

第7回

JavaScriptから始める プログラミング2016

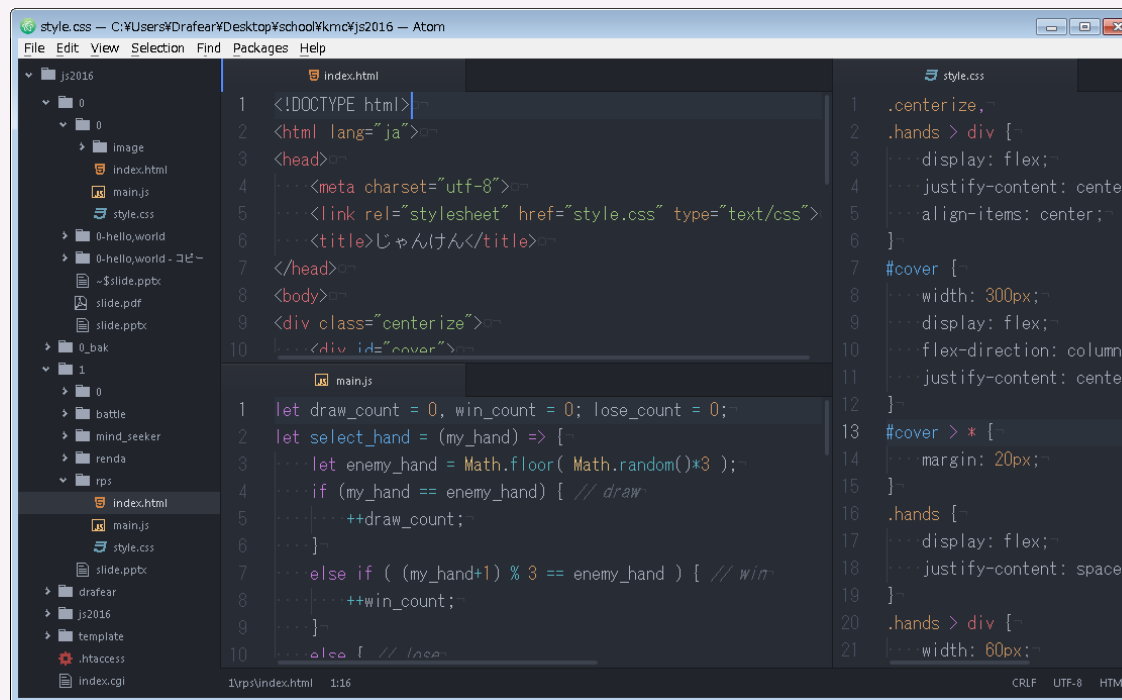
京都大学工学部情報学科
計算機科学コース3回
KMC2回 drafear



@drafear

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/>



The screenshot shows the Atom text editor interface. On the left is a file explorer showing a project structure with folders like '0', '1', 'battle', 'mind_seeker', 'renda', 'rps', and files like 'index.html', 'main.js', 'style.css', and 'slide.pptx'. The main editor area is split into two panes. The left pane shows 'index.html' with the following code:

```
1 <!DOCTYPE html>
2 <html lang="ja">
3 <head>
4   <meta charset="utf-8">
5   <link rel="stylesheet" href="style.css" type="text/css">
6   <title>じゃんけん</title>
7 </head>
8 <body>
9   <div class="centerize">
10    <div id="cover">
```

The right pane shows 'style.css' with the following code:

```
1 .centerize,
2 .hands > div {
3   display: flex;
4   justify-content: center;
5   align-items: center;
6 }
7 #cover {
8   width: 300px;
9   display: flex;
10  flex-direction: column;
11  justify-content: center;
12 }
13 #cover > * {
14   margin: 20px;
15 }
16 .hands {
17   display: flex;
18   justify-content: space-around;
19 }
20 .hands > div {
21   width: 60px;
```

重要度

- 今回も, 重要度の項目を設けます
 - 重要度 ★★★★★ : 必須. 自然と直面するかもしれない.
 - 重要度 ★★★★★☆ : 知らないとその機能の実現が厳しい.
 - 重要度 ★★★☆☆ : 知ってたら便利. 綺麗なコードを書くのに重要かも.
 - 重要度 ★★☆☆☆ : たまに使うかもしれない.
 - 重要度 ★☆☆☆☆ : ほとんど使わない. おまけ. 興味があれば.
 - 重要度 ☆☆☆☆☆ : マニアック. ライブラリ作るなら必要かも.

本日の内容

- 今日も紹介って感じなので
よさげなものを拾って帰って下さい

本日の内容

- JavaScript
 - for ... in, key in obj, delete
 - クロージャ
 - 値渡しと参照渡し
 - 条件演算子 (三項演算子)
 - メソッドチェーン
 - 文字列処理 (今回の主役)
 - Web API
- その他
 - 正規表現



1. 復習

復習

- 前回, 前々回のスライドを見てサッと復習する



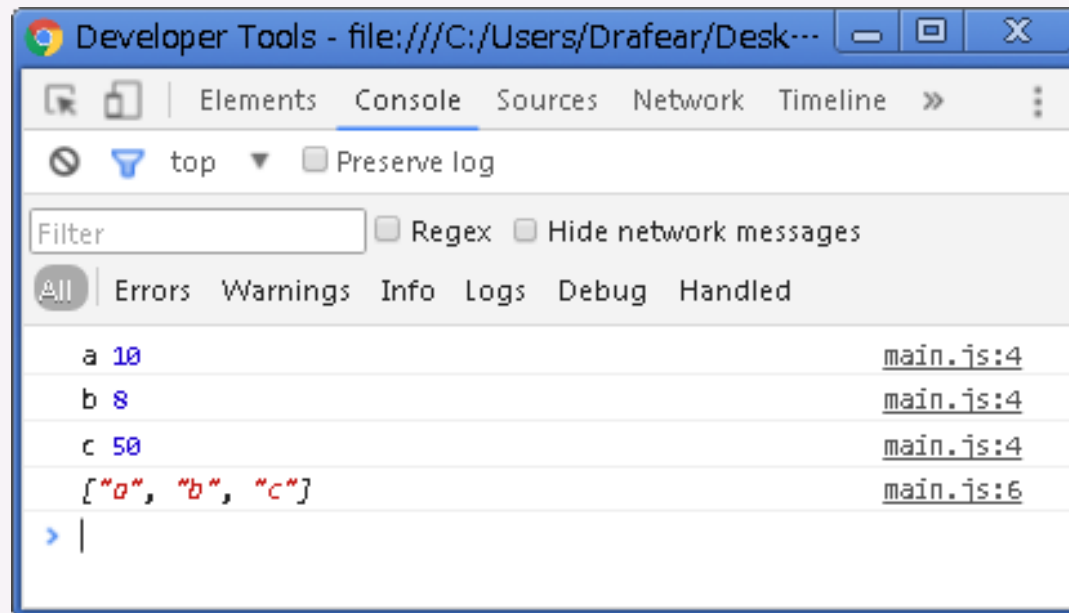
2. JavaScript いろいろ

for ... in

- `for (const 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`の全プロパティを配列で得る

