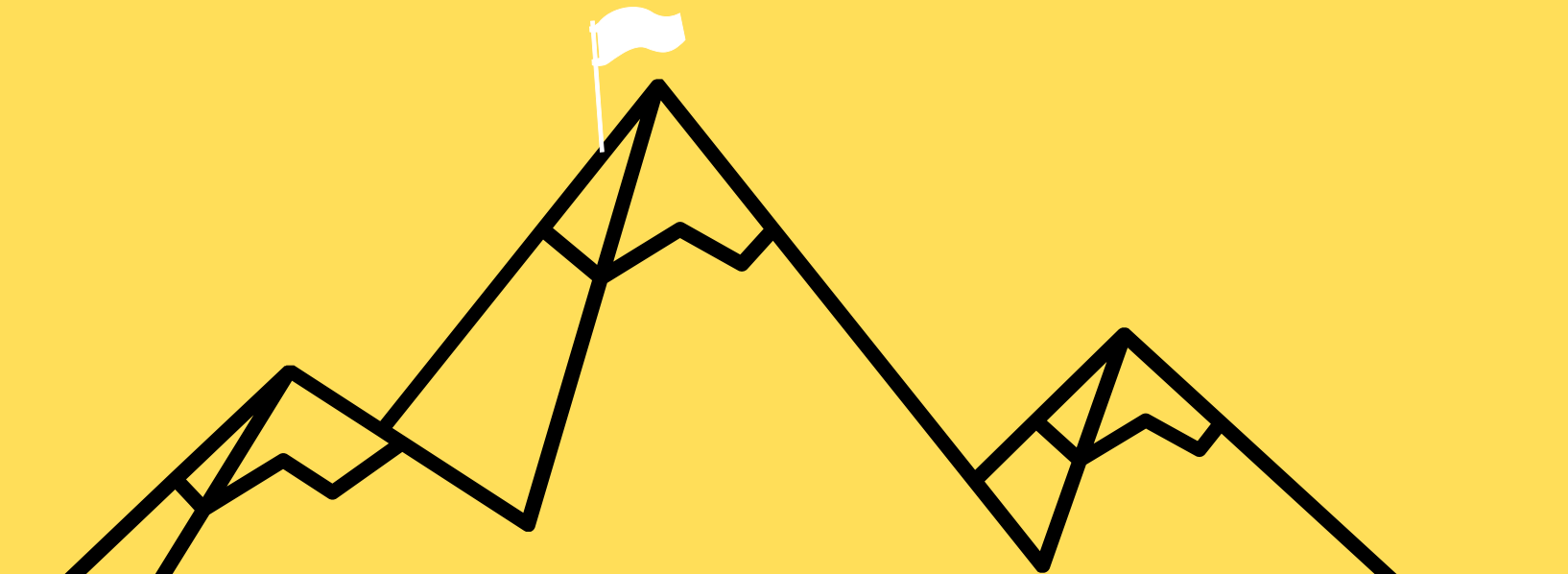
セクションのゴール

- アロー関数が使えるようになる
- 配列の便利メソッドが使えるようになる:
  - forEach
  - map
  - filter
  - find
  - reduce
  - some
  - every



