5.1 イントロダクション

配列と同様、ハッシュも利用頻度の高いオブジェクトです。本格的なRubyプログラミングを書くうえでは避けて通ることはできません。また、シンボルは少し変わったデータ型で、他言語の経験者でも「初めて見た」と思う人がいるかもしれません。最初は文字列と混同してしまうかもしれませんが、こちらもやはり使用頻度が高いデータ型なので、しっかり理解していきましょう。

5.1.1 この章の例題：長さの単位変換プログラム

この章では長さの単位を変換するプログラムを作成します。このプログラムを通じて、ハッシュの使い方を学びます。長さの単位変換プログラムの仕様は次のとおりです。

- メートル（m）、フィート（ft）、インチ（in）の単位を相互に変換する。
- 第1引数に変換元の長さ（数値）、第2引数に変換元の単位、第3引数に変換後の単位を指定する。
- メソッドの戻り値は変換後の長さ（数値）とする。端数が出る場合は小数第3位で四捨五入する。

単位の変換には**表5-1**の数値を使います。

表5-1　各単位の数値

単位	略称	m換算
メートル	m	1.00
フィート	ft	3.28
インチ	in	39.37

5.1.2 長さの単位変換プログラムの実行例

長さの単位変換プログラムの実行例を以下に示します。

```
convert_length(1, 'm', 'in')     #=> 39.37
convert_length(15, 'in', 'm')    #=> 0.38
convert_length(35000, 'ft', 'm') #=> 10670.73
```

なお、上の実行例は初期バージョンです。初期バージョンを実装したら、そこから徐々に引数の指定方法を改善していきます。

5.1.3 この章で学ぶこと

この章では以下のようなことを学びます。

- ハッシュ
- シンボル

冒頭でも述べたとおり、ハッシュもシンボルもRubyプログラムに頻繁に登場するデータ型です。この章の

内容を理解して、ちゃんと使いこなせるようになりましょう。

5.2 ハッシュ

ハッシュはキーと値の組み合わせでデータを管理するオブジェクトのことです。ほかの言語では連想配列やディクショナリ（辞書）、マップと呼ばれたりする場合もあります。

ハッシュを作成する場合は以下のような構文（ハッシュリテラル）を使います。ハッシュにおいてはキーと値の組み合わせが1つの要素になります。

```
# 空のハッシュを作る
{}

# キーと値の組み合わせ（要素）を3つ格納するハッシュ
{ キー1 => 値1, キー2 => 値2, キー3 => 値3 }
```

ハッシュはHashクラスのオブジェクトになっています。

```
# 空のハッシュを作成し、そのクラス名を確認する
{}.class #=> Hash
```

以下は国ごとに通貨の単位を格納したハッシュを作成する例です。

```
{ 'japan' => 'yen', 'us' => 'dollar', 'india' => 'rupee' }
```

改行して書くことも可能です。

```
{
  'japan' => 'yen',
  'us' => 'dollar',
  'india' => 'rupee'
}
```

配列と同様、最後にカンマが付いてもエラーにはなりません。

```
{
  'japan' => 'yen',
  'us' => 'dollar',
  'india' => 'rupee',
}
```

同じキーが複数使われた場合は、最後に出てきた値が使われます。ですが、特別な理由がない限りこのようなハッシュを作成する意味はありません。むしろ不具合である可能性のほうが高いでしょう。

```
{ 'japan' => 'yen', 'japan' => '円' } #=> {"japan"=>"円"}
```

ところで、ここまでの説明を読んで「あれ？」と思った方もいるかもしれません。そうです。ハッシュリテラルで使う{}はブロックで使う{}（「4.3.5　do ... endと{}」の項を参照）と使っている記号が同じですね。

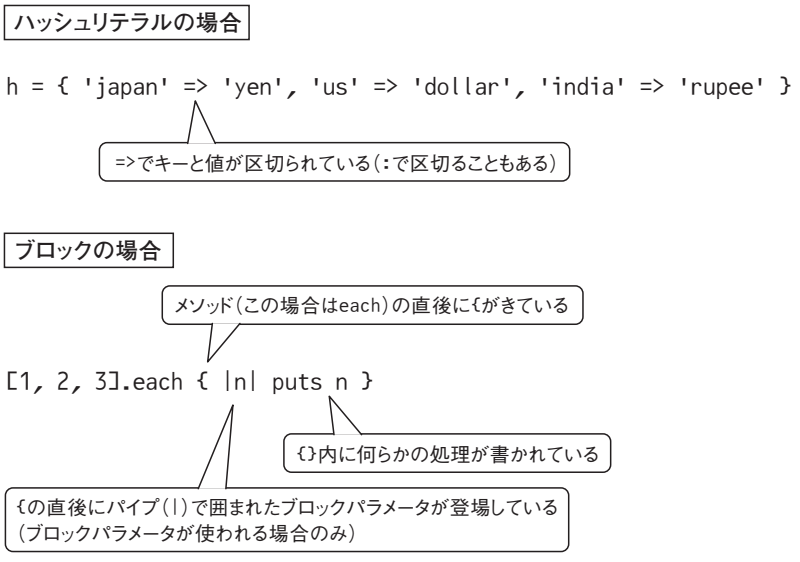
```
# ハッシュリテラルの{}
h = { 'japan' => 'yen', 'us' => 'dollar', 'india' => 'rupee' }

# ブロックを作成する{}
[1, 2, 3].each { |n| puts n }
```

