

第3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

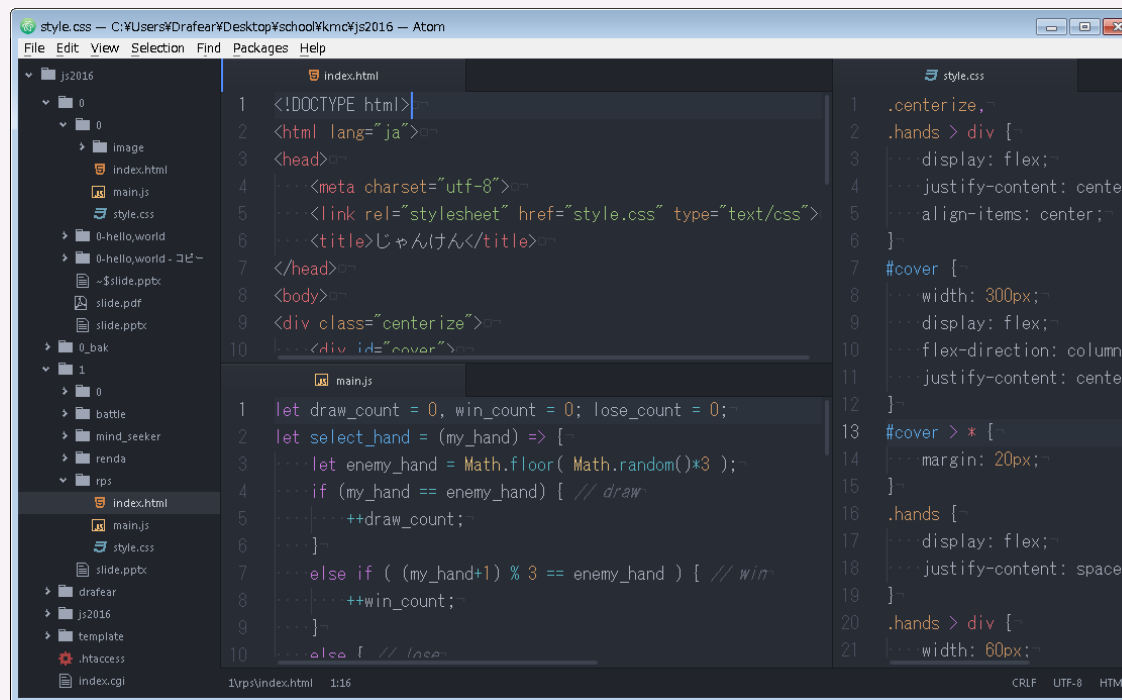
自己紹介

1. KMC 2回生
2. 新入生
3. KMC 3回生
4. KMC 4回生
5. KMC n回生

- 学部学科
- id (入部してたら) or 名前
- 趣味
- 好きなゲーム(あれば)

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の基礎的構文をマスターする
- CSSでレイアウトを組めるようになる

本日の内容

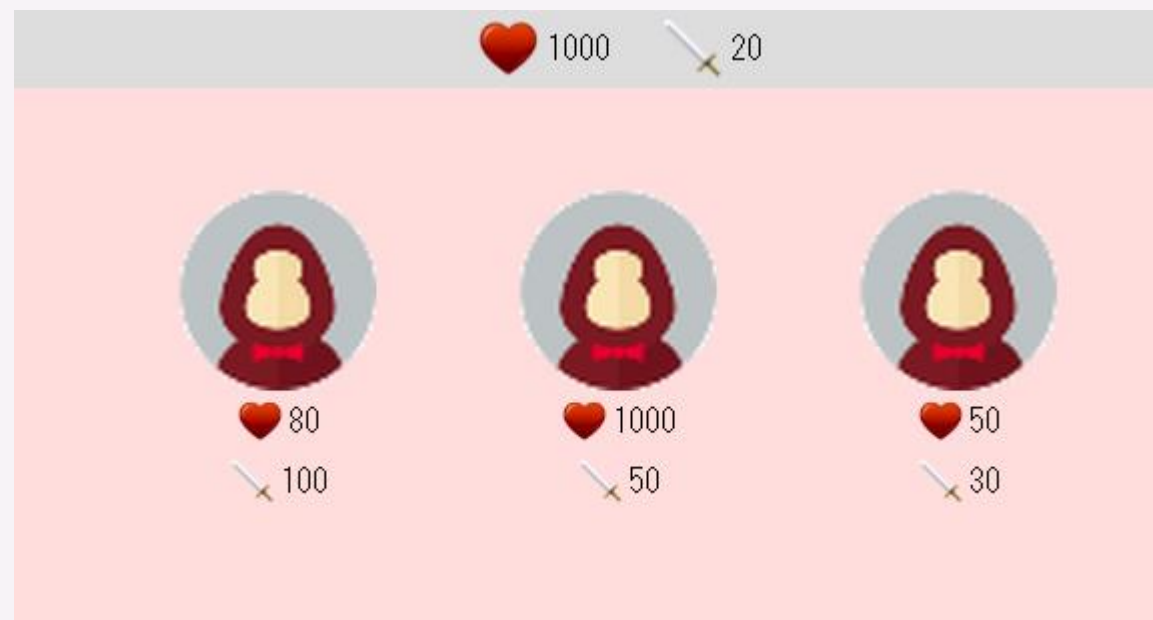
- JavaScript
 - 変数のスコープ
 - クロージャ
 - 再帰
 - 無限ループ
 - while文
 - continue, break
 - for文
 - 配列
 - 配列とfor文
 - elem.children

本日の内容

- CSS
 - セレクタ
 - インラインボックス と ブロックボックス
 - flex
 - CSS3 による装飾
- HTML
 - ul, li, ol
 - input

今日作るもの

- ゴリラを倒すゲーム



てんぷれ

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



1. 復習

復習

- 第1回, 第2回のスライドを見て復習していきます



2. JavaScript

変数のスコープ

- let と定義された行よりも下で、
同じ {} 内もしくはそれよりも {} が深いところから
その変数を読めます
- 同じ名前の変数が複数あれば, {} の深さが近いところ優先
- スコープ = 見える範囲
- 一番外に定義された変数をグローバル変数といいます

変数のスコープ

main.js

```
let a = 10; // グローバル変数
let func = () => {
  let b = 6;
  if (a === 10) {
    let c = 5;
    let a = 5;
    console.log(b); // 6
    console.log(a); // 5
    a = 30;
  }
  console.log(a); // 10
  console.log(c); // error
}
func();
console.log(b); // error
```

クロージャ

- 関数の中の変数を
関数の中の関数の中からも読み出せます
- このように(グローバル変数以外の)外の変数を読み書きする関数をクロージャといいます

main.js

```
let func = (v) => {  
  let a = 10;  
  return () => { // クロージャ  
    return v * a;  
  }  
}  
let f1 = func(10);  
let f2 = func(20);  
console.log( f1() ); // 100  
console.log( f2() ); // 200
```

再帰

- 自分の関数内で自分自身を呼ぶことを再帰といいます

main.js

```
// nの階乗 n! を求める
let fact = (n) => {
  if (n == 0) {
    return 1;
  }
  return n * fact(n-1);
}
console.log( fact(3) ); // 6
console.log( fact(10) ); // 3628800
```


演習

- Math.powを使わずに x^n を計算する関数 power を作ってみよう
 - $x \geq 0, n \geq 0$
 - *let power = (x, n) => { ... }*
- 時間が余った方は power 関数の高速化を考えてみよう

演習

$$x^n = \begin{cases} 0 & (n = 0) \\ x \cdot x^{n-1} & (n \geq 1) \end{cases}$$

計算回数: n 回程度

main.js

```
let power = (x, n) => { //  $x^n$  を計算する  
  if (n == 0) return 1;  
  return power(x, n-1) * x;  
}
```

演習

$$x^n = \begin{cases} 0 & (n = 0) \\ x \cdot x^{n-1} & (n: \text{odd}) \\ (x^{n/2})^2 & (n: \text{even}) \end{cases}$$

計算回数: $\log_2 n$ 回程度

main.js

```
let square = (x) => { // x2 を計算する
  return x * x;
}
let power = (x, n) => { // xn を計算する
  if (n == 0) return 1;
  if (n % 2 == 1) return x * power(x, n-1);
  return square( power(x, n/2) );
}
```

無限ループとbreak

- while (true) { 処理; }
 - 処理を永遠に繰り返す
- break
 - 処理を中断し, while (true) {} を抜ける

例

```
// n! を計算する
let fact = (n) => {
  let res = 1;
  while (true) {
    if (n == 0) break;
    res *= n;
    --n; // n -= 1; と等価
  }
  return res;
}
```

例の動作例

```
> to_odd(8)
> 1
> to_odd(20)
> 5
> to_odd(7)
> 7
```

インクリメント, デクリメント

- インクリメント
++a または a++
a += 1 とほぼ等価
- デクリメント
--a または a--
a -= 1 とほぼ等価

while文

- while (条件式) { 処理; }
 - 次と等価

```
while (true) {  
    if ( !(条件式) ) break;  
    処理;  
}
```

- 条件式 が 真(true) である間 (一連の)処理 を繰り返す
- 処理の途中で条件式が 偽(false) になっても途中では抜けない

while文

- 例

main.js

```
// n! を計算する
let fact = (n) => {
  let res = 1;
  while (n > 0) {
    res *= n;
    --n; // n -= 1; と等価
  }
  return res;
}
```

例の動作例

```
> fact(3)
> 6
> fact(0)
> 1
> fact(10)
> 3628800
```

continue

- while文の } の手前まで処理をスキップする
- ふつう, if文と一緒に使う
 - 次の2つは等価

```
while (条件式1) {  
    処理1;  
    if (条件式2) continue;  
    処理2;  
}
```

```
while (条件式1) {  
    処理1;  
    if ( !(条件式2) ) {  
        処理2;  
    }  
}
```


do ... while文

- do { 処理; } while (条件式);
 - 次と等価 (処理にcontinueを含まない場合)

```
while (true) {  
  処理;  
  if ( !(条件式) ) break;  
}
```

- 初回は条件式の真偽に関わらず実行するwhile文
- あまり使わないのでこんなのあったなーくらいでおk
- (普通のwhile文も今のところは
 こんなのあったなーくらいになると思いますがw)

演習

- 2で割れるだけ割った値を返す関数 `toOdd` を実装してみましょう
 - `let toOdd = (n) => {...}`

例の動作例

```
> toOdd(8)
> 1
> toOdd(20)
> 5
> toOdd(7)
> 7
```

演習

- 2で割れるだけ割った値を返す関数 toOdd を実装してみましょう

main.js

```
// 2で割れるだけ割った値を返す
let toOdd = (n) => {
  while (n % 2 === 0) {
    n /= 2;
  }
  return n;
}
```

for文

- for (初期化処理; 継続条件; ステップ処理) { 処理; }
 - 次と等価 (処理にcontinueを含まない場合)

```
初期化処理;  
while (継続条件) {  
    処理;  
    ステップ処理;  
}
```

1. 初期化処理を行う
2. 継続条件が偽なら終了
3. 処理を行う
break が実行された場合は処理を中断し, 終了
continue が実行された場合は処理を中断し, 4 に移る
4. ステップ処理を行う
5. 2に戻る

for文

- たいいてい, 次の形で使われる
 - i を 0 から n-1 までまわす (各i に対する処理を行う)
 - i によらない処理を n回 実行する
 - for (let i = 1; i <= n; ++i) と 1 から n まで回しても良いが 0 から始めることが多い (後述)

```
for (let i = 0; i < n; ++i) {  
  処理;  
}
```

for文

- 例(九九)

main.js

```
for (let i = 1; i <= 9; ++i) {  
  for (let j = 1; j <= 9; ++j) {  
    console.log(`${i} × ${j} = ${i*j}`);  
  }  
}
```

1 × 1 = 1

1 × 2 = 2

1 × 3 = 3

1 × 4 = 4

1 × 5 = 5

1 × 6 = 6

1 × 7 = 7

1 × 8 = 8

1 × 9 = 9

2 × 1 = 2

2 × 2 = 4

2 × 3 = 6

2 × 4 = 8

2 × 5 = 10

2 × 6 = 12

2 × 7 = 14

2 × 8 = 16

2 × 9 = 18

3 × 1 = 3

3 × 2 = 6

3 × 3 = 9

3 × 4 = 12

配列

- 複数のデータを扱えるもの

main.js

```
let ary = [1, 1, 4, 5, 1, 4];  
console.log(ary[0]); // 1  
console.log(ary[1]); // 1  
console.log(ary[2]); // 4  
console.log(ary[3]); // 5  
console.log(ary[4]); // 1  
console.log(ary[5]); // 4  
console.log(ary); // [1, 1, 4, 5, 1, 4]
```

