プログラミングスクールに通う前にやっておきたいRuby超基礎入門

♡ 56

Yuki Sako
2018/04/16 21:35

¥980 •••

現役でエンジニアをやりながらプログラミングスクールで講師をしております，迫佑樹と申します.

最近文系でもプログラミングを学び始める人とかが多くて，「独学でやったけど挫折した」とか，「プログラミングスクールの教材難しい...」っていう話をよく聞きます.

そこで今回私が，**プログラミングスクールに通う以前に知っておくと学習効率がアップするようにかなり丁寧にRubyの基礎をまとめてみました.**

このnoteの対象者

・プログラミングスクールへ通う前に基礎を触っておきたい方

・とりあえずプログラミングに触れてみたい方

・一度プログラミングをやってみたが，よくわからなかった方

・Rubyの分厚い本を読んだが挫折した方

このような方向けに，専門用語を使わずできる限り丁寧にRubyというプログラミング言語の基礎をまとめてみました．なお，**「Rubyの超基礎を最短で理解するためのnote」ということで，削れるところは削りできるだけインプット過多にならないように心がけました.**そのため一通り理解したら，他の書籍などと組み合わせて学習してみることをオススメします.

事前準備として，私のブログのCloud9登録方法解説ページ の手順に沿ってCloud9というサービスに登録を済ませておいてください．Windowsユーザでも Macユーザでも大丈夫です.

また，今回のnoteは質問対応付き！エラーが起きた際などは@yuki_99_sまでDMください．対応させていただきます.

約27000文字とボリューム多めですが，「知識をつける」→「実践してみ

る」の流れで自然にプログラミングを学べる構成にしたので，あまり長くは感じないと思います.

試される外資系マネージャー(仮)
@ryo_tkshm

面白い。
このゴールに向かって、着実に一歩ずつ進めて行くの分かり易いです。

> **迫 佑樹/プログラミング講師** @yuki_99_s
> 流行にのって，Rubyの超入門note書きました！
> 僕の教え方に納得して購入して欲しいので，約5000文字無料で見れるようにしてます！
> まだ半分くらいしか書けてないので早割中，随時更新してきます.
>
> 【早割中！】プログラミングスクールに通う前にやっておきたいRuby超基礎入門 note.mu/yuki_99_s/n/nf…

1　22:02 - 2018年4月16日

試される外資系マネージャー(仮)さんの他のツイートを見る

ゲスエンジニア/とだこうき
@cohki0305

人類はこれ以上わかりやすくプログラミングを説明できないんじゃないかってくらいわかりやすいですねw早割のうちに買いましょう！！！

【早割中！】プログラミングスクールに通う前にやっておきたいRuby超基礎入門｜Yuki Sako @yuki_99_s｜note（ノート）
note.mu/yuki_99_s/n/nf…

34　22:32 - 2018年4月16日

プログラミングスクールに通う前にやっておきたいRuby超基礎入...
現役でエンジニアをやりながらプログラミングスクールで講師をしております，迫佑樹と申します． 最近文系でもプログラミングを学
note.mu

ゲスエンジニア/とだこうきさんの他のツイートを見る

■目次

●Rubyってなんなの？

●質問対応について

●初めてのプログラム

・文字を表示してみよう

・計算してみよう

●変数・配列

・変数を使ってラベル付け

・配列で複数のものを一気に

・【ワークショップ】おみくじアプリを作ろう

～以下有料～

●ハッシュ

・配列にしたらわかりにくくなった？

・【ワークショップ】ToDo管理アプリを作ってみよう

●条件分岐 (4月17日追記)

・if文，unless文

・【ワークショップ】ToDo管理アプリを改造しよう

●繰り返し文 (4月19日追記)

・timesメソッド，each文

・【ワークショップ】ToDo管理アプリを完成させよう

●メソッド

・メソッドについて理解しよう

・【ワークショップ】ToDo管理アプリのプログラムを整理しよう

Rubyってなんなの？

Rubyとは，以下のような特徴を持つプログラミング言語のことです.

1. 短く，シンプルなコードで記述できる
2. オブジェクト指向型言語(後述)
3. 日本人が開発した言語のため，日本語学習サイトが豊富

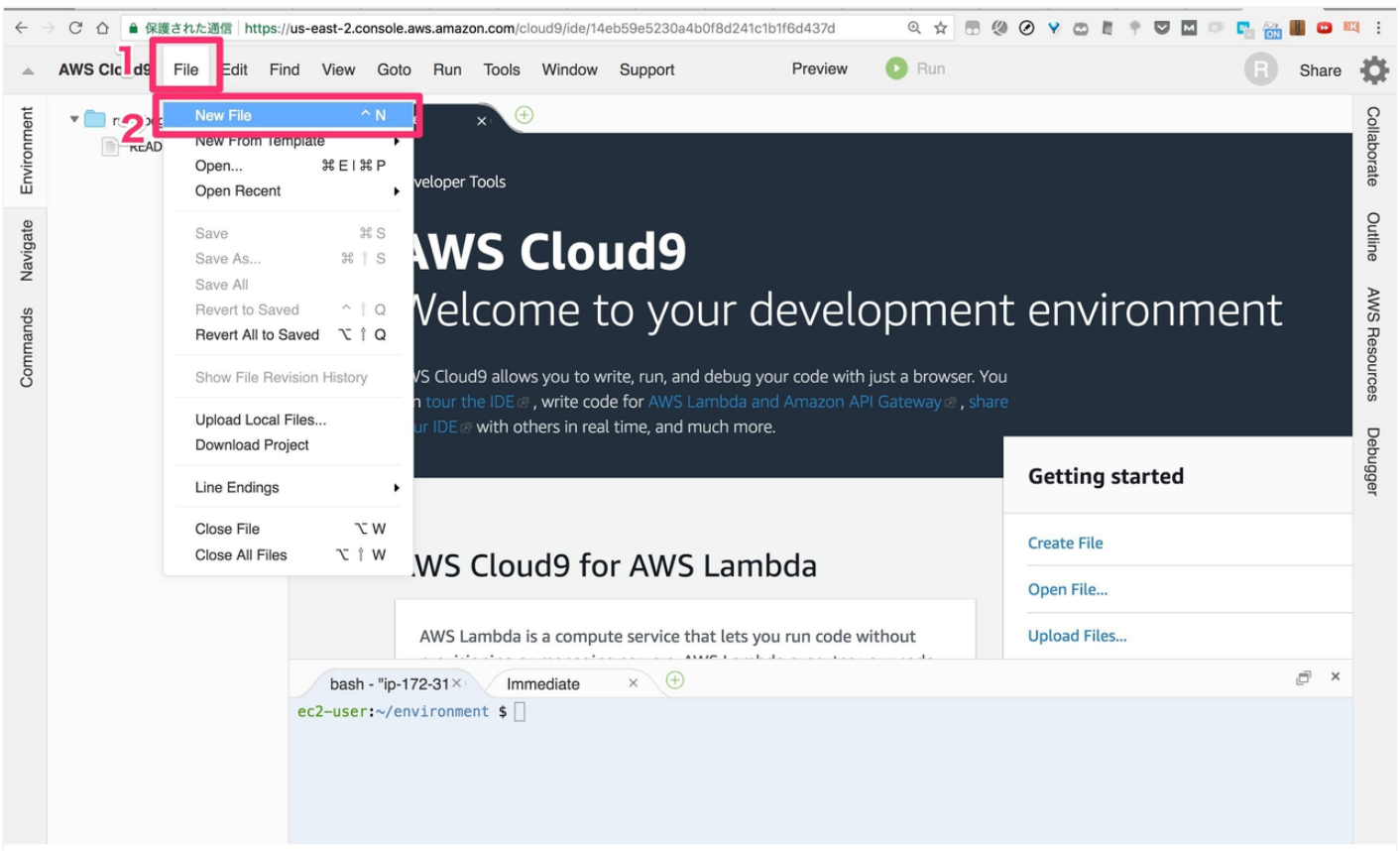
クックパッドやTwitterなどの有名Webアプリケーションにも活用されており，比較的初心者にも扱いやすいということで多くのプログラミングスクールのカリキュラムでも採用されています.

質問対応に関して

冒頭でも説明した通り，読者の方がこのnoteに書いてあるプログラムを実行した時，みなさんのタイプミスなどでエラーが出ることもあるかと思います．
どうしても解決できない際は，Cloud9のコード共有機能で確認させていただくので，よろしくお願いします．

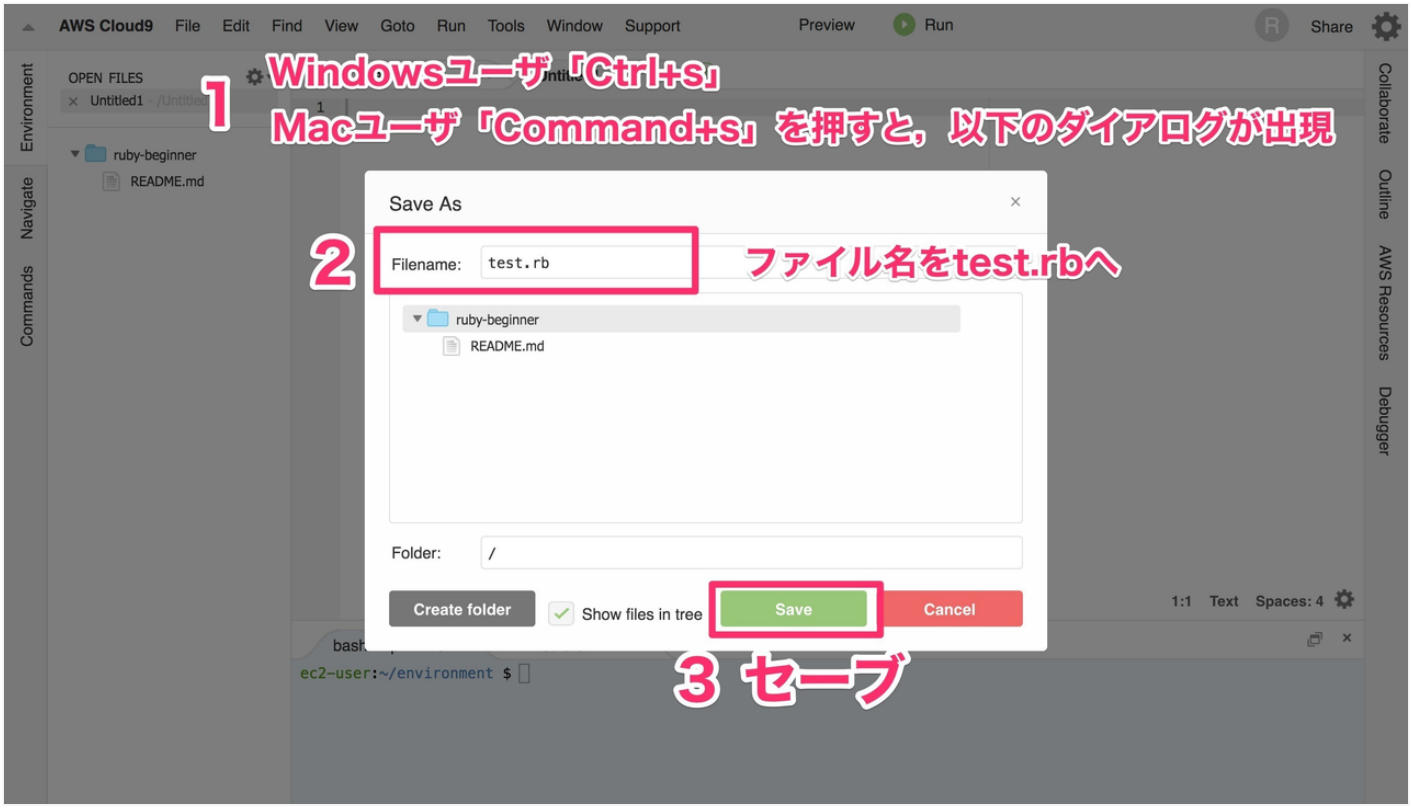
■初めてのプログラム

それでは，さっそくプログラムを書いていきましょう．Cloud9のメニューバーから「file->NewFile」を選択してファイルを作成します．まだCloud9の登録が済んでいない方は，こちらの手順で登録をお願いします．



新しいファイルが生成されたと思います．Windowsの方は「ctrl+s」，Macの方は「Command+s」を押してセーブしてみましょう．ファイル名は

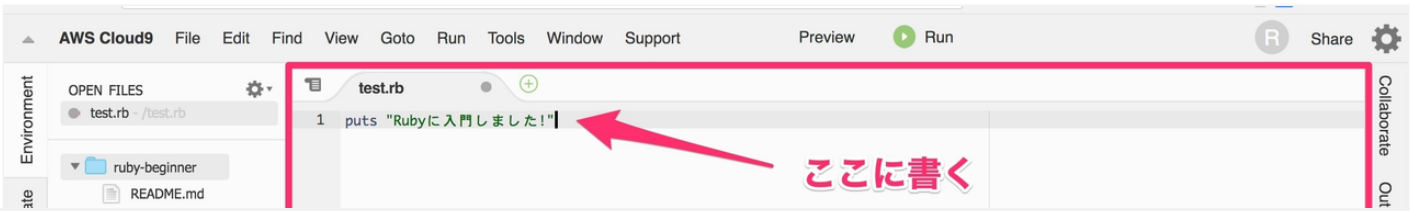
「test.rb」としてください.