Rubyのコードをたくさん読み書きすればそのうちぱっと見分けがつくようになるはずですが、最初のうちは難しいかもしれないので、見分け方のポイントを簡単に説明しておきます（**図5-1**）。

まず、ハッシュであれば{}の中にはキーと値の組み合わせを示す=>が入ります。ただし、=>の代わりに:で区切る場合もあります。これは「5.4.1　ハッシュのキーにシンボルを使う」の項で紹介します。

一方、ブロックでは{}の中には何らかの処理が書かれます。また、{は必ずメソッドの直後（引数がある場合は( )の直後）に登場します。ブロックパラメータが使われる場合は|n|のようにパイプ（|）で囲まれたパラメータが{の直後に登場するので、ブロックであることが一目瞭然です。

図5-1　ハッシュの{}とブロックの{}の見分け方



ただし、空のハッシュと空のブロックはどちらも{}なので、メソッドの直後に{が来ていればブロック、そうでなければハッシュと見分けるぐらいしかありません。とはいえ、空のハッシュを書くことはあっても、空のブロックを書くことはめったにないはずです。

```
# 空のハッシュ
h = {}

# 空のブロック（めったに使われない）
[1, 2, 3].each {}
```

書き方によってはハッシュのつもりで書いた{}がブロックと解釈されるケースもあります。そのようなコード例は「5.6.7　ハッシュリテラルの {} とブロックの {}」の項で紹介します。

5.2.1 要素の追加、変更、取得

あとから新しい要素を追加する場合は、次のような構文を使います。

```
ハッシュ[キー] = 値
```

以下は新たにイタリアの通貨を追加するコード例です。

```
currencies = { 'japan' => 'yen', 'us' => 'dollar', 'india' => 'rupee' }

# イタリアの通貨を追加する
currencies['italy'] = 'euro'

currencies #=> {"japan"=>"yen", "us"=>"dollar", "india"=>"rupee", "italy"=>"euro"}
```

すでにキーが存在していた場合は、値が上書きされます。

```
currencies = { 'japan' => 'yen', 'us' => 'dollar', 'india' => 'rupee' }

# 既存の値を上書きする
currencies['japan'] = '円'

currencies #=> {"japan"=>"円", "us"=>"dollar", "india"=>"rupee"}
```

ハッシュから値を取り出す場合は、次のようにしてキーを指定します。

```
ハッシュ[キー]
```

以下はハッシュからインドの通貨を取得するコード例です。なお、ハッシュはその内部構造上、キーと値が大量に格納されている場合でも、指定したキーに対応する値を高速に取り出せるのが特徴です。

```
currencies = { 'japan' => 'yen', 'us' => 'dollar', 'india' => 'rupee' }

currencies['india'] #=> "rupee"
```

存在しないキーを指定するとnilが返ります。

```
currencies = { 'japan' => 'yen', 'us' => 'dollar', 'india' => 'rupee' }

currencies['brazil'] #=> nil
```

5.2.2 ハッシュを使った繰り返し処理

eachメソッドを使うと、キーと値の組み合わせを順に取り出すことができます。キーと値は格納した順に取り出されます。ブロックパラメータがキーと値で2個になっている点に注意してください。

```
currencies = { 'japan' => 'yen', 'us' => 'dollar', 'india' => 'rupee' }

currencies.each do |key, value|
  puts "#{key} : #{value}"
end
```

```
#=> japan : yen
#   us : dollar
#   india : rupee
```

ブロックパラメータを1つにするとキーと値が配列に格納されます。

```
currencies = { 'japan' => 'yen', 'us' => 'dollar', 'india' => 'rupee' }

currencies.each do |key_value|
  key = key_value[0]
  value = key_value[1]
  puts "#{key} : #{value}"
end
#=> japan : yen
#   us : dollar
#   india : rupee
```

