JavaScript基礎実習

02. JavaScriptの文法(基礎)

株式会社ジードライブ

今回学ぶこと

- 変数
- 配列
- 分岐処理
- 反復処理
- 関数
- オブジェクト
- クラス

変数

- 変数を使用する際は、まず変数を**宣言**する必要がある
- Javaと異なり、型を明示しない

変数宣言の書式

```
let 変数名;
let 変数名, 変数名, 変数名;
```

※ let の代わりに、**const** や var を用いた記述も存在する

記述例

```
let price;
price = 100;
let item = 'りんご'; // 宣言と同時に値を代入
```

変数：const, let, var

- constは値の再代入ができない
  - letやvarに比べ堅牢なため、基本的にはconstを利用する

```
const x = 100;
x = 200;        // エラー: 再代入不可
const x = 300; // エラー: 再定義不可

let y = 100;
y = 200;        // 問題なし: 再代入可
let y = 300;    // エラー: 再定義不可

var z = 100;
z = 200;        // 問題なし: 再代入可
var z = 300;    // 問題なし: 再定義可
```

変数：const, let, var

- letはvarに比べスコープが狭く扱いやすい
  - let は宣言されたブロック ({ } で囲まれた領域) 内で参照することができる
  - バグの温床になるため、基本的にvarは使用しない

```
{
    var x = 10;
}
console.log(x);
```

10 が表示される

```
{
    let x = 10;
}
console.log(x);
```

変数 x が未定義のため
エラーが発生する

変数とデータ型

- JavaScriptの変数は、型を意識せずに利用することができる

```
let value;
value = 10;
value = 10000000000000000n;
value = 23.5;
value = 'こんにちは';
value = true;
value = ['山田太郎', 32, '東京都'];
value = {name: '山田太郎', age: 32, address: '東京都'};
```

BigInt型(Javaのlong型に相当)

文字列の記号はシングルクォート、ダブルクォートどちらも使用可能

配列:様々な型が混在できる

オブジェクトと呼ばれるデータ

文字列と数値の相互変換

- 文字列 ⇒ 数値
  - Number(文字列) で変換

```
const str = '123.45';
const num = Number(str);
```

- 数値 ⇒ 文字列
  - String(数値) で変換する方法以外にも、
    数値.toFixed(小数点以下の桁数) や 数値.toString(N進数)
    といった方法もある

```
const num = 123.45;
const str1 = String(num);       // '123.45'
const str2 = num.toFixed(1);    // '123.5'  四捨五入される
const str3 = num.toString(10); // '123.45' 10進数表記
```

変数の埋め込み

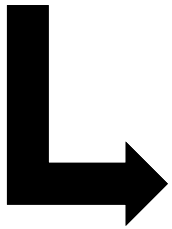
- バッククォート(`)で囲まれた文字列内には、変数を埋め込むことができる
  - バッククォートは、Shift + @ で入力できる
  - 変数は、${ … } 内に記述可能
    ⇒ 計算式も記述できる

```
const name = '山田太郎';
const price = 300;

console.log(`こんにちは${name}さん`);
console.log(`税込み金額は${price * 1.10}円です`);
```

演算子

- Javaと同様の演算子を使うことができる

```
console.log(5 + 3); // 加算
console.log(5 - 3); // 減算
console.log(5 * 3); // 乗算
console.log(5 / 3); // 除算
console.log(8 % 3); // 割り算の余り
console.log('私は' + '山田です'); // 文字列結合
```



コンソール

```
8
2
15
1.6666666666667
2
私は山田です
```

練習

- 練習02-1

配列

- 配列は [ ] を使って表現する
  - Javaと違い、様々なデータ型の値を混在して格納することができる

```
// 配列の作成
const user = ['山田太郎', 25, true];

// 配列の参照
console.log(user[1]);  // 25
user[1] = '東京都';
console.log(user[1]);  // 東京都
```

コンソール

```
25
東京都
```

配列の操作

- 配列内の要素数は固定ではなく、柔軟に扱うことができる
  - unshift(), push() で要素を追加できる
  - Javaと同様にlengthで要素数を取得できる

```
const fruits = ['りんご', 'バナナ', 'ぶどう'];

fruits.unshift('オレンジ'); // 先頭に追加
fruits.push('スイカ'); // 末尾に追加
console.log('フルーツの数:' + fruits.length); // 5
console.log(fruits[0]); // オレンジ
console.log(fruits[fruits.length - 1]); // スイカ
```

