第5回

JavaScriptから始めるプログラミング

京都大学工学部情報学科
計算機科学コース3回
KMC2回 drafear

自己紹介

- id
  - drafear(どらふぃあ, どらふぁー)
- 所属
  - 京都大学 工学部 情報学科 計算機科学コース 3回
- 趣味
  - ゲーム(特にパズルゲー), ボドゲ, ボカロ, twitter
- 参加プロジェクト ※青: 新入生プロジェクト
  - これ, 競プロ, ctf, 終焉のC++, coq, 組み合わせ最適化読書会

@drafear

@drafear_ku

@drafear_carryok

@drafear_evolve

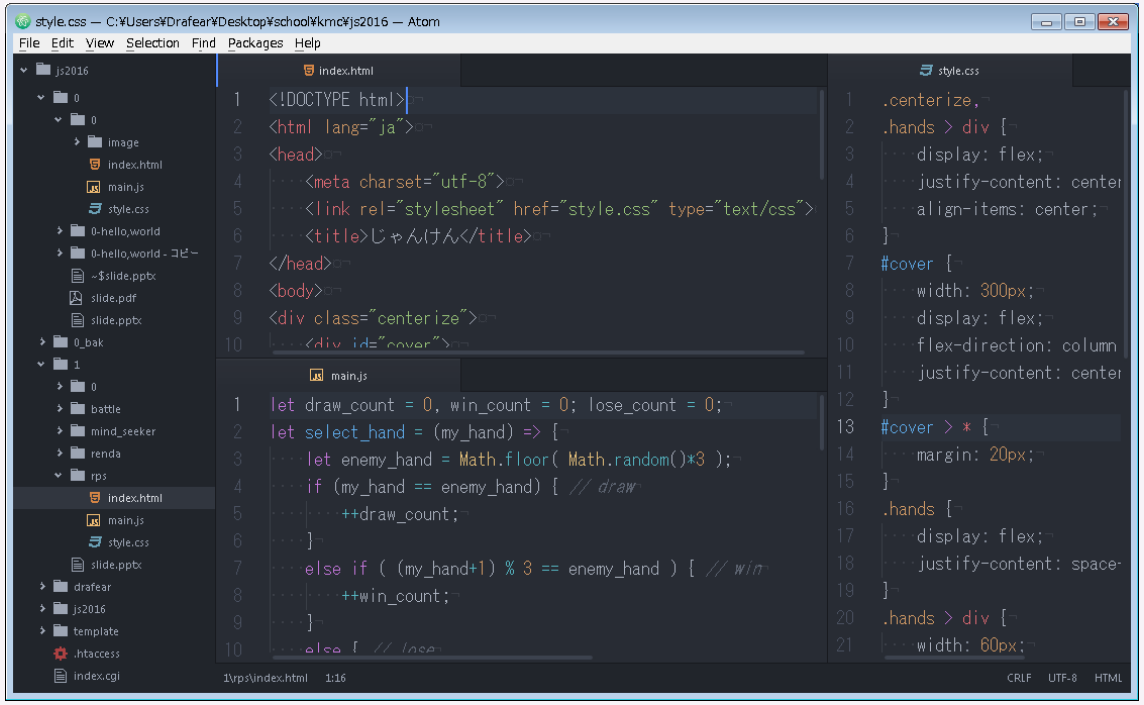
@drafear_sl

@gekimon_1d1a

@cuigames

この講座で使用するブラウザとエディタ

- Google Chrome
  - https://chrome.google.com
- Atom
  - https://atom.io/



今日の目標

- オブジェクト指向を理解する
- 避けゲーを作る

本日の内容

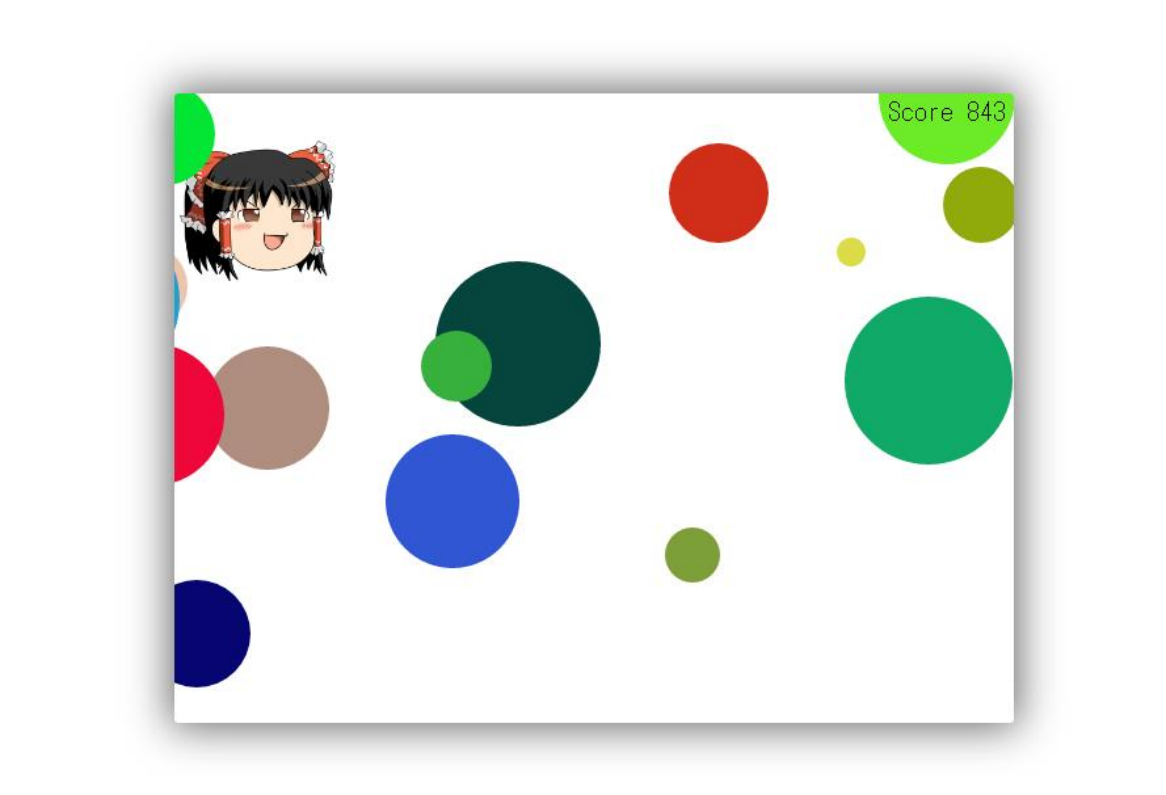
- JavaScript
  - 配列 push, pop, shift, unshift, splice, forEach, concat
  - 配列要素列挙 for … of
  - 定数(const)
  - クラスの基礎
  - getter / setter
  - 'use strict'
- CSS
  - position, top, bottom, left, right, z-index
  - overflow

本日の内容

- DOM操作
  - *elem*.dataset
  - *elem*.style
  - document.createElement
  - *elem*.appendChild, *elem*.removeChild
- その他
  - オブジェクト指向
  - ゲームの基本的構造

今日作るもの

- 避けゲー
  - http://drafear.ie-t.net/js2016/yukkuri/



てんぷれ

- 以下から雛形などをダウンロードしてください
  - https://github.com/kmc-jp/js2016

1. JavaScript

配列

- 値の追加　ary.push(val), ary.unshift(val)
- 値の削除　ary.pop(val), ary.shift(val), ary.splice(pos, num)

push

```
let ary = [100, 300, 200];
ary.push(2); // 末尾に 2 を追加
console.log(ary.length); // 4
console.log(ary); // [100, 300, 200, 2]
```

unshift

```
let ary = [100, 300, 200];
ary.unshift(2); // 先頭に 2 を追加
console.log(ary.length); // 4
console.log(ary); // [2, 100, 300, 200]
```

配列

- 値の追加　ary.push(val), ary.unshift(val)
- 値の削除　ary.pop(val), ary.shift(val), ary.splice(pos, num)

pop

```
let ary = [100, 300, 200];
ary.pop(); // 末尾の要素を削除
console.log(ary.length); // 2
console.log(ary); // [100, 300]
console.log( ary.pop() ); // 300
```

shift

```
let ary = [100, 300, 200];
ary.shift(); // 先頭の要素を削除
console.log(ary.length); // 2
console.log(ary); // [300, 200]
console.log( ary.shift() ); // 300
```

配列

- 値の追加　ary.push(val), ary.unshift(val)
- 値の削除　ary.pop(val), ary.shift(val), ary.splice(pos, num)

splice

```
let ary = [100, 300, 200, "hoge", "hage", 50, 1000];
ary.splice(4, 2); // 0から数えて4番目の要素から 2つ削除
console.log(ary.length); // 5
console.log(ary); // [100, 300, 200, "hoge", 1000]
```

splice

```
let ary = [100, 300, 200, "hoge", "hage", 50, 1000];
ary.splice(3, 1); // ary[3] を削除
console.log(ary.length); // 6
console.log(ary); // [100, 300, 200, "hage", 50, 1000]
console.log( ary.splice(0, 2) ); // [100, 300]
```

配列 – ここまでのまとめ

- 値の読み出し(参照) `ary[pos]`
- 値の書き換え(代入) `ary[pos] = val`
- 要素数 `ary.length`
- 末尾への要素の追加 `ary.push(val)`
- 先頭への要素の追加 `ary.unshift(val)`
- 末尾要素の削除 `ary.pop(val)`
- 先頭要素の削除 `ary.shift(val)`
- 連続する要素の削除 `ary.splice(pos, num)`
- 特定の要素を削除 `ary.splice(pos, 1)`

演習

- 2つの配列を受け取ってそれらを連結した配列を返す関数 concatArray を作ってみよう
  - $let\ concatArray = (ary1, ary2) => \{...\}$
  - 例) concatArray([10, 20], [5, 10]) → [10, 20, 5, 10]

演習

- 2つの配列を受け取ってそれらを連結した配列を返す関数 concatArray を作ってみよう

main.js

```
let concatArray = (ary1, ary2) => {
    let res = [];
    for (let i = 0; i < ary1.length; ++i) {
        res.push(ary1[i]);
    }
    for (let i = 0; i < ary2.length; ++i) {
        res.push(ary2[i]);
    }
    return res;
}
```

オブジェクトの性質

- 関数の引数の配列(やオブジェクト)の中身を変えると
  実は元の配列(やオブジェクト)にも影響する (詳しくは今度)

main.js (あまりよくない例)

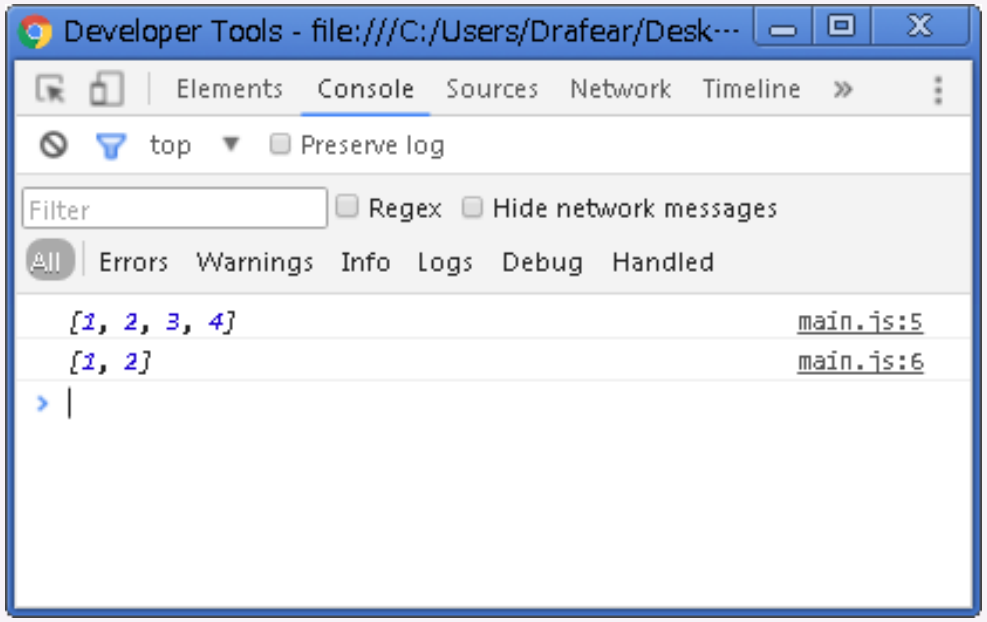
```
let concatArray = (ary1, ary2) => {
    for (let i = 0; i < ary2.length; ++i) {
        ary1.push(ary2[i]);
    }
    return ary1;
}
let a = [5, 10];
let b = [8, 30];
let c = catArray(a, b);
console.log(a); // [5, 10, 8, 30]
console.log(b); // [8, 30]
console.log(c); // [5, 10, 8, 30]
```

実はこんなのあります

- *ary1*.concat(*ary2*)
  - 配列*ary1*に配列*ary2*を連結したものを返す

main.js

```
let concatArray = (ary1, ary2) => {
    return ary1.concat(ary2);
}
let a = [1, 2];
let b = [3, 4];
console.log(concatArray(a, b));
console.log(a);
```



配列とオブジェクト

- 配列の配列やオブジェクトの配列もできる

main.ja

```
let points = [
    { x: 5, y: 3 },
    { x: 10, y: 7 },
    { x: 6, y: -11 },
];
console.log(points[0].x); // 5

let hoge = {
    hage: [1, 2, 3],
    fuga: [{ xxx: 10 }, 20],
};
console.log(hoge.hage[2]); // 3
console.log(hoge.fuga[0].xxx); // 10
```

配列とオブジェクト

- 実は配列はオブジェクトの1つ
- ただし, 配列を使うときはちゃんと [5, 7, 10] と書きましょう

main.js

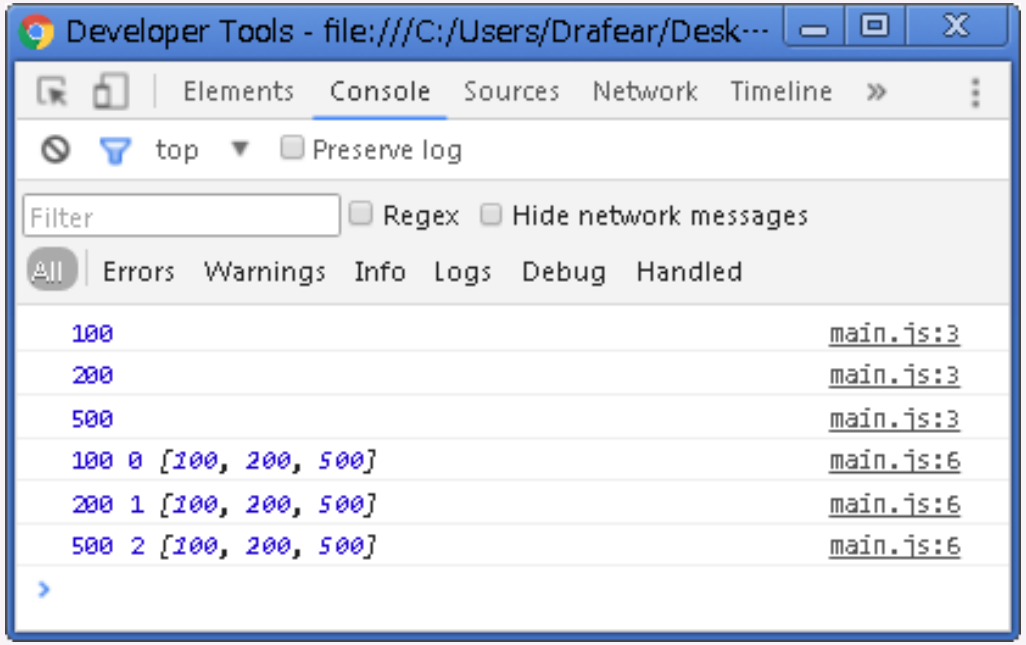
```
let ary = {
    "0": 5,
    "1": 7,
    "2": 10,
    ...
};
```

for ... of

- for (let *item* of *ary*) { ... }
  - 配列*ary*の要素が順にまわる (順に*item*に入っていく)
- *ary*.forEach((*item*, *index*, *self*) => { ... });
  - 配列*ary*の要素が順にまわる (順に関数が呼び出される)
  - *item*: 要素, *index*: 添字番号, *self*: 配列自分自身

main.js

```
let ary = [100, 200, 500];
for (let item of ary) {
    console.log(item);
}
ary.forEach((item, index, self) => {
    console.log(item, index, self);
});
```

