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

```ruby
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

**==**

```ruby
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`.

```ruby
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
```

**[]**

```ruby
hsh[ key ] → value
```

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

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

**[]=**

```ruby
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).

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

**clear**

```ruby
hsh.clear → hsh
```

Removes all key/value pairs from `hsh`.

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

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       →  {}
  h.default(2)    →  "cat"

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

**default=**  

1.8

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**  

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:0x001c997c08@­:6>
a = []                              →  []
p.call(a, 2)                        →  [nil, nil, 4]
```

**delete**  

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`.

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

```ruby
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.

```ruby
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.

```ruby
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.

```ruby
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.

```ruby
{}.empty? → true
```
### fetch

$hsh.fetch(\text{key} \langle, \text{default} \rangle) \rightarrow \text{obj}$

$hsh.fetch(\text{key} \mid \langle\text{key} | \text{block} \rangle \mid \rightarrow \text{obj}$

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.

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

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

**produces:**

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

### has_key?

$hsh.has_key?(\text{key}) \rightarrow \text{true or false}$

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

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

### has_value?

$hsh.has_value?(\text{value}) \rightarrow \text{true or false}$

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

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

### include?

$hsh.include?(\text{key}) \rightarrow \text{true or false}$

Synonym for `Hash#has_key?`.

### index

$hsh.index(\text{value}) \rightarrow \text{key}$

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`.

```ruby
h = { "a" => 100, "b" => 200 }
h.index(200) \rightarrow "b"
h.index(999) \rightarrow nil
```
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hsh.indexes( key )</code></td>
<td>Returns an array of the keys in the hash.</td>
</tr>
<tr>
<td><code>hsh.indexes( key )</code></td>
<td>Deprecated in favor of <code>Hash#values_at</code>.</td>
</tr>
<tr>
<td><code>hsh.indices( key )</code></td>
<td>Returns an array of the keys in the hash.</td>
</tr>
<tr>
<td><code>hsh.indices( key )</code></td>
<td>Deprecated in favor of <code>Hash#values_at</code>.</td>
</tr>
<tr>
<td><code>hsh.invert</code></td>
<td>Returns a new hash created by using <code>hsh</code>'s values as keys, and the keys as values. If <code>hsh</code> has duplicate values, the result will contain only one of them as a key—which one is not predictable.</td>
</tr>
<tr>
<td></td>
<td><code>h = { &quot;n&quot; =&gt; 100, &quot;m&quot; =&gt; 100, &quot;y&quot; =&gt; 300, &quot;d&quot; =&gt; 200, &quot;a&quot; =&gt; 0 }</code></td>
</tr>
<tr>
<td></td>
<td><code>h.invert</code> =&gt; <code>{0=&gt;&quot;a&quot;, 100=&gt;&quot;n&quot;, 200=&gt;&quot;d&quot;, 300=&gt;&quot;y&quot;}</code></td>
</tr>
<tr>
<td><code>hsh.key?( key )</code></td>
<td>Synonym for <code>Hash#has_key?</code>.</td>
</tr>
<tr>
<td><code>hsh.keys</code></td>
<td>Returns a new array populated with the keys from this hash. See also <code>Hash#values</code>.</td>
</tr>
<tr>
<td></td>
<td><code>h = { &quot;a&quot; =&gt; 100, &quot;b&quot; =&gt; 200, &quot;c&quot; =&gt; 300, &quot;d&quot; =&gt; 400 }</code></td>
</tr>
<tr>
<td></td>
<td><code>h.keys</code> =&gt; <code>&quot;a&quot;, &quot;b&quot;, &quot;c&quot;, &quot;d&quot;</code></td>
</tr>
<tr>
<td><code>hsh.length</code></td>
<td>Returns the number of key/value pairs in the hash.</td>
</tr>
<tr>
<td></td>
<td><code>h = { &quot;d&quot; =&gt; 100, &quot;a&quot; =&gt; 200, &quot;v&quot; =&gt; 300, &quot;e&quot; =&gt; 400 }</code></td>
</tr>
<tr>
<td></td>
<td><code>h.length</code> =&gt; <code>4</code></td>
</tr>
<tr>
<td></td>
<td><code>h.delete(&quot;a&quot;)</code> =&gt; <code>200</code></td>
</tr>
<tr>
<td></td>
<td><code>h.length</code> =&gt; <code>3</code></td>
</tr>
<tr>
<td><code>hsh.member?( key )</code></td>
<td>Synonym for <code>Hash#has_key?</code>.</td>
</tr>
<tr>
<td><code>hsh.merge( other_hash )</code></td>
<td>Returns a new hash containing the contents of <code>other_hash</code> and the contents of <code>hsh</code>. With no block parameter, overwrites entries in <code>hsh</code> with duplicate keys with those from <code>other_hash</code>. 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.</td>
</tr>
<tr>
<td>`hsh.merge( other_hash ) { key, old_val, new_val</td>
<td>block }<code>=&gt;</code>result_hash`</td>
</tr>
</tbody>
</table>
**merge!**