それでは，いよいよプログラムを実際に書いていきましょう.

```
puts "Rubyに入門しました！"
```

プログラムは以下の部分に記述をお願いします.



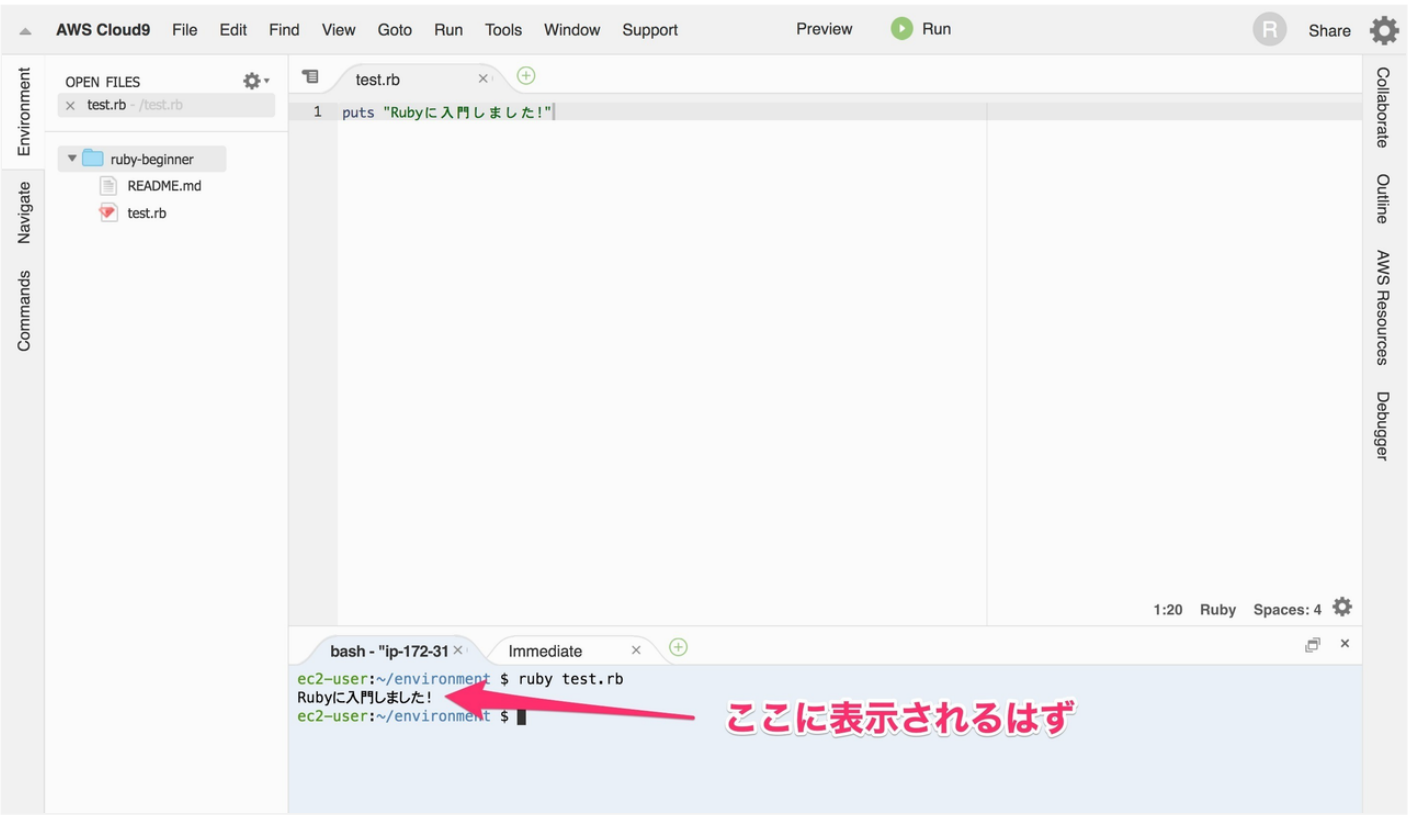
先ほどと同様にWindowsの方は「ctrl+s」，Macの方は「Command+s」を押してセーブをお願いします．(以後，セーブの指示はしません．プログラムを書いたら随時セーブをお願いします)

セーブができたら，以下のコマンドを打ってプログラムを実行していきます．コマンドとは，プログラムを実行するための命令文だと思ってください．

```
ruby test.rb
```

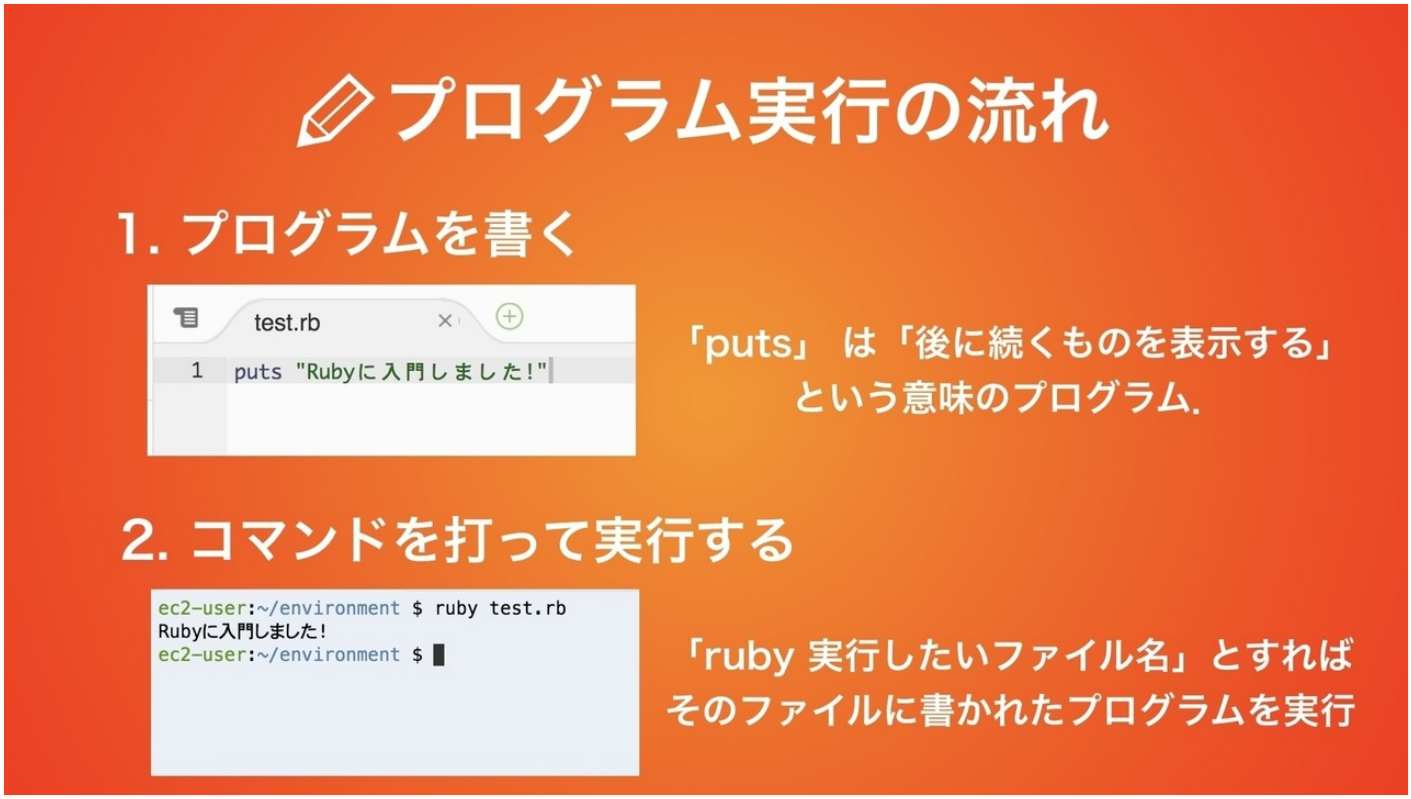
このコマンドを，以下の部分に打ちこんでみてください．

エンターキーを押してみると，いかがでしょうか？「Rubyに入門しました！」と表示されたはずです.



表示されなかった方は，セーブのし忘れかタイプミスのはずですので，再度ご確認をよろしくお願いします.

それでは，プログラム実行の流れを確認していきましょう．

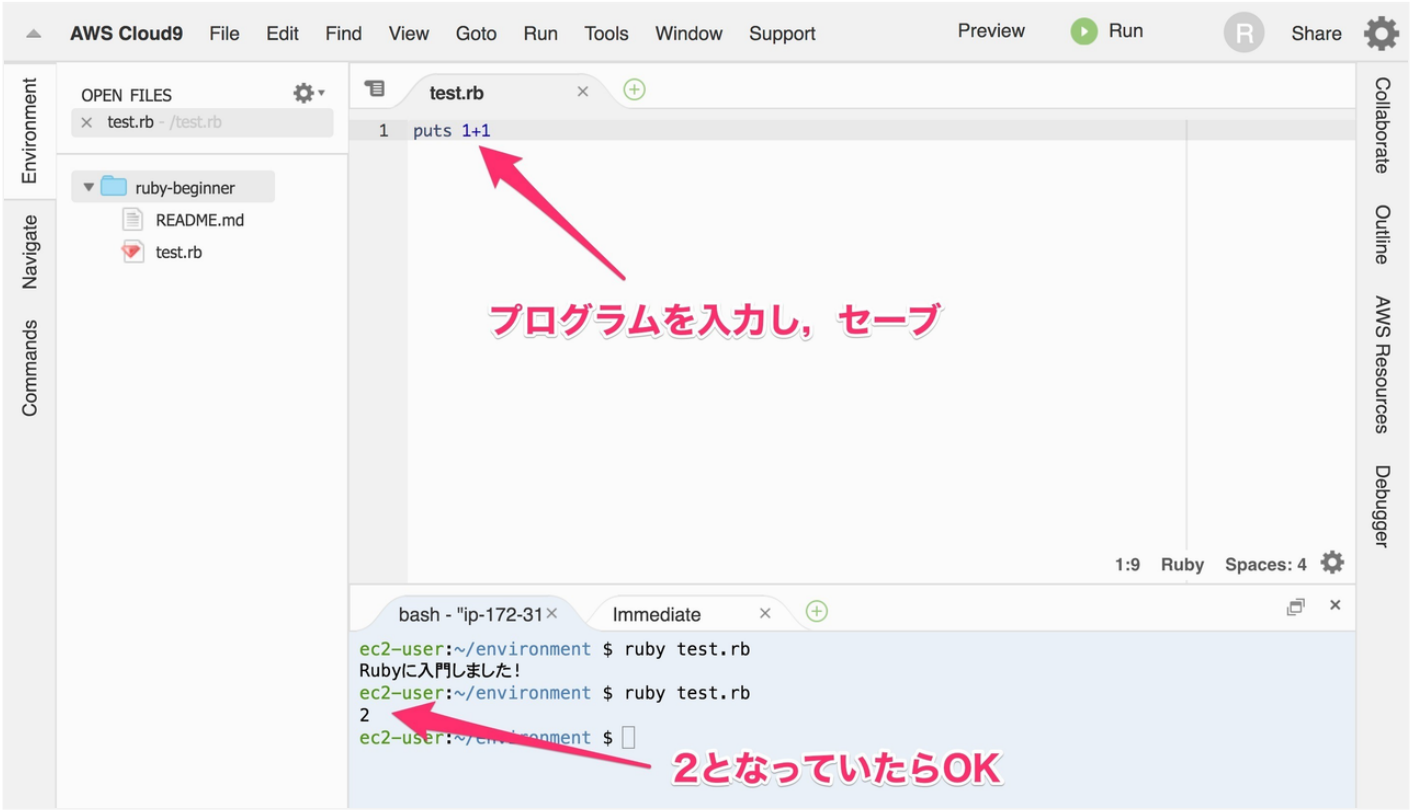


これ以降，「プログラムを入力してください」と指示をした場合，test.rbに打ち込んでいくものだと思っておいてください．また，「実行してください」といった場合，先程と同様に「ruby test.rb」というコマンドによってプログラムの実行をお願いします．

それでは，次に計算をプログラムにさせてみましょう．以下のプログラムを入力してください．

```
puts 1+1
```

実行してみると，「2」と表示されたでしょうか？



「puts」は，その後に続くものを表示するプログラムでしたね．今回は，「1+1」が計算されて2となるので，「puts 1+1」というプログラムを実行すると2と表示されたわけです．

Rubyでは，+，-，*，/，%がそれぞれ足し算，引き算，掛け算，割り算，余りの計算に対応しています．

試しに，以下のプログラムを書いてみましょう．

```
puts 5+2
puts 5-2
puts 5*2
puts 5/2
puts 5.0/2.0
puts 5%2
```

これを実行すると，以下のような結果になります.

```
ec2-user:~/environment $ ruby test.rb
7
3
10
2
2.5
1
```

上から，「足し算，引き算，掛け算，割り算(整数)，割り算(少数)，5を2で割った時のあまり」を計算しています.