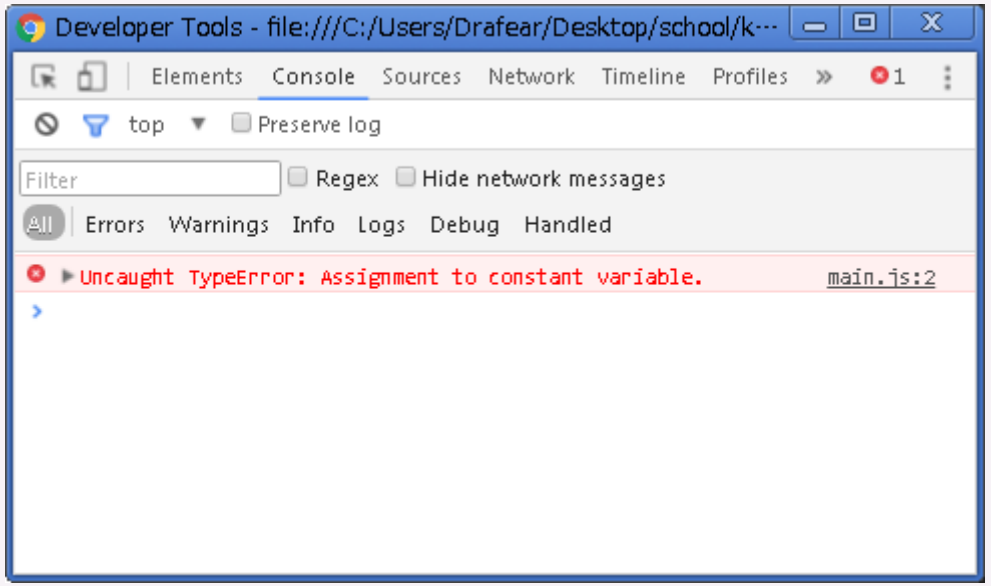


定数

- これまで let で変数定義してきましたが, const でもできます
  - const な変数は再代入不可！

main.js

```
const HOGE = 10;
HOGE = 20; // エラー
```

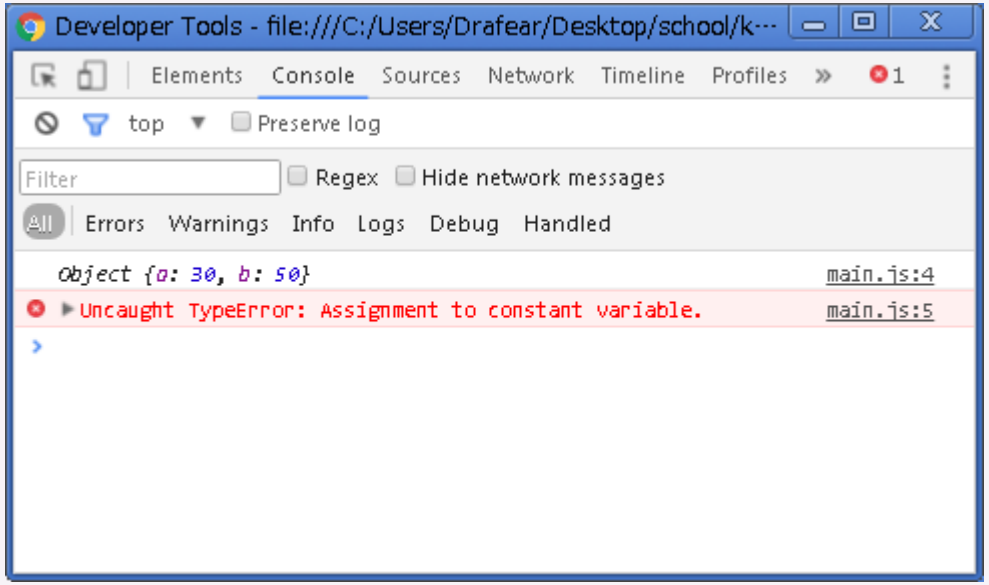


定数

- これまで let で変数定義してきましたが, const でもできます
  - const な変数は再代入不可！
  - 再代入ができないだけで, プロパティ変更はできます

main.js

```
const OBJ = { a: 10, b: 20 };
OBJ.a = 30;
OBJ.b = 50;
console.log(OBJ);
OBJ = { a: 3, b: 5 }; // エラー
```



定数

- let の完全下位互換では？？ let でよくね？？
  - 変更しないはずの変数の場合，ミスをエラーで教えてくれる！
  - 定数や関数には const 修飾子をつけよう！！
  - 「この変数は後から変更されることはないよ！」
    と表明して可読性うｐ！
  - 禁止したいことは禁止する！

const教

- ってのがあるらしい
- 出来る限り const をつけていく！
- const があると安心, ないと不安！

const教

- というわけで, Objectのプロパティにも const 付けたいですよね
- できます！！！！！！！！！！

const教

- というわけで, Objectのプロパティにも const 付けたいですよね
- できます！！！！！！！！！！
- が、この講座では触れないことにします
- 興味がある方, const教への入信を希望される方は defineProperty で検索！
- 今回は const に慣れよう！！

2. オブジェクト指向入門

オブジェクト指向とは？

- オブジェクト同士が互いに
  "相手の中身の細かい実装などを知らずに"
  やり取りする

どういうことだってばよ？

- 例えばGoogleで検索する時に,
「検索ワードを入力する」と「検索結果が表示される」

- Google検索システムの中の仕組みが
どうなっているのか分からなくても使える！

- Google検索システムは1つのオブジェクトであり,
あなた自身も1つのオブジェクトであり
互いにやり取りしている！！

カプセル化

- 中身を見せずに表面のインターフェースだけを与える考え方をカプセル化という

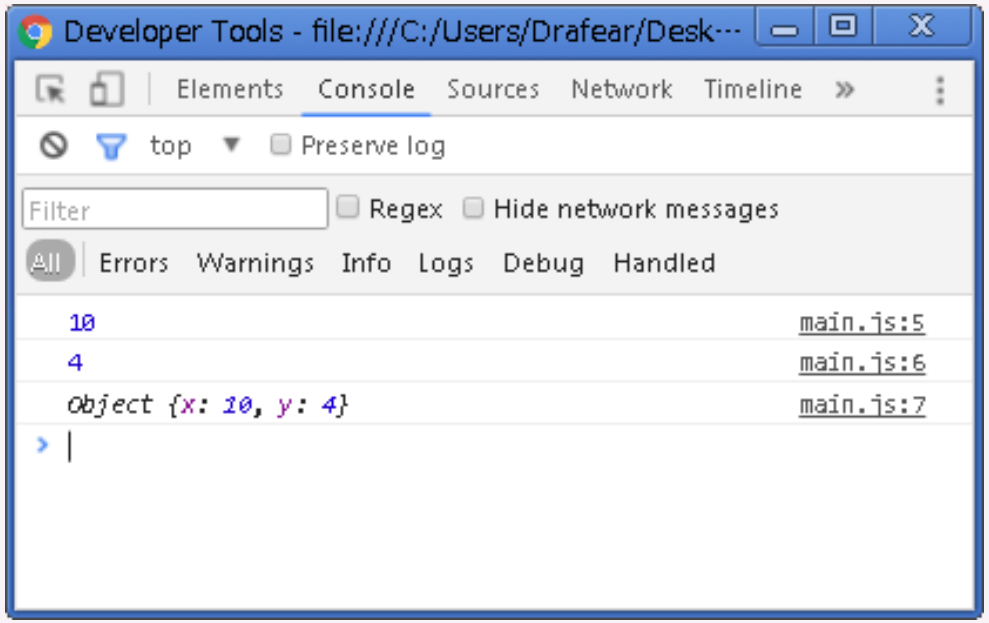
- では実際にJavaScriptで触っていきましょう

Objectを作る？？

- ベクトルを表すオブジェクトを作って返す関数makeVector

main.js

```
const makeVector = (x, y) => {
    return { x: x, y: y };
}
const v = makeVector(10, 4);
console.log(v.x);
console.log(v.y);
console.log(v);
```

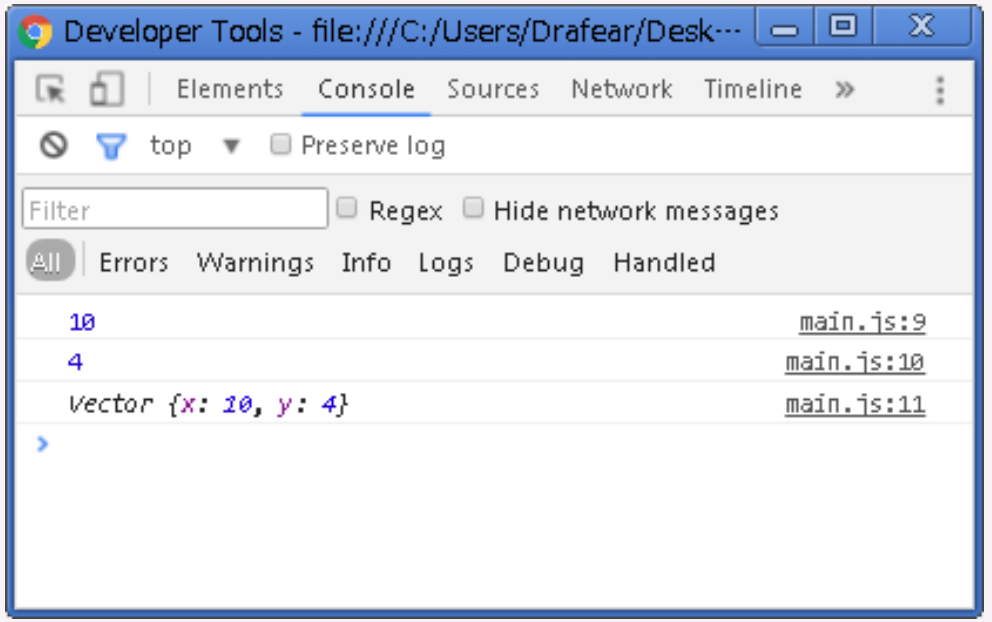


「class」を使って書いてみよう

- classは1つの型を表す (例: ベクトルは x成分 と y成分 を持つ)
- new class名(引数) でそのクラスのモノを1つ生成する
  - インスタンス化という
- 生成する際に, constructor という (見た目は異なるが)関数 が呼ばれる

main.js

```
class Vector {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
}
const v = new Vector(10, 4);
console.log(v.x);
console.log(v.y);
console.log(v);
```

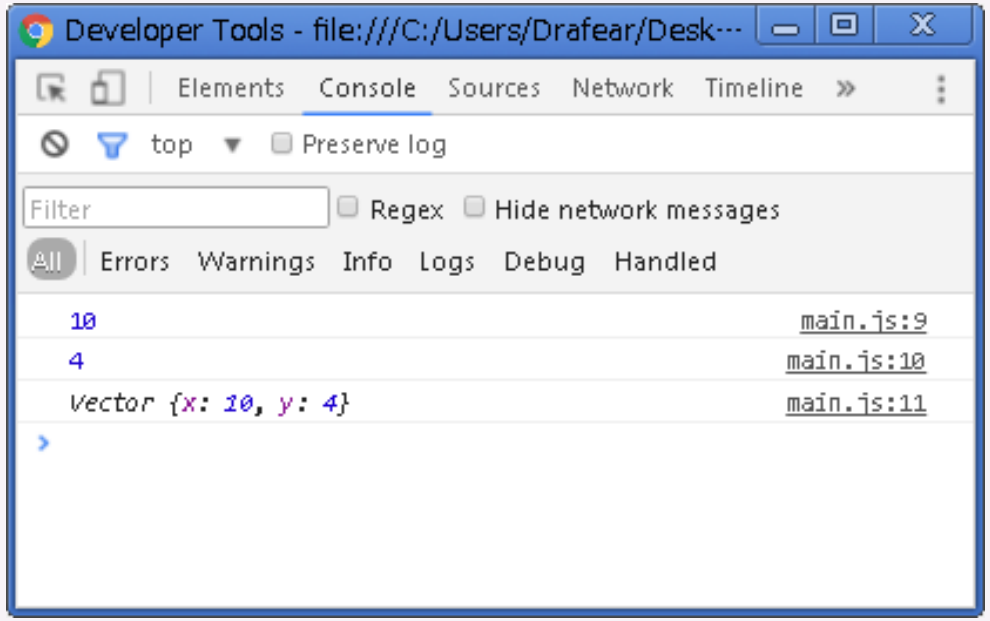


「class」を使って書いてみよう

- クラスのインスタンスが持つ変数(下の例の場合xとy)を メンバ変数 といい, クラス内の関数から this.メンバ変数名でアクセスする
  - 外からは インスタンスオブジェクト名.メンバ変数名

main.js

```
class Vector {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
}
const v = new Vector(10, 4);
console.log(v.x);
console.log(v.y);
console.log(v);
```