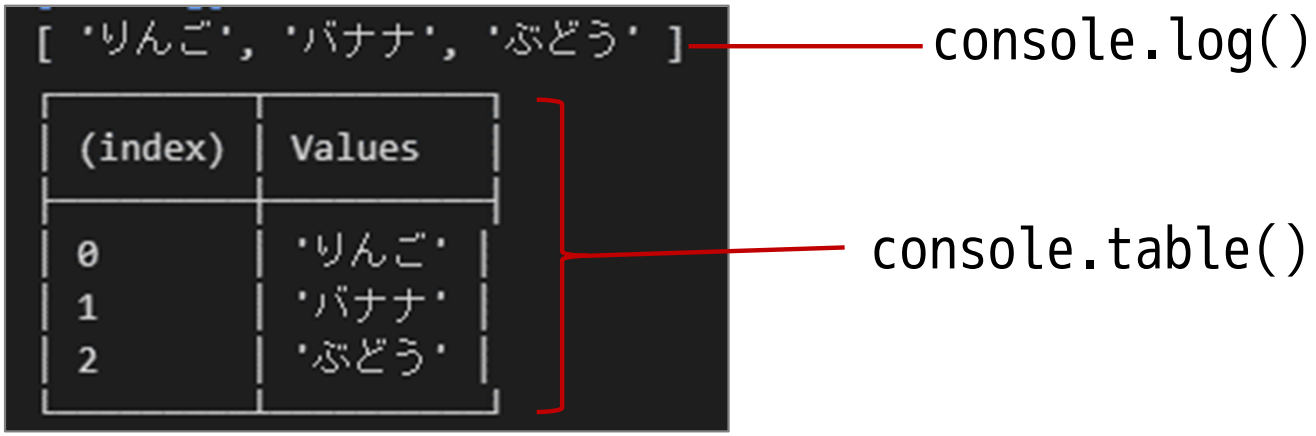
コンソール

```
5
オレンジ
スイカ
```

配列の出力

- 配列やオブジェクト(後述)は、console.table()を使うことで表形式で表示される

```
const fruits = ['りんご', 'バナナ', 'ぶどう'];

console.log(fruits)
console.table(fruits);
```



練習

- 練習02-2

分岐処理

- 分岐処理は、Javaと同様の書式で記述可能

```
if (式1) {
  処理1
} else if (式2) {
  処理2
} else {
  処理3
}
```

```
switch (式) {
  case 値1:
    // 処理1
    break;
  case 値2:
    // 処理2
    break;
  default:
    // 処理3
}
```

三項演算子

- JavaScriptでも三項演算子を使うことができる

```
const lang = 'ja';

console.log(lang === 'ja' ? 'こんにちは' :  'Hello');
```

こんにちは コンソール

関係演算子

- Javaと異なり、文字列も関係演算子で比較する

演算子	意味	例	評価結果
==	等しい	3 == '3'	true
===	厳密に等しい	3 === '3'	false
!=	等しくない	3 != '3'	false
!==	厳密には等しくない	3 !== '3'	true
>	より大きい	3 > 5	false
>=	以上	3 >= 5	false
<	より小さい	3 < 5	true
<=	以下	3 <= 5	true

==と===の違い

- ==は**値が等価**の場合trueとなる
  - 「数値を表す文字列」は数値と等価となる
- ===は**値と型が等しい(厳密等価)**場合trueとなる

例：数値と数字の比較

```
console.log(3 == '3');  // true
console.log(3 === '3'); // false
```

コンソール

```
true
false
```

基本的には === を使用する

falsy / truthy

- if文の条件式など、Boolean(真偽値)が求められる場面で、falseと見なされる値をfalsyな値と呼ぶ

falsyな値：

未定義　Not a Number(非数)

```
0, -0, 0n, 空文字, null, undefined, NaN
```

- 逆に上記以外はtrueと見なされる(truthyな値と呼ばれる)

```
// 変数nameの値が空文字の場合はfalse
// nameに何かしらの文字が入っている場合はtrueになる
if(!name) {
    alert('名前を入力してください');
}
```