5.2.3 ハッシュの同値比較、要素数の取得、要素の削除

==でハッシュ同士を比較すると、同じハッシュかどうかをチェックできます。このときすべてのキーと値が同じであればtrueが返ります。

```
a = { 'x' => 1, 'y' => 2, 'z' => 3 }

# すべてのキーと値が同じであればtrue
b = { 'x' => 1, 'y' => 2, 'z' => 3 }
a == b #=> true

# 並び順が異なっていてもキーと値がすべて同じならtrue
c = { 'z' => 3, 'y' => 2, 'x' => 1 }
a == c #=> true

# キー'x'の値が異なるのでfalse
d = { 'x' => 10, 'y' => 2, 'z' => 3 }
a == d #=> false
```

sizeメソッド（エイリアスメソッドはlength）を使うとハッシュの要素の個数を調べることができます。

```
{}.size #=> 0

{ 'x' => 1, 'y' => 2, 'z' => 3 }.size #=> 3
```

deleteメソッドを使うと指定したキーに対応する要素を削除できます。戻り値は削除された要素の値です。

```
currencies = { 'japan' => 'yen', 'us' => 'dollar', 'india' => 'rupee' }
currencies.delete('japan') #=> "yen"
currencies                 #=> {"us"=>"dollar", "india"=>"rupee"}
```

deleteメソッドで指定したキーがなければnilが返ります。ブロックを渡すと、キーが見つからないときにブロックの戻り値をdeleteメソッドの戻り値にできます。

```
currencies = { 'japan' => 'yen', 'us' => 'dollar', 'india' => 'rupee' }

# 削除しようとしたキーが見つからないときはnilが返る
currencies.delete('italy') #=> nil

# ブロックを渡すとキーが見つからないときの戻り値を作成できる
currencies.delete('italy') { |key| "Not found: #{key}" } #=> "Not found: italy"
```

さて、ここまでハッシュのごくごく基本的な使い方を見てきました。このままハッシュの解説を続けても良いのですが、Rubyの開発ではハッシュのキーにシンボルがよく使われます。そこでいったん、シンボルとは何か、ハッシュではなぜキーにシンボルをよく使うのか、ということを説明したあとで、またハッシュの解説に戻ることにします。

5.3 シンボル

Rubyにおける「シンボル」とは何なのでしょうか？　公式リファレンス[注1]では次のように説明されています。

> シンボルは任意の文字列と一対一に対応するオブジェクトです。
> 文字列の代わりに用いることもできますが、必ずしも文字列と同じ振る舞いをするわけではありません。
> 同じ内容のシンボルはかならず同一のオブジェクトです。

いかがでしょうか？　文章だけだと、「わかったような、わからないような」という感想を持つ人も多いのではないかと思います。シンボルと文字列は見た目にはよく似ています。ですが、両者は基本的に別物です。実際のコードを見ながら確認していきましょう。

シンボルは次のようにコロン（**:**）に続けて任意の名前を定義します（シンボルリテラル）。

:シンボルの名前

以下はシンボルを作成するコード例です。

```
:apple
:japan
:ruby_is_fun
```

わざわざ例で示すのも変かもしれませんが、上のシンボルとよく似た文字列を作るコード例です。

```
'apple'
'japan'
'ruby_is_fun'
```

ご覧のとおり、シンボルと文字列は（見た目には）よく似ています。

注1　https://docs.ruby-lang.org/ja/latest/class/Symbol.html

5.3.1 シンボルと文字列の違い

ではここからは、シンボルの特徴と、文字列との違いを説明していきます。まず、シンボルはSymbolクラスのオブジェクトになります。文字列はStringクラスのオブジェクトです。

```
:apple.class  #=> Symbol
'apple'.class #=> String
```

シンボルはRubyの内部で整数として管理されます。表面的には文字列と同じように見えますが、その中身は整数なのです。そのため、2つの値が同じかどうか調べる場合、文字列よりも高速に処理できます。

```
# 文字列よりもシンボルのほうが高速に比較できる
'apple' == 'apple'
:apple  == :apple
```

次に、シンボルは「同じシンボルであればまったく同じオブジェクトである」という特徴があります。このため、「大量の同じ文字列」と「大量の同じシンボル」を作成した場合、シンボルのほうがメモリの使用効率が良くなります。

まったく同じオブジェクトであるかどうかはobject_idを調べるとわかります。同じシンボルを複数作った場合と、同じ文字列を複数作った場合のobject_idを確認してみましょう。

```
:apple.object_id #=> 1143388
:apple.object_id #=> 1143388
:apple.object_id #=> 1143388

'apple'.object_id #=> 70223819213380
'apple'.object_id #=> 70223819233120
'apple'.object_id #=> 70223819227780
```