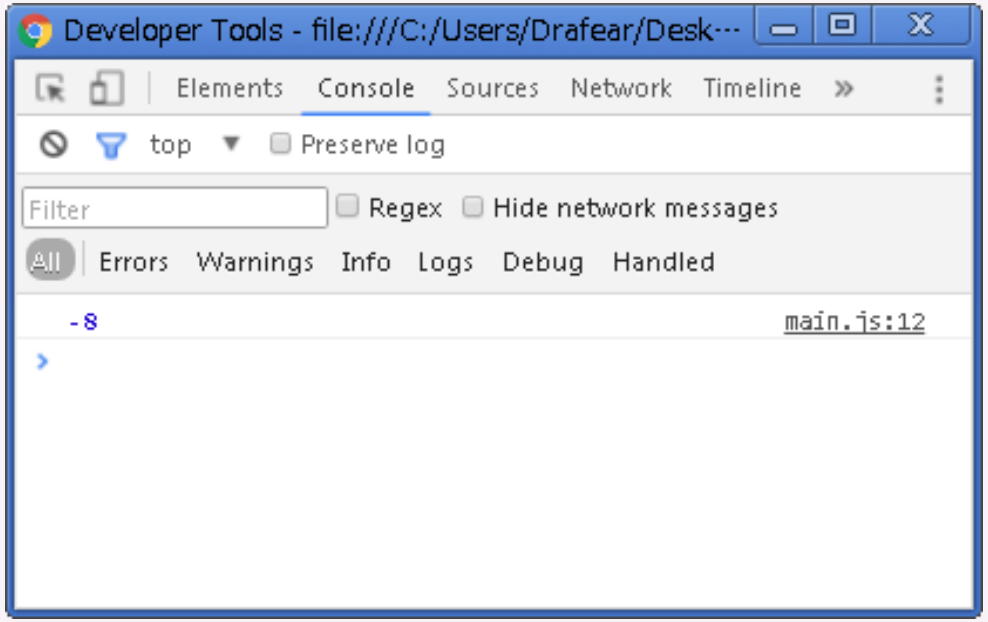


「class」を使って書いてみよう

- クラスは関数を持つことができ，
  メンバ関数 または メソッド という

main.js

```
class Vector {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
    dot(that) {
        return this.x * that.x + this.y * that.y;
    }
}
const v1 = new Vector(10, 4);
const v2 = new Vector(-2, 3);
console.log( v1.dot(v2) );
```

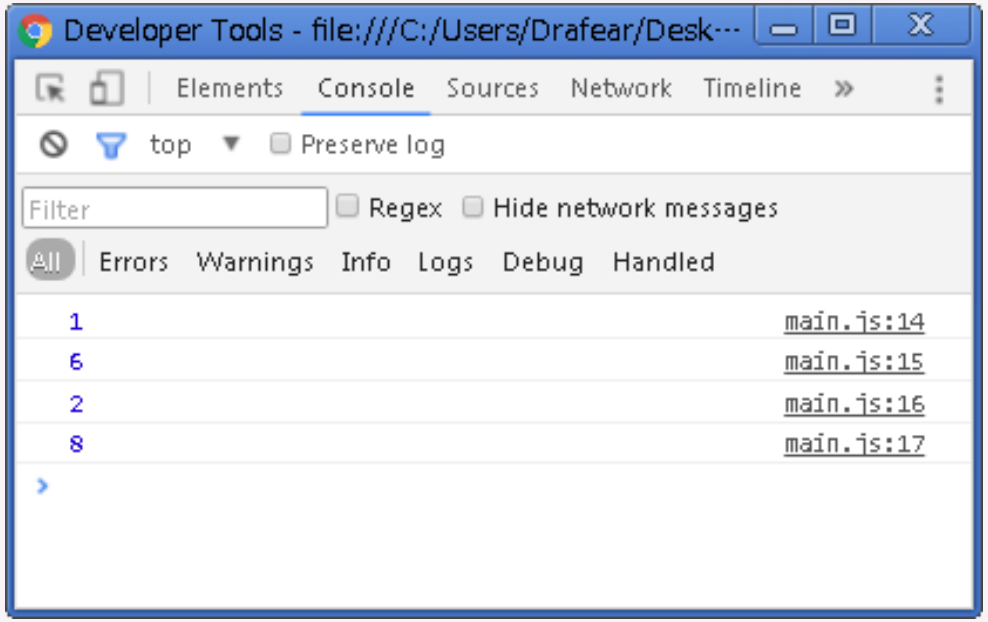


長方形クラス

- 横幅, 縦幅, 左下座標 から 右上座標 を計算する例

main.js

```
class Rect {
    constructor(x, y, w, h) {
        this.x = x;
        this.y = y;
        this.w = w;
        this.h = h;
    }
    getLeft() { return this.x; }
    getRight() { return this.x + this.w; }
    getTop() { return this.y; }
    getBottom() { return this.y + this.h; }
}
const rect = new Rect(1, 2, 5, 6);
console.log( rect.getLeft() );
console.log( rect.getRight() );
console.log( rect.getTop() );
console.log( rect.getBottom() );
```

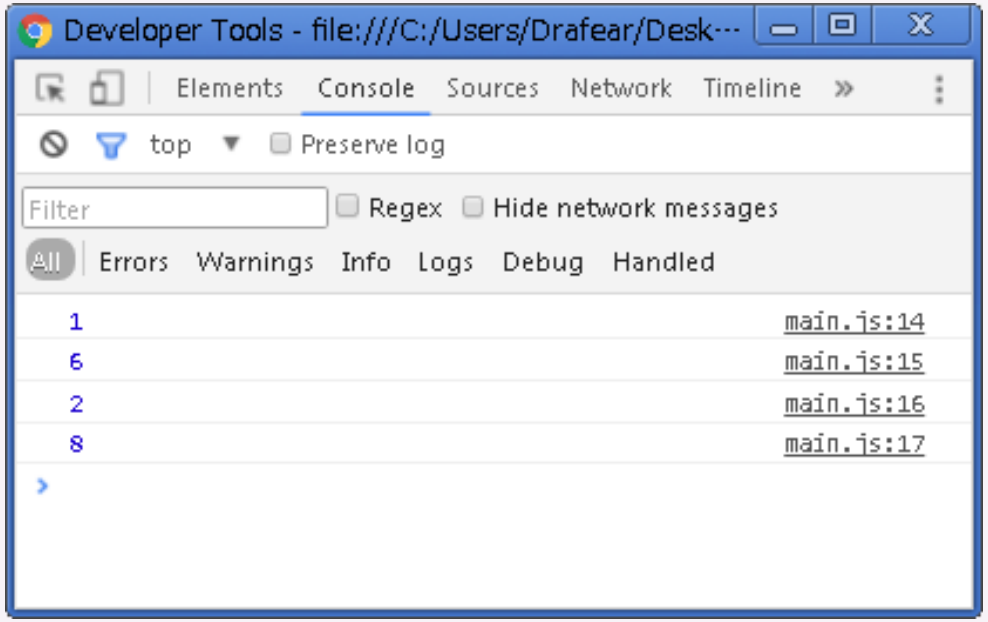


getter / setter (そこそこマニアック)

- getLeft() の括弧をとりたい！ → getter (読み取り専用)

main.js

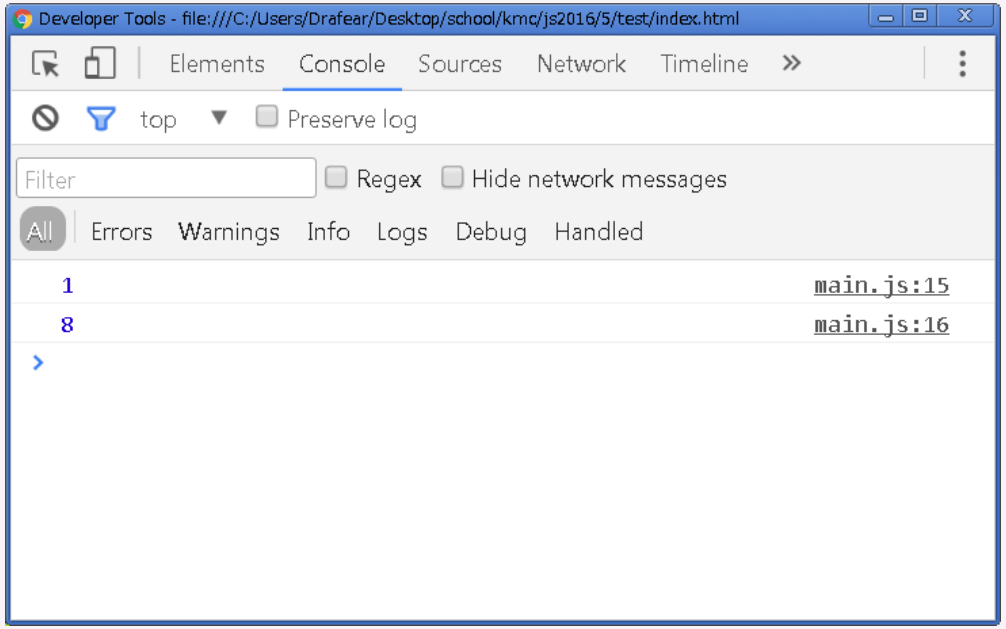
```
class Rect {
    constructor(x, y, w, h) {
        this.x = x;
        this.y = y;
        this.w = w;
        this.h = h;
    }
    get left() { return this.x; }
    get right() { return this.x + this.w; }
    get top() { return this.y; }
    get bottom() { return this.y + this.h; }
}
const rect = new Rect(1, 2, 5, 6);
console.log( rect.left );
console.log( rect.right );
console.log( rect.top );
console.log( rect.bottom );
```



getter / setter (そこそこマニアック)

- getter ・・・ 値の取得だけできる (参照時に呼び出される)

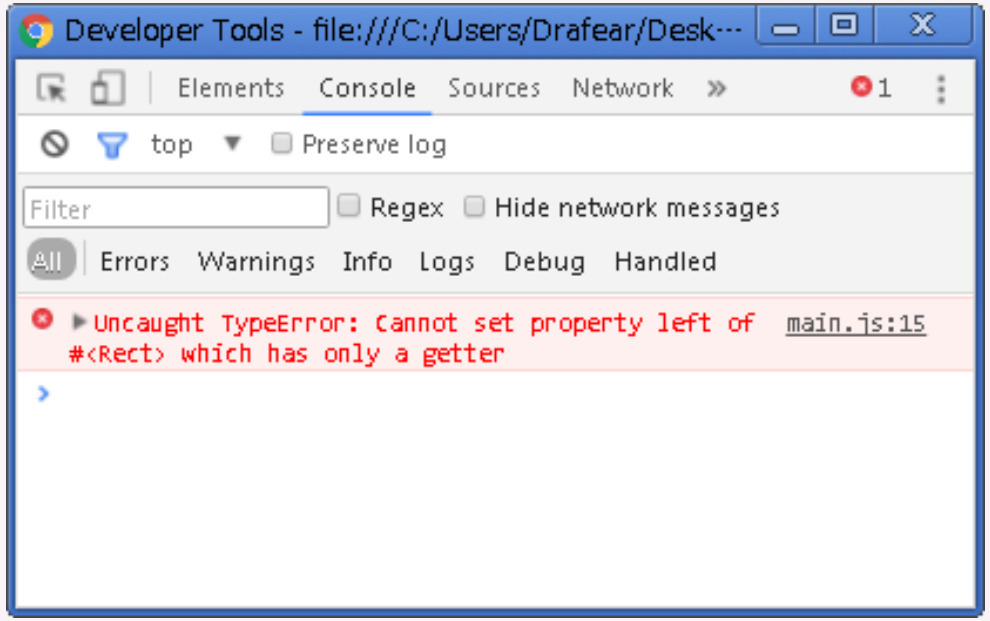
```
class Rect {
    constructor(x, y, w, h) {
        this._x = x;
        this._y = y;
        this._w = w;
        this._h = h;
    }
    get left() { return this._x; }
    get right() { return this._x + this._w; }
    get top() { return this._y; }
    get bottom() { return this._y + this._h; }
}
const rect = new Rect(1, 2, 5, 6);
rect.left = 0; // 無視される
console.log( rect.left ); // 1
console.log( rect.bottom ); // 8
```



'use strict' (これは重要！)

- 'use strict'
  - 文法などを厳しくチェック！ミスったらエラー！！今後から付けよう！

```
'use strict'
class Rect {
    constructor(x, y, w, h) {
        this.x = x;
        this.y = y;
        this.w = w;
        this.h = h;
    }
    get left() { return this._x; }
    get right() { return this._x + this._w; }
    get top() { return this._y; }
    get bottom() { return this._y + this._h; }
}
const rect = new Rect(1, 2, 5, 6);
rect.left = 0; // エラー
console.log( rect.left );
console.log( rect.bottom );
```



getter / setter (そこそこマニアック)

- setter ・・・ 値のセットだけできる (代入時に呼び出される)
  - 値がセットされたらHTML要素にも反映したり

main.js

```
'use strict'
class Game {
    constructor() {
        this._score = 0;
    }
    get score() { return this._score; }
    set score(val) {
        this._score = val;
        document.getElementById("score").innerText = this._score;
    }
}
const game = new Game();
game.score = 100;
```

カプセル化

- カプセル化
  - 外からのメンバ変数へのアクセスには必ず
    メソッド (やgetter, setter) を介してアクセス

カプセル化の例

```
class Rect {
    constructor(x, y, w, h) {
        this.x = x;
        this.y = y;
        this.w = w;
        this.h = h;
    }
    getLeft() { return this.x; }
    getRight() { return this.x + this.w; }
    getTop() { return this.y; }
    getBottom() { return this.y + this.h; }
    getWidth() { return this.w; }
    getHeight() { return this.h; }
}
```

カプセル化

- 例えば, Player に対して, 次のように命令すると動くといったオブジェクトを作ることができる
  - equip(weaponId) ・・・ 武器を装備
  - heal(val) ・・・ HPをval回復
  - damage(val) ・・・ valダメージを受ける
  - attack(enemy) ・・・ enemyに攻撃
  - isAlive() ・・・ 生きているか
  - move(direction) ・・・ directionの方向に移動
- 内部はどうなってるかわからないけど外から命令すれば動く

カプセル化

- 外部に提供する機能と内部実装を分離することで
  後から変更を加えやすくなる
- 外部からアクセスされる部分を変更してしまうと
  全体に影響が及ぶ (バグの原因ともなる)
- どこが外部からアクセスされるかが分かっていると
  変更しやすい！ (クラス内部の実装の変更だけにとどまる)
- そのために, 予めルールを定める！

カプセル化

- 例えば「クラスの外部からは必ずメソッドを介してアクセス」といったルールに従うと
  - メンバ変数の追加や削除, 意味の変更は後からいくらでもして良い
    (ただし各メソッドの実装も適宜変更する必要がある)
  - メソッドの削除や意味の変更はNG
  - メソッドの追加はOK

カプセル化

- でもクラス内からしかアクセスしてはいけないような
  メソッドも欲しいことがある

- そこで、外からアクセスNGのメソッド名の先頭に
  _ (アンダーバー) を付けて区別する, といった規則も設ける

- 他の言語には, 単なるルールではなく,
  プログラム的にそういった規則を付したりできる言語もある
  (間違ったアクセスをするとエラー)

カプセル化

main.js

```
class Rect {
    constructor(x, y, w, h) {
        this._init(x, y, w, h);
    }
    _init(x, y, w, h) {
        this.left = x;
        this.right = x+w;
        ...
    }
    getLeft() { return this.left; }
    getRight() { return this.right; }
    getTop() { return this.top; }
    ...
}
```

