

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(どらふいあ, どらふあー)
- 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



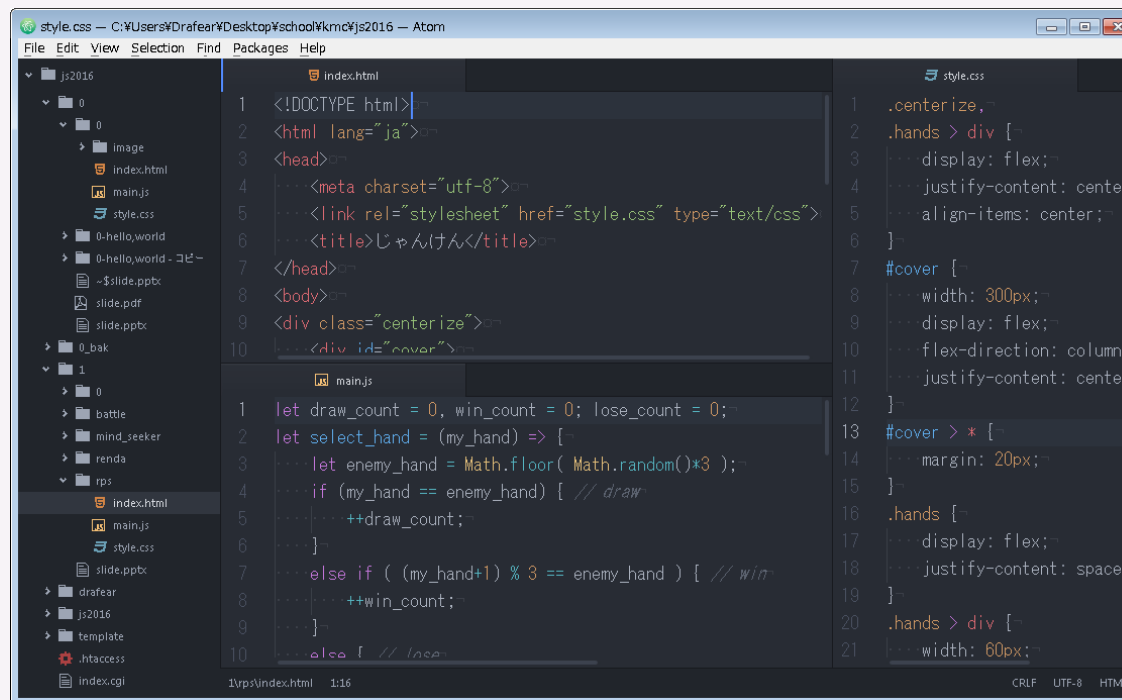
@gekimon\_1d1a



@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

- JavaScriptの基本的構文をマスターする
- CSSで好きなレイアウトを組めるようになる

# 本日の内容

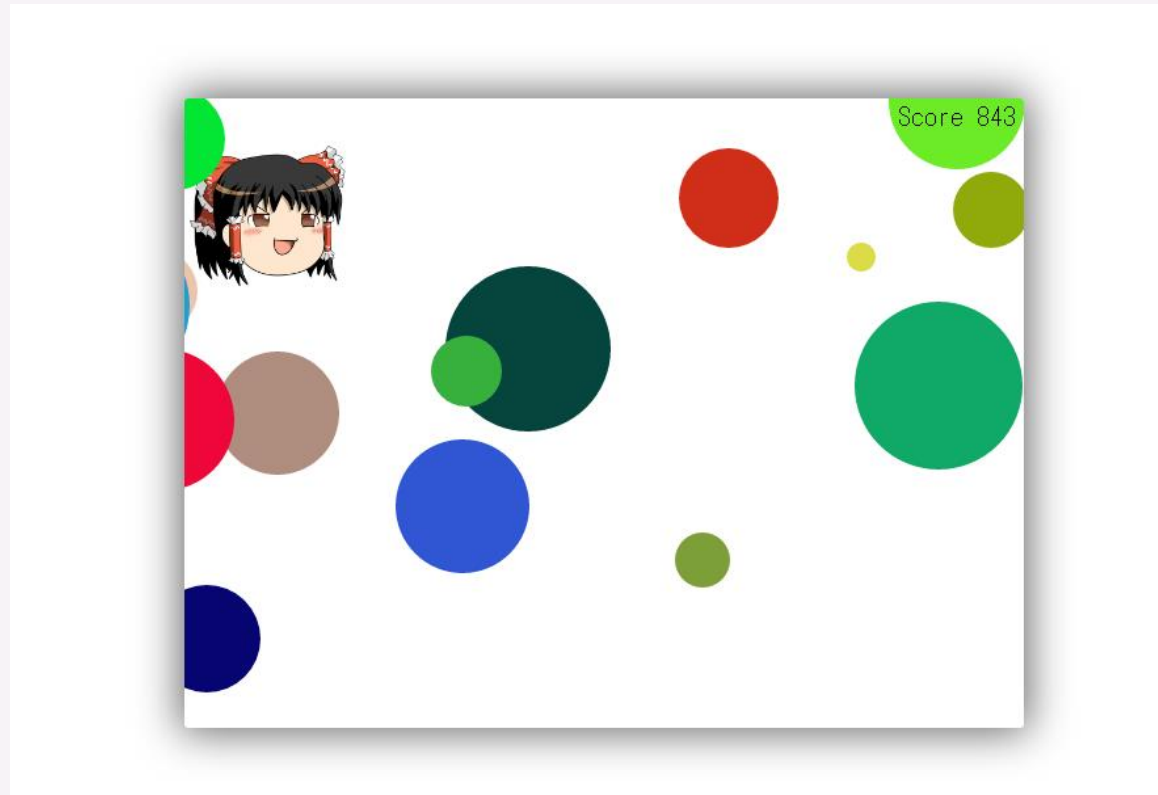
- JavaScript
  - 配列 push, pop, shift, unshift, splice, forEach, concat
  - 列挙 for ... in, for ... of
  - 定数(const)
  - クラスの基礎
  - getter / setter
  - 'use strict'
- CSS
  - box-shadow
  - position, top, bottom, left, right, z-index
  - overflow

# 本日の内容

- DOM操作
  - *elem*.dataset
  - *elem*.style
  - document.createElement
  - *elem*.appendChild, *elem*.removeChild
- その他
  - 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)
- 値の書き換え(代入)
- 要素数
- 末尾への要素の追加
- 先頭への要素の追加
- 末尾要素の削除
- 先頭要素の削除
- 連続する要素の削除
- 特定の要素を削除

