Chest

Adrien Vanègue, Pierre Laborde and Steven Costiou

March 12, 2024

Layout and typography based on the sbabook LaTeX class by Damien Pollet.

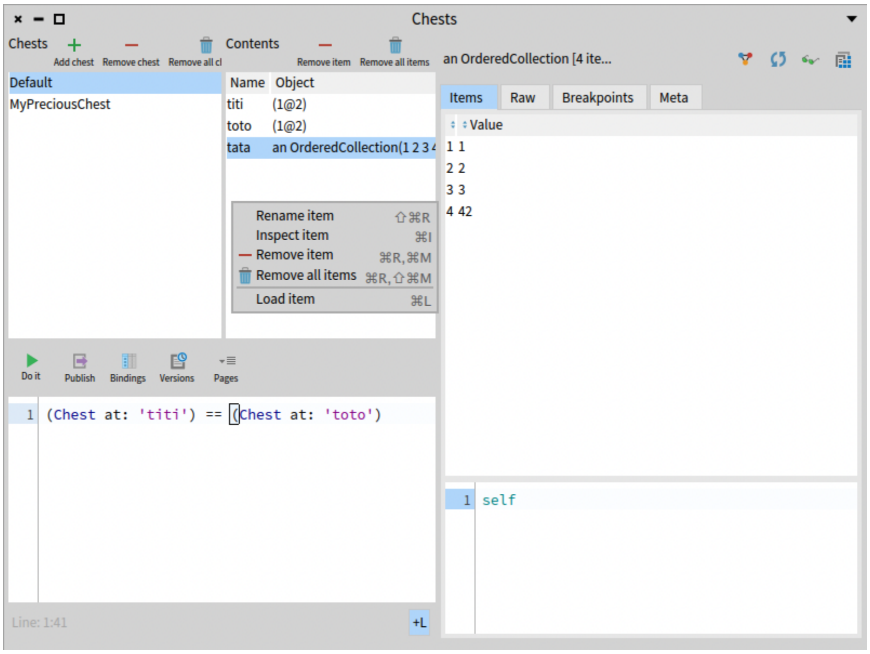
Contents

Illustrations

CHAPTER 1

Introduction

Chest allows you to store objects from anywhere, to keep them around to check equality...



Name	Object
titi	(1@2)
toto	(1@2)
tata	an OrderedCollection(1 2 3 4

	Value
1	1
2	2
3	3
4	42

```
(Chest at: 'titi') == (Chest at: 'toto')
```

```
self
```

Figure 1-1 Chest overview

1.1 Original repository

Link to original repositoryhttps://github.com/dupriezt/Chest

1.2 Install Chest

```
Metacello new
    baseline: 'Chest';
    repository: 'github://pharo-spec/Chest';
    load.
```

1.3 Open Chest

Chest is available in the **world menu** of Pharo. It is available as an **entry of the Debug menu**

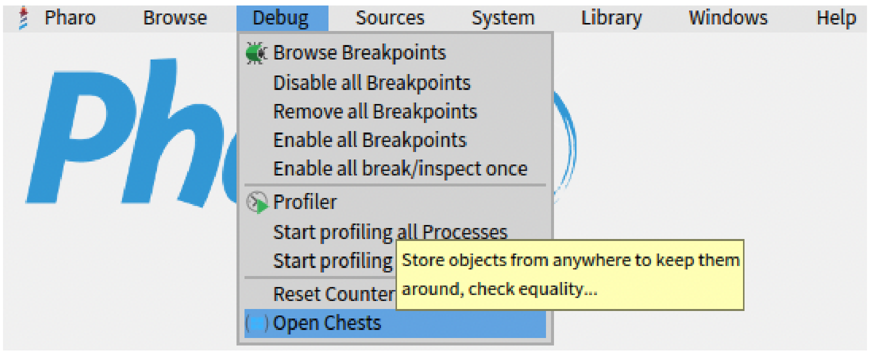


Figure 1-2 Chest in debug menu

You can also enable it as a debugger extension in the debugging settings of Pharo:

In the debugger, Chest will provide a view as a tree of all chests with the objects inside.

```
Maybe we need an image of the tree view here?
```

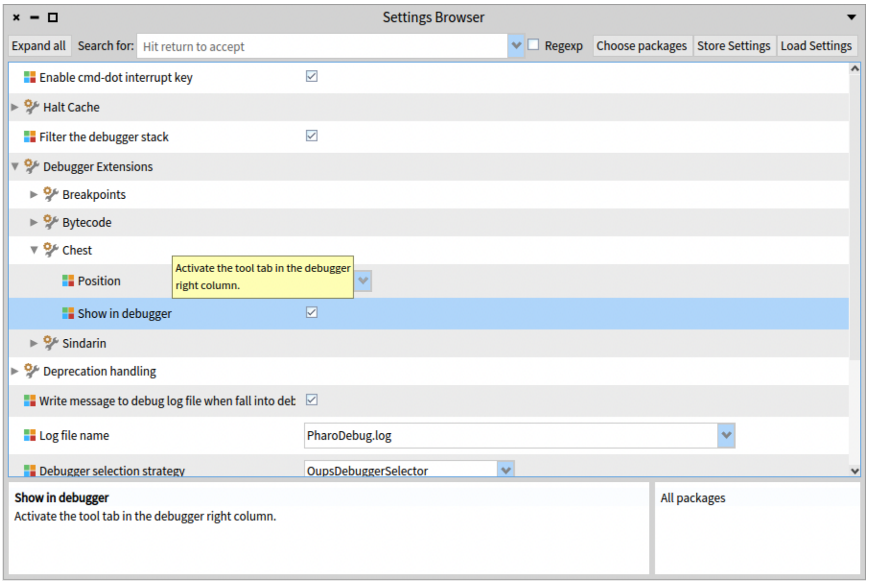
1.4 More Details

Name (= ID)

Each Chest instance has an ID (String). These IDs are unique. Two chests cannot have the same ID.

Default Chest

This is an instance of Chest that can be interacted with in the same way as any other Chest by sending the messages to the `Chest` class.



Enable cmd-dot interrupt key ✓
Halt Cache
Filter the debugger stack ✓
Debugger Extensions
Breakpoints
Bytecode
Chest
Position
Activate the tool tab in the debugger right column.
Show in debugger ✓
Sindarin
Deprecation handling
Write message to debug log file when fall into deb ✓
Log file name PharoDebug.log
Debugger selection strategy OupsDebuggerSelector

Show in debugger
Activate the tool tab in the debugger right column.

All packages

Figure 1-3 Chest debugger extension in the Pharo settings

Weak Chests

```
Chest weak
```

is a way to access the class `WeakChest`. This class can be interacted with the same way as you would with the `Chest` class.

`WeakChest` has its own default chest, and can be used to manipulate this default chest or to create new weak chests.