外から呼べるけど
呼んじゃダメ！

カプセル化

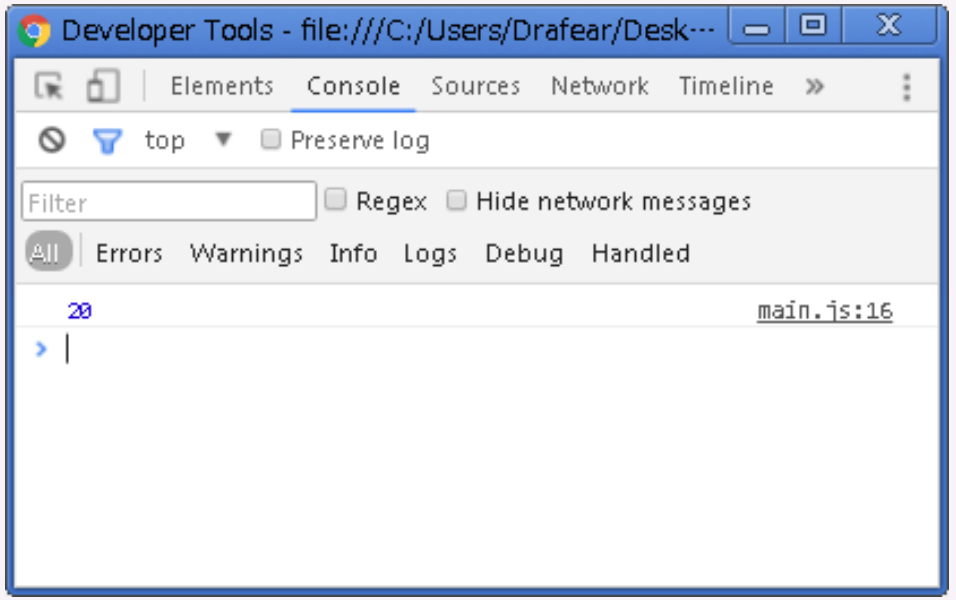
- でも, ちゃんとしたカプセル化は難しいので,
  今回は次のルールとする
  - 外からメンバ変数を書き換え不可. 読み取りはOK.
  - _ から始まるメソッドを外から呼んじゃダメ！
  - _ から始まらないメソッドは呼んでOK！

valueOf (かなりマニアック)

- valueOf ・・・ 演算されたときに演算できるように値変換する

main.js

```
'use strict'
class Score {
    constructor() {
        this._val = 0;
    }
    incr() { ++this._val; }
    valueOf() { return this._val; }
}
const score = new Score();
score.incr();
score.incr();
console.log(score * 10);
```

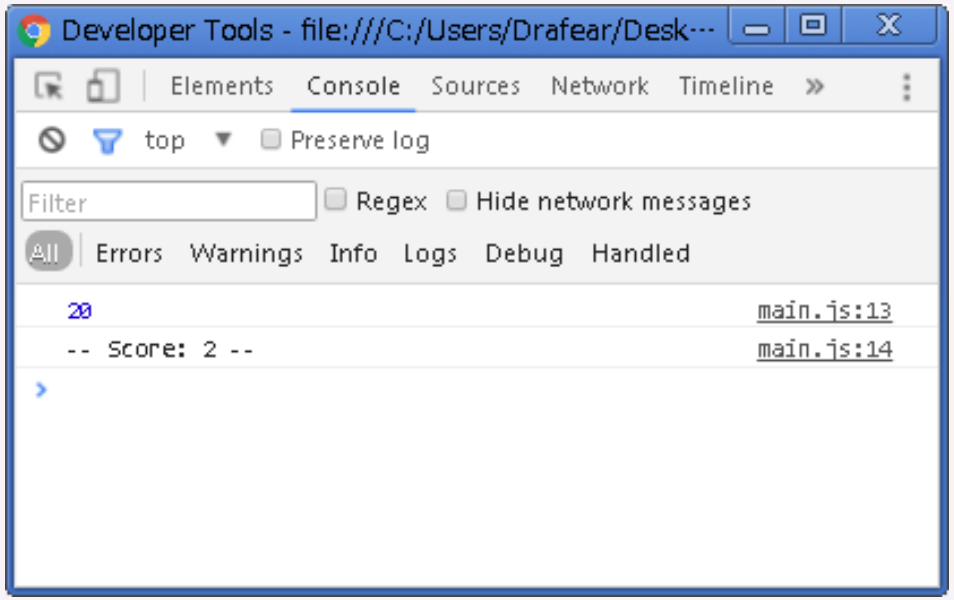


toString (かなりマニアック)

- toString ・・・ 文字列になってくれ！ってときになってくれる

main.js

```
'use strict'
class Score {
    constructor() {
        this._val = 0;
    }
    incr() { ++this._val; }
    valueOf() { return this._val; }
    toString() { return `Score: ${this._val}`; }
}
const score = new Score();
score.incr();
score.incr();
console.log(score * 10);
console.log(`-- ${score} --`);
```

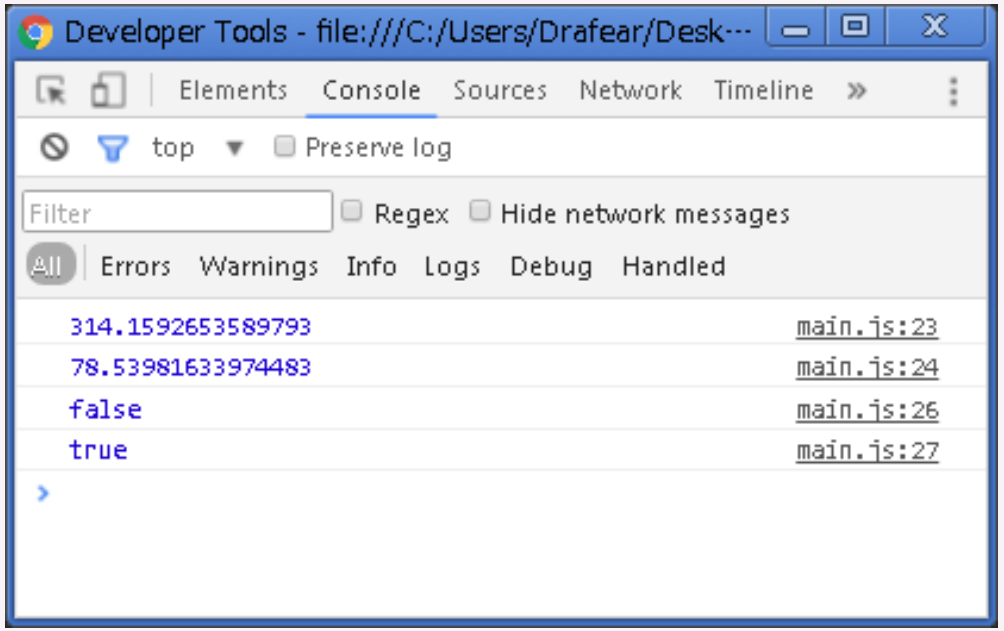


演習

- 円を表すクラスを作ってみよう
  - class Circle { … }

main.js

```
// new Circle(centerX, centerY, radius)
const c1 = new Circle(5, 0, 10);
const c2 = new Circle(5, 20, 5);
const c3 = new Circle(5, -10, 6);
// 面積
console.log( c1.area() );
console.log( c2.area() );
// 共通部分があるか(当たり判定)
console.log( c1.isHit(c2) );
console.log( c1.isHit(c3) );
```



演習

main.js

```
'use strict'
class Circle {
    constructor(x, y, r) {
        this.x = x;
        this.y = y;
        this.r = r;
    }
    area() {
        return Math.PI * this.r * this.r;
    }
    isHit(that) {
        const dx = this.x - that.x;
        const dy = this.y - that.y;
        const r = this.r + that.r;
        return dx * dx + dy * dy < r * r;
    }
}
```

3. CSS

position

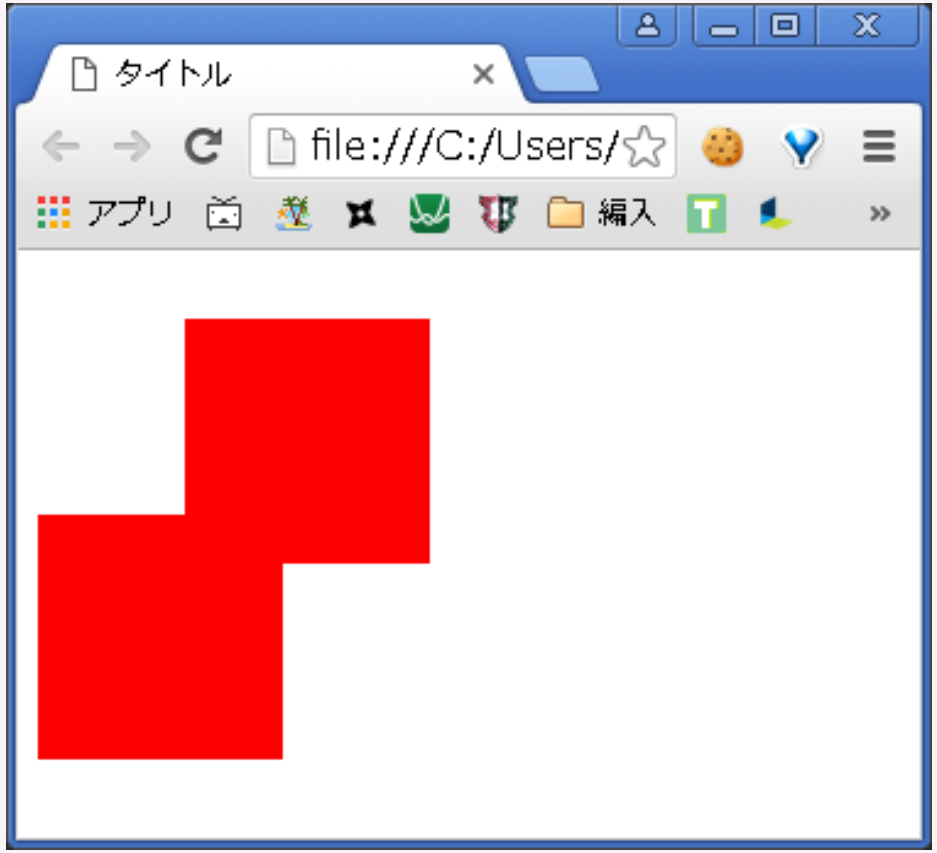
- position: relative;
  - 元の位置からずらす
  - top, bottom, left, right で指定

index.html

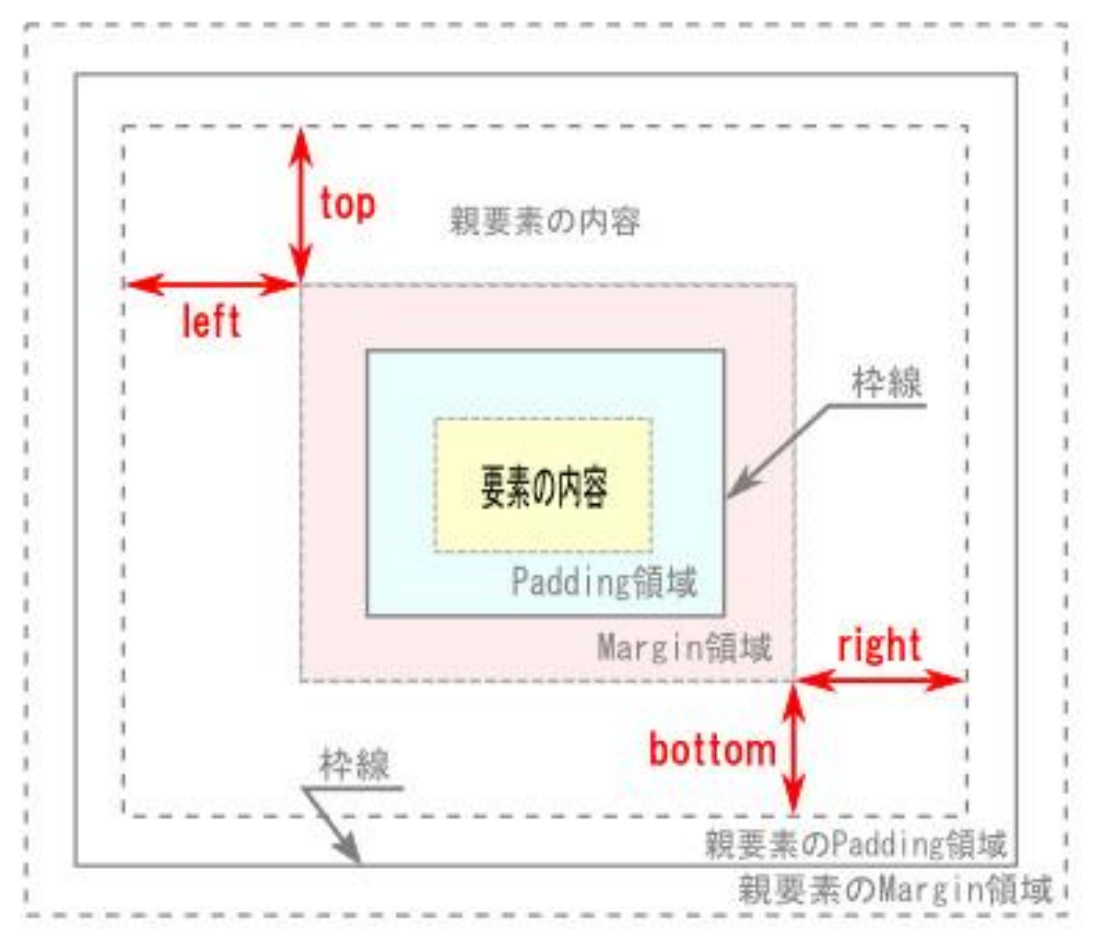
```
<div class="box1"></div>
<div class="box2"></div>
```

style.css

```
.box1, .box2 {
    width: 100px; height: 100px;
    background-color: red;
}
.box1 {
    position: relative;
    top: 20px; left: 60px;
}
```



top, bottom, left, right



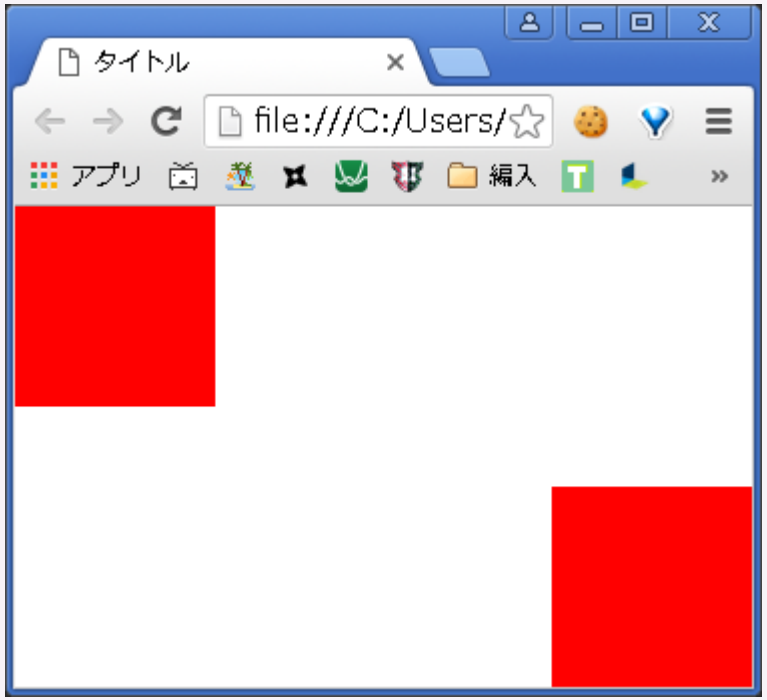
http://itpro.nikkeibp.co.jp/article/Reference/20091124/340985/

position

- position: absolute;
  - 絶対位置指定

style.css

```
.box1, .box2 {
    width: 100px; height: 100px;
    background-color: red;
    position: absolute;
}
.box1 {
    top: 0; left: 0;
}
.box2 {
    bottom: 0; right: 0;
}
```