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

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

1.8

```ruby
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}
```

**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.

```ruby
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`.

```ruby
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`.

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

\[ hsh.\text{select} \{ |\text{key, value} | \text{block} \} \rightarrow \text{array} \]

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

\[
\begin{align*}
\text{h = \{ "a" \rightarrow 100, "b" \rightarrow 200, "c" \rightarrow 300 \}} \\
h.\text{select} \{(k,v) \mid k > "a"\} &\rightarrow \{["b", 200], ["c", 300]\} \\
h.\text{select} \{(k,v) \mid v < 200\} &\rightarrow \{["a", 100]\}
\end{align*}
\]

shift  

\[ hsh.\text{shift} \rightarrow \text{array or nil} \]

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.

\[
\begin{align*}
h = \{ 1 \rightarrow "a", 2 \rightarrow "b", 3 \rightarrow "c" \} \\
h.\text{shift} &\rightarrow [1, "a"] \\
h &\rightarrow \{2 => "b", 3 => "c"\}
\end{align*}
\]

size  

\[ hsh.\text{size} \rightarrow \text{fixnum} \]

Synonym for Hash#length.

sort  

\[ hsh.\text{sort} \rightarrow \text{array} \]

\[ hsh.\text{sort} \{ |a, b| \text{block} \} \rightarrow \text{array} \]

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

\[
\begin{align*}
h = \{ "a" \rightarrow 20, "b" \rightarrow 30, "c" \rightarrow 10 \} \\
h.\text{sort} &\rightarrow \{["a", 20], ["b", 30], ["c", 10]\} \\
h.\text{sort} \{a,b\mid a[1] \leq b[1]\} &\rightarrow \{["c", 10], ["a", 20], ["b", 30]\}
\end{align*}
\]

store  

\[ hsh.\text{store} (\text{key, value}) \rightarrow \text{value} \]

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

to_a  

\[ hsh.\text{to_a} \rightarrow \text{array} \]

Converts \( hsh \) to a nested array of [key, value] arrays.

\[
\begin{align*}
h = \{ "c" \rightarrow 300, "a" \rightarrow 100, "d" \rightarrow 400, "c" \rightarrow 300 \} \\
h.\text{to_a} &\rightarrow \{["a", 100], ["c", 300], ["d", 400]\}
\end{align*}
\]

to_hash  

\[ hsh.\text{to_hash} \rightarrow hsh \]

See page 356.

to_s  

\[ hsh.\text{to_s} \rightarrow \text{string} \]

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.

\[
\begin{align*}
h = \{ "c" \rightarrow 300, "a" \rightarrow 100, "d" \rightarrow 400, "c" \rightarrow 300 \} \\
h.\text{to_s} &\rightarrow "a100c300d400"
\end{align*}
\]
### update

```ruby
hsh.update(other_hash) → hsh
```

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

1.8 Synonym for Hash#merge!.

### value?

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

Synonym for Hash#has_value?.

### values

```ruby
hsh.values → array
```

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

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

### values_at

```ruby
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.

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