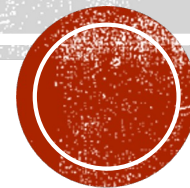


# JAVASCRIPT 進階

林新德

shinder.lin@gmail.com



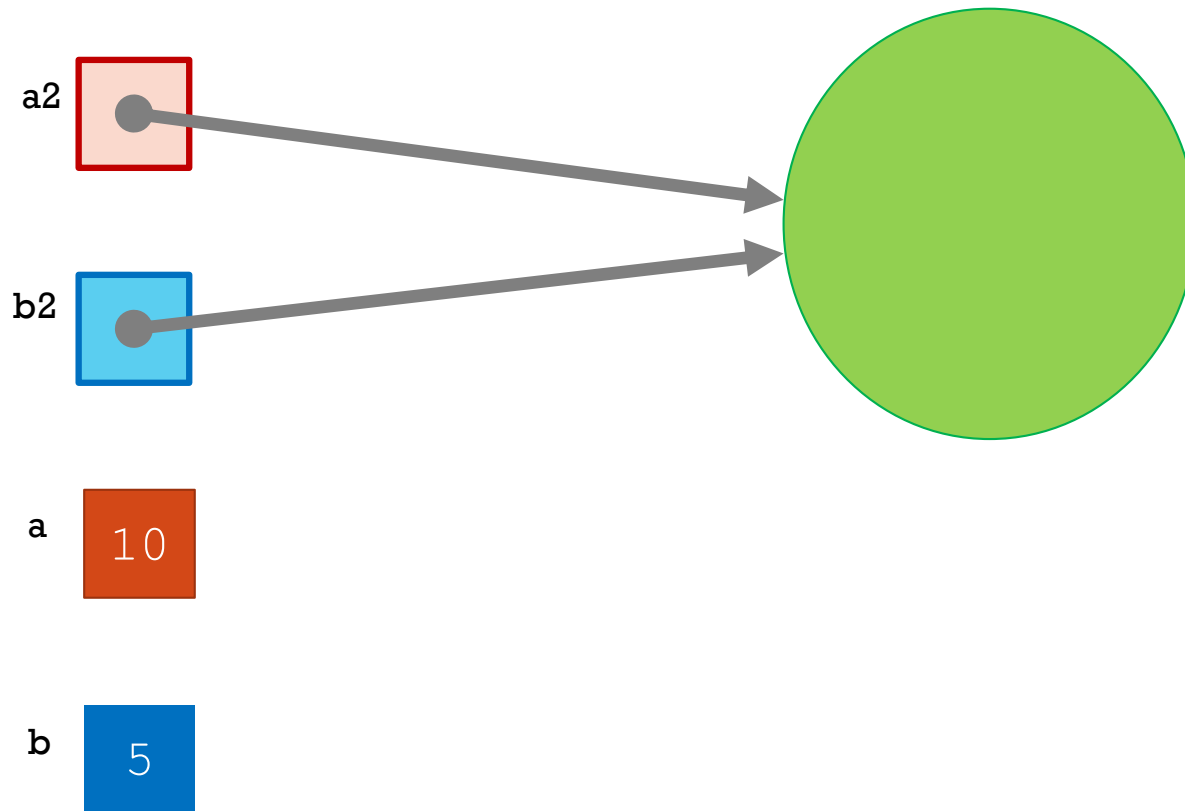
參考專案 <https://bitbucket.org/lzd0125/mfee33-js/>

# 課程涵蓋內容

- 事件浮出模型
- 可迭代類型
- 物件和陣列的複製
- 函式進階
  - 遞回
  - 閉包
- 陣列的高階方法
  - .filter()
  - .map()
  - .reduce()
  - .sort()
- 非同步程式設計
  - Promise 類型
  - async/await 運算子
- 物件導向程式設計
- ESM 模組操作 ( import, export )
- 不刷頁面變換網址
- 正規表示法 ( 補充資料 )



参照 (reference) 概念



標籤的事件處理器	說明
onclick ondblclick onmousedown onmousemove onmouseover onmouseout onmouseup	單擊滑鼠左鍵。 雙擊滑鼠左鍵。 按下滑鼠左鍵時。 滑鼠在元素上移動時。 滑鼠移入元素時。 滑鼠移開元素時。 放開滑鼠左鍵時。
onkeydown onkeypress onkeyup	按下按鍵時。 按住按鍵時重複觸發。 放開按鍵時。
onload	內容載入後，使用於<body>。
onresize	文件大小改變時。
onscroll	文件捲動時。
onblur	失焦時，用於表單內的元素。
onchange	內容改變時，用於<input>、<select>、<textarea>。
onfocus	取得焦點時，用於表單內的元素。
onreset	重置時，用於表單。
onselect	選取部份內容時，用於<input>、<textarea>。
onsubmit	送出表單時。