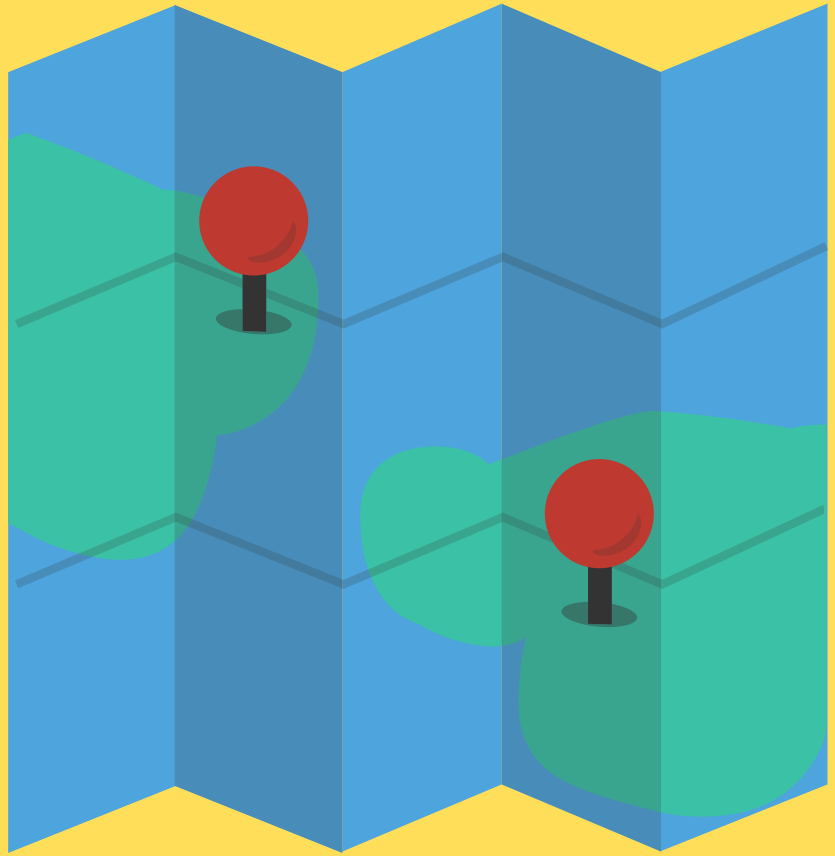
forEach

```
const nums = [9, 8, 7, 6, 5, 4, 3, 2, 1];

nums.forEach(function (n) {
  console.log(n * n)
  // 81, 64, 49, 36, 25, 16, 9, 4, 1
});

nums.forEach(function (el) {
  if (el % 2 === 0) {
    console.log(el)
    // 8, 6, 4, 2
  }
})
```

コールバック関数を受け取り、
配列の要素毎に関数が呼ばれる。

map

与えられた関数を配列のすべての要素に対して呼び出し、その結果からなる新しい配列を生成する



map

```
const texts = ['rofl', 'lol', 'omg', 'ttyl'];
const caps = texts.map(function (t) {
  return t.toUpperCase();
})
texts; //["rofl", "lol", "omg", "ttyl"]
caps; //["ROFL", "LOL", "OMG", "TTYL"]
```

