
- 1-n relationships
- Subset
- Subsequence
- Set
- Sequence
- Decision tables
- Outside specs2

title "Address" and 2 properties, each one on a distinct row. The `actualStreet()` and
`r()`
upposed to retrieve the relevant values from a database.

(see the Calculator example below) you can create a header row using the `th` method:

```
ld("a"), field("b"))
a", "b") using an implicit conversion of Any => Field[Any]
```

rm in a Specification is also very simple, you just chain it with the ^ operator:

```
SpecificationWithForms extends Specification with Forms { def is =

 address must be retrieved from the database with the proper street and number"
m("Address").
r(prop("street", actualStreet(123), "Oxford St")).
r(prop("number", actualNumber(123), 20))
```

capsulate and reuse this Form across specifications is to define a case class:

```
lass Address(street: String, number: Int) {
retrieve(addressId: Int) = {
 address = actualAddress(addressId)
m("Address").
r(prop("street", address.street, street)).
r(prop("number", address.number, number))

actualAddress(addressId: Int): AddressEntity  = ...
```

an use it like this:

```
AddressSpecification extends Specification with Forms { def is =
 address must be retrieved from the database with the proper street and number"
dress("Oxford St", 20).                        /** expected values */
retrieve(123)                                  /** actual address id */
```

eral rows at once

l way to add rows programmatically is to start from a seq of values and have a function creating a Row

```
 new Form").tr(addresses) { a: Address => Row.tr(field(a.number), field(a.stre
```

ent

Form into another Form

composed of other Forms to display composite information:

```
ress = Form("Address").
       tr(field("street", "Rose Crescent")).
```

```
    tr(field("number", 3))
son = Form("Person").
        tr(field("name", "Eric")).
        tr(address)
```

- Without any dependency on specs2
- Runners
  - Presentation
  - Dependencies
  - Arguments
    - API
      - Most/Least frequently used arguments
      - Shortcuts
      - Output directory
      - Storing previous results
      - Status flags
      - Diffs
      - StackTraceFilter
    - Command line
    - System properties
  - In the shell
    - Console output
    - Html output
    - JUnit XML output
    - Files Runner
  - In the console
  - Via SBT
    - with sbt 0.7.x
    - with sbt > 0.9.x
      - Test only arguments
      - Output formats
        - Html
        - Markdown
        - JUnit
        - Console
      - Files runner
      - Colors

played with the address as a nested table inside the main one on the last row. However in some case,

ave the rows of that Form to be included directly in the outer table. This can be done by *inlining* the

```
son = Form("Person").
        tr(field("name", "Eric")).
        tr(address.inline)            // address is inlined
```

s:

Crescent

Form into an Effect or a Prop

rms in specifications we can describe different levels of abstraction. If we consider the specification of xample, we want to be able to use a Form having 2 rows and describing the exact actions to do on the

```
oginForm = Form("login").
           tr(effect("click on login", clickOn("login"))).
           tr(effect("enter name",     enter("name", "me"))).
           tr(effect("enter password", enter("password", "pw"))).
           tr(effect("submit", submit))
```

d

purchase" scenario we want all the steps above to represent the login actions as just one step. One

nsform the login Form to an Effect or a Prop:

```
"purchase").
loginForm.toEffect("login")).
selectForm.toEffect("select goods")).
checkTotalForm.toProp("the total must be computed ok").bkWhiteLabel)
```

es fine, the detailed nested form is not shown:

- Via IDEA
- Via JUnit
- Via Eclipse
- Via Maven
- With your own
- Notifier
- NotifierRunner
- In sbt
- Exporter
- In sbt
- Philosophy
- The origins
- The score
- Conciseness
- Readability
- Extensibility
- Configuration
- Clear implementation
- User support
- Not hope
- Functional / immutable
- Chaining everything
- Arguments have to be supplied
- Concurrency is a breeze
- A simple structure
- Contexts
- Indentation
- Operators
- Forms
- But if you still want more specifications

be computed ok success

rm is embedded into an Effect, Errors will be reported
rm is embedded into a Prop, Failures will be reported, like that

be computed ok failed

Check Total

Compute total

Total	200	'100' is not equal to '200'

[click on failed cells to see the stacktraces]

cells to see the stacktraces]

S

many fields to be displayed on a Form you can use tabs:

```
on can have 2 addresses"^
"Addresses").tr {
("home",
ddress("Oxford St", 12).
ill("Oxford St", 12)).
("work",
ddress("Rose Cr.", 3).
ill("Rose Cr.", 3))
```

all will create a `Tabs` object containing the a first tab with "home" as the title and an Address form
Then every subsequent `tab` calls on the `Tabs` object will create new tabs:

rd St

Cr.

be created from a seq of values. Let's pretend we have a list of `Address` objects with a name and a

Address values. You can write:

```
resses") tab(adresses) { address: Address => tab(address.name, address.form) }
```

defined a form for a simple entity, let's see how we can reuse it with a larger entity:

tomer form defines a name attribute and embeds an instance of the Address form

ned by setting the name on one row and the Address form on the second row

this example, we define a slightly different Address form]

```
lass Address(street: String, number: Int) {
actualIs(address: AddressEntity) = {
rm("Address").
tr(prop("street", address.street, street)).
tr(prop("number", address.number, number))

}

case class Customer(name: String, address: Address) {
  def retrieve(customerId: Int) = {
    val customer = actualCustomer(customerId)
    Form("Customer").
      tr(prop("name", customer.name)(name)).
      tr(address.actualIs(customer.address))
  }
  def actualCustomer(customerId: Int) = ... // fetch from the database
}

class CustomerSpecification extends Specification with Forms { def is =
  "The customer must be retrieved from the database with a proper name and address" ^
    Customer(name = "Eric",
             address = Address(street = "Rose Crescent", number = 2)).
             retrieve(123)                                                          ^
                                                                                    e

}
```

As you also see above, named arguments can bring more readibility to the expected values.

Lazy cells

Fields, Props and Forms are added right away to a row when building a Form with the `tr` method. If it is necessary to add
them with a "call-by-name" behavior, the `lazyfy` method can be used:

```
def address = ... // build an Address Form
Form("Customer").
  tr(prop("name", customer.name)(name)).
  // the address Form will be built only when the Customer Form is rendered
  tr(lazyfy(address.actualIs(customer.address)))
```

Xml cells

Any xml can be "injected" on a row by using an `XmlCell`:

```
Form("Customer").
  tr(prop("name", customer.name)(name)).
  tr(XmlCell(<div><b>this is a bold statement</b><div>))
```

1-n relationships

When there are 1 - n relationships between entities the situation gets bit more complex.

For example you can have an "Order" entity, which has several "OrderLines". In that case there are several things that we might want to specify:

- the expected rows are included in the actual rows, with no specific order (this is the usual case)
- the expected rows are included in the actual rows, in the same order
- the expected rows are exactly the actual rows, with no specific order
- the expected rows are exactly the actual rows, in the same order

Let's see how to declare this. The 2 classes we're going to use are:

```
import Form._
import specification.Forms._

case class Order(orderId: Int) {
  lazy val actualLines = // those should be extracted from the actual order entity ret
    OrderLine("PIS", 1) ::
    OrderLine("PS", 2) ::
    OrderLine("BS", 3) ::
    OrderLine("SIS", 4) ::
    Nil

  def base = form("Order").th("name", "qty")
  def hasSubset(ls: OrderLine*)      = base.subset(actualLines, ls)
  def hasSubsequence(ls: OrderLine*) = base.subsequence(actualLines, ls)
  def hasSet(ls: OrderLine*)         = base.set(actualLines, ls)
  def hasSequence(ls: OrderLine*)    = base.sequence(actualLines, ls)
}

case class OrderLine(name: String, quantity: Int) {
  def form = tr(field(name), field(quantity))
}
```

The `OrderLine` class simply creates a form with 2 fields: name and quantity. The `Order` class is able to retrieve the actual order entity (say, from a database) and to extract `OrderLine` instances. It also has several methods to build Forms
depending on the kind of comparison that we want to do.

Subset

`Form.subset` uses the `FormDiffs.subset(a, b)` method to calculate the differences between the lines of a and b:

- lines existing in a but not b are left untouched
- lines existing in a and b are marked as success

- lines existing in b and not a are marked as failures

```
Order(123).hasSubset {
  OrderLine("BS", 3),
  OrderLine("PIS", 1),
  OrderLine("TDGL", 5)
}
```

This form returns:

Order	
name	**qty**
PIS	1
PS	2
BS	3
SIS	4
TDGL	5

Subsequence

`Form.subsequence` uses the `FormDiffs.subsequence(a, b)` method to calculate the differences and add them to the Form:

- lines existing in a but not b are left untouched
- lines existing in a and b in the same order are marked as success
- lines existing in b and not a are marked as failures

- lines existing in b and a but out of order are marked as failures

```
Order(123).hasSubsequence {
  OrderLine("PS", 2),
  OrderLine("BS", 3),
  OrderLine("PIS", 1),
  OrderLine("TDGL", 5)
}
```

This form returns:

Order	
name	**qty**
PIS	1
PS	2
BS	3
SIS	4
TDGL	5

Set

`Form.set` uses the `FormDiffs.set(a, b)` method to calculate the differences between the lines of a and b:

- lines existing in a but not b are marked as failures
- lines existing in a and b are marked as success

- lines existing in b and not a are marked as failures

```
Order(123).hasSet {
  OrderLine("BS", 3),
  OrderLine("PIS", 1),
  OrderLine("TDGL", 5)
}
```

This form returns:

Order	
name	**qty**
PIS	1
PS	2
BS	3
SIS	4
TDGL	5

Sequence

`Form.sequence` uses the `FormDiffs.sequence(a, b)` method to calculate the differences between the lines of a and b:

- lines existing in a but not b are marked as failures
- lines existing in a and b in the right order are marked as success

- lines existing in b and not a are marked as failures

```
Order(123).hasSequence {
  OrderLine("PS", 2),
  OrderLine("BS", 3),
  OrderLine("PIS", 1),
  OrderLine("TDGL", 5)
}
```

This form returns:

Order	
name	**qty**
PIS	1
PS	2
BS	3
SIS	4
TDGL	5

Decision tables

One very popular type of Forms are *decision tables*. A decision table is a Form where, on each row, several values are
used for a computation and the result must be equal to other values on the same row. A very simple example of this is a
calculator:

```
import Form._

case class Calculator(form: Form = Form()) {
  def tr(a: Int, b: Int, a_plus_b: Int, a_minus_b: Int) = Calculator {
    def plus = prop(a + b)(a_plus_b)
    def minus = prop(a - b)(a_minus_b)
    form.tr(a, b, plus, minus)
  }
}
object Calculator {
  def th(title1: String, titles: String*) = Calculator(Form.th(title1, titles:_*))
}
```

The `Calculator` object defines a `th` method to create the first `Calculator` Form, with the proper title. The `th` method:

- takes the column titles (there must be at least one title)
- creates a header row on the form
- returns a new Calculator containing this form (note that everything is immutable here)

The `Calculator` case class embeds a Form and defines a `tr` method which

- takes actual and expected values
- creates properties for the computations
- creates a form with a new row containing those fields and properties
- returns a new Calculator containing this form

And you use the `Calculator` Form like this:

```
class CalculatorSpecification extends Specification with Forms { def is  =
  "A calculator must add and subtract Ints" ^
    Calculator.
      th("a", "b", "a + b", "a - b").
      tr(1,   2,   3,       -1     ).
      tr(2,   2,   4,       0      )
}
```

Here is the output:

a	b	a + b	a - b
1	2	3	-1
2	2	4	0

And if something goes wrong:

a	b	a + b	a - b	
1	2	3	-1	
2	2	4	2	'0' is not equal to '2'

[click on failed cells to see the stacktraces]

And when it goes *very* wrong (like throwing an `error("very wrong")`), there will be red cells and stacktraces:

a	b	a + b	a - b	
1	2	3	-1	
2	2	4	2	very wrong

[click on failed cells to see the stacktraces]

Note that the Calculator class is not, in itself an Example. But there is an implicit definition automatically transforming `Any { def form: Form }` to `Example` so that an explicit call to `.form` is not necessary in order to include the Form in the
specification.

Total for specification Forms	
Finished in	295 ms
Results	11 examples, 170 expectations, 0 failure, 0 error

- User Guide
  - Quick Start
    - Unit
    - Acceptance
    - Execution
    - And much more!
  - Structure
    - Presentation
    - Declare examples
      - Styles
        - Acceptance specification
        - Unit specification
    - Results
      - Standard Matchers
    - Expectations
      - Functional
      - Thrown
      - All
        - Short-circuit
    - Pending until fixed
    - Auto-Examples
    - G/W/T
      - Sequencing
      - Extract methods
      - User regexps
      - Factory methods
      - G/W/T sequences
      - Multiple steps
      - ScalaCheck
      - Single step
      - Conversions
      - Unit specification
    - DataTables
  - Links
    - Inclusion
      - Inline
      - Html link
        - Html Link
    - Reference
    - Markdown url
  - Contexts

rs

y ways to define expectations in ***specs2***. You can define expectations with anything that returns a

d result

result

eck property

pectation

ple

esults

plest kind of result you can define for an expectation but also the least expressive!

mple:

```
hopefully true"          ! (1 != 2)
```

eful for simple expectations but a failure will give few information on what went wrong:

```
hopefully true"          ! (2 != 2) // fails with 'the value is false',...
```

results

l results can be used when you need specific result meanings:

s: the example is ok
e: there is a non-met expectation
r: a non-expected exception occurred
d: the example is skipped possibly at runtime because some conditions are not met. A more specific
e can
ed with `Skipped("my message")`
g: usually means "not implemented yet", but a specific message can be created with `Pending("my
e")`

results are also available to track the progress of features:

`Success` with the message "DONE"
`Pending` with the message "TODO"

ors

ators like `and`, `or`, `not` can be used to combine results. You can also use the `eventually` method to
ntil it is ok (this will actually work with anything convertible to a Result).

ults

- Isolation
- Scope
- Isolated variables
- Case classes
- Contexts inheritance
- Before/After
- In a mutable specification
- In an acceptance specification
- Around
- Outside
- AroundOutside
- BeforeExample
- Implicit context
- Composition
- Combinations
- Composition
- Steps/Actions
- Steps
- Actions
- Template
- For fragments
- Execution
- Layout
- Rules
- Formatting fragments
- Separating groups of examples
- Reset the levels
- Changing the indentation level
- Combinations
- Turning-off the automatic layout
- Unit specification
- Unit specifications
- How to?
- Declare arguments
- Pass arguments

e largest category of Results in ***specs2***. They cover many data types, can be composed and adapted
nes or be created from scratch by the user.

box

mon matchers are automatically available when extending the `Specification` trait:

Matchers

mmon type of matcher is ***beEqualTo*** to test for equality. There are different ways to use this matcher:

	Comment
qualTo(1)	the normal way
==(1)	with a shorter matcher
1	my favorite!
l 1	if you dislike underscores
= 1	for should lovers
	the ultimate shortcut
equalTo(1)	with a literate style

on
be equalTo(2)
2
qual 2
!=(2)

er types of equality:

	Comment
	same as be_== but can be used with some combinators like ^^^ or `toSeq` because the paramete
	checks if `(a:A) == (b:A)` when there is an implicit conversion from B (the type of b) to A (the ty
s	checks if a eq b (`a must be(b)` also works)
False	shortcuts for Boolean equality
	similar to a `===` b but will not typecheck if a and b don't have the same type

qualTo matcher is using the regular == Scala equality. However in the case of `Arrays`, Scala == is just us
or `Arrays`. So the beEqualTo matcher has been adapted to transform `Arrays` to `Seqs` before checking f
, 3) === Array(1, 2, 3)` (despite the fact that `Array(1, 2, 3) != Array(1, 2, 3)`).

rs can be used with Any objects:

e { case exp => ok }: to check if an object is like a given pattern (`ok` is a predefined value, `ko` is the
e { case exp => exp must beXXX }: to check if an object is like a given pattern, and verifies a cond
l
ullAs: when 2 objects must be null at the same time if one of them is null
Of(a, b, c): to check if an object is one of a given list
lass: to check the class of an object
uperclass: to check if the class of an object as another class as one of its ancestors
nterface: to check if an object is implementing a given interface
ignableFrom: to check if a class is assignable from another

nstanceOf[T]: to check if an object is an instance of type T

eral matchers to check Option and Either instances:

e checks if an element is Some(_)
e.which(function) checks if an element is Some(_) and satisfies a function returning a boolean
e.like(partial function) checks if an element is Some(_) and satisfies a partial function returning a
e checks if an element is None
oneAs checks if 2 values are equal to None at the same time
ht checks if an element is Right(_)
ht.like(partial function) checks if an element is Right(_) and satisfies a partial function returning
t checks if an element is Left(_)
t.like(partial function) checks if an element is Left(_) and satisfies a partial function returning a M

trings is very common. Here are the matchers which can help you:

ching (or be matching) checks if a string matches a regular expression
is a shortcut for beMatching(".*"+s+".*")
exp).withGroups(a, b, c) checks if some groups are found in a string
length checks the length of a string
size checks the size of a string (seen as an Iterable[Char])
pty checks if a string is empty
alTo(b).ignoreCase checks if 2 strings are equal regardless of casing
alTo(b).ignoreSpace checks if 2 strings are equal when trimmed
alTo(b).ignoreSpace.ignoreCase you can compose them
in(b) checks if a string contains another one
With(b) checks if a string starts with another one
th(b) checks if a string ends with another one

u need to do comparisons on Numerical values:

sThanOrEqualTo compares any Ordered type with <=

```
t be_<=(2)
t beLessThanOrEqualTo(2)
```

sThan compares any Ordered type with <

```
t be_<(2)
t beLessThan(2)
```

aterThanOrEqualTo compares any Ordered type with >=

```
t be_>=(1)
t beGreaterThanOrEqualTo(1)
```

aterThan compares any Ordered type with >

```
t be_>(1)
t beGreaterThan(1)
```

seTo checks if 2 Numerics are close to each other

```
ust beCloseTo(1, 0.5)
t be ~(5 +/- 2)
```

- Matchers
  - Boolean results
  - Standard results
  - Combinable matchers
  - Out of the box matchers
  - Optional
  - Custom matchers
  - With sequences
  - ScalaCheck
  - Arbitrary instances
  - With Generators
  - Test properties
  - Mock expectations
  - Creation and settings
  - Stubbing
  - Mocking and Stubbing at the same time
  - With matchers
  - Callbacks
- Add a title
- Use descriptions
- Enhance failures
- Share examples
- Create an index
- Tag examples
- In a unit specification
- Skip examples
- Debug statements
- Remove implicits

very compact ways of checking that some exceptions are thrown:

`A[ExceptionType]` checks if a block of code throws an exception of the given type
`A[ExceptionType](message = "boom")` additionally checks if the exception message is as expected
`A(exception)` or `throwAn(exception)` checks if a block of code throws an exception of the same type
nessage
`A[ExceptionType].like { case e => e must matchSomething }` or
`A(exception).like { case e => e must matchSomething }` allow to verify that the thrown exce
y
`A[ExceptionType](me.like { case e => e must matchSomething }` or
`A(exception).like { case e => e must matchSomething }` allow to verify that the thrown exce
y

ove matchers you can use `throwAn` instead of `throwA` if the exception name starts with a vowel for better