Commands in context

Store object into chest

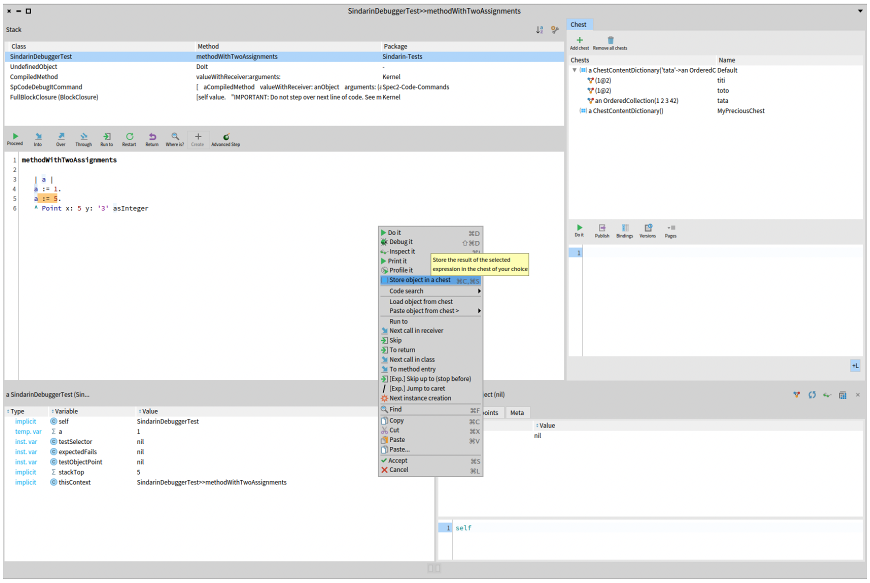
If you right-click on a code presenter with an `SpCodeInteractionModel` or `StDebuggerContextInteractionModel` (e.g: playground, debugger etc.), you can evaluate an expression and store the result in the chest of your choice, with the name of your choice:

Load object from a chest into a playground or a debugger

It's also possible to load objects from a chest into these code presenters:

And then variables can be seen from any other context if it has been loaded in a debugger and these variables can be seen in the debugger inspector:

Chest, as a debugger extension, provides a playground. All bindings between this playground and the debugger selected context are shared. So: all variables defined in this playground are recognized by the debugger and all variables from the debugger's selected context or loaded from Chest into the debugger are recognized by the playground. However, only the variables loaded from Chest (via the load menu entry described above) are displayed in the debugger inspector:



Class	Method	Package
SindarinDebuggerTest	methodWithTwoAssignments	Sindarin-Tests
UndefinedObject	DoIt	-
CompiledMethod	valueWithReceiver:arguments:	Kernel
SpCodeDebugItCommand	[aCompiledMethod valueWithReceiver: anObject arguments: (:	Spec2-Code-Commands
FullBlockClosure (BlockClosure)	[self value. "IMPORTANT: Do not step over next line of code. See m	Kernel

```
methodWithTwoAssignments

  | a |
  a := 1.
  a := 5.
  ^ Point x: 5 y: '3' asInteger
```

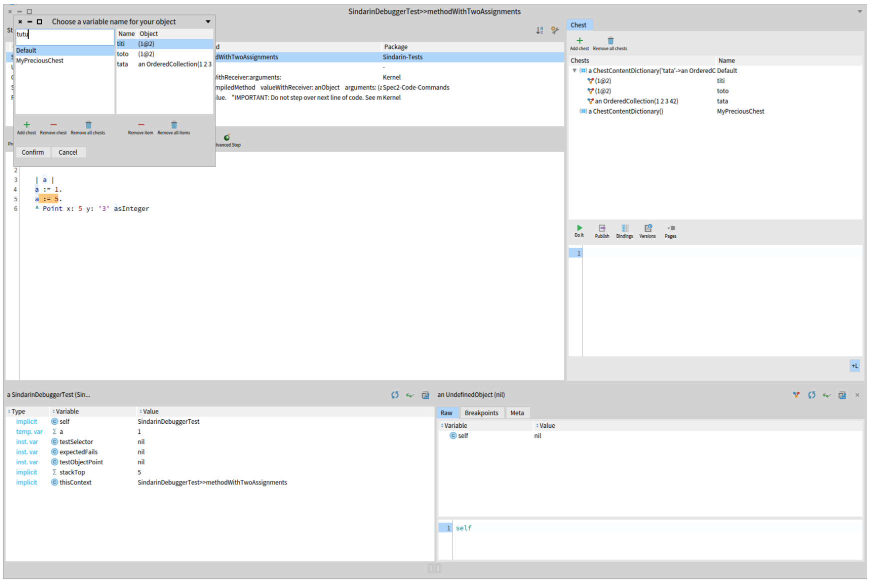
Chests	Name
a ChestContentDictionary('tata'->an OrderedC	Default
(1@2)	titi
(1@2)	toto
an OrderedCollection(1 2 3 42)	tata
a ChestContentDictionary()	MyPreciousChest

a SindarinDebuggerTest (Sin...

Type	Variable	Value
implicit	self	SindarinDebuggerTest
temp. var	a	1
inst. var	testSelector	nil
inst. var	expectedFails	nil
inst. var	testObjectPoint	nil
implicit	stackTop	5
implicit	thisContext	SindarinDebuggerTest>>methodWithTwoAssignments

```
self
```

Figure 1-4 Store object context menu, inside the debugger



tutu

Name	Object
titi	(1@2)
toto	(1@2)
tata	an OrderedCollection(1 2 3

Default
MyPreciousChest

```
  | a |
  a := 1.
  a := 5.
  ^ Point x: 5 y: '3' asInteger
```

Chests	Name
a ChestContentDictionary('tata'->an OrderedC	Default
(1@2)	titi
(1@2)	toto
an OrderedCollection(1 2 3 42)	tata
a ChestContentDictionary()	MyPreciousChest

a SindarinDebuggerTest (Sin...