main.js

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

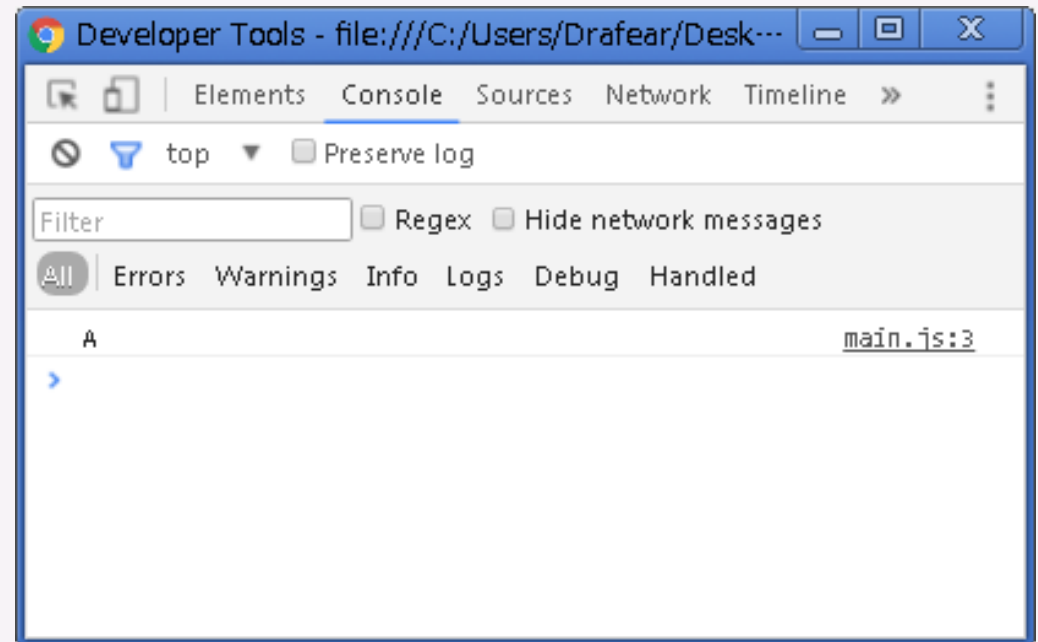


key in obj

- *key in obj*
 - オブジェクト *obj* にプロパティ *key* が設定されているか

main.js

```
let obj = { a: 5, b: 3 };  
if ( "a" in obj ) {  
    console.log("A");  
}  
if ( "c" in obj ) {  
    console.log("C");  
}
```

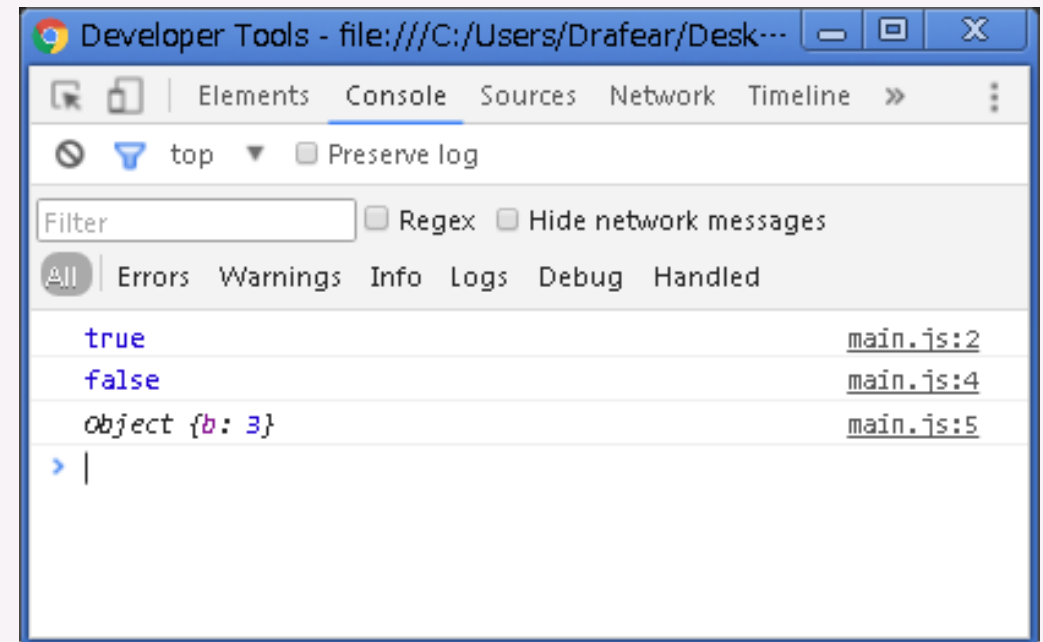


delete

- delete *obj.key*
- delete *obj["key"]*
 - オブジェクト*obj*からプロパティ(キー)*key*を削除する

main.js

```
let obj = { a: 5, b: 3 };  
console.log( "a" in obj );  
delete obj.a;  
console.log( "a" in obj );  
console.log(obj);
```



関数の代入

重要度 ★★★★★☆

- 数値や一般的なオブジェクトと同じように, 関数も再代入OK

関数代入例

```
'use strict'  
let func = () => {  
  console.log(10);  
};  
func = () => {  
  console.log(5);  
};  
func(); // 5
```

関数の代入

重要度 ★★★★★☆

- なので, もちろん以下のようなことができる

関数代入例2

```
'use strict'
let func;
const init = () => {
  func = () => {
    console.log("hello");
  };
};
init();
func(); // "hello"
```

関数外のローカル変数参照

- 関数は、その関数が作られたところでアクセスできる変数にはその関数内からもアクセスできる

変数参照例

```
'use strict'
let func;
const init = () => {
  let a = 10;
  func = () => {
    console.log(a);
  };
};
init();
func(); // 10
```

関数外のローカル変数参照

- 以下の例では, 何回呼びだされたかを表示する関数funcを作っている

変数参照例2

```
'use strict'
let func;
const init = () => {
  let cnt = 0;
  func = () => {
    ++cnt;
    console.log(cnt);
  };
};
init();
func(); // 1
func(); // 2
func(); // 3
```

関数を返す関数

重要度 ★★★★★☆

- もちろん, 関数を返す関数もOK

関数返却関数

```
'use strict'
const makeCounter = () => {
  let cnt = 0;
  return () => {
    ++cnt;
    console.log(cnt);
  };
};
const counter = makeCounter();
counter(); // 1
counter(); // 2
counter(); // 3
```


関数を返す関数

- counter1 と counter2 の cnt は別のものであることに注目

関数返却関数

```
'use strict'
const makeCounter = () => {
  let cnt = 0;
  return () => {
    ++cnt;
    console.log(cnt);
  };
};
const counter1 = makeCounter();
const counter2 = makeCounter();
counter1(); // 1
counter1(); // 2
counter1(); // 3
counter2(); // 1
counter2(); // 2
```

クロージャ

- 「関数を返す関数」とすることでメンバ変数を完全に隠蔽したクラスのようなものができる
- このように、関数外のローカル変数を参照する関数をクロージャという

条件演算子(三項演算子)

- 条件 ? 式1 : 式2
- if文のようなふるまいをする
- 条件 が true なら 式1 の評価値, false なら 式2 の評価値となる
- 式2 に更に「条件2 ? 式3 : 式4」と書くことで else if のようなことができる
- 条件演算子 は 3つの項(条件, 式1, 式2) を取る唯一の演算子なので 三項演算子とも呼ばれ, 三項演算子 = 条件演算子 である

条件演算子

```
'use strict'  
let x = 10;  
console.log(x < 5 ? 100 : 200); // 200  
console.log(x < 20 ? 100 : 200); // 100  
console.log(x < 5 ? 100 : x > 8 ? 200 : 300); // 200
```