## \*\* 事件的浮出模型

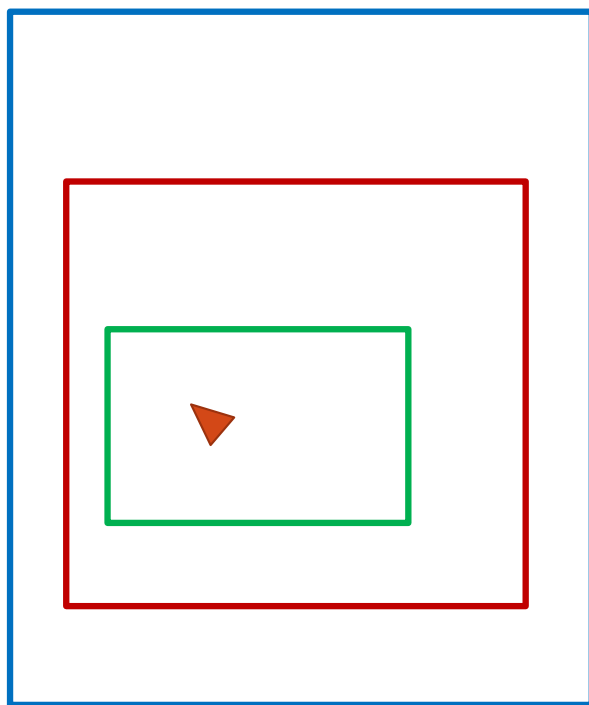


```
<button onclick="dosomething(event)">Hello</button>
<script>
  const btn = document.querySelector('button');
  function dosomething(evt){
    console.log('1');
  }
  btn.onclick = function(){
    console.log('2');
  };
  btn.addEventListener('click', function(){
    console.log('3');
  });
  btn.addEventListener('click', function(){
    console.log('5');
  });
  const listener = (event)=>{
    console.log(event);
  };
  btn.addEventListener('click', listener);
</script>
```

onclick只能選一種使用



# 事件浮出模型



## 10.1 事件浮出模型

```
<div class="rect">
  <div class="ball"></div>
</div>
<script>
  const rect = document.querySelector(".rect");
  const ball = document.querySelector(".ball");

  const handler = (event) => {
    console.log("target:", event.target);
    console.log("currentTarget:", event.currentTarget);
    console.log(event.eventPhase);
  };

  ball.addEventListener("click", handler);
  rect.addEventListener("click", handler);
  document.addEventListener("click", handler);
  window.addEventListener("click", handler);
</script>
```

```
<style>
  .rect {
    position: relative;
    width: 800px;
    height: 600px;
    background-color: lightcyan;
    border: 1px solid black;
  }
  .ball {
    position: absolute;
    width: 150px;
    height: 150px;
    border-radius: 50%;
    background-color: yellow;
    text-align: center;
    border: 1px solid black;
    left: 100px;
    top: 100px;
  }
</style>
```



## CAPTURING and AT TARGET

```
<div class="rect">
  <div class="ball"></div>
</div>
<script>
  const rect = document.querySelector(".rect");
  const ball = document.querySelector(".ball");

  const handler = (event) => {
    console.log("target:", event.target);
    console.log("currentTarget:", event.currentTarget);
    console.log(event.eventPhase);
  };
  ball.addEventListener("click", handler, true);
  rect.addEventListener("click", handler, true);
  document.addEventListener("click", handler, true);
  window.addEventListener("click", handler, true);
</script>
```