練習

- 練習02-3

反復処理

- 反復処理は、Javaと同様の書式で記述可能

```
for（初期処理; 継続条件式; 更新処理）{
  // 何らかの処理
}
```

```
while（継続条件式）{
  // 何らかの処理
}
```

```
do {
  // 何らかの処理
} while（継続条件式);
```

反復処理と配列

- 基本書式はJavaと同じ

```
const fruits = ["りんご", "バナナ", "ぶどう"];
// for(初期処理; 継続条件式; 更新処理) { … }
for(let i = 0; i < fruits.length; i++) {
    console.log(fruits[i]);
}
```

コンソール
りんご
バナナ
ぶどう

- Javaの拡張for文に相当する記法も存在する

```
const fruits = ["りんご", "バナナ", "ぶどう"];
// for(変数 of 配列変数) { … }
for(const item of fruits) {
    console.log(item);
}
```

itemに再代入を行う必要がある場合はlet

コンソール
りんご
バナナ
ぶどう

練習

- 練習02-4

関数

- 関数はJavaのメソッドのようなもの
- 命名規則については変数名の規則と同じ
  - 半角英数字、アンダースコア、ドル記号、日本語を使用可能
  - 名前の先頭は数字以外の文字のみ使用可能

関数定義の書式

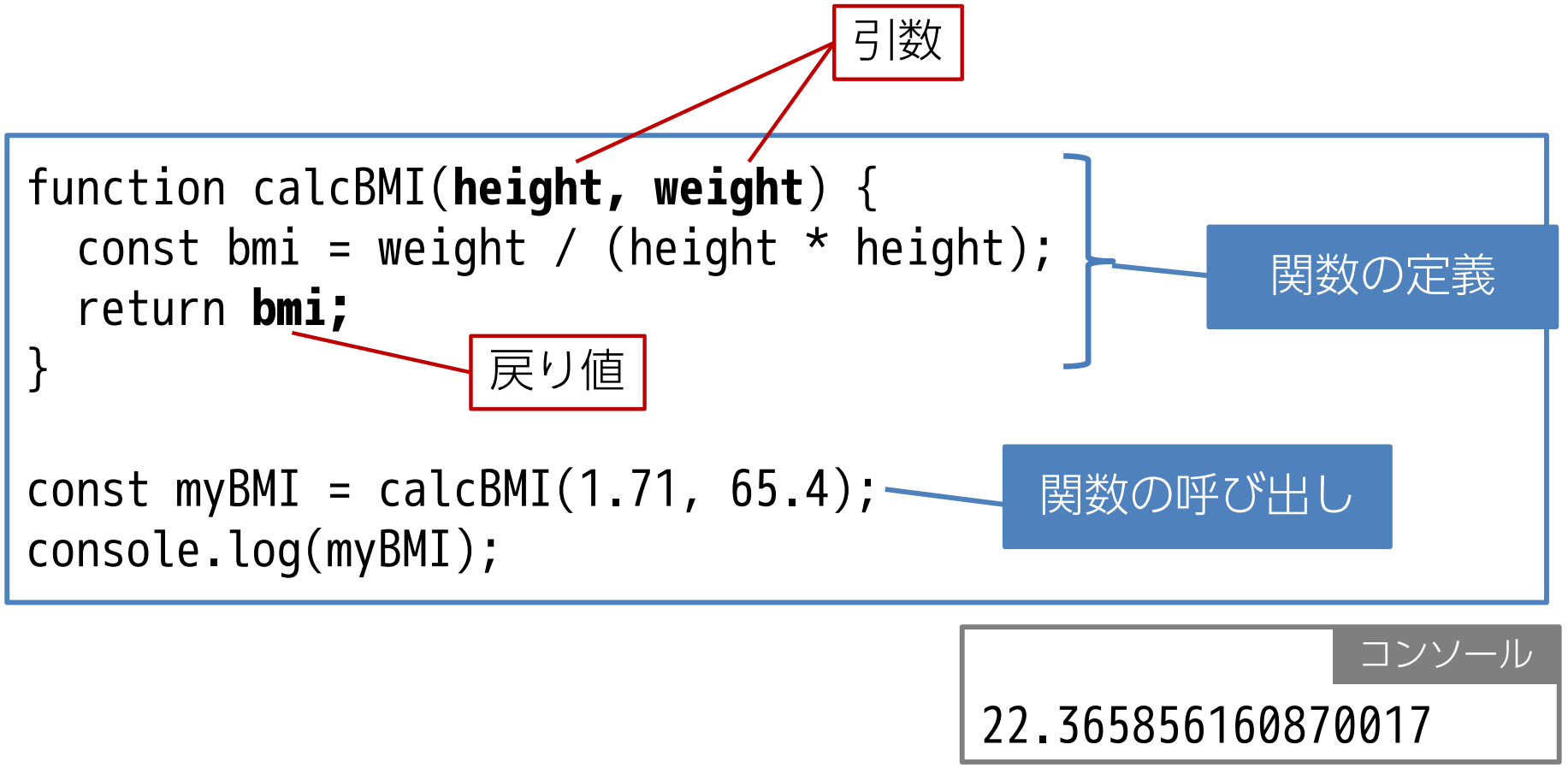
```
function 関数名(引数名, 引数名, … ) {

    // 何らかの処理

    return 戻り値;
}
```