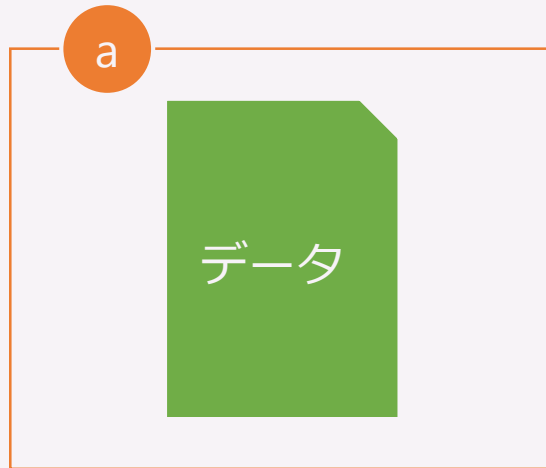
値渡しと参照渡し

- データ型によって $b = a$ とするときに処理される方法が2通りある

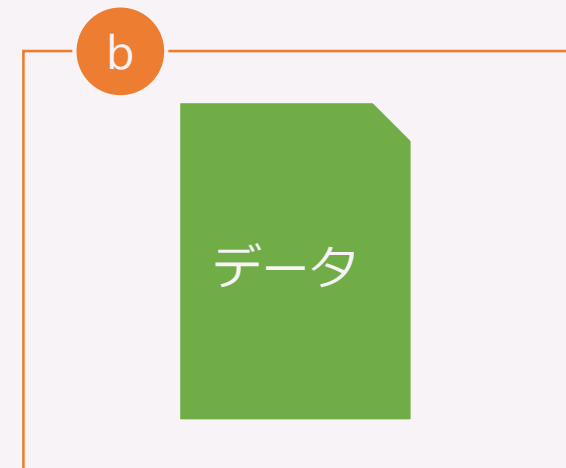
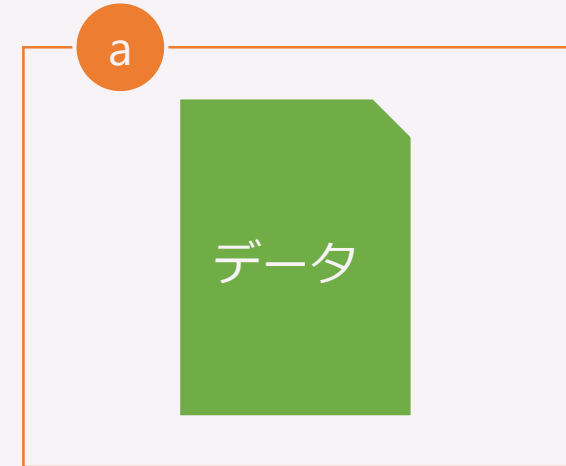
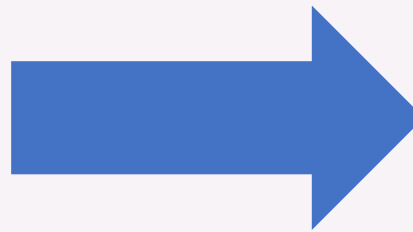
値渡しと参照渡し

重要度 ★★★★★

- 値渡し
 - データ全体が複製される

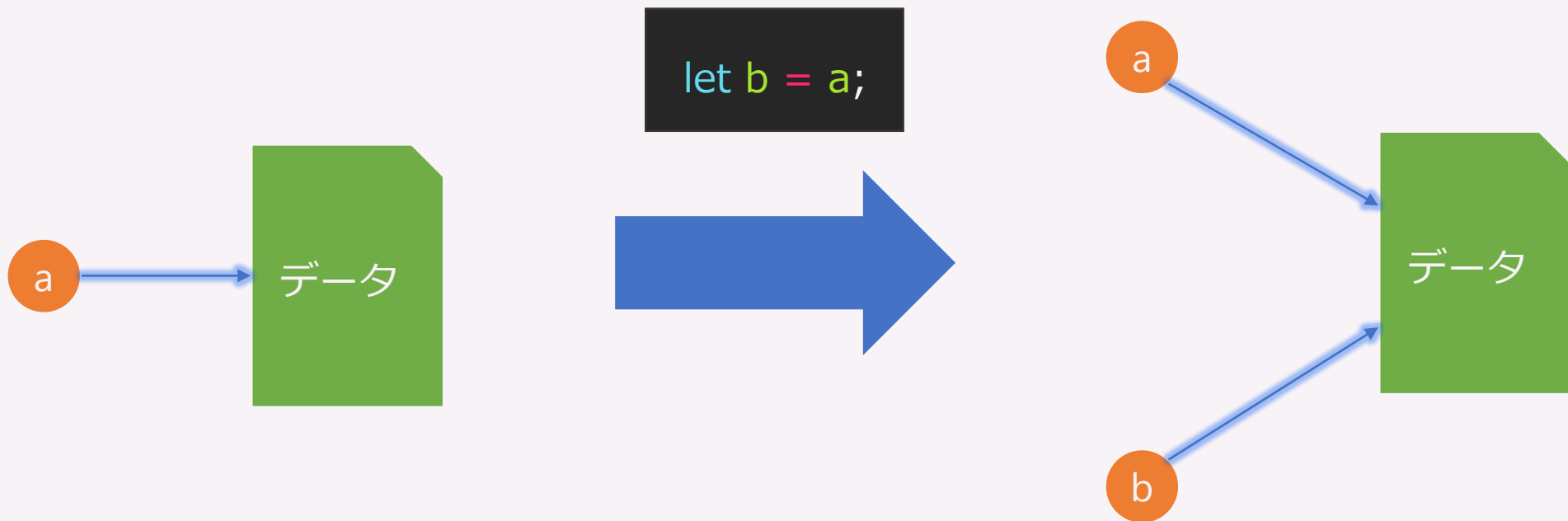


```
let b = a;
```



値渡しと参照渡し

- 参照渡し
 - データの場所(住所)をコピーする



値渡しと参照渡し

- 値渡し
 - $b = a$ とした後に b のデータを書き換えても a に影響しない
- 参照渡し
 - $b = a$ とした後に b のデータを書き換えると a にも影響する

値渡しと参照渡し

- 値渡し

- $b = a$ とした後に b のデータを書き換えても a に影響しない
- 以下の型の場合は値渡しされる
数値型(Number), 論理値型(Boolean), null型, undefined型

- 参照渡し

- $b = a$ とした後に b のデータを書き換えると a にも影響する
- 以下の型の場合は参照渡しされる
シンボル型(Symbol), 文字列型(String), オブジェクト(Object)

※厳密には, 文字列/文字列オブジェクト, シンボル/シンボルオブジェクト で異なる

値渡しと参照渡し

重要度 ★★★★★

- 例

main.js

```
'use strict'  
const Global = {};  
Global.func = (obj) => {  
  obj.a *= 2;  
};  
const obj1 = { a: 10 };  
Global.func(obj1);  
console.log(obj1); // { a: 20 }
```

メソッドチェーン

重要度 ★★★★★

- return this や return new Hoge() をすると続けてメソッドを呼べる！

main.js

```
class Creature {  
  constructor hp, atk {  
    [this.hp, this.atk] = [hp, atk];  
  }  
  clone() { // 自身を複製する  
    return new Creature(this.hp, this.atk);  
  }  
  damage(val = 1) { // ダメージを受ける  
    // 0以下になったら0にする  
    this.hp = Math.max(0, this.hp-val);  
    return this;  
  }  
  attack(that) {  
    that.damage(this.atk);  
    return this;  
  }  
};
```

main.js

```
const c1 = new Creature(15, 3);  
const c2 = new Creature(7, 1);  
const c3 = c2.clone();  
c1.attack(c2).attack(c3).attack(c3);  
console.log(c1.hp); // 15  
console.log(c2.hp); // 4  
console.log(c3.hp); // 1  
c1.clone().attack(c3);  
console.log(c3.hp); // 0
```

メソッドチェーン

重要度 ★★★★★

- 復習を兼ねた例

main.js

```
class Creature {
  constructor hp, atk {
    [this.hp, this.atk] = [hp, atk];
  }
  clone() { // 自身を複製する
    return new Creature(this.hp, this.atk);
  }
  damage(val = 1) { // ダメージを受ける
    // 0以下になったら0にする
    this.hp = Math.max(0, this.hp - val);
    return this;
  }
  attack(that) {
    that.damage(this.atk);
    return this;
  }
};
```

main.js

```
// どちらかが死ぬまで戦う
const battle = (c1, c2) => {
  while (c1.hp > 0 && c2.hp > 0) {
    // ターンプレイヤーが攻撃
    c1.attack(c2);
    // ターン入れ替え
    [c1, c2] = [c2, c1];
  }
};
const soldier = new Creature(700, 4);
const mage = new Creature(100, 30);
// 参照渡しなのでちゃんと
// soldier, mageに反映される
battle(soldier, mage);
// 結果！！
console.log(soldier.hp); // 0 敗北...
console.log(mage.hp); // 4 勝利！！
```