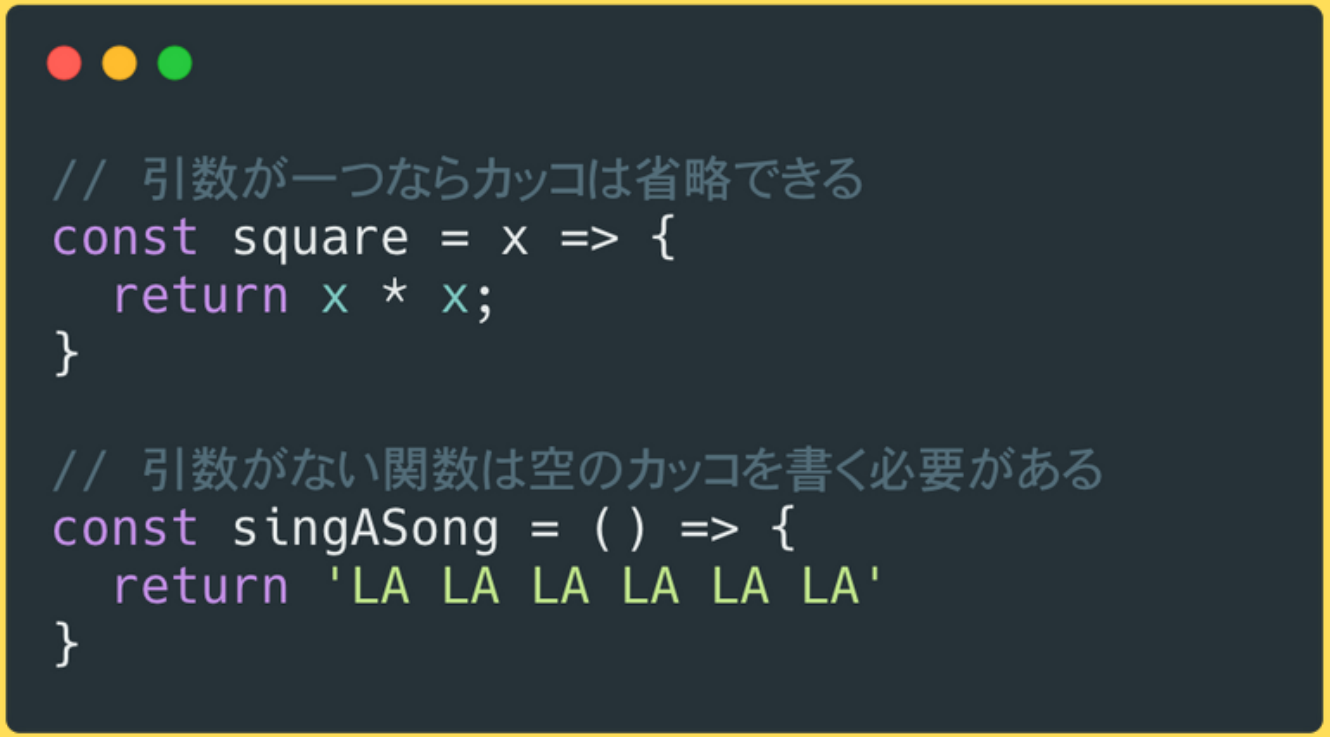
アロー関数

アロー関数

通常の関数式の簡潔な代替構文
（ただし、制限がある）

```
const square = (x) => {
  return x * x;
}

const sum = (x, y) => {
  return x + y;
}
```

アロー関数



```
// 引数が一つならカッコは省略できる
const square = x => {
  return x * x;
}

// 引数がない関数は空のカッコを書く必要がある
const singASong = () => {
  return 'LA LA LA LA LA LA'
}
```

暗黙的なreturn

下の関数はすべて同じ処理を行っている

```
const isEven = function(num) { // 通常の関数式
  return num % 2 === 0;
}
const isEven = (num) => { // 引数にカッコを使ったアロー関数
  return num % 2 === 0;
}
const isEven = num => { // 引数のカッコを省略
  return num % 2 === 0;
}
const isEven = num => ( // 暗黙的return
  num % 2 === 0
);
const isEven = num => num % 2 === 0; // 暗黙的return(1行版)
```

find

提供されたテスト関数を満たす
配列内の最初の要素の値を返す

```
let movies = [
  "The Fantastic Mr. Fox",
  "Mr. and Mrs. Smith",
  "Mrs. Doubtfire",
  "Mr. Deeds"
]
let movie = movies.find(movie => {
  return movie.includes('Mrs.')
}) //"Mr. and Mrs. Smith"

let movie2 = movies.find(m => m.indexOf('Mrs') === 0);
//"Mrs. Doubtfire"
```

filter

提供されたテスト関数を満たす要素からなる新しい配列を生成する

```
const nums = [9, 8, 7, 6, 5, 4, 3, 2, 1];
const odds = nums.filter(n => {
  return n % 2 === 1; // コールバック関数はtrueかfalseを返す
  // trueであればnはfilterされた新しい配列に追加される
});
// [9, 7, 5, 3, 1]

const smallNums = nums.filter(n => n < 5);
// [4, 3, 2, 1]
```

EVERY

配列内のすべての要素が指定されたテスト関数を
満たすかどうかをtrueかfalseで返す

```
const words = ["dog", 'dig', 'log', 'bag', 'wag'];

words.every(word => {
  return word.length === 3;
}) //true

words.every(word => word[0] === 'd'); //false

words.every(w => {
  let last_letter = w[w.length - 1];
  return last_letter === 'g'
}) //true
```