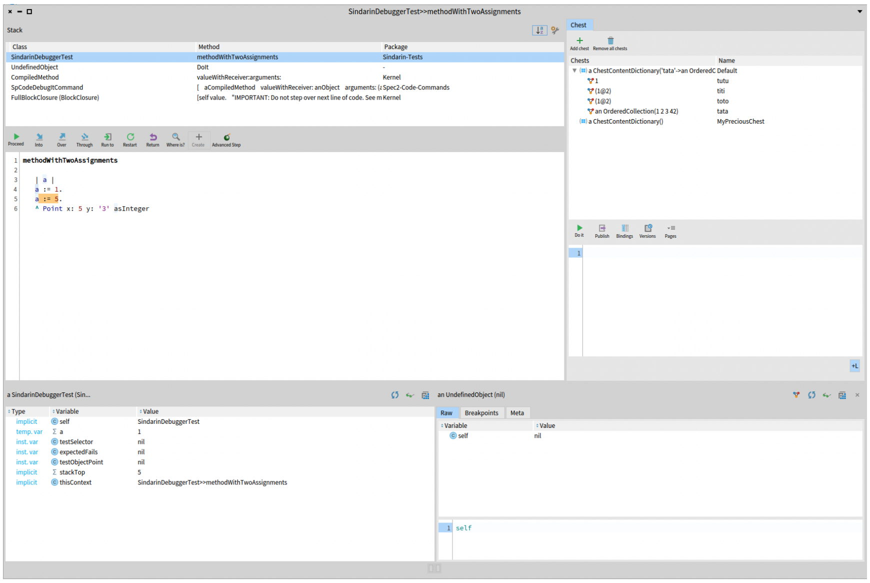
Type	Variable	Value
implicit	self	SindarinDebuggerTest
temp. var	a	1
inst. var	testSelector	nil
inst. var	expectedFails	nil
inst. var	testObjectPoint	nil
implicit	stackTop	5
implicit	thisContext	SindarinDebuggerTest>>methodWithTwoAssignments

an UndefinedObject (nil)

Variable	Value
self	nil

```
self
```

Figure 1-5 Store object popup


SindarinDebuggerTest>>methodWithTwoAssignments

Stack

Class	Method	Package
SindarinDebuggerTest	methodWithTwoAssignments	Sindarin-Tests
UndefinedObject	DoIt	-
CompiledMethod	valueWithReceiver:arguments:	Kernel
SpCodeDebugItCommand	[aCompiledMethod valueWithReceiver: anObject arguments: (a	Spec2-Code-Commands
FullBlockClosure (BlockClosure)	[self value. "IMPORTANT: Do not step over next line of code. See m	Kernel

```
methodWithTwoAssignments

   | a |
   a := 1.
   a := 5.
   ^ Point x: 5 y: '3' asInteger
```

Chest

Chests	Name
a ChestContentDictionary('tata'->an OrderedC	Default
1	tutu
(1@2)	titi
(1@2)	toto
an OrderedCollection(1 2 3 42)	tata
a ChestContentDictionary()	MyPreciousChest

a SindarinDebuggerTest (Sin...

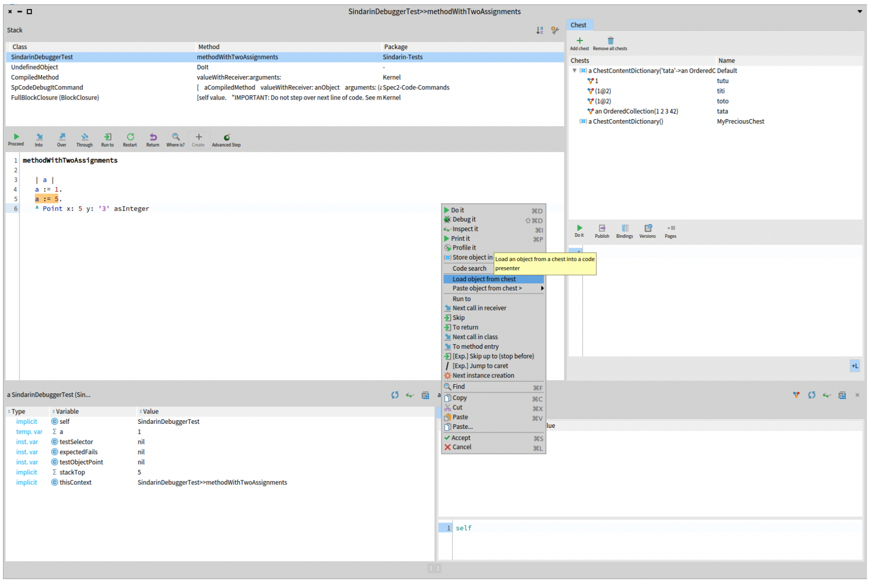
Type	Variable	Value
implicit	self	SindarinDebuggerTest
temp. var	a	1
inst. var	testSelector	nil
inst. var	expectedFails	nil
inst. var	testObjectPoint	nil
implicit	stackTop	5
implicit	thisContext	SindarinDebuggerTest>>methodWithTwoAssignments

an UndefinedObject (nil)

Variable	Value
self	nil

```
self
```

Figure 1-6 Chest tree view inside the debugger, after having stored an object


SindarinDebuggerTest>>methodWithTwoAssignments

Stack

Class	Method	Package
SindarinDebuggerTest	methodWithTwoAssignments	Sindarin-Tests
UndefinedObject	DoIt	-
CompiledMethod	valueWithReceiver:arguments:	Kernel
SpCodeDebugItCommand	[aCompiledMethod valueWithReceiver: anObject arguments: (a	Spec2-Code-Commands
FullBlockClosure (BlockClosure)	[self value. "IMPORTANT: Do not step over next line of code. See m	Kernel

```
methodWithTwoAssignments

   | a |
   a := 1.
   a := 5.
   ^ Point x: 5 y: '3' asInteger
```

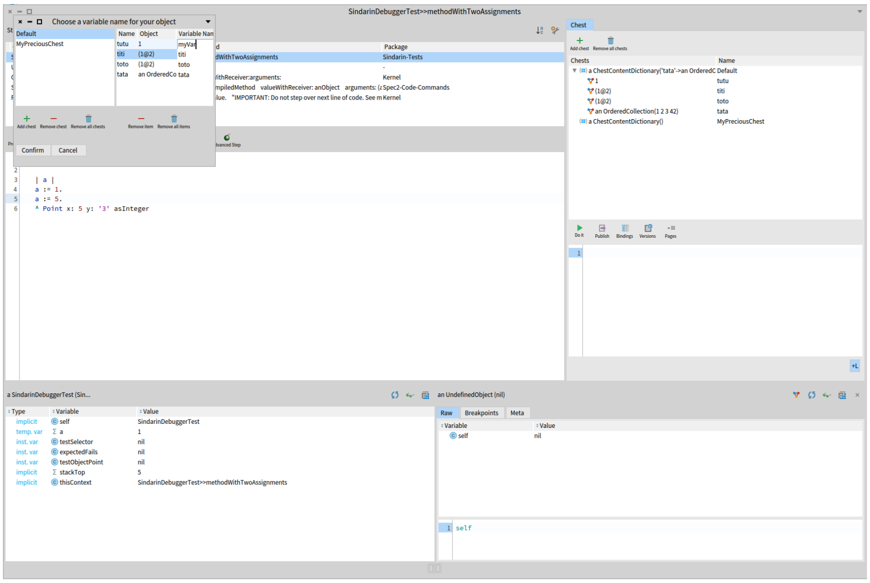
Chests	Name
a ChestContentDictionary('tata'->an OrderedC	Default
1	tutu
(1@2)	titi
(1@2)	toto
an OrderedCollection(1 2 3 42)	tata
a ChestContentDictionary()	MyPreciousChest

a SindarinDebuggerTest (Sin...