演習 (復習問題)

- ベクトルを表すクラスを作ってみよう
 - `class Vector { constructor(x, y) { ... } ... }`
 - `v1.add(v2)` : `v1+v2` を計算したVectorを返す. `v1, v2`には変更なし.
 - `v1.sub(v2)` : `v1-v2` を計算したVectorを返す. `v1, v2`には変更なし.
 - `v1.dot(v2)` : `v1`と`v2`の内積を返す.

main.js

```
const v1 = new Vector(1, 2);
const v2 = new Vector(3, 4);
const v3 = new Vector(5, 6);
const v12 = v2.sub(v1); // v2自身には変更が及ばない
const v23 = v3.sub(v2); // v3自身には変更が及ばない
console.log( v12.dot(v23) ); // 8
console.log( v1.add(v2).add(v3) ); // Vector { x: 9, y: 12 }
```

演習

- ベクトルを表すクラスを作ってみよう

main.js

```
class Vector {  
  constructor(x, y) {  
    this.x = x, this.y = y;  
  }  
  add(that) {  
    return new Vector(this.x + that.x, this.y + that.y);  
  }  
  sub(that) {  
    return new Vector(this.x - that.x, this.y - that.y);  
  }  
  dot(that) {  
    return this.x * that.x + this.y * that.y;  
  }  
}
```

演習

- 分割代入 ($\geq \nabla \leq$) / (好みの問題なのでどちらでも)

main.js

```
class Vector {  
  constructor(x, y) { [this.x, this.y] = [x, y]; }  
  add(that) {  
    const [{x:x1,y:y1}, {x:x2,y:y2}] = [this, that];  
    return new Vector(x1+x2, y1+y2);  
  }  
  sub(that) {  
    const [{x:x1,y:y1}, {x:x2,y:y2}] = [this, that];  
    return new Vector(x1-x2, y1-y2);  
  }  
  dot(that) {  
    const [{x:x1,y:y1}, {x:x2,y:y2}] = [this, that];  
    return x1*x2 + y1*y2;  
  }  
}
```



3. 文字列処理

エスケープシーケンス

- "" の文字列中に " を入りたい → ¥"
- " の文字列中に ' を入りたい → ¥'
- `` の文字列中に ` や \$ や { や } を入りたい → ¥` ¥\$ ¥{ ¥}

エスケープシーケンス

重要度 ★★★★★

- じゃあ文字列中に ¥ はどうやって入れるの？ → ¥¥

エスケープシーケンス

- 他にも特殊な文字を ¥ほげ で表します

¥n	改行
¥r	行の先頭にカーソルを移動
¥t	タブ
¥b	バックスペース

文字列中の文字へのアクセス

重要度 ★★★★★☆

- str[pos] または str.charAt(pos)
 - 読み取り専用 (readonly)

main.js

```
'use strict'  
let str = "kmc";  
console.log(str[2]); // c  
str[2] = "a"; // error
```

文字列切り出し

重要度 ★★★★★☆

- `str.substr(start [,size])`

main.js

```
'use strict'  
let str = "javascript";  
console.log( str.substr(4, 2) ); // 0から数えて4文字目から2文字: sc  
console.log( str.substr(7) ); // 0から数えて7文字目以降: ipt  
console.log( str.substr(-1) ); // 後ろから1文字: t
```

文字列比較

重要度 ★★★★★☆

- === !== > >= < <=
- 辞書順で比較する

main.js

```
'use strict'  
console.log("kmc" === "kmc"); // true  
console.log("kmc" !== "kmc"); // false  
console.log("kmc" > "kpc"); // false  
console.log("kmc" < "kpc"); // true  
console.log("kmc" <= "kmc"); // true
```

文字列の長さ

重要度 ★★★★★

- str.length

main.js

```
'use strict'  
console.log("kmc".length); // 3
```

文字列位置

重要度 ★★★★★

- `str1.indexOf(str2)`
 - `str1` 中に `str2` が含まれていれば最初に見つかった位置, 含まれていなければ `-1`

main.js

```
'use strict'  
let str = "kmkkmkmc";  
console.log( str.indexOf("mkm") ); // 1  
console.log( str.indexOf("kmc") ); // 4  
console.log( str.indexOf("aa") ); // -1
```

文字列分割

重要度 ★★★★★

- `str1.split(str2)`
 - `str1` を区切り文字 `str2` で分割し, 配列として得る

main.js

```
'use strict'  
let str = "1,2,3,4,100";  
console.log( str.split(",") ); // ["1", "2", "3", "4", "100"]
```


配列 → 文字列

- `ary.join(str)`
 - `ary`の各要素の間に`str`を挟んで連結し, 文字列にする

main.js

```
'use strict'  
console.log( ["i", "love", "you"].join(" ") ); // I love you
```

小文字, 大文字变换

重要度 ★★☆☆☆

- `str.toLowerCase()`, `str.toUpperCase()`

main.js

```
'use strict'  
let str = new String("kMc");  
console.log( str.toLowerCase() ); // kmc  
console.log( str.toUpperCase() ); // KMC
```

演習

1. 文字列 `str1` と文字列 `str2` が先頭から何文字一致しているかを返す関数 `getMatchLength` を作ってみよう
 - `const getMatchLength = (str1, str2) => { ... }`
2. 文字列 `src` 中にある文字列 `from` を全て文字列 `to` に置き換える関数 `replaceStr` を作ってみよう
 - 複数候補があるかもしれないが, 前から順に置き換える
 - `const replaceStr = (src, from, to) => { ... }`

演習

1. 文字列 str1 と文字列 str2 が先頭から何文字一致しているかを返す関数 getMatchLength を作ってみよう

main.js

```
const getMatchLength = (str1, str2) => {  
  const l = Math.min(str1.length, str2.length);  
  for (let i = 0; i < l; ++i) {  
    if (str1[i] !== str2[i]) {  
      return i;  
    }  
  }  
  return l;  
};
```

演習

1. 文字列 str1 と文字列 str2 が先頭から何文字一致しているかを返す関数 getMatchLength を作ってみよう

main.js (別解)

```
const getMatchLength = (str1, str2) => {  
  if (str1.length == 0 || str2.length == 0) return 0;  
  if (str1[0] !== str2[0]) return 0;  
  return getMatchLength(str1.substr(1), str2.substr(1))+1;  
};
```

演習

2. 文字列 `src` 中にある文字列 `from` を全て文字列 `to` に置き換える関数 `replaceStr` を作ってみよう (前から順に置き換える)
- 1. が解けた人用のパズル問題でした.
 - ちゃんと次でやる `replace` 関数があるので
今後はそっちを使いましょう

main.js

```
'use strict'  
const replace = (src, from, to) => {  
  return src.split(from).join(to);  
};
```

文字列置換

重要度 ★★★★★

- `str.replace(fromStr, toStr)`
 - `str` 中の文字列 `fromStr` があれば初めの1つだけ `toStr` に置換した文字列を返す

main.js

```
console.log( "wwwwww".replace("ww", "X") ); // Xwww  
console.log( "drafear".replace("a", "") ); // drfear
```

文字列置換

重要度 ★★★★★

- 一致するものを全て置換したい
- `str.replace(regExp, toStr)`

main.js

```
console.log( "wwwwww".replace(new RegExp("ww", "g"), "X") ); // XXw  
console.log( "drafear".replace(new RegExp("a", "g"), "") ); // drfer
```


文字列置換

重要度 ★★★★★

- new RegExp("ww", "g") って何？
 - 正規表現クラスを正規表現"ww", オプション"g" でインスタンス化！
 - というわけで...