---

can be checked with several matchers:

k if a traversable is empty
ust be empty
1, 2, 3) must not be empty

k if some elements are contained in the traversable
1, 2, 3) must contain(3, 2)

k if some elements are contained in the traversable in the same order
1, 2, 3, 4) must contain(2, 4).inOrder

k if only some elements are contained in the traversable
4, 2) must contain(2, 4).only

k if only some elements are contained in the traversable and in the same order
2, 4) must contain(2, 4).only.inOrder

k if a sequence contains another one
2, 4) must containAllOf(List(4, 2))
2, 4) must containAllOf(List(2, 4)).inOrder

k if 2 sequences are contained in each other (like equality but with no order)
2, 4, 1) must containTheSameElementsAs(List(1, 4, 2))

k if a sequence contains any element of another one
2, 4) must containAnyOf(List(4, 2))

k the size of an iterable
1, 2) must have size(2)
1, 2) must have length(2) // equivalent to size

k if a `Traversable[String]` contains matching strings
"Hello", "World") must containMatch("ll") // matches with .*ll.*
"Hello", "World") must containPattern(".*llo") // matches with .*llo

k if a `Traversable[String]` contains matching strings, but only once
"Hello", "World") must containMatch("ll").onlyOnce

- Parameters for the answers function
- Verification
- Order of calls
- Ignoring stubs
- Spies
- Functions/PartialFunctions
- Auto-boxing
- Byname

- DataTables
- Forms

- Forms
  - Fields
  - Effects
  - Properties
  - Styles
  - Simple form
  - Adding several rows at once
  - Nesting a Form into another Form
  - Nesting a Form into an Effect or a Prop
  - Using tabs
  - Aggregating forms
  - Lazy cells
  - Xml cells
  - 1-n relationships
  - Subset
  - Subsequence
  - Set
  - Sequence
  - Decision tables
- Outside specs2

k if one of the elements has a given property

```
"Hello", "world") must have(_.size >= 5)
```

k if a traversable has the same elements as another one, regardless of the order, recursively (

```
"Hello", "world") must haveTheSameElementsAs(List("World", "Hello"))
"Hello", "world") must haveTheSameElementsAs(List("World", "Hello"), equalArray
```

uality method

k if a sequence is sorted (works with any type `T` which has an `Ordering`)

```
, 2, 3) must beSorted
```

versable matchers

and `haveTheSameElementsAs` matchers can be "adapted" to use a different notion of equality than == w
of elements in a traversable.

```
T) => Boolean) can be used instead of ==. For example:

, 2, 3) must contain(4, 3, 2) ^^ ((i: Int, j: Int) => i-j <= 1)
```

ach value in the first list is contained in the second list with possibly an error margin of 1.

```
> Matcher[Il) can be used to compare values with a matcher. For example:

qualIgnoreCase = (s: String) => be_==(s.toLowerCase)
Eric", "Bob") must contain("bob", "eric") ^^ equalIgnoreCase
```

=> S) can be used to compare values with a function. For example:

```
sersFromDb = Seq(User(id=1, name="eric"), User(id=2, name="Bob"))
FromDb must contain(User(id=0, name="eric"), User(id=0, name="Bob") ^^^ ((_:Use
```

operator used here is slightly different. It is ^^^ instead of simply ^^ because the same function is used to
same time. On the other hand the first 2 operators are more or less using a function taking 2 parameters.

---

eir own matchers as well, to check keys and values:

ey checks if a Map has a given key

```
-> "1") must haveKey(1)
```

eys checks if a Map has several keys

```
-> "1", 2 -> "2") must haveKeys(1, 2)
```

alue checks if a Map has a given value

```
-> "1") must haveValue("1")
```

alues checks if a Map has several values

```
-> "1", 2 -> "2") must haveValue("1", "2")
```

air checks if a Map has a given pair of values

```
-> "1") must havePair(1 -> "1")
```

airs checks if a Map has some pairs of values

```
->"1", 2->"2", 3->"3") must havePairs(1->"1", 2->"2")
```

- Via IDEA
- Via JUnit
- Via Eclipse
- Via Maven
- With your own Runner
- Arguments
- NotifierRunner
- In sbt
- Exporter
- In sbt

also Partial Functions, so:

```
inedAt checks if a PartialFunction is defined for a given value
al must beDefinedAt(1)

inedBy checks if a PartialFunction is defined for a given value
urns another one
al must beDefinedBy(1 -> true)
```

ul to have literal Xml in Scala, it is even more useful to have matchers for it!

```
alToIgnoringSpace compares 2 Nodes, without considering spaces
/></a> must ==/(<a> <b/></a>)
/></a> must beEqualToIgnoringSpace(<a> <b/></a>)

alToIgnoringSpace can also do an ordered comparison
=/( ).ordered
```

XPath like matcher matching if a node is a direct child of another

```
\("b")
```

n also check attribute names

```
\("b", "name")
```

tribute names and values as well (values are checked using a regular expression, use the quote method if y

```
\("b", "n"->"v", "n2"->"v\\d")
```

content of a `Text` node

```
must \("a") \> "hello" (alias textIs)
must \("a") \>~ "h.*" (alias textMatches)
```

uivalent of `\\` for a "deep" match is simply `\\`

```
\\("c")
```

- Philosophy
  - The origins
  - The score
    - Conciseness
    - Readability
    - Extensibility
    - Configuration
    - Clear implementation
    - User support
    - A test we hope
    - Functional / immutable
    - Chaining everything
    - Arguments have to be provided
    - Concurrency is a breeze
    - A simple structure
    - Contexts indentation
    - Operators forms
    - But if you STILL want mutable specifications

ble data format essentially modeling recursive key-values. There are 2 matchers which can be used to verify
alues in Strings representing Json documents:

`ue)` checks if a value is present at the root of the document. This can only be the case if that document is a

`-> value)` checks if a pair is present at the root of the document. This can only be the case if that docun

`lue)` checks if a value is present anywhere in the document, either as an entry in an Array, or as the value

`value)` checks if a pair is present anywhere in a Map of the document

esting part comes from the fact that those matchers can be chained to search specific paths in the Json doc
he following document:

```
 from an example in the Lift project
on = {
n": {
e": "Joe",
": 35,
use": {
```

```
erson": {
"name": "Marilyn",
"age": 38
```

hese combinations:

```
ust /("person") */("person") /("age" -> 33.0) // by default numbers are parsed
```

for files is more or less mimicked as matchers which can operate on strings denoting paths or on Files:

- `alToIgnoringSep` checks if 2 paths are the same regardless of their separators
  `"c:\\\\temp\\\\hello" must beEqualToIgnoringSep("c:/temp/hello")`
- `beAnExistingPath` checks if a path exists
- `beAReadablePath` checks if a path is readable
- `beAWritablePath` checks if a path is writable
- `beAnAbsolutePath` checks if a path is absolute
- `beAHiddenPath` checks if a path is hidden
- `beAFilePath` checks if a path is a file
- `beADirectoryPath` checks if a path is a directory
- `havePathName` checks if a path has a given name
- `haveAsAbsolutePath` checks if a path has a given absolute path
- `haveAsCanonicalPath` checks if a path has a given canonical path
- `haveParentPath` checks if a path has a given parent path
- `listPaths` checks if a path has a given list of children
- `exist` checks if a file existsy

- `beReadable` checks if a file is readable

- `beWritable` checks if a file is writable
- `beAbsolute` checks if a file is absolute
- `beHidden` checks if a file is hidden
- `beAFile` checks if a file is a file
- `beADirectory` checks if a file is a directory
- `haveName` checks if a file has a given name
- `haveAbsolutePath` checks if a file has a given absolute path
- `haveCanonicalPath` checks if afile has a given canonical path
- `haveParent` checks if a file has a given parent path
- `haveList` checks if a file has a given list of children

A few matchers can help us check the contents of files or actually anything containing lines of Strings. We can check that
same lines:

- `(file1, file2) must haveSameLines`
- `file1 must haveSameLinesAs(file2)`

We can check that the content of one file is contained in another one:

- `file1 must containLines(file2)`

LinesContent

Files are not the only possible source of lines and it is useful to be able to check the content of a `File` with a `Seq[Stri`

- `file1 must haveSameLinesAs(Seq(line1, line2, line3))`

This is because those 2 types implement the `org.specs2.text.LinesContent` trait, defining:

- a name for the overall content
- a method for returning the lines
- a default method for computing the differences of 2 sequences of lines (in case you need to override this logic)

So if you have a specific type `T` which you can represent as a `Seq[String]`, you can create an implicit `LinesContent` able to use the `ContentMatchers`:

```
implicit val linesforMyType: LinesContent[T] = new LinesContent[T] {
  def name(t: T) = "My list of lines"
  def lines(t: T): Seq[String] = ... // your implementation goes here
}
```

Order

It is possible to relax the constraint by requiring the equality or containment to be true regardless of the order of lines:

- `(file1, file2) must haveSameLines.unordered`
- `file1 must haveSameLinesAs(file2).unordered`
- `file1 must containLines(file2).unordered`

Missing only

By default, `(file1, file2) must haveSameLines` will report misplaced lines if any, that is, lines of `f1` which appea
the right position. However if `file2` is big, this search might degrade the performances. In that case you can turn it off w

`(file1, file2) must haveSameLines.missingOnly`

Show less differences

If there are too many differences, you can specify that you only want the first 10:

- `(file1, file2) must haveSameLines.showOnly(10.differences).unordered`

In the code above `10.differences` builds a `DifferenceFilter` which is merely a filtering function: `(lines1: Seq lines2: Seq[String]) => (Seq[String], Seq[String])`. The parameter `lines1` is the sequence of lines not second content while `lines2` is the sequence of lines not found in the first content.

The examples above show how to use matchers:

- the general form for using a matcher is: `a must matcher`
- but can use `should` instead of `must` if you prefer
- for most matchers you can use a form where the `be` word (or the `have` word) is detached
- you can as well negate a matcher by adding `not` before it (or after it, as a method call)

Optional

These other matchers need to be selectively added to the specification by adding a new trait:

Optional Matchers

That's only if you want to match the result of other matchers!

```
// you need to extend the ResultMatchers trait
class MatchersSpec extends Specification with ResultMatchers { def is =
  "beMatching is using a regexp" ! {
    ("Hello" must beMatching("h.*")) must beSuccessful
  }
}
```

In the rare case where you want to use the Scala interpreter and execute a script:

```
class ScalaInterpreterMatchersSpec extends Specification with ScalaInterpreterMatchers
  def interpret(s: String): String = // you have to provide your own Scala interpreter

    "A script" can {
      "be interpreted" in {
        "1 + 1" >| "2"
      }
    }
}
```

Scala provides a parsing library using parser combinators.

You can specify your own parsers by:

- extending the `ParserMatchers` trait
- associating the `val parsers` variable with your parsers definition
- using the `beASuccess`, `beAFailure`, `succeedOn`, `failOn`, `errorOn` matchers to specify the results of parsing strings. `beAPartialSuccess`, `be aPartialSuccess`, `succeedOn.partially` will allow a successful match the input
- using `haveSuccessResult` and `haveFailureMsg` to specify what happens *only* on success or failure. Those m a String or a matcher so that
  . `haveSuccessResult("r") <==> haveSuccessResult(beMatching(".*r.*") ^^ ((_:Any).toStr`
  . `haveFailingMsg("m") <==> haveFailingMsg(beMatching(".*r.*"))`

For example, specifying a Parser for numbers could look like this:

```
import util.parsing.combinator.RegexParsers
import NumberParsers.{number, error}

class ParserSpec extends Specification with matcher.ParserMatchers {  def is =
  "Parsers for numbers"

    "beASuccess and succeedOn check if the parse succeeds"
    { number("1") must beASuccess }
    { number("1i") must beAPartialSuccess }
    { number must succeedOn("12") }
    { number must succeedOn("12ab").partially }
    { number must succeedOn("12").withResult(12) }
    { number must succeedOn("12").withResult(equalTo(12)) }
    { number("1") must haveSuccessResult("1") }

    "beAFailure and failOn check if the parse fails"
    { number must failOn("abc") }
    { number must failOn("abc").withMsg("string matching regex.*expected") }
    { number must failOn("abc").withMsg(matching(".*string matching regex.*expected.*")
    { number("i") must beAFailure }
    { number("i") must haveFailureMsg("i' found") }

    "beAnError and errorOn check if the parser errors out completely"
    { error must errorOn("") }
    { error("") must beAnError }
```

```
  val parsers = NumberParsers
}

object NumberParsers extends RegexParsers {
  /** parse a number with any number of digits */
  val number: Parser[Int] = "\\\\d+".r ^^ {_.toInt}
  /** this parser returns an error */
  val error: Parser[String] = err("Error")
}
```

Sometimes you just want to specify that a block of code is going to terminate. The `TerminationMatchers` trait is here mix in that trait, you can write:

```
Thread.sleep(100) must terminate

// the default is retries=0, sleep=100.millis
Thread.sleep(100) must terminate(retries=1, sleep=60.millis)
```

Note that the behaviour of this matcher is a bit different from the `eventually` operator. In this case, we let the current T during the given `sleep` time and then we check if the computation is finished, then, we retry for the given number of `ret`

In a further scenario, we might want to check that triggering another action is able to unblock the first one:

```
action1 must terminate.when(action2)
action1 must terminate.when("starting the second action", action2)
action1 must terminate(retries=3, sleep=100.millis).when(action2)
```

When a second action is specified like that, `action1` will be started and `action2` will be started on the first retry. Othen to specify that `action1` can *only* terminate when `action2` is started, you write:

```
action1 must terminate.onlyWhen(action2)
```

It is highly desirable to have acyclic dependencies between the packages of a project. This often leads to describing the structure as "layered": each package on a layer can only depend on a package on a lower layer. ***specs2*** helps you enfor property with specific matchers.

Layers definition

First you need to define the packages and their expected dependencies. Mix-in the `org.specs2.specification.Ana` define, (taking ***specs2*** as an example):

```
layers (
  "runner",
  "reporter",
  "specification mutable",
  "mock      form",
  "matcher",
  "execute",
  "reflect    xml  time html",
  "collection control io text main data").withPrefix("org.specs2")
```

The above expression defines layers as an ordered list of `String`s containing space-separated package names. It is sup `withPrefix` declaration to factor out the common package prefix between all these packages.

By default, the packages are supposed to correspond to directories in the `src/target/scala-<version>/classes` project has a different layout you can declare another target directory:

```
  layers(...).inTargetDir("out/classes")
```

Inclusion/Exclusion

Every rule has exceptions :-). In some rare cases, it might be desirable to exclude a class from being checked on a given this, you can use the `include/exclude` methods on the `Layer` class:

```
  layers (
    "runner",
    "reporter",
    "specification mutable".exclude("mutable.SpecificationWithJUnit"),
    "mock      form",
    "matcher",
    "execute",
    "reflect  xml  time html",
    "collection control io text main data").withPrefix("org.specs2")
```

The `include/exclude` methods accept a list of regular expressions to:

- exclude fully qualified class names (generally, only `exclude` will be necessary)
- re-include fully qualified class names if the exclusion list is to big

Verification

Now you've defined layers, you can use the `beRespected` matcher to check if all the dependencies are verified:

```
 val design = layers(...)
 design must beRespected
```

If some dependencies are not respected:

```
  those dependencies are not satisfied:
  org.specs2.main x-> org.specs2.io because org.specs2.io.FileSystem -> org.specs2.main.A
  org.specs2.main x-> org.specs2.io because org.specs2.io.FileSystem -> org.specs2.main.A
```

Layers as an Example

The `Analysis` trait allows to directly embed the layers definition in a `Specification` and turn it into an `Example`:

```
  class DependenciesSpec extends Specification with Analysis { def is =
    "this is the application design" ^
      layers(
        "gui commandline",
        "controller",
        "backend"
      )
  }
```

Alternative implementation

Another implementation of the same functionality is available through the `org.specs2.analysis.CompilerDepende` trait. This implementation uses the compiler dependency analysis functionality but needs more time, since it recompiles t

The source files are taken from the `src/main/scala` directory by default but you can change this value by using the `Layers.inSourceDir` method.

While this implementation is slower than the Classycle one, it might retrieve more dependencies, for example when cons in class files.

Note: since this functionality relies on the scala compiler library, so you need to add it to your build file:

```
// use sbt's scalaVersion Setting to define the scala-compiler library version
libraryDependencies <<= scalaVersion { scala_version => Seq(
  "org.specs2" %% "specs2" % "1.10" % "test",
  "org.scala-lang" % "scala-compiler" % scala_version % "test")
}
```

Custom

There are many ways to create matchers for your specific usage. The simplest way is to reuse the existing ones:

- using logical operators

```
def beBetween(i: Int, j: Int) = be_>=(i) and be_<=(j)

// create a Seq Matcher from a Matcher
def allBeGreaterThan2: Matcher[Seq[Int]]   = be_>=(2).forall     // fail after the fi
def allBeGreaterThan3: Matcher[Seq[Int]]   = be_>=(2).foreach    // like forall but e
def haveOneGreaterThan2: Matcher[Seq[Int]] = be_>=(2).atLeastOnce
```

- adapting the actual value

```
// This matcher adapts the existing `be_<=` matcher to a matcher applicable to `Any`
def beShort = be_<=(5) ^^ { (t: Any) => t.toString.size }
def beShort = be_<=(5) ^^ { (t: Any) => t.toString.size aka "the string size" }

// !!! use a BeTypedEqualTo matcher when using aka and equality !!!
def beFive = be_===(5) ^^ { (t: Any) => t.toString.size aka "the string size" }

// The adaptation can also be done the other way around when it's more readable
def haveExtension(extension: =>String) = ((_:File).getPath) ^^ endWith(extension)
```

- adapting the actual and expected values. This matcher compares 2 `Human` objects but set their `wealth` field to 0
  so that the equals method will not fail on that field:

```
def beMostlyEqualTo = (be_==(_:Human)) ^^^ ((_:Human).copy(wealth = 0))
// then
Human(age = 20, wealth=1000) must beMostlyEqualTo(Human(age = 20, wealth=1)) toResult
```

- using `eventually` to try a match a number of times until it succeeds:

```
val iterator = List(1, 2, 3).iterator
iterator.next must be_==(3).eventually
// Use eventually(retries, n.millis) to use another number of tries and waiting time
```

- using `when` or `unless` to apply a matcher only if a condition is satisfied:

```
1 must be_==(2).when(false)                       // will return a success
1 must be_==(2).unless(true)                      // same thing

1 must be_==(2).when(false, "don't check this")   // will return a success
1 must be_==(2).unless(true, "don't check this")  // same thing
```

- using `iff` to say that a matcher must succeed if and only if a condition is satisfied:

```
1 must be_==(1).iff(true)                        // will return a success
1 must be_==(2).iff(true)                        // will return a failure
1 must be_==(2).iff(false)                       // will return a success
1 must be_==(1).iff(false)                       // will return a failure
```

- using `orSkip` to return a `Skipped` result instead of a Failure if the condition is not met

```
1 must be_==(2).orSkip
1 must be_==(2).orSkip("Precondition failed")    // prints "Precondition failed: '1'
1 must be_==(2).orSkip((ko:String) => "BAD "+ko) // prints "BAD '1' is not equal to '
```

- using `orPending` to return a `Pending` result instead of a Failure if the condition is not met

```
1 must be_==(2).orPending
1 must be_==(2).orPending("Precondition failed")    // prints "Precondition failed: '
1 must be_==(2).orPending((ko:String) => "BAD "+ko) // prints "BAD '1' is not equal t
```

- using `mute` to change a Matcher so that it returns MatchResults with no messages. This is used in Forms to create
  properties showing no messages when they fail

Another easy way to create matchers, is to use some implicit conversions from functions to Matchers:

```
val m: Matcher[String]  = ((_: String).startsWith("hello"), "doesn't start with hello")
val m1: Matcher[String] = ((_: String).startsWith("hello"), "starts with hello", "doesn
val m2: Matcher[String] = ((_: String).startsWith("hello"), (s:String) => s+ " doesn't
val m3: Matcher[String] = ((_: String).startsWith("hello"), (s:String) => s+ " starts w
val m4: Matcher[String] = (s: String) => (s.startsWith("hello"), s+" doesn't start with
val m5: Matcher[String] = (s: String) => (s.startsWith("hello"), s+ "starts with hello"
```

And if you want absolute power over matching, you can define your own matcher:

```
class MyOwn extends Matcher[String] {
  def apply[S <: String](s: Expectable[S]) = {
    result(s.value.isEmpty,
           s.description + " is empty",
           s.description + " is not empty",
           s)
  }
}
```

In the code above you have to:

- define the `apply` method (and its somewhat complex signature)

- use the protected `result` method to return: a Boolean condition, a message when the match is ok, a message when the
  match is not ok, the "expectable" value. Note that if you change the expectable value you need to use the `map` method
  on the `s` expectable (`s.map(other)`). This way you preserve the ability of the Expectable to throw an Exception if
  a subsequent match fails

- you can use the `description` method on the `Expectable` class to return the full description of the expectable including
  the optional description you setup using the `aka` method

With sequences

If you have the same "MatchResult" expression that you'd like to verify for different values you can write one of the following:

```
// stop after the first failure
((_:Int) must be_>(2)).forall(Seq(3, 4, 5))
```

```
forall(Seq(3, 4, 5)) ((_:Int) must be_>(2))
// check only the elements defined for the partial function
forallWhen(Seq(3, 10, 15)) { case a if a > 3 => a must be_>(5) }

// try to match all values and collect the results
((_:Int) must be_>(2)).foreach(Seq(3, 4, 5))
foreach(Seq(3, 4, 5)) ((_:Int) must be_>(2))
// check only the elements defined for the partial function
foreachWhen(Seq(3, 10, 15)) { case a if a > 3 => a must be_>(5) }

// succeeds after the first success
((_:Int) must be_>(2)).atLeastOnce(Seq(3, 4, 5))
atLeastOnce(Seq(3, 4, 5)) ((_:Int) must be_>(2))
// check only the elements defined for the partial function
atLeastOnceWhen(Seq(3, 4, 10)) { case a if a > 3 => a must be_>(5) }
```

ScalaCheck

A clever way of creating expectations in ***specs2*** is to use the ScalaCheck library.