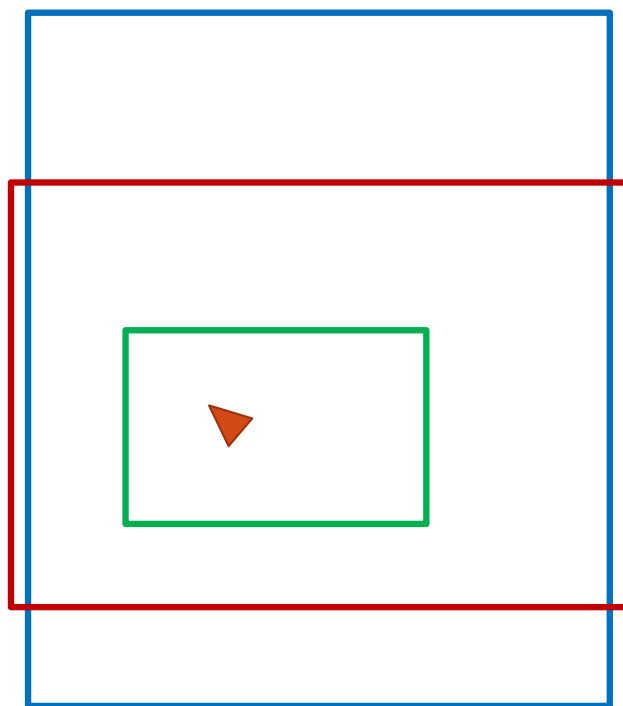


- 將上頁範例修改為：

```
rect.addEventListener('click', function (event) {  
    console.log('1:', event.eventPhase, event.currentTarget, event.target);  
    event.stopPropagation(); // 阻斷事件  
}, true); // useCapture
```



# 滑鼠事件座標



內容頁 `event.pageX`  
`event.pageY`

顯示區 `event.clientX`  
`event.clientY`

被點擊的元素 `event.offsetX`  
`event.offsetY`



```
<div class="rect">
  <div class="ball"></div>
</div>
<div id="info"></div>
<script>
  const rect = document.querySelector(".rect");
  const ball = document.querySelector(".ball");
  const info = document.querySelector("#info");

  rect.addEventListener("mousemove", function (event) {
    info.innerText = `
      client: ${event.clientX}, ${event.clientY},
      page:   ${event.pageX},    ${event.pageY},
      offset: ${event.offsetX},  ${event.offsetY},
    `;
  });
</script>
```



# getBoundingClientRect()

```
<style>
* {
  margin: 0;
  padding: 0;
}
.face {
  position: relative;
  width: 600px;
  height: 600px;
  border-radius: 50%;
  background-color: #3ecc6c;
  border: 1px solid black;
}
.eye {
  position: absolute;
  left: 200px;
  top: 300px;
}
</style>
```

```
<style>
.eye-white {
  position: absolute;
  top: -60px;
  left: -60px;
  width: 100px;
  height: 100px;
  background-color: #fafff9;
  border-radius: 50%;
  border: 10px solid black;
}
.eye-black {
  position: absolute;
  top: -25px;
  left: 0px;
  width: 50px;
  height: 50px;
  background-color: #0a093b;
  border-radius: 50%;
}
</style>
```

轉動的眼睛 - 1

```
<div class="face">
  <div class="eye">
    <div class="eye-white"></div>
    <div class="eye-black"></div>
  </div>
  <div class="eye" style="left:300px;top:200px">
    <div class="eye-white"></div>
    <div class="eye-black"></div>
  </div>
  <div class="eye" style="left:400px;top:300px">
    <div class="eye-white"></div>
    <div class="eye-black"></div>
  </div>
</div>
```



## 轉動的眼睛 - 2

```
<script>
  const eyes = document.querySelectorAll('.eye');
  window.addEventListener('mousemove', function(event){
    eyes.forEach(function(eye){
      const rect = eye.getBoundingClientRect();
      console.log(rect);
      const dx = event.pageX - rect.x;
      const dy = event.pageY - rect.y;
      const ang = Math.atan2(dy, dx)/Math.PI*180; // degree

      eye.style.transform = `rotate(${ang}deg)`;
    });
  });
</script>
```