`ary[pos]`

`ary[pos] = val`

`ary.length`

`ary.push(val)`

`ary.unshift(val)`

`ary.pop(val)`

`ary.shift(val)`

`ary.splice(pos, num)`

`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;  
}
```

# 演習

- 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

main.js (あまりよくない例)

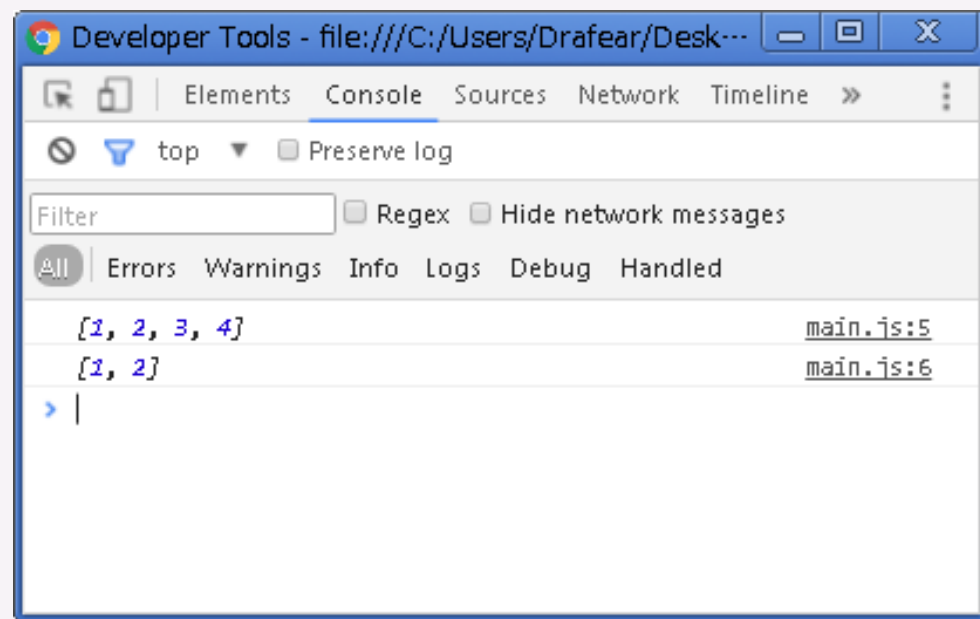
```
let concatArray = (ary1, ary2) => {  
  for (let i = 0; i < ary2.length; ++i) {  
    ary1.push(ary2[i]);  
  }  
  return ary1;  
}  
let a = [5, 10];  
let b = [8, 30];  
let c = concatArray(a, b);  
console.log(a); // [5, 10, 8, 30]  
console.log(b); // [8, 30]  
console.log(c); // [5, 10, 8,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

```
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

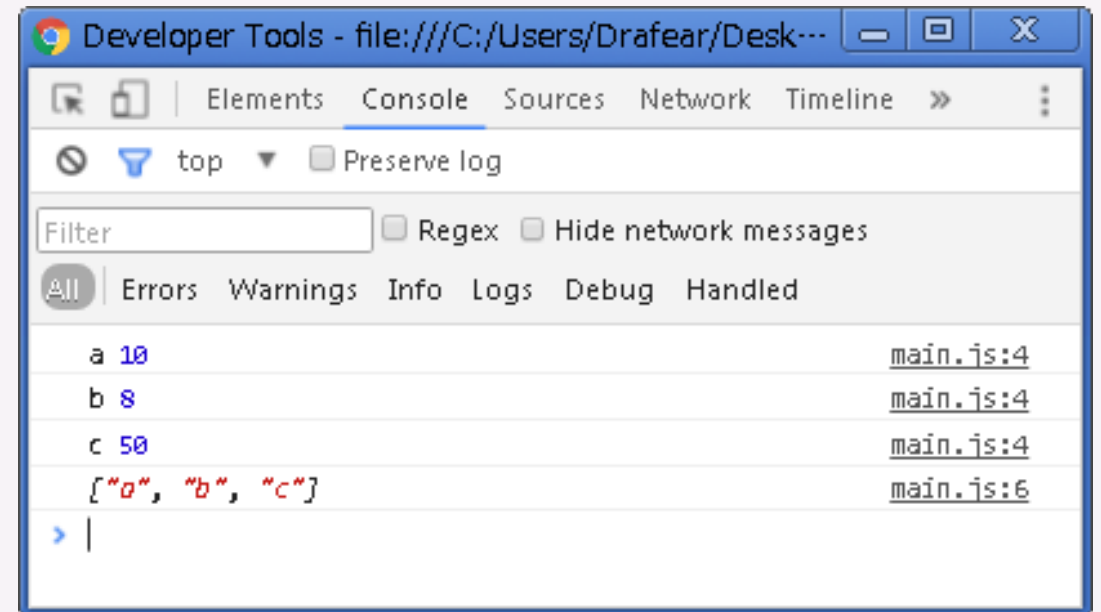
```
let ary = {  
  "0": 5,  
  "1": 7,  
  "2": 10,  
  length: 3,  
  push: () => { ... },  
};
```

# for ... in

- `for (let key in obj) { ... }`
  - オブジェクト`obj`のプロパティを列挙する (`key`に順に入っていく)
- `Object.keys(obj)`
  - オブジェクト`obj`の全てのプロパティを配列で得る

main.js

```
let obj = { a: 10, b: 8 };  
obj.c = 50;  
for (let key in obj) {  
    console.log(key, obj[key]);  
}  
console.log(Object.keys(obj));
```

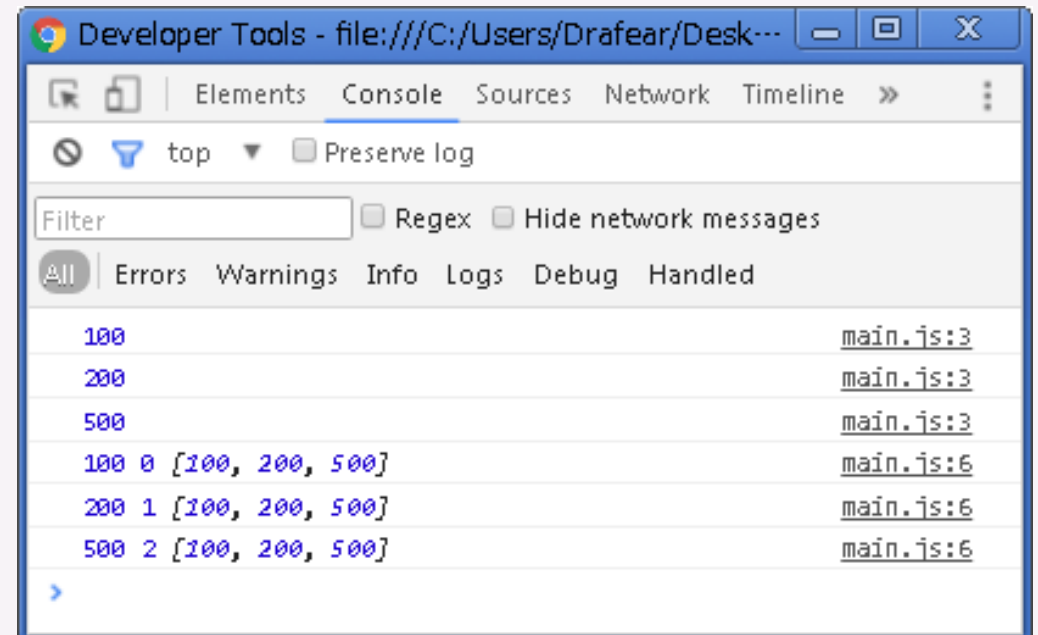


# for ... of

- `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);
});
```

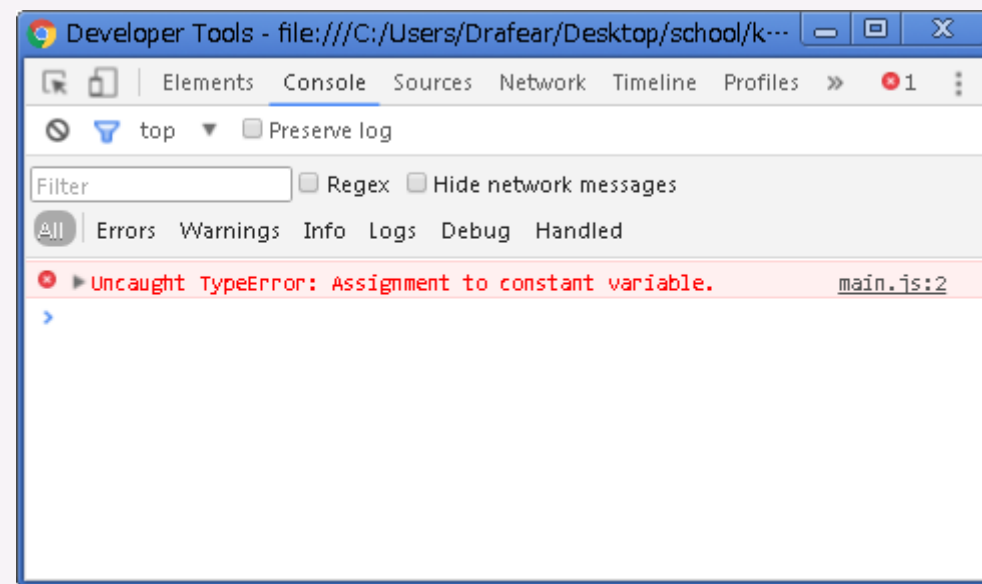


# 定数

- 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` な変数は後から変更不可!

main.js

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

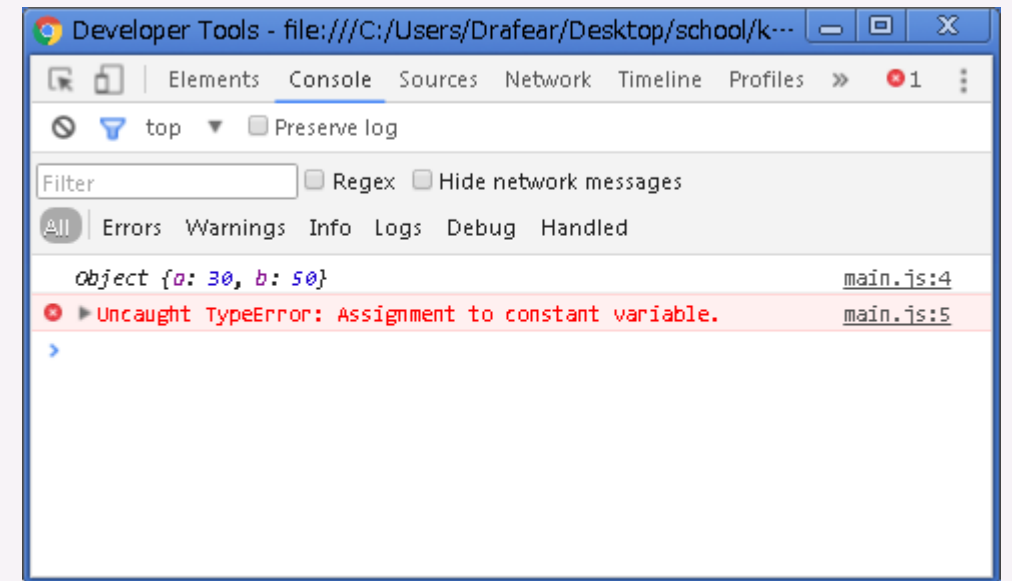


# 定数

- 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 な変数は後から変更不可!
  - 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！」  
と表明して可読性アップ！
  - 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

- といふわけで, Objectのプロパティにも const 付けたいですね
- できます!!!!!!