To declare ScalaCheck properties you first need to extend the `ScalaCheck` trait. Then you can pass functions returning any kind of `Result` (`Boolean`, `Result`, `MatchResult`) to the `prop` method and use the resulting `Prop` as your example body:

```
"addition and multiplication are related" ! prop { (a: Int) => a + a == 2 * a }
```

The function that is checked can either return:

```
// a Boolean
"addition and multiplication are related" ! prop { (a: Int) => a + a == 2 * a }

// a MatchResult
"addition and multiplication are related" ! prop { (a: Int) => a + a must_== 2 * a }

// a Prop
"addition and multiplication are related" ! prop { (a: Int) => (a > 0) ==> (a + a must_=
```

Note that if you pass functions using `MatchResults` you will get better failure messages so you are encouraged to do so.

Arbitrary instances

By default ScalaCheck uses `Arbitrary` instances taken from the surrounding example scope. However you'll certainly need to generate your own data from time to time. In that case you can create an `Arbitrary` instance and make sure it is in the scope of the function you're testing:

```
// this arbitrary will be used for all the examples
implicit def a = Arbitrary { for { a <- Gen.oneOf("a", "b"); b <- Gen.oneOf("a", "b") }

"a simple property" ! ex1

def ex1 = check((s: String) => s must contain("a") or contain("b"))
```

You can also be very specific if you want to use an `Arbitrary` instance only on one example. In that case, just replace the `check` method with the name of your `Arbitrary` instance:

```
"a simple property"       ! ex1
"a more complex property" ! ex2
```

```
implicit def abStrings = Arbitrary { for { a <- Gen.oneOf("a", "b"); b <- Gen.oneOf("a",
def ex1 = abStrings((s: String) => s must contain("a") or contain("b"))

// use a tuple if there are several parameters to your function
def ex2 = (abStrings, abStrings)((s1: String, s2: String) => s must contain("a") or cont
```

With Generators

ScalaCheck also allows to create `Props` directly with the `Prop.forAll` method accepting `Gen` instances:

```
"a simple property"       ! ex1
"a more complex property" ! ex2

def abStrings = for { a <- Gen.oneOf("a", "b"); b <- Gen.oneOf("a", "b") } yield a+b

def ex1 = forAll(abStrings) { (s: String) => s must contain("a") or contain("b") }
def ex2 = forAll(abStrings, abStrings) { (s1: String, s2: String) => s must contain("a")
```

Test properties

ScalaCheck test generation can be tuned with a few properties. If you want to change the default settings, you have to use implicit values:

```
implicit val params = set(minTestsOk -> 20) // use display instead of set to get additio
```

It is also possible to specifically set the execution parameters on a given property:

```
"this is a specific property" ! prop { (a: Int, b: Int) =>
  (a + b) must_== (b + a)
}.set(minTestsOk -> 200, workers -> 3)
```

The parameters you can modify are:

- `minTestsOk`: minimum of tests which must be ok before the property is ok (default=100)
- `maxDiscarded`: if the data generation discards too many values, then the property can't be proven (default=500)
- `minSize`: minimum size for the "sized" data generators, like list generators (default=0)
- `maxSize`: maximum size for the "sized" data generators (default=100)
- `workers`: number of threads checking the property (default=1)

Mock expectations

At the moment only the Mockito library is supported.

Mockito allows to specify stubbed values and to verify that some calls are expected on your objects. In order to use those functionalities, you need to extend the `org.specs2.mock.Mockito` trait:

```
import org.specs2.mock._
class MockitoSpec extends Specification { def is =

  "A java list can be mocked"                                                    ^
    "You can make it return a stubbed value"                                     ! c().s
    "You can verify that a method was called"                                    ! c().v
    "You can verify that a method was not called"                                ! c().v
                                                                                 end

  case class c() extends Mockito {
```

```
    val m = mock[java.util.List[String]] // a concrete class would be mocked with: mock[
    def stub = {
      m.get(0) returns "one"             // stub a method call with a return value
      m.get(0) must_== "one"             // call the method
    }
    def verify = {
      m.get(0) returns "one"             // stub a method call with a return value
      m.get(0)                           // call the method
      there was one(m).get(0)            // verify that the call happened
    }
    def verify2 = there was no(m).get(0) // verify that the call never happened
  }
}
```

Creation and settings

Mockito offers the possibility to provide specific settings for the mock being created:

- its name

  val m = mock[List[String]].as("list1")

- "smart" return values

  val m = mock[List[String]].smart

- specific return values

  val m = mock[List[String]].defaultReturn(10)

- specific answers

  // a function InvocationOnMock => V is used in place of the org.mockito.stubbing.Answer type for better conciseness
  val helloObject = (p1: InvocationOnMock) => "hello "+p1.toString
  val m = mock[List[String]].defaultAnswer(helloObject)

- extra interfaces

  val m = mock[List[String]].extraInterface[Cloneable]
  val m = mock[List[String]].extraInterfaces(classesOf[Cloneable, Serializable])

Now, if you want to combine several of those settings together you need to call the `settings` method:

```
val m = mock[List[String]].settings(name = "list1",
                                    defaultReturn = 10,
                                    extraInterfaces = classesOf[Cloneable, Serializable]
// or
val m = mock[List[String]].settings(smart = true,
                                    extraInterface = classeOf[Cloneable]))
```

Finally, in case the Mockito library gets new settings, you can declare the following:

```
val settings = org.mockito.Mockito.withSettings
val m = mock[List[String]](settings)
```

Stubbing

Stubbing values is as simple as calling a method on the mock and declaring what should be returned or thrown:

```
m.get(1) returns "one"
m.get(2) throws new RuntimeException("forbidden")
```

You can specify different consecutive returned values by appending thenReturns or thenThrows:

```
m.get(1) returns "one" thenReturns "two"
m.get(2) throws new RuntimeException("forbidden") thenReturns "999"
```

Mocking and Stubbing at the same time

It is also possible to create a mock while stubbing one of its methods, provided that you declare the type of the expected mock:

```
val mocked: java.util.List[String] = mock[java.util.List[String]].contains("o") returns
mocked.contains("o") must beTrue
```

With matchers

The built-in Mockito argument matchers can be used to specify the method arguments for stubbing:

```
m.get(anyInt()) returns "element"
m.get(999) must_== "element"
```

specs2 matchers can also be passed directly as arguments:

```
m.get(===(123)) returns "one"
```

Callbacks

In some rare cases, it is necessary to have the return value depend on the parameters passed to the mocked method:

```
m.get(anyInt) answers { i => "The parameter is " + i.toString }
```

The function passed to answers will be called with each parameter passed to the stubbed method:

```
m.get(0)          // returns "The parameter is 0"
m.get(1)          // the second call returns a different value: "The parameter is 1"
```

Parameters for the answers function

Because of the use of reflection the function passed to answers will receive only instances of the `java.lang.Object` type.

More precisely, it will:

- pass the mock object if both the method has no parameters and the function has one parameter:
  `mock.size answers { mock => mock.hashCode }`
- pass the parameter if both the method and the function have one parameter:
  `mock.get(0) answers { i => i.toString }`
- pass the parameter and the mock object if the method has 1 parameter and the function has 2:
  `mock.get(0) answers { (i, mock) => i.toString + " for mock " + mock.toString }`

In any other cases, if `f` is a function of 1 parameter, the array of the method parameters will be passed and if the function has 2 parameters, the second one will be the mock.

Verification

By default Mockito doesn't expect any method to be called. However if your writing interaction-based specifications you want to specify that some methods are indeed called:

```
there was one(m).get(0)             // one call only to get(0)
there was no(m).get(0)              // no calls to get(0)

// were can also be used
there were two(m).get(0)            // 2 calls exactly to get(0)
```

```
there were three(m).get(0)          // 3 calls exactly to get(0)
there were 4.times(m).get(0)        // 4 calls exactly to get(0)

there was atLeastOne(m).get(0)      // at least one call to get(0)
there was atLeastTwo(m).get(0)      // at least two calls to get(0)
there was atLeastThree(m).get(0)    // at least three calls to get(0)
there was atLeast(4)(m).get(0)      // at least four calls to get(0)

there was atMostOne(m).get(0)       // at most one call to get(0)
there was atMostTwo(m).get(0)       // at most two calls to get(0)
there was atMostThree(m).get(0)     // at most three calls to get(0)
there was atMost(4)(m).get(0)       // at most four calls to get(0)
```

It is also possible to add all verifications inside a block, when several mocks are involved:

```
got {
  one(m).get(0)
  two(m).get(1)
}
```

Order of calls

The order of method calls can be checked by creating calls and chaining them with then:

```
val m1 = mock[List[String]]
val m2 = mock[List[String]]

m1.get(0)
m1.get(0)
m2.get(0)

there was one(m1).get(0) then one(m1).get(1)

// when several mocks are involved, the expected order must be specified as an implicit
implicit val order = inOrder(m1, m2)
there was one(m1).get(0) then one(m2).get(0)
```

Ignoring stubs

When specifying the behavior of an object in relation to others you may want to verify that some mocks have been called as collaborators and you don't really want to specify what happens to other mocks because they are just playing the role of stubs.

In this case the `ignoreStubs` method can be used:

```
val (stub1, stub2) = (mock[AStub], mock[AStub])
...
...
there were noMoreCallsTo(ignoreStubs(stub1, stub2))
```

This method is also available with the `inOrder` method:

```
implicit val order = inOrder(ignoreStubs(list1, list2))
```

For more documentation about this Mockito functionality, please read here.

Spies

Spies can be used in order to do some "partial mocking" of real objects:

```
val spiedList = spy(new LinkedList[String])

// methods can be stubbed on a spy
spiedList.size returns 100
```

```
  // other methods can also be used
  spiedList.add("one")
  spiedList.add("two")

  // and verification can happen on a spy
  there was one(spiedList).add("one")
```

However, working with spies can be tricky:

```
 // if the list is empty, this will throws an IndexOutOfBoundsException
 spiedList.get(0) returns "one"
```

As advised in the Mockito documentation, doReturn must be used in that case:

```
 doReturn("one").when(spiedList).get(0)
```

Functions/PartialFunctions

It is possible to verify method calls where parameters are functions by specifying how the passed function will react to a given set of arguments.
Given the following mock:

```
  trait Amount {
    // a method showing an amount precision
    def show(display: Function2[Double, Int, String, String]) = ...
  }
  val amount = mock[Amount]
```

If the mock is called with this function:

```
  amount.show((amount: Double, precision: Int) => "%2."+precision+"f" format amount)
```

Then it is possible to verify how the mock was called:

```
  // with sample arguments for the function and the expected result
  there was one(amount).show((32.4456, 2) -> "32.45")

  // with a matcher for the result
  there was one(amount).show((32.4456, 2) -> endWith("45"))

  // with any Function2[A, B, R]
  there was one(amount).show(anyFunction2)
```

Auto-boxing

Auto-boxing might interfere with the mocking of PartialFunctions. Please have a look at this for a discussion.

Byname

Byname parameters can be verified but this will not work if the specs2 jar is not put first on the classpath, before the mockito jar. Indeed specs2 redefines a Mockito class for intercepting method calls so that byname parameters are properly handled.

DataTables

DataTables are a very effective way of grouping several similar examples into one. For example, here is how to specify the addition of integers by providing one example on each row of a table:

```
  class DataTableSpec extends Specification with DataTables { def is =
    "adding integers should just work in scala"  ! e1

    def e1 =
```

```
     "a"   | "b" | "c" |                                  // the header of the table, wi
      2    !  2  !  4  |                                  // an example row
      1    !  1  !  2  |> {                               // the > operator to "execute"
      (a, b, c) =>  a + b must_== c                        // the expectation to check on
    }
  }
```

Note that there may be implicit definition conflicts when the first parameter of a row is a String. In that case you can use the `!!` operator to disambiguate (and `||` in the header for good visual balance).

Forms

Forms are a way to represent domain objects or service, and declare expected values in a tabular format. They are supposed to be used with the HtmlRunner to get human-readable documentation.

Forms can be designed as reusable pieces of specification where complex forms can be built out of simple ones.


Here's how to use Forms

Outside specs2

The ***specs2*** matchers are a well-delimited piece of functionality that you should be able to reuse in your own test framework. You can reuse the following traits:

- `org.specs2.matcher.MustMatchers` (or `org.specs2.matcher.ShouldMatchers`) to write anything like `1 must be_==(1)` and
  get a `Result` back
- You can also use the side-effecting version of that trait called
  `org.specs2.matcher.MustThrownMatchers` (or `ShouldThrownMatchers`).
  It throws a `FailureException` as soon as an expectation is failing. Those traits can also be used in a regular Specification if you have several expectations per example and if you don't want to chain them with `and`.
- Finally, in a JUnit-like library you can use the `org.specs2.matcher.JUnitMustMatchers` trait which throws
  `AssertionFailureErrors`

Without any dependency on specs2

The Testing page of the ***spray*** project explains how you can define a testing trait in your library which can be used with specs2 or scalatest or any framework defining the following methods:

- `fail(msg: String): Nothing`
- `skip(msg: String): Nothing`

In specs2, those 2 methods are defined by the `org.specs2.matcher.ThrownMessages` trait

```
trait ThrownMessages { this: ThrownExpectations =>
  def fail(m: String): Nothing = failure(m)
  def skip(m: String): Nothing = skipped(m)
}
```

Total for specification Matchers	

Finished in	815 ms
Results	47 examples, 536 expectations, 0 failure, 0 error

- User Guide
  - Quick Start
  - Structure
    - Presentation
    - Declare examples
      - Styles
        - Acceptance specification
        - Unit specification
      - Results
        - Standard Matchers
        - Expectations
        - Functional matchers
        - All
        - Pending until fixed
      - Auto-Examples
      - G/W/T
        - Sequencing
        - Extract methods
        - User regexps
        - Factory methods
        - G/W/T sequences
        - Multiple steps
        - ScalaCheck
        - Single step
        - Conversions
        - Unit specification
      - DataTables
    - Links
      - Inclusion
        - Inline
        - Html link
          - Html Link
      - Referencing
        - Markdown
        - url
    - Contexts

ophy

- Unit
- Acceptance
- Execution
- And much more!

en created as an evolution of the ***specs*** project.

as learning project to explore Scala's DSL possibilities for doing Behaviour-Driven-Development (BDD).
y first objectives of specs were:

eness: it should be possible to express a software specification with the least amount of ceremony

lity: an *executable* specification should read like a purely textual one, by a non-programmer

bility: it should be possible to add new matchers, new runners,...

ration: there should sensible defaults and an easy way to override those values

plementation: since this is an open-source project with no business constraint, there's no excuse for
ng a crystal clear implementation, right?

ser support: it's not because something is free that it should be buggy! Moreover this is also a good
he design. A good design should be easy to fix and evolve

rs of use, let's have a look at what worked and what didn't.

s

was achieved thanks to the incredible power of implicits in Scala which provide lots of way to create an
for specifying software. However, the implementation of that syntactic support has several drawbacks:

rovides implicits by inheriting methods from the `Specification` class. While this is very convenient
so considerably polluting the namespace of the user code *inside* the specification. This pollution leads
ble bugs (an implicit conversion being applied when you don't expect it) and possible namespace

eserved" words like `should`, `can`, `in` come from previous BDD libraries like rspec. But they are not
convenient and sometimes one wants to write `"my example should provide"` just to avoid having
xample under `""my example" should"` start with `"provide..."`. There is a way to do this in ***specs***
requires the creation of an ad-hoc method

eople like to structure their specifications with other keywords like Given-When-Then which require
al methods in the library for no more added value than just displaying those words

of a specification written with ***specs*** largely depends on the writer of the specification. Since a
created by interleaving text and code, if the amount of code is too large then the textual content of the
largely lost and the developer cannot read it with just one glance.

s prototyped in ***specs*** to alleviate this issue: LiterateSpecifications. The idea was to use Scala support
o allow the writer to write pure text and insert, at the right places, some executable examples. It turns
lementation of this approach is fairly different from what was done for "regular" examples and cluttered
gn.

elease, ***specs*** has been extended in different ways. Many users have created their own matchers (
ave been integrated to the main library).

and, the additional runners which were developed for TeamCity or sbt were written by me, as well as
damental enhancements, like SpecificationContexts.

we can draw on this subject is that it's difficult to design something that's truly extensible without any
re requirements!

n

defaults have been debated and some "opinionated" decisions have been taken, like the decision to
mple without expectation as "Pending" for example. Unfortunately the configuration mechanism offered
change the defaults) was not really appealing (a mix of properties-file-reflection-system-properties)
v evidence that anyone actually used it.

mentation

was certainly *not* achieved. There are several reasons for this. The design of the examples execution is
ain one.

xamples are executed "on-demand" when a runner "asks" the specification for its successes and
ecification then asks each example for its status and an example knows that by executing himself. The
xample doesn't really have enough information to know it's full execution context: is there some code
rehand for data setup? Or after all other examples, because it's the last example of the specification
disconnection is required then?

ggravated by a "magic" feature provided by ***specs***: the automatic reset of local variables.
very convenient for the user but the implementation is a nightmare! In order to make as if each
xecuted in isolation, as if other examples did not modify surrounding variables, a new specification is
t one example is executed inside that specification. This works, but at the expense of carefully copying
m the cloned specification to the original one. More than *20* issues were created because of that only

t

rt has always been responsive, in terms of bug fixes and enhancements. However the object-oriented
s with lots of variables and side-effects around made some bugs difficult to diagnose and made some
downright impossible. The best example of an "impossible" feature to implement is the concurrent
amples.
riables all around the place, there's little chance to ever get it right.

pe

f ***specs2*** was precisely started to fight the complexities and issues of ***specs***. In order to do that while
to the original vision for ***specs***, a new design compromise was necessary with new design principles:

use mutable variables!
mple structures
the dependencies (no cycles)

- Isolation
- Scope
- Isolated variables
- Case classes
- Contexts inheritance
- Before/After
- In a mutable specification
- In an acceptance specification
- Around
- Outside
- AroundOutside
- BeforeExample
- Implicit context
- Composition
- Combinations
- Composition
- Steps/Actions
- Steps
- Actions
- Template
- For examples
- Execution
- Layout
- Rules
- Formatting fragments
- Separating groups of examples
- Reset the levels
- Changing the indentation level
- Combinations
- Turning-off the automatic layout
- Unit specification
- Unit specifications
- How to?
- Declare arguments
- Pass arguments

- Add a title
- Use descriptions
- Enhance failures
- Share examples
- Create an index
- Tag examples
- Run a unit specification
- Skip examples
- Debug statements
- Remove implicits
- Matchers
- Boolean results
- Standard results
- Combinators
- Match results
- Out of the box
- Optional
- Custom
- With sequences
- ScalaCheck
- Arbitrary instances
- With Generators
- Test properties
- Mock expectations
- Creation and settings
- Stubbing
- Mocking and Stubbing
- At the same time
- With matchers
- Callbacks

the implicits scopes

in the paragraphs below, this is a compromise in the sense that there is a bit more burden on the has to write more code to get his specification into shape.