ご覧のとおり、シンボルはすべて同じIDになりますが、文字列は3つとも異なるIDになります。

最後に、シンボルはイミュータブルなオブジェクトです（イミュータブルなオブジェクトについては「4.7.14　ミュータブル？　イミュータブル？」を参照）。文字列のように破壊的な変更はできないため、「何かに名前を付けたい。名前なので誰かによって勝手に変更されては困る」という用途に向いています。

```
# 文字列は破壊的な変更が可能
string = 'apple'
string.upcase!
string #=> "APPLE"

# シンボルはイミュータブルなので、破壊的な変更は不可能
symbol = :apple
symbol.upcase! #=> undefined method `upcase!' for :apple:Symbol (NoMethodError)
```

5.3.2 シンボルの特徴とおもな用途

シンボルの特徴をまとめると次のようになります。

・表面上は文字列っぽいので、プログラマにとって理解しやすい。

・内部的には整数なので、コンピュータは高速に値を比較できる。
・同じシンボルは同じオブジェクトであるため、メモリの使用効率が良い。
・イミュータブルなので、勝手に値を変えられる心配がない。

シンボルがよく使われるのは、ソースコード上では名前を識別できるようにしたいが、その名前が必ずしも文字列である必要はない場合です。

代表的な利用例はハッシュのキーです。ハッシュのキーにシンボルを使うと、文字列よりも高速に値を取り出すことができます。

```
# 文字列をハッシュのキーにする
currencies = { 'japan' => 'yen', 'us' => 'dollar', 'india' => 'rupee' }
# 文字列を使って値を取り出す
currencies['japan'] #=> "yen"

# シンボルをハッシュのキーにする
currencies = { :japan => 'yen', :us => 'dollar', :india => 'rupee' }
# シンボルを使って値を取り出す（文字列より高速）
currencies[:japan] #=> "yen"
```

プログラム上で区分や状態を管理したいときもシンボルがよく使われます。たとえば、以下のようにタスクの状態を0から2の整数値で管理する場合、コンピュータにとって処理効率は良いですが、人間は数値の意味を脳内で変換する必要があるため、可読性はあまり良くありません。

```
# タスクの状態を整数値で管理する（処理効率は良いが、可読性が悪い）
status = 2

case status
when 0 # todo
  'これからやります'
when 1 # doing
  '今やってます'
when 2 # done
  'もう終わりました'
end
#=> "もう終わりました"
```

もちろん、ここからさらに定数を導入するなどして可読性を上げることもできます。ですが、それよりも次のように各状態をシンボルで管理すれば、処理効率を保ったまま可読性を上げることができます。

```
# タスクの状態をシンボルで管理する（処理効率も可読性も良い）
status = :done

case status
when :todo
  'これからやります'
when :doing
  '今やってます'
when :done
  'もう終わりました'
```

```
end
#=> "もう終わりました"
```

シンボルはこのあとにもたくさん登場するので、Rubyがどういう用途でシンボルを使っているのか注目してみてください。シンボルについてはこれぐらいの内容を理解しておけば、いったんは大丈夫です。では再びハッシュの説明に戻ります。

5.4 続・ハッシュについて

5.4.1 ハッシュのキーにシンボルを使う

さて、先ほども説明したように、（設計上、文字列でもシンボルでもどちらでも良いのなら）ハッシュのキーには文字列よりもシンボルのほうが適しています。ハッシュのキーにシンボルを使うと次のようなコードになります。

```
# ハッシュのキーをシンボルにする
currencies = { :japan => 'yen', :us => 'dollar', :india => 'rupee' }
# シンボルを使って値を取り出す
currencies[:us] #=> "dollar"

# 新しいキーと値の組み合わせを追加する
currencies[:italy] = 'euro'
```

しかし、シンボルがキーになる場合、=>を使わずに“**シンボル：値**”という記法でハッシュを作成できます。コロンの位置が左から右に変わる点に注意してください。

```
# =>ではなく、"シンボル: 値"の記法でハッシュを作成する
currencies = { japan: 'yen', us: 'dollar', india: 'rupee' }
# 値を取り出すときは同じ
currencies[:us] #=> "dollar"
```

キーも値もシンボルの場合は、次のようになります。

```
{ japan: :yen, us: :dollar, india: :rupee }
```

上のハッシュは下のハッシュとまったく同じです。

```
{ :japan => :yen, :us => :dollar, :india => :rupee }
```

コロン（**:**）同士が向き合うので不自然な印象を受けるかもしれませんが、Rubyではこのようなハッシュの記法がよく登場します。=>を使うよりも簡潔に書けるため、本書ではこれ以降、“**シンボル：値**”の記法を使っていきます。