配列

- 値の読み出し(参照) `ary[i]`
- 値の書き換え(代入) `ary[i] = hoge`
- 要素数 `ary.length`

main.js

```
let ary = [1, 1];  
ary[1] = 5;  
console.log(ary.length); // 2  
console.log(ary); // [1, 5]  
ary[2] = 10;  
console.log(ary.length); // 3  
console.log(ary); // [1, 5, 10]
```


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 – 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)`

配列とfor

- 例

main.js

```
// 配列の各要素の和を計算する
let calcSum = (ary) => {
  let res = 0;
  for (let i = 0; i < ary.length; ++i) {
    res += ary[i];
  }
  return res;
}
```

例の動作例

```
> calcSum([10, 20, 3])
> 33
> calcSum([])
> 0
```

どうでも良い話

```
for (let i = 0; i < ary.length; ++i)
```

VS

```
for (let i = 0; i < ary.length; i++)
```

- ++i の方が一般的に高速 (最近はそうでもない言語が増えてきた)
- i++ って書く人が多いらしい。なんでや。
- i++) の方が shiftキー をカチャカチャしなくて済むからか！？
- 私は ++i って書く老害
- hoge += fuga の感覚で 変数名書いてから ++ 書きたい

演習

1. 配列の中の最小値を求める関数 `minElement` を作ってみよう
 - `let minElement = (ary) => { ... }`
 - 例) `minElement([10, 50, 30, 8, 9])` → 8
2. 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 `catArray` を作ってみよう
 - `let catArray = (ary1, ary2) => { ... }`
 - 例) `catArray([10, 20], [5, 10])` → [10, 20, 5, 10]

演習

1. 配列の中の最小値を求める関数 minElement を作ってみよう

main.js

```
let minElement = (ary) => {  
  let res = ary[0];  
  for (let i = 1; i < ary.length; ++i) {  
    res = Math.min(res, ary[i]);  
  }  
  return res;  
}
```


演習

2. 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 `catArray` を作ってみよう

main.js

```
let catArray = (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 catArray = (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]
```

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: () => { ... },  
};
```



3. CSS

セレクトクの基礎

- .クラス名 { ... } と書いてきましたが .クラス名 以外にも指定の仕方(セレクトク)はいくつかあります
 - idで指定する
 - #id { ... }
 - タグで指定する
 - tag { ... }



index.html

```
<div class="class1">A</div>
<div id="id1">B</div>
<div>C</div>
```

style.css

```
div { border: 3px solid black; }
.class1 { background-color: red; }
#id1 { background-color: green; }
```

セレクトタの基礎

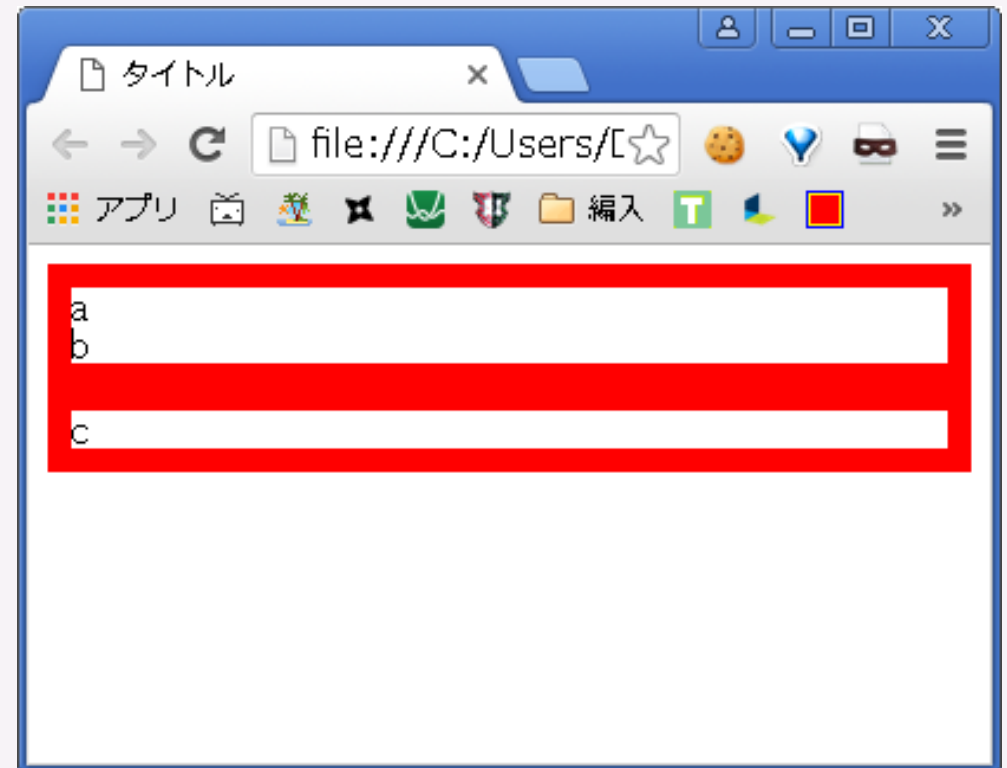
- セレクトタ1 > セレクトタ2
 - セレクトタ1の子でセレクトタ2にマッチするもの
 - 子とはその要素の直下の要素

index.html

```
<div class="class1">  
  <div>  
    <div>a</div>  
    <div>b</div>  
  </div>  
  <div>c</div>  
</div>
```

style.css

```
.class1 > div {  
  border: 10px solid red;  
}
```



セレクトタの基礎

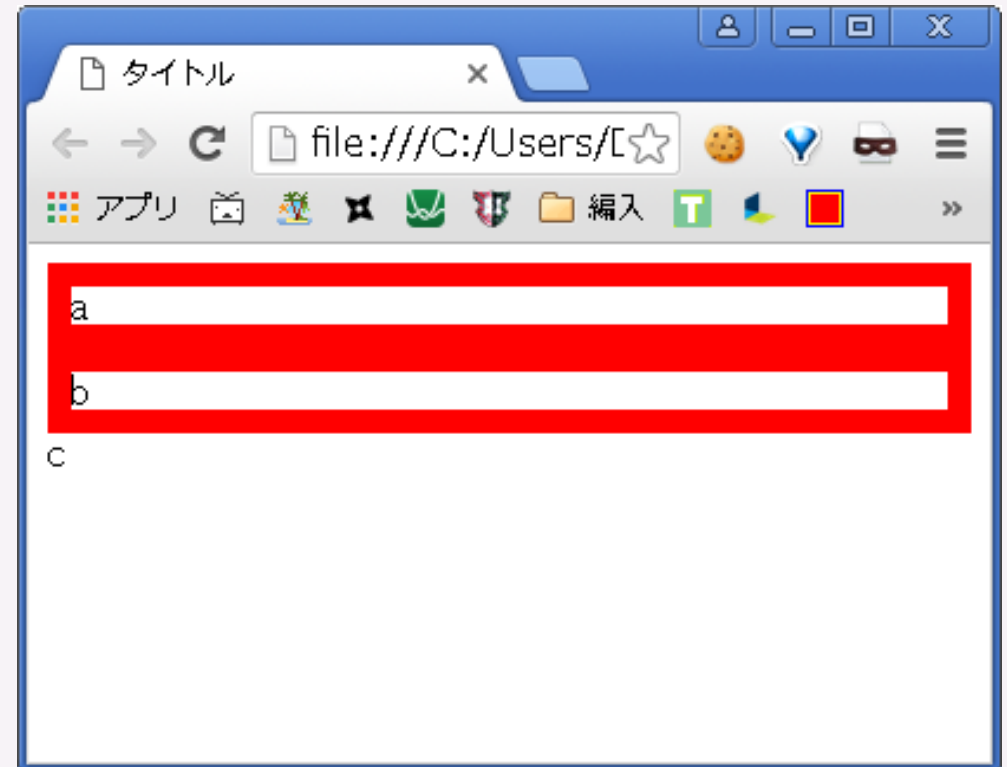
- セレクトタ1 > セレクトタ2
 - セレクトタ1の子でセレクトタ2にマッチするもの
 - 子とはその要素の直下の要素

index.html

```
<div class="class1">  
  <div>  
    <div>a</div>  
    <div>b</div>  
  </div>  
  <div>c</div>  
</div>
```

style.css

```
.class1 > div > div {  
  border: 10px solid red;  
}
```

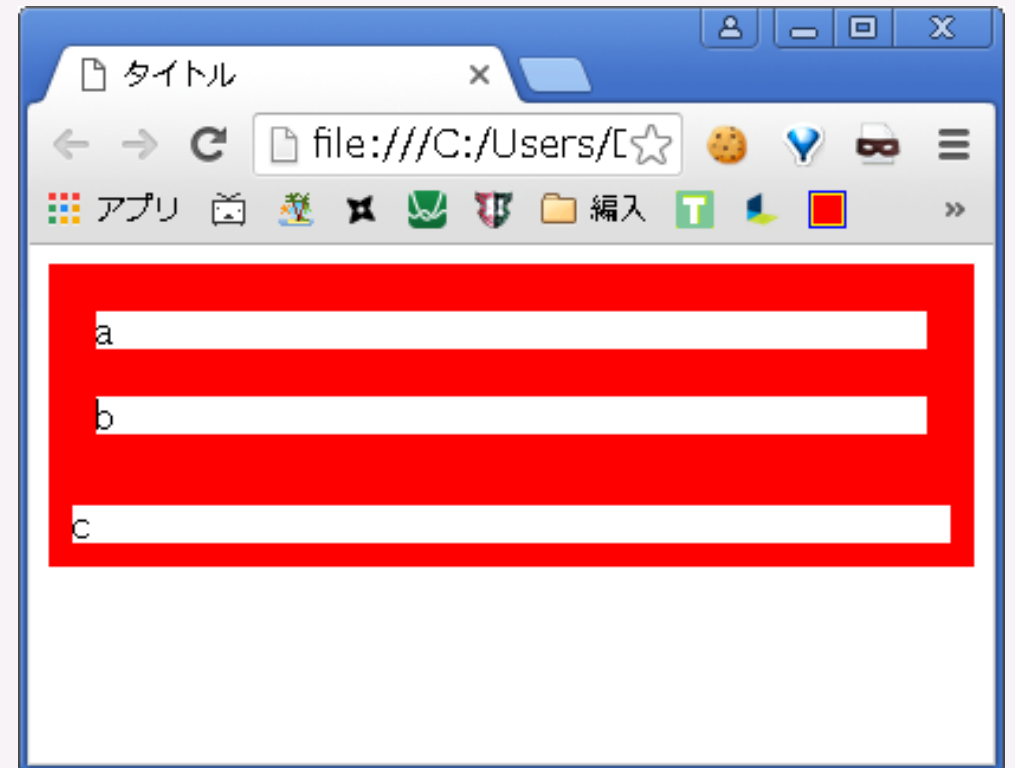


セレクトタの基礎

- セレクトタ1 セレクトタ2
 - セレクトタ1の子孫でセレクトタ2にマッチするもの
 - 子孫とは 子 または 子孫の子 (子, 子の子, 子の子の子, ...)

