慣れないうちはアロー関数が出てくる度にネットでアロー関数の書き方を検索するか頭の中で一度処理を整理している人も多いかと思います。しかし、慣れてしまえばコードを短く記述することが可能なので通常の記述方法を忘れてしまうかもしれません。

アロー関数は矢印(=>)を含んだ関数です。

本文書はアロー関数を利用したmap関数、forEach関数、find関数、filter関数の説明を行っているのでこの機会にアロー関数の書き方と各関数の使用方法も一緒にマスターしましょう。

目次

関数の書き方

JavaScriptで関数を記述する方法はいくつかありますが、一番慣れているのは下記の記述方法ではないでしょか。

```
function add(a, b){
  console.log(a + b)
}
```

下記のように記述することも可能です。

```
let add = function(a, b){
  console.log(a + b)
}
```

どちらで実行しても結果は同じです。

```
add(1,3)
4
```

アロー関数を使った書き方

処理内容は同じですが、先程までの関数の書き方をアロー関数で書くと下記のようになります。

```
let add = (a, b) => {
  console.log(a + b)
}
```

さらにconsole.logのように1行しかない場合は下記のように記述することも可能です。

```
let add = (a, b) => console.log(a + b)
```

add関数では引数がありましたが、引数がない場合は下記のように記述することができます。

```
let hello = () => console.log('Hello World!')
```

アロー関数を使わない場合は、下記の通りです。

```
let hello = function(){
    console.log('Hello World!')
}
```

引数がない場合も関数を実行する際は()を忘れないようにしましょう。

```
hello()
Hello World!
```

returnの省略

returnで戻り値のある場合の関数の書き方について説明します。これまでの流れで下記のように記述できることはわかるかと思います。

```
let add = (a, b) => {
  return a + b
}
```

上記はreturnを省略しても記述することができます。

```
let add = (a, b) => a + b
```

引数が1つしかない場合は、カッコの省略も可能です。

```
let add =  a => a + 20
```

map関数にアロー関数

map関数とは

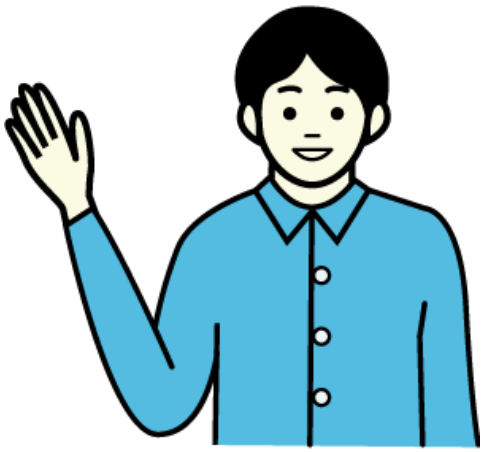
map関数は配列の各要素に関して、個別に操作を行うことができる関数です。言葉で説明するよりも動作させて見たほうが理解は簡単なので、動作確認を行います。

map関数を利用して配列priceに入った数字を1.1倍して新しい配列に入れています。配列の要素それぞれに1.1倍する操作を行っています。**操作が行われた要素は新しい配列の要素となります。**

```
const price = [100,500,1000]

tax_price = price.map(function(value){

  return value*1.1

})
```

```
console.log(tax_price)

>[110, 550, 1100]
```

valueは任意の名前をつけられるのでvalueである必要はありません。map関数でもforEach関数でも同様なのでわかりやすいように任意の名前をつけてください。



map関数でアロー関数

関数で学んだアロー関数を使用すると下記のように短縮することが可能です。

```
tax_price = price.map(value => value*1.1 )
```

map関数ではオプションで配列のindexも表示させることができます。indexをつかって下記のような処理も可能です。

```
const price = [100,500,1000]

tax_price = price.map( (value,index) => value*1.1*index )

console.log(tax_price);

[ 0, 550, 2200 ]
```

forEach関数にアロー関数

forEach関数とは

map同様に配列の各要素に関して、個別に操作を行うことができる関数です。**map関数は戻り値がありますが、forEach関数には戻り値がありません。**forEach関数に戻り値がないということに注目してforEachの動きとmapとの違いを確認しましょう。