main.js

```
console.log( "wwwww".replace(new RegExp("ww", "g"), "X") ); // XXw  
console.log( "drafear".replace(new RegExp("a", "g"), "") ); // drfer
```



4. 正規表現

正規表現 #とは

- 文字列の集合を表す手法のひとつ
- 一般的なプログラミング言語では, 正規表現を文字列で表す
- つまり, 文字列 で 文字列の集合 を表現する
- 有限オートマトンの表現能力と等価

正規表現 #とは

- "hogehoge" は正規表現であり
文字列集合 {"hogehoge"} を表す

メタ文字

- "." は 改行文字¥n 以外の任意の1文字を表す
 - "h.ge" は {"hage", "hbge", ..., "hoge", ..., "hzge", "h1ge", ...} を表す
 - "..." は 任意の3文字を表す
- "reg1|reg2" はreg1またはreg2を表す
 - "bbb|a." は {"bbb", "aa", "ab", ..., "aZ", ...} を表す
- () でグループ化ができる
 - "windows(7|8|10)" は {"windows7", "windows8", "windows10"} を表す

メタ文字

- "X*" は X(1文字 or 1グループ) が0回以上繰り返されたものを表す
 - "ab*" → {aの後にbが1文字以上続く文字列} = {"a", "ab", "abb", ...}
 - "(aa|bb)*" → {"", "aa", "bb", "aaaa", "aabb", "bbbaa", "bbbb", "aaaaaa", "aaaabb", ...}
- "X+" は X(1文字 or 1グループ) が1回以上繰り返されたものを表す
 - "a+b" → {"ab", "aab", ...}
- "X?" は X(1文字 or 1グループ) が0回または1回繰り返されたものを表す
 - "a?" → {"", "a"}

正規表現による文字列検索

- ある正規表現で表現される文字列をある文字列中から検索する
- JavaScriptでは `str.match(new RegExp("正規表現", "オプション"))` で正規表現で表現される文字列を `str` 中から検索し、マッチした文字列を配列として得る
- `str.match(new RegExp("正規表現", "オプション"))`
 - "g" オプションは「マッチしたもの全て」を表す

main.js

```
console.log( "xixav".match(new RegExp("x.", "g")) ); // ["xi", "xa"]
```

正規表現による文字列検索

- `new RegExp("正規表現", "オプション")` と書くかわりに
`/正規表現/オプション` と書ける

main.js

```
console.log( "xixav".match(/x./g) ); // ["xi", "xa"]
```


正規表現による文字列検索

- * は実は最長マッチを表す

main.js

```
console.log( "rarar".match(/r.*r/g) ); // ["rarar"]
```

最短マッチ

- 最短マッチは *?

main.js

```
console.log( "rarar".match(/r.*r/g) ); // ["rarar"]  
console.log( "rarar".match(/r.*?r/g) ); // ["rar"]
```

正規表現による文字列検索

- マッチしなければ null となる

main.js

```
console.log( "rarar".match(/hoge/) ); // null
```

正規表現による文字列検索

- null は falsy なので正規表現にマッチする文字列を含んでいるかを判定するには
if (str.match(/reg/)) { ... }

main.js

```
'use strict'  
if ( "hage".match(/age/) ) {  
    console.log("マッチしました!");  
}  
else {  
    console.log("マッチせず...");  
}
```

先頭文字, 終端文字

- 正規表現では, 擬似的な文字として, 文字列の先頭を表す先頭文字 (先頭の1文字ではなく, その前に擬似的に挿入される文字) と, 文字列の終端を表す終端文字がある
- 先頭文字は「^」, 終端文字は「\$」で表現する

main.js

```
console.log( "drafear".match(/^dra/) ); // ["dra"]  
console.log( "drafear".match(/^fear/) ); // null  
console.log( "drafear".match(/fear$/) ); // ["fear"]
```

ここまでのまとめ

^	先頭文字
\$	終端文字
*	直前文字/グループは0回以上繰り返されてもマッチする
+	直前文字/グループは1回以上繰り返されてもマッチする
?	直前文字/グループはあってもなくてもマッチする
(reg)	グループ化する
x y	xまたはyにマッチする

演習

次の文字列であるかどうかを判定する関数 `check` を作ってみよう
(文字列 を受け取って `true` か `false` を返す)

1. `a`が含まれる文字列
2. 3文字以上の文字列
3. `a`から始まって`b`で終わる文字列
4. 奇数長の文字列
5. `a`が3個以上含まれる文字列

演習

1. aが含まれる文字列

main.js

```
'use strict'
const check = (str) => {
  if ( str.match(/a/) ) return true;
  return false;
};
console.log( check("hage") ); // true
console.log( check("hoge") ); // false
```


演習

2. 3文字以上の文字列

main.js

```
const check = (str) => {  
  return Boolean(str.match(/.../));  
};
```

main.js (別解)

```
const check = (str) => {  
  return str.length >= 3;  
};
```

演習

3. aから始まってbで終わる文字列

main.js

```
const check = (str) => {  
  return Boolean(str.match(/^a.*b$/));  
};
```

演習

4. 奇数長の文字列

main.js

```
const check = (str) => {  
  return Boolean(str.match(/^(..)*$/));  
};
```

main.js (別解)

```
const check = (str) => {  
  return str.length % 2 === 1;  
};
```

演習

5. aが3個以上含まれる文字列

main.js

```
const check = (str) => {  
  return Boolean(str.match(/a.*a.*a/));  
};
```

補足

- 普通 if 文の中で書くので `return Boolean(str.match(...))` と書くことはなさそう
- マッチしたか否かだけ知りたい場合, `test` や `search` の方が高速
 - `match`より高速ってだけで, 知らなくても全然問題ない
- 次の2つは等価 (`test` 関数にも興味があれば調べてみてください)

main.js

```
const check = (str) => {  
  return Boolean(str.match(/a.*a.*a/));  
};
```

main.js

```
const check = (str) => {  
  return str.search(/a.*a.*a/) >= 0;  
};
```

メタ文字

<code>x{n}</code>	直前文字/グループはちょうどn回の繰り返しにマッチする
<code>x{n,}</code>	直前文字/グループはn回以上の繰り返しにマッチする
<code>x{n,m}</code>	直前文字/グループはn回以上m回以下の繰り返しにマッチする

メタ文字

[xyz]	その文字集合のどれか1文字にマッチする. a-z や A-Z, 0-9 などとも書ける.
[^xyz]	その文字集合に含まれないどれか1文字にマッチする.

main.js

```
console.log( "abwwcabw".match(/[abc]*/) ); // ["ab"]  
console.log( "<h1>hello</h1>".match(/<[^>]*>/) ); // ["<h1>"]
```

メタ文字

¥d	[0-9] に同じ
¥D	[^0-9] に同じ
¥w	[A-Za-z0-9_] に同じ
¥W	[^A-Za-z0-9_] に同じ
¥s	スペース, タブ, 改行など, 1個のホワイトスペース文字にマッチ
¥S	¥s 以外の1文字にマッチ

グループ化して取り出す

- グループ化すると, `match` 関数でマッチしたときにその内容が戻り値の配列の中に入る
 - この括弧を キャプチャリング括弧 と呼ぶ
 - キャプチャリングするときは `g` オプション付けちゃダメ

main.js

```
console.log( "x=10, y=30".match(/x=(\d*)/) ); // ["x=10", "10"]
console.log( "<h1>hello</h1>".match(/<([^\>][^\>]*)>/) ); // ["<h1>", "h1"]
console.log( "a:10".match(/^(.*):(.*)$/) ); // ["a:10", "a", "10"]
```

非キャプチャリング括弧

- キャプチャリングしたくない場合は (x) と書くかわりに (?:x) と書く

main.js

```
console.log( "ab10aa10ac20".match(/(?:aa|bb|cc)(\d+)/) ); // ["aa10", "10"]
```

replace

- str.replace でも正規表現が使える

main.js

```
// 数字だけ取り出して数値に変換する  
console.log( +"asf892aw15k".replace(/¥D/g, "") ); // 89215  
// pxを取り去る  
console.log( +"10px".replace(/px$/, "") ); // 10
```