Type	Variable	Value
implicit	self	SindarinDebuggerTest
temp. var	a	1
inst. var	testSelector	nil
inst. var	expectedFails	nil
inst. var	testObjectPoint	nil
implicit	stackTop	5
implicit	thisContext	SindarinDebuggerTest>>methodWithTwoAssignments

```
self
```

Figure 1-7 Load object context menu, inside the debugger



Default	Name	Object	Variable Name
MyPreciousChest	tutu	1	myVar
	titi	(1@2)	titi
	toto	(1@2)	toto
	tata	an OrderedCo	tata

```
| a |
a := 1.
a := 5.
^ Point x: 5 y: '3' asInteger
```

Chests	Name
a ChestContentDictionary('tata'->an OrderedC	Default
1	tutu
(1@2)	titi
(1@2)	toto
an OrderedCollection(1 2 3 42)	tata
a ChestContentDictionary()	MyPreciousChest

a SindarinDebuggerTest (Sin...

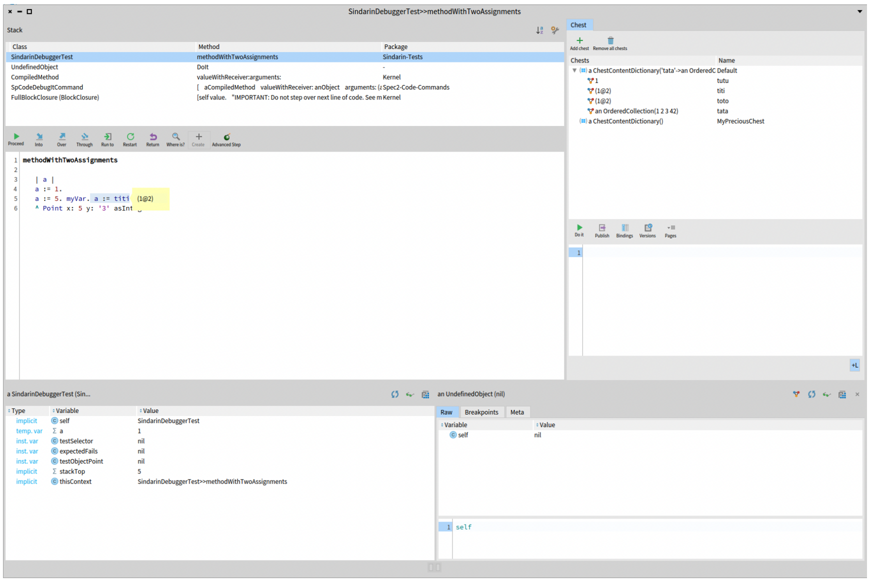
Type	Variable	Value
implicit	self	SindarinDebuggerTest
temp. var	a	1
inst. var	testSelector	nil
inst. var	expectedFails	nil
inst. var	testObjectPoint	nil
implicit	stackTop	5
implicit	thisContext	SindarinDebuggerTest>>methodWithTwoAssignments

an UndefinedObject (nil)

Variable	Value
self	nil

```
self
```

Figure 1-8 Load object popup



Class	Method	Package
SindarinDebuggerTest	methodWithTwoAssignments	Sindarin-Tests
UndefinedObject	DoIt	-
CompiledMethod	valueWithReceiver:arguments:	Kernel
SpCodeDebugItCommand	[aCompiledMethod valueWithReceiver: anObject arguments: (a	Spec2-Code-Commands
FullBlockClosure (BlockClosure)	[self value. "IMPORTANT: Do not step over next line of code. See m	Kernel

```
methodWithTwoAssignments

    | a |
    a := 1.
    a := 5. myVar. a := titi  (1@2)
    ^ Point x: 5 y: '3' asInt
```

Chests	Name
a ChestContentDictionary('tata'->an OrderedC	Default
1	tutu
(1@2)	titi
(1@2)	toto
an OrderedCollection(1 2 3 42)	tata
a ChestContentDictionary()	MyPreciousChest

a SindarinDebuggerTest (Sin...

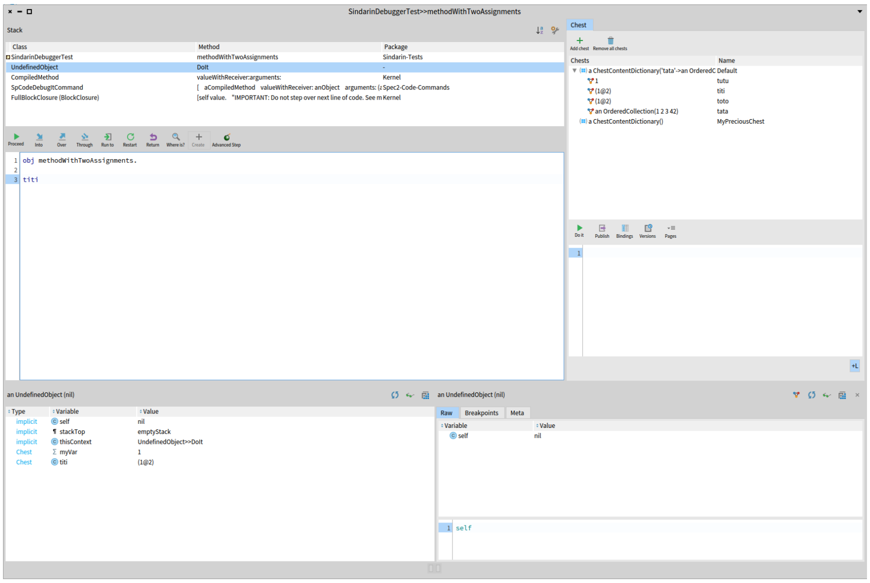
Type	Variable	Value
implicit	self	SindarinDebuggerTest
temp. var	a	1
inst. var	testSelector	nil
inst. var	expectedFails	nil
inst. var	testObjectPoint	nil
implicit	stackTop	5
implicit	thisContext	SindarinDebuggerTest>>methodWithTwoAssignments

an UndefinedObject (nil)

Variable	Value
self	nil

```
self
```

Figure 1-9 Usage of the Chest variable inside the debugger, after having loaded the object



SindarinDebuggerTest>>methodWithTwoAssignments

Stack

Class	Method	Package
SindarinDebuggerTest	methodWithTwoAssignments	Sindarin-Tests
UndefinedObject	DoIt	-
CompiledMethod	valueWithReceiver:arguments:	Kernel
SpCodeDebugItCommand	[aCompiledMethod valueWithReceiver: anObject arguments: (a	Spec2-Code-Commands
FullBlockClosure (BlockClosure)	[self value. "IMPORTANT: Do not step over next line of code. See m	Kernel

```
obj methodWithTwoAssignments.

titi
```