```
index.html
<div class="class1">
  <div>
    <div>a</div>
    <div>b</div>
  </div>
  <div>c</div>
</div>
```

```
style.css
.class1 div {
  border: 10px solid red;
}
```



セレクトタの基礎

- idやclass名で更なる条件を設定して絞り込む

index.html

```
<div class="select">
  <div>elem1</div>
  <div class="selected">elem2</div>
  <div>elem3</div>
  <div>elem4</div>
</div>
```

style.css

```
.select > div {
  width: 40px; height: 20px;
  border: 1px solid black;
  margin: 10px;
}
.select > div.selected {
  background-color: rgb(200, 220, 255);
}
```



セレクトタの基礎

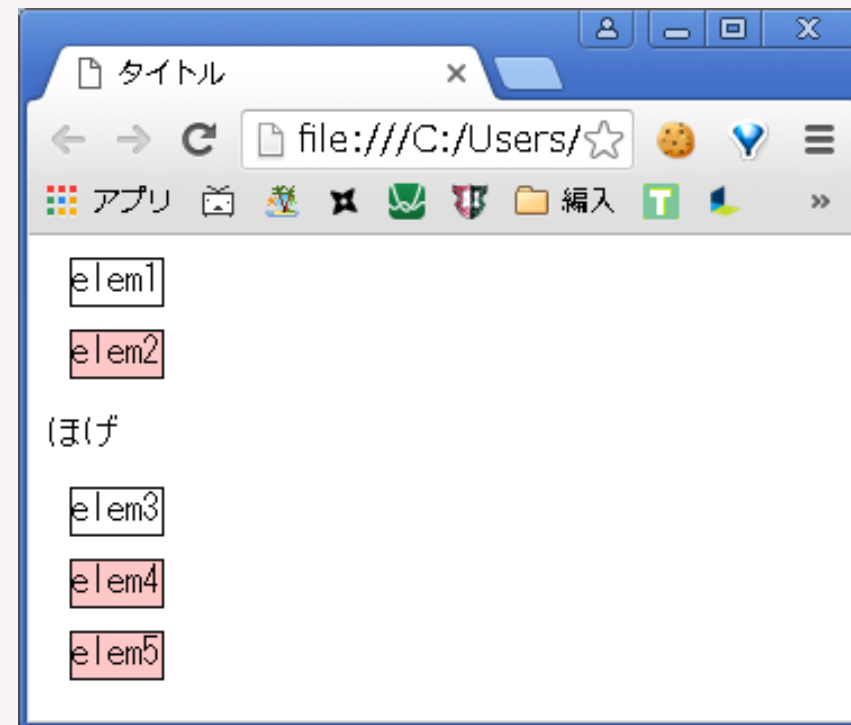
- セレクトタ1 + セレクトタ2
 - 直後の兄弟

index.html

```
<div>elem1</div>
<div>elem2</div>
<p>ほげ</p>
<div>elem3</div>
<div>elem4</div>
<div>elem5</div>
```

style.css

```
div {
  width: 40px; height: 20px;
  border: 1px solid black;
  margin: 10px;
}
div + div {
  background-color: rgb(255, 200, 200);
}
```



セレクトタの基礎

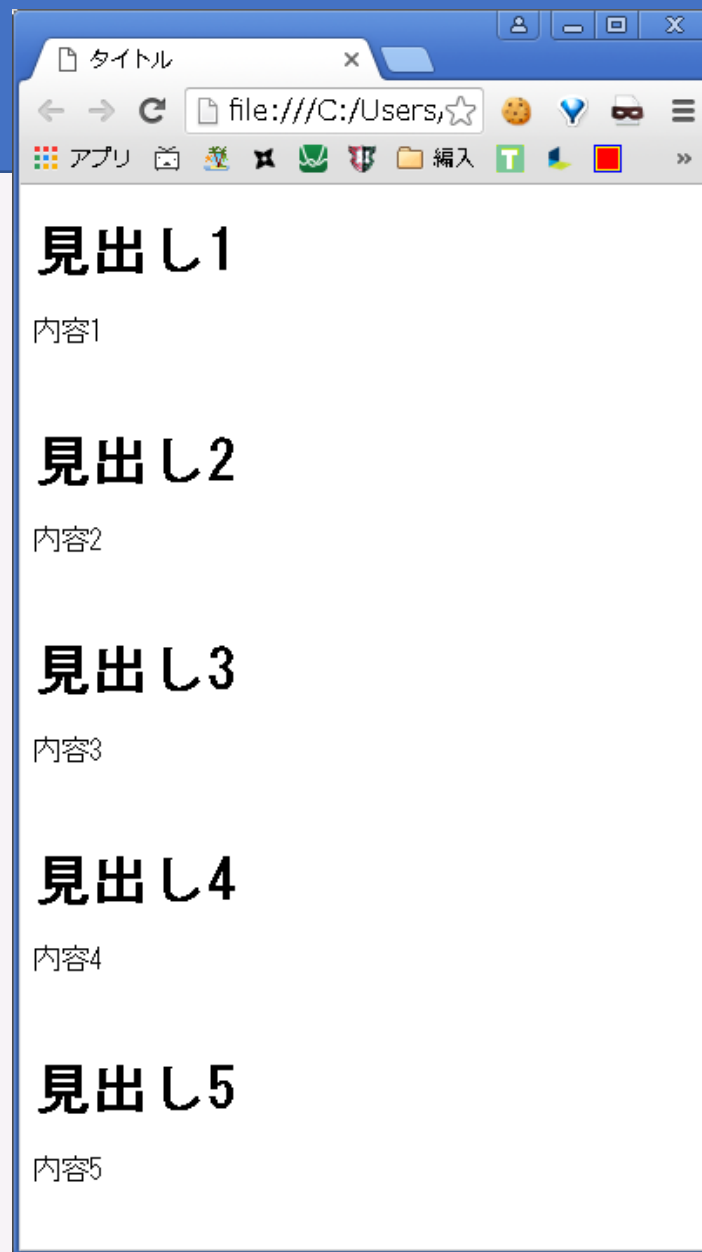
- セレクトタ1 ~ セレクトタ2
 - それ以降の兄弟

index.html

```
<h1>見出し1</h1>
<p>内容1</p>
<h1>見出し2</h1>
<p>内容2</p>
<h1>見出し3</h1>
<p>内容3</p>
<h1>見出し4</h1>
<p>内容4</p>
<h1>見出し5</h1>
<p>内容5</p>
```

style.css

```
h1 ~ h1 {
  margin-top: 50px;
}
```



セレクトタの基礎

- まとめ
 - #id, .className, tagName
 - セレクトタ1 セレクトタ2
 - 子孫
 - セレクトタ1 > セレクトタ2
 - 子(直下)
 - セレクトタ1 + セレクトタ2
 - 直後の兄弟
 - セレクトタ1 ~ セレクトタ2
 - それ以降の兄弟
 - セレクトタ.className, セレクトタ#id
 - 更なる絞り込み

ボックス

- ブロックボックス
 - 要素の直後に改行が入ります
 - width, height, margin, padding の指定ができます
 - 横幅を指定しなければ親要素の100%となります



<https://nulab-inc.com/ja/blog/nulab/css-basics-for-engineer-boxmodel/>

ボックス

- インラインボックス
 - 文章の一部のようにふるまい, 横に並びます
 - `width, height` の指定ができません
 - `margin` は左右方向にしか指定できません
 - `padding` は期待通りに動きません



<https://nulab-inc.com/ja/blog/nulab/css-basics-for-engineer-boxmodel/>

ボックス

- 指定
 - CSS の display プロパティで指定でき, inline または block と指定します

ボックス

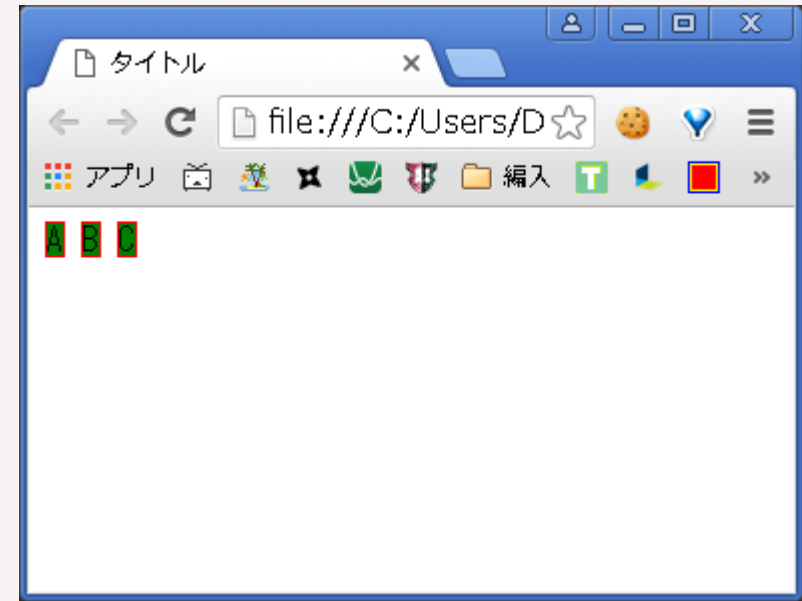
- span タグは display の初期値が inline です
- div タグは display の初期値が block です

index.html

```
<span class="class1">A</span>  
<span class="class1">B</span>  
<span class="class1">C</span>
```

style.css

```
.class1 {  
  background-color: green;  
  border: 1px solid red;  
}
```



ボックス

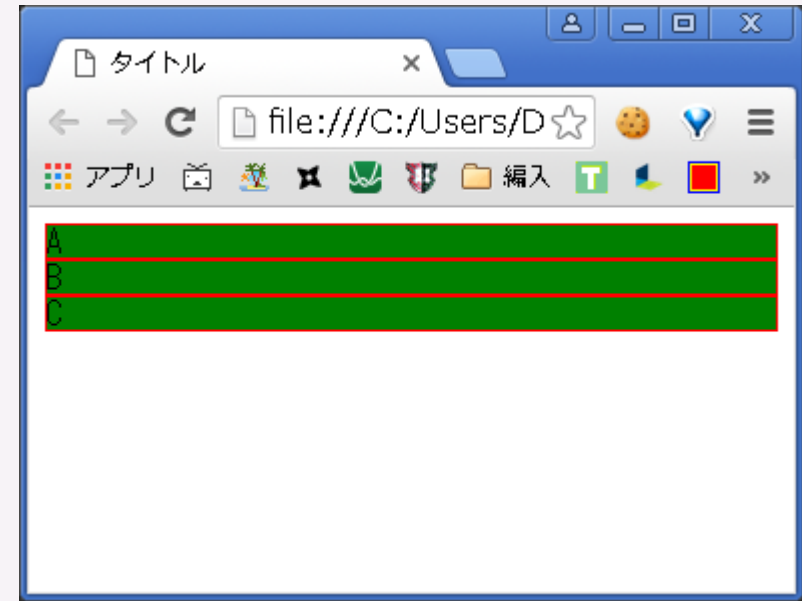
- span タグは display の初期値が inline です
- div タグは display の初期値が block です

index.html

```
<div class="class1">A</div>  
<div class="class1">B</div>  
<div class="class1">C</div>
```

style.css

```
.class1 {  
  background-color: green;  
  border: 1px solid red;  
}
```



ボックス

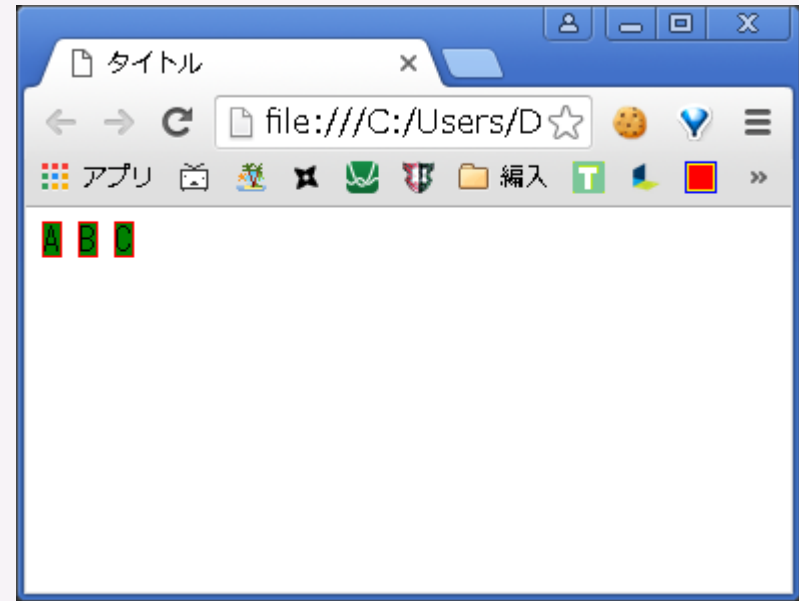
- span タグは display の初期値が inline です
- div タグは display の初期値が block です

index.html

```
<div class="class1">A</div>  
<div class="class1">B</div>  
<div class="class1">C</div>
```

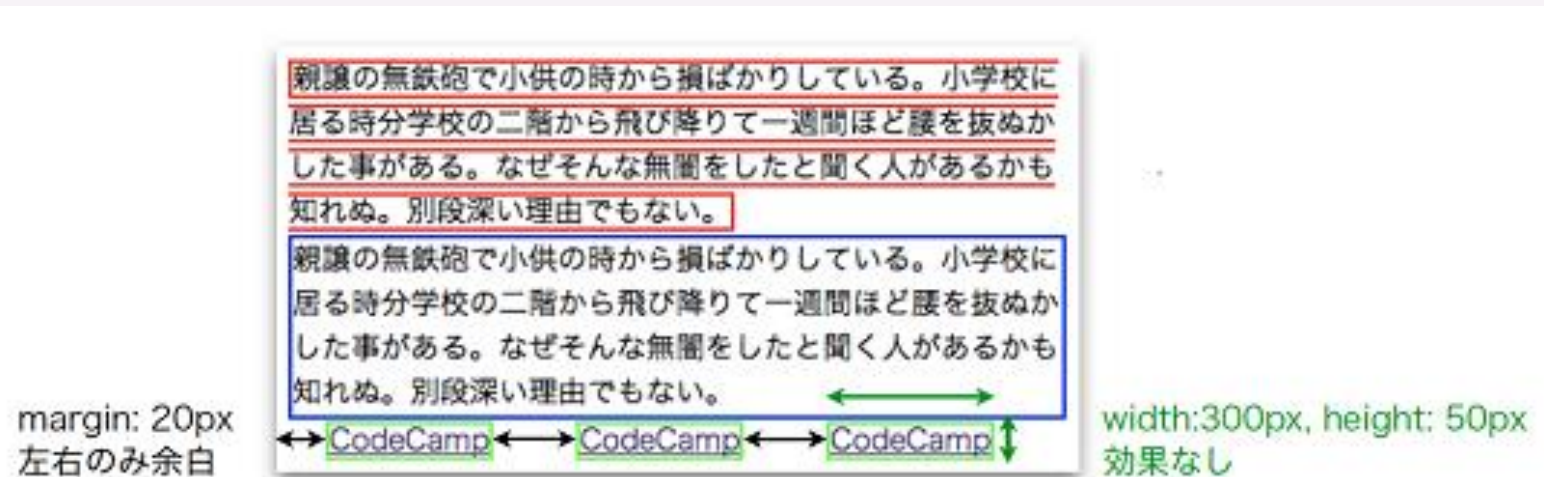
style.css