# const教

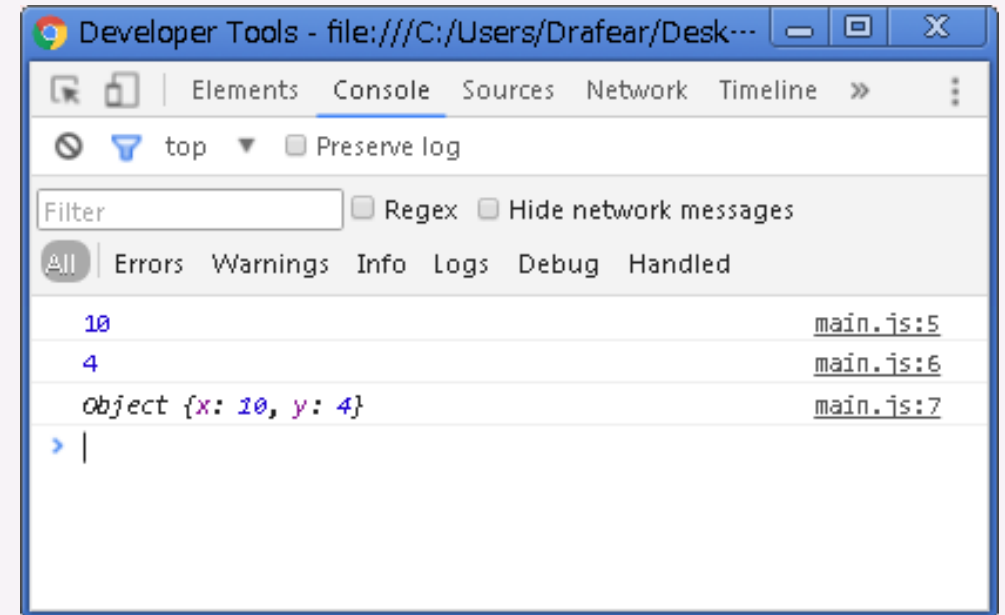
- というわけで, Objectのプロパティにも const 付けたいですね
- できます!!!!!!!!!!!!
- が、次回にします
- 今回は const に慣れよう!!

# 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

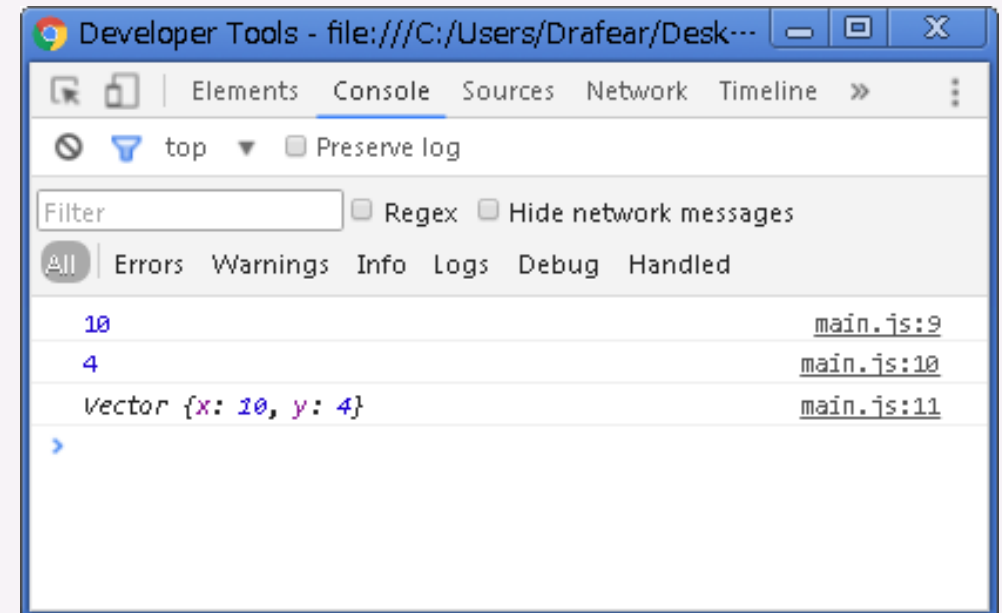
```
const makeVector = (x, y) => {  
  return { x: x, y: y };  
}  
const v = makeVector(10, 4);  
console.log(v.x);  
console.log(v.y);  
console.log(v);
```



# classで書いてみよう

- classは1つの型を表す
- 例えばベクトルを表すクラスは 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
- 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

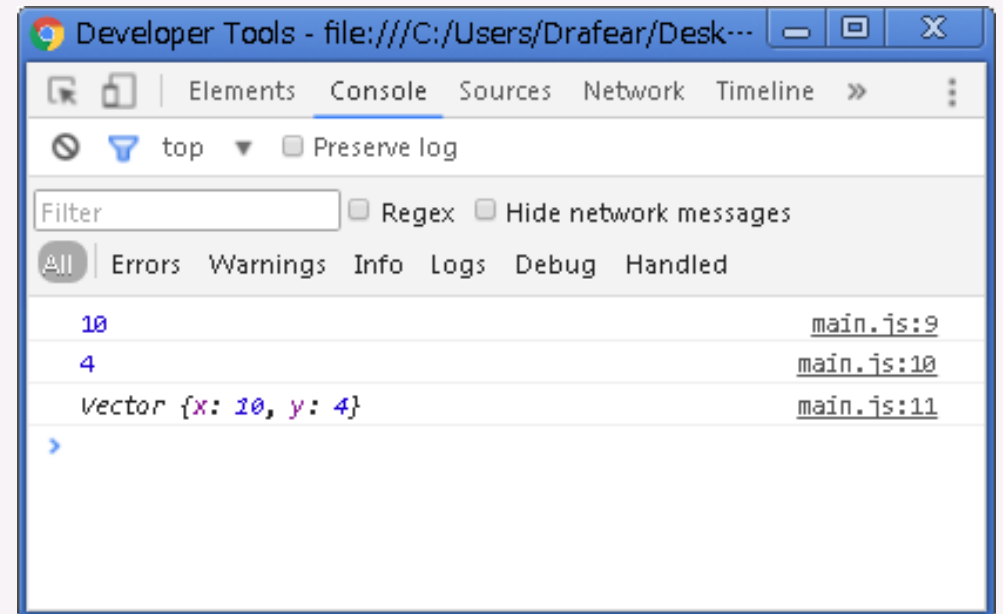
```
main.js  
  
class Vector {  
  constructor(x, y) {  
    this.x = x;  
    this.y = y;  
  }  
}  
  
const v = new Vector(10, 4);  
console.log(v.x);  
console.log(v.y);  
console.log(v);
```



# classで書いてみよう

- 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.変数名** でアクセスする

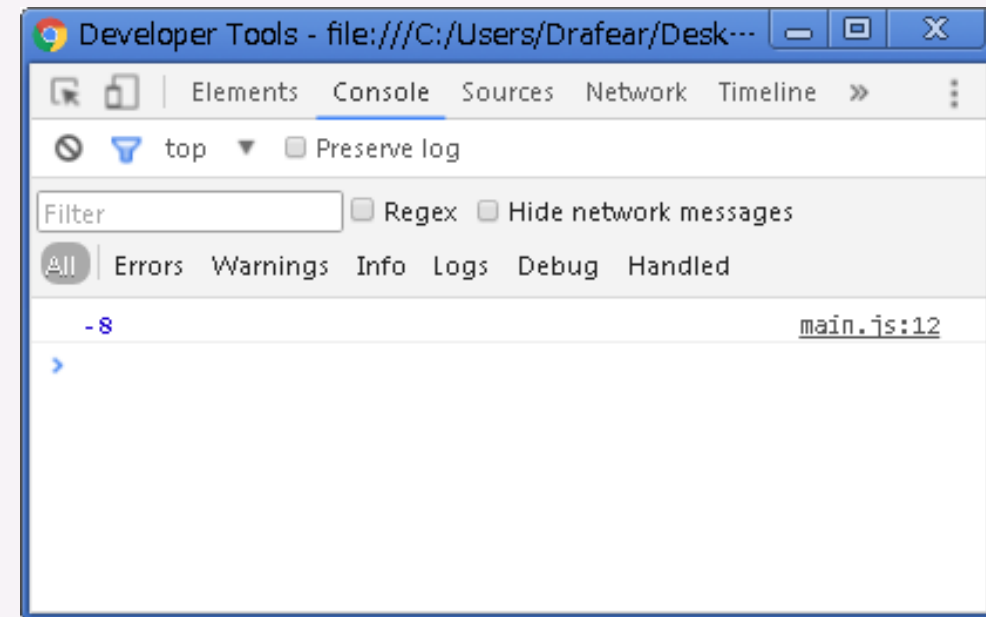
```
main.js  
  
class Vector {  
  constructor(x, y) {  
    this.x = x;  
    this.y = y;  
  }  
}  
  
const v = new Vector(10, 4);  
console.log(v.x);  
console.log(v.y);  
console.log(v);
```



