Extracted from:

Programming Ruby

The Pragmatic Programmers' Guide

Second Edition

Class Hash < Object

A Hash is a collection of key/value pairs. It is similar to an Array, except that indexing is done via arbitrary keys of any object type, not an integer index. The order in which keys and/or values are returned by the various iterators over hash contents may seem arbitrary and will generally not be in insertion order.

Hashes have a *default value*. This value is returned when an attempt is made to access keys that do not exist in the hash. By default, this value is nil.

Mixes in

Enumerable:

all?, any?, collect, detect, each_with_index, entries, find, find_all, grep, include?, inject, map, max, member?, min, partition, reject, select, sort, sort_by, to_a, zip

Class methods

[ ] Hash[⟨ *key* => *value* ⟩*] → *hsh*

Creates a new hash populated with the given objects. Equivalent to creating a hash using the literal { *key*=>*value*, ... }. Keys and values occur in pairs, so there must be an even number of arguments.

```
Hash["a", 100, "b", 200]       →   {"a"=>100, "b"=>200}
Hash["a" => 100, "b" => 200]   →   {"a"=>100, "b"=>200}
{ "a" => 100, "b" => 200 }     →   {"a"=>100, "b"=>200}
```

new Hash.new → *hsh*
Hash.new(*obj*) → *hsh*
Hash.new {| *hash*, *key* | *block* } → *hsh*

1.8 Returns a new, empty hash. If this hash is subsequently accessed by a key that doesn't correspond to a hash entry, the value returned depends on the style of new used to create the hash. In the first form, the access returns nil. If *obj* is specified, this single object will be used for all *default values*. If a block is specified, it will be called with the hash object and the key, and it should return the default value. It is the block's responsibility to store the value in the hash if required.

```
h = Hash.new("Go Fish")
h["a"] = 100
h["b"] = 200
h["a"]              →   100
h["c"]              →   "Go Fish"
# The following alters the single default object
h["c"].upcase!      →   "GO FISH"
h["d"]              →   "GO FISH"
h.keys              →   ["a", "b"]
```

```
# While this creates a new default object each time
h = Hash.new {|hash, key| hash[key] = "Go Fish: #{key}" }
h["c"]            →   "Go Fish: c"
h["c"].upcase!    →   "GO FISH: C"
h["d"]            →   "Go Fish: d"
h.keys            →   ["c", "d"]
```

Instance methods

==

hsh == *obj* → `true` or `false`

Equality—Two hashes are equal if they have the same default value, they contain the same number of keys, and the value corresponding to each key in the first hash is equal (using ==) to the value for the same key in the second. If *obj* is not a hash, attempt to convert it using `to_hash` and return *obj* == *hsh*.

```
h1 = { "a" => 1, "c" => 2 }
h2 = { 7 => 35, "c" => 2, "a" => 1 }
h3 = { "a" => 1, "c" => 2, 7 => 35 }
h4 = { "a" => 1, "d" => 2, "f" => 35 }
h1 == h2    →   false
h2 == h3    →   true
h3 == h4    →   false
```

[ ]

hsh[*key*] → *value*

Element Reference—Retrieves the *value* stored for *key*. If not found, returns the default value (see `Hash.new` for details).

```
h = { "a" => 100, "b" => 200 }
h["a"]    →   100
h["c"]    →   nil
```

[ ]=

hsh[*key*] = *value* → *value*

Element Assignment—Associates the value given by *value* with the key given by *key*. *key* should not have its value changed while it is in use as a key (a `String` passed as a key will be duplicated and frozen).

```
h = { "a" => 100, "b" => 200 }
h["a"] = 9
h["c"] = 4
h    →   {"a"=>9, "b"=>200, "c"=>4}
```

clear

hsh.clear → *hsh*

Removes all key/value pairs from *hsh*.

```
h = { "a" => 100, "b" => 200 }    →   {"a"=>100, "b"=>200}
h.clear                           →   {}
```