```
.class1 {  
  display: inline;  
  background-color: green;  
  border: 1px solid red;  
}
```



ボックス

- インラインボックスとブロックボックスの違い
 - 上と下がインライン, 真ん中がブロックです
 - 要は, inline は単に囲むだけ, block は長方形領域を生成する



https://blog.codecamp.jp/block_inline

flexbox

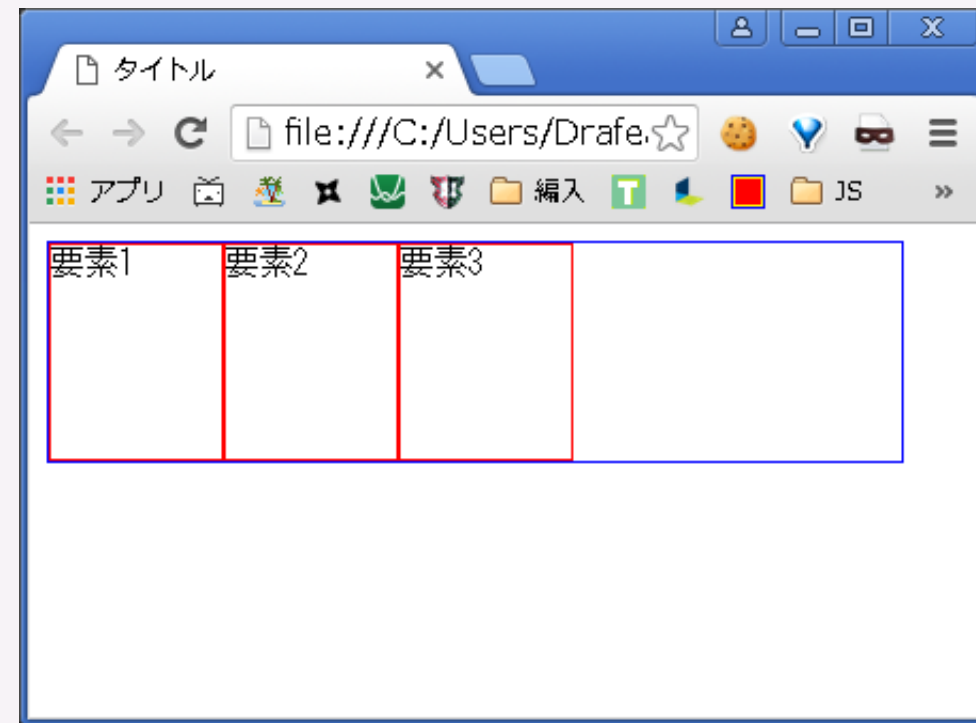
- 至高のレイアウトツール flexbox

index.html

```
<div class="flex-container">  
  <div>要素1</div>  
  <div>要素2</div>  
  <div>要素3</div>  
</div>
```

style.css

```
.flex-container {  
  width: 400px;  
  border: 1px solid blue;  
  display: flex;  
}  
.flex-container > div {  
  width: 80px;  
  height: 100px;  
  border: 1px solid red;  
}
```

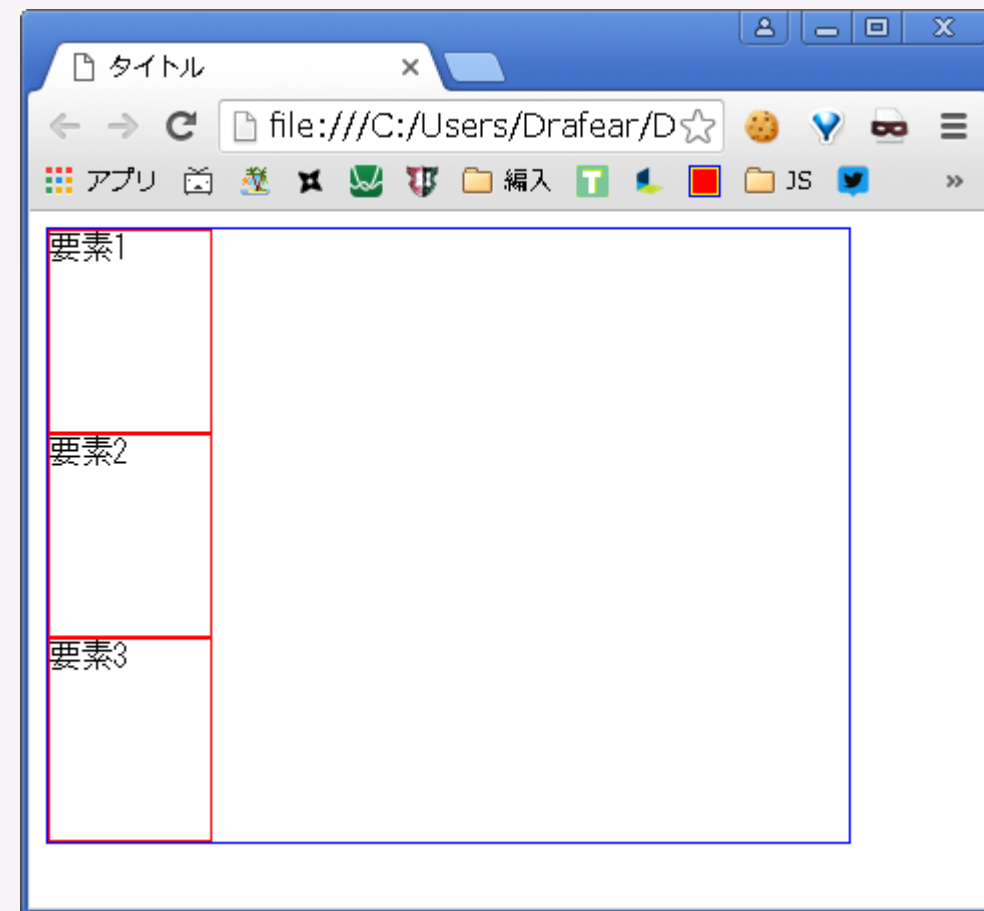


flexbox

- 至高のレイアウトツール flexbox
 - こんな感じで設定していきます

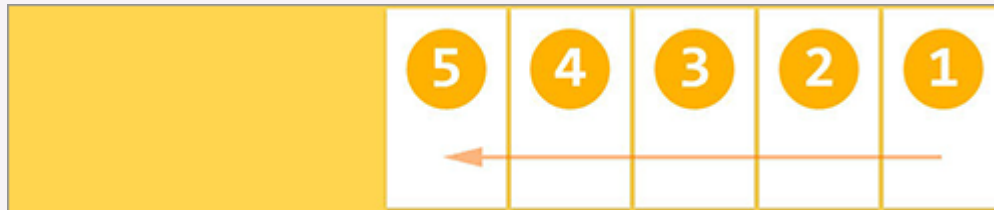
style.css