Chest

Chests	Name
a ChestContentDictionary('tata'->an OrderedC	Default
1	tutu
(1@2)	titi
(1@2)	toto
an OrderedCollection(1 2 3 42)	tata
a ChestContentDictionary()	MyPreciousChest

an UndefinedObject (nil)

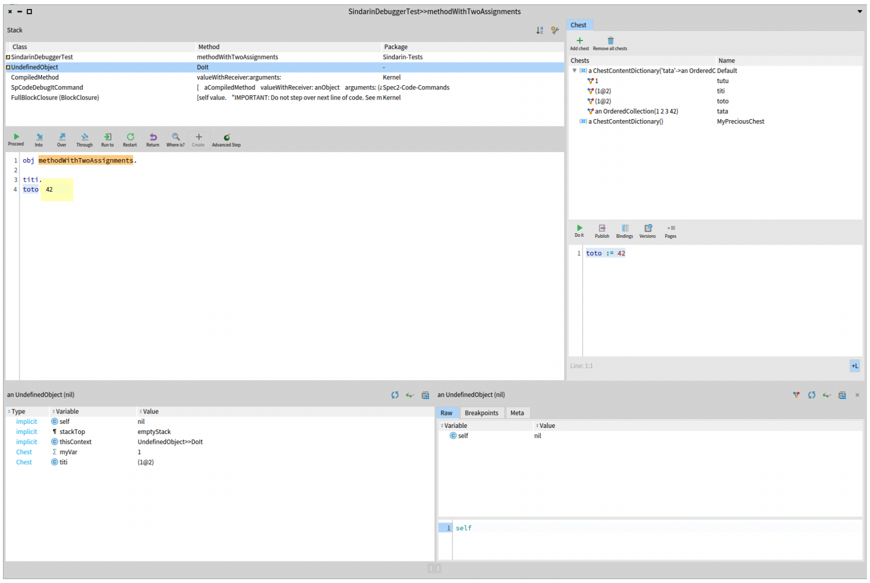
Type	Variable	Value
implicit	self	nil
implicit	stackTop	emptyStack
implicit	thisContext	UndefinedObject>>DoIt
Chest	myVar	1
Chest	titi	(1@2)

an UndefinedObject (nil)

Variable	Value
self	nil

```
self
```

Figure 1-10 Chest variable, in the debugger inspector



SindarinDebuggerTest>>methodWithTwoAssignments

Stack

Class	Method	Package
SindarinDebuggerTest	methodWithTwoAssignments	Sindarin-Tests
UndefinedObject	DoIt	-
CompiledMethod	valueWithReceiver:arguments:	Kernel
SpCodeDebugItCommand	[aCompiledMethod valueWithReceiver: anObject arguments: (a	Spec2-Code-Commands
FullBlockClosure (BlockClosure)	[self value. "IMPORTANT: Do not step over next line of code. See m	Kernel

```
obj methodWithTwoAssignments.

titi.
toto  42
```

Chest

Chests	Name
a ChestContentDictionary('tata'->an OrderedC	Default
1	tutu
(1@2)	titi
(1@2)	toto
an OrderedCollection(1 2 3 42)	tata
a ChestContentDictionary()	MyPreciousChest

```
toto := 42
```

an UndefinedObject (nil)

Type	Variable	Value
implicit	self	nil
implicit	stackTop	emptyStack
implicit	thisContext	UndefinedObject>>DoIt
Chest	myVar	1
Chest	titi	(1@2)

an UndefinedObject (nil)

Variable	Value
self	nil

```
self
```

Figure 1-11 Variable from Chest playground, not visible in the debugger inspector

Inject code to access an object inside a chest, in a playground or in the debugger

To make it easier to access the content of a chest in a playground or in the debugger, it is possible to use the **Paste object from chest >** sub-menu. This sub-menu allows to choose a chest and the key of an object inside the chest.

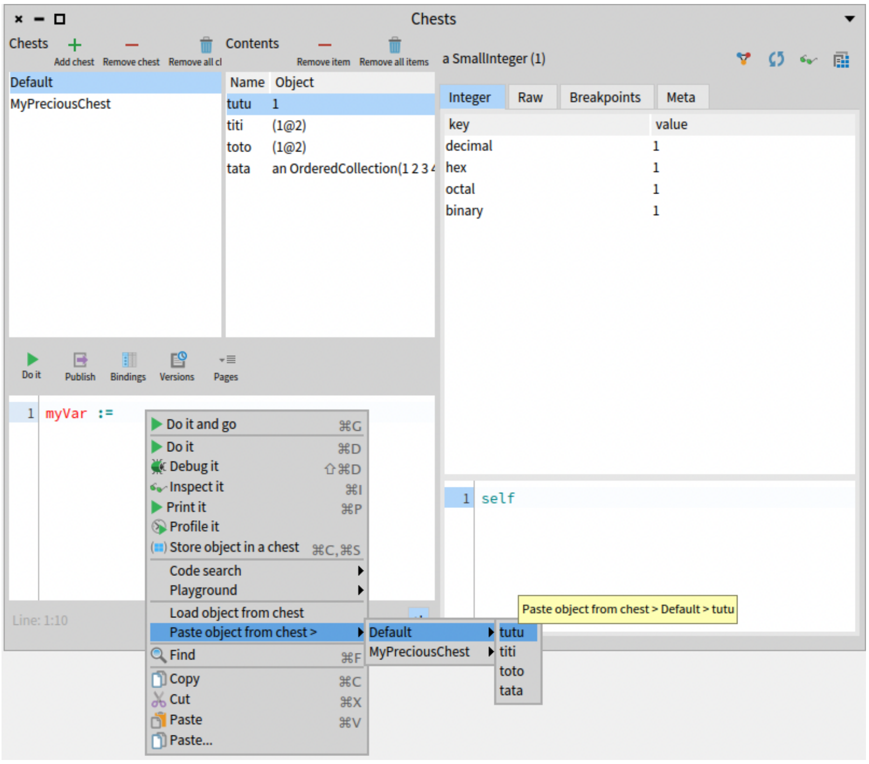


Figure 1-12 Paste object from Chest context menu

When clicking on a variable name, the necessary code to access the corresponding object is pasted where you had put your cursor in the playground/debugger:

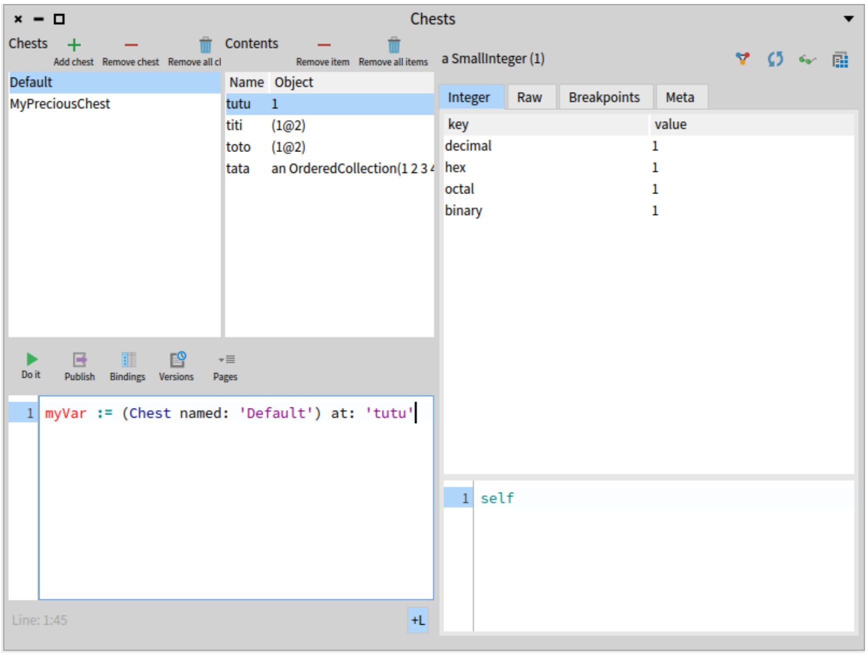
This command allows you to access objects from a chest, without relying on your memory of the `Chest` API.

Simpler code injection in any spec code presenter

The method described above to access objects in chests implies that your objects need to be stored in a chest with a name that you must remember...

Sometimes, you would like to access objects quickly, without needing to give a name to an object that is stored...

That's why it is possible to copy / paste (=inject code to access) to / from a "Clipboard Chest", just as you would do to copy/paste text to/from the clipboard.



Default	
MyPreciousChest	

Name	Object
tutu	1
titi	(1@2)
toto	(1@2)
tata	an OrderedCollection(1 2 3 4

key	value
decimal	1
hex	1
octal	1
binary	1

```
myVar := (Chest named: 'Default') at: 'tutu'
```

```
self
```

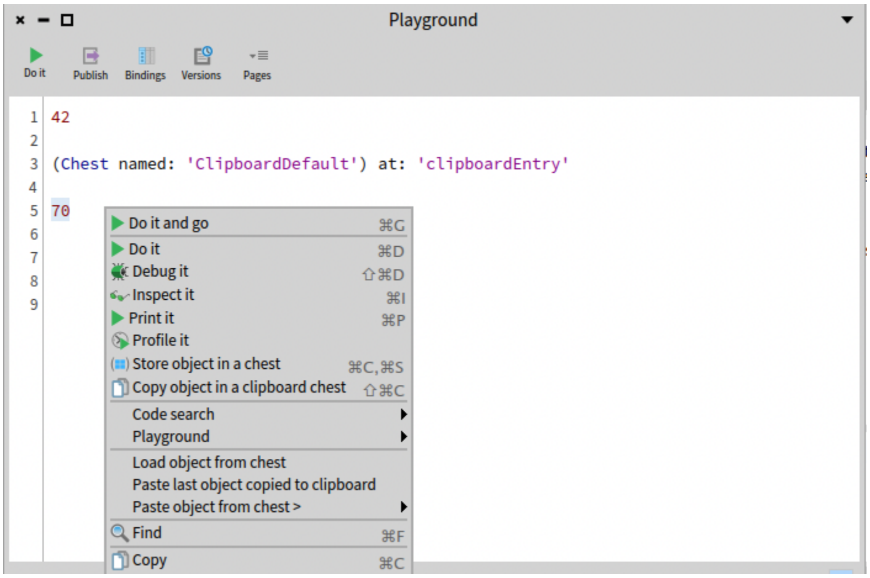
Figure 1-13 Injected code for Chest, after having pasted the object

To do that, you just need to select the expression that you want to evaluate and whose results should be stored in the ”Clipboard Chest”, and then select **Copy object in a clipboard chest** in the context menu:

This will store the result of the expression into the default clipboard chest. Please note, that if you ”copy” another object to the clipboard chest, it will replace the previous copied object. Note also that the clipboard chest is a weak chest, so your object inside the clipboard chest can become `nil` if it gets garbage-collected.

In order to inject the code to access the object, you should then select in the context menu: **Paste last object copied to clipboard**:

And the code to access your object is now pasted:

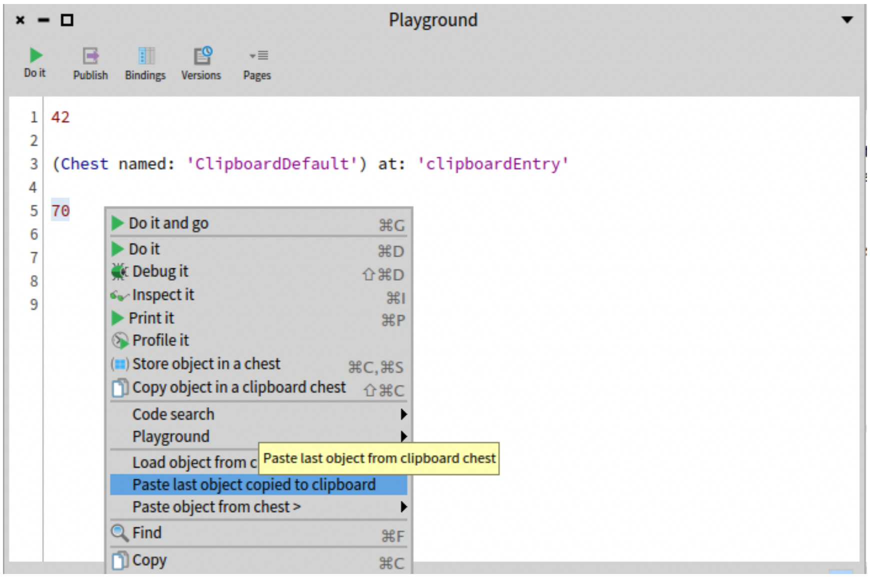


```
42

(Chest named: 'ClipboardDefault') at: 'clipboardEntry'

70
```