default *hsh*.default(*key*=nil) → *obj*

1.8

Returns the default value, the value that would be returned by *hsh*[*key*] if *key* did not exist in *hsh*. See also `Hash.new` and `Hash#default=`.

```
h = Hash.new                          →   {}
h.default                             →   nil
h.default(2)                          →   nil

h = Hash.new("cat")                   →   {}
h.default                             →   "cat"
h.default(2)                          →   "cat"

h = Hash.new {|h,k| h[k] = k.to_i*10}  →   {}
h.default                             →   0
h.default(2)                          →   20
```

default= *hsh*.default = *obj* → *hsh*

Sets the default value, the value returned for a key that does not exist in the hash. It is not possible to set the a default to a `Proc` that will be executed on each key lookup.

```
h = { "a" => 100, "b" => 200 }
h.default = "Go fish"
h["a"]     →   100
h["z"]     →   "Go fish"
# This doesn't do what you might hope...
h.default = proc do |hash, key|
  hash[key] = key + key
end
h[2]       →   #<Proc:0x001c94e0@-:6>
h["cat"]   →   #<Proc:0x001c94e0@-:6>
```

default_proc *hsh*.default_proc → *obj* or nil

1.8

If `Hash.new` was invoked with a block, return that block; otherwise return `nil`.

```
h = Hash.new {|h,k| h[k] = k*k }   →   {}
p = h.default_proc                 →   #<Proc:0x001c997c@-:1>
a = []                             →   []
p.call(a, 2)
a                                  →   [nil, nil, 4]
```

delete *hsh*.delete(*key*) → *value*
hsh.delete(*key*) {| *key* | *block* } → *value*

1.8

Deletes from *hsh* the entry whose key is to *key*, returning the corresponding value. If the key is not found, returns `nil`. If the optional code block is given and the key is not found, pass it the key and return the result of *block*.

```
h = { "a" => 100, "b" => 200 }
h.delete("a")                              →  100
h.delete("z")                              →  nil
h.delete("z") {|el| "#{el} not found" }    →  "z not found"
```

delete_if

hsh.delete_if {| *key*, *value* | *block* } → *hsh*

Deletes every key/value pair from *hsh* for which *block* is true.

```
h = { "a" => 100, "b" => 200, "c" => 300 }
h.delete_if {|key, value| key >= "b" }   →   {"a"=>100}
```

each

hsh.each {| *key*, *value* | *block* } → *hsh*

Calls *block* once for each key in *hsh*, passing the key and value as parameters.

```
h = { "a" => 100, "b" => 200 }
h.each {|key, value| puts "#{key} is #{value}" }
```

produces:

```
a is 100
b is 200
```

each_key

hsh.each_key {| *key* | *block* } → *hsh*

Calls *block* once for each key in *hsh*, passing the key as a parameter.

```
h = { "a" => 100, "b" => 200 }
h.each_key {|key| puts key }
```

produces:

```
a
b
```

each_pair

hsh.each_pair {| *key*, *value* | *block* } → *hsh*

Synonym for `Hash#each`.

each_value

hsh.each_value {| *value* | *block* } → *hsh*

Calls *block* once for each key in *hsh*, passing the value as a parameter.

```
h = { "a" => 100, "b" => 200 }
h.each_value {|value| puts value }
```

produces:

```
100
200
```

empty?

hsh.empty? → `true` or `false`

Returns `true` if *hsh* contains no key/value pairs.

```
{}.empty?   →   true
```

fetch

hsh.fetch(*key* ⟨ , *default* ⟩) → *obj*
hsh.fetch(*key*) {| *key* | *block* } → *obj*

Returns a value from the hash for the given key. If the key can't be found, several options exist: With no other arguments, it will raise an `IndexError` exception; if *default* is given, then that will be returned; if the optional code block is specified, then that will be run and its result returned. `fetch` does not evaluate any default values supplied when the hash was created—it only looks for keys in the hash.

```
h = { "a" => 100, "b" => 200 }
h.fetch("a")                            →   100
h.fetch("z", "go fish")                 →   "go fish"
h.fetch("z") {|el| "go fish, #{el}"}    →   "go fish, z"
```

The following example shows that an exception is raised if the key is not found and a default value is not supplied.

```
h = { "a" => 100, "b" => 200 }
h.fetch("z")
```

produces:

```
prog.rb:2:in `fetch': key not found (IndexError)
from prog.rb:2
```

has_key?