このように，Rubyでは「整数同士の割り算は結果も整数に，少数の割り算は少数になる」ということを覚えておいてください.

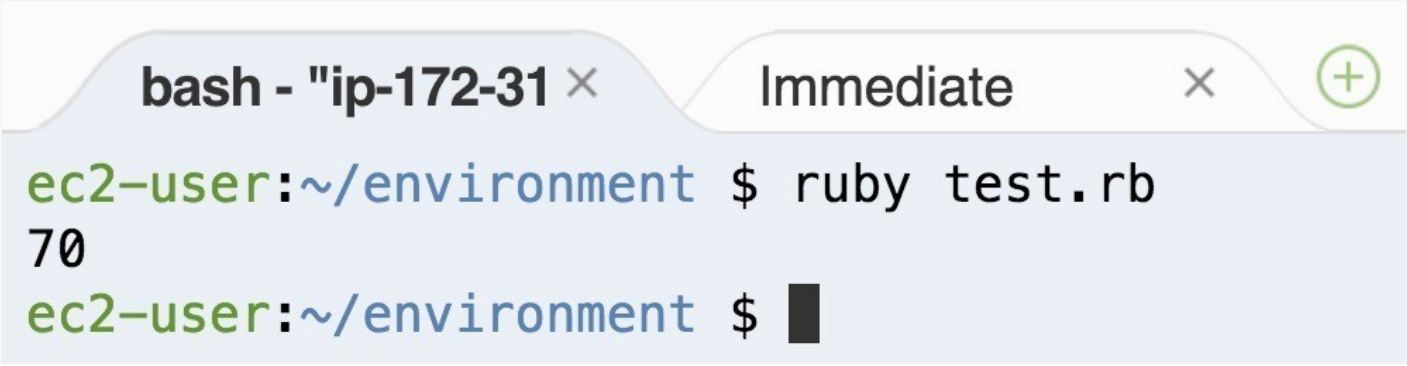
さて，これでRubyを使った計算までができるようになりました.

■変数・配列

さて，それでは次に以下のようなプログラムを実行してみましょう.

```
puts (80+70+60)/3
```

実行結果は，以下のようになったはずです.



80と70と60を足して3で割っただけなので，70になるのはわかるはずですが，これが何を意味しているかをパッとわかった方は少ないのではないでしょうか？

実は，このプログラムは「国語と数学と英語」のテストの平均点を求めるプログラムだったんです.

ただ，こうは思いませんでしたか？**『こんなの数字を足して割ってるだけで，国語と数学と英語の点数を表してるなんてわからない！』**

そう，このような数字を羅列しただけのプログラムってわかりにくいんですよ.

ということで，この数字にラベル付けをしてあげましょう.

```
japanese = 80
math = 70
english = 60
```

```
puts (japanese+math+english)/3
```

こうすることで，最後の4行目をみたときに，『あ，国語と数学と英語の平均点を計算していくんだな』と予想がつくんじゃないでしょうか？

この場合，1行目でjapanese = 80とすることで，4行目のjapaneseは80という数字を表しているのと等価です.

結果的に，「(japanese+math+english)/3」というのは「(80+70+60)/3」と同じ意味になるのですが，しっかりラベル付けした方が後から見たときわかりやすいですよね.

このラベルのことを，専門用語で**「変数」**と呼びます．上記の例だと，mynameとageが変数となり，"田中"にmynameというラベルをつけることを**「mynameという変数に"田中"を代入する」**と呼ぶので覚えておいてください．

それでは，先ほどの点数の平均点を求めるプログラムをさらに改変して，「7科目の平均点」を求めてみましょう．国語・数学・理科・歴史・政治・英語・情報の7科目のテストの平均を求めるプログラムは，こんな感じになるはず．

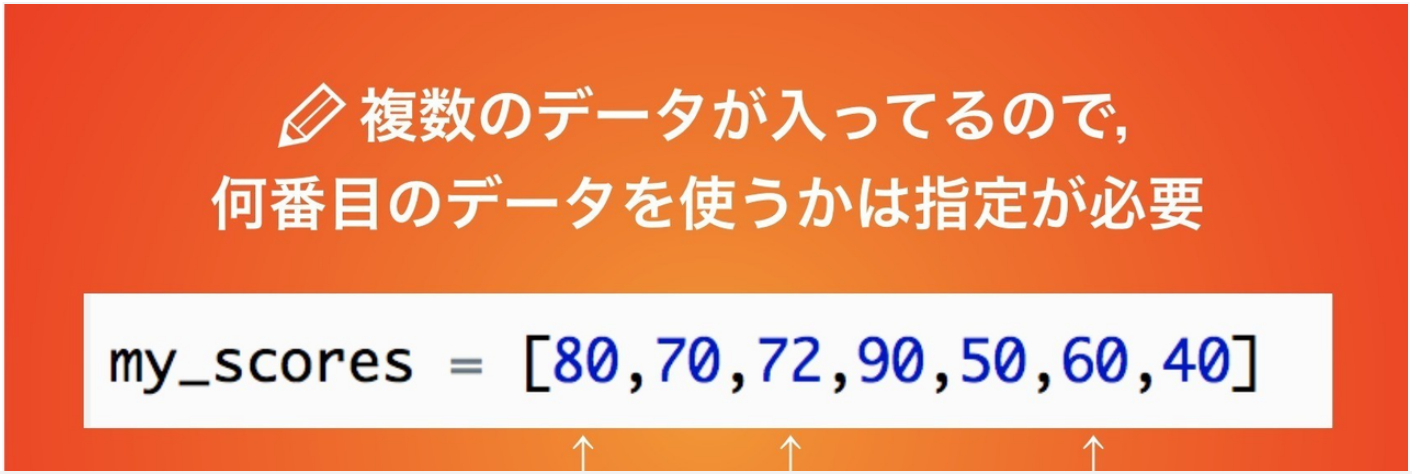
```
japanese = 80
math = 70
science = 72
history = 90
politics = 50
english = 60

information = 40
puts (japanese+math+science+history+politics+english+information)/7.0
```

うーん... これだと，科目の数だけ変数が必要になってきますね...

今回にと，変数は7科目あるせいで7つの変数を用意しなければならないため，無駄にコードが長くなってしまっています．そこで**『配列』**を導入して見ましょう．

実は，『配列』と呼ばれるものを使えば，複数のデータを一括で管理することができるんです！

ここで，配列には複数のデータが管理されているので，使うときは**何番目のデータを使うか**を明示してあげる必要があります．



それでは，少し配列の実験をしてみましょう．次のようなプログラムを書いてみてください．

```
my_scores = [80,70,72,90,50,60,40]

puts my_scores[0]
puts my_scores[1]
puts my_scores[6]
```

以下のような実行結果が得られたでしょうか？

```
bash - "ip-172-31 ×    Immediate ×
ec2-user:~/environment $ ruby test.rb
80
70
40
```

puts my_scores[0]とは，「my_scoresの0番目に入っているデータを表示」というプログラムなので，まず80が表示されたわけです．puts my_scores[1]，puts my_scores[6]も同様に，my_scoresの1番目と6番目に入ってるデータを表示したため，上記のような実行結果を得ることができました．

さて，それではこの配列を使って，平均点を出すプログラムを作って見ましょう.

```
my_scores = [80,70,72,90,50,60,40]
puts (my_scores[0]+my_scores[1]+my_scores[2]+my_scores[3]+my_scores[4]+my_scores[
```

これで，先ほど7科目の平均点を出すのに8行かかってたのを，わずか2行に直すことができました.

足し算しているところがあまり綺麗ではありませんが，これは後ほど直していくことにします.

■ワークショップ おみくじアプリを作ろう

さて，それでは簡単なおみくじアプリを作るワークショップをやってみましょう.

以下のような条件を満たしているのがおみくじアプリです.

・プログラムを実行すると，ランダムに運勢が出る
・運勢は，大吉や中吉，小吉など複数ある

さて，2個目の「運勢は複数ある」というのがキーポイントです. **複数のデータを管理するために有効なもの，覚えていますか？ そう，配列です！**

まずは簡単化のため，毎回大吉が出るおみくじを作ってみましょう．以下のようなコードになるはずです．

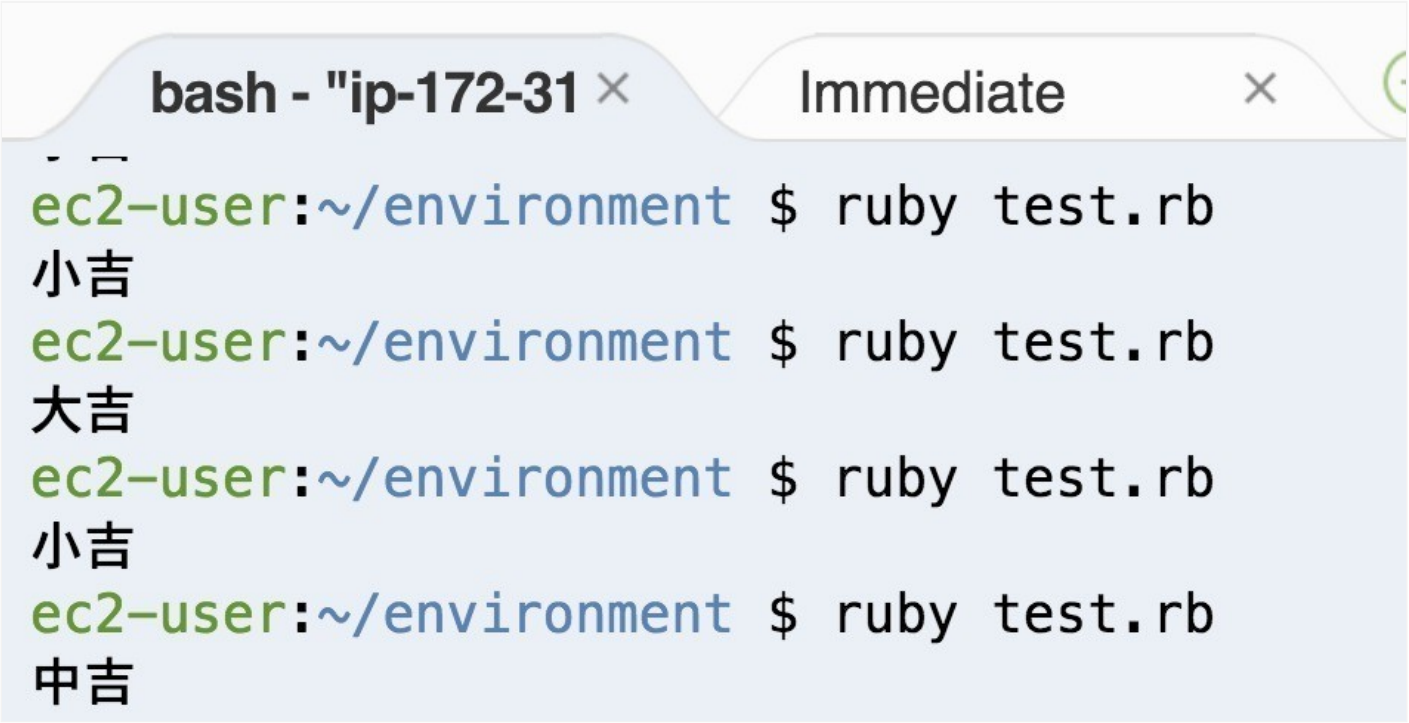
```
omikuji = ["大吉","中吉","小吉"]
puts omikuji[0]
```

「omikuji」という配列には，大吉・中吉・小吉という3つの運勢が記録されています．そして，2行目で0番目を指定して表示しているわけなので，実行すると毎回大吉が出るわけです．

さて，あとはこれをランダムで運勢が表示されるように変更すればOKですね．以下のようにプログラムを書いてみてください．

```
omikuji = ["大吉","中吉","小吉"]
puts omikuji.sample
```

この，「.sample」とは「配列の中からランダムで1つ選んでね」という処理を意味しています．こうすることで，実行するごとに違った運勢が出るおみくじが完成しました！



今回は，「omikuji.sample」として，「omikujiという配列の中からランダムで1つ選んで」という意味の処理をさせました．このように「△△.〇〇」の〇〇のことを「メソッド」と呼びます．

メソッドは他にも用意されていて，例えば「配列の中に入っているデータの数を返すlengthメソッド」というものもあります．

プログラムを，以下のように変えてみてください．

```
omikuji = ["大吉","中吉","小吉"]
puts omikuji.length
```

すると，実行結果は『3』となったはずです．omikijiという配列は3つのデータを保存しているため．omikuji.lengthの結果が3になったわけですね．

このように，Rubyには便利なメソッドがたくさん用意されており，今回のようにおみくじアプリなどをすごくシンプルに開発することができるというわけです．

■ハッシュ

さて，それでは次に，ハッシュについて学んでいきましょう．

先ほど配列について学んだ際，以下のようなプログラムを書きましたね．

```
my_scores = [80,70,72,90,50,60,40]
puts (my_scores[0]+my_scores[1]+my_scores[2]+my_scores[3]+my_scores[4]+my_scores[
```

さて，ここで問題です．歴史の点数は何点だったでしょうか！

…そうです．変数が減ってスッキリしたのはいいんですが，**どの点数がどの科目を表しているのかが全然わからなくなっちゃったんです！**

科目を表している物が全然わからなくなりそうだなぁ…

せっかく変数を導入し，ラベル付けしてわかりやすくしたのに，『行数が増えたから』という理由で1つにまとめたら結果，またわからなくなってしまいました.