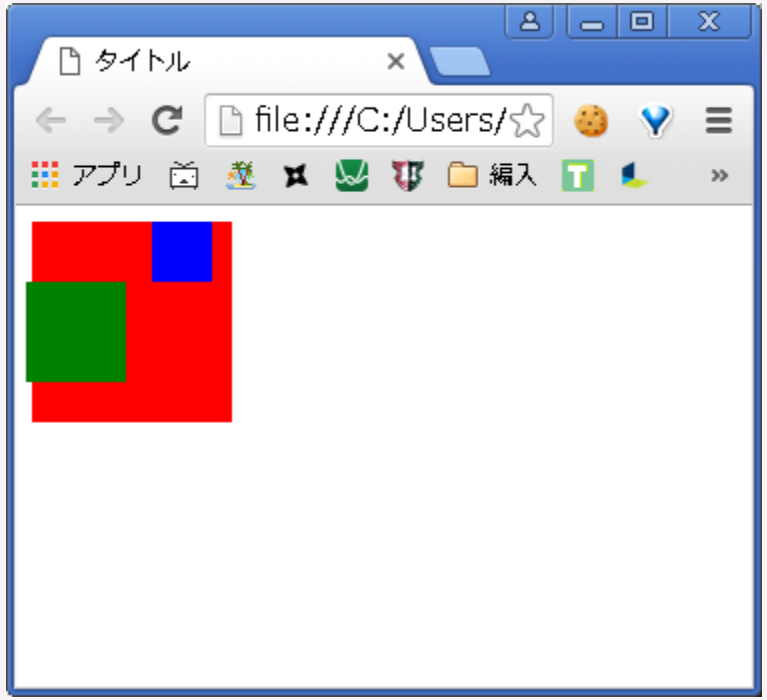
position

- 配置方法
  - position: relative;
    - 通常位置を基準とした相対座標
  - position: absolute;
    - positionが設定された一番近い祖先要素を基準とした絶対座標
    - なければページの左上が基準
  - position: fixed;
    - ブラウザの表示領域を基準とした絶対座標
    - スクロールしてもついてくる

position: absolute;

index.html

```
<div class="ancestor">
    <div class="parent">
        <div class="box"></div>
    </div>
</div>
```

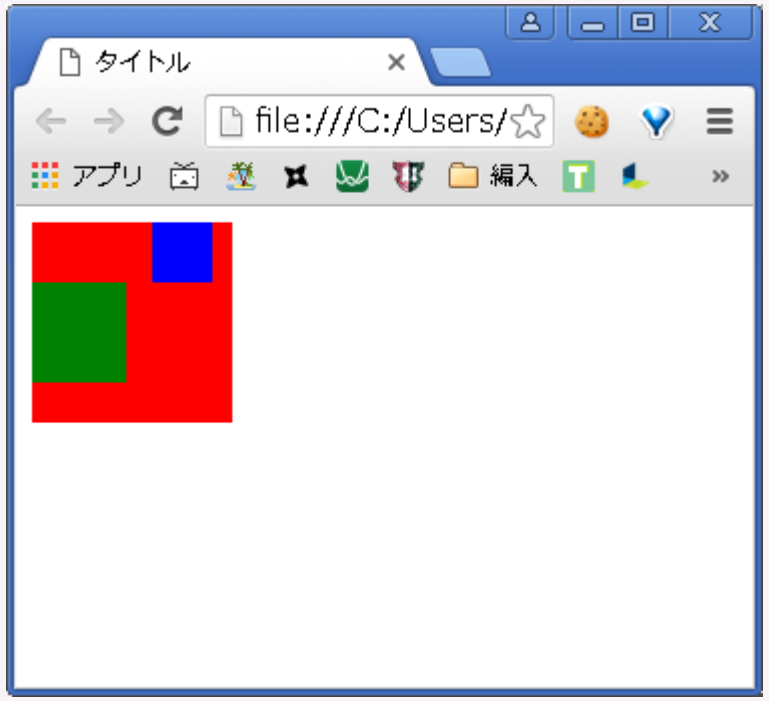


style.css

```
.ancestor {
    position: relative;
    width: 100px; height: 100px;
    background-color: red;
}
.parent {
    width: 30px; height: 30px;
    margin-left: 60px;
    background-color: blue;
}
.box {
    width: 50px; height: 50px;
    position: absolute;
    background-color: green;
    top: 30px; left: -3px;
}
```

overflow

- overflow: hidden;
  - はみ出した要素を非表示にする



style.css

```
.ancestor {
    position: relative;
    width: 100px; height: 100px;
    background-color: red;
    overflow: hidden;
}
.parent {
    width: 30px; height: 30px;
    margin-left: 60px;
    background-color: blue;
}
.box {
    width: 50px; height: 50px;
    position: absolute;
    background-color: green;
    top: 30px; left: -3px;
}
```

overflow

- overflow: visible | hidden | scroll | auto;
  - visible: デフォルト値. はみ出てもそのまま表示.
  - hidden: はみ出た要素は非表示.
  - scroll: スクロールバーを付ける. スクロールバーは常に表示.
  - auto: ブラウザ依存. いい感じに表示してくれるかもしれない.
    例えばはみ出たときだけスクロールバーが出たり, など.

overflow

- overflow-x: x方向のみ設定;
- overflow-y: y方向のみ設定;

4. DOM

elem.dataset

- 要素に勝手に新しい属性を設定できないのでそんなときに！！
- 要素に文字列データを設定できる
- HTMLの属性では data-hogehoge="fugafuga" と設定
- JavaScriptでは elem.dataset.hogehoge = "fugafuga"

main.js

```
const elem = document.getElementById("elem0");
elem.dataset.index = "0";
console.log(elem.dataset.index);
```

実行結果

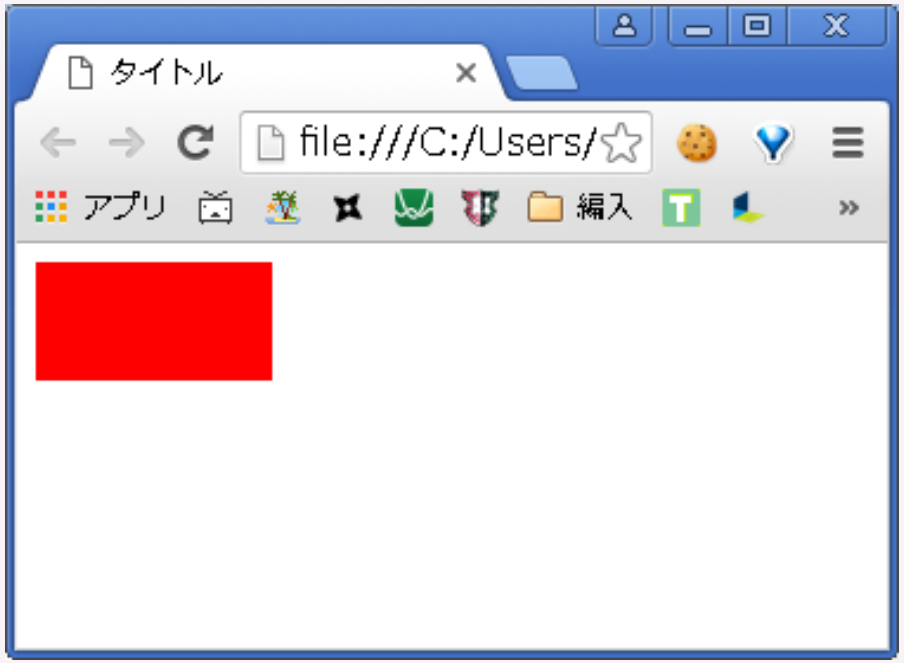
```
0
```

elem.style

- HTML要素にclassを設定せずに直接cssを記述できる

index.html

```
<div style="width: 100px; height: 50px; background-color: red;"></div>
```



elem.style

- HTML要素にclassを設定せずに直接cssを記述できる
  - デザインはできるだけ style.css に分離したい
  - デザインの種類が限られているなら class 設定したりしてまとめたい

index.html

```
<div style="width: 100px; height: 50px; background-color: red;"></div>
```

elem.style

- JavaScriptからもアクセスできる
  - elem.style.width = "100px";
  - elem.style.height = "50px";
  - elem.style.backgroundColor = "rgb(11, 45, 14)";

※ CSSで hoge-fuga といったプロパティ名が JavaScript では hogeFuga と -x が 大文字1文字の X で表される

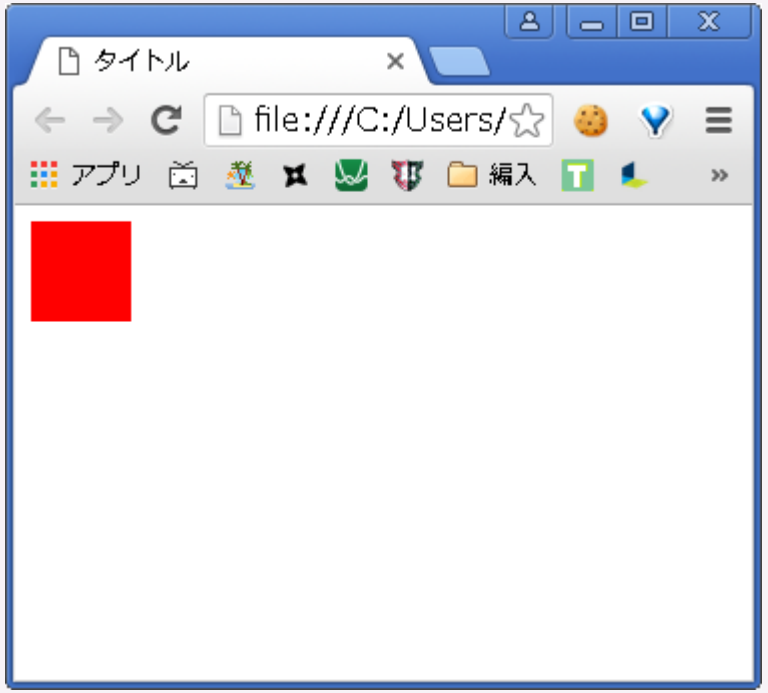
動的要素生成

index.html

```
<div id="screen">
    <div class="box"></div>
</div>
```

style.css

```
.box {
    width: 50px; height: 50px;
    background-color: red;
}
```



動的要素生成

- 前スライドのものと等価

index.html

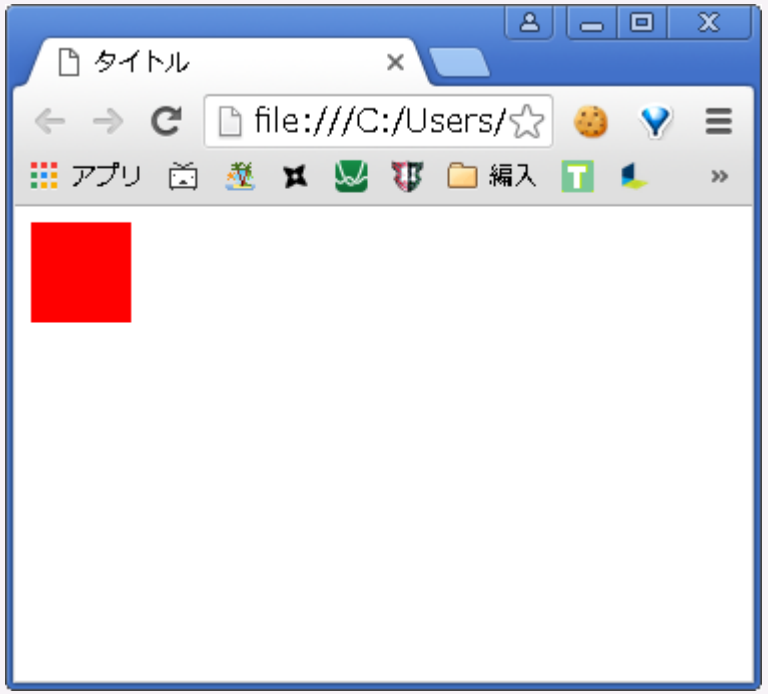
```
<div id="screen"></div>
```

style.css

```
.box {
    width: 50px; height: 50px;
    background-color: red;
}
```

main.js

```
const box = document.createElement("div");
box.classList.add("box");
document.getElementById("screen").appendChild(box);
```



動的要素生成

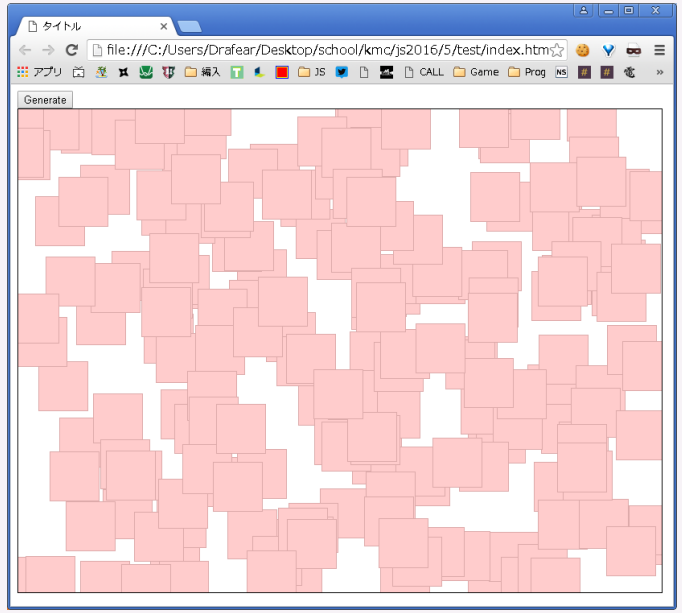
- document.createElement("*tagName*")
  - *tagName*の要素を生成する
- *parentElem*.appendChild(*childElem*)
  - *parentElem*の子に*childElem*を追加する

演習

- ボタンをクリックすると画面上のランダムな位置に箱が生成されるプログラムを書いてみよう
  - 画面幅やボックスのデザインは自分で決めてもおｋ

index.html

```
<button id="genBtn">Generate</button>
<div id="screen"></div>
```



style.css

```
#screen {
    position: relative;
    width: 800px; height: 600px;
    border: 1px solid black;
    overflow: hidden;
}
.box {
    width: 60px; height: 60px;
    position: absolute;
    background-color: #ffcccc;
    border: 1px solid #ddaaaa;
}
```

演習

index.html

```
const genBox = (screen) => {
    const box = document.createElement("div");
    const x = Math.floor( Math.random()*800 )-30;
    const y = Math.floor( Math.random()*600 )-30;
    box.style.top = `${y}px`;
    box.style.left = `${x}px`;
    box.classList.add("box");
    screen.appendChild(box);
};
document.getElementById("genBtn").addEventListener("click", (e) => {
    const screen = document.getElementById("screen");
    genBox(screen);
});
```