Figure 1-14 Copy object to Chest Clipboard context menu



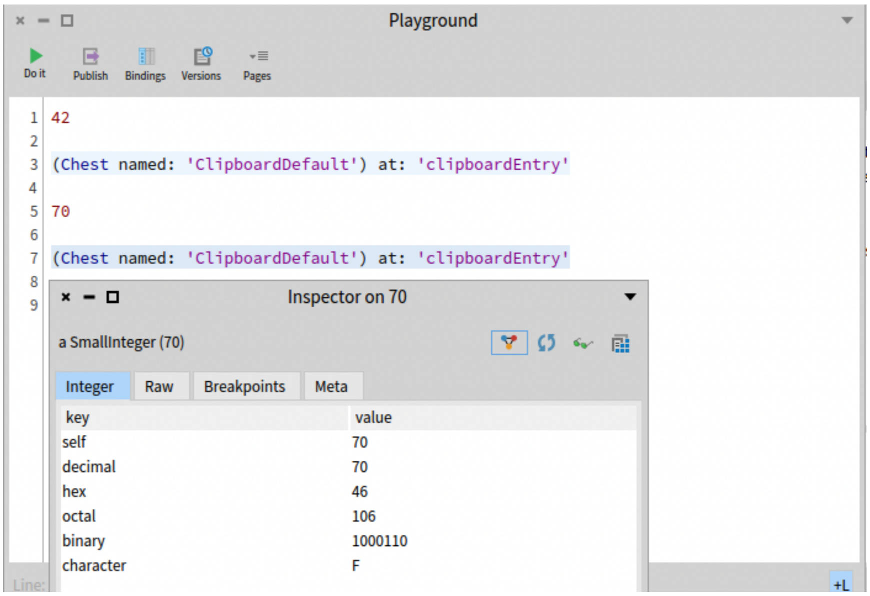
```
42

(Chest named: 'ClipboardDefault') at: 'clipboardEntry'

70
```

Playground
Do it
Publish
Bindings
Versions
Pages
Do it and go ⌘G
Do it ⌘D
Debug it ⇧⌘D
Inspect it ⌘I
Print it ⌘P
Profile it
Store object in a chest ⌘C, ⌘S
Copy object in a clipboard chest ⇧⌘C
Code search
Playground
Load object from c
Paste last object from clipboard chest
Paste last object copied to clipboard
Paste object from chest >
Find ⌘F
Copy ⌘C

Figure 1-15 Paste last object from Chest Clipboard context menu



```
42

(Chest named: 'ClipboardDefault') at: 'clipboardEntry'

70

(Chest named: 'ClipboardDefault') at: 'clipboardEntry'
```

a SmallInteger (70)

key	value
self	70
decimal	70
hex	46
octal	106
binary	1000110
character	F

Figure 1-16 Injected code for Chest, after having pasted the object from Chest Clipboard

CHAPTER 2

API Description

In this chapter, we detail the API of Chest and give examples of its usage.

2.1 Chest class-side API

In this section, we describe the API that can be used on the class `Chest`.

How to create instances

- `Chest class>>#new`

creates a chest with a default name that is in the form of `Chest_autoIncrementedNumber`.

In the examples below, if no other chest has been created before, the names of the created chests are respectively ”Chest*1” and ”Chest*2”:

```
Chest new. "Chest_1"
Chest new. "Chest_2"
```

- `Chest class>>#newNamed:`

creates a chest with the name given in parameter if no other chest is already named so, else an exception `ChestKeyAlreadyInUseError` is raised.

For example, if no other chest is already called ”toto”, the piece of code below creates a chest that is named as ”toto”:

```
Chest newNamed: 'toto'. "its name is 'toto'"
```

On the contrary, if another chest is already called ”toto”, the same piece of code would raise a `ChestKeyAlreadyInUseError` because two chests cannot have the same name:

```
Chest newNamed: 'toto'. "ChestKeyAlreadyInUseError as a chest named
    'toto' already exists"
```

Accessors

- `Chest class>>#allChests`

returns an ordered collection that contains all chest instances.

If we suppose that there are no other chests than the ones created above, the piece of code returns a collection with two chests: the default chest and the chest named ”toto”:

```
Chest allChests "{DefaultChest. totoChest}"
```

- `Chest class>>#chestDictionary`

returns a dictionary containing all chests with their name as key.

If we suppose that there are no other chests than the ones created above, the piece of code returns a dictionary with two chests: the default chest, with ’Default’ as a key, and the chest named ”toto”, with ’toto’ as a key:

```
Chest chestDictionary "{'Default' -> DefaultChest. 'toto' ->
    totoChest}"
```

- `Chest class>>#named:`

returns the chest that is named as the string in argument if it exists, else raises an exception `KeyNotFound`.

The expression below can be used to get the chest named ’toto’, if it exists:

```
Chest named: 'toto' "chest named 'toto'"
```

However, no chest is named ’titi’, the expression below would raise a `KeyNotFound`:

```
Chest named: 'titi' "KeyNotFound"
```

- `Chest class>>#defaultInstance` (or `Chest class>>#default`).

You can use most of Chest’s instance-side API on the class `Chest` itself. These methods returns the default chest that is used when you use `Chest`’s instance-side API directly on `Chest class`. If the default chest doesn’t already exist, calling these methods lazily create it.

For example, the expression below returns `true`:

```
Chest defaultInstance == Chest default "true"
```

- `Chest class>>#announcer`

helper method that returns the unique instance of `ChestAnnouncer`, which propagates changes related to chests to any subscriber.

The example belows subscribes `self` to the `ChestAnnouncer` singleton to 3 different events; when a new chest chest has been created, when the content of a chest has changed and when a chest has been removed:

```
Chest announcer weak when: ChestCreated send: #eventNewChest: to:
    self;
                        when: ChestUpdated send:
    #eventContentOfChestUpdated: to: self.
                        when: ChestRemoved send: #eventChestRemoved:
```

```
    to: self.
```

When the event happens, the methods that are called (in the example: `#eventNewChest:`, `#eventContentOfChestUpdated:` and `#eventChestRemoved:`) take the related event as an argument.

- `Chest class>>#weak`

returns the class `WeakChest`, subclass of `Chest`. You can use the same API on this class as you would on `Chest class`, in order to create or access chests that hold weak references to objects. This means, that storing your objects in a weak chest doesn't prevent them from being garbage-collected.

For example, the expression below creates a new weak chest named as "wtiti", if it doesn't already exist:

```
Chest weak newNamed: 'wtiti' "weak chest named as 'wtiti'"
```

How to perform actions

- `Chest class>>#inChest:at:put:`

puts the object in third argument with the name given in second argument in the chest named as first argument. This chest is lazily created if it doesn't exist yet.

In the example below, the object `42` is stored as "toto" in the chest named "toto":

```
(Chest named: 'toto') at: 'toto' put: 42. "stores 42 as 'toto' in
    chest named as 'toto'.
```

If the chest "toto" didn't exist, it would be created automatically. *Also, please note that a chest cannot contain the same object twice. So, if the chest named as "toto" already contained the object `42`, it would simply be renamed as "toto" in this chest.*

- `Chest class>>#unsubscribe:`

helper method that unsubscribes its argument from the unique instance of `ChestAnnouncer`.

For example, the expression below unsubscribes `self` from the `ChestAnnouncer` singleton:

```
Chest unsubscribe: self
```

2.2 Chest instance-side API

In this section, we describe the API that can be used on an instance of `Chest`. Methods marked by the symbol `(*)` can also be used on the class `Chest` itself. In this case, it is equivalent to using the API on the default instance of `Chest`.