5.4.2 キーや値に異なるデータ型を混在させる

ハッシュのキーは同じデータ型にする必要はありません。たとえば文字列とシンボルを混在させることもできます。しかし、無用な混乱を招くので必要がない限りデータ型はそろえたほうがいいでしょう。

```
# 文字列のキーとシンボルのキーを混在させる（良いコードではないので注意）
hash = { 'abc' => 123, def: 456 }

# 値を取得する場合はデータ型を合わせてキーを指定する
hash['abc'] #=> 123
hash[:def]  #=> 456

# データ型が異なると値は取得できない
hash[:abc]  #=> nil
hash['def'] #=> nil
```

一方、ハッシュに格納する値に関しては、異なるデータ型が混在するケースもよくあります。

```
person = {
  # 値が文字列
  name: 'Alice',
  # 値が数値
  age: 20,
  # 値が配列
  friends: ['Bob', 'Carol'],
  # 値がハッシュ
  phones: { home: '1234-0000', mobile: '5678-0000' }
}

person[:age]             #=> 20
person[:friends]         #=> ["Bob", "Carol"]
person[:phones][:mobile] #=> "5678-0000"
```

5.4.3 メソッドのキーワード引数とハッシュ

Rubyにはキーがシンボルのハッシュによく似た記法でメソッドの引数を指定する「キーワード引数」という機能があります。キーワード引数を使うと、メソッド呼び出し時にどの引数がどんな意味を持つのか対応関係がわかりやすくなります。たとえば、以下のような架空のメソッドがあったとします。

```
def buy_burger(menu, drink, potato)
  # ハンバーガーを購入
  if drink
    # ドリンクを購入
  end
  if potato
    # ポテトを購入
  end
end

# チーズバーガーとドリンクとポテトを購入する
```

```
buy_burger('cheese', true, true)

# フィッシュバーガーとドリンクを購入する
buy_burger('fish', true, false)
```

ここではちゃんとメソッドの引数を確認したあとなので、とくに違和感はないかもしれません。しかし、とくに説明もなく次のようなコードを見せられたらどうでしょうか？

```
buy_burger('cheese', true, true)
buy_burger('fish', true, false)
```

みなさんはこのコードを見て2つめと3つめの引数が何を表しているのか、ぱっと理解できますか？　こういうケースではメソッドのキーワード引数を使うと可読性が上がります。メソッドのキーワード引数は次のように定義します。

```
def メソッド名(キーワード引数1: デフォルト値1, キーワード引数2: デフォルト値2)
  # メソッドの実装
end
```

たとえば、先ほどのbuy_burgerメソッドでキーワード引数を使うと次のようになります。

```
def buy_burger(menu, drink: true, potato: true)
  # 省略
end
```

キーワード引数を持つメソッドを呼び出す場合は、キーがシンボルのハッシュを作るときと同じように、“**引数名：値**”の形式で引数を指定します。

```
buy_burger('cheese', drink: true, potato: true)
buy_burger('fish', drink: true, potato: false)
```

キーワード引数を使わない場合と比べると、引数の役割が明確になりましたね。

```
# キーワード引数を使わない場合
buy_burger('cheese', true, true)
buy_burger('fish', true, false)

# キーワード引数を使う場合
buy_burger('cheese', drink: true, potato: true)
buy_burger('fish', drink: true, potato: false)
```

キーワード引数にはデフォルト値が設定されているので、引数を省略することもできます。

```
# drinkはデフォルト値のtrueを使うので指定しない
buy_burger('fish', potato: false)

# drinkもpotatoもデフォルト値のtrueを使うので指定しない
buy_burger('cheese')
```

キーワード引数は呼び出し時に自由に順番を入れ替えることができます。

```
buy_burger('fish', potato: false, drink: true)
```

存在しないキーワード引数を指定した場合はエラーになります。

```
buy_burger('fish', salad: true) #=> unknown keyword: :salad (ArgumentError)
```

キーワード引数のデフォルト値は省略することもできます。デフォルト値を持たないキーワード引数は、呼び出し時に省略することができません。

```
# デフォルト値なしのキーワード引数を使ってメソッドを定義する
def buy_burger(menu, drink:, potato:)
  # 省略
end

# キーワード引数を指定すれば、デフォルト値ありの場合と同じように使える
buy_burger('cheese', drink: true, potato: true)

# キーワード引数を省略するとエラーになる
buy_burger('fish', potato: false) #=> missing keyword: :drink (ArgumentError)
```

ちなみに、キーワード引数を使うメソッドを呼び出す場合、**を手前に付けることでハッシュをキーワード引数として渡すこともできます。この内容は「5.6.5　ハッシュを明示的にキーワード引数に変換する**」の項で詳しく説明します。

```
params = { drink: true, potato: false }
# **を付けてハッシュをキーワード引数として利用する
buy_burger('fish', **params)
```