要素のサイズ

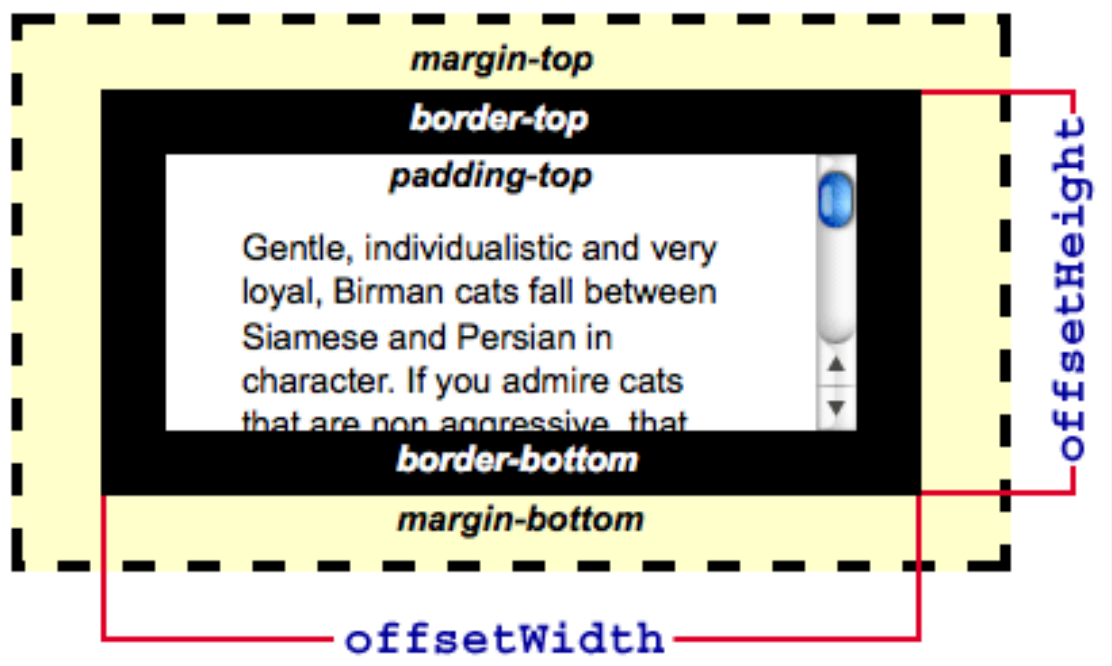
- *screen*の横幅とかとれたら便利そう

index.html

```
const genBox = (screen) => {
    const box = document.createElement("div");
    box.classList.add("box");
    const x = Math.floor( Math.random()*screen.offsetWidth - box.offsetWidth/2 );
    const y = Math.floor( Math.random()*screen.offsetHeight - box.offsetHeight/2 );
    box.style.top = `${y}px`;
    box.style.left = `${x}px`;
    screen.appendChild(box);
};
document.getElementById("genBtn").addEventListener("click", (e) => {
    const screen = document.getElementById("screen");
    genBox(screen);
});
```

要素のサイズ

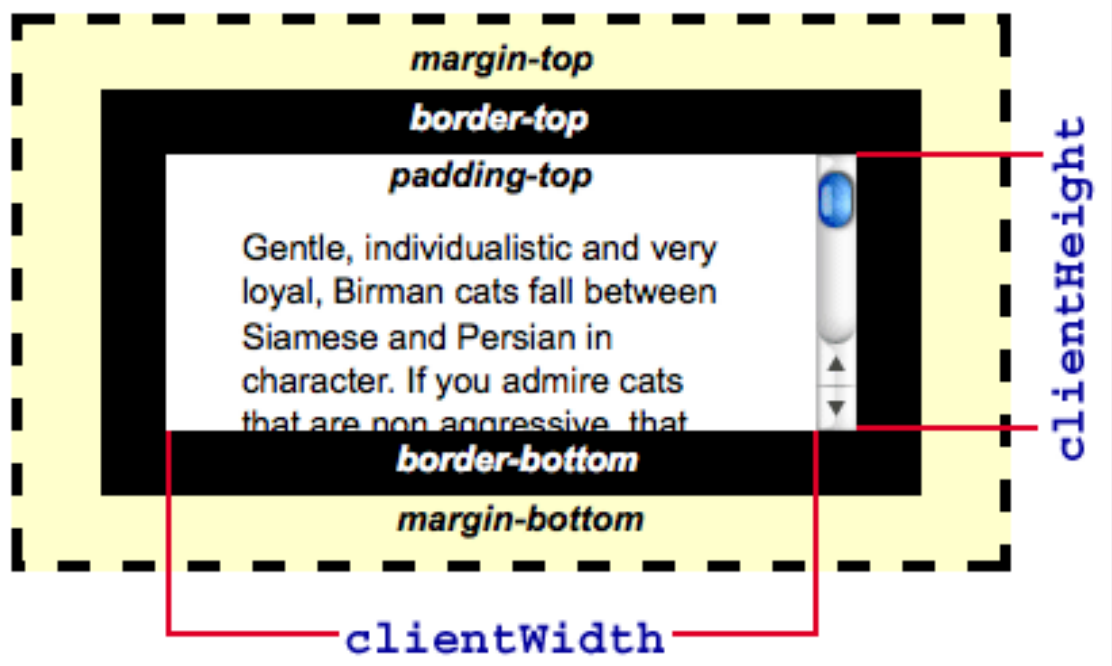
- *elem*.offsetWidth, *elem*.offsetHeight



https://developer.mozilla.org/ja/docs/Web/API/HTMLElement/offsetWidth

要素のサイズ

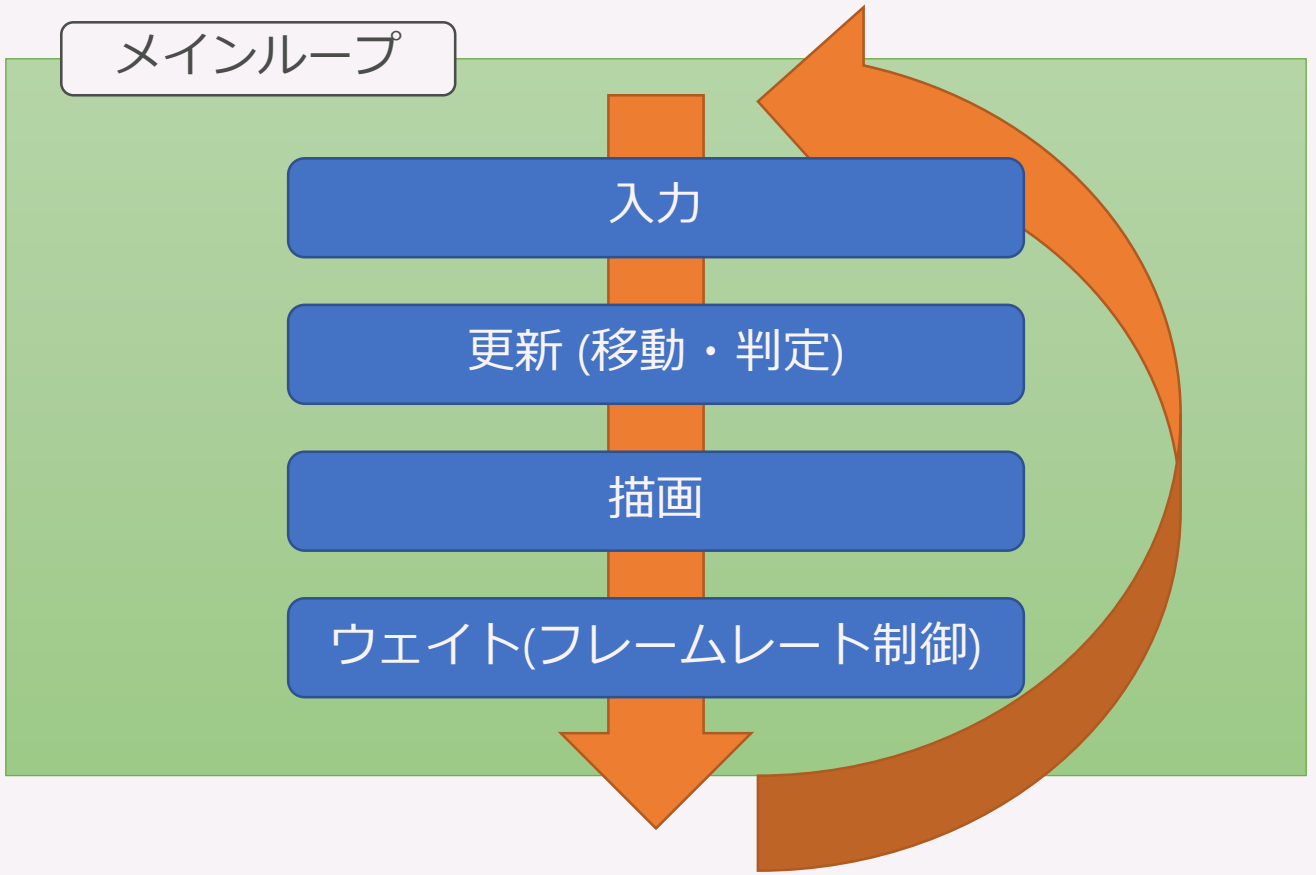
- *elem*.clientWidth, *elem*.clientHeight



https://developer.mozilla.org/en-US/docs/Web/API/Element/clientWidth

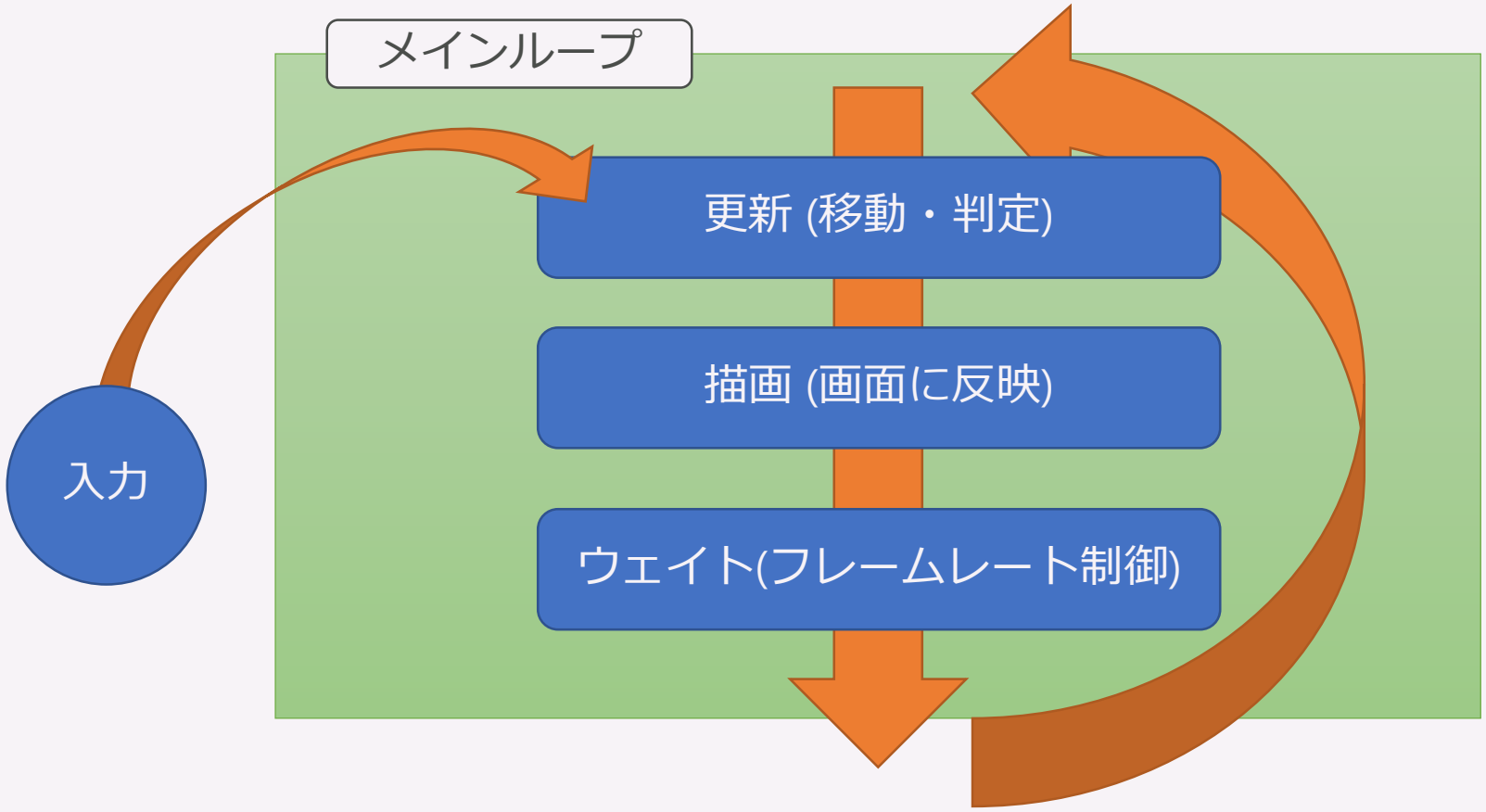
5. ゲームプログラミング入門

ゲームの一般的な手続き



JSでの手続き

- 描画はブラウザが勝手にやってくれる



ゲームを作るのに必要な機能

- 入力装置(キーボード, マウス等) からの入力
- (好きな位置への)画像の描画(表示)
- 音楽再生
- ウェイト処理

入力装置(キーボード, マウス等) からの入力

- キーボードからの入力
  - keydown, keyup, keypress イベント
  - event.keyCode
- キーコード一覧
  - http://www.programming-magic.com/file/20080205232140/keycode_table.html

main.js

```
// 入力されたキーの番号(キーコード)を表示する
document.addEventListener("keydown", (e) => {
    console.log(e.keyCode);
});
```

入力装置(キーボード, マウス等) からの入力

- メインループからの読み出し

main.js

```
document.addEventListener("keydown", (e) => {
    key[e.keyCode] = true;
});
document.addEventListener("keyup", (e) => {
    key[e.keyCode] = false;
});
// メインループ内の更新関数
const update = () => {
    if (key[38]) { // ↑
        ...
    }
}
```

入力装置(キーボード, マウス等) からの入力

- マウスからの入力
  - mousedown, mouseup, mousedown イベント
  - event.clientX, event.clientY

main.js

```
document.addEventListener("mousemove", (e) => {
    console.log({ x: event.clientX, y: event.clientY });
});
```

(好きな位置への)画像の描画(表示)

- img タグ
- css の position を absolute; にして top, left をいじる

index.html

```
<img id="player" src="画像ファイルへのパス" alt="代替テキスト">
```

main.js

```
let playerX = 10, playerY = 20;
const player = document.getElementById("player");
player.style.left = `${playerX}px`;
player.style.top = `${playerY}px`;
```

style.css

```
#player {
    position: absolute;
}
```

音楽再生

- 今度やります

ウェイト処理

- requestAnimationFrame
  - 関数を渡すと, およそ $\frac{1}{60}$ 秒後 に 1回だけ 実行される
  - 正確には次にブラウザが描画しようとする直前

main.js

```
const mainloop = () => {
    update(); // 更新処理
    draw(); // 描画処理
    requestAnimationFrame(mainloop); // ウェイト処理
}
```