# classで書いてみよう

- 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) );
```

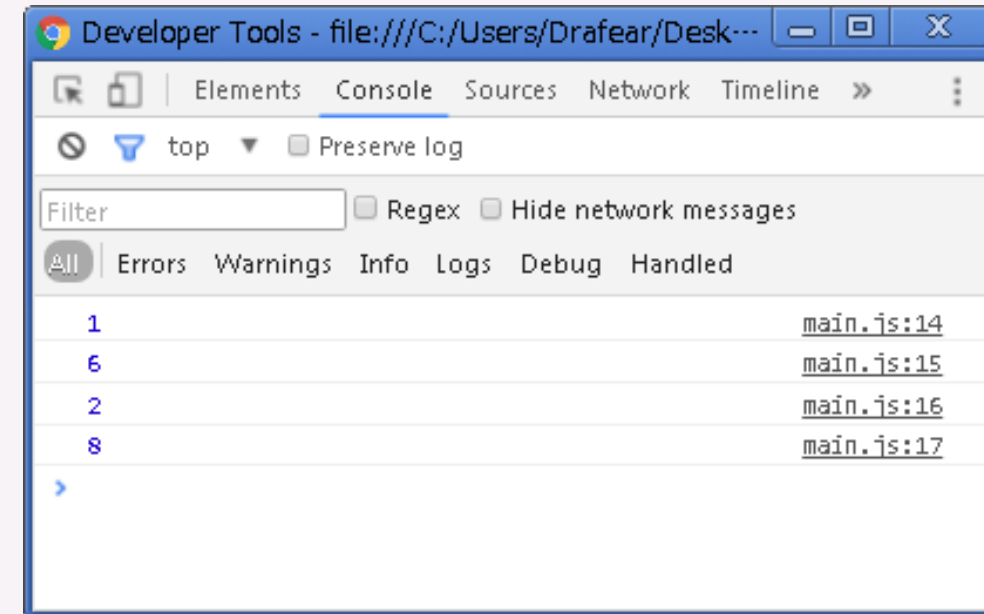


# classで書いてみよう

- 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() );
```





# classで書いてみよう

- プライベート変数 (⇔パブリック変数)
  - 外からアクセス(できるけれども)して欲しくないメンバ変数
  - JavaScript では変数名の前に `_` を付けて表現

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; }  
  getWidth() { return this._w; }  
  getHeight() { return this._h; }  
}
```

# カプセル化

- カプセル化
  - 外からメンバ変数を見えなくし, メソッドのみ提供

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; }  
  getWidth() { return this._w; }  
  getHeight() { return this._h; }  
}
```

# カプセル化

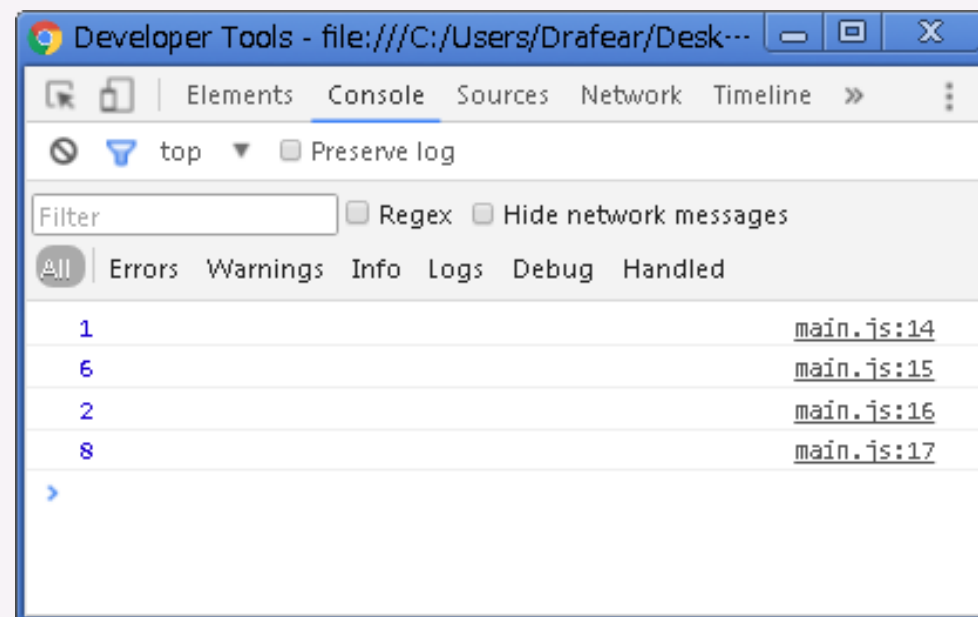
- 例えば, Player に対して, 次のように命令すると動くといったオブジェクトが出来上がる
  - equip(weaponId) . . . 武器を装備
  - heal(val) . . . HPをval回復
  - damage(val) . . . valダメージを受ける
  - attack(enemy) . . . enemyに攻撃
  - isAlive() . . . 生きているか
  - move(direction) . . . directionの方向に移動
- 内部はどうなってるかわからないけど外から命令すれば動く
- 外部からのアクセスと内部実装を分離することで後から変更を加えやすくなる
- 外部に提供するメソッドは機能！

# getter / setter

- `getLeft()` の括弧をとりたい！

main.js

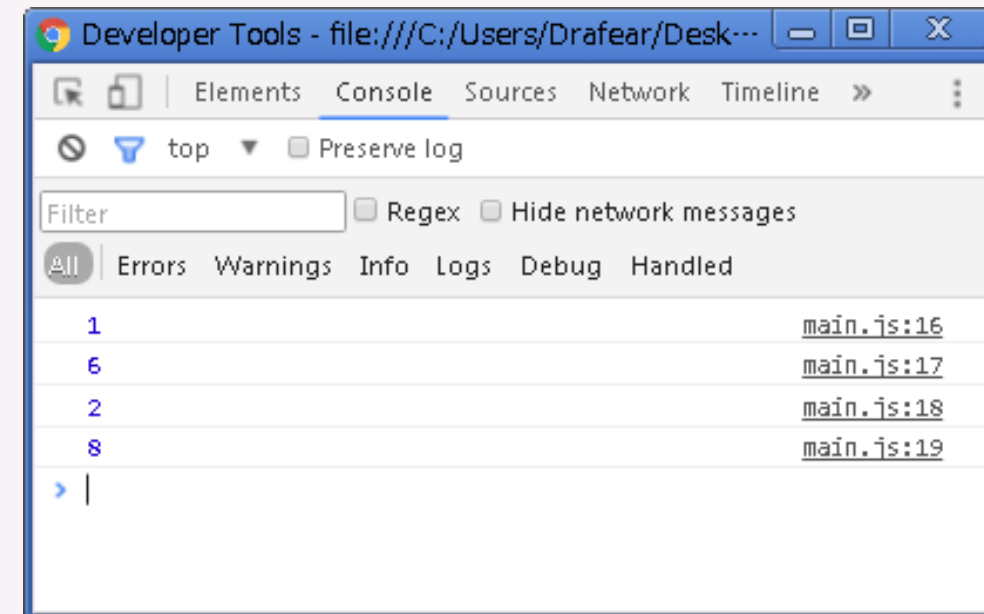
```
class Rect {  
  constructor(x, y, w, h) {  
    this._x = x;  
    this._y = y;  
    this._w = w;  
    this._h = h;  
  }  
  getLeft() { return this._x; }  
  getRight() { return this._x + this._w; }  
  getTop() { return this._y; }  
  getBottom() { return this._y + this._h; }  
  getWidth()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



# getter / setter

- getter . . . 値を取得することだけできる(書き換え不可)

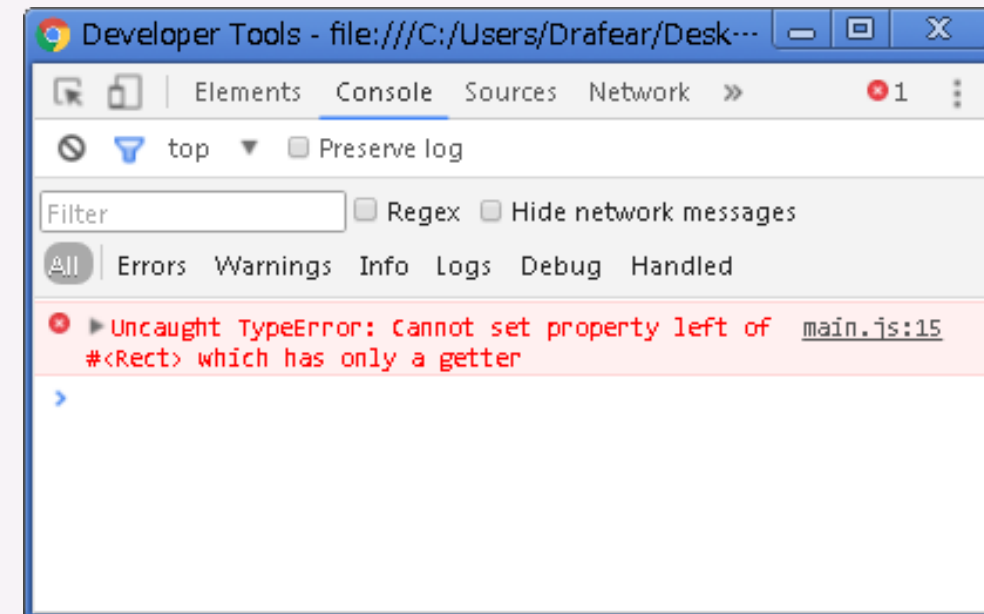
```
main.js  
  