演習

次の文字列処理または判定を行う関数 f を作ってみよう

1. アルファベット小文字または大文字を全て削除する
- "aw25aw_@" -> "25_@"
2. div要素のHTMLから内容を取得する
- '<div id="hoge">hello</div>' → hello
3. 数字, 英小文字, 英大文字, 記号(数字でも英小文字でも英大文字でもないもの) のうち3種類以上含まれているか判定する

演習

1. アルファベット小文字または大文字を全て削除する
- "aw25aw_@" -> "25_@"

main.js

```
const f = (str) => {  
  return str.replace(/[A-Za-z]/g, "");  
};
```

演習

2. div要素のHTMLから内容を取得する
- '<div id="hoge">hello</div>' → hello

main.js

```
const f = (str) => {  
  return str.match(/<div>(.*?)<\/div>/, "")[1];  
};
```

演習

3. 数字, 英小文字, 英大文字, 記号(数字でも英小文字でも英大文字でもないもの) のうち3種類以上含まれているか判定する

main.js

```
const f = (str) => {  
  let cnt = 0;  
  if ( str.match(/¥d/) ) ++cnt;  
  if ( str.match(/[a-z]/) ) ++cnt;  
  if ( str.match(/[A-Z]/) ) ++cnt;  
  if ( str.match(/[^¥da-zA-Z]/) ) ++cnt;  
  return cnt >= 3;  
};
```

\$数字

- `str.replace(reg, toStr)` の `toStr` で `$1`, `$2`, ... を使うと, マッチしたそのグループの文字列になる
 - `$1` なら 1番目のグループ の文字列を表す

例

```
console.log( "<div>yeah</div>".replace(/<(.*?)>/g, "[$1]") ); // [div]yeah[/div]
```


¥数字・先読み・後読み

- jsでは先読みが実装されていない

	¥1, ¥2, ...	n番目のグループと同じ文字列
肯定後読み	<code>x(?=y)</code>	yが直後に続くxにマッチ (マッチした文字列にyは含まない)
否定後読み	<code>x(?!y)</code>	yが直後に続かないxにマッチ
肯定先読み	<code>(?<=x)y</code>	xが直前にあるyにマッチ
否定先読み	<code>(?<!x)y</code>	xが直前にないyにマッチ

例

```
console.log( "aaabaa".match(/(a|b)*/) ); // ["aaabaa"]
console.log( "aaabaa".match(/(a|b)¥1*/) ); // ["aaa"]
// 1,145,141,919 (3桁区切りでカンマ)
console.log( "1145141919".replace(/(¥d)(?=(¥d¥d¥d)+$)/g, "$1,") );
```



4. Web API いろいろ

File API

- ローカルファイルの読み込みを実現する
- 例としてドラッグ&ドロップしたファイルの中身を表示する
 - まずはドラッグ&ドロップされたファイルの情報を表示してみよう

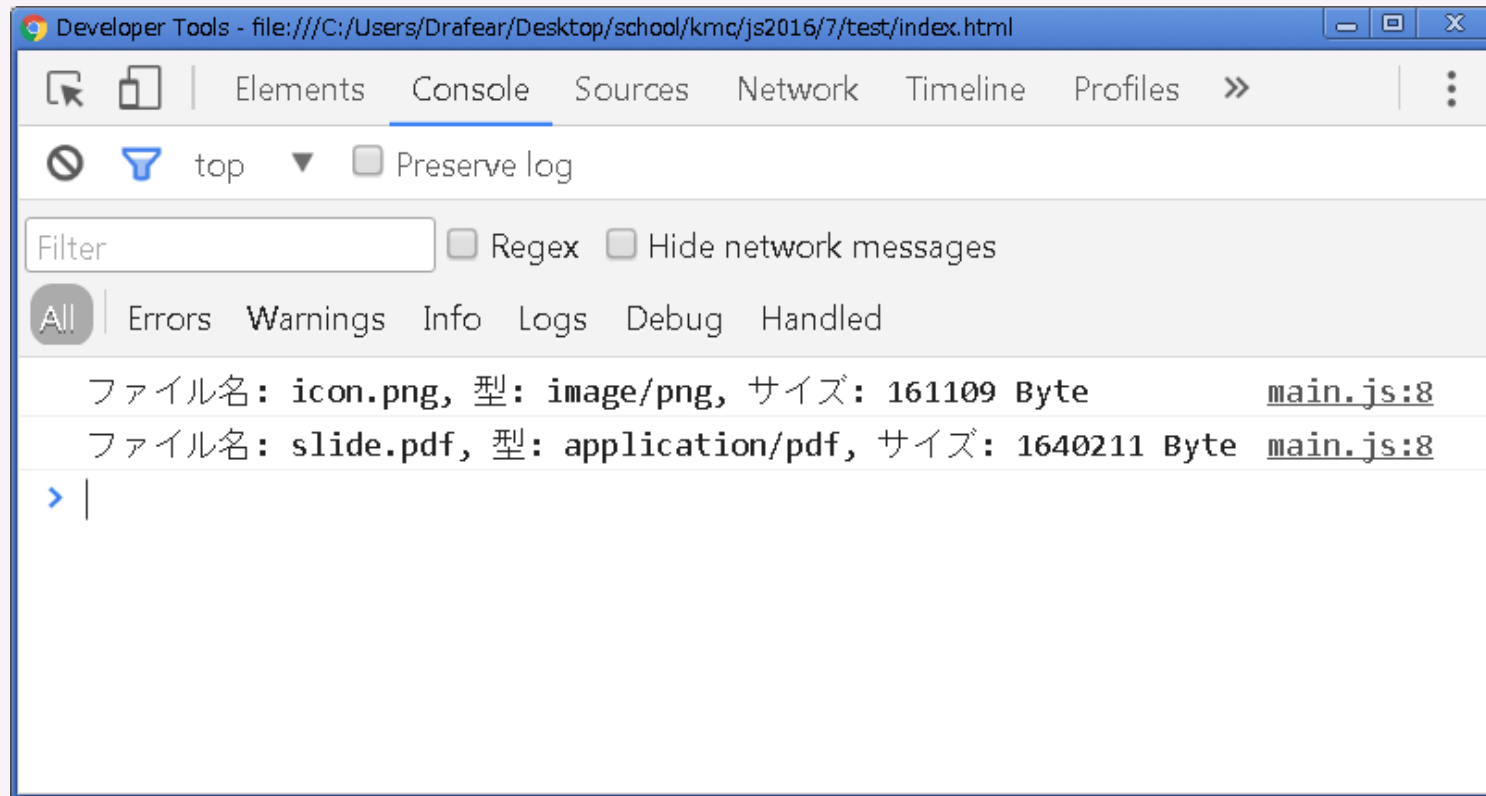
File API

main.js

```
'use strict'
document.addEventListener("drop", (e) => {
  // D&Dされたファイル情報リストを配列(っぽいもの)で得る
  const files = e.dataTransfer.files;
  // 各ファイルに対して処理
  for (let i = 0; i < files.length; ++i) {
    const file = files[i];
    console.log(`ファイル名: ${file.name}, 型: ${file.type}, サイズ: ${file.size} Byte`);
  }
  // ブラウザが勝手にファイルを開く処理を中止
  e.preventDefault();
});
document.addEventListener("dragover", (e) => {
  // ブラウザが勝手にファイルを開く処理を中止
  e.preventDefault();
});
```

File API

- ドラッグ&ドロップしてみると...