/ immutable

erything

les were the subject of enough grief, I decided it was high time to do without them. This decision has a he way a user writes a specification. A specification can not anymore be a set of unrelated "blocks" ck is added to the parent specification through a side effect:

```
le is ok" in { 1 must_== 1 }        // those examples are added to the specific
 example is ok" in { 2 must_== 2 }  // variable
```

s" have to form a sequence:

```
le is ok"                 ! e1^               // notice the ^ operator here
 example is ok"           ! e2
{ 1 must_== 1 }
{ 2 must_== 2 }
```

drawback of not having side-effects. The presence of ^ everywhere produces unwanted syntactic ecification. One way of minimizing that noise is to make good use of an editor with column editing and nbols on the print margin of the screen. The specification can then be read as having 2 columns, one he with the implementation and the formatting directives.

ciple applies to the Examples bodies and has a major consequence: you have to explicitly chain

```
le on strings" ! e1                // will never fail!
{
 must have size(10000)             // because this expectation will not be retu
 must startWith("hell")
```

```
rrect way of writing the example is
le on strings" ! e1               // will fail
"hello" must have size(10000) and
           startWith("hell")
```

en as a limitation as well but also as an opportunity for writing better specifications. It's been indeed everal places that there should be only one expectation per example, now the design of ***specs2*** ages it!

tive consequence of that design decision is that debugging the library should be almost brainless. gramming is like having a pipe-line. If you don't like the output, you just cut the pipeline in smaller e the ins and outs of each and decide where things went wrong.

l, based on early feedback, a *mutable* version of the Specification trait was later introduced in order to DSL for *unit* specifications, where the code is interleaved with the descriptions (see below). The ited to **1** variable and to the construction phase of the specification.

have to be supplied

iguration" of a Specification in ***specs*** is realized with side-effects too. If you want to declare that the specification will share variables you can add `shareVariables()` at the top of the specification.

sible anymore in specs2, so you have to explicitly pass arguments at the top of your specification and n the rest:

- Parameters for the answers function
- Verification
- Order of calls
- Ignoring Stubs
- Spies
- Functions/PartialFunctions
- Auto-boxing
- Dynamic DataTables
- Forms
- Forms
- Fields
- Effects
- Properties
- Styles
- Simple form
- Adding several rows at once
- Nesting a Form into another Form
- Nesting a Form into an Effect or a Prop
- Using tabs
- Aggregating forms
- Lazy cells
- Xml cells
- 1-n relationships
- Subset
- Subsequence
- Set
- Sequence
- Decision tables
- Outside specs2

```
fication { def is =  args(color=false)   ^ // will not output colors
st of the specs"                         ^ end
```

y is a breeze

he expected advantages of using functional programming techniques and thanks to Scalaz
the concurrent execution of examples is just one line of code!

```
nts.map(f => promise(executeFragment(arguments <| fs.arguments)(f))).sequence.g
```

ructure

omes from the desire to unify the traditional ***specs*** approach of using blocks with `should` and `in`
a more *Literate* approach of having just free text.

there is no fundamental difference between an "Acceptance Testing" specification and a "Unit Test"
his is just a matter of the scale at which you're looking at things.

nd that having restrictions on the words I was supposed to use for my specification text didn't help me
appropriate descriptions of the system behavior or features.

of this principle is that a specification is composed of "Fragments" which can be some "Text" or some
ply appended together. You can use whatever words you want to describe the examples `should`, `can`,
r.

d structures serve 2 important purposes in ***specs***! They are used to control the scopes of variables
able to examples and to compute the indentation when displaying the results.

e done in ***specs2***?

pper context for an example, with "fresh" variables, which can be possibly inherited from a "parent"
ot require any support from the library (difficult to get bugs with that, right :-) ?).

case class instances for each Example. Here is a demonstration:

```
 user logs in"                        ^
st history must be shown"             ! history().isShown^
selects tickets"                      ^
list must be displayed"               ! tickets().list^
total amount must be displayed"       ! tickets().total^
e buys tickets                        ^
s favorite payment type is shown"     ! buy().favorite

in {
gedIn = false
in = loggedIn.true
out = loggedIn.false

s history() extends Login {

hown = loggedIn must beTrue

s tickets() extends Login {

t = pending
al = pending

s buy() extends Login {
kets = new tickets()
```

```
orite ^ pending
```

tion above, each example is using its own instance of a case class, having its own local variables r be overwritten by another example. Parent context is inherited by means of delegation. For example, ntext, there is an available `tickets` instance placing the system in the desired context.

r win here because the library doesn't have to propose new concepts, a new API to offer context unctionalities to: create contexts, share them, reuse them,...

e "simple structure" above, there is no need for adding curly braces `{...}` to separate the specification makes the specification text remarkably close to what's going to be displayed when reported.

ntation is a feature but it doesn't have to be. For example you could just write the specification above

```
 user logs in"                    ^
st history must be shown"         ! history().isShown^
selects tickets"                  ^
list must be displayed"           ! tickets().list^
total amount must be displayed"   ! tickets().total^
e buys tickets"                   ^
s favorite payment type is shown" ! buy().favorite
```

ve ***specs2*** compute something reasonable for the indentation along the following rules:

example follows a text, it is indented
ssive examples will be at the same indentation level
text follows an example, this means that you want to describe a "subcontext", so the next examples will nted with one more level

most likely to bring appropriate results but there are additional formatting elements which can be r to adjust the indentation or just skip lines: `p, br, t, bt, end, endbr, endp`.

jor operators used by ***specs2*** when building a Specification: `^` and `!`.

k" specification fragments together and `!` is used to declare the body of an example. The choice of s is mostly the result of the precedence rules in Scala. `+` binds more strongly than `!`, and `!` more . This means that you don't need to add brackets to:

gs with +: `"this is"+"my string" ^ "ok?"`
an example: `"this is some text" ^ "and this is an example description" !`
s

icit structure but just a "flow" of Fragments allows to insert other types of Fragments in a ***specs2***

n - expectation" format for specifying software is sometimes too verbose and tables are a much more f packing up descriptions and expectations. This idea is not new and a tool like Fitnesse has been y of writing specifications for years now.

his idea further with 3 features:

es (called "Forms") are statically compiled so adding a new column in a decision table will not fail at (and IDEs provide refactoring tools for better productivity)

- Via IDEA
- Via JUnit
- Arguments
- Via Eclipse
- Via Maven
- With your own Notifier
  - NotifierRunner in sbt
  - Exporter in sbt
- Philosophy
  - The origins
  - The score
  - Conciseness
  - Readability
  - Extensibility
  - Configuration
  - Clear implementation
  - User support
  - A new hope
  - Functional / immutable
  - Chaining everything
  - Arguments have to be supplied
  - Concurrency is a breeze
  - A simple structure
  - Contexts
  - Indentation
  - Operators
  - Forms
  - But if you STILL want mutable specifications

ns are not limited to simple n x m grids and can be nested inside each other. This helps a lot in
g domain objects where you have aggregates and lists of items

ns presentation and implementation can be encapsulated in the same class to be reused as a coherent
other specifications

back (for now) of this approach is that it is not possible to see, in real-time, a modification done on a
e seen in a browser with Fitnesse. There needs to be a compilation step, which in Scala, is not

STILL want mutable specifications

ast 2 very good reason for that.

t a smooth migration path from ***specs*** to ***specs2*** because rewriting specifications from scratch, with a
tax, does not bring a lot of value to your project

fragments hides the implementation of the examples and you have to navigate too much between the
the example code to understand what's going on. This is especially true when writing *unit*
ations where it's convenient to interleave short descriptions with blocks of code

ot a black-or-white language and mutation is definitely part of the toolbox. In the case of a specification
the advantages: less syntax, and the drawbacks: uncontrolled side-effects.

2 it is possible to create specifications which look almost like the ones which can be created with
it less functionalities:

```
org.specs2.mutable._   // similar to the mutable package for Scala collections

yMutableSpecification extends Specification {
 specification" should {
ild examples with side-effects" in { success }
en use side-effects to avoid chaining expectations" in {
1 must_== 2
// the rest won't be executed
success
```

things to know are:

cts are only used to build the specification fragments, by mutating a variable
also used to short-circuit the execution of an example as soon as there is a failure (by throwing an
n)
uild fragments in the body of examples or execute the same specification concurrently, the sky should
n
" management is to be done with case classes or traits (see `org.specs2.examples.MutableSpec`)

ies control

mpediment to software evolution is circular dependencies between packages in a project. The new
makes sure that a layered architecture is maintained, from low-level packages to high-level ones:

ation mutable
rm

ml html time json

- Design
  - Presentation
  - Structure
  - Creation
    - Creating Fragments
    - Mutable Specification
  - Execution
  - Reporting
  - Dependencies

n control id text main data control Implicit definitions control

a specification is no longer executable on its own, contrary to the ***specs*** design. It always need a

is dependency specification is not yet enforced automatically in ***specs2*** test suite, but this kind of e implemented in the future.

initions control

tension to be solved here. On one hand, I want to encourage conciseness so that one should not have ny traits on top of the Specification declaration to get the desired features. On the other hand, the more the more implicits you bring in.

mise is the following:

- The `BaseSpecification` trait only allows to build Text fragments and Examples, without even any Matchers
- On top of it, the `Specification` trait stacks lots of convenient functionalities to
  . use a concise notation for arguments
  . use matchers (with both `must` and `should`)
  . use predefined fragments and results (like `p`, `br`, `success`, `pending`,...)
  . and more
- Specific traits are available to selectively deactivate features. For instance `NoAutoExamples` deactivates the creation of examples from simple expectations.

This way, if there is any conflict when inheriting from the `Specification` trait, it should be possible to either:

- downgrade to the `BaseSpecification` and add the non-conflicting traits
- mix-in specific traits to remove the problematic implicit definitions

---

Total for specification Philosophy	
Finished in	753 ms
Results	5 examples, 0 failure, 0 error

- User Guide
  - Quick Start
  - Structure
    - Unit
    - Acceptance
    - Execution
    - And much more!
    - Presentation
    - Declare examples
    - Styles
      - Acceptance specification
      - Unit specification
    - Results
      - Standard
      - Matchers
      - Expectations
      - Functional
      - Thrown
      - All
      - Short-circuit
    - Pending until fixed
    - AutoExamples
    - G/W/T
      - Sequencing
      - Extract methods
      - User regexps
      - Factory methods
      - G/W/T sequences
      - Multiple steps
      - ScalaCheck
      - Single step
      - Conversions
      - Unit specification
    - DataTables
    - Links
      - Inline
      - Html link
      - Html Link
    - Reference
    - Markdown
    - url
    - Contexts

Start

jor styles of specifications with ***specs2***:

cifications where the specification text is interleaved with the specification code. It is generally used to a single class

nce specifications where all the specification text stands as one and the implementation code is re.

erally used for acceptance or integration scenarios

ce

ons extend the `org.specs2.mutable.Specification` trait and are using the `should/in` format:

```
g.specs2.mutable._

loWorldSpec extends Specification {

ello world'string" should {
ain 11 characters" in {
llo world" must have size(11)

t with 'Hello'" in {
llo world" must startWith("Hello")

with 'world'" in {
llo world" must endWith("world")
```

ecifications extend the `org.specs2.Specification` trait and must define a method called `is`:

```
g.specs2._

loWorldSpec extends Specification { def is =

s a specification to check the 'Hello world' string"                ^
                                                                    p^
ello world' string should"                                          ^
ain 11 characters"                                                  ! e1^
t with 'Hello'"                                                     ! e2^
with 'world'"                                                       ! e3^
                                                                    end

= "Hello world" must have size(11)
= "Hello world" must startWith("Hello")
= "Hello world" must endWith("world")
```

l lists *specification fragments* which can be:

xt, to describe the system you're specifying
nple: a description and some executable code returning a result
ng fragments: p adds a blank line and starts a new block of examples
separated by the ^ character in order to build a list of them.

ow to execute your specification, you use a *runner* which will display the results:

```
cp ... specs2.run HelloWorldSpec

dSpec

 specification to check the 'Hello world' string

o world' string should
 11 characters
ith 'Hello'
h 'world'

 specification HelloWorldSpec
in 0 second, 58 ms
s, 0 failure, 0 error
```

more!

e the rest of this User Guide to learn how to:

- Layout
- Isolation
- Scope
- Isolated variables
- Case classes
- Contexts inheritance
- Before/After
- In a unnamed specification
- In an acceptance specification
- Around
- Outside
- AroundOutside
- BeforeExample
- Implicit context
- Composition
- Combinations
- Composition
- Steps/Actions
- Steps
- Actions
- Template
- For fragments
- Execution
- Tables
- Formatting
- Fragments
- Separating groups of examples
- Reset the levels
- Changing the indentation level
- Combinations
- Turning-off the automatic layout
- Unit specification
- Unit specifications
- How to?
- Declare arguments
- Pass arguments

many ***specs2*** matchers to specify precise expectations
ontexts to setup/teardown data for your examples
/ link specifications and reuse examples
ckito or ScalaCheck
maven/junit to execute a specification
our specification as an html document (like this one!)

fication QuickStart
6 ms
examples, 0 failure, 0 error

- User Guide
  - Quick Start
    - Unit
    - Acceptance
    - Execution
    - And much more!
  - Structure
    - Presentation
    - Declare examples
    - Styles
      - Acceptance specification
      - Unit specification
    - Results
      - Standard Matchers
      - Expectations
        - Functional
        - Thrown
        - All
          - Short-circuit
      - Pending until fixed
      - Auto-Examples
      - G/W/T
      - Sequencing
      - Extract methods
      - User regexps
      - Factory methods
      - G/W/T sequences
      - Multiple steps
      - ScalaCheck
      - Single step
      - Conversions
      - Unit specification
      - DataTables
    - Links
      - Inclusion
        - Inline
        - Html link
          - Html Link
      - Reference
      - Markdown
      - url
    - Contexts

S

ion

y ways to execute ***specs2*** specifications:

ommand line, with a console output, and the `specs2.run` runner
ommand line, with a html output, and the `specs2.html` runner
ommand line, with a console or a html output, and the `specs2.files` runner

ur own reporting tool implementing the `Notifier` interface (simple) or the `Exporter` interface (with a ss to the executed specification)

ncies

able for Scala 2.9.0 onwards and uses the following libraries, as specified using the sbt dsl:

versions of specs2 available for Scala 2.8.1 but they miss some "context" functionalities.

	Comment
`" %% alaz-core" % "6.0.1"`	mandatory. This jar bundles the scalaz classes but renamed as `org.specs2.internal.scalaz._`.
`tools.testing" %% " % "1.9"`	if using ScalaCheck
`o" % "mockito-all" %`	if using Mockito. Note: specs2.jar must be placed before mockito.jar on the classpath
`est" % "hamcrest-all" %`	if using Hamcrest matchers with Mockito
`junit" % "4.7"`	if using JUnit
`tools.testing" % face" % "0.5"`	provided by sbt when using it
`n" % "pegdown" %`	if using the html runner
`" % "classycle" %`	if using the `org.specs2.specification.Analysis` trait
`lang" % iler" % "2.9.1"`	if using the `org.specs2.specification.Analysis` trait with the `CompilerDependencyFinder` trait

s

y arguments which will control the execution and reporting. They can be passed on the command line, ide the specification, using the `args(name=value)` syntax:

```
pec extends Specification { def is = args(xonly=true)    ^
 spec title"                                             ^
```

```
ill not be indented"                                    ^
ant expectation"                                        ! success
```

- Isolation
- Scope
- Isolated variables
- Case classes
- Contexts inheritance
- Before/After
- In a mutable specification
- In an acceptance specification
- Around
- Outside
- AroundOutside
- BeforeExample
- Implicit context
- Composition
- Combinations
- Composition
- Steps/Actions
- Steps
- Actions
- Template
- For fragments
- Execution
- Layout
- Rules
- Formatting fragments
- Separating groups of examples
- Reset the levels
- Changing the indentation level
- Combinations
- Turning-off the automatic layout
- Unit specification
- Unit specifications
- How to?
- Declare arguments
- Pass arguments

pecification, the available arguments are the following:

	Default value	Description
	.*	regular expression specifying the examples to execute. Use `ex .*brilliant.*` on the command line
	""	execute only the fragments tagged with any of the comma-separated list of tags: "t1,t2,..."
	""	do not execute the fragments tagged with any of the comma-separated list of tags: "t1,t2,..."
	false	select only previously failed/error examples
	""	select only some previously executed examples based on their status
	".*Spec"	regular expression to use when executing specifications with the FilesRunner
	false	only report the text of the specification without executing anything
	false	skip all the examples
	false	skip all examples after the first failure or error
	false	skip all examples after the first skipped result
	false	don't execute examples concurrently
	false	execute each example in its own specification to get "fresh" local variables
	`Runtime.getRuntime.availableProcessors`	number of threads to use for concurrent execution
	false	never store statistics
	false	remove previously stored statistics
	false	only report failures and errors
	""	only report some examples based on their status
	true	use colors in the output (`nocolor` can also be used on the command line)
	false	don't indent automatically text and examples
	true	interpret text as Markdown in the html reporter
	false	report the stacktrace for failures
	`org.specs2.text.SmartColors`	define alternative colors (replace failureColor from being yellow to magenta for example)
	false	show individual execution times
own	false	print more information when Markdown formatting fails
	`SmartDiffs`	use a specific algorithm to display differences

	true	true takes an AutoExample description from the file, false from the expectation ok message
	`DefaultStackTraceFilter`	use a StackTraceFilter instance for filtering the reported stacktrace elements
	false	if true, will parse the html files and check that local or http hrefs can be accessed
	false	if true, will not create a table of contents on the generated html page
	String	name of a class extending the `org.specs2.reporter.Notifier` trait
	String	name of a class extending the `org.specs2.reporter.Exporter` trait

frequently used arguments

uments above can be set in a specification with `args(name=value)`. However Scala would not allow od to accept all the possible
parameters (because a method can only have up to 22 parameters). This is why the least frequently s (not in italics) can be set with an object called `args`, having separate methods for setting all the "category". For example:

```
ct(specName = ".*Test", include="slow")
ute(threadsNb = 2)
rt(showTimes = true, xonly = true)
```

e available shortcuts for some arguments

	Equivalent	Description
gs: String)	`args(include=tags)`	
gs: String)	`args(exclude=tags)`	
les: String)	`args(ex=examples)`	
	`args(wasIssue=true)`	
String)	`args(was=status)`	
	`args(plan=true)`	
	`args(skipAll=true)`	
	`args(stopOnFail=true)`	
	`args(stopOnSkip=true)`	
	`args(sequential=true)`	
	`args(isolated=true)`	
	`args(xonly=true)`	
tatus: String)	`args(showOnly=status)`	
	`args(noindent=true)`	
	`args(noindent=true, sequential=true)`	for specifications where text must not be indented and examples be executed in order
	`args(plan=true, noindent=true)`	for specifications with no examples at all and free display of text
pectations	`args.report(fromSource=false)`	create the example description for the ok