class Rect {  
  constructor(x, y, w, h) {  
    this._x = x;  
    this._y = y;  
    this._w = w;  
    this._h = h;  
  }  
  get left() { return this._x; }  
  get right() { return this._x + this._w; }  
  get top() { return this._y; }  
  get bottom() { return this._y + this._h;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



# 'use strict'

- 'use strict' . . . ミスったらエラー！！ 今後から付けよう！

```
main.js
'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

- setter . . . 値のセットだけできる
  - 値がセットされたらHTML要素にも反映したり

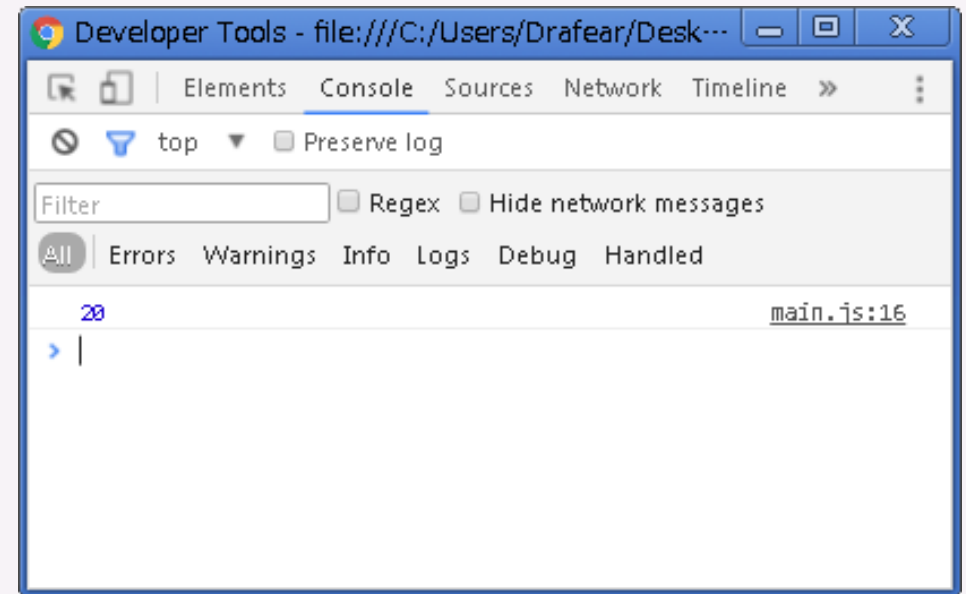
main.js

```
'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;
```

# valueOf

- 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);
```

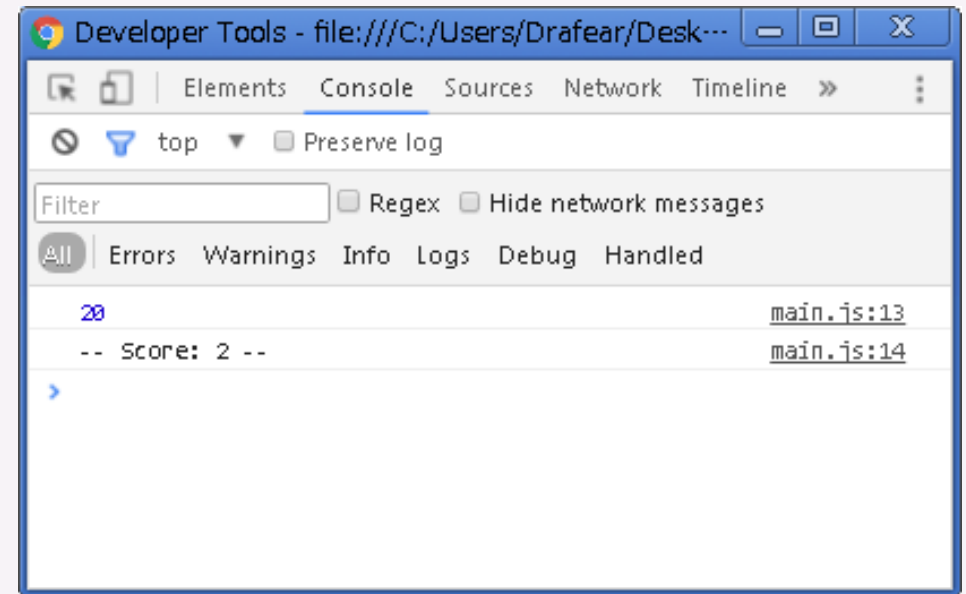




# toString

- toString . . . 文字列になってくれ！ってときになってくれるらしい

```
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} --`);
```

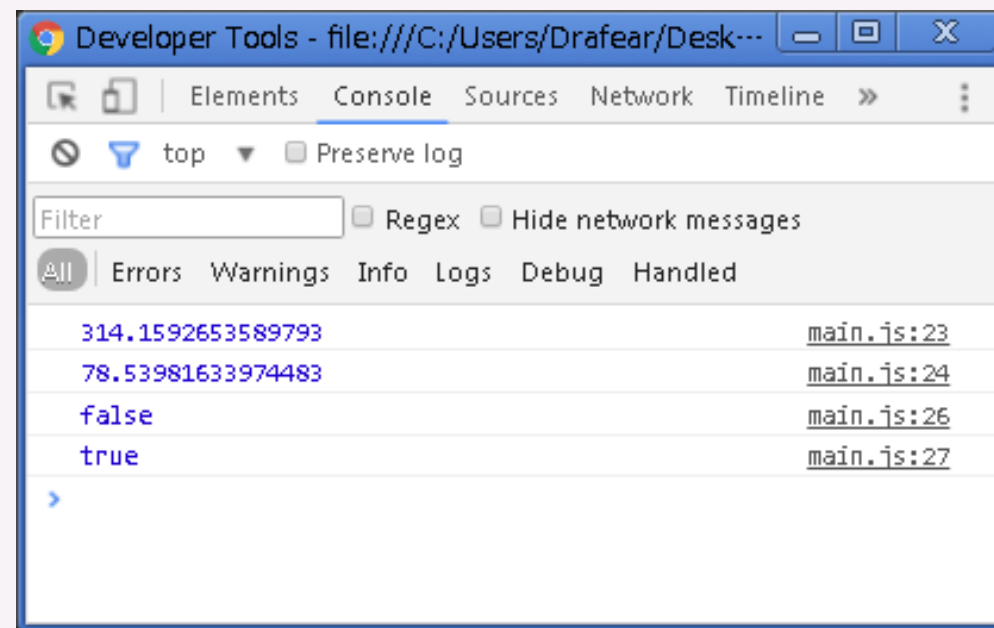


# 演習

- 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;
  }
}
```



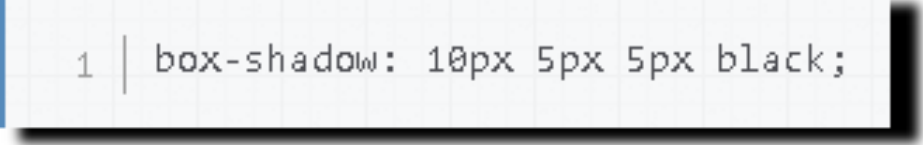
## 2. CSS

# box-shadow

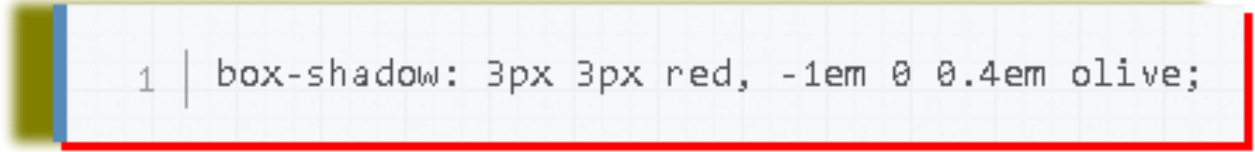
- box-shadow [inset] *dx dy* ぼかし距離 広がり距離 *color*



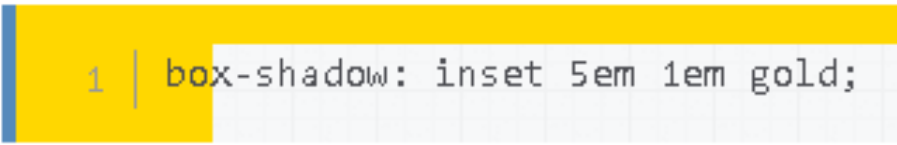
```
1 | box-shadow: 60px -16px teal;
```