1行でシンプルに書けて，さらにわかりやすく書く方法はないのでしょうか？実はあるんです，以下のようなプログラムを書いてみてください！

```
my_scores = {"国語" => 80,"数学" => 70,"理科"=>72,"歴史"=>90,"政治"=>50,"英語"=>60,"情
puts my_scores["国語"]
```

※途中にある「=>」という記号は「=」と「>」をくっつけたものです.

これを実行すると，国語の点数である80と表示されたはずです.

先ほどのコードの1行目「my_scores = {"国語" => 80,"数学" => 70,"理科"=>72,"歴史"=>90,"政治"=>50,"英語"=>60,"情報"=>40}」の波かっこのところが**ハッシュ**と呼ばれるものになっています.

データを使いたい時は，my_scores["国語"]といったように，ラベル名を指定してあげれば使うことが可能です．(ちなみに，このラベル名のことを厳密にはキーと呼びます)

それでは，このハッシュを使用して7科目の平均点を求めるプログラムを書いてみましょう.

```
my_scores = {"国語" => 80,"数学" => 70,"理科"=>72,"歴史"=>90,"政治"=>50,"英語"=>60,"情
puts (my_scores["国語"]+my_scores["数学"]+my_scores["理科"]+my_scores["歴史"]+my_sco
```

こうしてあげれば，「どの科目が何点だったか」というわかりやすさをある程度保ったまま，平均点を計算することができるようになっています.

■ワークショップ タスク管理アプリの開発

さて，それではタスク管理アプリを作っていきましょう！今回作るアプリの仕様はこんな感じ！このタスク管理アプリは，今後の習うことを使って，次章以降のワークショップで改善していきます.

・締め切り・タスクを2つまで追加できる

・随時追加後，メモを閲覧できる

さて，まずは文字の入力をできるようになる必要がありますね．では，以下のプログラムを入力してみてください.

```
puts "タスクを入力してください"
task = gets
puts "入力されたタスクは"
puts task
puts "です。"
```

これを実行し，「タスクを入力してください」と出た後にタスクを打ち込むと，以下のようになると思います.

上記のプログラムで新しく出てきたのは1つだけ，「gets」という処理が出てきました．これは「入力を受け取る」という意味になります.

今回，「task = gets」とすることで入力された文字をtaskという変数に保存しています.

そして4行目で「puts task」とやることで，打ち込んだタスクを表示しているというわけです.

それでは，締め切りも追加してみましょう.

```
puts "締め切りを入力してください"
deadline = gets
puts "タスクを入力してください"
task = gets

puts "---入力されたタスク---"
puts deadline
puts "までに"
puts task
```

実行結果はこんな感じになるはずです.

なんか改行が多すぎて見にくいですね. 「puts」を「print」に変えると改行せずに表示してくれるようになります. なので, 一部printに変えてみることにします.

```
print "締め切りを入力してください→"
deadline = gets
print "タスクを入力してください→"
task = gets

puts "---入力されたタスク---"
print "【締め切り】"
puts deadline
print "【やること】"
puts task
```

こうすると, だいぶ見た目も綺麗になったんじゃないかなと思います.

ただ, 今回のこの「締め切りとやること」って, セットで1つのToDoなわけです. その場合は, 今回学んだハッシュを使うほうがいいですね.

ハッシュを使って書き直してみると以下ようになります.

```
todo = {"締め切り" => "未設定","タスク" => "未設定"}

print "締め切りを入力してください→"
todo["締め切り"] = gets
print "タスクを入力してください→"
todo["タスク"] = gets

puts "---入力されたタスク---"
print "【締め切り】"
puts todo["締め切り"]
print "【やること】"
puts todo["タスク"]
```

ハッシュで書き直しただけなので，実行結果は変わりません．

実行結果は変わりませんが，ハッシュで書くことによってプログラム上でも「タスクと締め切り」がToDoという1つの変数で管理できていることになることになり，あとで見直すときにわかりやすくなっているはずです．

さて，それでは最後に2つまでタスクを管理できるように拡張してみましょう！

複数のデータを保存するには，先ほど学んだ「配列」と呼ばれるものを使えばいいのでしたね．

```
todo_list = [{"締め切り" => "未設定","タスク" => "未設定"},
             {"締め切り" => "未設定","タスク" => "未設定"}]

print "1つめの締め切りを入力してください→"
todo_list[0]["締め切り"] = gets
```

```
print "1つめのタスクを入力してください→"
todo_list[0]["タスク"] = gets

puts "---現在の1つめのタスク---"
print "【締め切り】"
puts todo_list[0]["締め切り"]
print "【やること】"
puts todo_list[0]["タスク"]

print "2つめの締め切りを入力してください→"
todo_list[1]["締め切り"] = gets

print "2つめのタスクを入力してください→"
todo_list[1]["タスク"] = gets

puts "---現在の1つめのタスク---"
print "【締め切り】"
puts todo_list[0]["締め切り"]
print "【やること】"
puts todo_list[0]["タスク"]
puts "---現在の2つめのタスク---"
print "【締め切り】"
puts todo_list[1]["締め切り"]
print "【やること】"
puts todo_list[1]["タスク"]
```

少し長くなってきましたが，やってることはシンプルです．配列とハッシュが混ざると途端に混乱する方がいるので，少し整理してみましょう．

なので，todo_list[0]["締め切り"]と記述すると，**「配列の0番目に入っているハッシュの締め切り」**を取得することができるというわけです．

これを実行すると，このように2つのタスクを登録することができるようになります．

なります.

余力がある人は，3つ以上のタスクを同時に保存できるようにカスタマイズしてみてください！同様のやり方でいけるはずです.

プログラムをよくみると，同じような処理が何度も続いていますね．これは，次章以降で「繰り返し」と呼ばれる処理を学べばシンプルに記述することができるようになります.

■条件分岐

さて，先ほどのToDoアプリについてなのですが，『ToDoを追加したら，その後にToDo一覧が表示される』という仕様になっていました.

ユーザが**「ToDoを追加」したいのか「ToDo一覧を確認」したいのかを判定する仕組みがない**から，「追加した後にとりあえず全部ToDoを表示」という仕様にせざるをなかったんです.

「もしもユーザがToDoを追加したければ，この処理をさせる」といったように，条件に応じて行う処理を変えることを**条件分岐**と呼びます.

さて，それでは以下のようなプログラムを書いて実行してみてください.

```
num = 10
if num > 5
   puts "numは5よりも大きいです"
end
```

このような実行結果になったでしょうか？

では，1行目のnumという変数の値を3に変えて実行してみましょう．以下のように変更してください．

```
num = 3
if num > 5
   puts "numは5よりも大きいです"
end
```

これを実行すると，なにも表示されなかったはずです．

さて，感覚的にもわかって来たかもしれませんが，if num > 5というのは，「numという数字が5よりも大きかった際に中の処理が実行されるような仕組みになっています．

ifの後に条件を書き，その条件が満たされている時に，if〜endの中に書いた処理が実行されるというわけです．

以下の図に示すように条件を表す書き方は，「a > b」「a>=b」「a==b」「a<=B」「a<b」など，複数存在してしているので覚えておいてください．条件を「and」や「or」で組み合わせることで，「条件をどっちも満たしてる時のみこの処理をさせる」とか「どちらか満たしてたら処理をさせる」ということも可能となります．

それでは，次に以下のようなプログラムを書いてみてください．

```
num = 5
if num < 6
   puts "numは6より小さいです"
elsif num >= 10
   puts "numは10以上です"
else
   puts "numは6以上，10より小さいです"
end
```

このnumを例えば5,12,7と変えて実行してみましょう．すると

・numが5の時 → numは6より小さいです

・numが12の時 → numは10以上です

・numが7の時 → numは6以上，10より小さいです

といったような実行結果になったと思います．**「elsif 条件」とすると，「1個目の条件が当てはまっていなかったら，2つ目の条件で確認する」という**

1回目の条件が当てはまっていなかったら，2つ目の条件で確認する」という
たことができるようになります．また，**「else」と「end」で囲った部分の処理は，「どの条件にも当てはまらなかった場合」に実行されるようになっています．**

これでおおよそ，条件分岐(if文)の基礎は抑えられたはずです．それでは次のワークショップで，さっき作ったToDoリストを改善していきましょう！

■ワークショップ ToDoリストアプリにモードを作ろう

それでは，先ほど作ったToDoリストアプリに，2つのモードを追加していきます．

・ToDo追加モード: ToDoを追加することができる
・ToDo確認モード: ToDo一覧を見ることができる

はい，結構そのままですw それでは，プログラムを書いていきましょう．まずは，モード切替の部分だけを書いていきます．

```
puts "【ToDoアプリを起動しました】"
puts "  [show] ToDoを確認する"
puts "  [add] ToDoを追加する"
print "  showまたはaddと入力してください→"
mode = gets.chomp!
```

```
if mode == "show"
   puts "【ToDo確認モードを選択しました】"
elsif mode == "add"
   puts "【ToDo追加モードを選択しました】"
else
   puts "エラーです．プログラムを終わります"
   exit
end
```

ここで，5行目に「gets.chomp!」という見慣れない処理が入っていると思います．「gets」で入力した文字を受け取ることができるのは前回説明した通りですが，**入力した後エンターキーを押すため，この入力した文字は最後に改行が入るんですよ**．「.chomp!」と書いてあげると，この余計な改行を省くことができるというわけです．

入力した文字を「mode」という変数に記録し，そのmodeの中に入っている文字をif文を使って比較しているという流れですね．

そして，下から2行目にある「exit」は「プログラムを終了する」という意味の処理です．

実行してみると，このようになるはず．

このように，入力した文字によって表示される文章が変わっていたら成功です！

さて，それではToDoの追加，確認をできるようにしてみましょう！前回ハッシュのワークショップで作ったプログラムを参考にしながら，プログラムを書き換えてみてください.