関数の利用 (呼び出し)

```
関数名(引数, 引数, …);
```

関数

- 関数の定義に際しても、データ型を気にしなくてよい



関数

- 引数を変えて、同じ名前の関数を定義する(Javaのオーバーロード)ことはできないが、引数に初期値を設定することは可能

第２引数の初期値を設定

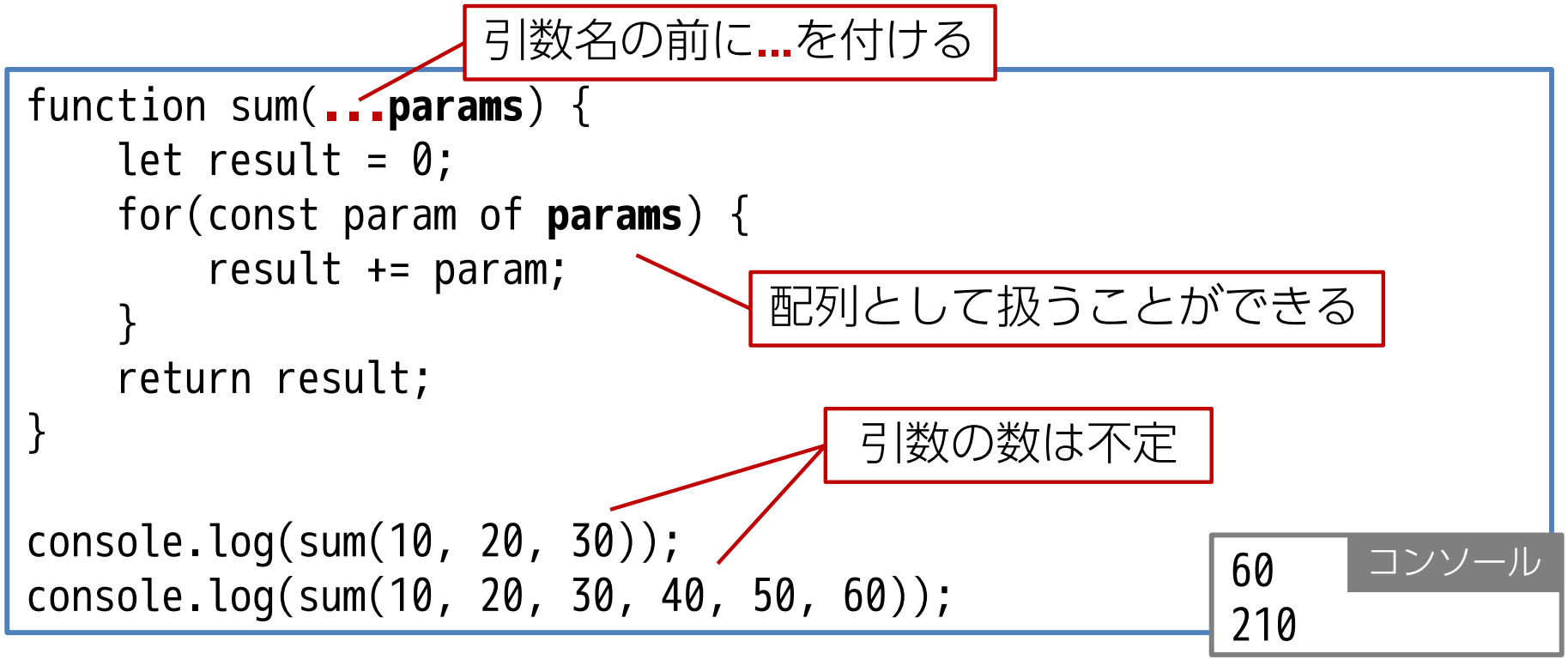
```
function sayHello(name, lang = 'ja') {
  switch(lang) {
     case 'ja' : console.log('こんにちは ' + name); break;
     case 'it' : console.log('Ciao ' + name); break;
     default: console.log('Hello ' + name);
  }
}

sayHello('山田太郎さん'); //こんにちは 山田太郎さん
```