File API

- 今回は document に addEventListener しましたが一般の要素に対してもできるので是非試してみてください

File API

- 実はまだ File API を触っていない...
- FileReader API を使ってファイルを読み込んでみよう！

main.js

```
'use strict'
const loadFile = (file) => { ... };
document.addEventListener("drop", (e) => {
  const files = e.dataTransfer.files;
  for (let i = 0; i < files.length; ++i) {
    loadFile(files[i]);
  }
  e.preventDefault();
});
document.addEventListener("dragover", (e) => {
  e.preventDefault();
});
```

File API

- FileReader.readAsText(file)
 - 非同期で読み込んでテキストデータとして得る
 - 非同期とはバックグラウンドで並列的に行うこと
(その処理の終了を待たずに他の処理を行える)

```
js
```

```
'use strict'
const loadFile = (file) => {
  const reader = new FileReader();
  reader.addEventListener("load", (e) => {
    console.log(e.currentTarget.result);
  });
  reader.readAsText(file);
};
```


File API

- そういえばこうすることもできるよね??
 - クロージャの項参照

js

```
'use strict'
const loadFile = (file) => {
  const reader = new FileReader();
  reader.addEventListener("load", (e) => {
    console.log(result);
  });
  reader.readAsText(file);
};
```

File API

- reader.readAsText によくわからないオブジェクトを渡したけど単純にファイルパスを渡したりできないのか？
 - 端的に言うとできません
 - セキュリティ上の問題で, ユーザから何かしらの要求がないとファイル操作をできないようになっている
 - ファイルパスで読み込めたとすると, ユーザが意図していないのに勝手に読み込んでそしてどこかのサーバにアップしたりできるので例えばCドライブ以下の全ファイルをよこせ！！的なことも...

File API

- 他にも色々機能はあるが割愛
 - 画像ファイルを読み込んで表示
(`reader.readAsDataURL`)
 - 擬似的なファイル(URL Scheme)を生成してダウンロードリンクを取得
(`Blob`, `document.URL.createObjectURL`)

Battery Status API

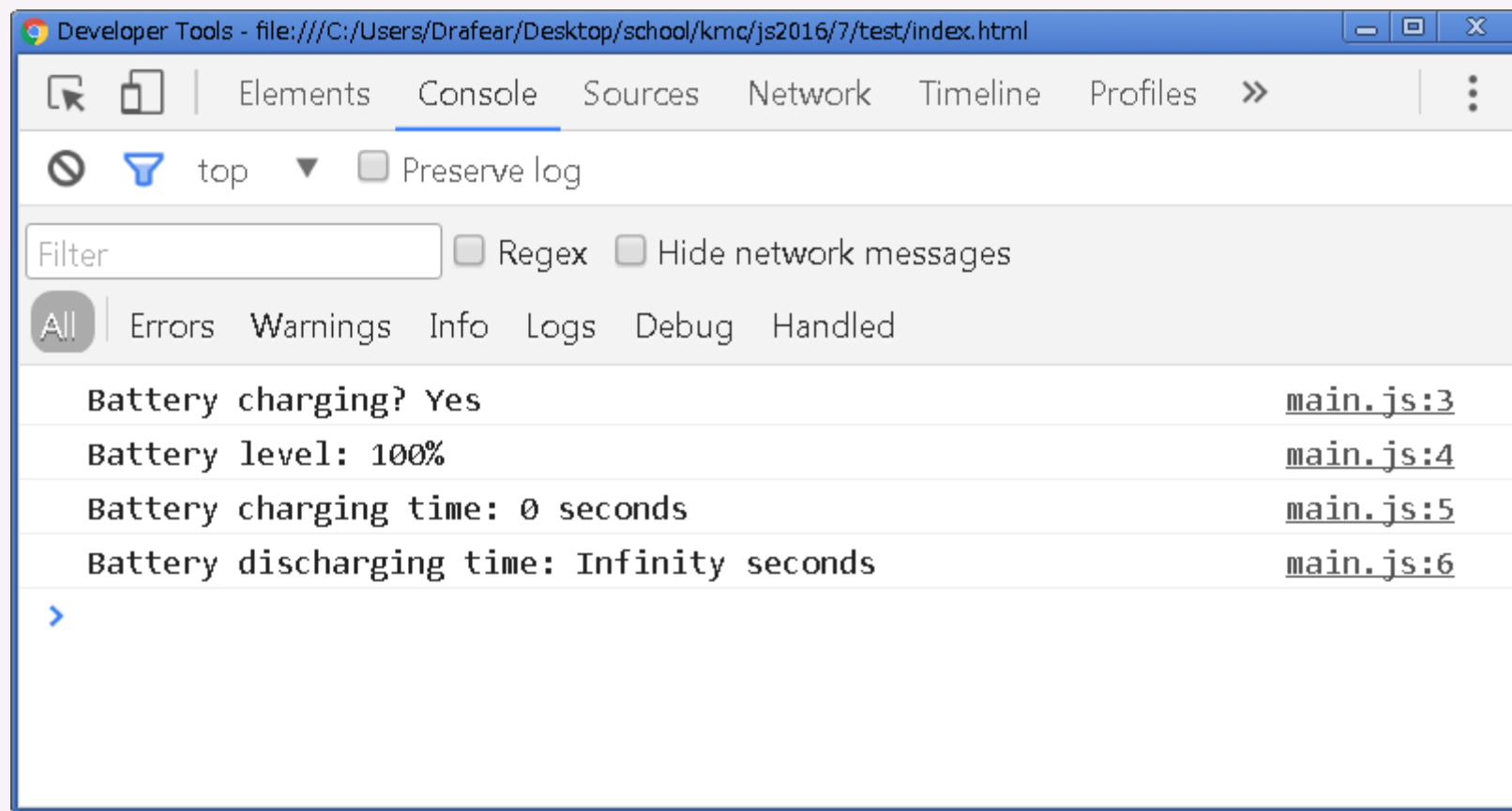
- navigator.getBattery() でバッテリーの情報を取得する
- navigator.getBattery().then(func)で情報取得に成功したときに実行される関数funcを登録する
 - 詳しくは Promise で検索！

JS

```
'use strict'
navigator.getBattery().then((battery) => {
  console.log("Battery charging? " + (battery.charging ? "Yes" : "No"));
  console.log("Battery level: " + battery.level * 100 + "%");
  console.log("Battery charging time: " + battery.chargingTime + " seconds");
  console.log("Battery discharging time: " + battery.dischargingTime + " seconds");
});
```

Battery Status API

- 実行例



Battery Status API

- 充電状態の変化に反応するイベントもあります！

JS

```
'use strict'
navigator.getBattery().then((battery) => {
  battery.addEventListener('chargingchange', (e) => {
    console.log("Battery charging? " + (battery.charging ? "Yes" : "No"));
  });
  battery.addEventListener('levelchange', (e) => {
    console.log("Battery level: " + battery.level * 100 + "%");
  });
  battery.addEventListener('chargingtimechange', (e) => {
    console.log("Battery charging time: " + battery.chargingTime + " seconds");
  });
  battery.addEventListener('dischargingtimechange', (e) => {
    console.log("Battery discharging time: " + battery.dischargingTime + " seconds");
  });
});
```

Geolocation API

- GPSを使える！！

- ただしユーザの許可が必要
- 使おうとすると「許可しますか？」と出る
- デモなど:

https://developer.mozilla.org/ja/docs/WebAPI/Using_geolocation

JS

```
'use strict'  
navigator.geolocation.getCurrentPosition((position) => {  
  console.log(`緯度: ${position.coords.latitude}`);  
  console.log(`経度: ${position.coords.longitude}`);  
});
```

Pointer Lock API

- マウ斯卡ーソルを動けなくする
 - FPSゲームを作るときなどに便利
- ローカルでは実行が難しいので以下で使い方を見してみる
 - <http://hai3.net/blog/2013/06/23/javascript-pointer-lock/>

Device Orientation API

- デバイスの傾きを取得！すごい！
 - https://developer.mozilla.org/en-US/docs/Web/API/Detecting_device_orientation