```
.flex-container {  
  width: 400px;  
  border: 1px solid blue;  
  display: flex;  
  flex-direction: column;  
}
```

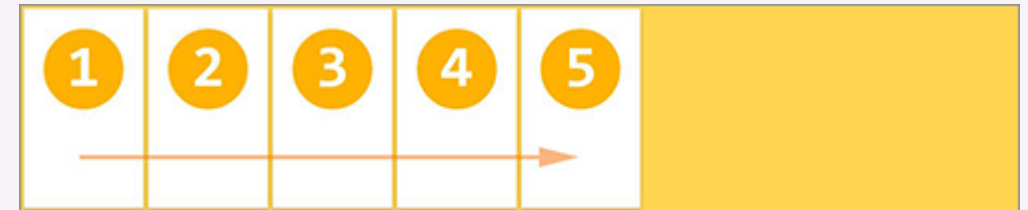


flexbox

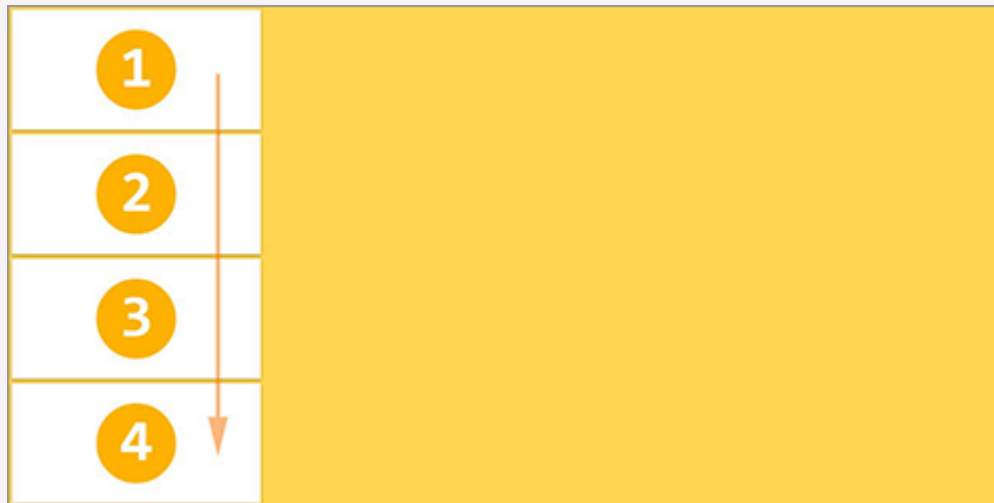
- flex-direction 向き



row(default)



row-reverse



column



column-reverse

flexbox

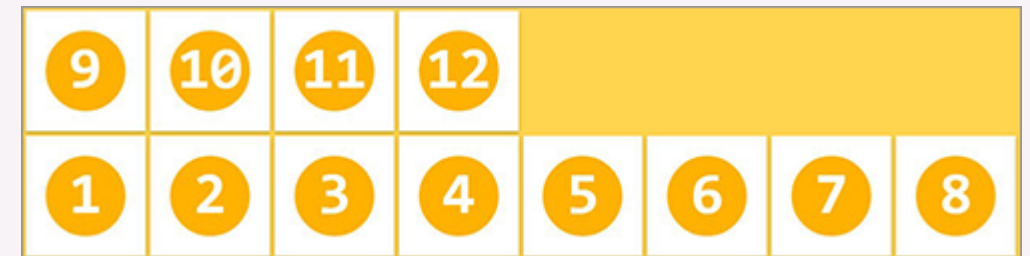
- flex-wrap 折り返し



nowrap(default)



wrap



wrap-reverse

flexbox

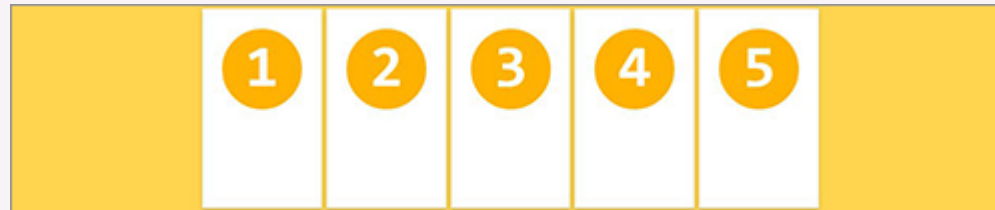
- justify-content 主軸方向の配置方法 (余白の使い方)



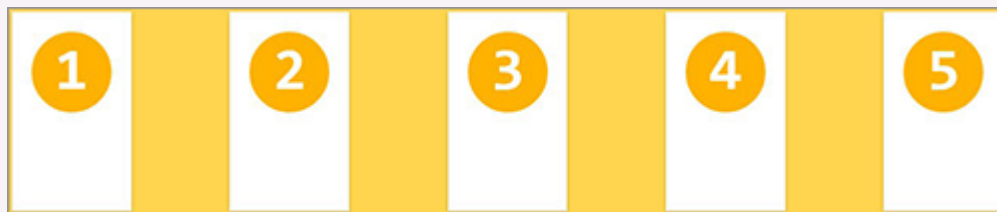
flex-start(default)



flex-end



center



space-between



space-around

flexbox

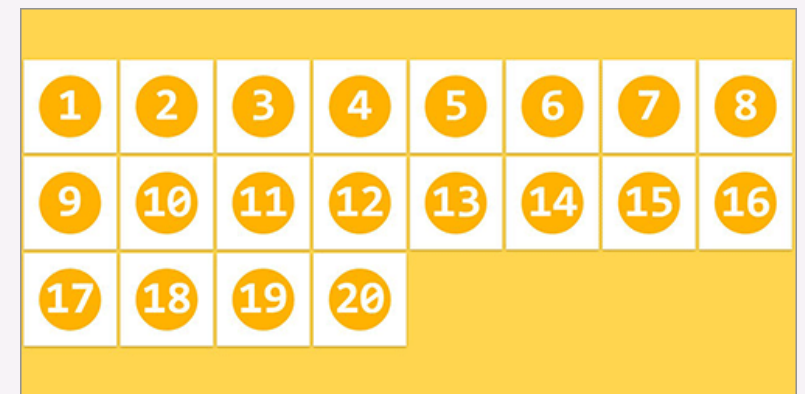
- align-content クロス軸の配置方法 (全体としての余白の使い方)



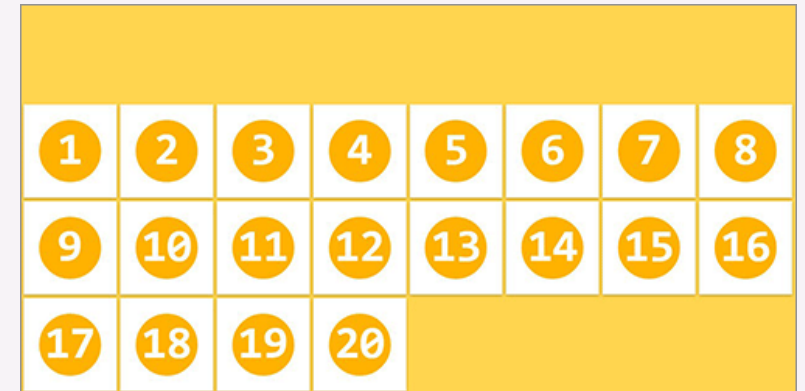
stretch(default)



flex-start



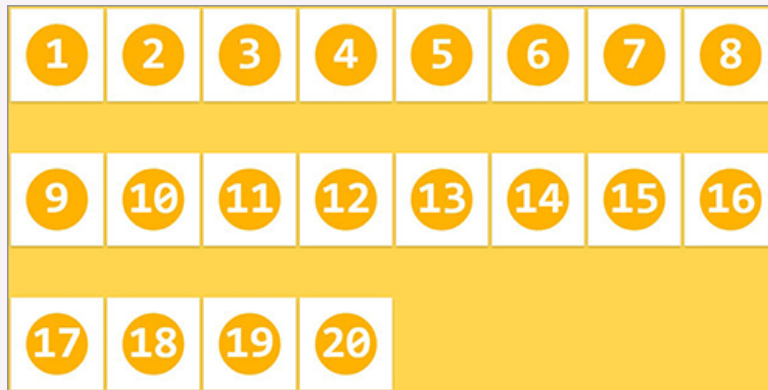
center



flex-end

flexbox

- align-content クロス軸の配置方法 (全体としての余白の使い方)



space-between



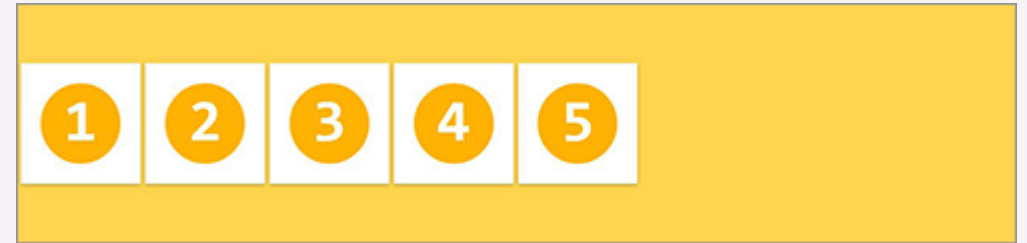
space-around

flexbox

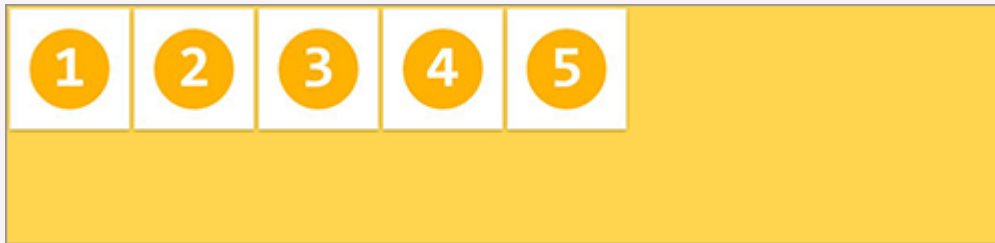
- align-items クロス軸の配置方法 (各行での余白の使い方)



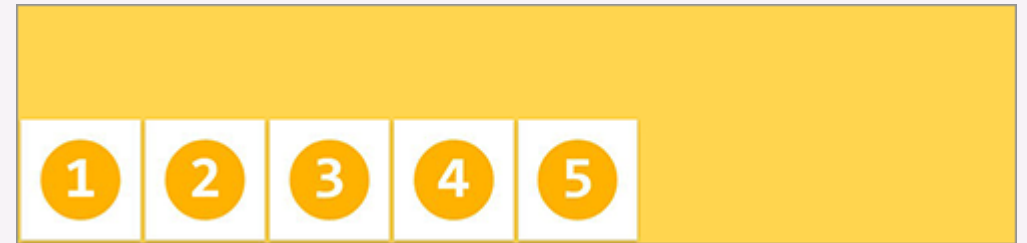
stretch(default)



center



flex-start



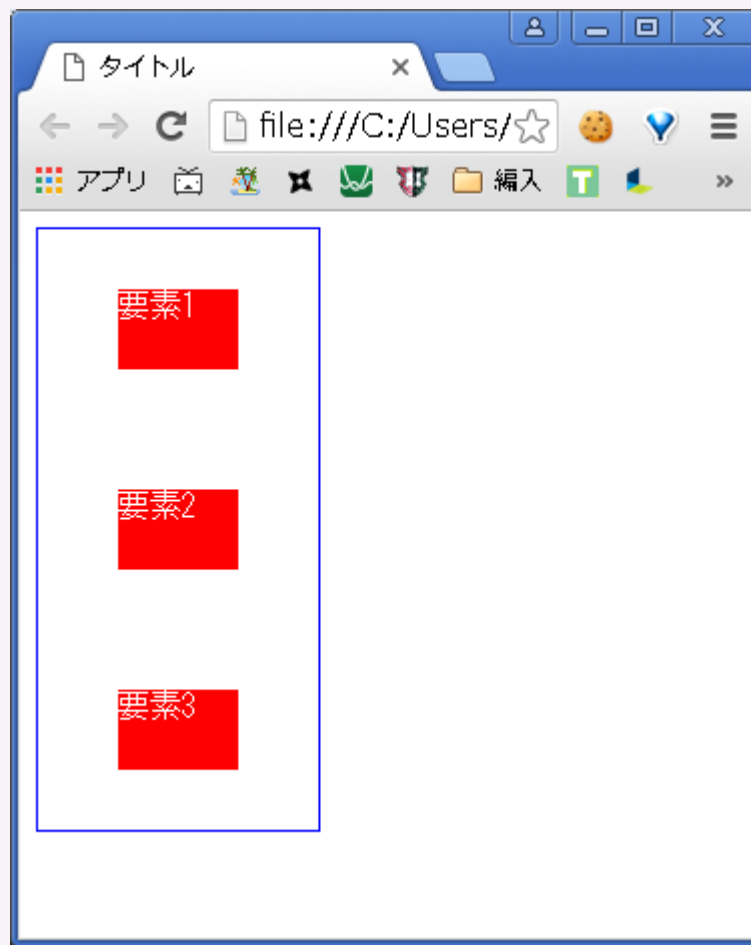
flex-end



baseline

演習

- 次の表示をするものを作ってみよう



演習

index.html

```
<div class="top">
  <div>要素1</div>
  <div>要素2</div>
  <div>要素3</div>
</div>
```

style.css

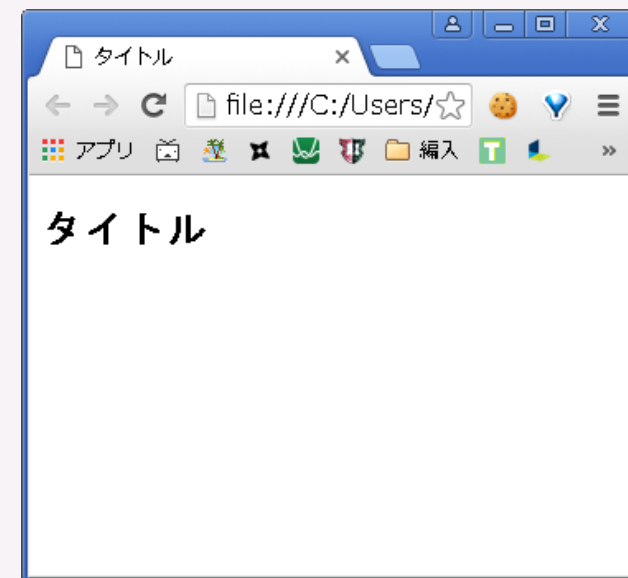
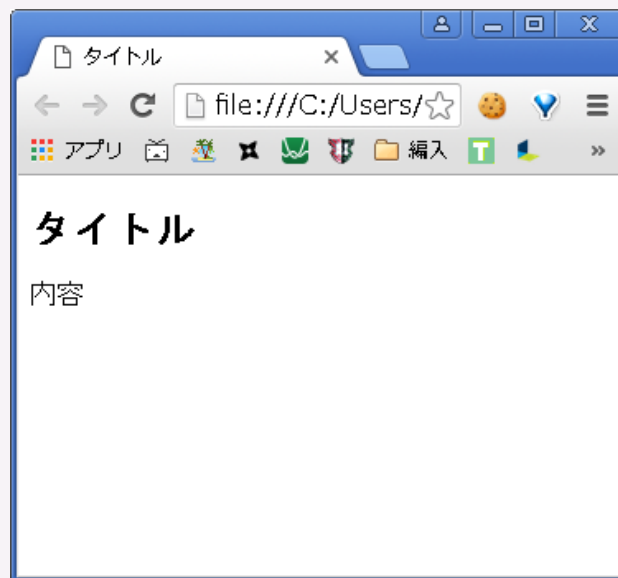
```
.top {
  width: 140px;
  height: 300px;
  border: 1px solid blue;
  display: flex;
  flex-direction: column;
  justify-content: space-around;
  align-items: center;
}
.top > div {
  width: 60px;
  height: 40px;
  background-color: red;
  color: white;
}
```

演習(復習)

- タイトル<h2>～</h2>をクリックすると内容<div>～</div>の表示非表示が切り替わるようにしてみよう
 - index.html も適宜編集して下さい

index.html

```
<h2>タイトル</h2>  
<div>内容</div>
```



演習(復習)

- タイトル<h2>～</h2>をクリックすると内容<div>～</div>の表示非表示が切り替わるようにしてみよう

index.html

```
<h2 id="title">タイトル</h2>  
<div id="content">内容</div>
```

style.css

```
.hide {  
  display: none;  
}
```

main.js

```
document.getElementById("title").addEventListener("click", (e) => {  
  document.getElementById("content").classList.toggle("hide");  
});
```


一般化しよう！

- 次の実装を考える
 - showHide クラスが設定された要素の子には h2要素 と p要素 がこの順で1つずつ並んでいる
 - h2要素 がクリックされると p要素 の 表示/非表示 が反転する
 - p要素 は div要素 と似ているが, "段落" の意味を持ちデフォルトで margin を持つ

index.html

```
<div class="showHide">  
  <h2>見出し</h2>  
  <p>内容</p>  
</div>
```

show / hide

- とりあえず css

index.html

```
<div class="showHide">  
  <h2>見出し</h2>  
  <p>内容</p>  
</div>
```

style.css

```
.showHide > h2 {  
  cursor: pointer;  
}  
.showHide > h2:hover {  
  text-decoration: underline;  
}  
.hide {  
  display: none;  
}
```

show / hide

- js部分

- document.getElementsByClassName(className)
 - そのクラス名を持つ要素を配列で得る
- elem.querySelector(selector)
 - cssのセレクタを渡すと, その要素以下でそのセレクタにマッチする最初の要素を得る

index.html