```
1 | box-shadow: 10px 5px 5px black;
```



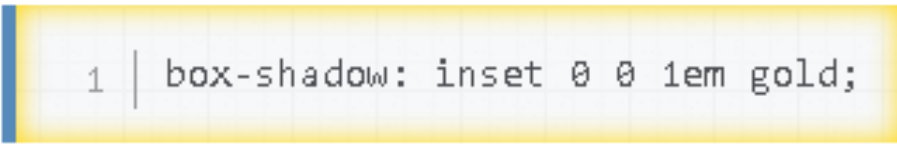
```
1 | box-shadow: 3px 3px red, -1em 0 0.4em olive;
```



```
1 | box-shadow: inset 5em 1em gold;
```



```
1 | box-shadow: 0 0 1em gold;
```



```
1 | box-shadow: inset 0 0 1em gold;
```

# box-shadow

- box-shadow [inset] *dx dy* [(ぼかし距離) [広がり距離] [*color*]
  - inset: 内側に影
  - dx, dy: x方向, y方向のずれ(offset)

```
1 | box-shadow: 60px -16px teal;
```

```
1 | box-shadow: 10px 5px 5px black;
```

```
1 | box-shadow: 3px 3px red, -1em 0 0.4em olive;
```

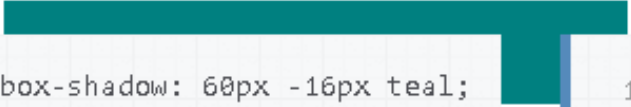
```
1 | box-shadow: inset 5em 1em gold;
```

```
1 | box-shadow: 0 0 1em gold;
```


```
1 | box-shadow: inset 0 0 1em gold;
```

# box-shadow

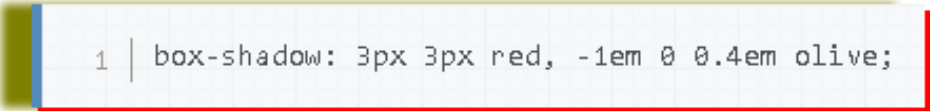
- box-shadow [inset] *dx dy* [(ぼかし距離) [広がり距離] [color]
  - ぼかし距離: グラデーションをかけていく距離 (default: 0)
  - 広がり距離: 同じ色のまま広がっていく距離 (default: 0)
  - color: 色 (default: ブラウザによって異なる)



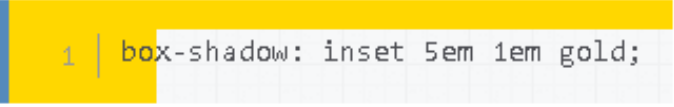
```
1 | box-shadow: 60px -16px teal;
```




```
1 | box-shadow: 10px 5px 5px black;
```



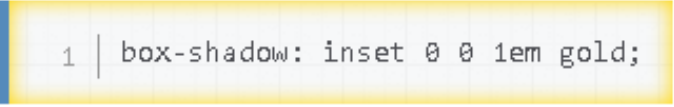
```
1 | box-shadow: 3px 3px red, -1em 0 0.4em olive;
```



```
1 | box-shadow: inset 5em 1em gold;
```



```
1 | box-shadow: 0 0 1em gold;
```



```
1 | box-shadow: inset 0 0 1em gold;
```

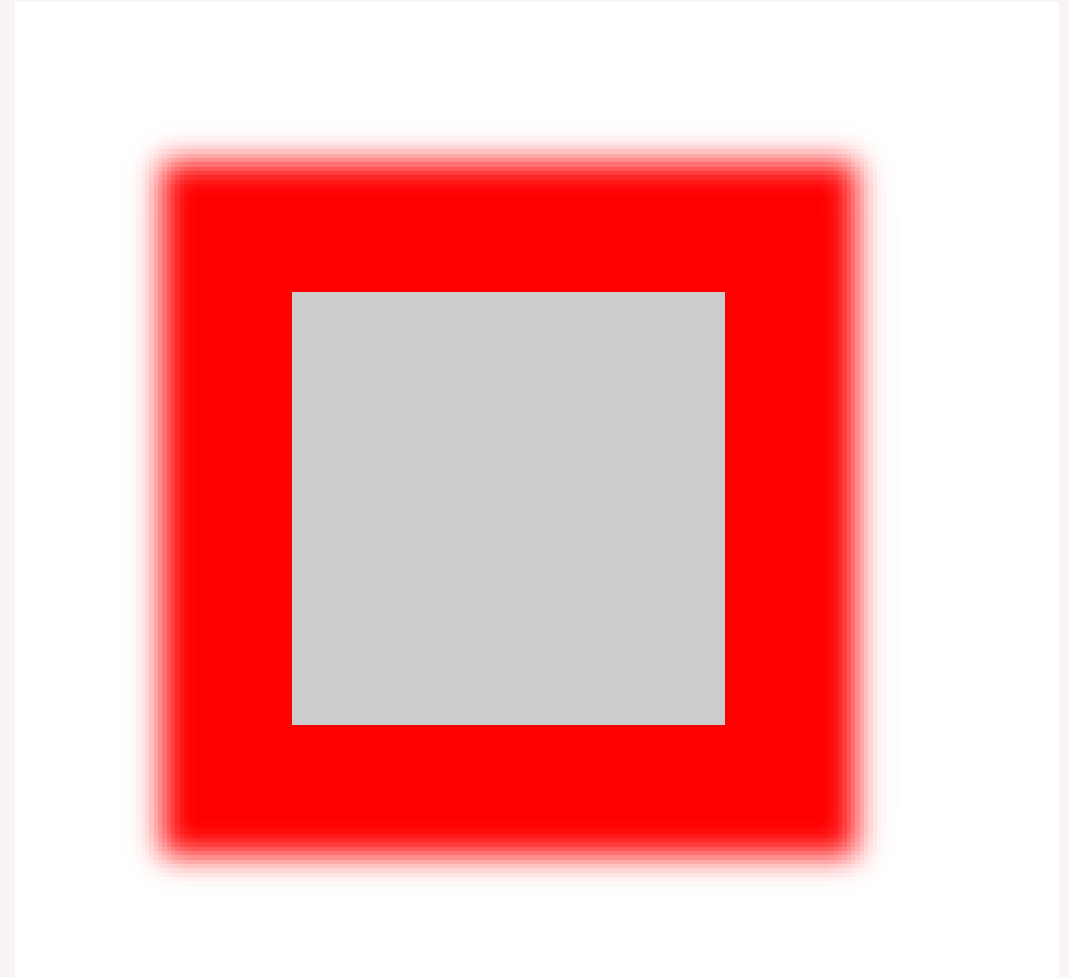
# box-shadow

index.html

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

style.css