- Matchers
  - Boolean results
  - Standard results
  - Combinators
  - Match results
  - Out of the box
  - Optional
  - Custom
  - With sequences
- ScalaCheck
  - Arbitrary instances
  - With Generators
  - Test properties
- Mock expectations
  - Creation and settings
  - Stubbing
    - Mocking and Stubbing at the same time
  - With matchers
  - Callbacks
- Add a title
- Use descriptions
- Enhance failures
- Share examples
- Create an index
- Tag examples
- In a unit specification
- Skip examples
- Debug statements
- Remove implicits

	message of the expectation instead of the source file
`args.report(traceFilter=NoStackTraceFilter)`	the stacktraces are not filtered
`args.report(diffs=SmartDiffs(show, separators, triggerSize, shortenSize, diffRatio, full)`	to display the differences when doing equality comparison

- Parameters for the answers function
- Verification Order of calls
- Ignoring stubs
- Spies
- Functions/PartialFunctions
- Auto-boxing
- Byname
- DataTables
- Forms
- Fields
- Effects
- Properties
- Styles
- Simple form
- Adding several rows at once
- Nesting a Form into another Form
- Nesting a Form into an Effect or a Prop
- Using tabs
- Aggregating forms
- Lazy cells
- Xml cells
- 1-n relationships
- Subset
- Subsequence
- Set
- Sequence
- Decision tables
- Outside specs2

ctory

ated during the execution of a specification will be created in the `target/specs-report` directory.
e that by setting the
`tDir` system property.

vious results

cation has been executed its statistics and failed examples will be stored by default in a specific `stats`
ed in the output directory. This data can be used on subsequent runs to:

rends in statistics
e the statuses of the links of an index page
nly previously failed examples for execution

this functionality (for performance reasons for example) with the `args.store(never=true)`
nt (or `neverstore` on the command line)
e previous statistics with the `args.store(reset=true)` argument (or `resetstore` on the
nd line)

irectory can also be redefined independently of the output directory with the `specs2.statsDir`
e.

s

`howOnly` arguments expect a String made of "status flags". For example, `xonly` is equivalent to
!"). Here is the list of all the flags which you can use to control the selection of fragments before
eir display:

tion
ful example
xample
ample
example
example
s

guments the values you can specify are:

ll not show anything (default is true)
`tors` allows to change the separators used to show the differences (default is "[]")
`rSize` controls the size above which the differences must be shown (default is 20)
`nSize` controls the number of characters to display around each difference (default is 5)
`tio` percentage of differences above which the differences must not be shown (default is 30)

splays the full original expected and actual strings

pecify your own enhanced algorithm for displaying difference by providing an instance of the
main.Diffs trait:

```
iffs {
return true if the differences must be shown */
how: Boolean
return true if the differences must be shown for 2 different strings */
how(expected: String, actual: String): Boolean
return the diffs */
howDiffs(expected: String, actual: String): (String, String)
return true if the full strings must also be shown */
howFull: Boolean
return the separators to use */
eparators: String
```

Filter

ter argument takes an instance of the `org.specs2.control.StackTraceFilter` trait to define
s should be filtered in a report. By default the `DefaultStackTraceFilter` filter will exclude lines
llowing packages:

```
ecs2
\\\\.
\\\., java\\\.
\., com.intellij, org.eclipse.jdt, org.junit
```

at you want, you can either:

`ludeTrace(patterns: String*)` to create a new `StackTraceFilter` which will include only
es matching
tterns

`ludeTrace(patterns: String*)` to create a new `StackTraceFilter` which will exclude only
es matching
tterns

`ludeAlsoTrace(patterns: String*)` to add new include patterns to the
`tStackTraceFilter`

`ludeAlsoTrace(patterns: String*)` to add new exclude patterns to the
`tStackTraceFilter`

`org.specs2.control.IncludeExcludeStackTraceFilter` class to define both include and
patterns

our own logic by extending the `org.specs2.control.StackTraceFilter`

l line

nd line you can pass the following arguments:

	Value format	Comments
	regexp	
	csv	
	csv	
	boolean	
	String	see: Status flags
	regexp	

- Via IDEA
- Via JUnit
  - Arguments
- Via Eclipse
- Via Maven
- With your own
  - Notifier
    - NotifierRunner
    - In sbt
  - Exporter
    - In sbt
- Philosophy
  - The origins
  - The score
    - Conciseness
    - Readability
    - Extensibility
    - Configuration
    - Clear implementation
    - User support
  - A new hope
    - Functional / immutable
      - Chaining everything
      - Arguments have to be supplied
      - Concurrency is a breeze
      - A simple structure
      - Contexts
      - Indentation
      - Operators
      - Forms
    - But if you STILL want mutable specifications

	boolean	
	boolean	
	boolean	
	boolean	
	int	
	boolean	
	boolean	
	boolean	
	String	see: Status flags
	boolean	
	boolean	
	map	e.g. text:be, failure:m (see the Colors section)
	boolean	
	boolean	
	boolean	
	boolean	
	boolean	
	boolean	
	boolean	
	regexp-csv/regexp-csv	comma-separated include patterns separated by / with exclude patterns
	String	name of a class extending the `org.specs2.reporter.Notifier` trait
	String	name of a class extending the `org.specs2.reporter.Exporter` trait

ava regular expression, csv a list of comma-separated values, map is a list of csv pairs key:value]

rgument accepting values (like `include`) can be passed with a dash when those values contain

```
TaggedSpec --include issue 123 -- other arguments
```

see above, `--` can be used to signal the end of some argument values

roperties

ny argument to ***specs2*** from system properties. This is particularly useful for passing arguments to

olean argument, you need to pass `-Dspecs2.name` or `-Dname`
ng argument, you need to pass `-Dspecs2.name=value` or `-Dname=value`

t `-Dname=value` can be convenient, `-Dspecs2.name=value` is recommended to avoid conflicts
ies.

ll

utput

ecification com.company.SpecName in the console is very easy:

```
specs2.run com.company.SpecName [argument1 argument2 ...]
```

ut

l pages to be produced for your specification you'll need to execute:

```
specs2.html com.company.SpecName [argument1 argument2 ...]
```

JUnit XML output

Many Continuous Integration systems rely on JUnit XML reports to display build and test results. It is possible to produce
those result by using the `specs2.junitxml` object:

```
scala -cp ... specs2.junitxml com.company.SpecName [argument1 argument2 ...]
```

Files Runner

The `specs2.files` object will, by default, select and execute Specifications found in the test source directory:

- the source directory is defined as `src/test/scala` but can be changed by adjusting the system property `specs2.srcTestDir`
- the specifications files are selected as classes or object which names match `.*Spec`. This value can be changed by
  passing a different `specName` value as a command-line argument
- `console` or `html` has to be passed on the command-line to specify which kind of output you want

You can also extend the `org.specs2.runner.FilesRunner` trait and override its behavior to implement something more appropriate
to your environment if necessary.

In the console

The `specs2.run` object has an `apply` method to execute specifications from the Scala console:

```
scala> specs2.run(spec1, spec2)

scala> import specs2._  // same thing, importing the run object
scala> run(spec1, spec2)
```

If you want to pass specific arguments you can import the `specs2.arguments` object member functions:

```
scala> import specs2.arguments._

scala> specs2.run(spec1)(nocolor)
```

Or you can set implicit arguments which will be used for any specification execution:

```
scala> import specs2.arguments._
scala> implicit val myargs = nocolor

scala> specs2.run(spec1)
```

Via SBT

with sbt 0.7.x

In order to use ***specs2*** with sbt 0.7.x you need first to add the following lines to your sbt project:

```
def specs2Framework = new TestFramework("org.specs2.runner.SpecsFramework")
override def testFrameworks = super.testFrameworks ++ Seq(specs2Framework)
```

Then, depending on the naming of your specification, you have to specify which classes you want to include for reporting:

```
override def includeTest(s: String) = { s.endsWith("Spec") || s.contains("UserGuide") }
```

with sbt > 0.9.x

In this case you don't need to do much because ***specs2*** will be recognized out-of-the-box. However, if you want to filter some specifications you need to add this to your `build.sbt` file (see here for more information):

```
// keep only specifications ending with Spec or Unit
testOptions := Seq(Tests.Filter(s => Seq("Spec", "Unit").exists(s.endsWith(_))))
```

If you don't want the specifications to be executed in parallel:

```
parallelExecution in Test := false
```

If you want to pass arguments available for all specifications:

```
testOptions in Test += Tests.Argument("nocolor", "neverstore")
```

If you want the examples results to be displayed as soon as they've been executed you need to add:

```
logBuffered := false
```

Test-only arguments

When you execute one test only, you can pass the arguments on the command line:

```
> test-only org.specs2.UserGuide -- xonly
```

Output formats

Html

The `html` argument is available with sbt to allow the creation of the html report from the command line.

```
> test-only org.specs2.UserGuide -- html
```

```
// in your build.sbt file
testOptions in Test += Tests.Argument("html")
```

Markdown

The `markup` argument can be used to create ".md" files (to use with websites like GitHub):

```
> test-only org.specs2.UserGuide -- markup
```

In this case the markup text in the Specifications is not interpreted.

JUnit

Similarly, JUnit xml output files can be created by passing the `junitxml` option:

```
> test-only org.specs2.examples.HelloWorldUnitSpec -- junitxml

// in your build.sbt file
testOptions in Test += Tests.Argument("junitxml")
```

Console

If you want to get a console output as well, don't forget to add the `console` argument:

```
> test-only org.specs2.UserGuide -- html console

// in your build.sbt file
testOptions in Test += Tests.Argument("html", "console")
```

Files runner

Any `FilesRunner` object can also be invoked by sbt, but you need to specify `console` or `html` (or both) on the command line:

```
> test-only allSpecs -- console
```

Colors

By default, the reporting will output colors. If you're running on windows you might either:

- use the following tip to install colors in the DOS console
- or pass `nocolor` as a command line argument

Then, there are different ways to set-up the colors you want to use for the output

From system properties

The so-called "SmartColors" argument will check if there are colors defined as specs2 properties. If so, the colors used
to output text in the Console will be extracted from those properties:

e.g. `-Dspecs2.color.failure=m` will use magenta for failures.

The property names and default values are:

Property	Default value
`color.text`	white
`color.success`	green
`color.failure`	yellow
`color.error`	red
`color.pending`	blue

`color.skipped`	cyan
`color.stats`	blue

The default values above are provided for a black background. If you have a white background you can use the `specs2.whitebg` property and then the default values will be:

Property	Default value
`color.text`	black
`color.success`	green
`color.failure`	magenta
`color.error`	red
`color.pending`	blue
`color.skipped`	cyan
`color.stats`	blue

All the available colors are listed here, with their corresponding abbreviation which you can use to refer to them as well:

Color	Abbreviation
white	w
green	g
yellow	y
red	r
blue	be
cyan	c
black	bk
magenta	m

From command-line arguments

It is also possible to set colors by passing the `colors` argument. This argument must be a list of `key:value` pairs (comma-separated) where keys are taken from the property names above without the `color.` prefix and values from the abbreviated color names.

For example you can pass on the command line:

```
colors text:blue,failure:magenta
```

to have the text colored in blue and the failures in Magenta.

If the `colors` option contains `whitebg` then the default colors are considered to be `InvertedColors`

Through the API

Finally you can change the color scheme that's being used on the console by implementing your own `org.specs2.text.Colors` trait or override values in the existing `ConsoleColors` class. For example if you want to output magenta everywhere yellow is used you can write:

```
  object MyColors = new org.specs2.text.ConsoleColors { override val failureColor = magent

  class MyColoredSpecification extends Specification { def is = colors(MyColors) ^
     // the failure message will be magenta
     "this is a failing example" ! failure
  }
```

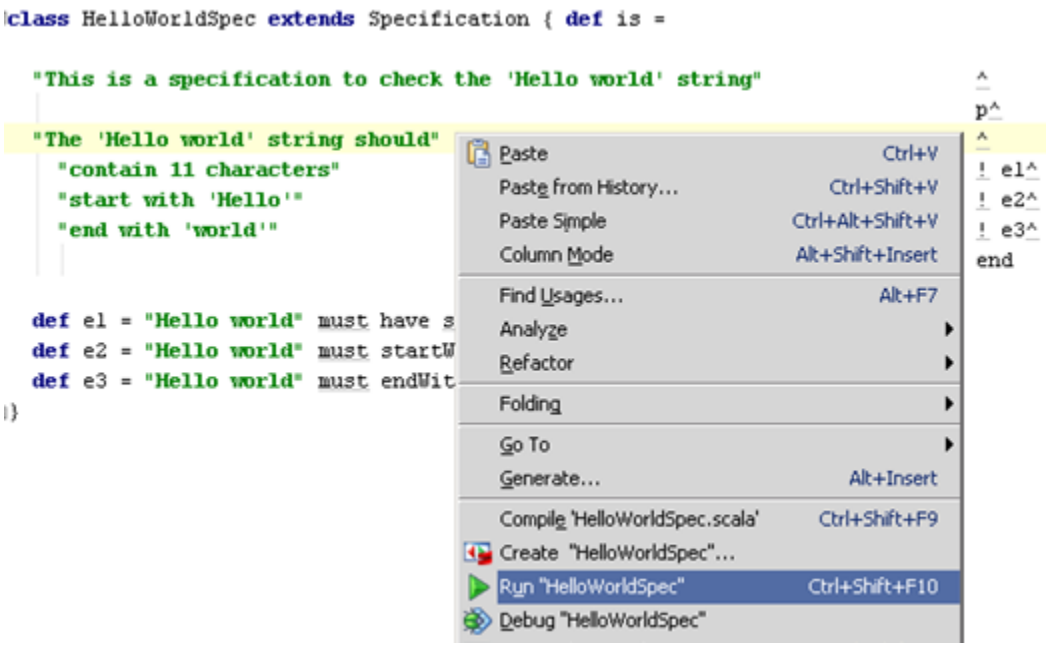
Note also that the the color support for sbt on Windows is a bit tricky. You need to follow the instructions here then add to your script launching sbt:

```
-Djline.terminal=jline.UnsupportedTerminal
```

Via IDEA

IntelliJ offers a nice integration with ***specs2***. You can:

- Execute a specification by selecting its name and pressing CTRL+SHIFT+F10
- Execute a single example by selecting its description and pressing CTRL+SHIFT+F10



But also:

- Provide command-line arguments in the "Test options"
- "Jump to Test" and "Jump to Source"

Via JUnit

It is possible to have ***specs2*** specifications executed as JUnit tests. This enables the integration of ***specs2*** with Maven and the JUnit runners of your IDE of choice.

There are 2 ways of enabling a Specification to be executed as a JUnit test: the verbose one and the simpler one. The simplest one is to extend `SpecificationWithJUnit`:

```
class MySpecification extends SpecificationWithJUnit {
  def is = // as usual....
}
```

You can use the second one if your IDE doesn't work with the first one:

```
import org.junit.runner._
import runner._

@RunWith(classOf[JUnitRunner])
class MySpecification extends Specification {
   def is = // as usual....
}
```

[*some tricks described on the specs website can still be useful there*]

Arguments

You can pass arguments to the `JUnitRunner` for generating the html files for the specifications or for displaying the console output. To do that, you can use the `-Dspecs2.commandline` property and pass it the `html` or `console` values.

Via Eclipse

There is unfortunately no specific Eclipse plugin at the moment and specifications have to be executed as [JUnit test cases](#Via JUnit).

Via Maven

You can either:

- execute specifications as [JUnit test cases](#Via JUnit).
- use the Maven specs2 plugin to generate JUnit XML reports and HTML reports

With your own

Notifier

The `org.specs2.reporter.Notifier` trait can be used to report execution events. It notifies of the following:

- specification start: the beginning of a specification, with its name
- specification end: the end of a specification, with its name
- context start: the beginning of a sub-level when the specification is seen as a tree or Fragments
- context end: the end of a sub-level when the specification is seen as a tree or Fragments
- text: any Text fragment that needs to be displayed
- example start
- example result: success / failure / error / skipped / pending

All those notifications come with a location (to trace back to the originating fragment in the Specification) and a duration when relevant (i.e. for examples and actions).

NotifierRunner

The `NotifierRunner` class can be instantiated with a custom `Notifier` and used from the command line.

In sbt

You can also use a custom `Notifier` from inside sbt by passing the `notifier` argument with a `Notifier` implementation class name:

```
sbt>test-only *BinarySpec* -- notifier com.mycompany.reporting.FtpNotifier
```

Exporter

The `org.specs2.reporter.Exporter` trait can be used to collect `ExecutedFragments` and report them as desired. The only method to implement is:

```
def export(implicit args: Arguments): ExecutingSpecification => ExecutedSpecification
```

- `args` is an `Arguments` object created from command line options
- `ExecutingSpecification` is a list of fragments which might or might not have finished their execution
- `ExecutedSpecification` must be a list of executed fragments

Please see the API of each class to see how to use them.