戻り値がないため、tax_priceの配列を準備して各要素の処理の中でtax_price配列に値を入れています。

```
const price = [100,500,1000]

tax_price =[]

price.forEach(function(value){

  tax_price.push( value * 1.08)

})
```

forEach関数でアロー関数

map関数と同様の方法で短縮を行うことが可能です。

```
price.forEach(value => tax_price.push( value * 1.08))
```

forEachの要素は関数でも実行可能

JavaScriptの配列の要素には関数を入れることができるので、forEachを実行すると下記のように関数を実行することができます。

```
let hello_world = () => console.log('Hello World')
let hello_javascript = () => console.log('Hello JavaScript')

const hello = [hello_world, hello_javascript]

hello.forEach(value => value())
```

forEach関数をより深く

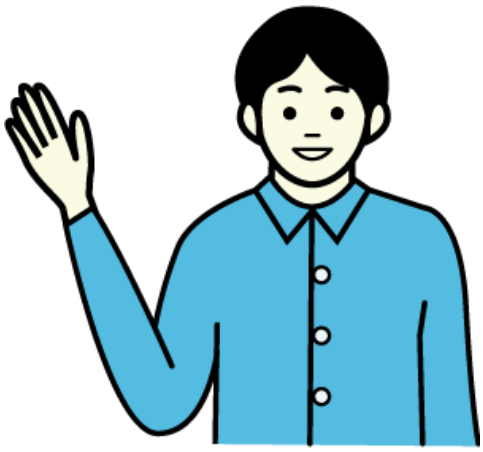
forEach関数は配列の要素だけではなく配列のindexも取得することができます。

```
price.forEach(function(value,index){

  console.log(value + price[index])

})
>200
>1000
>2000
```

indexを使用する場合にアロー関数を利用する下記のようになります。

```
price.forEach((value,index) => console.log(value + price[index]))
```

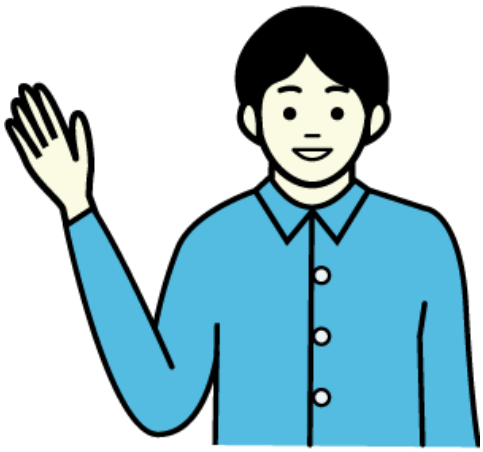
配列には値のみ入ると考えていると上のコードを見た時に何が起きているのはわからないかもしれません。



map関数とforEach関数

戻り値が必要ない下記のような場合は、mapを使ってもforEachを使っても結果は同じです。

map関数は戻り値がありますが、forEach関数には戻り値がありません。



```
price.map(value => console.log(value))

price.forEach(value => console.log(value))
```

find関数でアロー関数

配列の中である条件を満たす要素を見つかる関数です。条件を満たした最初の要素のみ取り出すことができます。

```
const price = [100, 500 ,1000]

over_400 = price.find(function(value){
    return value > 400
})

console.log(over_400)
>400
```

上記の条件は500, 1000で満たすことができますが、取り出すのは条件を最初に満たした500のみです。

アロー関数で記述すると下記のようになります。

```
over_400 = price.find(value => value > 400)
```

filter関数でアロー関数

find関数では条件を満たす1つの要素のみを取り出しましたが、filterでは条件を満たすすべての要素を取り出すことができます。

```
const price = [100, 500 ,1000]

over_400 = price.filter(function(value){
    return value > 400
})

console.log(over_400)
>[500,1000]
```

アロー関数で記述すると下記のようになります。

```
over_400 = price.filter(value => value > 400)
```

reduce関数でアロー関数

reduce関数は利用頻度が少ないかもしれないのでなじみがない人もいるかもしれません。しかしreduce関数が役に立つ場面はあるのでしっかりと理解しておきましょう。

reduce関数の構文は下記の通りです。他のloop関数とは異なり初期値を与えることができます。初期値は省略することも可能です。

配列.reduce(関数[, 初期値])

下記のようにreduce関数は配列の合計などに利用することができます。

```
const test = [10,20,30]

const sum = test.reduce(((pre, cur) => pre + cur),100)

console.log(sum) //160
```

初期値がある場合はpreに100が入りcurにtest配列の最初の10が入ります。100+10の結果が次のpreの値となります。これが1回目のループです。2回目のループはpreには110、curにはtest配列の2番目の値20が入り、110+20の結果が次のpreの値となります。最後の3回目はpreの130とtest配列の3番目の値30が入り160となります。

まとめ

JavaScriptでコードを記述するとどれも同じ頻度で利用する機会があります。特にreduce関数は最初は使いにくいなと感じるかもしれませんがアロー関数と同様に慣れてしまえば簡単に活用することができます。