```
<div class="showHide">  
  <h2>見出し</h2>  
  <p>内容</p>  
</div>
```

main.js

```
let toggleShowHide = (e) => { ... } // show/hide 切り替え  
let initShowHide = () => { // クリックイベントを設定する  
  let elems = document.getElementsByClassName("showHide");  
  for (let i = 0; i < elems.length; ++i) {  
    elems[i].querySelector("h2").addEventListener("click", toggleShowHide);  
  }  
}  
initShowHide();
```

show / hide

- js部分

- document.[getElementsByClassName](#)(className)
 - そのクラス名を持つ要素を配列で得る
- document.[getElementsByTagName](#)(tagName)
 - 指定したタグ名の要素を配列で得る
- elem.[querySelector](#)(selector)
 - cssのセレクタを渡すと, その要素以下でそのセレクタにマッチする最初の要素を得る
- elem.[querySelectorAll](#)(selector)
 - cssのセレクタを渡すと, その要素以下でそのセレクタにマッチする全ての要素を配列で得る

show / hide

- js部分

- elem.parentElement
 - 親要素

index.html

```
<div class="showHide">  
  <h2>見出し</h2>  
  <p>内容</p>  
</div>
```

main.js

```
let toggleShowHide = (e) => { // show/hide 切り替え  
  e.currentTarget.parentElement.querySelector("p").classList.toggle("hide");  
}  
let initShowHide = () => { // クリックイベントを設定する  
  let elems = document.getElementsByClassName("showHide");  
  for (let i = 0; i < elems.length; ++i) {  
    elems[i].querySelector("h2").addEventListener("click", toggleShowHide);  
  }  
}  
initShowHide();
```

show / hide

- js部分
 - elem.`parentElement`
 - 親要素
 - elem.`children`
 - 子要素 (配列)

show / hide

- 使用例

index.html

```
<div class="showHide">
  <h2>見出し1</h2>
  <p>内容1</p>
</div>
<div class="showHide">
  <h2>見出し2</h2>
  <p>内容2</p>
</div>
<div class="showHide">
  <h2>見出し3</h2>
  <p>内容3</p>
</div>
```

show / hide

- 使用例

- div は意味を持たない
- html要素に意味を持たせるとwebページを巡回するプログラムなどが読みやすい
- articleはそれ1つで独立したコンテンツその部分だけで1つのコンテンツとして機能する
- headerはヘッダ
- sectionはそれ1つで独立したコンテンツにはならないが, 見出しをつけられる1つの章や節
- 例示など詳しくは第六回以降にやります

index.html

```
<article>
  <header>
    <h1>見出し</h1>
  </header>
  <section class="showHide">
    <h2>見出し1</h2>
    <p>内容1</p>
  </section>
  <section class="showHide">
    <h2>見出し2</h2>
    <p>内容2</p>
  </section>
  <section class="showHide">
    <h2>見出し3</h2>
    <p>内容3</p>
  </section>
</article>
```


セレクトタの基礎(再掲)

- idやclass名で更なる条件を設定して絞り込む

index.html

```
<div class="select">  
  <div>elem1</div>  
  <div class="selected">elem2</div>  
  <div>elem3</div>  
  <div>elem4</div>  
</div>
```

style.css

```
.select > div {  
  width: 40px; height: 20px;  
  border: 1px solid black;  
  margin: 10px;  
}  
.select > div.selected {  
  background-color: rgb(200, 220, 255);  
}
```

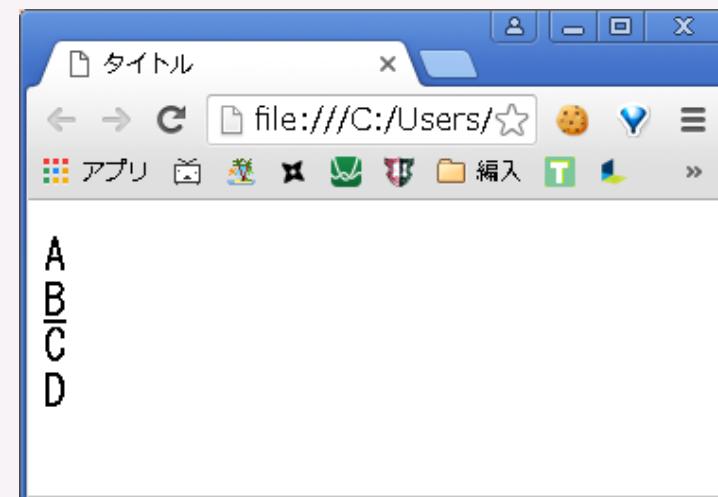


演習

- 1つだけ選択できるものを作ってみよう
 - 一般化はしなくても良いです
- できた方は
 - デザインしてみよう
 - 一般化してみよう
 - タブコントロールを作ってみよう

index.html

```
<ul class="select">
  <li class="selected">A</li>
  <li>B</li>
  <li>C</li>
  <li>D</li>
</ul>
```



style.css

```
.select {
  list-style: none;
  padding: 0;
}
.select > li:hover,
.select > li.selected {
  text-decoration: underline;
}
```

演習

main.js

```
let select = (e) => { // e.currentTarget を選択
  let clearTargets = e.currentTarget.parentNode.children;
  for (let i = 0; i < clearTargets.length; ++i) {
    clearTargets[i].classList.remove("selected");
  }
  e.currentTarget.classList.add("selected");
}

let initSelect = () => { // select初期化
  let elems = document.getElementsByClassName("select");
  for (let i = 0; i < elems.length; ++i) {
    let chs = elems[i].children;
    for (let j = 0; j < chs.length; ++j) {
      chs[j].addEventListener("click", select);
    }
  }
}

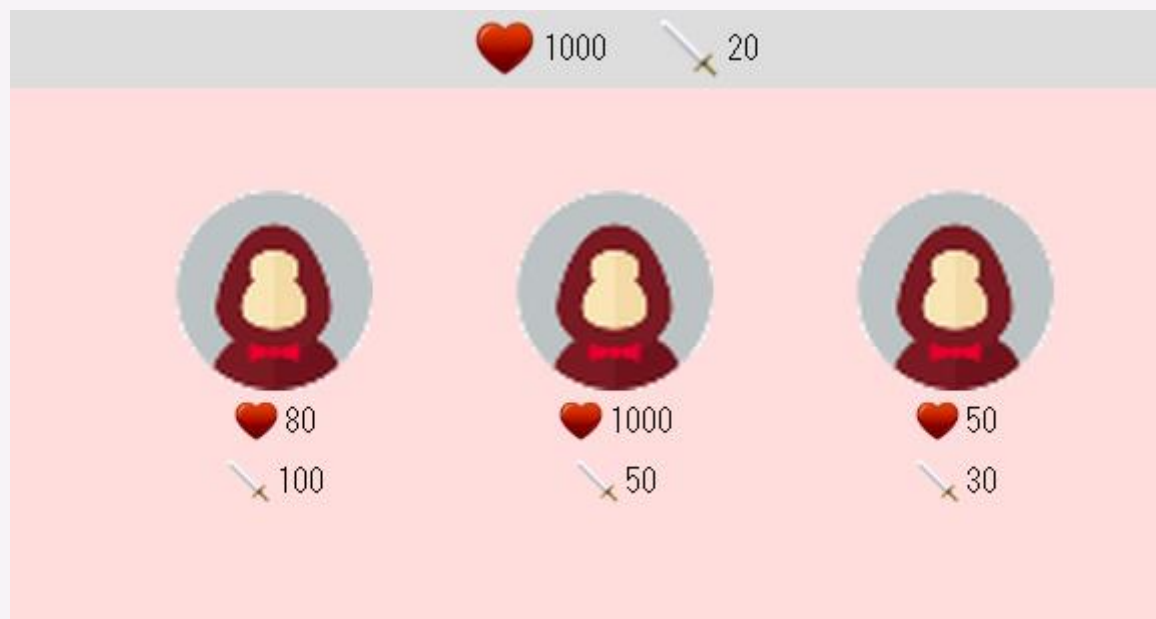
initSelect();
```



4. ゴリラ倒す

ゲーム仕様

- 面倒なのでプレイしてくれ
 - <http://drafear.ie-t.net/gorilla/>
- 攻撃すると全員攻撃してくる
- 倒した敵の攻撃力分攻撃力上昇



素材

- ハートと剣は二次配布おkらしいので
テンプレをダウンロードしたときについてきてるはず
- なかったらこちらからー
 - <https://github.com/kmc-jp/js2016/>
- ゴリラに替わるものは探してくれ

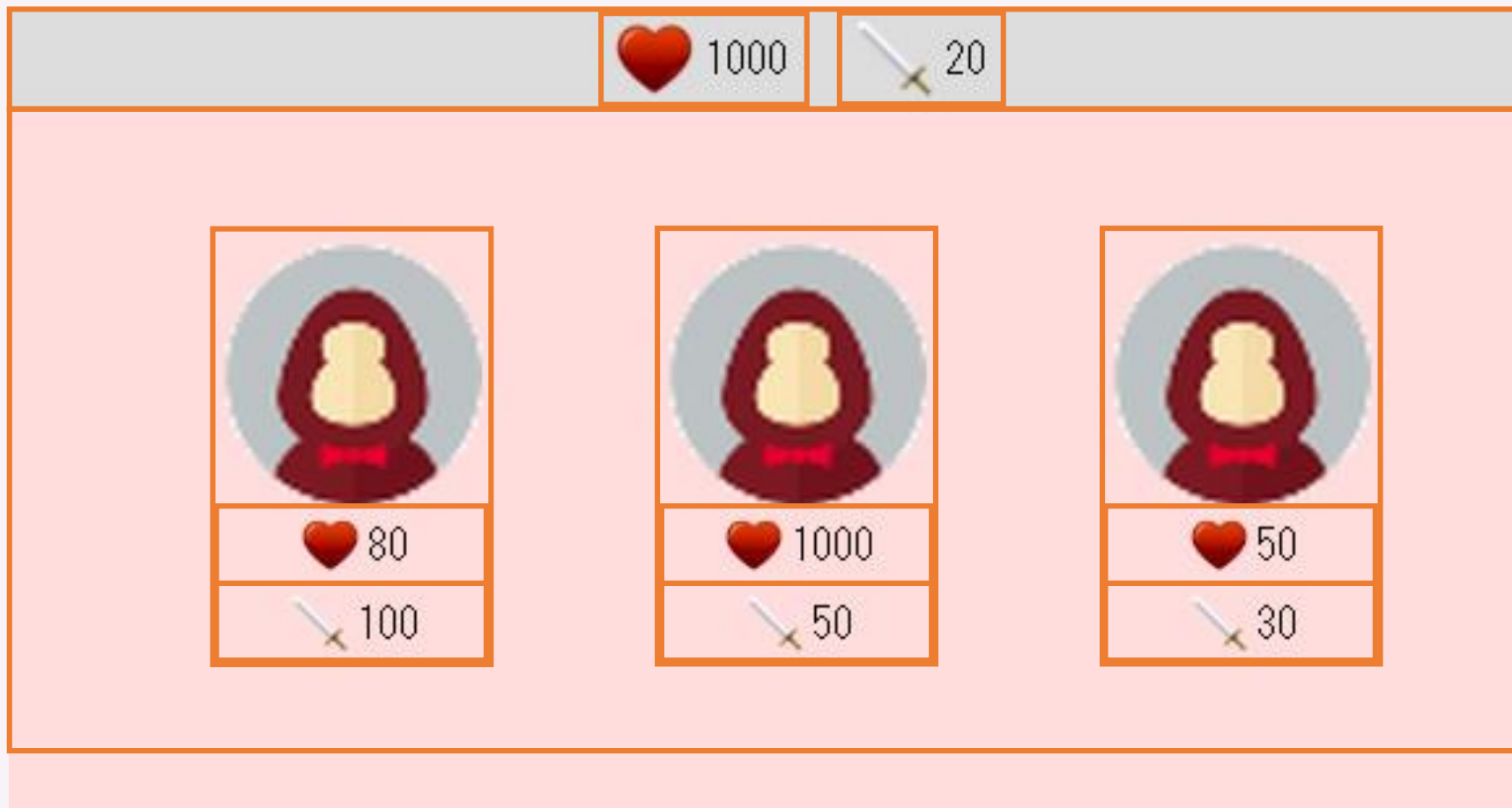


ディレクトリ構造



HTML構造

div



プレイヤー部分

div



プレイヤー部分



style.css

```
.player {
  display: flex;
  width: 100vw;
  height: 40px;
  justify-content: center;
  background-color: #ddd;
}

.player > div {
  display: flex;
  align-items: center;
  margin: 0 10px;
}

.player > div > img {
  box-sizing: border-box;
  width: 40px;
  height: 40px;
  padding: 5px;
}
```

プレイヤー部分

index.html