In sbt

You can use a custom `Exporter` from inside sbt by passing the `exporter` argument with a `Exporter` implementation class name:

```
sbt>test-only *BinarySpec* -- exporter com.mycompany.reporting.FtpExporter
```

Total for specification Runners	
Finished in	762 ms
Results	1 example, 0 failure, 0 error

- User Guide
  - Quick Start
  - Structure
    - Unit
    - Acceptance
    - Execution
    - And much more!
    - Presentation
    - Display examples
    - Styles
    - Acceptance specification
    - Unit specification
    - Results
    - Standard
    - Matchers
    - Expectations
    - Functional
    - Thrown
    - All
    - Skipped
    - Pending until fixed
    - Auto-Examples
    - G/W/T
    - Sequencing
    - Extract methods
    - User regexps
    - Factory methods
    - G/W/T sequences
    - Multiple steps
    - ScalaCheck
    - Single step
    - Conversions
    - Unit specification
    - DataTables
    - Links
    - Inclusion
    - Inline
    - Html link
    - Html Link
    - Reference
    - Markdown
    - url
    - Contexts

ire

ion

u will learn how to:

examples and expectations
cifications together
ontexts and actions to execute before/after examples
he execution strategy
e specification text

xamples

t guide describes 2 styles of specifications, the *unit* style and the *acceptance* style. Both styles actually
ation as a list of *fragments*.

e specification

ce specification you build a list of *fragments* with the ^ operator:

```
my specification                        ^
ample 1"                                ! e1^
ample 2"                                ! e2

success
success
```

here is a list of 3 fragments, a Text fragment and 2 Example fragments. The examples are declared

`iption" ! body`. Their "bodies" are provided by 2 methods returning a `Result`, separated from the
xt.

ecific recommendation on how you should name those methods but you can either use short names or
k
ter readability:

```
my specification"                       ^
ample 1"                                ! `first example`^
ample 2"                                ! `second example`

t example` = success
nd example` = success
```

ush this idea further by writing:

```
my specification"                       ^
ample 1`                                ^
ample 2`
```

- Isolation
  - Scope
  - Isolated variables
  - Case classes
  - Contexts inheritance
  - Before/After
  - In a mutable specification
  - In an acceptance specification
  - Around
  - Outside
  - AroundOutside
  - BeforeExample
  - Implicit context
  - Composition
  - Combinations
  - CompositionAround
  - SeparateExamples
  - Steps
  - Actions
  - Template
  - For fragments
  - Execution
  - Layout
  - Rules
  - Formatting fragments
  - Separating groups of examples
  - Reset the levels
  - Changing the indentation level
  - Combinations
  - Turning-off the automatic layout
  - Unit specification
- Unit specifications
- How to?
  - Isolate arguments
  - Pass arguments

```
example 1` = success
example 2` = success
```

od refactoring capabilities is a must-have in that case,...)

cation

tion uses should/in blocks which build the Fragments by adding them to a mutable protected

```
lo world' string" should {
n 11 characters" in {
o world" must have size(11)

with 'Hello'" in {
o world" must startWith("Hello")

th 'world'" in {
o world" must endWith("world")
```

ation the following methods are used:

eate an Example containing a `Result`

to create a `Group` of Examples, with a the preceding Text fragment appended with `should`

equivalent to writing this in an `org.specs2.Specification`:

```
lo world" string should" ^
n 11 characters" ! {
o world" must have size(11)

with 'Hello'" ! {
o world" must startWith("Hello")

th 'world'" ! {
o world" must endWith("world")
```

ications section shows all the methods which can be used to build unit specifications fragments.

created by following a piece of text with `!` and providing anything convertible to an `execute.Result`:

ard result
er result
an value

`esult` values are provided by the `StandardResults` trait (mixed-in with `Specification`), and

provided by ***specs2***.

s: the example is ok
e: there is a non-met expectation
r: a unexpected exception occurred
d: the example is skipped possibly at runtime because some conditions are not met
g: usually means "not implemented yet"

results are also available to track the progress of features:

`Success` with the message "DONE"
`Pending` with the message "TODO"

dy of an example is made of *expectations* using matchers:

```
must_== 1
```

o the Matchers guide to learn all about matchers and how to create expectations.

ons

ecification trait in ***specs2*** is functional: the Result of an example is always given by the last
s body. This example will never fail because the first expectation is "lost":

```
le on strings" ! e1                // will never fail!
{
 must have size(10000)             // because this expression will not be retu
 must startWith("hell")
```

way of writing the example is:

```
le on strings" ! e1               // will fail
"hello" must have size(10000) and
            startWith("hell")
```

ctionality encourages a specification style where every expectation is carefully specified and is
d practice by some. However you might see it as an annoying restriction. You can avoid it by mixing-in
s2.matcher.`ThrownExpectations` trait. With that trait, any failing expectation will throw a
ption and the rest of the example will not be executed.

n additional method `failure(message)` to throw a `FailureException` at will.

`hrownExpectations` traits is mixed in the `mutable.Specification` trait used for *unit*
nd, if you wish, you revert back to *not* throwing exceptions on failed expectations by mixing-in the
`matcher.NoThrownExpectations` trait.

- Add a title
- Use descriptions
- Enhance failures
- Share examples
- Create an index
- Tag examples
  - In a unit specification
- Skip examples
- Debug statements
- Remove implicits

- Matchers
  - Boolean results
  - Standard results
  - Combinators
  - Match results
  - Out of the box
  - Optional
  - Custom
  - With sequences
  - ScalaCheck
    - Arbitrary instances
    - With Generators
    - Test properties
  - Mock expectations
  - Creation and settings
  - Stubbing
    - Mocking and Stubbing at the same time
  - With matchers
  - Callbacks

- Parameters for the answers function
- Verification
- Order of calls
- Ignoring stubs
- Spies
- Functions/PartialFunctions
- Auto-boxing
- Byname

- Data Tables
- Forms
- Forms
- Fields
- Effects
- Properties
- Styles
- Simple form
- Adding several rows at once
- Nesting a Form into another Form
- Nesting a Form into an Effect or a Prop
- Using tabs
- Aggregating forms
- Lazy cells
- Xml cells

- 1-n relationships
- Subset
- Subsequence
- Set
- Sequence

- Decision tables

- Outside specs2

`s2.specification.AllExpectations` trait goes further and gives you the possibility to have all
an Example to be reported without stopping at the first one. This enables a type of specification where it
efine lots of expectations inside the body of an example and get a maximum of information on what
passes:

```
g.specs2._
ecification._

ExpectationsSpec extends mutable.Specification with AllExpectations {
s example all the expectations are evaluated" >> {
 2  // this fails
 3  // this also fails
 1

is no Collision with this example" >> {
= 11 // this fails
= 12
= 31 // this also fails
```

ample above hints at a restriction for this kind of Specification. The failures are accumulated for each
tating a shared variable. "Mutable" means that the concurrent execution of examples will be an issue if
o avoid this, the `AllExpectations` trait overrides the Specification arguments so that the
ecomes isolated unless it is already `isolated` or `sequential`.

may want to stop the execution of an example if one expectation is not verified. This is possible with

```
example all the expectations are evaluated" >> {
            // this is ok
).orThrow   // this fails but is never executed
```

`rSkip` will skip the rest of the example in case of a failure.

ntil fixed

s may be temporarily failing but you may not want the entire test suite to fail just for those examples.
menting them out and then forgetting about those examples when the code is fixed, you can append
`lFixed` to the Example body:

```
mple fails for now" ! {
== 2
UntilFixed

th a more specific message
mple fails for now" ! {
== 2
UntilFixed("ISSUE-123")
```

bove will be reported as `Pending` until it succeeds. Then it is marked as a failure so that you can
move the `pendingUntilFixed` marker.

mples

ation is about showing the use of a DSL or of an API, you can elid a description for the Example. This used in ***specs2*** to specify matchers:

- Runners

```
ecks if an element is None"                    ^
t beNone }                                     ^
must not be none }                             ^
```

e text of the example will be extracted from the source file and the output will be:

```
cks if an element is None
ust beNone
) must not be none
```

can also be used in mutable specifications but the need to be declared by using the `eg` (*exempli gratia* eviation for "for example"):

```
Examples extends mutable.Specification {
ust beNone }.eg
) must not be none }.eg
```

remember about this feature:

ce file is expected to be found in the `src/test/scala` directory. This can be overriden by specifying cs2.srcTestDir system property

action of the source code is rudimentary and may fail on specifications which are built dynamically

lines of code can be extracted provided that the block ends with a `Result` and that there is a nt following the block to be extracted. The best way to ensure that is to always add an `end` fragment nd of the Specification

e to extract must be in the same directory as the package of the specification class it belongs to. If a ation is declared in `package com.mycompany.accounting` then its source file has to be in the company/accounting directory for Auto-Examples to be working

robustness, but different results, you can use the `descFromExpectations` argument (creates an romSource=false) argument) to take the "ok message" from the expectation as the example ion:

```
tputs: List(1, 2) must contain(1)
t(1, 2) must contain(1) }

tputs: 'List(1, 2)' contains '1'
romExpectations ^
t(1, 2) must contain(1) }
```

ated is the Given/When/Then style of writing specifications. This style is supported by interspersing , with Given/When/Then `RegexSteps` which extract meaningful values from the text. Here's an ication for a simple calculator:

```
when-then example for the addition"                ^
the following number: ${1}"                        ^ number1 ^
second number: ${2}"                               ^ number2 ^
```

```
should get: ${3}"                                    ^ result ^
                                                     end
```

```
mber1 extends Given[Int] {
ract(text: String): Int = extract1(text).toInt

s Addition(n1: Int, n2: Int) {
: Int = n1 + n2

mber2 extends When[Int, Addition] {
ract(number1: Int, text: String) = Addition(number1, extract1(text).toInt)

sult extends Then[Addition] {
ract(addition: Addition, text: String): Result = addition.add must_== extract1(
```

- Via IDEA
- Via JUnit
- Via Eclipse
- Via Maven
- With your own
  - Notifier
  - NotifierRunner
  - In sbt
  - Exporter
  - In sbt
- Philosophy
  - The origins
  - The vision
  - Conciseness
  - Readability
  - Extensibility
  - Configuration
  - Clear implementation
  - User support
  - A new hope
  - Functional / immutable
  - Chaining everything
  - Arguments have to be supplied
  - Concurrency is a breeze
  - A simple structure
  - Contexts
  - Indentation
  - Operators
  - Forms
  - But if you STILL want mutable specifications
- Arguments

xplanation of the object definitions that support the G/W/T style:

1 is a `Given` step. It is parametrized with the type `Int` meaning that its `extract` method is supposed
t an Int from the preceding text. It does so by using the `extract1` inherited method, which parses the
`${}` expressions and return a tuple (with 1 element here) containing all the values enclosed in `${}`.

2 is a When step. It is paramerized with an `Int`, the result from the previous extraction, and an
on which is the result of extracting the second number and putting the 2 together. In that case the
which must be defined is `extract(Int, String): Addition`.

e `result` object defines the outcome of the Addition. Its `extract` method takes an `Addition` and
ent text to return a `Result`

g

nce can contain more than just 3 steps. However the compiler will check that:

iven[T] extractor can start a sequence
iven[S], a `When[T, S]` or a `Then[T]` extractor can follow a `Given[T]` extractor
hen[T1, T2, S]` or a `Then[T1, T2]` can follow a sequence of `Given[T1], Given[T2]`
rs (up to 8 Given steps, after that types are paired)
hen[S, U]` extractor or a `Then[S]` can follow a `When[T, S]` extractor
hen[S]` can follow a `Then[S]` extractor

ncrete, here are a few valid sequences:

] / When[T, S] / Then[S]
] / Given[T2] / Given[T2] / When[T, T1, T2, R] / Then[R]
] / Given[T2] / Given[T3] / Given[T4] / Then[T, T1, T2, T3, T4]
] / Given[T2] / ... / Given[T8] / Then[T, T1, T2, T3, T4, T5, T6, (T7, T8)]
] / When[T, S] / Then[S] / Then[S]
] / Then[T] / Then[T]
] / When[T, S] / When[S, U] / Then[U]

hods

en, `Then` classes provide several convenience methods to extract strings from the preceding text: the
`xtract2,...`
xtracts the values delimited by `${}` for up to 10 values.

os

way of declaring Given/When/Then steps, the text is left completely void of markers to extract
ues. The user then
y a regular expression where groups are used to show where those values are:

```
mber1 extends Given[Int]("Given the following number: (.*)") {
ract(text: String): Int = extract1(text).toInt
```

- of using this way is that the text is left in it's pristine form, the drawback is that most of the text is places, adding more maintenance burden.

thods

e factory and implicit conversion methods to create Given/When/Then steps by passing functions and / essions:

- a function `String... => T` to a `Given[T]` step (*note the use of and after `readAs` and `groupAs`*)

```
s assumes that the Int to extract is delimited with ${}
mber1: Given[Int] = (s: String) => s.toInt
number1.extract("pay ${100} now") === 100

// this uses a regular expression with capturing groups matching the full text
val number1: Given[Int] = readAs(".*(\\d+).*") and { (s: String) => s.toInt }
number1.extract("pay 100 now") === 100

// this uses capturing groups directly
val number1: Given[Int] = groupAs("\\d+") and { (s: String) => s.toInt }
number1.extract("pay 100 now") === 100

// if the Given step is only side-effecting we can omit the `and` call
// this simplifies the use of Given steps in Unit Specifications
val number1: Given[Unit] = groupAs("\\d+") { (s: String) => value = s.toInt }
```

- convert a function `T => String... => S` to a `When[T, S]` step (*note the use of and after `readAs` and `groupAs`*)

```
// this assumes that the Int to extract is delimited with ${}
val number2: When[Int, (Int, Int)] = (n1: Int) => (s: String) => (n1, s.toInt)
number2.extract(100, "with a discount of ${10}%") === (100, 10)

// this uses a regular expression with capturing groups matching the full text
val number2: When[Int, (Int, Int)] = readAs(".*(\\d+).*") and { (n1: Int) => (s: Stri
number2.extract(100, "with a discount of 10%") === (100, 10)

// this uses capturing groups directly
val number2: When[Int, (Int, Int)] = groupAs("\\d+") and { (n1: Int) => (s: String) =
number2.extract(100, "with a discount of 10%") === (100, 10)
```

- convert a function `T => String... => Result` to a `Then[T]` step (*note the use of `then` after `readAs` and `groupAs`*)

```
// this assumes that the Int to extract is delimited with ${}
val number3: Then[(Int, Int)] = (n: (Int, Int)) => (s: String) => discount(n._1, n._2
number3.extract((100, 10), "the result is ${90}") must beSuccessful

// this uses a regular expression with capturing groups matching the full text
val number3: Then[(Int, Int)] = readAs(".*(\\d+).*") then { (n: (Int, Int)) => (s: St
number3.extract((100, 10), "the result is 90") must beSuccessful

// this uses capturing groups directly
val number3: Then[(Int, Int)] = groupAs("\\d+") then { (n: (Int, Int)) => (s: String)
number3.extract((100, 10), "the result is 90") must beSuccessful

// if the Then step is only side-effecting we can omit the `then` call
// this simplifies the use of Then steps in Unit Specifications
val number3: Then[Unit] = groupAs("\\d+") { (s: String) => value must_== s.toInt }
```

G/W/T sequences

Given the rule saying that only a `Then` block can follow another `Then` block you might think that it is not possible to start another G/W/T
sequence in the same specification! Fortunately it is possible by just terminating the first sequence with an `end` fragment:

```
"A given-when-then example for the addition"                ^
  "Given the following number: ${1}"                        ^ number1 ^
  "And a second number: ${2}"                               ^ number2 ^
  "Then I should get: ${3}"                                 ^ addition ^
                                                            end^
"A given-when-then example for the multiplication"          ^
  "Given the following number: ${1}"                        ^ number1 ^
  "And a second number: ${2}"                               ^ number2 ^
  "Then I should get: ${2}"                                 ^ multiplication ^
                                                            end
```

Multiple steps

If there are lots of consecutive `When` steps collecting the same kind of arguments, it will be easier to collect them in a `Seq[T]` rather than a `TupleN[T]`:

```
"A given-when-then example for the addition"                ^
  "Given the following number: ${1}"                        ^ number1 ^
  "And a second number: ${2}"                               ^ number2 ^
  "And a third number: ${3}"                                ^ number3

val number1: Given[Int]               = (_:String).toInt
val number2: When[Int, (Int, Int)]    = (n1: Int) => (s: String) => (n1, s.toInt)
val number3: When[Seq[Int], Seq[Int]] = (numbers: Seq[Int]) => (s: String) => numbers :+
```

ScalaCheck

Once you've created a given G/W/T sequence, you can be tempted to copy and paste it in order to check the same scenario with different values. The trouble with this is the duplication of text which leads to more maintenance down the road.

This can be avoided and even enhanced by using ScalaCheck to generate more values for the same scenario. For the calculator above you could write:

```
import org.scalacheck.Gen._
import specification.gen._

class GivenWhenThenScalacheckSpec extends Specification with ScalaCheck { def is =

  "A given-when-then example for a calculator"                                ^
    "Given a first number n1"                                                 ^ numbe
    "And a second number n2"                                                  ^ numbe
    "When I add them"                                                         ^ add ^
    "Then I should get n1 + n2"                                               ^ resul
                                                                              end


  object number1 extends Given[Int] {
    def extract(text: String) = choose(-10, 10)
  }
  object number2 extends When[Int, (Int, Int)] {
    def extract(number1: Int, text: String) = for { n2 <- choose(-10, 10) } yield (numbe
  }
  object add extends When[(Int, Int), Addition] {
    def extract(numbers: (Int, Int), text: String) = Addition(numbers._1, numbers._2)
  }
  object mult extends When[(Int, Int), Multiplication] {
```

```
    def extract(numbers: (Int, Int), text: String) = Multiplication(numbers._1, numbers.
  }
  object result extends Then[Addition] {
    def extract(text: String)(implicit op: Arbitrary[Addition]) = {
      check { (op: Addition) => op.calculate must_== op.n1 + op.n2 }
    }
  }
  case class Addition(n1: Int, n2: Int) extends Operation { def calculate: Int = n1 + n2
}
```

The main differences with a "normal" G/W/T sequence are:

- the import of step classes from `org.specs2.specification.gen` instead of `org.specs2.specification`
- the return values from the `extract` methods of the `Given` and `When` steps which must return ScalaCheck generators (cf `number1` and `number2`). For the `add` step there is an implicit conversion transforming any value of type `T` to a `Gen[T]`
- the use of the ScalaCheck trait to access the `check` function transforming a function to a `org.scalacheck.Prop` and then to a `Result`
- the `extract` method of the `Then` step takes an implicit `Arbitrary[T]` parameter which is used by the `check` method to create a ScalaCheck property

Single step

A `GivenThen` step can be used to extract values from a single piece of text and return a `Result`:

```
"given the name: ${eric}, then the age is ${18}" ! new GivenThen {
  def extract(text: String) = {
    val (name, age) = extract2(text)
    age.toInt must_== 18
  }
}
```

You can also use the `so` object. This object provides an `apply` method expecting a `PartialFunction` and does the value extraction:

```
import org.specs2.specification.so

"given the name: ${eric}, then the age is ${18}" ! so { case (name: String, age: String)
  age.toInt must_== 18
}
```

Conversions

Given / When / Then steps are invariant in their type parameters. This might be detrimental to reuse. For example, if you've defined a `Then[X]` step to check something about a value of type `X`, it would make sense to reuse the same step with a value of type `Y` when `Y <: X`. In order to do this you can use some implicit conversions which will translate steps between types when it makes sense:

```
val thenX = new Then[X] {
  def extract(x: X, s: String) = success // check something about x
}
// thenX can be reused as a Then[Y] step because Y <: X
val thenY: Then[Y] = thenX
```

Unit specification

Given / When / Step can also be used in a unit specification by using the << operator and local variables:

```
  "A given-when-then example for a calculator".txt.br

    "Given the following number: ${1}" << { s: String =>
      a = s.toInt
```

```
    }
    "And a second number: ${2}" << { s: String =>
      b = s.toInt
    }
    "When I use this operator: ${+}" << { s: String =>
      result = Operation(a, b, s).calculate
    }
    "Then I should get: ${3}" << { s: String =>
      result === s.toInt
    }
    "And it should be > ${0}" << { s: String =>
      result must be_>(s.toInt)
    }

  var a, b, result: Int = 0

  case class Operation(n1: Int, n2: Int, operator: String) {
    def calculate: Int = if (operator == "+") n1 + n2 else n1 * n2
  }
```

If you want to use your own regular expression parsing, the << operator also accepts `Given[Unit]` and `Then[Unit]` steps:

```
  "Given the following number: 1" << readAs(".*(\\\\d).*") { s: String =>
    a = s.toInt
  }
  "And a second number: 2" << groupAs("\\\\d") { s: Seq[String] =>
    b = s.head.toInt
  }
  "When I use this operator: +" << groupAs("[\\\\+\\\\-]") { s: String =>
    result = Operation(a, b, s).calculate
  }
  "Then I should get: 3" << groupAs("\\\\d") { s: String =>
    result === s.toInt
  }
  "And it should be > 0" << groupAs("\\\\d") { s: String =>
    result must be_>(s.toInt)
  }
```

Similarly, ScalaCheck generator and properties are supported:

```
  "Given a first number n1" << {
    n1 = choose(-10, 10)
  }
  "And a second number n2" << {
    n2 = choose(-10, 10)
  }
  "When I add them" << {
    operation = Arbitrary {
      for (a1 <- n1; a2 <- n2) yield Addition(a1, a2)
    }
  }
  "Then I should get n1 + n2" << check { (op: Addition) =>
    op.calculate must_== op.n1 + op.n2
  }

  var n1, n2: Gen[Int] = null
  implicit var operation: Arbitrary[Addition] = null
```

DataTables

DataTables are generally used to pack lots of expectations inside one example. A DataTable which is used as a `Result` in the body of an Example will only be displayed when failing. If, on the other hand you want to display the

table even when successful, to document your examples, you can omit the example description and inline the DataTable directly in the specification:

```
class DataTableSpec extends Specification with DataTables { def is =

  "adding integers should just work in scala"  ^ {
    "a"   | "b" | "c" |
     2    !  2  !  4  |
     1    !  1  !  2  |> {
     (a, b, c) =>  a + b must_== c
  }
}
```

This specification will be rendered as:

```
adding integers should just work in scala
+  a | b | c |
   2 | 2 | 4 |
   1 | 1 | 2 |
```

Links

There are 2 ways to "link" specifications:

- by including another specification, to create a parent-child relationship
- by creating a reference to another specification, to create a peer relationship

Inclusion

There is a simple mechanism for including a "children" specification in a given specification. You can simply add the child specification as if it was a simple fragment:

```
"This is an included specification"     ^
  childSpec
```

Otherwise, if you want to include several specifications at once you can use the `include` method:

```
"This is the included specifications"         ^
  include(childSpec1, childSpec2, childSpec3)
```

The effect of doing so is that all the fragments of the children specification will be inlined in the parent one. This is exactly what is done in this page of the user guide, but with a twist

```
include(xonly, new GivenWhenThenSpec)        ^
include(xonly, exampleTextIndentation)       ^
include(xonly, resetTextIndentation)         ^
```

In the code above there are specific arguments to the included specifications so that they are only displayed when there are failures.