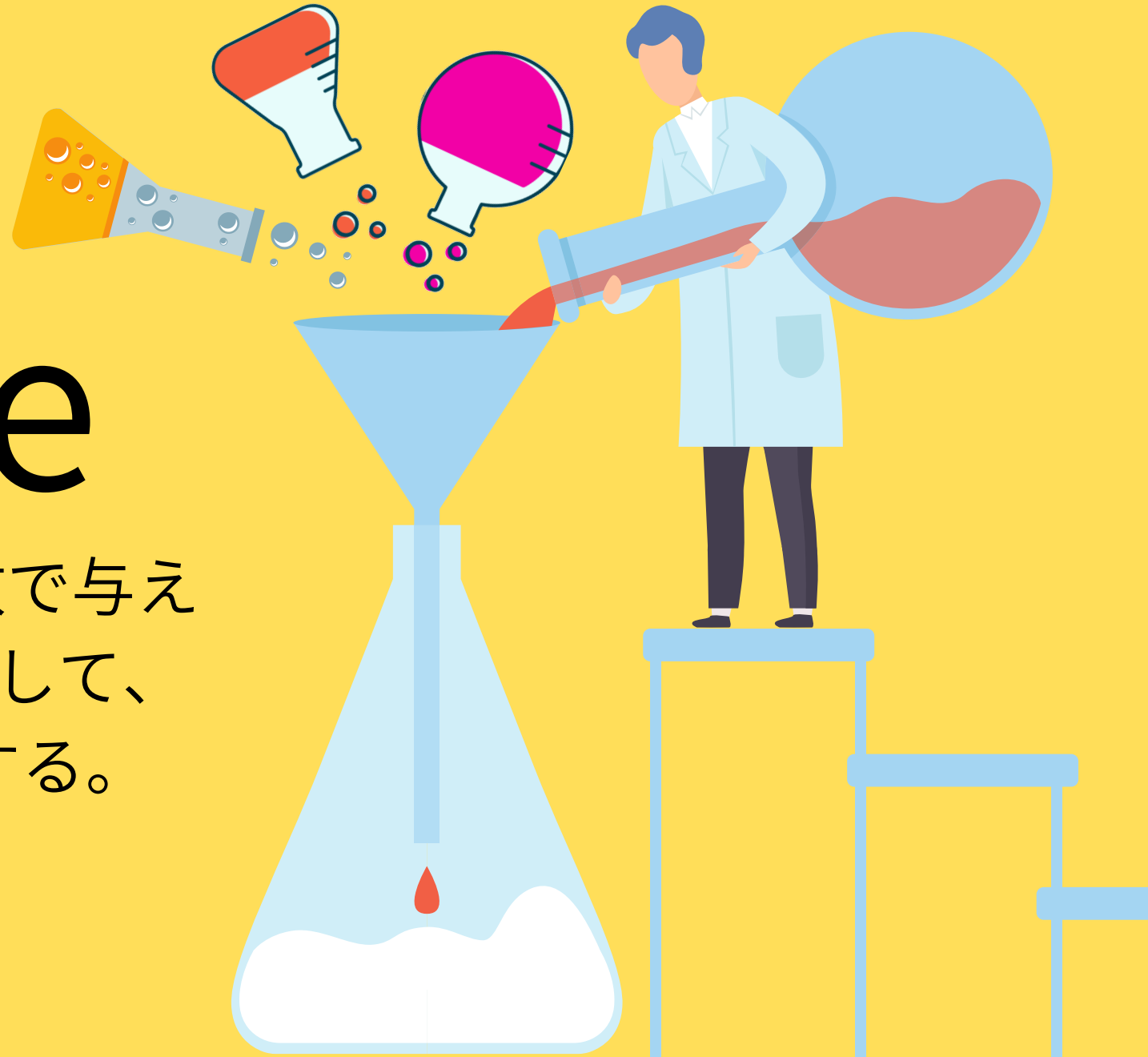
SOME

everyに似ているが、
「一つでも」テスト関数を満たす要素があればtrueを返す

```
const words = ['dog', 'jello', 'log', 'cupcake', 'bag', 'wag'];

// 4文字より多い単語はある?
words.some(word => {
  return word.length > 4;
}); // true

// 'Z'から始まる単語はある?
words.some(word => word[0] === 'Z'); // false

// 'cake'を含む単語はある?
words.some(w => w.includes('cake')); // true
```

reduce

配列の各要素に対して (引数で与えられた) reducer関数を実行して、「単一の出力値」を生成する。



配列内の数字の合計値

```
[3, 5, 7, 9, 11].reduce((accumulator, currentValue) => {
  return accumulator + currentValue;
});
```

Callback	accumulator	currentValue	return value
1回目	3	5	8
2回目	8	7	15
3回目	15	9	24
4回目	24	11	35

最大値を求める

```
let grades = [89, 96, 58, 77, 62, 93, 81, 99, 73];

const topScore = grades.reduce((max, currVal) => {
  if (currVal > max) return currVal;
  return max;
});
topScore; // 99

// Math.maxと暗黙的returnを使ったバージョン
const topScore = grades.reduce((max, currVal) => (
  Math.max(max, currVal)
));
```

初期値

```
[4, 5, 6, 7, 8].reduce((accumulator, currentValue) => {
  return accumulator + currentValue;
});
//RETURNS: 30


[4, 5, 6, 7, 8].reduce((accumulator, currentValue) => {
  return accumulator + currentValue;
}, 100);
//RETURNS: 130
```

集計

```
const votes =
['y','y','n','y','n','y','n','y','n','n','n','y','y'];
const tally = votes.reduce((tally, vote) => {
  tally[vote] = (tally[vote] || 0) + 1;
  return tally;
}, {}); //INITIAL VALUE: {}

tally; //{y: 7, n: 6}
```