hsh.has_key?(*key*) → `true` or `false`

Returns `true` if the given key is present in *hsh*.

```
h = { "a" => 100, "b" => 200 }
h.has_key?("a")   →   true
h.has_key?("z")   →   false
```

has_value?

hsh.has_value?(*value*) → `true` or `false`

Returns `true` if the given value is present for some key in *hsh*.

```
h = { "a" => 100, "b" => 200 }
h.has_value?(100)   →   true
h.has_value?(999)   →   false
```

include?

hsh.include?(*key*) → `true` or `false`

Synonym for `Hash#has_key?`.

index

hsh.index(*value*) → *key*

Searches the hash for an entry whose value == *value*, returning the corresponding key. If multiple entries has this value, the key returned will be that on one of the entries. If not found, returns `nil`.

```
h = { "a" => 100, "b" => 200 }
h.index(200)   →   "b"
h.index(999)   →   nil
```

indexes *hsh*.indexes(⟨ *key* ⟩$^+$) → *array*

1.8 Deprecated in favor of `Hash#values_at`.

indices *hsh*.indices(⟨ *key* ⟩$^+$) → *array*

1.8 Deprecated in favor of `Hash#values_at`.

invert *hsh*.invert → *other_hash*

Returns a new hash created by using *hsh*'s values as keys, and the keys as values. If *hsh* has duplicate values, the result will contain only one of them as a key—which one is not predictable.

```
h = { "n" => 100, "m" => 100, "y" => 300, "d" => 200, "a" => 0 }
h.invert   →   {0=>"a", 100=>"n", 200=>"d", 300=>"y"}
```

key? *hsh*.key?(*key*) → `true` or `false`

Synonym for `Hash#has_key?`.

keys *hsh*.keys → *array*

Returns a new array populated with the keys from this hash. See also `Hash#values`.

```
h = { "a" => 100, "b" => 200, "c" => 300, "d" => 400 }
h.keys   →   ["a", "b", "c", "d"]
```

length *hsh*.length → *fixnum*

Returns the number of key/value pairs in the hash.

```
h = { "d" => 100, "a" => 200, "v" => 300, "e" => 400 }
h.length        →   4
h.delete("a")   →   200
h.length        →   3
```

member? *hsh*.member?(*key*) → `true` or `false`

Synonym for `Hash#has_key?`.

merge *hsh*.merge(*other_hash*) → *result_hash*
hsh.merge(*other_hash*) {| key, old_val, new_val | *block* } → *result_hash*

1.8 Returns a new hash containing the contents of *other_hash* and the contents of *hsh*. With no block parameter, overwrites entries in *hsh* with duplicate keys with those from *other_hash*. If a block is specified, it is called with each duplicate key and the values from the two hashes. The value returned by the block is stored in the new hash.

```
h1 = { "a" => 100, "b" => 200 }
h2 = { "b" => 254, "c" => 300 }
h1.merge(h2)                → {"a"=>100, "b"=>254, "c"=>300}
h1.merge(h2) {|k,o,n| o}    → {"a"=>100, "b"=>200, "c"=>300}
h1                          → {"a"=>100, "b"=>200}
```

merge! *hsh*.merge!(*other_hash*) → *hsh*
hsh.merge!(*other_hash*) {| key, old_val, new_val | *block* } → *hsh*

1.8

Adds the contents of *other_hash* to *hsh*, overwriting entries with duplicate keys with those from *other_hash*.

```
h1 = { "a" => 100, "b" => 200 }
h2 = { "b" => 254, "c" => 300 }
h1.merge!(h2)                → {"a"=>100, "b"=>254, "c"=>300}
h1 = { "a" => 100, "b" => 200 }
h1.merge!(h2) {|k,o,n| o}    → {"a"=>100, "b"=>200, "c"=>300}
h1                           → {"a"=>100, "b"=>200, "c"=>300}
```

rehash *hsh*.rehash → *hsh*

Rebuilds the hash based on the current hash values for each key. If values of key objects have changed since they were inserted, this method will reindex *hsh*. If `Hash#rehash` is called while an iterator is traversing the hash, an `IndexError` will be raised in the iterator.

```
a = [ "a", "b" ]
c = [ "c", "d" ]
h = { a => 100, c => 300 }
h[a]       → 100
a[0] = "z"
h[a]       → nil
h.rehash   → {["z", "b"]=>100, ["c", "d"]=>300}
h[a]       → 100
```

reject *hsh*.reject {| *key*, *value* | *block* } → *hash*

Same as `Hash#delete_if`, but works on (and returns) a copy of *hsh*. Equivalent to *hsh*`.dup.delete_if`.