Accessors

- `Chest>>#name:`

returns the receiver chest's name.

For example, the expression below evaluates to `true`:

```
(Chest named: 'toto') name = 'toto' "true"
```

- `Chest>>#contents:` (*)

returns a copy of the receiver chest's contents, as a dictionary that contains all objects in the chest with their name as key.

For example, evaluating the piece of code below returns a dictionary that associates 42 to ”toto” and 144 to ”titi”:

```
| c |
c := Chest new.
Chest inChest: c name at: 'toto' put: 42.
Chest inChest: c name at: 'titi' put: 144.
c contents "{'toto' -> 42. 'titi' -> 144}"
```

- `Chest>>#at: :` (*)

returns the object, contained in the receiver chest, whose name is the string in argument if it exists, else an exception `KeyNotFound` is raised.

For example, if the chest named as ”toto” stores 42 as ”titi”, then the expression below returns 42:

```
(Chest named: 'toto') at: 'titi'. "42"
```

On the contrary, if no object is named as ”titi” in the chest named as ”toto”, then the same expression raises a `KeyNotFound`:

```
(Chest named: 'toto') at: 'titi' "KeyNotFound"
```

How to perform actions

- `Chest>>#add:` (*)

adds the object in argument to the receiver chest, with a default name that is in the form of `chestName_autoIncrementedNumber`.

In the piece of code below, we add 42 then 144 to a new chest named ”MyChest”. Thus, 42 is stored with the key ”MyChest*1” and 144 is stored with the key ”MyChest*2”:

```
| c |
c := Chest newNamed: 'MyChest'.
c add: 42. "MyChest_1 -> 42"
c add: 144 "MyChest_2 -> 144"
```

- `Chest>>#at:put:` (*)

adds the object in second argument to the receiver chest with the name in first argument if no other object is already named so, else an exception `ChestKeyAlreadyInUseError` is raised.

In the example below, in a new chest, 42 is stored as ”toto”. Then, trying to store 144 as ”toto” raises a `ChestKeyAlreadyInUseError` because two objects cannot have the same name:

```
| c |
c := Chest new.
c at: 'toto' put: 42. "toto -> 42"
c at: 'toto' put: 144 "ChestKeyAlreadyInUseError"
```

Also, please note that a chest cannot contain the same object twice. So, if the chest named as "toto" already contained the object 42, it would simply be renamed as "toto" in this chest.

- `Chest>>#at:put:ifPresent:` (*)

adds the object in second argument to the receiver chest with the name in first argument if no other object is already named so, else the block in third argument is evaluated with zero argument.

In the exemple below, in a new chest, `42` is stored as "toto". Then, we try to store `144` as "toto" and if the key is already use, we try to store it as "titi" instead. As the key "toto" is already used by `42`, `144` is stored as "titi":

```
| c |
c := Chest new.
c at: 'toto' put: 42 ifPresent: [Chest at: 'titi' put: 42]. "toto ->
    42"
c at: 'toto' put: 144 ifPresent: [ Chest at: 'titi' put: 144 ] "titi
    -> 144"
```

- `Chest>>#remove:` (*)

removes the object in argument from the receiver chest if it is there, else an exception `KeyNotFound` is raised.

For example, if the chest named "toto" contains the object `42`, then the following code snippet successfully removes `42` from this chest. Then it tries to remove `42` from this chest a second time, in which case a `KeyNotFound` is raised because the object is not there anymore:

```
| c |
c := Chest named: 'toto'.
c contents includes: 42 "true"

c remove: 42. "42 is removed from c"

c contents includes: 42. "false"

c remove: 42 "KeyNotFound"
```

- `Chest>>#removeObjectNamed:` (*)

removes from the receiver chest the object named as the argument if the key exists, else an exception `ObjectNotInChestError` is raised.

For example, if the chest named "toto" stores an object named as "tata", then the following code snippet removes the object named as "tata" from this chest. Then it tries to remove a second time the object named as "tata" from this chest, in which case an `ObjectNotInChestError` is raised because the key isn't used anymore.

```
| c |
c := Chest named: 'toto'.
c contents includesKey: 'tata' "true"

c removeObjectNamed: 'tata'. "obj named 'tata' is removed from c"

c contents includesKey: 'tata'. "false"

c removeObjectNamed: 'tata' "ObjectNotInChestError"
```

- `Chest>>#empty`:

removes the entire contents of the receiver chest:

```
| c |
c := Chest new.
c at: 'toto' put: 42.
c at: 'titi' put 144.

c contents isEmpty. "false"

c empty.

c contentns isEmpty "true"
```

- `Chest>>#remove`

completely deletes the receiver chest from the list of existing chests. It cannot be accessed afterwards:

```
| c cname |
c := Chest new.
cname := c name.
Chest chestDictionary includesKey: cname. "true"

c remove.

Chest chestDictionary includesKey: cname. "false"
```

- `Chest>>#name:`

renames the receiver chest as the string in argument if no other chest is already named so, else an exception `ChestKeyAlreadyInUseError` is raised.

For example, if there exist only two chests named ”toto” and ”titi”, then it is possible to rename ”toto” into ”tata” but not into ”titi”, in which case a `ChestKeyAlreadyInUseError` is raised:

```
| toto titi |
toto := Chest named: 'toto'.
titi := Chest named: 'titi'.

toto name: 'tata'.
toto name "tata".
toto name: 'titi'. "ChestKeyAlreadyInUseError"
toto name "tata"
```

- `Chest>>#renameObject:into::`

inside the receiver chest, renames the object in first argument into the string in second argument if the object is in the chest, else an exception `ObjectNotInChestError` is raised, and if no other object is already named so else an exception `ChestKeyAlreadyInUseError` is raised.

If the chest named as "toto" contains only `42` as "titi" and `144` as "tata", then the piece of code below renames `42` to "tutu":

```
(Chest named: 'toto') at: 'titi'. "42"
(Chest named: 'toto') renameObject: 42 into: 'tutu'.
(Chest named: 'toto') at: 'tutu'. "42"
(Chest named: 'toto') at: 'titi' "KeyNotFound"
```

Under the same circumstances, the following code snippet raises an `ObjectNotInChestError` because it tries to rename `666` that is not in the chest named as "toto":

```
(Chest named: 'toto) contents includes: 666 "false"
(Chest named: 'toto') renameObject: 666 into 'anyValidName'
    "ObjectNotInChestError"
```

Finally, trying to rename `144` into "tutu" will raise a `ChestKeyAlreadyInUseError` because `42` is already named so:

```
(Chest named: 'toto') renameObject: 144 into: 'tutu'
    "ChestKeyAlreadyInUseError"
```

- `Chest>>#inspectAt::`

inspect the object named as the argument in the receiver chest:

For example, the following expression opens an inspector on the object stored as "titi" in the chest named "toto":

```
(Chest named: 'toto') inspectAt: 'titi'
```