少し長いですが，同じような処理が多いです．初心者の方は，定着させるという意味も込めて書いてみてくださいね！

```
todo_list = [{"締め切り" => "未設定","タスク" => "未設定"},
             {"締め切り" => "未設定","タスク" => "未設定"},
             {"締め切り" => "未設定","タスク" => "未設定"}]

puts "【モードを選択】"
puts "  [show] ToDoを確認する"
puts "  [add] ToDoを追加する"
print "  showまたはaddと入力してください→"
mode = gets.chomp!

if mode == "show"
   puts "【ToDo確認モードを選択しました】"
   puts "現在ToDoはありません"
elsif mode == "add"
   puts "【ToDo追加モードを選択しました】"
   print "1つめの締め切りを入力してください→"
   todo_list[0]["締め切り"] = gets.chomp!
   print "1つめのタスクを入力してください→"
   todo_list[0]["タスク"] = gets.chomp!
else
   puts "エラーです．プログラムを終わります"
   exit
end

puts "【モードを選択】"
puts "  [show] ToDoを確認する"
puts "  [add] ToDoを追加する"
print "  showまたはaddと入力してください→"
mode = gets.chomp!

if mode == "show"
   puts "【ToDo確認モードを選択しました】"
   print "1. "
   print todo_list[0]["締め切り"]
```

```
    print "までに"
    puts todo_list[0]["タスク"]
    print "2. "
elsif mode == "add"
    puts "【ToDo追加モードを選択しました】"
    print "2つめの締め切りを入力してください→"
    todo_list[1]["締め切り"] = gets.chomp!

    print "2つめのタスクを入力してください→"
    todo_list[1]["タスク"] = gets.chomp!
else
    puts "エラーです．プログラムを終わります"
    exit
end

puts "【モードを選択】"
puts "  [show] ToDoを確認する"
puts "  [add] ToDoを追加する"
print "  showまたはaddと入力してください→"
mode = gets.chomp!

if mode == "show"
    puts "【ToDo確認モードを選択しました】"
    print "1. "
    print todo_list[0]["締め切り"]
    print "までに"
    puts todo_list[0]["タスク"]
    print "2. "
    print todo_list[1]["締め切り"]
    print "までに"
    puts todo_list[1]["タスク"]
elsif mode == "add"
    puts "【ToDo追加モードを選択しました】"
    print "3つめの締め切りを入力してください→"
    todo_list[2]["締め切り"] = gets.chomp!

    print "3つめのタスクを入力してください→"
    todo_list[2]["タスク"] = gets.chomp!
else
    puts "エラーです．プログラムを終わります"
    exit
end
```

これを実行してみると，こんな感じになります．

だいぶそれっぽいToDoアプリになってきましたね．しかしこのアプリ，1回目に「show」を選び，2回目に「add」，3回目に「show」を選ぶとどうなるでしょうか？

このToDoアプリの欠点をあげてみましょう．

・同じようなコードを何度も書いている
・1番が未設定なのに2番から設定されてしまう
・メモを3つまでしか追加できない

次章で繰り返しという処理を学ぶと，これらを**一気に全部解決**することができるようになります！これができると，結構しっかりしたToDoアプリになりますのでお楽しみに！

■繰り返し

さて，先ほど『同じようなコードを何度も書いている』という問題に直面しましたね．

例えば，『5回，"こんにちは"と表示するプログラム』となると，今まで使った知識だけを使う場合は以下のようになりますね．

```
puts "こんにちは"
puts "こんにちは"
puts "こんにちは"
puts "こんにちは"
puts "こんにちは"
```

これでは，100回同じことをさせるとなると気が遠くなるくらいコピペする必要が出てきますよね...

実は，以下のようなプログラムを書いてあげると，同じことを何度も書く必要がなくなるんです.

```
5.times do
 puts "こんにちは"
end
```

これを実行してみると，以下のようになったはずです.

かなりシンプルな形になっているので，わかりやすいですが『do〜end』の中に書かれた処理が指定された回数文繰り返される構造になっています.

ちなみに，doの後に「||」の中に変数を作ってあげると，その変数で「何回

めのループか」というのを確認することが可能です．実際に，以下のようなプログラムを書いてみてください．

```
10.times do |num|
   print num
   puts "回めの繰り返しです"
end
```

これを実行すると，以下のような結果になるはずです．

この「num」という変数に，「何回目の繰り返しか」がしっかり保存されていますね．今回は例として「num」という名前をつけましたが，もちろん名前は何でも良いです．以下のプログラムのように，「count」という変数名にしても問題はありません，わかりやすい名前をつけておいてくださね．

```
10.times do |count|
  print count
  puts "回めの繰り返しです"
end
```

さて，それではこれを使って配列に保存されているデータを1つずつ取り出すような処理を書いてみることにしましょう．

最初の方にやった「おみくじ」のプログラムを思い出しながら以下のようなプログラムを書いてみてください.

```
omikuji = ["大吉","中吉","小吉"]
(omikuji.length).times do |num|
   print num
   puts "回めの繰り返しです"
end
```

ここで,「.length」と書いてあげると「配列の中にデータがいくつ入っているか」を取得することができたのでした. 今回は3つのデータが入っている

ので「omikuji.length」は3となります. つまり「(omikuji.length).times do」は「3.times do」と同じなので, 以下のような結果になるというわけです.

さて, そして「何回目の繰り返しか」というのは「num」という変数に保存されていたのでした. つまり, **numが0〜2の中で変化するので, omikuji[num]としてあげればomikujiという配列に保存されている0番目〜2番目のデータを取得することができる**というわけです.

詳しくは後ほど図解しますが, 以下のようなプログラムを実行して確認してみましょう.

```
omikuji = ["大吉","中吉","小吉"]
```

```
(omikuji.length).times do |num|
   puts omikuji[num]
end
```

実行すると，以下のようになったはずです.

配列の中身が1つずつ表示されていることがわかりますね．少し複雑になってきたので，整理してみましょう.

繰り返しと配列と変数が同時に出てくるとちょっとややこしいですよね．実は，もっとシンプルな書き方があります！

以下のように書いて実行してみてください！

```
omikuji = ["大吉","中吉","小吉"]
omikuji.each do |content|
   puts content
end
```

このコードを実行してみると，先程と同じ実行結果が得られたと思います.

「.each」と書くと，配列の中から1つずつデータを取り出して，「| |」で囲

った変数に入れてくれます．それを3行目でputsによって表示しているという流れ．

これ，結構つまずきやすいところですが，**「配列から繰り返し1つずつデータを取り出しているだけ」**と考えてもらえればOKです．

Ryusei Kiyose @lplj59_ng_lit · 2018年1月5日
@yuki_99_s さこぽん、@tomoki_sun が、Rubyのeachで手こずってるらしいっす

迫 佑樹/プログラミング講師
@yuki_99_s

```
vec = [1,2,3]
vec.each do |content|
  puts content
end
```

ってしたら，１２３って出力されるねんな．
こんな感じで，配列の中にあるものを一個ずつ取り出して| |で囲まれた変数に入れることで，doとendの中で使えるようになってるの！

2　20:22 - 2018年1月5日

迫 佑樹/プログラミング講師さんの他のツイートを見る

1回目の例では「times」という繰り返し文を使って，配列の「何番目のデータを取得するか」を指定していきましたが，2回目の例では「each」を使って配列のデータを一つずつ取得する方法を学びました．2回目の例でやったeachの方がシンプルでよく使うので，覚えておいてくださいね．

さて，それでは先ほどの掲示板アプリを拡張していきましょう！

■ワークショップ ToDo管理アプリを完成させよう

まずは，ToDoの一覧のところに絞って，todo_listという配列に入ったデータをeach文を使って表示してみるプログラムを書いて実行してみましょう.

```
todo_list = [{"締め切り" => "2018年4月20日","タスク" => "ToDoアプリを作る"},
          {"締め切り" => "2018年5月15日","タスク" => "ブログを1記事書く"},
          {"締め切り" => "2018年5月20日","タスク" => "レポートを書く"}]

todo_list.each do |todo|
   print todo["締め切り"]
   print "までに"
   puts todo["タスク"]
end
```

このプログラムを実行すると，以下のようになります.

もうeach文にも慣れてきたかと思いますが，一応図解しておくとこんな感じ.

今までは，「todo_list[1]["締め切り"]」といったように，「配列の何番目か」を毎回指定していましたが，each文で1つずつ取り出せるようになったのでそんなめんどくさいことが不要になったというわけです.

さて，それではメインのToDo管理アプリのプログラムへ戻りましょう.

前回，以下のようなコードを書きました．(このコードは前回と全く同じものです)

```
todo_list = [{"締め切り" => "未設定","タスク" => "未設定"},
            {"締め切り" => "未設定","タスク" => "未設定"},
            {"締め切り" => "未設定","タスク" => "未設定"}]
puts "【モードを選択】"
puts "  [show] ToDoを確認する"
puts "  [add] ToDoを追加する"
print "  showまたはaddと入力してください→"
mode = gets.chomp!
if mode == "show"
  puts "【ToDo確認モードを選択しました】"
  puts "現在ToDoはありません"
elsif mode == "add"
  puts "【ToDo追加モードを選択しました】"
  print "1つめの締め切りを入力してください→"
  todo_list[0]["締め切り"] = gets.chomp!
  print "1つめのタスクを入力してください→"
  todo_list[0]["タスク"] = gets.chomp!
else
  puts "エラーです．プログラムを終わります"
  exit
end
puts "【モードを選択】"
puts "  [show] ToDoを確認する"
puts "  [add] ToDoを追加する"
print "  showまたはaddと入力してください→"
mode = gets.chomp!
if mode == "show"
  puts "【ToDo確認モードを選択しました】"
  print "1. "
  print todo_list[0]["締め切り"]
  print "までに"
  puts todo_list[0]["タスク"]
  print "2. "
elsif mode == "add"
  puts "【ToDo追加モードを選択しました】"
  print "2つめの締め切りを入力してください→"
  todo_list[1]["締め切り"] = gets.chomp!
  print "2つめのタスクを入力してください→"
  todo_list[1]["タスク"] = gets.chomp!
else
  puts "エラーです．プログラムを終わります"
  exit
end
```

```
puts "【モードを選択】"
puts "  [show] ToDoを確認する"
puts "  [add] ToDoを追加する"
print "  showまたはaddと入力してください→"
mode = gets.chomp!
if mode == "show"
  puts "【ToDo確認モードを選択しました】"
  print "1. "
  print todo_list[0]["締め切り"]
  print "までに"
  puts todo_list[0]["タスク"]
  print "2. "
  print todo_list[1]["締め切り"]
  print "までに"
  puts todo_list[1]["タスク"]
elsif mode == "add"
  puts "【ToDo追加モードを選択しました】"
  print "3つめの締め切りを入力してください→"
  todo_list[2]["締め切り"] = gets.chomp!
  print "3つめのタスクを入力してください→"

  todo_list[2]["タスク"] = gets.chomp!
else
  puts "エラーです．プログラムを終わります"
  exit
end
```

このコードって，**よくみたら追加と確認の処理を繰り返しやっているだけなんです．**なので22行目以降をばっさり切り落として，以下のように書き換えてみてください．

```
todo_list = [{"締め切り" => "未設定","タスク" => "未設定"},
          {"締め切り" => "未設定","タスク" => "未設定"},
          {"締め切り" => "未設定","タスク" => "未設定"}]
puts "【モードを選択】"
puts "  [show] ToDoを確認する"
puts "  [add] ToDoを追加する"
print "  showまたはaddと入力してください→"
mode = gets.chomp!
if mode == "show"
 puts "【ToDo確認モードを選択しました】"
 puts "現在ToDoはありません"
elsif mode == "add"
```

```
 puts "【ToDo追加モードを選択しました】"
 print "1つめの締め切りを入力してください→"
 todo_list[0]["締め切り"] = gets.chomp!
 print "1つめのタスクを入力してください→"
 todo_list[0]["タスク"] = gets.chomp!
else
 puts "エラーです．プログラムを終わります"
 exit
end
```

このプログラムの「ToDo確認モード」のところを先ほど学んだeachで書き直してみると，以下のようになりますね．

```
todo_list = [{"締め切り" => "未設定","タスク" => "未設定"},
         {"締め切り" => "未設定","タスク" => "未設定"},
         {"締め切り" => "未設定","タスク" => "未設定"}]
puts "【モードを選択】"
puts "  [show] ToDoを確認する"
puts "  [add] ToDoを追加する"
print "  showまたはaddと入力してください→"
mode = gets.chomp!
if mode == "show"
   puts "【ToDo確認モードを選択しました】"
   todo_list.each do |todo|
       print todo["締め切り"]
       print "までに"
       puts todo["タスク"]
   end
elsif mode == "add"
   puts "【ToDo追加モードを選択しました】"
   print "1つめの締め切りを入力してください→"
   todo_list[0]["締め切り"] = gets.chomp!
   print "1つめのタスクを入力してください→"
   todo_list[0]["タスク"] = gets.chomp!
else
   puts "エラーです．プログラムを終わります"
   exit
end
```

この状態で実行して確認モードを選択すると，まだ未設定ではありますが

この状態で実行して確認モードを選択すると，また未設定ではめりますか，
3つの締め切りとタスクが表示されたのではないでしょうか？

では，ToDoを追加する処理を書いていきましょう．未設定と出るのはかっこ悪いので，1行目の配列の中身を空っぽにして，追加モードの中の処理を以下のように変更してみてください！

```
todo_list = []
puts "【モードを選択】"
puts "  [show] ToDoを確認する"
puts "  [add] ToDoを追加する"
print "  showまたはaddと入力してください→"
mode = gets.chomp!
if mode == "show"
  puts "【ToDo確認モードを選択しました】"
  todo_list.each do |todo|
      print todo["締め切り"]
      print "までに"
      puts todo["タスク"]
  end
elsif mode == "add"
  puts "【ToDo追加モードを選択しました】"
  print "締め切りを入力してください→"
  deadline = gets.chomp!
  print "タスクを入力してください→"
  task = gets.chomp!
  todo_list.push({"締め切り" => deadline,"タスク" => task})
else
  puts "エラーです．プログラムを終わります"
  exit
end
```

「push」という新しい処理が出てきていますね．この「push」とは追加モー

ドのところだけ抜粋して解説すると，以下のような感じになっています.