reject! *hsh*.reject! {| *key*, *value* | *block* } → *hsh* or `nil`

Equivalent to `Hash#delete_if`, but returns `nil` if no changes were made.

replace *hsh*.replace(*other_hash*) → *hsh*

Replaces the contents of *hsh* with the contents of *other_hash*.

```
h = { "a" => 100, "b" => 200 }
h.replace({ "c" => 300, "d" => 400 })   → {"c"=>300, "d"=>400}
```

select

hsh.select {| *key, value* | *block* } → *array*

Returns a new array consisting of `[key, value]` pairs for which the block returns true. Also see `Hash#values_at`.

```
h = { "a" => 100, "b" => 200, "c" => 300 }
h.select {|k,v| k > "a"}   →   [["b", 200], ["c", 300]]
h.select {|k,v| v < 200}   →   [["a", 100]]
```

shift

1.8

hsh.shift → *array* or `nil`

Removes a key/value pair from *hsh* and returns it as the two-item array `[` *key, value* `]`. If the hash is empty, returns the default value, calls the default proc (with a key value of `nil`), or returns `nil`.

```
h = { 1 => "a", 2 => "b", 3 => "c" }
h.shift    →   [1, "a"]
h          →   {2=>"b", 3=>"c"}
```

size

hsh.size → *fixnum*

Synonym for `Hash#length`.

sort

hsh.sort → *array*
hsh.sort {| *a, b* | *block* } → *array*

Converts *hsh* to a nested array of `[` *key, value* `]` arrays and sorts it, using `Array#sort`.

```
h = { "a" => 20, "b" => 30, "c" => 10  }
h.sort                       →   [["a", 20], ["b", 30], ["c", 10]]
h.sort {|a,b| a[1]<=>b[1]}   →   [["c", 10], ["a", 20], ["b", 30]]
```

store

hsh.store(*key, value*) → *value*

Synonym for Element Assignment (`Hash#[]=`).

to_a

hsh.to_a → *array*

Converts *hsh* to a nested array of `[` *key, value* `]` arrays.

```
h = { "c" => 300, "a" => 100, "d" => 400, "c" => 300  }
h.to_a   →   [["a", 100], ["c", 300], ["d", 400]]
```

to_hash

hsh.to_hash → *hsh*

See page 356.

to_s

hsh.to_s → *string*

Converts *hsh* to a string by converting the hash to an array of `[` *key, value* `]` pairs and then converting that array to a string using `Array#join` with the default separator.

```
h = { "c" => 300, "a" => 100, "d" => 400, "c" => 300  }
h.to_s   →   "a100c300d400"
```

Hash

update *hsh*.update(*other_hash*) → *hsh*
hsh.update(*other_hash*) {| key, old_val, new_val | *block* } → *hsh*

1.8 Synonym for `Hash#merge!`.

value? *hsh*.value?(*value*) → `true` or `false`

Synonym for `Hash#has_value?`.

values *hsh*.values → *array*

Returns an array populated with the values from *hsh*. See also `Hash#keys`.

```
h = { "a" => 100, "b" => 200, "c" => 300 }
h.values   →   [100, 200, 300]
```

values_at *hsh*.values_at(⟨ *key* ⟩$^+$) → *array*

1.8 Returns an array consisting of values for the given key(s). Will insert the *default value* for keys that are not found.

```
h = { "a" => 100, "b" => 200, "c" => 300 }
h.values_at("a", "c")        →   [100, 300]
h.values_at("a", "c", "z")   →   [100, 300, nil]
h.default = "cat"
h.values_at("a", "c", "z")   →   [100, 300, "cat"]
```