Inline

When you include a specification in another one the console will display the beginning and end statistics of the included specification. If you just want to insert the "middle" fragments of the included specification you can use `inline`:

```
inline(otherSpecification)
```

Html link

In order to create a User Guide such as this one, you might want the included specification to be written to another html file. In this case, you need a "Link":

```
link(new QuickStart)
```

This declaration will include the child specification so it is executed when the parent specification is executed. However during the reporting, only a Html link will be created in the parent file, referencing a separate file for the children specification. On the other hand if you "hide" the specification, the link will not be printed out:

```
link((new QuickStart).hide)
```

Html Link

It is possible to customize the generated Html link with the following syntax:

```
"a " ~ ("quick start guide", new QuickStart)
```

The ~ operator is used to create a `HtmlLink` where:

- "a" is the beginning of the text
- "quick start guide" is the text that will be highlighted as a url link
- `new QuickStart` is the specification to include, the url being derived from the specification class name

Several variations are possible on this pattern, depending which part of the link you want to be highlighted:

```
"before text" ~ ("text to highlight", specification, "after text")
"before text" ~ ("text to highlight", specification, "after text", "tooltip")
"text to highlight" ~ specification
"text to highlight" ~ (specification, "after text")
"text to highlight" ~ (specification, "after text", "tooltip")
```

Reference

Sometimes you just want to reference another specification without triggering its execution. For example when creating an index page:

```
see(new MailSenderSpec)
```

This will generate a html link in the main specification based on the referenced specification name. If you want to customize that link you can use the following syntax:

```
"before text" ~/ ("text to highlight", specification, "after text")
"before text" ~/ ("text to highlight", specification, "after text", "tooltip")
"text to highlight" ~/ specification
"text to highlight" ~/ (specification, "after text")
"text to highlight" ~/ (specification, "after text", "tooltip")
```

Markdown url

If you just want to reference the url of the html page that's being generated for a given specification in a paragraph of text, you can use the `markdownUrl` method:

```
"For more information you can read "+DetailedSpec.markdownUrl
// or
"For more information you can read "+DetailedSpec.markdownUrl("the detailed specificatio
```

```
// or
"For more information you can read "+"the detailed specification".markdownUrl(DetailedSp
```

Contexts

In a specification some examples are very simple and just check that a function is behaving as expected. However other examples can be more complex and require a more elaborate set-up of data to:

- to create inter-related domain objects
- to put the environment (database, filesystem, external system) in the appropriate state

And there are usually 3 difficulties in doing that:

1. *Variables isolation*: making sure that each example can be executed with its own data without being impacted by the undesired side-effects of other examples
2. *Before/After code*: running code before or after every example without repeating that code in the body of each example
3. *Global setup/teardown code*: setting some state when this could take lots of resources, so you need to do it just once before anything runs

How does a library like JUnit solves this?

1. *Variables isolation*: for each test run a new class instance is created so that there are new "fresh" variables for the current test case
2. *Before/After code*: there are `@Before` and `@After` annotations to declare once the code that must be executed before or after each example
3. *Global setup/teardown code*: there are `@BeforeClass` and `@AfterClass` annotations dedicated to that kind of code

Now let's see how this can be achieved with ***specs2***.

Isolation

specs2 solves this issue in 2 ways:

- simply by relying on Scala features, by creating a new trait or a case class to open a new `Scope` with fresh variables
- by cloning the specification on each example execution when the `isolated` argument is provided

Scope

Let's see an example of using a `Scope` with a mutable specification:

```
import org.specs2.specification.Scope

class ContextSpec extends mutable.Specification {
  "this is the first example" in new trees {
    tree.removeNodes(2, 3) must have size(2)
  }
  "this is the first example" in new trees {
    tree.removeNodes(2, 3, 4) must have size(1)
  }
}

/** the `trees` context */
trait trees extends Scope {
  val tree = new Tree(1, 2, 3, 4)
}
```

Each example of that specification gets a new instance of the `trees` trait. So it will have a brand new `tree` variable and even if this data is mutated by an example, other examples will be isolated from these changes.

Now you might wonder why the `trees` trait is extending the `org.specs2.specification.Scope` trait? The reason is that the body of an Example only accepts objects which are convertible to a `Result`. By extending `Scope` we can take advantage of an implicit conversion provided by the `Specification` trait to convert our context object to a `Result`.

Scopes are a way to create a "fresh" object and associated variables for each example being executed. The advantages are that:

- those classes can be reused and extended
- the execution behavior only relies on language constructs

However, sometimes, we wish to go for a more concise way of getting fresh variables, without having to create a specific trait to encapsulate them. That's what the `isolated` argument is for.

Isolated variables

The `isolated` argument changes the execution method so that each example is executed in a brand new instance of the Specification:

```
class IsolatedSpec extends mutable.Specification {
  isolated

  "Each example should be executed in isolation" >> {

    val tree = new Tree(1, 2, 3, 4)
    "the first example modifies the tree" >> {
      tree.removeNodes(2, 3) must have size(2)
    }
    "the second example gets an unmodified version of the tree" >> {
      tree.removeNodes(2, 3, 4) must have size(1)
    }
  }
}
```

Since there is a new Specification for each example, then all the variables accessible to the example will be seen as new.

Note: this technique will not work if the Specification is defined with a constructor having parameters because it won't be possible to create a new instance.

Case classes

The same kind of variable isolation can be achieved in acceptance specifications by using case classes:

```
class ContextSpec extends Specification { def is =
  "this is the first example" ! trees().e1 ^
  "this is the first example" ! trees().e2
}

case class trees() {
  val tree = createATreeWith4Nodes

  def e1 = tree.removeNodes(2, 3) must have size(2)
  def e2 = tree.removeNodes(2, 3, 4) must have size(1)
}
```

In this case we don't need to extend the `Scope` trait because the examples `e1` and `e2` already return `Results`.

Contexts inheritance

One very cool property of using traits to define context variables is that we can use inheritance to describe more and more specific contexts:

```
trait LoggedIn extends Scope {
  val user = logInUser
  // do something with the user
}

trait HasAPendingOrder extends LoggedIn {
  val order = createPendingOrder
  // the user is logged in
  // now do something with the user and his order
}
```

Before/After

If you want to run some code before or after each example, the `Before` and `After` traits are there to help you (they both extend the `Scope` trait). In the following examples we'll only show the use of `After` because `Before` most of the time unnecessary:

```
class ContextSpec extends mutable.Specification {
  "this is the first example" in new trees {
    tree.removeNodes(2, 3) must have size(2)
  }
  "this is the first example" in new trees {
    tree.removeNodes(2, 3, 4) must have size(1)
  }
}

trait trees extends Scope {
  setupDB
  lazy val tree = getATreeWith4NodesFromTheDatabase
}
```

Indeed when you have setup code you can do anything you want in the body of your context trait and this will be executed before the example body. However this wouldn't work with teardown code, so let's see how to use the `After` trait.

In a mutable specification

You make your context trait extend the `mutable.After` trait:

```
class ContextSpec extends mutable.Specification {
  "this is the first example" in new trees {
    tree.removeNodes(2, 3) must have size(2)
  }
  "this is the first example" in new trees {
    tree.removeNodes(2, 3, 4) must have size(1)
  }
}

trait trees extends mutable.After {
  lazy val tree = getATreeWith4NodesFromTheDatabase
  def after = cleanupDB
}
```

In this case, the clean-up code defined in the `after` method will be executed after each example. This is possible because the `mutable.After` trait extends the Scala `DelayedInit` trait allowing to insert code around the execution of the body of an object.

Note: the `org.specs2.mutable.{ Before, After, BeforeAfter }` traits only work for scala > 2.9.0 because previous Scala versions don't provide the `DelayedInit` trait.

In an acceptance specification

In that case you would extend the `specification.After` trait and use the `apply` method:

```
class ContextSpec extends Specification { def is =
  "this is the first example" ! trees().e1 ^
  "this is the first example" ! trees().e2

  case class trees() extends specification.After {
    lazy val tree = getATreeWith4NodesFromTheDatabase
    def after = cleanupDB

    // this is equivalent to: def e1 = this.apply { ... }
    def e1 = this { tree.removeNodes(2, 3) must have size(2) }
    def e2 = this { tree.removeNodes(2, 3, 4) must have size(1) }
  }
}
```

Now we have both variable isolation and non-duplication of set-up code!

But there is more to it. The next paragraphs will show how to:

1. execute the body of each example inside a specific context: `Around`
2. set-up a context object (say a http query) and pass it to each example: `Outside`
3. declare a `before` method for all the examples of a Specification without even having to create a context object
4. use an implicit context to avoid duplication
5. create a new context object by combining existing ones

Around

Some examples need to be executed in a given context. For example you're testing a web application and your specification code needs to have your example executed inside an Http session.

In that case you can extend the `Around` trait and specify the `around` method:

```
object http extends Around {
  def around[T <% Result](t: =>T) = openHttpSession("test") {
    t  // execute t inside a http session
  }
}

"this is a first example where the code executes inside a http session" ! http(e1)
"and another one"                                                       ! http(e2)
```

Note that the context here is an object instead of a trait or case class instance because in this specification we don't need any variable isolation. We also take the advantage that objects extending `Context` traits (like `Before` / `After` / `Around`,...) have an `apply` method so we can directly write `http(e1)` meaning `http.apply(e1)`.

Outside

`Outside` is bit like `Around` except that you can get access to the application state that you're setting in your Context object. Let's see that with an example (with a mutable Specification for a change):

```
object http extends Outside[HttpReq] with Scope {
  // prepare a valid HttpRequest
  def outside: HttpReq = createRequest
}

// use the http request in each example
"this is a first example where the code executes uses a http request" in http { (request
  success
}
"and another one" in http { (request: HttpReq) =>
  success
}
```

AroundOutside

We can also combine both the `Around` and the `Outside` behaviors with the `AroundOutside` trait:

```
object http extends AroundOutside[HttpReq] {
  // create a context
  def around[T <% Result](t: =>T) = {
    createNewDatabase
    // execute the code inside a databaseSession
    inDatabaseSession { t }
  }
  // prepare a valid HttpRequest
  def outside: HttpReq = createRequest
}

"this is a first example where the code executes uses a http request" ! http((request: H
"and another one"                                                     ! http((request: H
```

BeforeExample

When you just need to have set-up code executed before each example and if you don't need to have variable isolation, you can simply use the `BeforeExample` trait.

The `BeforeExample` trait allows you to define a `before` method exactly like the one you define in the `Before` trait and apply it to all the examples of the specification:

```
class MySpecification extends mutable.Specification with BeforeExample {
  def before = cleanDatabase

  "This is a specification where the database is cleaned up before each example" >> {
    "first example" in { success }
    "second example" in { success }
  }
}
```

As you can guess, the `AfterExample`, `AroundExample`,... traits work similarly by requiring the corresponding `after`, `around`,... methods to be defined.

Implicit context

The `BeforeExample` trait is a nice shortcut to avoid the creation of a context object, but there is another possibility to avoid the repetition of the context name for each example. If your specification is:

```
class ContextSpec extends mutable.Specification {
  object myContext = new Before { def before = cleanUp }

  "This is a specification where the database is cleaned up before each example" >> {
    "first example" in myContext { 1 must_== 1 }
    "second example" in myContext { 1 must_== 1 }
  }
}
```

You can simply mark your context object as `implicit` and it will be automatically passed to each example:

```
class ContextSpec extends mutable.Specification {
  implicit object myContext = new Before { def before = cleanUp }

  "This is a specification where the database is cleaned up before each example" >> {
    "first example"  in { 1 must_== 1 }
    "second example" in { 1 must_== 1 }
  }
}
```

There is just one gotcha that you need to be aware of. If your implicit context is an `Outside[String]` context this will not work:

```
class ContextSpec extends mutable.Specification {
  implicit object myContext = new Outside[String] { def outside = "hello" }

  "This is a specification uses a new String in each example" >> {
    "first example"  in { (s: String) => s must_== s }
    "second example" in { (s: String) => s must_== s }
  }
}
```

Indeed in both examples above the `s` string that will be passed is the Example description as specified here.

Composition

Combinations

specs2 contexts can be combined in several ways. When you want to define both `Before` and `After` behavior, you can do it by simply extending those 2 traits:

```
case class withFile extends Before with After {
  def before = createFile("test")
  def after  = deleteFile("test")
}
```

But, as we've seen with the `AroundOutside` example, ***specs2*** likes to help save keystrokes so you can directly extend the `BeforeAfter` trait:

```
case class withFile extends BeforeAfter {
  def before = createFile("test")
  def after  = deleteFile("test")
}
```

Similarly you can use `BeforeAfterAround` instead of `Before with After with Around`.

Composition

Contexts can be also be *composed* but only if they are of the same type, `Before` with `Before`, `After` with `After`,...

```
case class withFile extends Before {
  def before = createFile("test")
}
case class withDatabase extends Before {
  def before = openDatabase("test")
}
val init = withFile() compose withDatabase()

"Do something on the full system" ! init(success)
```

Steps/Actions

Steps

Some set-up actions are very time-consuming and should be executed only once for the whole specification. This can be achieved by inserting some silent `Steps` in between fragments:

```
class DatabaseSpec extends Specification { def is =

  "This specification opens a database and execute some tests"     ^ Step(openDatabase)
    "example 1"                                                    ! success ^
    "example 2"                                                    ! success ^
                                                                   Step(closeDatabase)^
                                                                   end
}
```

The examples are (by default) executed concurrently between the 2 steps and the "result" of those steps will never be reported unless if there is a failure.

Actions

`Steps` are very useful because they will really be executed sequentially, before anything else, but if you need to execute some actions which are completely independent of the rest of the specification, there is an equivalent to `Step` adequately called `Action`:

```
class DatabaseSpec extends Specification { def is =

  "This specification opens a database and execute some tests"     ^ Step(openDatabase)
    "example 1"                                                    ! success ^
    "add 1 to the number of specification executions"              ^ Action(db.execution
    "example 2"                                                    ! success ^
                                                                   Step(closeDatabase)^
                                                                   end
}
```

Of course, `Steps` and `Actions` are not the privilege of acceptance specifications:

```
class DatabaseSpec extends mutable.Specification {

  textFragment("This specification opens a database and execute some tests")
  step(openDatabase)

  "example 1" in success

  textFragment("add 1 to the number of specification executions")
  action(db.executionsNb += 1)
```

```
  "example 2" in success
  step(closeDatabase)
}
```

Template

There may still be some duplication of code if you have to use the same kind of set-up procedure for several specifications.

If that's the case you can define your own `Specification` trait doing the job:

```
import org.specs2._
import specification._

trait DatabaseSpec extends Specification {
  /** the map method allows to "post-process" the fragments after their creation */
  override def map(fs: =>Fragments) = Step(startDb) ^ fs ^ Step(cleanDb)
}
```

The `DatabaseSpec` above will insert, in each inherited specification, one `Step` executed before all the fragments, and one executed after all of them.

For fragments

When using a Unit Specification, it can be useful to use variables which are only used for a given set of examples. This can be easily done by declaring local variables, but this might lead to duplication. One way to avoid that is to use the `org.specs2.mutable.NameSpace` trait:

```
trait context extends mutable.NameSpace {
  var variable1 = 1
  var variable2 = 2
}

"this is the first block" >> new context {
  "using one variable"      >> { variable1 === 1 }
  "using a second variable" >> { variable2 === 2 }
}
"this is the second block" >> new context {
  "using one variable"      >> { variable1 === 1 }
  "using a second variable" >> { variable2 === 2 }
}
```

Execution

This section summarizes the execution algorithm of a specification based on its fragments:

1. all the fragments are divided into groups delimited by `Steps`
2. if the `sequential` argument is present, each fragment goes to its own group
3. groups are executed sequentially and all the fragments of a given group are executed concurrently
4. if the `isolated` argument is present, each example is executed in its own version of the Specification
5. if the `isolated` argument is present, all the `Steps` preceding an example are executed before that example
6. if the Specification inherits from the `AllExpectations` trait, then it is executed as an `isolated` Specification unless it is already set as `sequential`
7. if the `stopOnFail` argument is present, all the examples in the next group of fragments will be skipped if there is a failure in one of the previous groups

8. if the `stopOnSkip` argument is present, all the examples in the next group of fragments will be skipped if there is a skipped in one of the previous groups
9. if there is a `Step(stopOnFail = true)`, all the examples in the next group of fragments will be skipped if there is a failure in the group before the `Step`

Layout

For an *acceptance* specification you can tweak the layout of Texts and Examples.

Rules

The layout of text in ***specs2*** is mostly done automatically so that the text in the source code should look like the displayed text after execution.