最後に，このプログラムは1回追加or確認をしたら終了してしまうようになっているので，とりあえず100回繰り返すような仕様にしてみましょう.

2行目に「100.times do」を追加し，末尾に「end」を追加することでメインの処理を100回繰り返すことができるようになっています.

```
todo_list = []
100.times do
   puts "【モードを選択】"
   puts "  [show] ToDoを確認する"
   puts "  [add] ToDoを追加する"
   print "  showまたはaddと入力してください→"
   mode = gets.chomp!
   if mode == "show"
     puts "【ToDo確認モードを選択しました】"
     todo_list.each do |todo|
         print todo["締め切り"]
         print "までに"
         puts todo["タスク"]
     end
   elsif mode == "add"
     puts "【ToDo追加モードを選択しました】"
     print "締め切りを入力してください→"
     deadline = gets.chomp!
     print "タスクを入力してください→"
     task = gets.chomp!
     todo_list.push({"締め切り" => deadline,"タスク" => task})
   else
     puts "エラーです．プログラムを終わります"
     exit
   end
end
```

これを実行すると，ほぼ理想としていたToDo管理アプリができていることが

これと実行すると，私が想像していたToDo管理ツールができていることがわかると思います．

これで，以下のような仕様を満たすToDoアプリをかなり少ない行数で作ることができました．

・ToDoを幾つでも保存しておける
・締め切り及びタスクをユーザが追加することができる
・締め切り及びタスクを一覧形式で閲覧できる
・ToDoの追加モードと確認モードをユーザが切り替えられる

今回はRubyの基礎ということで解説はしませんが，データベースと連携し，Webブラウザ上で表示させられるようになるとWebアプリケーションになるわけです．

また，繰り返し文には今回紹介した「times」以外にも「while」「for」「loop」など複数あります．今回は「繰り返し」という基本概念を抑えてもらうことを目的としたので，混乱を避けるために「times」のみ紹介しました．

興味がある方はさらに繰り返しについて詳しく調べてみてください．

プログラムを処理のまとまりごとに分けて読みやすくするメソッドという手法について学び，終わりにしたいと思います！

■メソッド

それでは最後にメソッドについて学んで行きましょう.

メソッドとは簡単にいうと，**「処理のかたまりをまとめたもの」**だと思ってください.

同じような処理を何度も行うときには，処理をしっかりとまとめておくと，あとで見たときにとてもわかりやすくなります.

それでは，実際に以下のようなプログラムを書いてみてください.

```
10.times do |num|
  puts num
end
```

このプログラムは，以前にやったように0〜9までカウントアップを行うようなプログラムになっています.

それでは，この「0〜9までカウントアップさせる」という処理に「count_up」という名前をつけてまとめてみることにしましょう.

詳しくは次に説明するので，とりあえず以下のプログラムを書いてみてください．(ちなみに，理由は後述しますが以下のプログラムを実行しても何も表示されません.

```
def count_up
  10.times do |num|
     puts num
  end
end
```

この，処理のまとまりを作るための上記のプログラムは以下のような構造になっています.

ちなみに，この「def」とは「定義する」を意味するdefineの略です．まとまりとして処理を定義しているということですね.

さて，先ほどのプログラムを実行しても，何も表示されなかったはず．それが正しいのです.

というのも，先ほどの段階では**「まとまりを作っただけ」であり，「そのま**

とまった処理を使っていない」というわけなんですよ．言い換えると，商品をパッケージ化したけど，まだその商品を誰も使っていない状態です．

それでは，以下のようにcount_upという処理を使うために，一番最後に「count_up」と書いてみてください！

```
def count_up
  10.times do |num|
     puts num
  end
end

count_up
```

こうしてあげると，しっかりとカウントアップされるようになったのではないかなと思います．

今回，「count_up」という名前をつけて0〜9までカウントアップする処理をまとめました．なので，以下のようにcount_upを2個書けば0〜9まで2回カウントアップしてくれます．

```
def count_up
  10.times do |num|
     puts num
  end
end
```

```
count_up
count_up
```

これを実行すると，以下のようにしっかりと2回カウントアップされていることがわかりますね.

ここで流れを復習しておくと，処理をまとめるときにやるべきことは以下の2つ.

1. メソッドを定義する (処理のまとまりを作る)
2. メソッドを使う (まとまった処理を使う)

さて，先ほどは0〜9までカウントアップする処理をまとめたのですが，さらには「いくつカウントアップするか」を指定することができるようにして行きましょう．以下のようにプログラムを変更してみてください.

```
def count_up(count_number)
  count_number.times do |num|
     puts num
  end
end

count_up(5)
```

これを実行してみると，以下のような結果になったはずです

これを実行してみると，以下のような結果になったはずです．

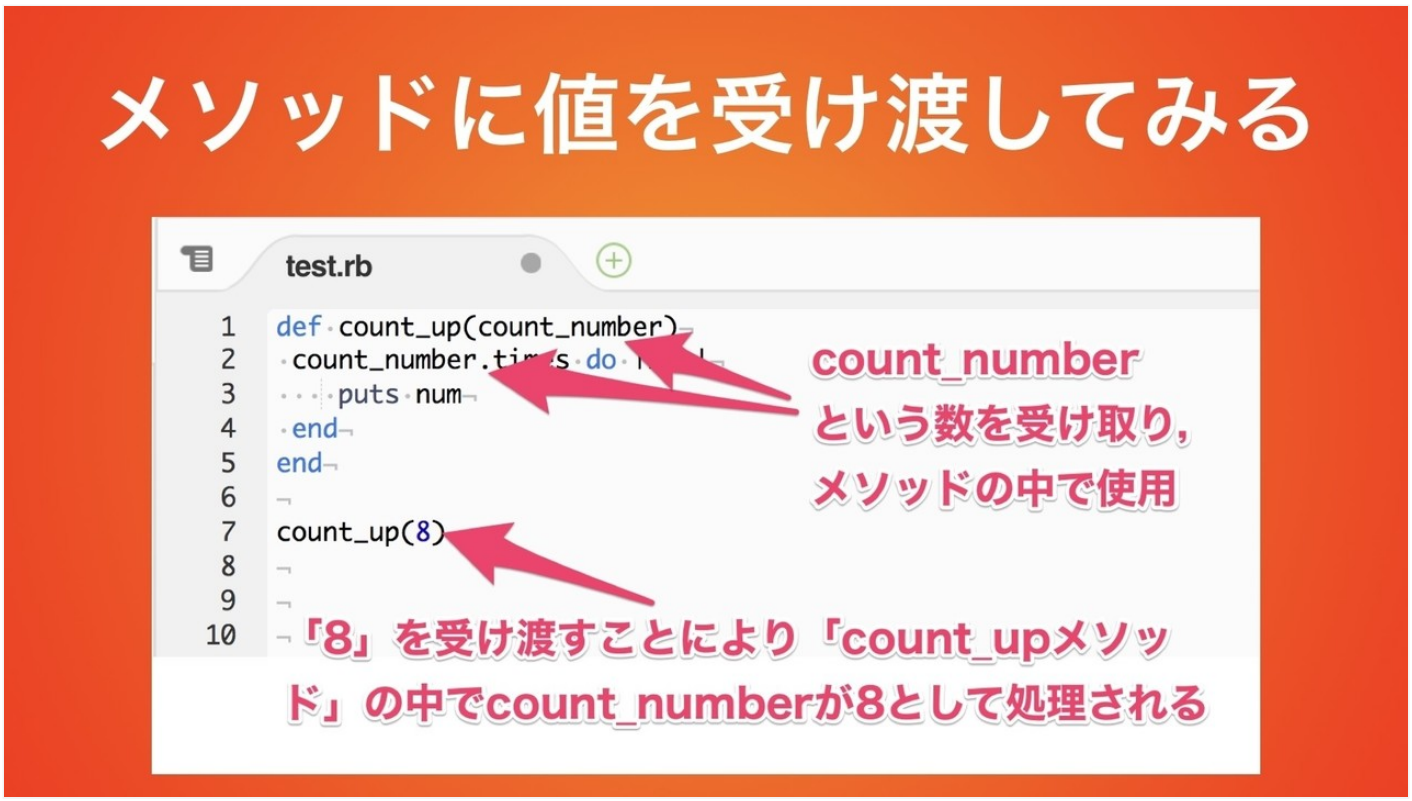
では，一番最後の「count_up」のかっこの中の数字を8とかに変えてみてください．

```
def count_up(count_number)
 count_number.times do |num|
    puts num
 end
end

count_up(8)
```

実行してみると，このようになりますね．

```
ec2-user:~/environment $ ruby test.rb
0
1
2
3
4
5
6
7
```

もうお分かりの通り，最後の「count_up」の中に渡す数字によって，カウントアップさせる数字を変えることができています．



メソッドで値を受け渡す手順は以下の通りです.

1. 『なんという変数名で値を受け取り，どう使うのか』を定義部分で決定
2. メソッドを使用する際に，実際に値を受け渡す

ちなみに，メソッドに受け渡す値は複数でも大丈夫です.

例えば，以下のようなプログラムを書いてみましょう.

```
def sum(a,b)
  a+b
end

c = sun(3,5)
puts c
```

上の3行で，「受け取ったaとbという数字を足す」というメソッドを作っています．そして，「c = sum(3,5)」とすることで，足した結果をcという変数に保存し，最後にcを表示しているといった流れになっています.