```
.box {  
  width: 100px; height: 100px;  
  background-color: #ccc;  
  box-shadow: 0 0 10px 30px red;  
}
```





# box-shadow

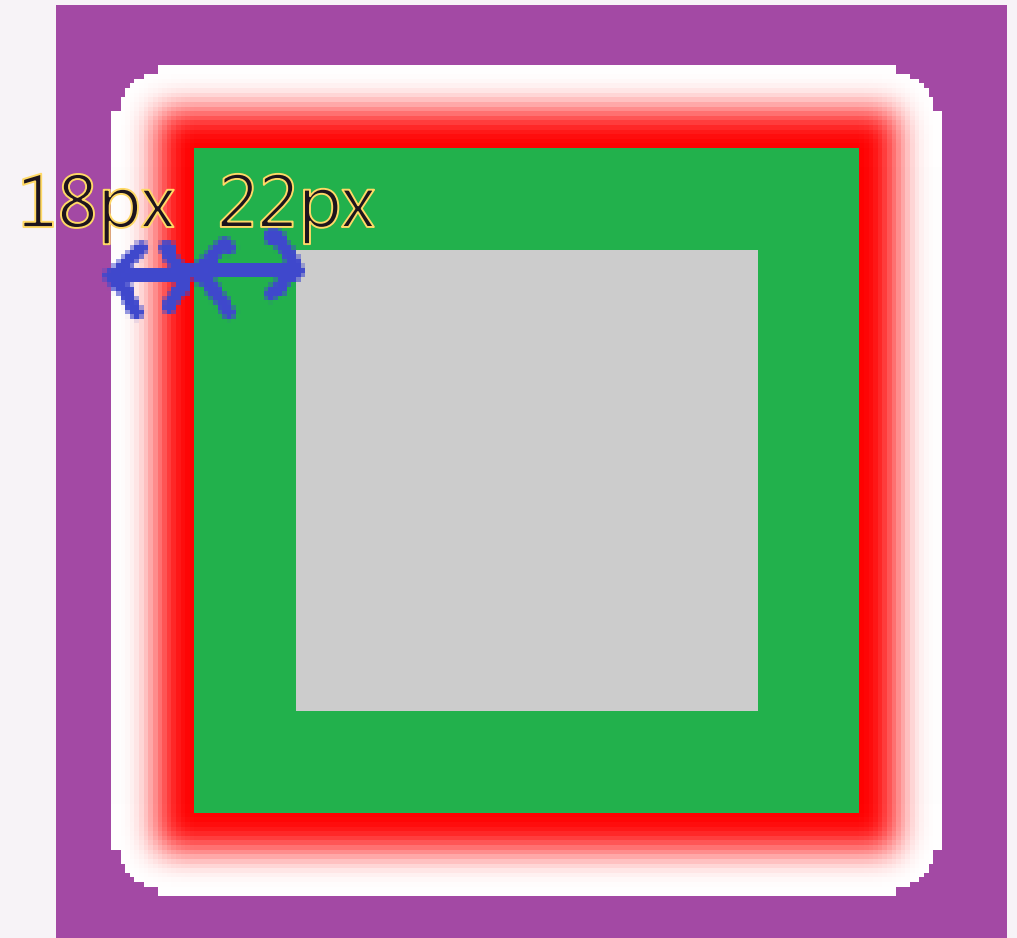
- 全く同じ色のところをペイントで塗りつぶして測ってみた

index.html

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

style.css

```
.box {  
  width: 100px; height: 100px;  
  background-color: #ccc;  
  box-shadow: 0 0 10px 30px red;  
}
```



# position

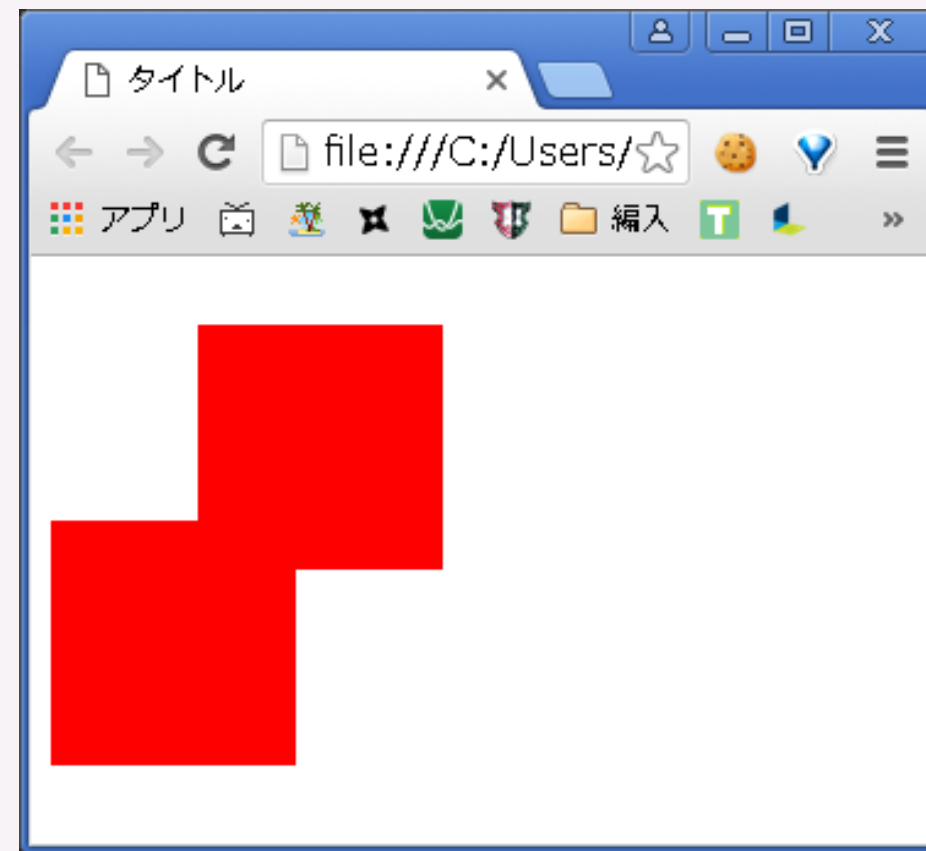
- position: relative;
  - 元の位置からずらす
  - top, bottom, left, right で指定

index.html

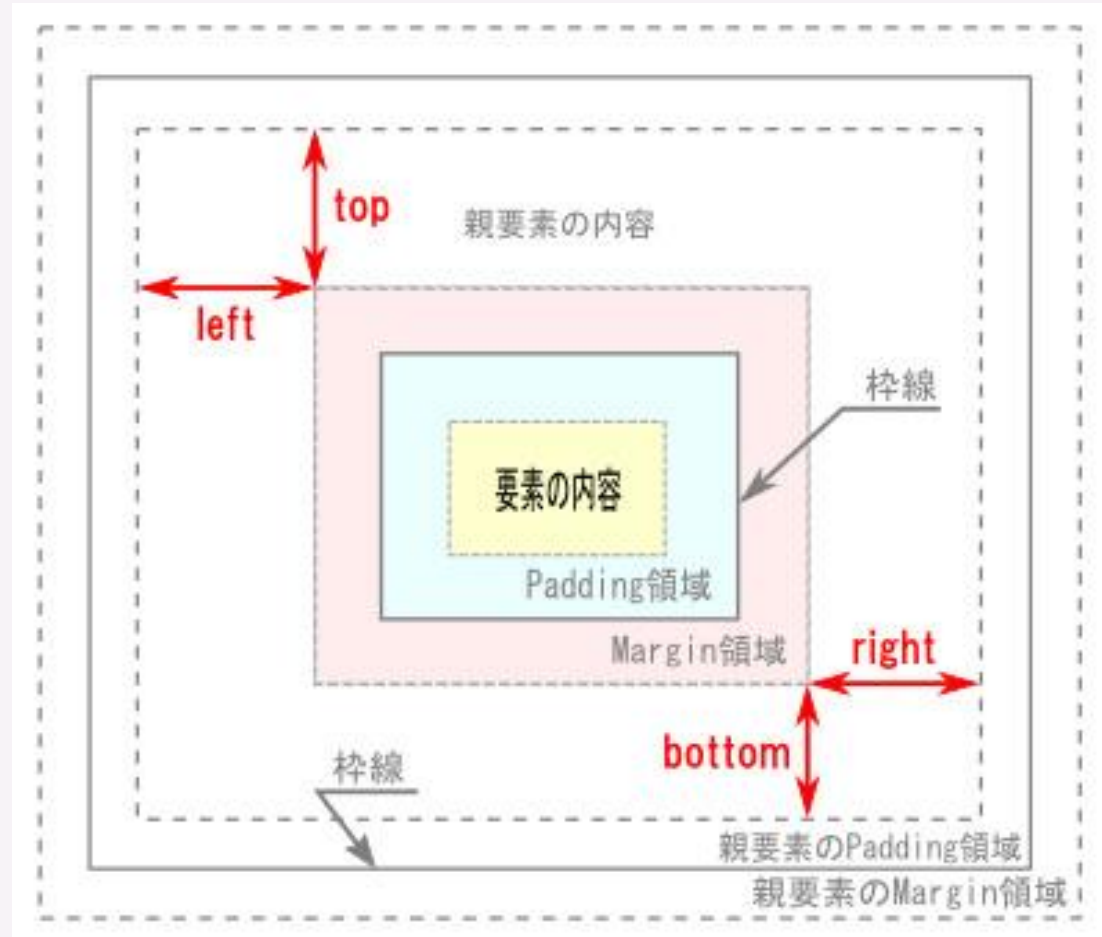
```
<div class="box1"></div>  
<div class="box2"></div>
```

style.css

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



# top, bottom, left, right

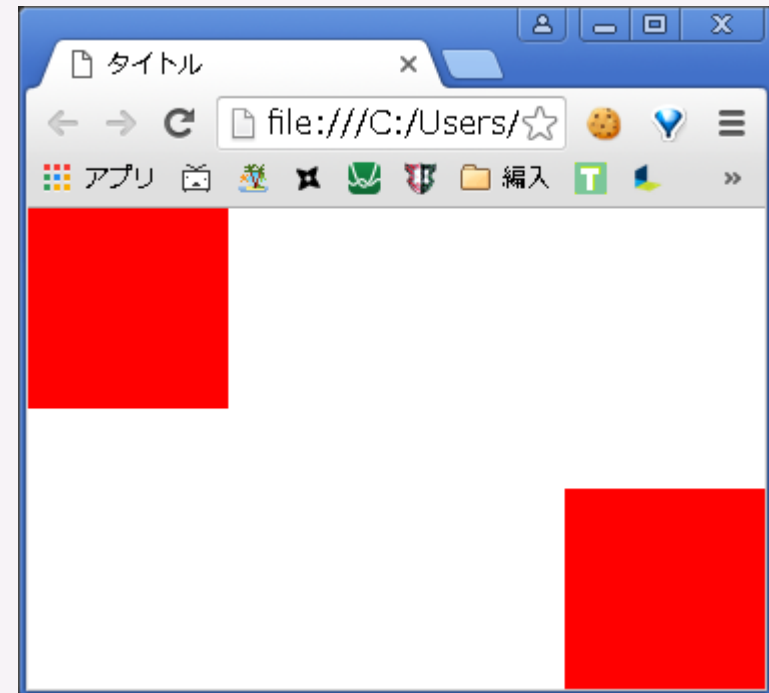


# position

- 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;  
}
```