By default the layout of a specification will be computed automatically based on intuitive rules:

- when an example follows a text, it is indented
- 2 successive examples will be at the same indentation level
- when a text follows an example, this means that you want to describe a "subcontext", so the next examples will be indented with one more level

Let's see a standard example of this. The following fragments:

```
"this is some presentation text"      ^
  "and the first example"             ! success^
  "and the second example"            ! success
```

will be executed and displayed as:

```
this is some presentation text
+ and the first example
+ and the second example
```

If you specify a "subcontext", you will get one more indentation level:

```
"this is some presentation text"      ^
  "and the first example"             ! success^
  "and the second example"            ! success^
  "and in this specific context"      ^
    "one more example"                ! success^
```

will be executed and displayed as:

```
this is some presentation text
+ and the first example
+ and the second example
  and in this specific context
  + one more example
```

Formatting fragments

Given the rules above, you might need to use some *formatting fragments* to adjust the display

Separating groups of examples

The best way to separate blocks of examples is to add a blank line between them by using p (as in "paragraph"):

```
"this is some presentation text"      ^
  "and the first example"             ! success^
  "and the second example"            ! success^
                                      p^
```

```
"And another block of examples"       ^
  "with this example"                 ! success^
  "and that example"                  ! success
```

This will be displayed as:

```
this is some presentation text
+ and the first example
+ and the second example

And another block of examples
+ with this example
+ and that example
```

That looks remarkably similar to the specification code, doesn't it? What `p` does is:

- add a blank line (this can also be done with a simple `br`)
- decrement the current indentation level by 1 (Otherwise the new Text would be seen as a subcontext)

Reset the levels

When you start having deep levels of indentation, you might need to start the next group of examples at level 0. For example, in this specification

```
"There are several options for displaying the text"      ^
  "xonly displays nothing but failures"                  ! success^
  "there is also a color option"                         ^
    "rgb=value uses that value to color the text"        ! rgb^
    "nocolor dont color anything"                        ! nocolor^
                                                         p^
"There are different ways of hiding the text"            ^
    "by tagging the text"                                ! hideTag
```

Even with `p` the next group of examples will not start at level 0. What you need to do in that case is use `end`:

```
"There are several options for displaying the text"      ^
  "xonly displays nothing but failures"                  ! success^
  "there is also a color option"                         ^              // this text wil
    "rgb=value uses that value to color the text"        ! rgb^         // and the follo
    "nocolor dont color anything"                        ! nocolor^
                                                         end^
"There are different ways of hiding the text"            ^              // this text wil
  "by tagging the text"                                  ! hideTag^
                                                         end
```

This will be displayed as:

```
There are several options for displaying the text
+ xonly displays nothing but failures
  there is also a color option
  + rgb=value uses that value to color the text
  + nocolor dont color anything
There are different ways of hiding the text
+ by tagging the text
```

And if you want to reset the indentation level *and* add a blank line you can use `end ^ br` (or `endbr` as seen in "Combinations" below).

Changing the indentation level

If, for whatever reason, you wish to have more or less indentation, you can use the `t` and `bt` fragments (as in "tab" and "backtab"):

```
"this text"                                          ^ bt^
"doesn't actually have an indented example"          ! success

"this text"                                          ^ t^
    "has a very indented example"                    ! success
```

The number of indentation levels (characterized as 2 spaces on screen) can also be specified by using `t(n)` or `bt(n)`.

Combinations

Some formatting elements can be combined:

- `p` is actually `br ^ bt`
- `endbr` is `end ^ br`
- `endp` is `end ^ p` (same effect as `endbr` but shorter :-))

Turning-off the automatic layout

You can turn off that automatic layout by adding the `noindent` argument at the beginning of your specification:

```
class MySpecWithNoIndent extends Specification {
  def is = noindent ^ ....
}
```

Unit specification

Formatting fragments can be used in a unit specification as well. 2 forms are supported, either as a single declaration:

```
"this is an example" >> { 1 === 1 }
p // add a paragraph
"this is another example" >> { 2 === 2 }
```

Or as a postfix operator on fragments:

```
"this is some text and a paragraph".p
"this is an example and a paragraph" >> {
  1 must_== 1
} p
```

There are also 2 additional postfix operations which can be used to start new paragraphs. Instead of using `endp` to end a group of examples and starte a new one:

```
"This is a first block of examples".p
{ 1 === 1 }.eg;
{ 2 === 2 }.eg.endp

"And a second block".p
{ 3 === 3 }.eg;
{ 4 === 4 }.eg
```

You can use `newp` (or `newbr`) to the same effect:

```
"This is a first block of examples".p
{ 1 === 1 }.eg;
{ 2 === 2 }.eg

"And a second block".newp
{ 3 === 3 }.eg;
{ 4 === 4 }.eg
```

A shortcut is also available to indent a 'subexample' locally:

```
"this is the first major example" >> { ok }
    "this is minor and should be indented" >> { ok } lt;
  "this is the second major example" >> { ok }
}
```

This will output:

```
this is a group of examples
+ this is the first major example
  + this is minor and should be indented
+ this is the second major example
```

Unit specifications

Those are all the methods which you can use to create fragments in a unit specification:

- `can`: create a group of Examples, with the preceding Text fragment appended with `can`

  ```
  "a configuration" can {
    "have a name" in { ... }
  }
  ```

- `>>`: create an Example or a group of Examples (with no appended text)

  ```
  "a configuration may" >> {
    "have a name" in { ... }
  }
  ```

Note that you can use a `for` loop to create examples with `>>`:

```
"this system has 5 examples" >> {
  (1 to 5) foreach { i => "example "+i >> ok }
}
```

And you can also use a `for` loop with the `in` operator to create a block of expectations:

```
"this example has 5 expectations" in {
  (1 to 5) foreach { i => i must_== i }
}
```

- `title`: give a title to the Specification

  ```
  "My spec title".title
  // file path can be used to specify a different path for the html reporting
  "My spec title".title(filePath = "com/MySpec.html")
  ```

- `args`: create arguments for the specification

- `.txt` or `textFragment`: create a `Text` fragment

  ```
  "this is a text fragment".txt

  textFragment("this is a text fragment")
  ```

- `step`: create a `Step`

  ```
  step { initializeDatabase() }
  ```

- `action`: create an `Action`

  ```
  action { justDoIt }
  ```

- `link`: create a link to another specification

  ```
  link("how" ~ ("to do hello world", new HelloWorldSpec))
  ```

- `see`: add a link to another specification without including its fragments for execution

  ```
  see(new HelloWorldSpec)
  ```

- `include` to include another specification

  ```
  include(new HelloWorldSpec)
  ```

- `p, br, t, bt, end, endp`: add a formatting fragment

To make things more concrete here is a full example:

```
import mutable._
import specification._
import execute.Success

/**
 * This specification shows how to use the mutable.Specification trait to create a unit
 * where the fragments are built using a mutable variable
 */
class MutableSpec extends Specification {

  // A title can be added at the beginning of the specification
  "MutableSpec".title
  // arguments are simply declared at the beginning of the specification if needed
  args(xonly=true)

  "This is a unit specification showing the use of different methods".txt

  // a step to execute before the specification must be declared first
  step {
    // setup your data or initialize your database here
    success
  }

  "'Hello world'" should {
    "contain 11 characters" in {
      "Hello world" must have size(11)
    }
    "start with 'Hello'" in {
      "Hello world" must startWith("Hello")
    }
    /**
     * a failing example will stop right away, without having to "chain" expectations
     */
    "with 'world'" in {
      // Expectations are throwing exception by default so uncommenting this line will
      // stop the execution right away with a Failure
      // "Hello world" must startWith("Hi")

      "Hello world" must endWith("world")
    }
  }
  /**
   * "Context management" is handled through the use of traits or case classes
   */
  "'Hey you'" should {
    // this one uses a "before" method
    "contain 7 characters" in context {
      "Hey you" must have size(7)
    }
```

```
    // System is a Success result. If the expectations fail when building the object, th
    "contain 7 characters" in new system {
      string must have size(7)
    }
    // otherwise a case class can be used but the example body will be further down the
    "contain 7 characters" in system2().e1
  }
  // you can add links to other specifications with `link`
  // they will be executed when this one is executed. If you don't want this to happen
  // you can use `see` instead of `link`
  link("how" ~ ("to do hello world", new HelloWorldSpec))
  // you can include other specifications with `include`
  include(new HelloWorldSpec)

  // a step to execute after the specification must be declared at the end
  step {
    // close the database here
    success
  }


  object context extends Before {
    def before = () // do something to setup the context
  }
  // we need to extend Scope to be used as an Example body
  trait system extends Scope {
    val string = "Hey you"
  }
  case class system2() {
    val string = "Hey you"
    def e1 = string must have size(7)
  }
}
```

How to?

Declare arguments

Arguments are usually passed on the command line but you can also declare them at the beginning of the specification, to be applied only to that specification.
For example, you can turn off the concurrent execution of examples with the `args(sequential=true)` call:

```
class ExamplesOneByOne extends Specification { def is =

  // there is a shortcut for this argument called 'sequential'
  args(sequential=true)             ^
  "first example"                   ! e1 ^
  "the the second one"              ! e2 ^
                                    end
}
```

For the complete list of arguments and shortcut methods read the Runners page.

Pass arguments

Some specifications can depend on the arguments passed on the command line, for example to fine-tune the behaviour of some Context objects. If you need to do this, you can add an `Arguments` parameter to the Specification class. This parameter will be setup when the specification is instantiated:

```
class DependOnCommandLine(args: Arguments) extends mutable.Specification {
  skipAllUnless(!args.commandLine.contains("DB"))
  "database access" >> { dbAccess must beOk }
}
```

Alternatively, if you need to keep your specification as a trait, you can mix-in the `org.specs2.main.CommandLineArguments` trait. This trait has an `arguments` variable which will contain the command-line arguments:

```
class CommandedSpecification extends mutable.Specification with CommandLineArguments {
  if (arguments.sequential) "this is" >> ok
  else                      "this is" >> ko
}
```

Note that the `arguments` instance gives you access to all the specs2 arguments values like `sequential` but also to any of your own command line argument values:

- `arguments.commandLine.value("tag"): Option[String]`
- `arguments.commandLine.int("timeout"): Option[Int]`
- `arguments.commandLine.boolean("integration"): Boolean`

Add a title

Usually the title of a specification is derived from the specification class name. However if you want to give a more readable name to your specification report you can do the following:

```
class MySpec extends Specification { def is =
   "My beautiful specifications".title                          ^
                                                                p^
   "The rest of the spec goes here"                             ^ end
}
```

The title can be defined either:

- at the beginning of the specification
- just after the arguments of the specification

Use descriptions

The description of an Example can be used to create an expectation in the example body:

```
"This is a long, long, long description" ! ((s: String) => s.size must be_>(10))
```

Enhance failures

Most of the time, the message displayed in the case of a matcher failure is clear enough. However a bit more information is sometimes necessary to get a better diagnostic on the value that's being checked. Let's say that you want to check a "ticket list":

```
// will fail with "List(ticket1, ticket2) doesn't have size 3" for example
machine.tickets must have size(3) // machine is a user-defined object
```

If you wish to get a more precise failure message you can set an alias with the `aka` method (*also known as*):

```
// will fail with "the created tickets 'List(ticket1, ticket2)' doesn't have size 3"
machine.tickets aka "the created tickets" must haveSize(3)
```

There is also a shortcut for `value aka value.toString` which is simply `value.aka`.

And when you want other ways to customize the description, you can use:

- `post`: `"a" post "is the first letter"` prints `a is the first letter`
- `as`: `"b" as ((s:String) => "a"+s+"c")` prints `abc`
- `showAs`: `Seq(1, 2, 3, 4).showAs((_:Seq[Int]).filter(isEven).mkString("|"))` prints `2|4`. This one is especially useful to filter out big data structures (lists, maps, xml...) before the failure display

Share examples

In a given specification some examples may look similar enough that you would like to "factor" them out and share them between
different parts of your specification. The best example of this situation is a specification for a Stack of limited size:

```
class StackSpec extends Specification { def is =

  "Specification for a Stack with a limited capacity".title                  ^
                                                                             p^
  "A Stack with limited capacity can either be:"                             ^ endp^
    "1. Empty"                                                               ^ anEmptyS
    "2. Normal (i.e. not empty but not full)"                                ^ aNormalS
    "3. Full"                                                                ^ aFullSta

  def anEmptyStack =                                                         p^
    "An empty stack should"                                                  ^
      "have a size == 0"                                                     ! empty().
      "throw an exception when sent #top"                                    ! empty().
      "throw an exception when sent #pop"                                    ! empty().

  def aNormalStack =                                                         p^
    "A normal stack should"                                                  ^
      "behave like a non-empty stack"                                        ^ nonEmpty
      "add to the top when sent #push"                                       ! nonFullS

  def aFullStack =                                                           p^
    "A full stack should"                                                    ^
      "behave like a non-empty stack"                                        ^ nonEmpty
      "throw an exception when sent #push"                                   ! fullStac

  def nonEmptyStack(stack: =>SizedStack) = {                                 t^
    "have a size > 0"                                                        ! nonEmpty
    "return the top item when sent #top"                                     ! nonEmpty
    "not remove the top item when sent #top"                                 ! nonEmpty
    "return the top item when sent #pop"                                     ! nonEmpty
    "remove the top item when sent #pop"                                     ! nonEmpty
  }

  /** stacks creation */
  def newEmptyStack  = SizedStack(maxCapacity = 10, size = 0)
  def newNormalStack = SizedStack(maxCapacity = 10, size = 2)
  def newFullStack   = SizedStack(maxCapacity = 10, size = 10)

  /** stacks examples */
```

```
    case class empty() {
      val stack = newEmptyStack

      def e1 = stack.size must_== 0
      def e2 = stack.top must throwA[NoSuchElementException]
      def e3 = stack.pop must throwA[NoSuchElementException]
    }

    def nonEmpty(createStack: =>SizedStack) = new {
      val stack = createStack

      def size = stack.size > 0

      def top1 = stack.top must_== stack.size
      def top2 = {
        stack.top
        stack.top must_== stack.size
      }

      def pop1 = {
        val topElement = stack.size
        stack.pop must_== topElement
      }

      def pop2 = {
        stack.pop
        stack.top must_== stack.size
      }
    }

    case class nonFullStack() {
      val stack = newNormalStack

      def e1 = {
        stack push (stack.size + 1)
        stack.top must_== stack.size
      }
    }
    case class fullStack() {
      val stack = newFullStack

      def e1 = stack push (stack.size + 1) must throwAn[Error]
    }
  }
```

Create an index

Here's something you can do to automatically create an index page for your specifications:

```
  import org.specs2._
  import runner.SpecificationsFinder._

  class index extends Specification { def is =

    examplesLinks("Example specifications")

    // see the SpecificationsFinder trait for the parameters of the 'specifications' metho
    def examplesLinks(t: String) = specifications().foldLeft(t.title) { (res, cur) => res
  }
```

The specification above creates an index.html file in the `target/specs2-reports` directory. The specifications method
creates specifications using the following parameters:

- `path`: glob pattern to filter specification files. Default value is `**/*.scala`
- `pattern`: pattern to use when trying to retrieve the specification names from the source files. Default value = `.*Spec`
- `filter`: function to keep only some specifications depending on their name. Default value = `(name: String) => true`
- `basePath`: the path where to start the search. Default value: the `specs2.srcTestDir` system value = `src/test/scala`
- `verbose`: boolean indicating if information about finding files and specifications must be printed. Default value = `false`

Tag examples

Tags can be used in a Specification to include or exclude some examples or a complete section of fragments from the execution. Let's have a look at one example:

```
/**
 * use the org.specs2.specification.Tags trait to define tags and sections
 */
class TaggedSpecification extends Specification with Tags { def is =
  "this is some introductory text"                          ^
  "and the first group of examples"                         ^
    "example 1"                                             ! success ^ tag("feature1",
    "example 2"                                             ! success ^ tag("integration
                                                            ^ p^
  "and the second group of examples"                        ^          section("checkin"
    "example 3"                                             ! success^
    "example 4"                                             ! success^ section("checkin"
}
```

In that specification we're defining several tags and sections:

- `feature 1` is a tag that's applied to `example1` (the *preceding* Fragment)
- `feature 2` is a tag that's applied to `example2` (the *preceding* Fragment)
- `checkin` marks a section which goes from the Text `and the second group of examples` to `example 4`

Armed with this, it is now easy to include or exclude portions of the specification at execution time:

- `args(include="feature1")` will only include `example 1`
- `args(exclude="integration")` will include everything except `example 2`
- `args(include="checkin,unit")` will include anything having either `checkin` OR `unit`: i.e. `example 1` and the second group of examples (`example 3` and `example 4`)
- `args(include="feature1 && unit")` will include anything having `feature1` AND `unit`: i.e. `example 1`
- `args(include="feature1 && unit, checkin")` will include anything having `feature1` AND `unit`, OR having `checkin`: i.e. `example 1`, `example 3`, `example4`

In a unit specification

A *unit* specification will accept the same `tag` and `section` methods but the behavior will be slightly different:

```
import org.specs2.mutable._

/**
 * use the org.specs2.mutable.Tags trait to define tags and sections
 */
class TaggedSpecification extends Specification with Tags {
  "this is some introductory text" >> {
    "and the first group of examples" >> {
      tag("feature 1", "unit")
      "example 1" in success
```

```
      "example 2" in success tag("integration")

    }
  }
  section("checkin")
  "and the second group of examples" >> {
    "example 3" in success
    "example 4" in success
  }
  section("checkin")

  "and the last group of examples" >> {
    "example 5" in success
    "example 6" in success
  } section("slow")
}
```

For that specification above:

- when the `tag` call is inserted on a new line, the tagged fragment is the one just *after* the tag method call:
  `example 1`
  is tagged with `feature1 and unit`,
- when the `tag` is appended to an example, it applies to that example: `example 2` is tagged with `integration`
- when the `section` call is inserted on a new line, this opens a section for all the following fragments. This should
  be closed by a corresponding `section` call on a new line. For example, `example 3` and `example 4` are part of the
  "checkin" section
- when the `section` call is appended to a block of Fragments on the same line, all the fragments of that block are part of
  the section: `example 5` and `example 6` are tagged with `slow`

Skip examples

You can skip all the examples of a specification by using the `skipAllIf` or `skipAllUnless` methods:

```
class EmailSpecification extends mutable.Specification {
  skipAllIf(serverIsOffLine)
  "test email" >> { sendEmail must beOk }
}
```

Debug statements

When quick and hacky `println` statements are what you want, the `Debug` trait, mixed in every `Specification`, provides useful methods:

- `pp` or "print and pass", prints a value to the console, then return it to be used in the rest of the expression: "graph.pp must haveSize(3)"
- `pp(condition)` prints a value if a condition holds
- `pp(f: T => Boolean)` prints a value if a condition on that value holds

Remove implicits

By default, the `Specification` trait imports quite a few implicit definitions (following a "batteries included" approach). However there might be some conflicts with implicits existing in your own user code. Among the usual examples of conflicts are conflicts with the `===` sign in Scalaz and the `Duration` methods in Akka.

An easy way to avoid this situation is to "deactivate" the specs2 implicits by mixing-in the relevant trait from this list:

- `org.specs2.control.NoDebug`: deactivate the `pp` method on objects
- `org.specs2.time.NoTimeConversions`: deactivate the `millis`, `seconds`,... methods on `Int`s and `Long`s
- `org.specs2.main.NoArgProperties`: deactivate the `toOption: Option[T]` method on any value of type `T`
- `org.specs2.matcher.NoCanBeEqual`: deactivate the `===` method on any type `T`
- `org.specs2.matcher.NoMustExpectations`: deactivate the `must`, `must_==`,... methods on any value of type `T`
- `org.specs2.matcher.NoShouldExpectations`: deactivate the `should`, `should_==`,... methods on any value of type `T`
- `org.specs2.specification.NoAutoExamples`: deactivate the conversions from `Boolean/Result/MatchResult/DataTable` to `Fragment` or `Example`. Specific versions of this trait can be selectively used, on either `Boolean` or `Result` or `MatchResult` or `DataTable`. For example: `org.specs2.specification.NoBooleanAutoExamples` can be used to avoid the `^` method being used on booleans
- `org.specs2.specification.NoFragmentsBuilder`: deactivate the implicit conversions from `String` to `Fragments`
- `org.specs2.specification.mutable.NoFragmentsBuilder`: deactivate the implicit conversions from to remove `in`, `>>`, `should` and `can` methods from `Strings`

Total for specification Structure	
Finished in	318 ms
Results	18 examples, 0 failure, 0 error