それでは最後に，このメソッドを使ってToDoアプリをわかりやすく直していくワークショップをやってみましょう！

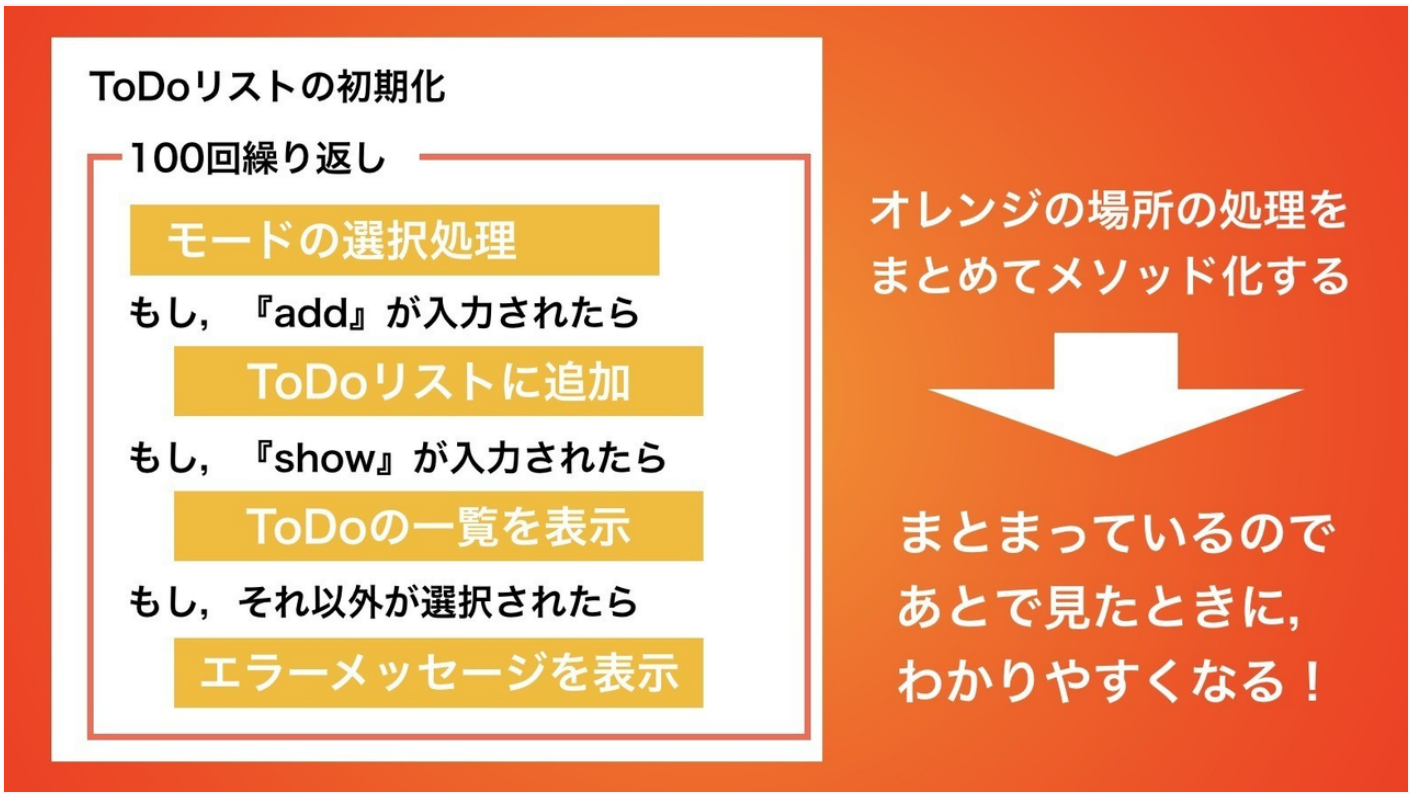
■ワークショップ ToDo管理アプリの処理をまとめよう

いよいよ最後のワークショップとなりました．前回のワークショップの時点で，以下のようなプログラムが完成していました．(こちらは前回のコードと同じです)

```
todo_list = []
100.times do
  puts "【モードを選択】"
  puts "  [show] ToDoを確認する"
  puts "  [add] ToDoを追加する"
  print "  showまたはaddと入力してください→"
  mode = gets.chomp!
```

```
  if mode == "show"
    puts "【ToDo確認モードを選択しました】"
    todo_list.each do |todo|
        print todo["締め切り"]
        print "までに"
        puts todo["タスク"]
    end
  elsif mode == "add"
    puts "【ToDo追加モードを選択しました】"
    print "締め切りを入力してください→"
    deadline = gets.chomp!

    print "タスクを入力してください→"
    task = gets.chomp!
    todo_list.push({"締め切り" => deadline,"タスク" => task})
  else
    puts "エラーです．プログラムを終わります"
    exit
  end
end
```

このプログラムの構造を，日本語で書いてみると以下のようになっています．



では，まずmodeを選択する処理をまとめて「select_mode」というメソッドを作って分けてみましょう．メソッドは，「def メソッド名」とするんでした．わかりやすくするために「#」で始まるコメント文を入れておきました．**「#」で始まる文はプログラムとしては認識されないため，メモなどに使うことが可能**です．

```
#メソッド定義部分
def select_mode
 puts "【モードを選択】"
 puts "  [show] ToDoを確認する"
 puts "  [add] ToDoを追加する"
 print "  showまたはaddと入力してください→"
 gets.chomp!
end

#これより下がメインの処理
todo_list = []
100.times do
 mode = select_mode
 if mode == "show"
   puts "【ToDo確認モードを選択しました】"
   todo_list.each do |todo|
       print todo["締め切り"]
       print "までに"
       puts todo["タスク"]
   end
 elsif mode == "add"
   puts "【ToDo追加モードを選択しました】"
   print "締め切りを入力してください→"
   deadline = gets.chomp!
   print "タスクを入力してください→"
   task = gets.chomp!
   todo_list.push({"締め切り" => deadline,"タスク" => task})
 else
   puts "エラーです．プログラムを終わります"
   exit
 end
end
```

『putsする処理』と「ユーザも文字を受け取る処理」をまとめて

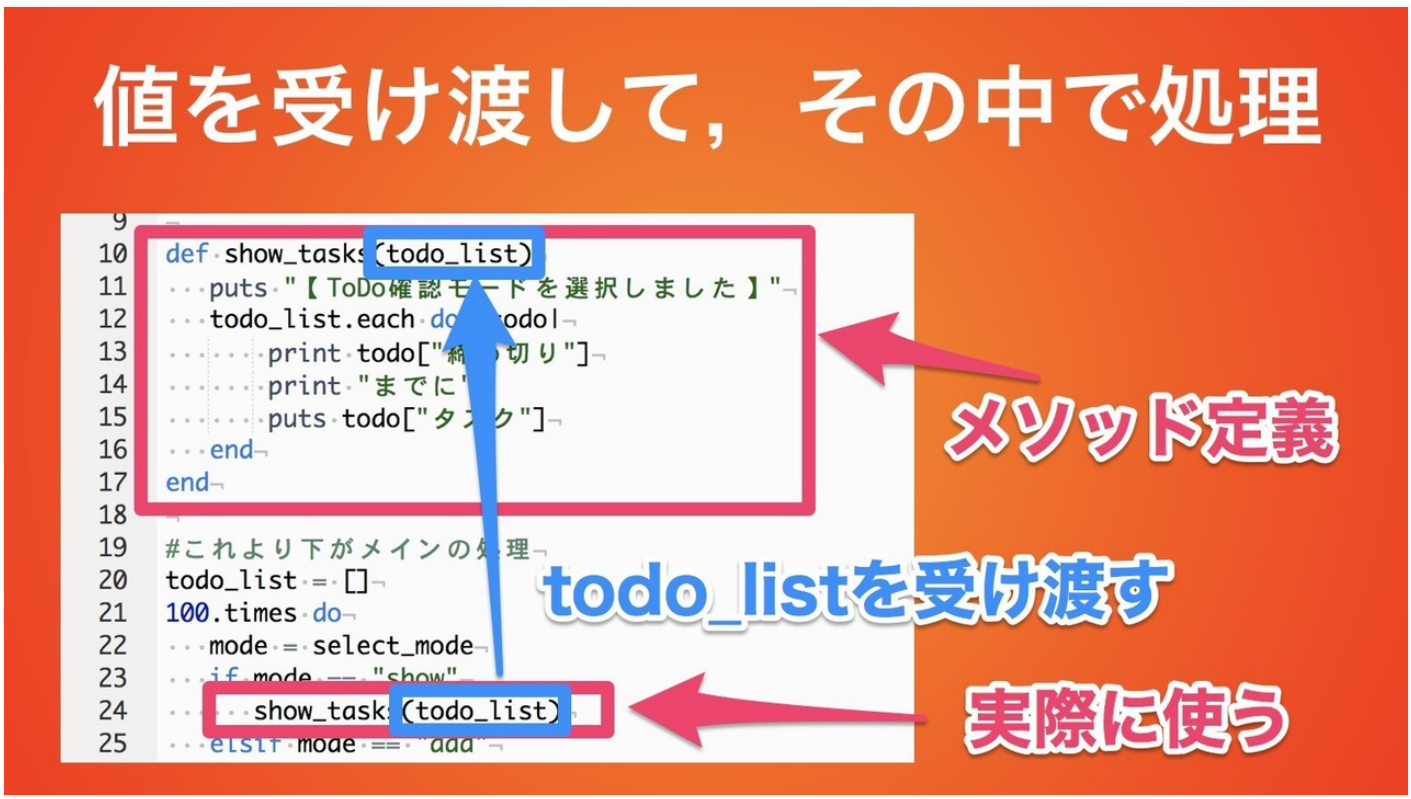
「select_mode」という名前をつけてみました．そして，「mode = select_mode」とすることで，入力したモードの情報を「mode」という変数に入れています．

それでは次に，タスクの一覧を表示させる「show_tasks」というメソッドを作ってみます．今回は「show_tasks(todo_list)」とすることで，ToDoリストをメソッドに渡し，その中で表示の処理を行なっている感じになっています．

```
#メソッド定義部分
def select_mode
  puts "【モードを選択】"
  puts "  [show] ToDoを確認する"
  puts "  [add] ToDoを追加する"
  print "  showまたはaddと入力してください→"
  gets.chomp!
end

def show_tasks(todo_list)
  puts "【ToDo確認モードを選択しました】"
  todo_list.each do |todo|
      print todo["締め切り"]
      print "までに"
      puts todo["タスク"]
  end
end

#これより下がメインの処理
todo_list = []
100.times do
  mode = select_mode
  if mode == "show"
     show_tasks(todo_list)
  elsif mode == "add"
     puts "【ToDo追加モードを選択しました】"
     print "締め切りを入力してください→"
     deadline = gets.chomp!
     print "タスクを入力してください→"
```

```
        task = gets.chomp!
        todo_list.push({"締め切り" => deadline,"タスク" => task})
    else
        puts "エラーです．プログラムを終わります"
        exit
    end
end
```

今回は，「show_tasks」というメソッドに対して，「todo_list」を受け渡しています．ここは少しややこしいところでもあるので以下の図で立ち止まって理解してみてください！



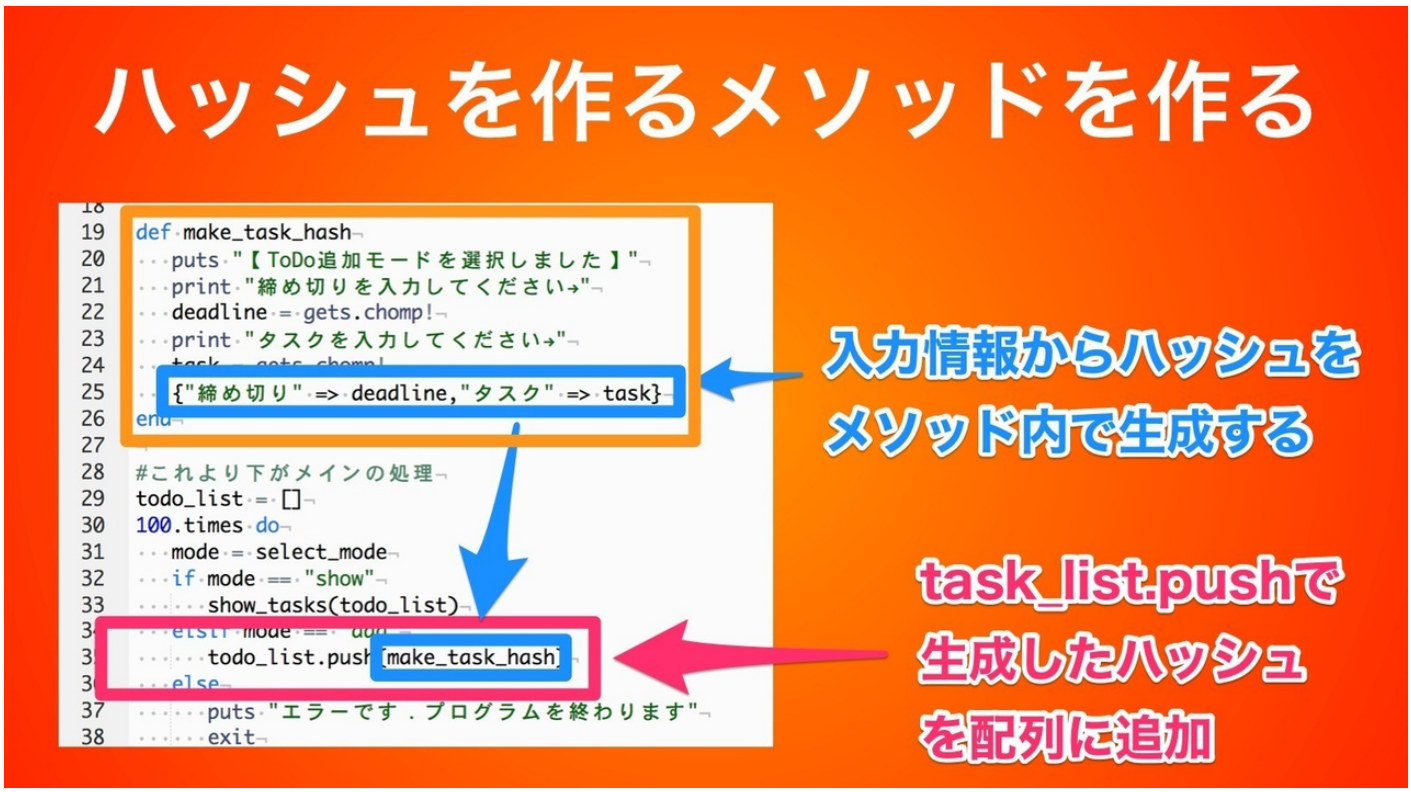
さて，これで「モード選択」と「タスク一覧の表示」をメソッドとして切り分けることができました．では次に，「タスクの追加」をメソッドに分けていきくことを考えています．**今回はハッシュを作る部分だけを「make_task_hash」というメソッドに分けていきましょう！**

```
# メソッド定義部分
```

```ruby
#メソッド定義部分
def select_mode
  puts "【モードを選択】"
  puts "  [show] ToDoを確認する"
  puts "  [add] ToDoを追加する"
  print "  showまたはaddと入力してください→"
  gets.chomp!
end

def show_tasks(todo_list)
  puts "【ToDo確認モードを選択しました】"
  todo_list.each do |todo|
      print todo["締め切り"]
      print "までに"
      puts todo["タスク"]
  end
end

def make_task_hash
  puts "【ToDo追加モードを選択しました】"
  print "締め切りを入力してください→"
  deadline = gets.chomp!
  print "タスクを入力してください→"
  task = gets.chomp!
  {"締め切り" => deadline,"タスク" => task}
end

#これより下がメインの処理
todo_list = []
100.times do
  mode = select_mode
  if mode == "show"
     show_tasks(todo_list)
  elsif mode == "add"
     todo_list.push(make_task_hash)
  else
     puts "エラーです．プログラムを終わります"
     exit
  end
end
```