では、ここまでに学んだ知識を使って例題を解いてみましょう。例題の解説が終わったら、再びハッシュやシンボルに関する応用的なトピックを説明していきます。

5.5　例題：長さの単位変換プログラムを作成する

まず、長さの単位変換プログラムの仕様をもう1回確認しておきましょう。

- メートル（m）、フィート（ft）、インチ（in）の単位を相互に変換する。
- 第1引数に変換元の長さ（数値）、第2引数に変換元の単位、第3引数に変換後の単位を指定する。
- メソッドの戻り値は変換後の長さ（数値）とする。端数が出る場合は小数第3位で四捨五入する。

単位の変換には**表5-2**の数値を使います。

表5-2　各単位の数値（再掲）

単位	略称	m換算
メートル	m	1.00
フィート	ft	3.28
インチ	in	39.37

長さの単位変換プログラムの実行例を以下に示します。

```
convert_length(1, 'm', 'in')     #=> 39.37
convert_length(15, 'in', 'm')    #=> 0.38
convert_length(35000, 'ft', 'm') #=> 10670.73
```

では今から一緒にこのプログラムを作っていきましょう。

5.5.1 テストコードを準備する

今回もまずテストコードから書いていきます。testディレクトリにconvert_length_test.rbというファイルを作成してください。

```
ruby-book/
├── lib/
└── test/
    └── convert_length_test.rb
```

次に、convert_length_test.rbを開き、以下のようなコードを書いてください。

```
require 'minitest/autorun'

class ConvertLengthTest < Minitest::Test
  def test_convert_length
    assert_equal 39.37, convert_length(1, 'm', 'in')
  end
end
```

ファイルを保存したらテストコードを実行します。

```
$ ruby test/convert_length_test.rb
省略

  1) Error:
ConvertLengthTest#test_convert_length:
NoMethodError: undefined method `convert_length' for #<ConvertLengthTest:0x00007fadc20 ……省略>
    test/convert_length_test.rb:5:in `test_convert_length'

1 runs, 0 assertions, 0 failures, 1 errors, 0 skips
```

convert_lengthメソッドを作っていないので、当然テストは失敗します。

では続いてlibディレクトリにconvert_length.rbというファイルを作成してください。

```
ruby-book/
├── lib/
│   └── convert_length.rb
└── test/
    └── convert_length_test.rb
```

convert_length.rbを開いてconvert_lengthメソッドを実装します。といっても、最初は単純に固定値を返すだけの実装で済ませます。

```
def convert_length(length, unit_from, unit_to)
  39.37
end
```

それからconvert_length_test.rbに戻り、実装コードを読み込みましょう。

```
require 'minitest/autorun'
require_relative '../lib/convert_length'

# 省略
```

こうすれば、とりあえずテストはパスするはずです。

```
$ ruby test/convert_length_test.rb
省略
1 runs, 1 assertions, 0 failures, 0 errors, 0 skips
```

さて、これで実装コードを書いていく準備が整いました。これからが本番です！

5.5.2 いろんな単位を変換できるようにする

では2つめの検証コードを追加してみましょう。今度はインチからメートルへの変換です。

```
require 'minitest/autorun'
require_relative '../lib/convert_length'

class ConvertLengthTest < Minitest::Test
  def test_convert_length
    assert_equal 39.37, convert_length(1, 'm', 'in')
    assert_equal 0.38, convert_length(15, 'in', 'm')
  end
end
```

（変な言い方ですが）期待どおりテストが失敗することを確認します。

```
$ ruby test/convert_length_test.rb
省略
  1) Failure:
ConvertLengthTest#test_convert_length [test/convert_length_test.rb:7]:
Expected: 0.38
  Actual: 39.37

1 runs, 2 assertions, 1 failures, 0 errors, 0 skips
```

ではconvert_lengthメソッドをちゃんと実装していきましょう。今回はメートルとその他の単位との比率をハッシュで定義し、そのハッシュを使って単位を変換することにします。メートルやその他の単位を表すハッシュは次のように書けます。

```
units = { 'm' => 1.0, 'ft' => 3.28, 'in' => 39.37 }
```

また、変換後の長さを求める式は次のようになります。

変換前の単位の長さ ÷ 変換前の単位の比率 × 変換後の単位の比率

1メートルをインチに直すのであれば、

1 ÷ 1.0 × 39.37 = 39.37

になり、1フィートをメートルに直すのであれば、

1 ÷ 3.28 × 1.0 = 0.30（割り切れないので小数第3位を四捨五入）

になります。これらの考えを組み合わせると、Rubyのコードは次のように書けます。

```
def convert_length(length, unit_from, unit_to)
  units = { 'm' => 1.0, 'ft' => 3.28, 'in' => 39.37 }
  (length / units[unit_from] * units[unit_to]).round(2)
end
```

units[unit_from]やunits[unit_to]で各単位の比率をハッシュから取得していることは、みなさんもうおわかりでしょう。端数は小数第3位で四捨五入することになっているので、計算した結果はround(2)で四捨五入しています。