Screen Orientation API

- スクリーンの向きを取得したり
向きの変化を監視するイベントなど
 - https://developer.mozilla.org/ja/docs/WebAPI/Managing_screen_orientation

Vibration API

- バイブを鳴らす
 - <http://hi-posi.co.jp/tech/?p=150>

JS

```
'use strict'  
navigator.vibrate(200); // バイブを200ms鳴らす  
navigator.vibrate(0); // バイブを止める  
// 200ms鳴らして100ms休憩して300ms鳴らす  
navigator.vibrate([200, 100, 300]);
```

Notification API

- 通知機能
 - https://developer.mozilla.org/ja/docs/WebAPI/Using_Web_Notifications
- 興味があれば読んでみてくださいmm

Web Storage API

- ブラウザを閉じててもデータを残しておける
- cookieのようなもの
- プロトコル, ドメイン, ポート番号ごとに保存される
 - <http://abc.sample.com:8080/hoge/fuga/a.html>

Web Storage API

- 保存

`localStorage.key = val`

- 取得

`localStorage.key`

- 削除

`localStorage.removeItem(key)`

- 全削除

`localStorage.clear()`

JS

```
'use strict'
```

```
// ページを更新するたびにカウントが1ずつ増える
```

```
localStorage.test = +(localStorage.test || 0) + 1;
```

```
console.log(localStorage.test);
```

Web Storage API

- 保存されてあるキーの総数
 localStorage.length
- 保存されてあるi番目のキーを取得
 localStorage.key(i)

JS

```
'use strict'  
for (let i = 0; i < localStorage.length; ++i) {  
    console.log(localStorage.key(i));  
}
```

Web Storage API

- ウィンドウを閉じるまでだけデータを残したい場合は `localStorage` のかわりに `sessionStorage` を使う

Web Storage API

- storageへの読み込み/書き込みアクセス監視
window.addEventListener("storage", (e) => { ... })
e.key : キー
e.oldValue: 前の値
e.newValue: 次の値
e.url: アクセスを行ったURL
e.storageArea: storageオブジェクト

Gamepad API

- ゲームパッドの入力を受け取れる
- 接続されているゲームパッドの情報を取得できる
- ゲームパッドの接続状態の変化をイベント処理できる

Drag API

- 要素のドラッグ・要素へのドロップイベントを監視できる
 - <http://www.html5rocks.com/ja/tutorials/dnd/basics/>

Clipboard API

- コピー・カット・ペーストのイベントを監視できる
- コピー・カット・ペーストを行える

ボタンクリックでテキストコピー

```
document.querySelector("#btn").addEventListener("click", (e) => {  
  document.querySelector("#textbox").select();  
  document.execCommand("copy");  
});
```

History API

- ブラウザ履歴を取得できる
- ブラウザ履歴に追加できる
 - 戻るを押したときの戻り先を追加できる

履歴に追加する例

```
'use strict'  
const state = {};  
const title = "test";  
const url = "test.html";  
history.pushState(state, title, url);
```

Location

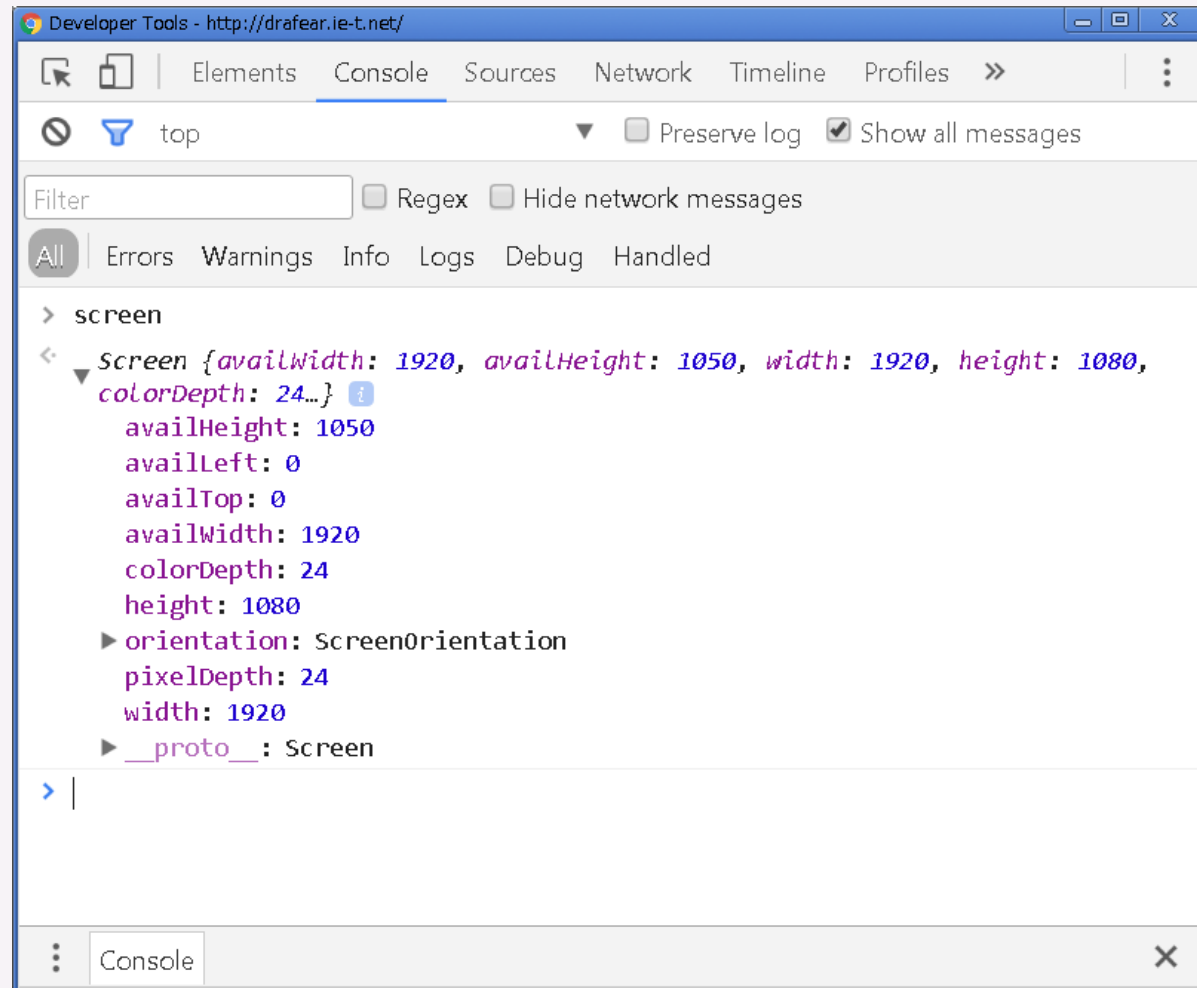
- 現在のページの情報取得, 変更できる

ページ切り替え例

```
'use strict'  
// a.html に移動  
location.href = "a.html";
```

Screen

- モニタの情報や描画領域のサイズを取得できる



Window

- ウィンドウの情報を取得できる
- ウィンドウを作成・消去・移動できる
- スクロール位置を取得・変更できる

a.htmlを新しいタブで開く

```
window.open("a.html", null);
```


Date

- 時刻を扱える
- 現在時刻を取得できる
 - http://hakuhin.jp/js/date.html#DATE_00

現在時刻を表示する

```
console.log( new Date() );
```

時間を計測する(ミリ秒)

```
const d1 = new Date();  
for (let i = 0; i < 100000000; ++i);  
const d2 = new Date();  
console.log(d2 - d1);
```

実際に使ってみる

- 実際に使って何かを作ってみよう
- 例えば Web Storage API を使って
今まで作ったゲームにセーブ機能を追加したり
 - クッキーの数を保存
 - ジャンケンの対戦履歴を保存
 - 避けゲーのハイスコアを保存