ウェイト処理

- 「ウェイト処理」と書いてますが requestAnimationFrame を呼び出すとその後の全ての処理が $\frac{1}{60}$ 秒間停止するわけではなくて, $\frac{1}{60}$ 秒後に mainloop を登録する感じ
- その時になって登録されてたら実行される

main.js

```
const mainloop = () => {
    update(); // 更新処理
    draw(); // 描画処理
    requestAnimationFrame(mainloop); // ウェイト処理
}
```

おまけ

- 設定したrequestAnimationFrameを取り消すには
  - requestId = requestAnimationFrame(func);
  - cancelAnimationFrame(requestId);

main.js

```
let requestId;
const mainloop = () => {
    update(); // 更新処理
    draw(); // 描画処理
    requestId = requestAnimationFrame(mainloop); // ウェイト処理
}
const cancel() = () => {
    if (requestId !== undefined) {
        cancelAnimationFrame(requestId); // キャンセル
    }
};
```

1/60 秒より短くするには

- setTimeout(*function*, *delay*, [*param1*, *param2*, ...]);
  - *delay* [ms] 後に 関数*function* を実行する
  - 関数*function* に引数を渡したい場合は,
    setTimeout第三引数以降に指定する

main.js

```
// 1秒後に "Hello!!" とアラートを出す
setTimeout(() => {
    alert("Hello!!");
}, 1000);
```

1/60 秒より短くするには

- Math.floor(1000/30) は
  1秒(1000ms)間に30回(くらい)実行するという意味
- 関数*mainloop*の最初にsetTimeoutを入れているのは
  update および *draw* 後だとズレが大きくなってしまう
  かもしれないから
- *update*中や*draw*中に次の*mainloop*が呼び出されることはない

main.js

```
const mainloop = () => {
    setTimeout(mainloop, Math.floor(1000/30)); // ウェイト処理
    update(); // 更新処理
    draw(); // 描画処理
}
```

1/60 秒より短くするには

- setTimeout(*function*, *delay*, [*param1*, *param2*, ...]);
  - setTimeout を実行してから *delay* [ms] 後以降に
    ビジー状態(何か処理をしている状態) でなければ
    関数*function* を実行する
  - ふつう, *delay* [ms] よりも遅くに実行される

main.js

```
// 1秒後おきに "Hello!!" とアラートを出す
const loop = () => {
    setTimeout(loop, 1000);
    alert("Hello!!");
}
loop();
```

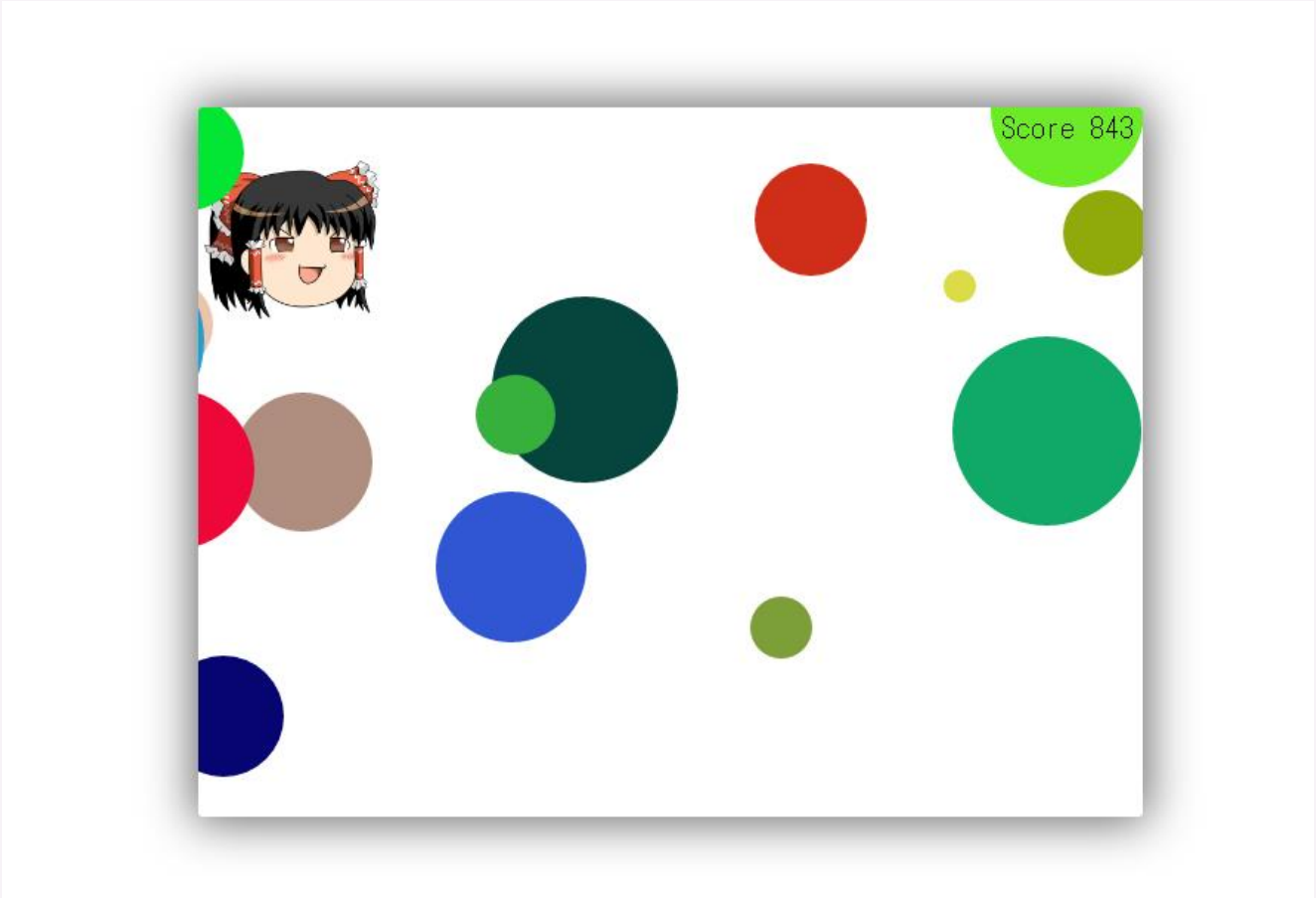
setTimeout vs requestAnimationFrame

- setTimeoutの方が便利そうだけど
  mainloop の意味的には requestAnimationFrame の方が
  合っているので, こちらを使う

6. 避けゲー作る

避けゲー

- http://drafear.ie-t.net/js2016/yukkuri/



画面

index.html

```
<div class="cover centering">
    <div id="screen">
    </div>
</div>
```

style.css

```
body {
    margin: 0;
}
.centering {
    display: flex;
    justify-content: center;
    align-items: center;
}
.cover {
    width: 100vw; height: 100vh;
}
#screen { ... }
```

設定

- PLAYER_SLOW_SPEED
  - Shiftキーを押している間の低速移動の移動速度

main.js

```
const Settings = {
    SCREEN_WIDTH: 640,
    SCREEN_HEIGHT: 480,
    PLAYER_WIDTH: 128,
    PLAYER_HEIGHT: 128,
    PLAYER_SLOW_SPEED: 3,
    PLAYER_FAST_SPEED: 6,
};
```

メインループ

- これだけ！
  - あとはキー入力と, ゲーム本体の処理を書いていく

main.js

```
const game = new Game(document.getElementById("screen"));
const mainloop = () => {
    game.update();
    game.draw();
    requestAnimationFrame(mainloop);
};
```

キー入力処理

- 押されたらtrueに, 離されたらfalseに更新

main.js

```
const key = [];
document.addEventListener("keydown", (e) => {
    key[e.keyCode] = true;
});
document.addEventListener("keyup", (e) => {
    key[e.keyCode] = false;
});
```

Gameクラスを実装する

- とりあえず画面の初期化とプレイヤー表示をする

画面の初期化とプレイヤー表示

- css の #screen で width, height を指定していたら消しましょう

main.js

```
class Game {
    constructor(screen) {
        this.screen = screen;
        this.screen.style.width = `${Settings.SCREEN_WIDTH}px`;
        this.screen.style.height = `${Settings.SCREEN_HEIGHT}px`;
        this.player = new Player(Settings.PLAYER_WIDTH/2, Settings.SCREEN_HEIGHT/2, this.screen);
        this.screen.appendChild(this.player.elem);
    }
    update() {}
    draw() {
        this.player.draw();
    }
}
```

画面の初期化とプレイヤー表示

main.js

```
class Player {
    constructor(x, y) {
        this.x = x;
        this.y = y;
        this.isDie = false;
        this.elem = this._initElement();
    }
    _initElement() {
        const player = document.createElement("img");
        player.alt = "player";
        player.src = "image/player.png";
        player.classList.add("player");
        player.style.width = `${Settings.PLAYER_WIDTH}px`;
        player.style.height = `${Settings.PLAYER_HEIGHT}px`;
        return player;
    }
    move() {}
    draw() { ... }
}
```

プレイヤーを表示する

main.js

```
class Player {
    ...
    draw() {
        this.elem.style.left = `${this.x-Settings.PLAYER_WIDTH/2}px`;
        this.elem.style.top = `${this.y-Settings.PLAYER_HEIGHT/2}px`;
    }
}
```

style.css

```
.player {
    position: absolute;
}
```

プレイヤーの移動処理

- 次に, プレイヤーの移動処理を行う

プレイヤーの移動処理

main.js

```
class Game {
    ...
    _move() {
        if (!this.player.isDie) {
            this.player.move();
        }
    }
    update() {
        this._move();
    }
}
```

プレイヤーの移動処理

main.js

```
class Player {
    ...
    move() {
        let dx = 0;
        let dy = 0;
        if (key[37] > 0) --dx; // ←
        if (key[38] > 0) --dy; // ↑
        if (key[39] > 0) ++dx; // →
        if (key[40] > 0) ++dy; // ↓
        // Shiftキーが押されていたら
        const speed = key[16] ? Settings.PLAYER_SLOW_SPEED : Settings.PLAYER_FAST_SPEED;
        dx *= speed, dy *= speed;
        this.x += dx;
        this.y += dy;
    }
}
```

プレイヤーの移動処理

main.js

```
class Player {
    ...
    move() {
        let dx = 0;
        let dy = 0;
        if (key[37]) --dx; // ←
        if (key[38]) --dy; // ↑
        if (key[39]) ++dx; // →
        if (key[40]) ++dy; // ↓
        // Shiftキーが押されていたら
        const speed = key[16] ? Settings.PLAYER_SLOW_SPEED : Settings.PLAYER_FAST_SPEED;
        dx *= speed, dy *= speed;
        this.x += dx;
        this.y += dy;
    }
}
```

プレイヤーの移動処理

- 動いたら成功！

プレイヤーの移動範囲制限

- 今のままだと, プレイヤーが画面外に出られるので, 移動範囲を制限する

プレイヤーの移動範囲制限

main.js

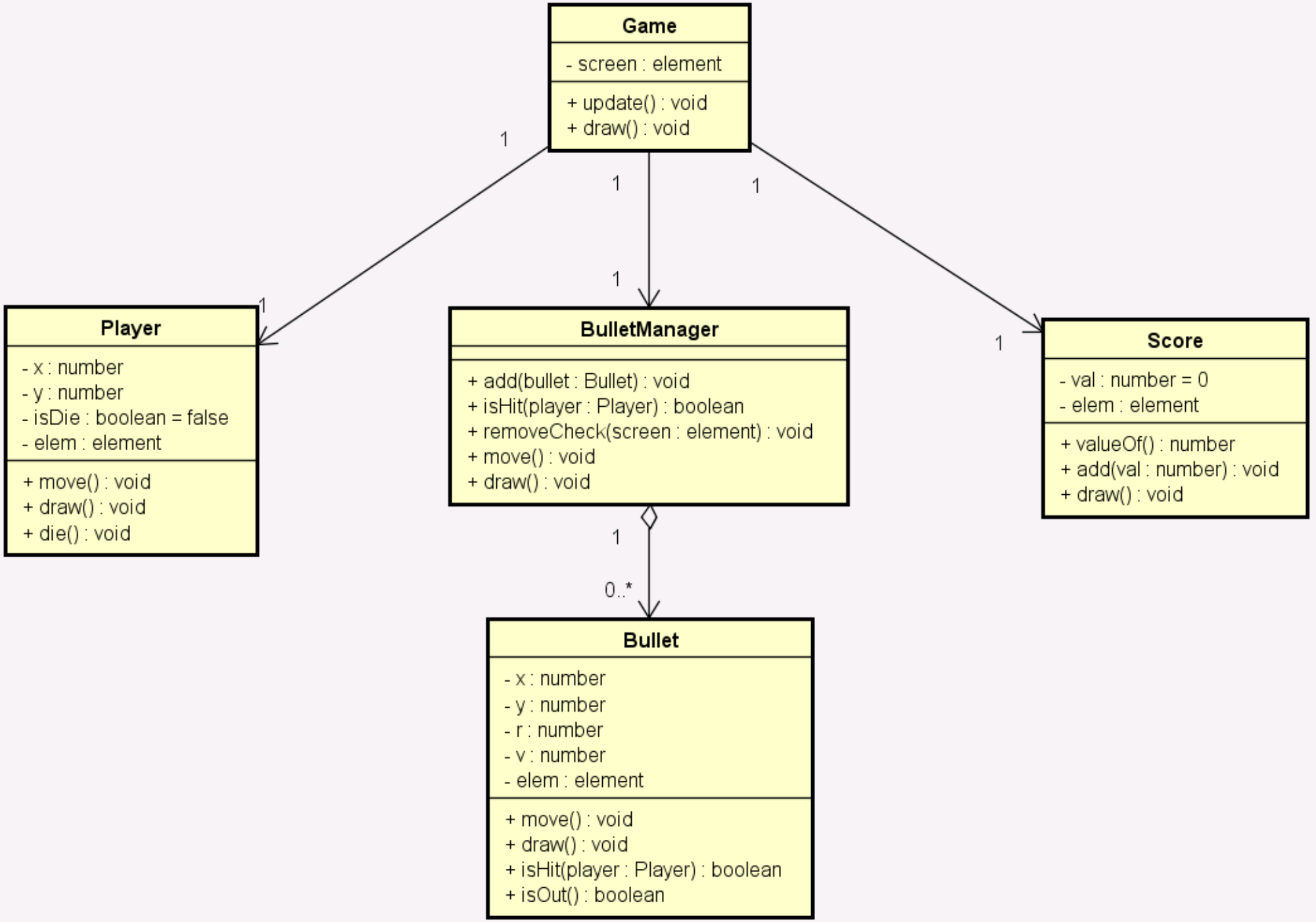
```
class Player {
    ...
    move() {
        ...
        if ( this.x < 0 ) this.x = 0;
        if ( this.x > Settings.SCREEN_WIDTH-1 ) this.x = Settings.SCREEN_WIDTH-1;
        if ( this.y < 0 ) this.y = 0;
        if ( this.y > Settings.SCREEN_HEIGHT-1 ) this.y = Settings.SCREEN_HEIGHT-1;
    }
}
```