「make_task_hash」というメソッドを作って，その最後で「{"締め切り" => deadline,"タスク" => task}」というハッシュを生成していますね．**つまり，このハッシュが「make_task_hash」というメソッド内で生成されたわけです．**

そして最後に，**todo_list.push(make_task_hash)としてあげれば，「make_task_hash」内で作られたハッシュをtodo_listという配列に追加しているという流れ**です．



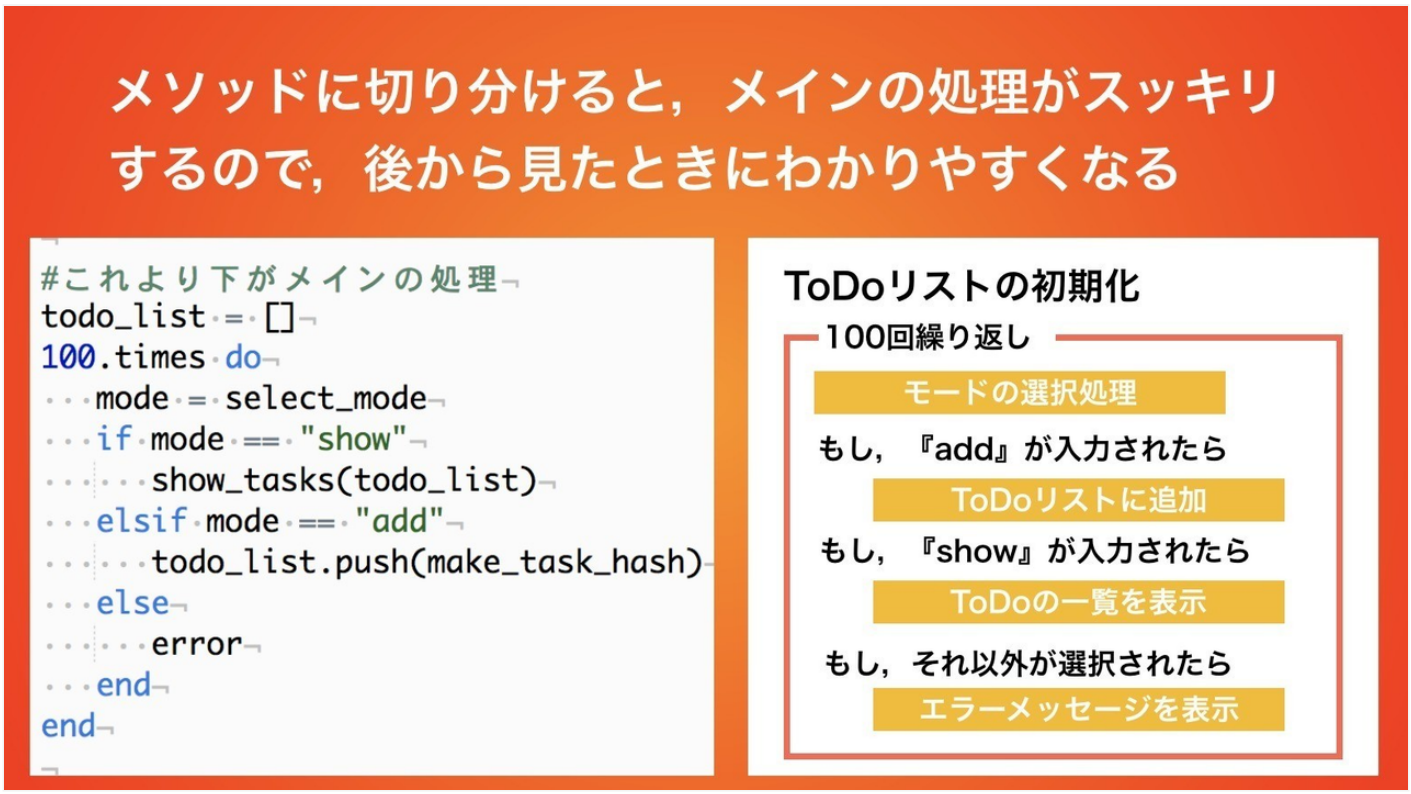
ややこしくなってきた方は，メソッドとして切り分けた後のプログラムと，切り分ける前のプログラムを見返してみてください．難しそうに見えますが，「処理を切り分けているだけ」だということを思ってプログラムを比較してみると意外と簡単だったりします．

それでは最後に，「error」というメソッドに，エラー文を切り分けてみれば完成です．

```ruby
#メソッド定義部分
def select_mode
  puts "【モードを選択】"
  puts "  [show] ToDoを確認する"
  puts "  [add] ToDoを追加する"
  print "  showまたはaddと入力してください→"
  gets.chomp!
end

def show_tasks(todo_list)
  puts "【ToDo確認モードを選択しました】"
  todo_list.each do |todo|
      print todo["締め切り"]
      print "までに"
      puts todo["タスク"]
  end
end

def make_task_hash
  puts "【ToDo追加モードを選択しました】"
  print "締め切りを入力してください→"
  deadline = gets.chomp!
  print "タスクを入力してください→"
  task = gets.chomp!
  {"締め切り" => deadline,"タスク" => task}
end

def error
  puts "エラーです．プログラムを終わります"
  exit
end

#これより下がメインの処理
todo_list = []
100.times do
  mode = select_mode
  if mode == "show"
     show_tasks(todo_list)
  elsif mode == "add"
     todo_list.push(make_task_hash)
  else
     error
  end
end
```

「#これより下がメインの処理」と書いてあるところをみてみると，かなり単純になったのではないでしょうか？**最初に作ったToDoリストの構造を**

単純になったのではないでしょうか？ **最初に作ったToDoリスト全体の構造を，プログラムをみるだけで瞬間的に把握できるような形になってると思います.**

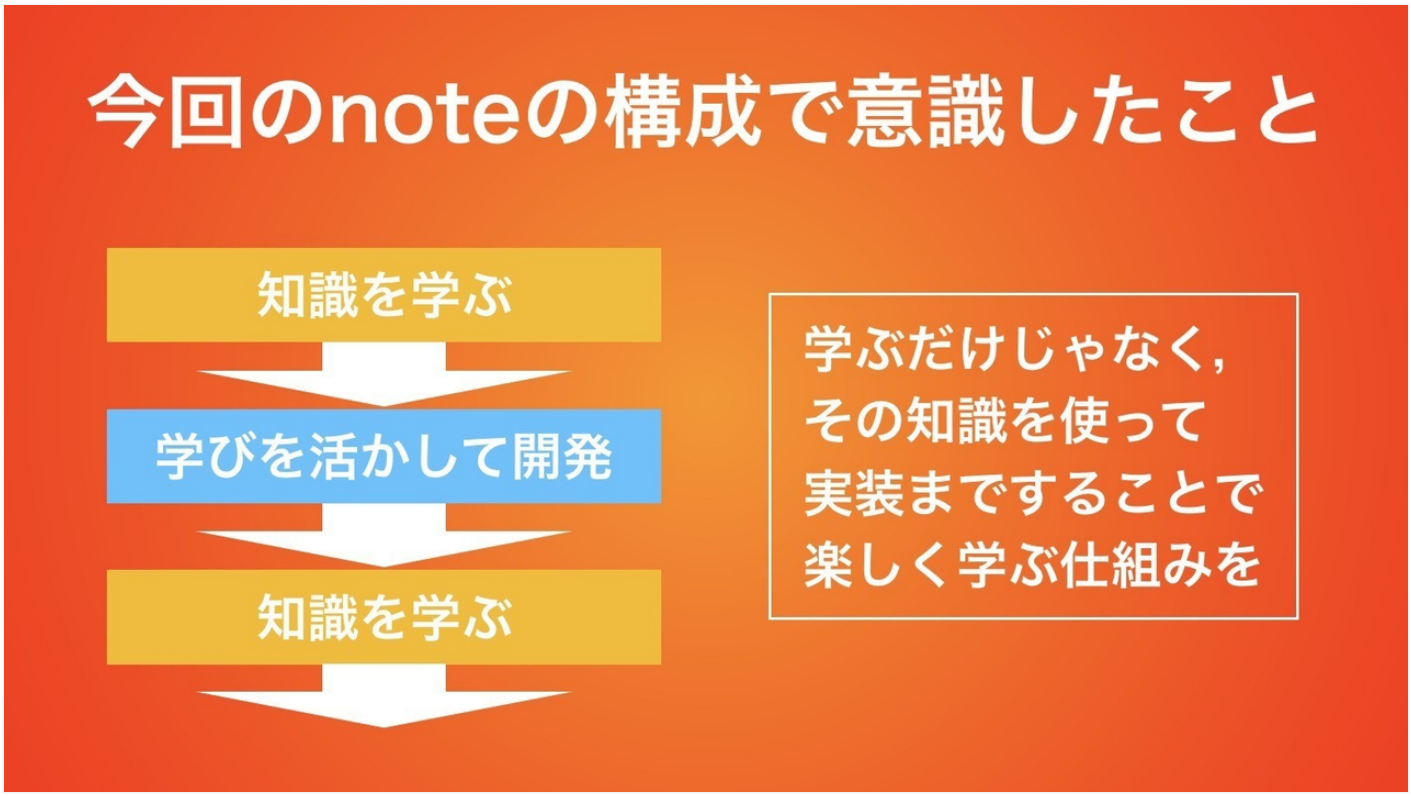


これで，ToDo管理アプリは完成とします，お疲れ様でした！

最後に

今回は，Rubyの基礎文法を学びながら，ToDo管理アプリを作ってみました.

このnoteを作るにあたって，**以下のように『知識』と『実践』を交互に入れて，身につけたスキルを使いながらToDo管理アプリを作っていくような構成**にしました.



また，途中でわざとToDoアプリの使いにくい部分などを残しておき，その後に繰り返し文などを学ぶのに繋げやすいようにしてみました.

今後，プログラミングを学んでいく際も同じで，**何か新しい知識を学んだらそれを使って何か簡単なプログラムを組んでいましょう.** それが知識を定着させるにあたってもっとも近道です.

今回は，簡単なToDoアプリを作ってみましたが，あれをブラウザに描画させればWebアプリケーション化することもできます. これからもプログラミングを続けていってください！

最後になりますが，本noteを最後までお読みいただき，ありがとうございま

した.

#プログラミング #テクノロジー #Ruby #チュートリアル #初心者向け
#TODO #技術記事 #content #count #num #メソッド定義部分def

この記事が気に入ったら、サポートをしてみませんか？気軽にクリエイターを支援できます。

サポートをする

このクリエイターのおすすめノート

【200万達成】ガチで稼いでみる思考過程・ToDoをリアルタイムで公開してみる

♡ 29 Yuki Sako

‹ 大学生の私がブログ収益39万を突破した戦略のすべて

【200万達成】ガチで稼いでみる思考過程・ToDoをリアルタイムで公開... ›

Yuki Sako

学生フリーエンジニアブロガーの迫佑樹です．ブログは月間最高10万PVで，ブログを書籍化した結果Amazonの教育カテゴリで売れ筋ランキング1位を獲得．noteでは厳選した良いコンテンツのみを販売いたします．

Ruby

他 1つ のマガジンに含まれています

フォロー

コメントする...

こちらもおすすめ

Pythonによるスクレイピング①入門編　ブログの記事をCS...

♡ 364　3

Dai

【Pythonによるデータ分析①】Pythonでツイッターのオリ...

♡ 90　0

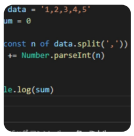
Dai

Illustrator スクリプト入門 01(はじめの一歩編)

♡ 76　0

Seiji Miyazawa

JavaScriptをはじめよう #3 データ操作をしてみよう



♡ 13　0

erukiti

WEBアプリケーションフレームワーク「Django」でブログ...

♡ 47　9

中西瑛太

PythonによるWebスクレイピング②Google検索の結果から...

♡ 58　0

Dai

すべてをjsにまとめる思想を理解する - webpackハンズオ...

♡ 65　0

こんぴゅ

Kotlinでコードを自動生成する

♡ 5　0

NIKKEIスタッフ