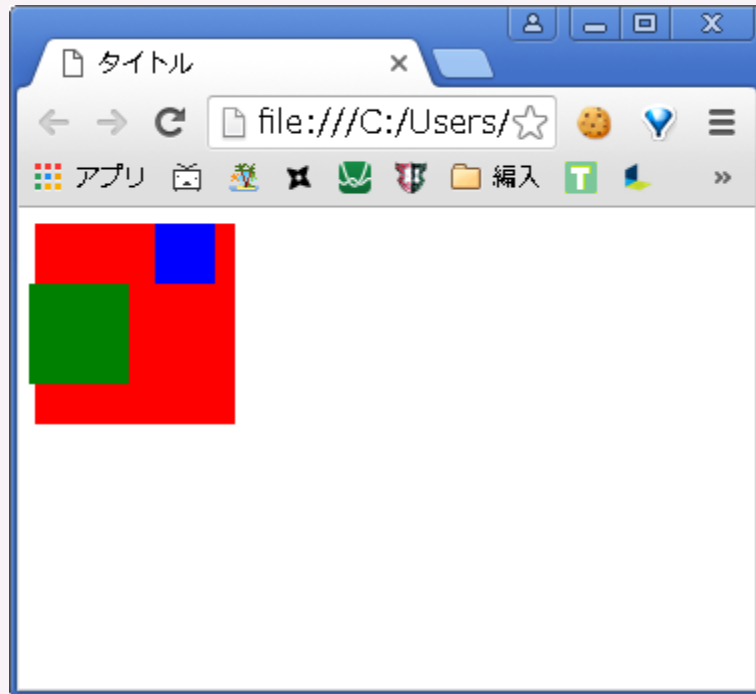
# position

- 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>
```

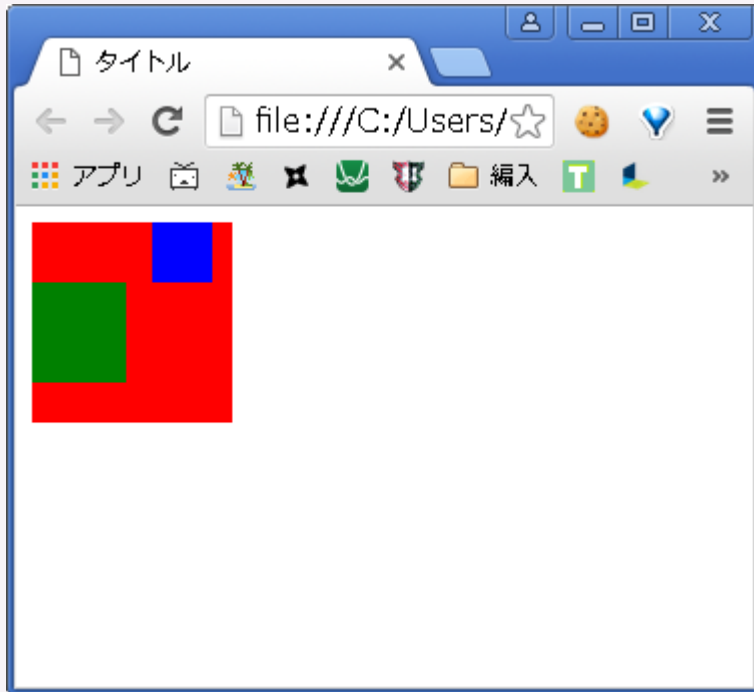


style.css

```
.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.





# 3. DOM操作

# 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

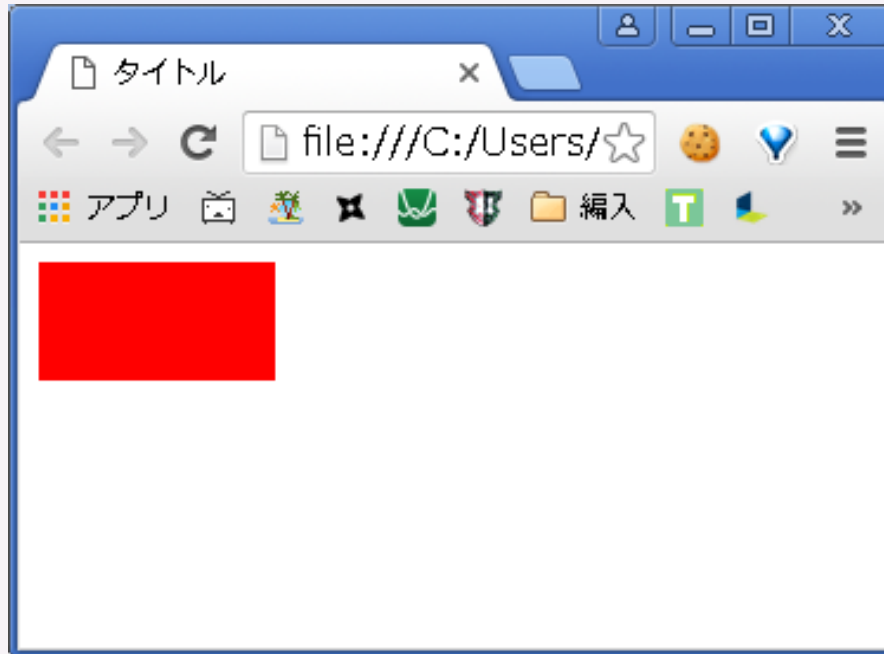
> 0

# 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)";

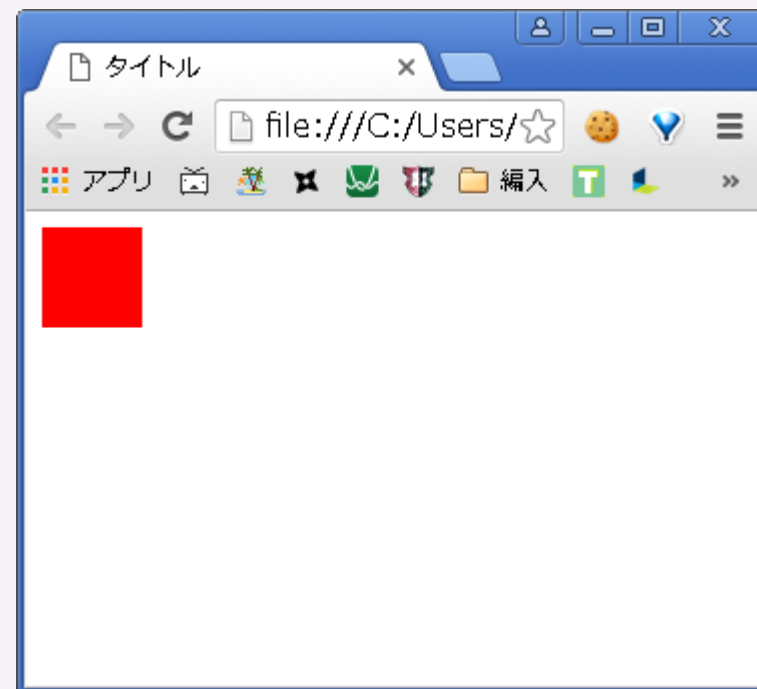
# 動的要素生成

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);
```