```
<div class="player">
  <div>
    
    <span id="playerHp">0</span>
  </div>
  <div>
    
    <span id="playerAtk">0</span>
  </div>
</div>
```

100vw = ブラウザ横幅100%
100vh = ブラウザ縦幅100%

margin: [上下] [左右];

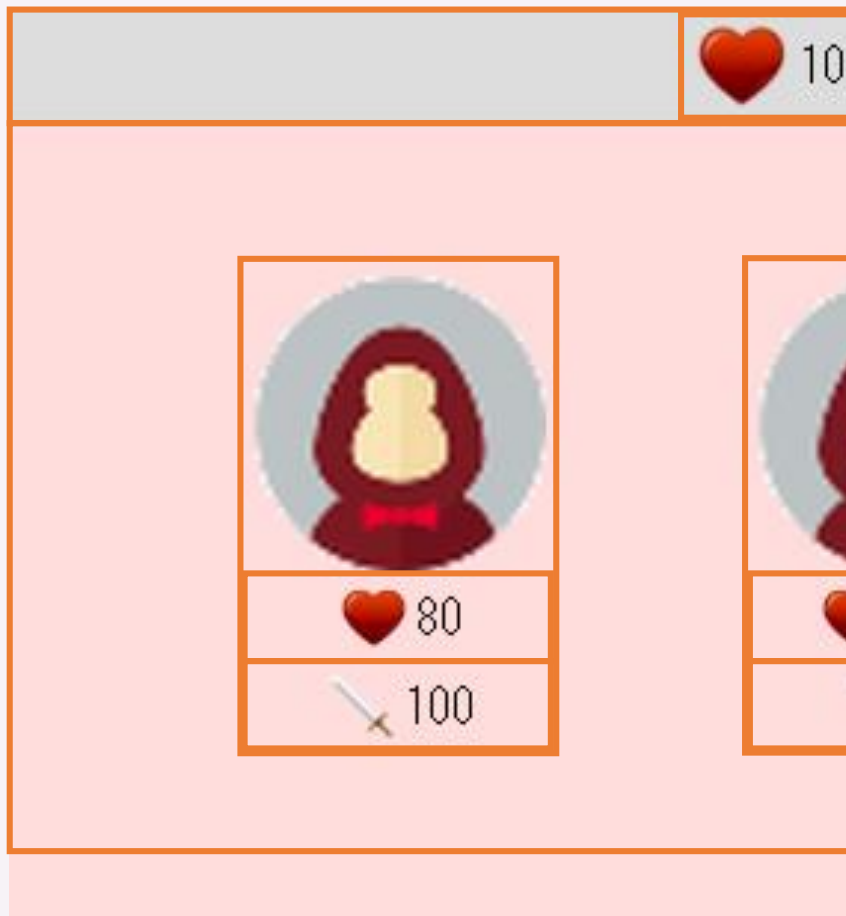
style.css

```
.player {
  display: flex;
  width: 100vw;
  height: 40px;
  justify-content: center;
  background-color: #ddd;
}

.player > div {
  display: flex;
  align-items: center;
  margin: 0 10px;
}

.player > div > img {
  box-sizing: border-box;
  width: 40px;
  height: 40px;
  padding: 5px;
}
```

敵部分



index.html

```
<div id="enemy" class="enemy">
  <div>
    
  </div>
  <div>
    
    <span id="enemyHp0">0</span>
  </div>
  <div>
    
    <span id="enemyAtk0">0</span>
  </div>
</div>
...
</div>
```

敵部分

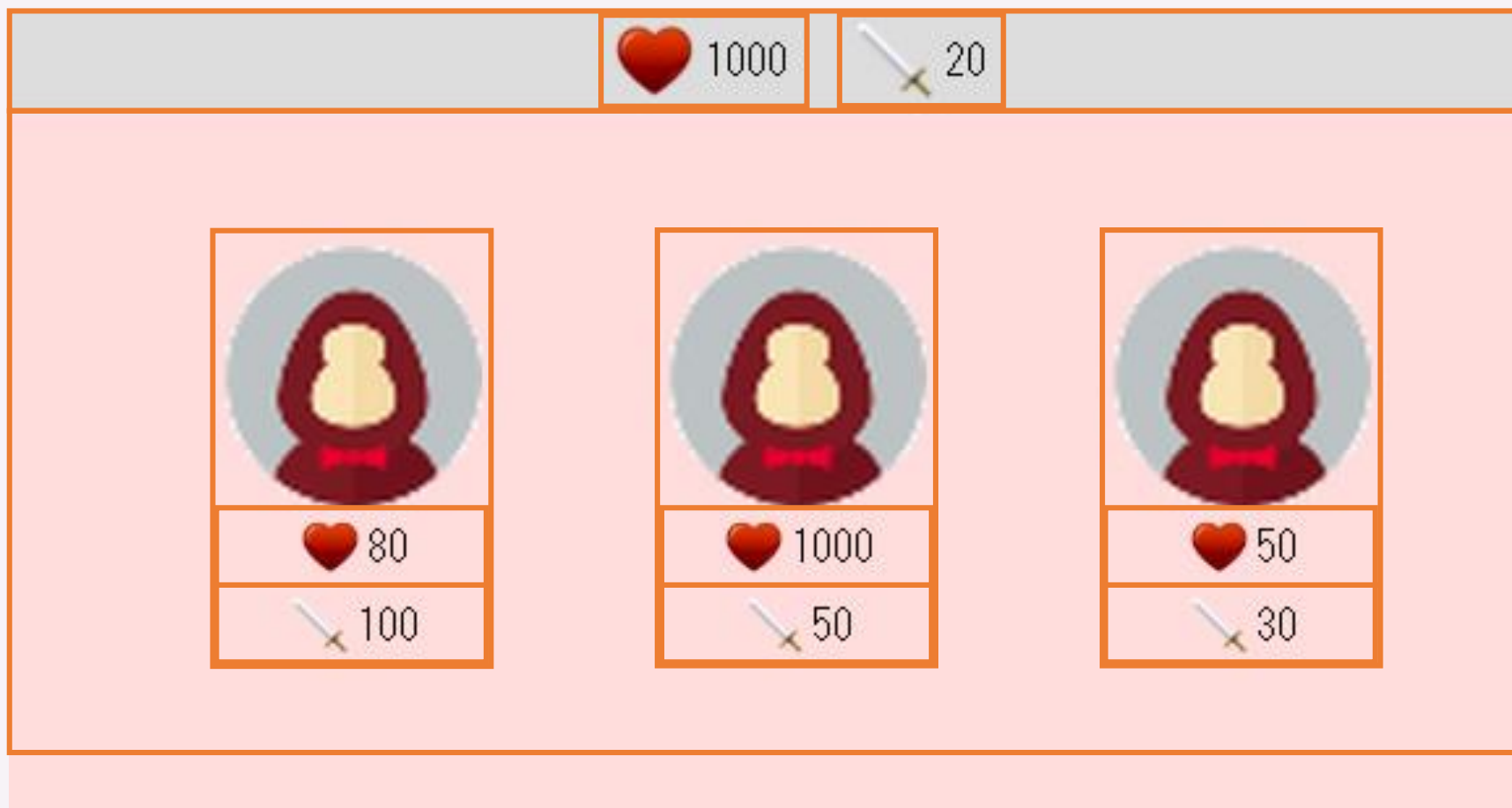
index.html

```
<div id="enemy" class="enemy">
  <div>
    
    <div>
      
      <span id="enemyHp0">0</span>
    </div>
    <div>
      
      <span id="enemyAtk0">0</span>
    </div>
  </div>
  <div>
    
    <div>
      
      <span id="enemyHp1">0</span>
    </div>
  </div>
```

```
    <div>
      
      <span id="enemyAtk1">0</span>
    </div>
  </div>
  <div>
    
    <div>
      
      <span id="enemyHp2">0</span>
    </div>
    <div>
      
      <span id="enemyAtk2">0</span>
    </div>
  </div>
</div>
```

HTML構造

div



敵部分

style.css

```
.enemy {  
  display: flex;  
  justify-content: center;  
  width: 100vw;  
}  
.enemy > div {  
  display: flex;  
  flex-direction: column;  
  align-items: center;  
  box-sizing: border-box;  
  width: 110px;  
  margin: 50px 30px;  
  padding: 1px;  
}
```

```
.enemy > div: hover {  
  border: 1px solid red;  
  padding: 0;  
  cursor: pointer;  
}  
.enemy > div > img {  
  width: 100px;  
  height: 100px;  
}  
.enemy > div > div {  
  display: flex;  
  align-items: center;  
  justify-content: center;  
}  
.enemy > div > div > img {  
  box-sizing: border-box;  
  width: 30px;  
  height: 30px;  
  padding: 4px;  
}
```

死んだ敵を非表示にする部分

- !important
 - 同じプロパティの指定が複数箇所で行われていた場合優先される

style.css

```
.hide {  
  display: none !important;  
}
```


JavaScript部分

main.js

```
// プレイヤーと敵の能力値(初期値)  
let player = { hp: 1000, atk: 20 };  
let enemy = [  
  { hp: 80, atk: 100 },  
  { hp: 1000, atk: 50 },  
  { hp: 50, atk: 30 },  
];
```

JavaScript部分

main.js

```
// プレイヤーと敵の能力値(初期値)  
let player = { hp: 1000, atk: 20 };  
let enemy = [  
  { hp: 80, atk: 100 },  
  { hp: 1000, atk: 50 },  
  { hp: 50, atk: 30 },  
];
```

JavaScript部分

main.js

```
// creature(player/enemy)が活着ているか
let isAlive = (c) => { ... }
// プレイヤーが敵に攻撃する
let attack = (e) => {
  ... // プレイヤーが死んでいたら何もしない
  ... // プレイヤーの攻撃
  ... // 敵の攻撃
  update(); // 画面に反映
}
// 画面に反映する
let update = () => { ... }
// 最初にする処理
let init = () => {
  ... // 敵をクリックしたときのイベント設定. attackをそのまま呼ぶ.
  update();
}
init();
```

JavaScript部分

main.js

```
// 最初にする処理
let init = () => {
  let chs = document.getElementById("enemy").children;
  for (let i = 0; i < chs.length; ++i) {
    enemy[i].elem = chs[i];
    chs[i].addEventListener("click", attack);
    chs[i].dataset.index = i;
  }
  update();
}
```

JavaScript部分

main.js

```
// 画面に反映する
let update = () => {
  // player
  document.getElementById("playerHp").innerText = player.hp;
  document.getElementById("playerAtk").innerText = player.atk;
  // enemy
  for (let i = 0; i < enemy.length; ++i) {
    if ( !isAlive(enemy[i]) ) {
      enemy[i].elem.classList.add("hide")
    }
    else {
      document.getElementById(`enemyHp${i}`).innerText = enemy[i].hp;
      document.getElementById(`enemyAtk${i}`).innerText = enemy[i].atk;
    }
  }
}
```

JavaScript部分

main.js