プレイヤーの移動処理

- ちゃんと動きましたか？？

後の実装方針

- 弾を実装する前に・・・

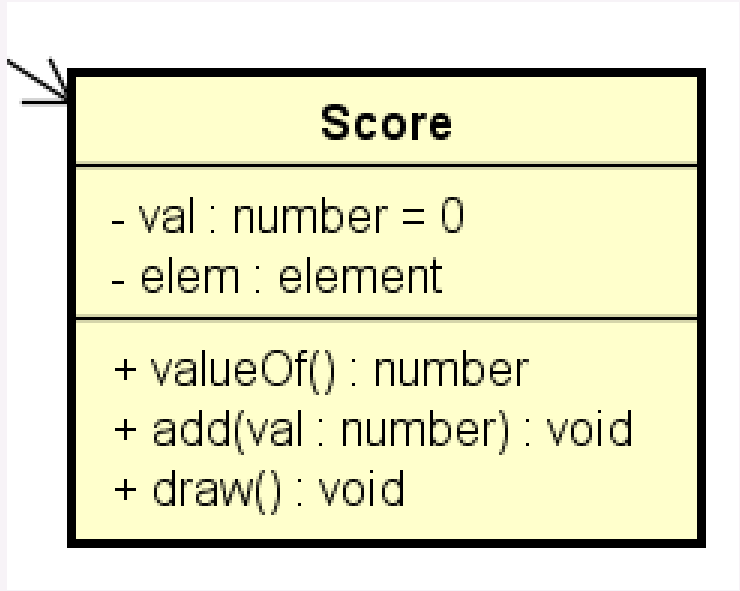

Game
- screen : element
+ update() : void
+ draw() : void
1
1
1
1
1
1
Player
- x : number
- y : number
- isDie : boolean = false
- elem : element
+ move() : void
+ draw() : void
+ die() : void
BulletManager
+ add(bullet : Bullet) : void
+ isHit(player : Player) : boolean
+ removeCheck(screen : element) : void
+ move() : void
+ draw() : void
Score
- val : number = 0
- elem : element
+ valueOf() : number
+ add(val : number) : void
+ draw() : void
1
0..*
Bullet
- x : number
- y : number
- r : number
- v : number
- elem : element
+ move() : void
+ draw() : void
+ isHit(player : Player) : boolean
+ isOut() : boolean

実装方針

- 前スライドはUML(Unified Modeling Language)のクラス図
- UMLとは仕様や設計を図示する記法を定めたもの
  - つまり世界共通言語
- クラス図はクラス間の関係を表す
  - GameはPlayerとBulletManagerとScoreを操作する
    - Playerに対して 移動, 描画, 死ぬ 操作ができる
    - BulletManagerに対して …
  - BulletManagerはBulletを管理する
  - isHitで当たり判定をとる

Scoreクラスの定義

- まずは簡単なScoreクラスから定義・実装していきましょう
- Scoreクラス
  - 値val と 対応する要素 elem を持つ
  - スコアの値の取得, スコアへの加算, 描画処理 が行える

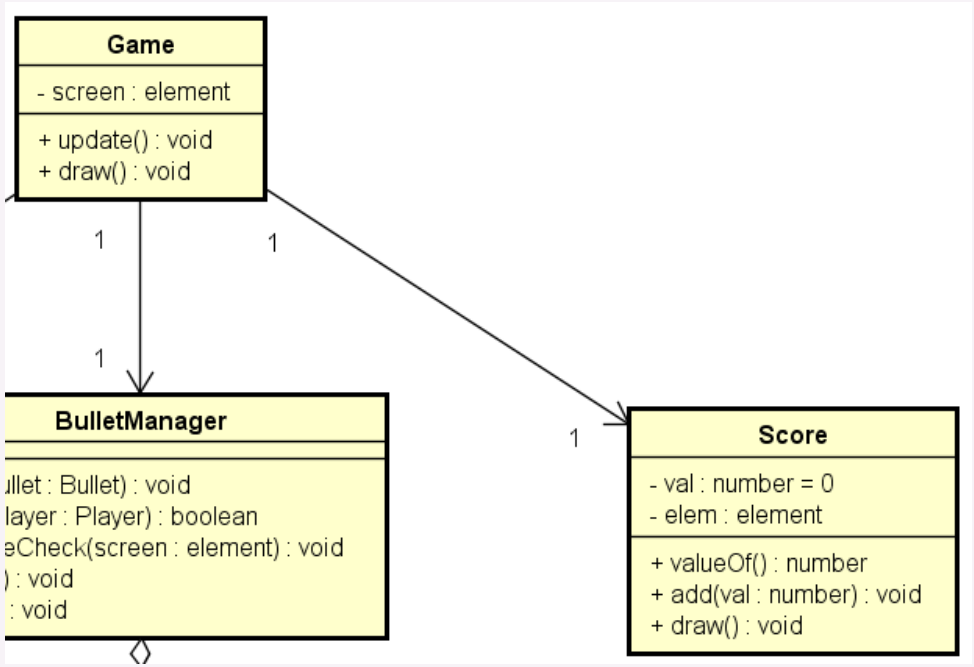


Scoreクラスの実装


```
class Score {
    constructor(screen) {
        this.val = 0;
        this.elem = document.createElement("div");
        this.elem.classList.add("score");
        screen.appendChild(this.elem);
    }
    valueOf() { ... }
    add(val) { ... }
    draw() { ... }
}
```

GameクラスからScoreクラスを操作する

- GameクラスがScoreクラスをどう操作するか
  - 毎フレームScoreに1加える
  - 描画処理をさせる



GameクラスからScoreクラスを操作する

```
class Game {
    constructor(screen) {
        ... // 前と同じ
        this.score = new Score();
        this.screen.appendChild(this.score.elem);
    }
    _move() {
        this.bullets.move();
        if (!this.player.isDie) {
            this.player.move();
            this.score.add(1);
        }
    }
    update() { ... } // 前と同じ
    draw() {
        ... // 前と同じ
        this.score.draw();
    }
}
```

GameクラスからScoreクラスを操作する

- Scoreが増えていれば成功

GameクラスからScoreクラスを操作する

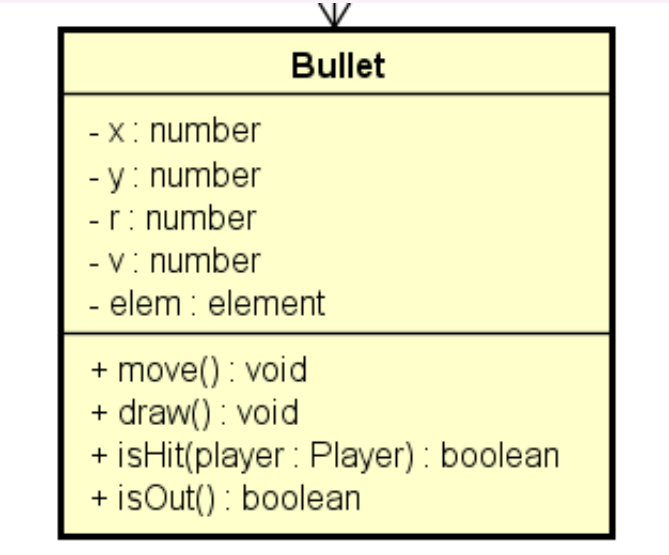
- CSSも書いておこう
  - beforeはそれの直前に要素を追加する. contentは要素の中身の文字.

style.css (例)

```
.player,
.score {
    position: absolute;
}
.score {
    top: 0;
    right: 0;
    padding: 5px;
    font-size: 20px;
    z-index: 10;
}
.score:before {
    content: "Score ";
}
```

弾の定義

- 次に弾を実装しましょう
- 弾をまず定義します
  - x座標, y座標, 半径, 速度, 対応する要素 を持つ
  - 移動処理, 描画処理, プレイヤーとの当たり判定, 画面外判定 ができる
  - 速度は正でありx軸の負の方向に動く

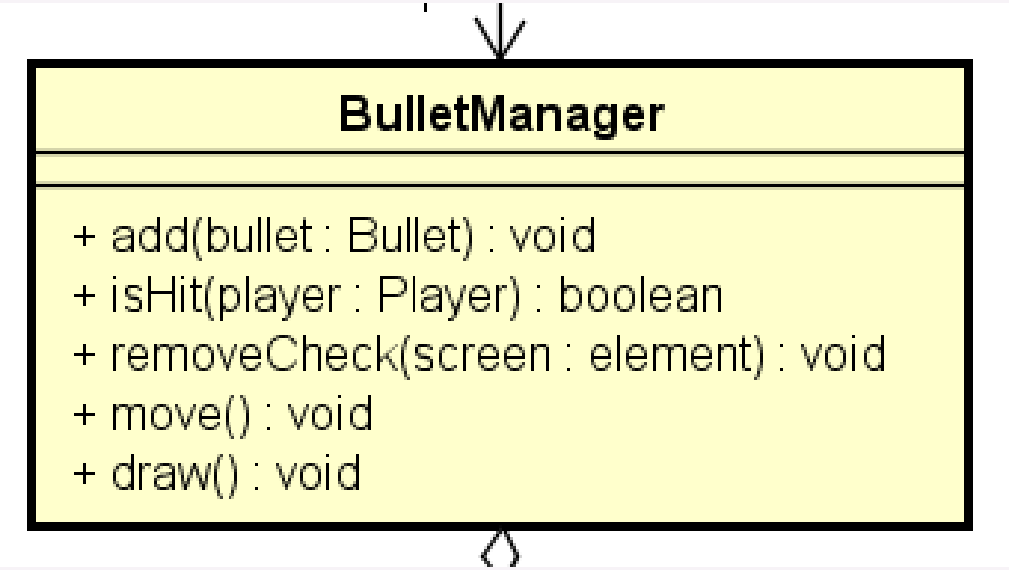


弾の実装

```
class Bullet {
    constructor(x, y, r, v, color) {
        this.x = x;
        this.y = y;
        this.r = r;
        this.v = v;
        this.elem = document.createElement("div");
        this.elem.classList.add("bullet");
        this.color = color;
    }
    isHit(player) { ... }
    isOut() { ... }
    move() { ... }
    draw() {
        this.elem.style.left = `${Math.floor(this.x)-this.r}px`;
        this.elem.style.top = `${this.y-this.r}px`;
        this.elem.style.backgroundColor = this.color;
        this.elem.style.width = `${this.r*2}px`;
        this.elem.style.height = `${this.r*2}px`;
        this.elem.style.borderRadius = `${this.r}px`;
    }
}
```

弾管理クラスの定義

- 弾管理クラス
  - 弾の配列を持つ
  - 管理する弾を追加する操作, プレイヤーとの当たり判定, 画面外に出た弾の除去, 移動処理, 描画処理ができる

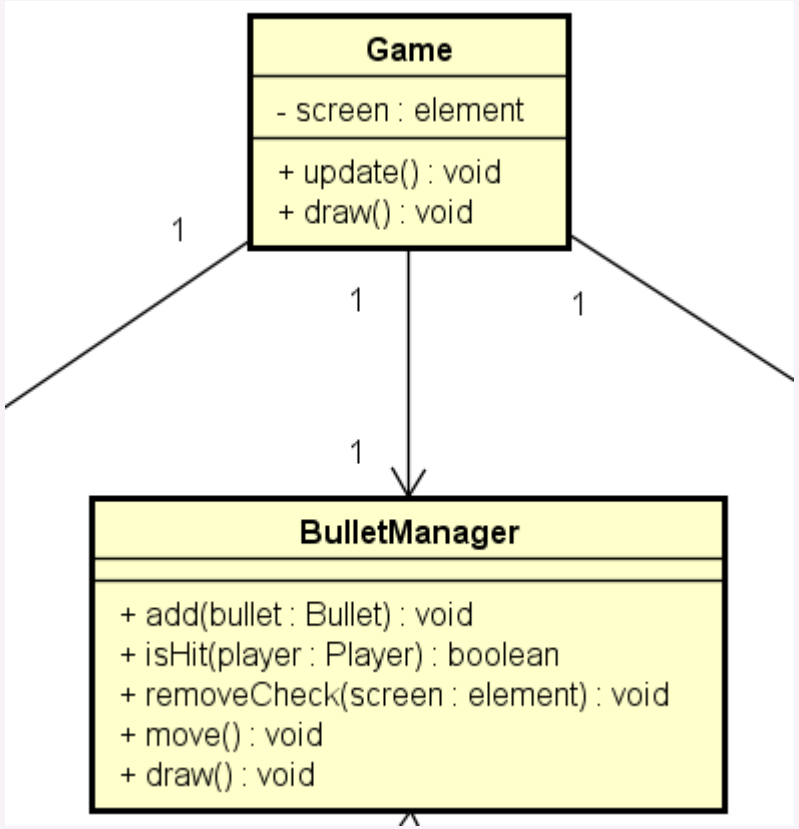


弾管理クラスの実装

```
class BulletManager {
    constructor() {
        this.bullets = [];
    }
    add(bullet) { ... }
    removeCheck(screen) {
        const nextBullets = [];
        for (const bullet of this.bullets) {
            if ( bullet.isOut() ) {
                screen.removeChild(bullet.elem);
            }
            else {
                nextBullets.push(bullet);
            }
        };
        this.bullets = nextBullets;
    }
    isHit(player) { ... }
    move() { ... }
    draw() { ... }
}
```

GameがBulletManagerを扱うようにする

- GameクラスがBulletManagerをどう操作するか
  - 移動処理を行う
  - 一定確率で弾を生成して追加する
  - 画面外の弾を除去する処理を呼ぶ
  - Playerと当たり判定をとって
    当たっていたらPlayerに死んでもらう
  - 描画処理を行う



```
class Game {
    constructor(screen) {
        ... // 前と同じ
        this.bullets = new BulletManager();
    }
    _move() {
        ... // 前と同じ
        this.bullets.move();
    }
    _genBullet() { // 弾を生成する
        ...
        const bullet = new Bullet(...);
        return bullet;
    }
    _genBulletCheck() { ... } // 弾を追加するかもしれない
    _hitCheck() { ... } // プレイヤーが弾に当たっていたら死んでもらう
    update() {
        this._move();
        this._genBulletCheck();
        this.bullets.removeCheck(this.screen);
        this._hitCheck();
    }
    draw() {
        ... // 前と同じ
        this.bullets.draw();
    }
}
```

GameクラスからScoreクラスを操作する

- CSSを書いて終了！

style.css (例)

```
.player,
.bullet,
.score {
    position: absolute;
}
.bullet {
    z-index: 1;
}
```

完成？

- まだまだ拡張できるゾ
  - スコアに応じて難易度上昇
  - 弾の軌道, 角度を変えてみる
  - プレイヤーの当たり判定を大きくする
  - プレイヤーの当たり判定を見やすくする
  - HP制にする
  - アイテムを追加する
  - ボム(必殺技)を追加する
  - しっかりとカプセル化する

今後の予定

- 6/12(日) 13:00 ～ 16:00
  - JavaScriptをより柔軟に扱えるように
  - CSSのデザイン力を幅広く！
  - トップページを作る
- 6/19(日) 13:00 ～ 16:00
  - 文字列処理と正規表現
  - 他のサイトのデータをとってくる
  - Web API いろいろ
- 6/26(日) 13:00 ～ 16:00
  - ランキングサーバを作ってみる