第２引数を省略⇒初期値が使われる

```
こんにちは 山田太郎さん
```
コンソール

残余引数

- 残余引数は不定数の引数を**配列**として受け取ることができる構文
  - 残余引数は、最後の引数として設定する必要がある



練習

- 練習02-5

オブジェクトとは

- JavaScriptのオブジェクトは、キー(名前)とバリュー(値)の組で表されるプロパティの集合体
  - プロパティにはあらゆる種類の値を格納することができる
- オブジェクトは全体を { } で囲んで表現する
  - この書き方をオブジェクトリテラルと呼ぶ

```
// オブジェクトの作成
const item = {
  name: "りんご",
  price: 100
};
```

複数のプロパティがある場合はカンマで区切る

プロパティ名と値はコロンで区切る

メソッド

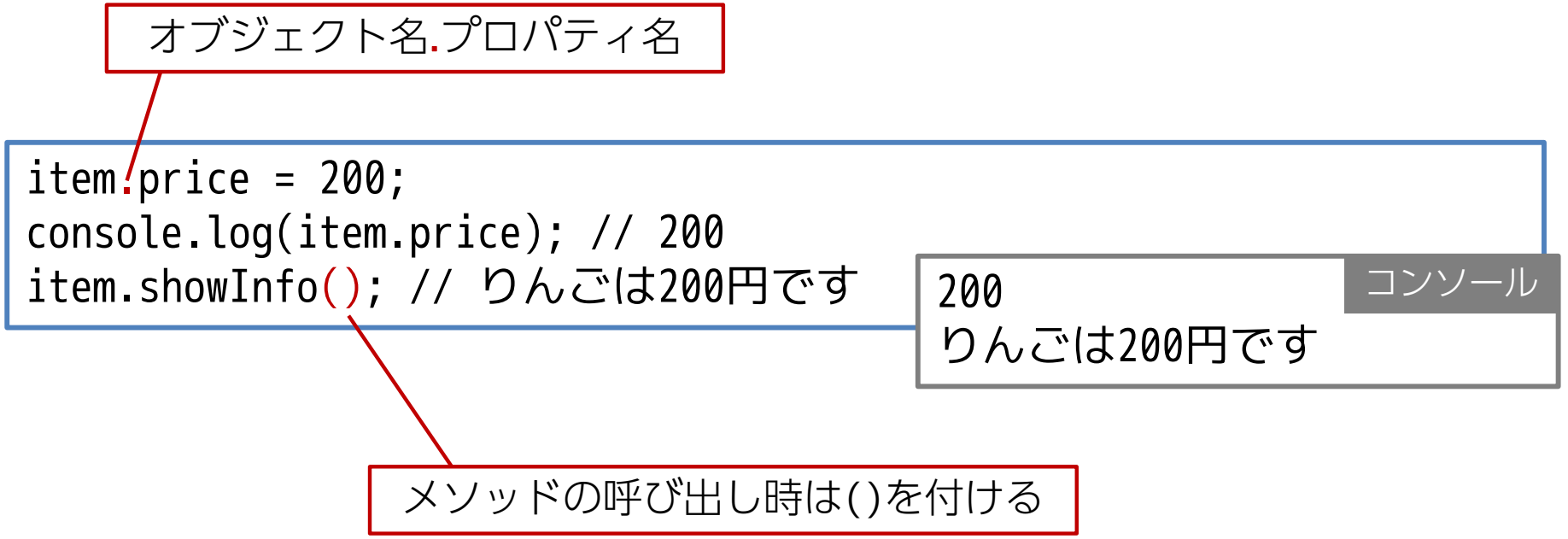
- 関数オブジェクトを格納したプロパティを特に**メソッド**と呼ぶ

```
// メソッドを持つオブジェクトの作成
const item = {
  name: "りんご",
  price: 100,
  showInfo: function() {
    console.log(`${this.name}は${this.price}円です`);
  }
};
```