```
// プレイヤーが敵に攻撃する
let attack = (e) => {
  // プレイヤーが死んでいたら何もしない
  if ( !isAlive(player) ) return;
  // プレイヤーの攻撃
  let idx = e.currentTarget.dataset.index;
  enemy[idx].hp -= player.atk;
  if ( !isAlive(enemy[idx]) ) { // 倒した場合
    player.atk += enemy[idx].atk;
  }
  // 敵の攻撃
  ...
  // 画面に反映
  update();
}
```

JavaScript部分

main.js

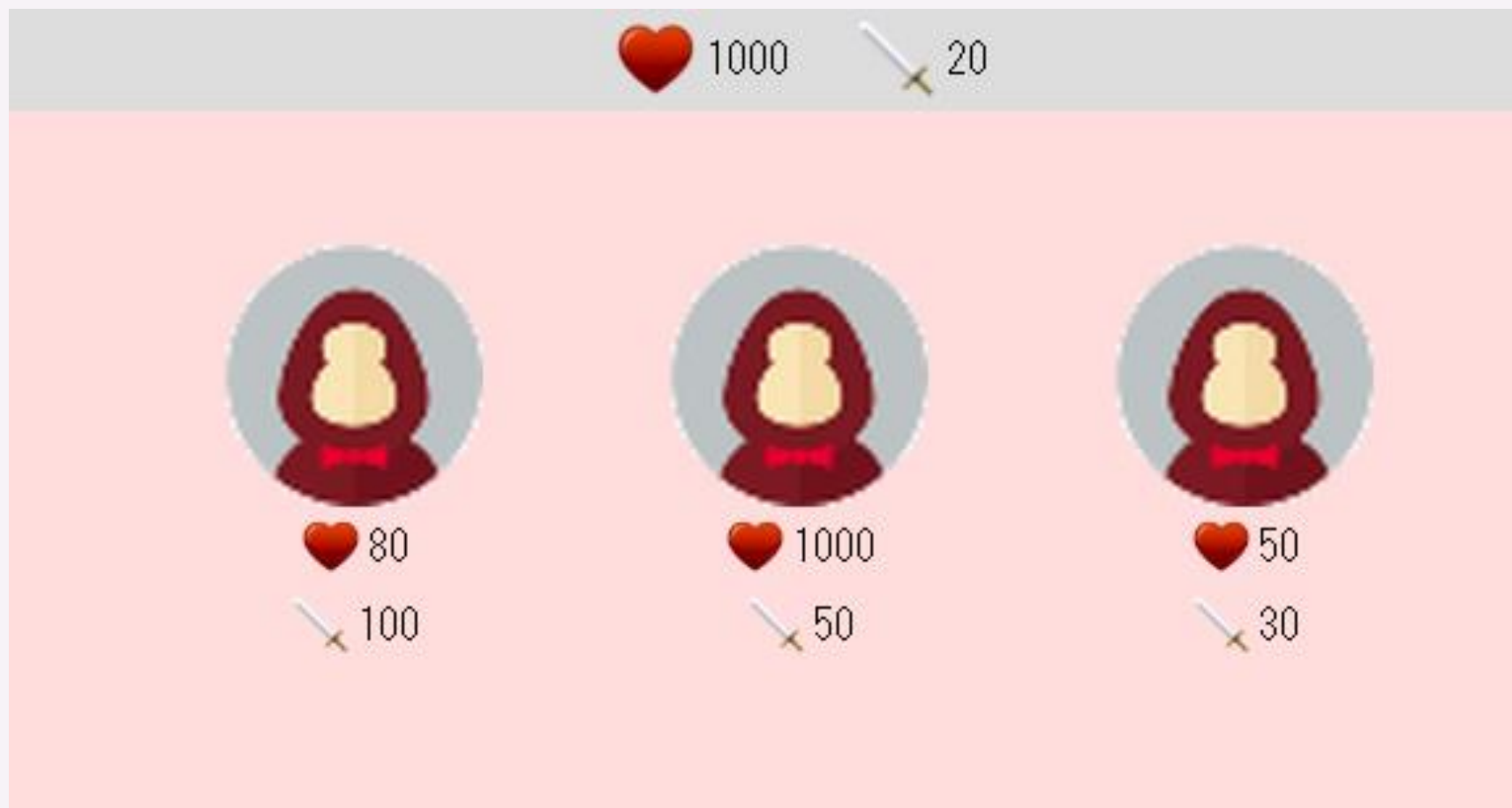
```
// プレイヤーが敵に攻撃する
let attack = (e) => {
  ...
  // 敵の攻撃
  for (let i = 0; i < enemy.length; ++i) {
    // 生きている敵だけ攻撃してくる
    if ( isAlive(enemy[i]) ) {
      player.hp -= enemy[i].atk;
    }
  }
  // プレイヤーのhpの最小値は0
  if (player.hp < 0) {
    player.hp = 0;
  }
  // 画面に反映
  update();
}
```

JavaScript部分

main.js

```
// creatureが活着ているか  
let isAlive = (c) => {  
  return c.hp > 0;  
}
```


完成！お疲れ様です！！

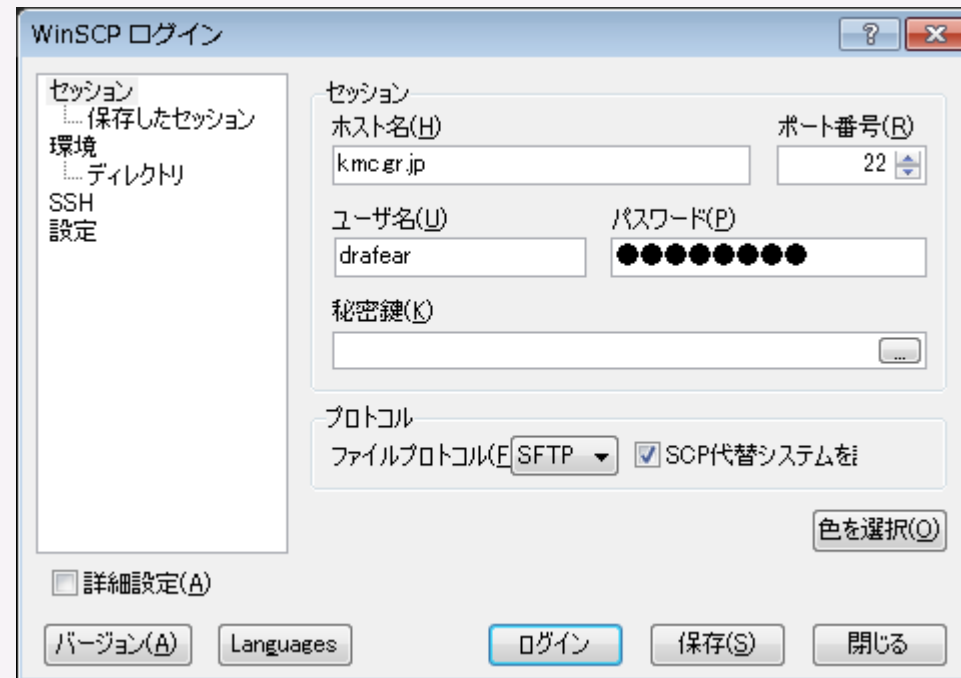


作品をアップロードしてみよう

- WinSCPを使って内部ページにアップロードしてみる
 - <http://www.forest.impress.co.jp/library/software/winscp/>

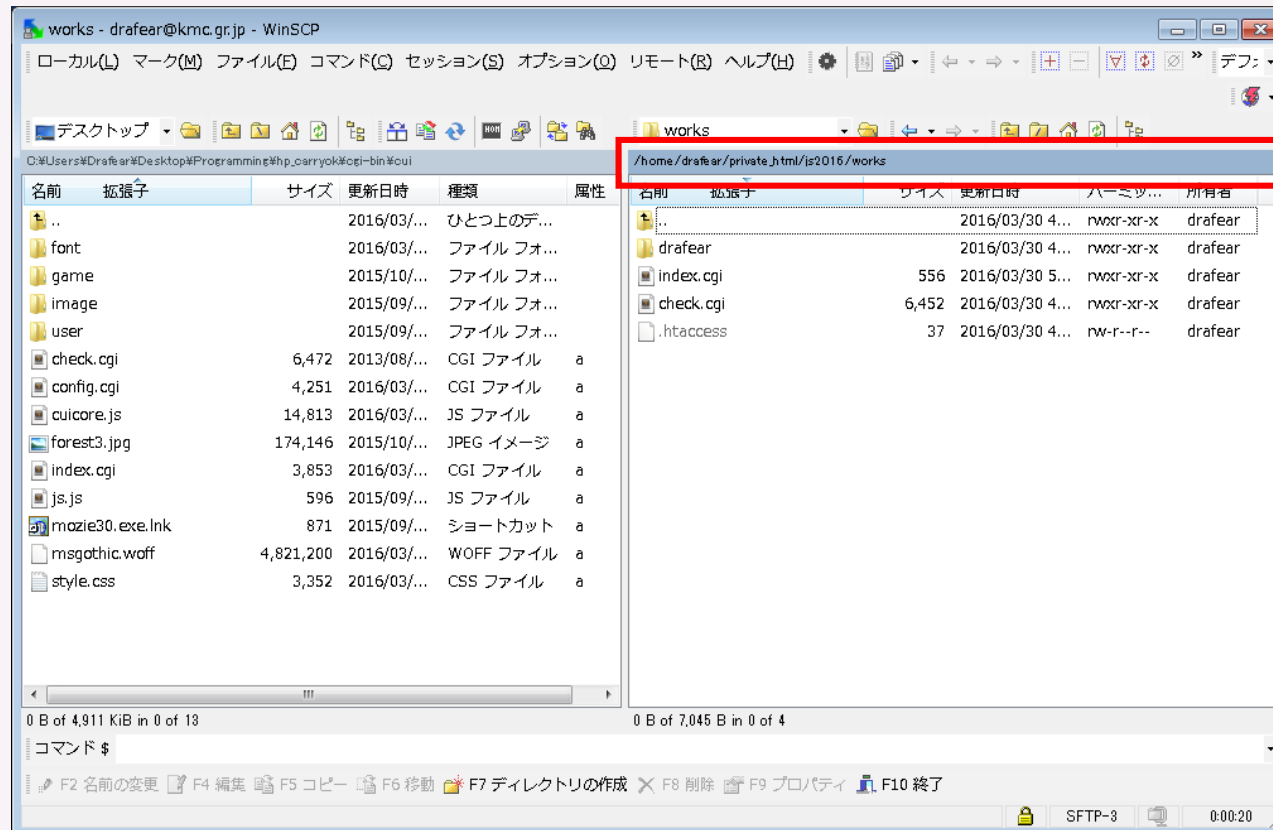
作品をアップロードしてみよう

- 次のように入れてログインして下さい
 - ユーザ名, パスワード は KMC の wiki にログインするときに用いるものを入れて下さい



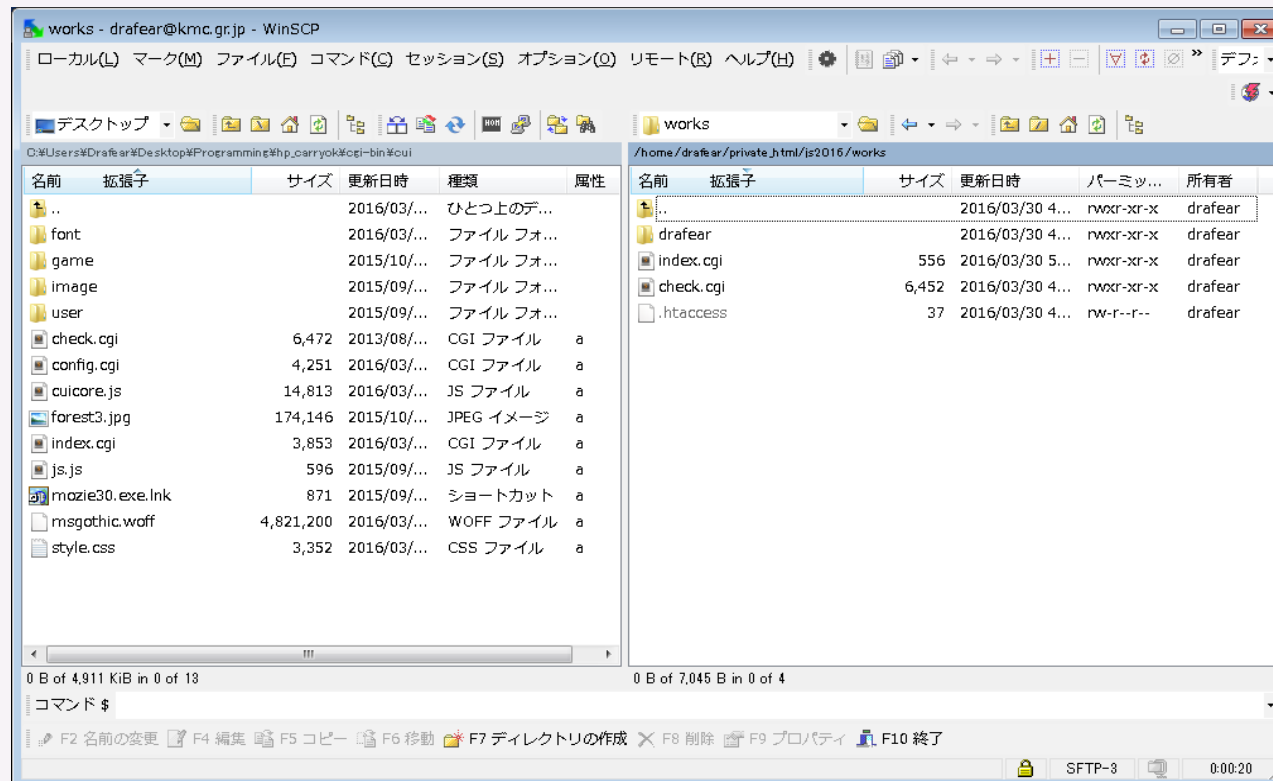
作品をアップロードしてみよう

- ここをダブルクリックして次のパスを入力して下さい
 - /home/drafeear/private_html/js2016/works



作品をアップロードしてみよう

- そこに自分のidの名前のディレクトリを作って、
その中に名前 rps でディレクトリを作って、
その中に index.html と main.js と style.css を入れて下さい



今後の予定

- 5/22(日) 13:00 ~ 16:00
 - CSSでレイアウトを学ぶ
- 5/29(日) 13:00 ~ 16:00
 - 復習回
 - 何かを作ろう
- 6/5(日) 13:00 ~ 16:00