さあ、これでテストを実行するとパスするはずです。やってみましょう。

```
$ ruby test/convert_length_test.rb
省略
1 runs, 2 assertions, 0 failures, 0 errors, 0 skips
```

はい、バッチリですね！　3つめの実行例（フィートからメートルへの変換）も検証してみます。

```
require 'minitest/autorun'
require_relative '../lib/convert_length'

class ConvertLengthTest < Minitest::Test
  def test_convert_length
    assert_equal 39.37, convert_length(1, 'm', 'in')
    assert_equal 0.38, convert_length(15, 'in', 'm')
    assert_equal 10670.73, convert_length(35000, 'ft', 'm')
  end
end
```

テストコードを保存したら、テストを実行してみてください。

```
$ ruby test/convert_length_test.rb
省略
1 runs, 3 assertions, 0 failures, 0 errors, 0 skips
```

こちらも問題なくパスしました！

5.5.3 convert_lengthメソッドを改善する

さて、これだけで終わってしまうとちょっとあっけないので、もう少しこのメソッドを改善してみましょう。まず、メソッドの引数には'm'や'ft'のような文字列を渡していますが、ここでは必ずしも文字列でなくても良い気がします。また、長さの単位はハッシュのキーにもなっています。こういうときに最適なのが……そう、シンボルです！　長さの単位は文字列ではなく、シンボルを渡すようにしてみましょう。

先にテストコードを修正してシンボルを使うようにします。

```
require 'minitest/autorun'
require_relative '../lib/convert_length'

class ConvertLengthTest < Minitest::Test
  def test_convert_length
    assert_equal 39.37, convert_length(1, :m, :in)
    assert_equal 0.38, convert_length(15, :in, :m)
    assert_equal 10670.73, convert_length(35000, :ft, :m)
  end
end
```

それからテストコードを実行して、テストが失敗することを確認します。

```
$ ruby test/convert_length_test.rb
省略
  1) Error:
ConvertLengthTest#test_convert_length:
TypeError: nil can't be coerced into Integer
    /ruby-book/lib/convert_length.rb:3:in `/'
    /ruby-book/lib/convert_length.rb:3:in `convert_length'
    test/convert_length_test.rb:6:in `test_convert_length'

1 runs, 0 assertions, 0 failures, 1 errors, 0 skips
```

シンボルと文字列ではそのままでは互換性がないため、ハッシュから値が取得できずにテストは失敗してしまいました（数値ではなくnilで割り算することになってしまい、TypeErrorが発生しています）。

そこで、ハッシュのキーをシンボルに変更します。キーがシンボルになったので、=>もなくしてシンボルの右側にコロンを付ける記法に書き直しましょう。

```
def convert_length(length, unit_from, unit_to)
  units = { m: 1.0, ft: 3.28, in: 39.37 }
  (length / units[unit_from] * units[unit_to]).round(2)
end
```

これでテストを実行すればパスするはずです。

```
$ ruby test/convert_length_test.rb
省略
1 runs, 3 assertions, 0 failures, 0 errors, 0 skips
```

ご覧のとおり、テストがちゃんとパスしました。

あと、convert_length(1, :m, :in)だと引数の意味が若干わかりにくい気がします。convert_

length(1, from: :m, to: :in)のようにキーワード引数を使うと、引数の意味がより明確になりそうです。というわけで、キーワード引数を使うように再度テストコードを修正します。

```
require 'minitest/autorun'
require_relative '../lib/convert_length'

class ConvertLengthTest < Minitest::Test
  def test_convert_length
    assert_equal 39.37, convert_length(1, from: :m, to: :in)
    assert_equal 0.38, convert_length(15, from: :in, to: :m)
    assert_equal 10670.73, convert_length(35000, from: :ft, to: :m)
  end
end
```

もちろんテストは失敗します。

```
$ ruby test/convert_length_test.rb
省略
  1) Error:
ConvertLengthTest#test_convert_length:
ArgumentError: wrong number of arguments (given 2, expected 3)
    /ruby-book/lib/convert_length.rb:1:in `convert_length'
    test/convert_length_test.rb:6:in `test_convert_length'

1 runs, 0 assertions, 0 failures, 1 errors, 0 skips
```

テストが失敗することを確認したら、キーワード引数を受け取るように実装コードを変更しましょう。デフォルト値はなくてもいいのですが、ここではどちらもメートル（:m）を受け取るようにします。引数の名前がunit_fromやunit_toからfromとtoに変わっている点に注意してください。

```
def convert_length(length, from: :m, to: :m)
  units = { m: 1.0, ft: 3.28, in: 39.37, }
  (length / units[from] * units[to]).round(2)
end
```

