Class: Hash (Ruby 2.1.2)

fetch(key [, default]) → obj
fetch(key) {| key | block } → obj
Returns a value from the hash for the given key. If the key can't be found, there are several options: With no other arguments, it will raise an `KeyError` 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.

```
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 (KeyError)
 from prog.rb:2
```

flatten → an_array
flatten(level) → an_array
Returns a new array that is a one-dimensional flattening of this hash. That is, for every key or value that is an array, extract its elements into the new array. Unlike Array#flatten, this method does not flatten recursively by default. The optional *level* argument determines the level of recursion to flatten.

```
a =  {1=> "one", 2 => [2,"two"], 3 => "three"}
a.flatten    # => [1, "one", 2, [2, "two"], 3, "three"]
a.flatten(2) # => [1, "one", 2, 2, "two", 3, "three"]
```

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?(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
```

hash → fixnum

Compute a hash-code for this hash. Two hashes with the same content will have the same hash code (and will compare using `eql?`).

include?(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
```

to_s → string

inspect → string

Return the contents of this hash as a string.

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

Also aliased as: to_s

invert → new_hash

Returns a new hash created by using *hsh*'s values as keys, and the keys as values.

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

keep_if {| key, value | block } → hsh

keep_if → an_enumerator

Deletes every key-value pair from *hsh* for which *block* evaluates to false.

If no block is given, an enumerator is returned instead.

key(value) → key

Returns the key of an occurrence of a given value. If the value is not found, returns `nil`.

```
h = { "a" => 100, "b" => 200, "c" => 300, "d" => 300 }
h.key(200)   #=> "b"
```

```
h.key(300)   #=> "c"
h.key(999)   #=> nil
```

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
```

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 → 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?(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
```

merge(other_hash) → new_hash

merge(other_hash){|key, oldval, newval| block} → new_hash

Returns a new hash containing the contents of *other_hash* and the contents of *hsh*. If no block is specified, the value for entries with duplicate keys will be that of *other_hash*. Otherwise the value for each duplicate key is determined by calling the block with the key, its value in *hsh* and its value in *other_hash*.

```
h1 = { "a" => 100, "b" => 200 }
h2 = { "b" => 254, "c" => 300 }
h1.merge(h2)   #=> {"a"=>100, "b"=>254, "c"=>300}
h1.merge(h2){|key, oldval, newval| newval - oldval}
               #=> {"a"=>100, "b"=>54,  "c"=>300}
```

```
h1              #=> {"a"=>100, "b"=>200}
```

merge!(other_hash) → hsh
merge!(other_hash){|key, oldval, newval| block} → hsh
Adds the contents of *other_hash* to *hsh*. If no block is specified, entries with duplicate keys are overwritten with the values from *other_hash*, otherwise the value of each duplicate key is determined by calling the block with the key, its value in *hsh* and its value in *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 }
h2 = { "b" => 254, "c" => 300 }
h1.merge!(h2) { |key, v1, v2| v1 }
                #=> {"a"=>100, "b"=>200, "c"=>300}
```

rassoc(obj) → an_array or nil
Searches through the hash comparing *obj* with the value using `==`. Returns the first key-value pair (two-element array) that matches. See also `Array#rassoc`.

```
a = {1=> "one", 2 => "two", 3 => "three", "ii" => "two"}
a.rassoc("two")    #=> [2, "two"]
a.rassoc("four")   #=> nil
```

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 `RuntimeError` 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 {|key, value| block} → a_hash
reject → an_enumerator
Returns a new hash consisting of entries for which the block returns false.

If no block is given, an enumerator is returned instead.

```
h = { "a" => 100, "b" => 200, "c" => 300 }
h.reject {|k,v| k < "b"}  #=> {"b" => 200, "c" => 300}
h.reject {|k,v| v > 100}  #=> {"a" => 100}
```

reject! {| key, value | block } → hsh or nil
reject! → an_enumerator
Equivalent to `Hash#delete_if`, but returns `nil` if no changes were made.

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 {|key, value| block} → a_hash
select → an_enumerator
Returns a new hash consisting of entries for which the block returns true.

If no block is given, an enumerator is returned instead.

```
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}
```

select! {| key, value | block } → hsh or nil
select! → an_enumerator
Equivalent to `Hash#keep_if`, but returns `nil` if no changes were made.

shift → anArray or obj
Removes a key-value pair from *hsh* and returns it as the two-item array `[` *key, value* `]`, or the hash's default value if the hash is empty.

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

size → 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
```

store(key, value) → value

Element Assignment¶ ↑

Associates the value given by `value` with the key given by `key`.

```
h = { "a" => 100, "b" => 200 }
h["a"] = 9
h["c"] = 4
h   #=> {"a"=>9, "b"=>200, "c"=>4}
h.store("d", 42) #=> {"a"=>9, "b"=>200, "c"=>4, "d"=>42}
```

`key` should not have its value changed while it is in use as a key (an `unfrozen String` passed as a key will be duplicated and frozen).

```
a = "a"
b = "b".freeze
h = { a => 100, b => 200 }
h.key(100).equal? a #=> false
h.key(200).equal? b #=> true
```

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   #=> [["c", 300], ["a", 100], ["d", 400]]
```

to_h → hsh or new_hash
Returns `self`. If called on a subclass of Hash, converts the receiver to a Hash object.

to_hash => hsh
Returns `self`.