メソッド

オブジェクトの利用

- 定義したオブジェクトは、オブジェクト名.プロパティ名で利用する



オブジェクトの利用

- ドットの代わりに [ ] を使用するブラケット記法も存在する
  - プロパティ名を動的に設定したい場面で利用する

```
// ドットの代りに [ ] を使うこともできる(ブラケット記法)
item['name'] = 'バナナ';
item['price'] = 100;
item['showInfo']();
```

バナナは100円です コンソール

[ ]内には、プロパティ名を入れる
ブラケット記法は、動的にプロパティ名を設定したい場面で使用する

プロパティやメソッドの追加

- JavaScriptのオブジェクトでは、動的にプロパティやメソッドを追加することができる

```
// userオブジェクトの定義
const user = {
    ...
};

// オブジェクトを生成した後にプロパティやメソッドを追加できる
user.address = "東京都";
user.showAddress = function() {
    console.log(`住所は${this.address}です`);
};
```

オブジェクト内のオブジェクト

- オブジェクトのプロパティとしてオブジェクトを持つことができる

例：jobプロパティとしてオブジェクトを設定

```
const user = {
  name: '山田太郎',
  job: {
    name: "デザイナー",
    company: "ABCデザイン事務所",
    showInfo: function() {
      console.log(`${this.company} [${this.name}]`);
    }
  }
};
```

オブジェクト内のオブジェクト

- オブジェクト内のオブジェクトに対しても、他のプロパティやメソッドと同様、ドットで接続して利用することができる

```
user.job.name = 'ディレクター';
user.job.showInfo();
```

コンソール

```
ABCデザイン事務所 [ディレクター]
```

---

console.log()を使うと、オブジェクトの持つプロパティやメソッドを確認することができる

```
console.log(user);
```

```
{
  name: '山田太郎',
  job: {
    name: 'ディレクター',
    company: 'ABCデザイン事務所',
    showInfo: [Function: showInfo]
  }
}
```

変数を使ったオブジェクトの定義

- オブジェクトのプロパティ値を変数で定義する場面で、プロパティ名と変数名が一致する場合は、省略して記述できる

```
const name = '山田太郎';
const age = 25;

const user = {name, age};
```

```
// 省略しない場合
const user = {
    name: name,
    age: age
};
```

練習

- 練習02-6

コンストラクタ関数

- オブジェクトはコンストラクタ関数を使い、new演算子で生成することができる
  - 関数名は大文字で始める

```
// コンストラクタ関数(プロパティの定義)
function User(name, age) {
    this.name = name;
    this.age = age;
}

// メソッドの定義(prototypeという概念を使う)
User.prototype.showInfo = function() {
    console.log(this.name + '(' + this.age + ')');
}

// コンストラクタ関数からインスタンス(オブジェクト)を生成
const u1 = new User('山田太郎', 32);
const u2 = new User('本田一郎', 28);
```

クラス構文

- コンストラクタ関数の糖衣構文としてクラス構文が存在する

```
class User {
  name;
  age;

  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  showInfo() {
    console.log(this.name + '(' + this.age + ')');
  }
}
```

フィールドの定義(省略可)
⇒ コンストラクタ内で定義すればよい

コンストラクタ:
インスタンス生成時に実行される特別な関数。フィールド(プロパティ)はコンストラクタ内で定義する。

メソッドの定義

プロパティやメソッドにアクセスする際は、thisが必須

インスタンスの生成

- クラスからインスタンス(オブジェクト)を生成するには、new演算子を使用する

```
class User {
  …前頁の定義
}

const u1 = new User("山田太郎", 32);
const u2 = new User("本田一郎", 28);
u1.showInfo();
u2.showInfo();
```

コンソール

```
山田太郎(32)
本田一郎(28)
```

練習

- 練習02-7