これでテストもパスするはずです。

```
$ ruby test/convert_length_test.rb
省略
1 runs, 3 assertions, 0 failures, 0 errors, 0 skips
```

問題ありませんね。

最後に、メソッドの中で作成しているハッシュについて見直してみましょう。このハッシュはとくにキーや値が追加されたり変更されたりしないので、メソッドを実行するたびに作りなおす必要はありません。こういうオブジェクトは、メソッドの外で定数として保持しておくほうが実行効率が良くなります（定数については第7章で詳しく説明するので、ここではとりあえず手順どおりにコードを変更してください）。

というわけで次のようにハッシュをメソッドの外に移動させ、定数化します。

```
UNITS = { m: 1.0, ft: 3.28, in: 39.37 }
def convert_length(length, from: :m, to: :m)
```

```
  (length / UNITS[from] * UNITS[to]).round(2)
end
```

この変更はメソッドの呼び出し方や戻り値に変化がないので（つまり、純粋なリファクタリング）、テストコードはそのままでもパスします。

```
$ ruby test/convert_length_test.rb
省略
1 runs, 3 assertions, 0 failures, 0 errors, 0 skips
```

これでconvert_lengthメソッドは完成です。お疲れ様でした！

ここからあとはまだ説明していないハッシュやシンボルに関する知識を紹介していきます。

Column

メソッド定義側のキーワード引数はシンボルっぽいがシンボルではない

Ruby初心者さんの中には、キーワード引数をシンボルとして参照できないことに疑問を持つ方がいらっしゃるようです。たとえば、こういうイメージです。

```
def buy_burger(menu, drink: true, potato: true)
  # なんで if :drink じゃないの？
  if drink
    # 省略
  end
  # なんで if :potato じゃないの？
  if potato
    # 省略
  end
end
```

確かにdrink: true, potato: trueの部分に着目すると、コロン（:）が付いているせいでキーがシンボルのハッシュが書かれているように見えます。ですが、これは記法が似ているだけでシンボルではありません。あくまでメソッドの引数です。よって、第1引数のmenuと同じようにdrink、potatoのように参照します。

```
def buy_burger(menu, drink: true, potato: true)
  # キーワード引数もメソッドの引数の1つなので、menuと同様にdrinkと書く
  if drink
    # 省略
  end
# ...
```

:drink、:potatoのように書いてしまうと、これはメソッドの引数ではなく、ただのシンボルを書いたことになります。

```
def buy_burger(menu, drink: true, potato: true)
  # :drinkと書いた場合はメソッドの引数ではなく、ただのシンボルになる
  if :drink
    # 省略
  end
# ...
```

ただし、上の話はあくまで「メソッドを定義する側」に注目した場合です。メソッドを呼び出す側になると話が変わってきます。

```
# buy_burgerメソッドを呼び出す場合のdrink:やpotato:はシンボルか否か？
buy_burger('cheese', drink: true, potato: true)
```

ややこしいですが、メソッドを呼び出す側に登場する引数については「シンボルだ」と言えます。なぜなら、メソッド定義側と異なり、呼び出し側は=>を使って呼び出すこともできるからです。よって、シンボルそのもの（シンボルのリテラル）を書いていることになります。

```
# 呼び出す側はどっちの記法でも呼び出せる（ただし通常は上の書き方を使う）
buy_burger('cheese', drink: true, potato: true)
buy_burger('cheese', :drink => true, :potato => true)
```

また、シンボルを変数に格納しても呼び出せます。このことからも呼び出し側は「シンボルを渡している」と言えます。

```
# 変数経由で呼び出すこともできる
# 注意：この例はあくまで実験目的であって、実際にこんなコードを書くことはない
key_1 = :drink
key_2 = :potato
buy_burger('cheese', key_1 => true, key_2 => true)
```

「呼び出す側はシンボルとして引数を渡すが、受け取る側はシンボルでない普通のメソッド引数として値を受け取る」というのは少し不自然な気がしますが、この点については実際にどんどん使って慣れてもらうしかありません。

5.6 ハッシュとキーワード引数についてもっと詳しく

5.6.1 ハッシュで使用頻度の高いメソッド

ハッシュにも数多くのメソッドがありますが、その中でも使用頻度が高いと思われるメソッドを紹介します。

- keys
- values
- has_key?/key?/include?/member?

■keys

keysメソッドはハッシュのキーを配列として返します。

```
currencies = { japan: 'yen', us: 'dollar', india: 'rupee' }
currencies.keys #=> [:japan, :us, :india]
```

■values

valuesメソッドはハッシュの値を配列として返します。