## \*\* 可迭代的類型

- 常見的可迭代類型：Array、NodeList、HTMLCollection
- 單純 Object 類型是**不可**迭代的
- 如何判斷是否為可迭代類型：擁有 Symbol.iterator 方法
- 可展開為陣列 [...可迭代物件]
- 可使用 for/of 迴圈



```
<div class="mydiv">123</div>
<div class="mydiv">456</div>
<script>
  const ar = [7, 9, 3, 5];
  const obj = {name: 'shinder'};
  const divs = document.querySelectorAll('.mydiv');

  console.log(ar[Symbol.iterator]);
  // f values() { [native code] }

  console.log(obj[Symbol.iterator]);
  // undefined

  console.log(divs[Symbol.iterator]);
  // f values() { [native code] }
</script>
```





```
<div class="mydiv">123</div>
<div class="mydiv">456</div>
<script>
  const ar = [7, 9, 3, 5];
  const obj = {name: 'shinder'};
  const divs = document.querySelectorAll('.mydiv');
  let ar2, obj2, divs2;

  ar2 = [...ar];
  try {
    const obj2 = [...obj];
  } catch(ex){
    console.log(ex); // TypeError: obj is not iterable
  }
  divs2 = [...divs];
</script>
```



```
<div class="mydiv">123</div>
<div class="mydiv">456</div>
<script>
  const divs = document.querySelectorAll('.mydiv');
  for(let d of divs){
    console.log(d);
  }
</script>
```





## \*\* 物件和陣列的複製

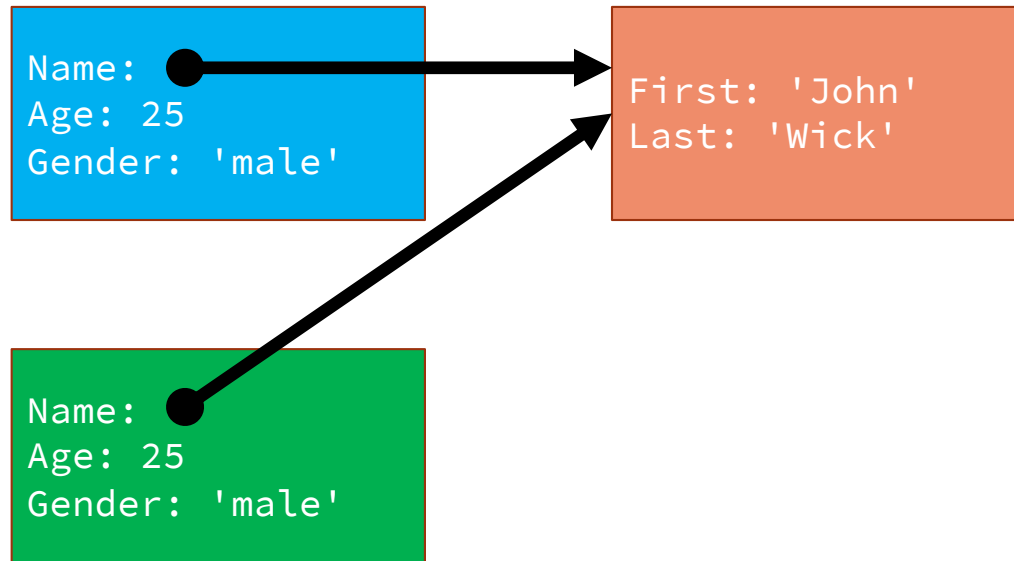
- 淺層複製（單層複製）
- 可使用展開運算子（...）

```
const ar = [12, 99, 'aaa', ['bill', 25, 'male'] ];
const br = ar;
const cr = ar.slice(); // first level copy
const dr = [...ar];    // first level copy
ar[3][0] = 'flora';
ar[0] = 100;

console.log('ar:', ar);
console.log('br:', br);
console.log('cr:', cr);
```



展開複製為淺層複製



- 深層複製（舊的用法透過JSON轉換）

```
const ar = [12, 99, 'aaa', ['bill', 25, 'male']];  
const br = ar;  
const str = JSON.stringify(ar);  
const cr = JSON.parse(str);  
ar[3][0] = 'flora';  
ar[0] = 100;  
console.log('ar:', ar);  
console.log('br:', br);  
console.log('cr:', cr);
```

- 目前深層複製建議使用 structuredClone()

<https://developer.mozilla.org/en-US/docs/Web/API/structuredClone>



```
const p1 = {name: 'John', age: 23};
const p2 = {name: 'David', age: 25};
const p3 = {name: 'Bill', age: 24};
p1.next = p2;
p2.next = p3;
p3.next = p1;
console.log(p1);

const r1 = window.structuredClone(p1);
r1.name = r1.name.toUpperCase();
r1.next.name = r1.next.name.toUpperCase();
r1.next.next.name = r1.next.next.name.toUpperCase();

console.log(r1);
```







## \*\* 函式進階

### 遞迴 ( Recursion )

- 自己呼叫自己的函式。

```
function f(n) {  
    return n <= 1 ? 1 : n * f(n - 1);  
}  
console.log(f(6));
```



- 費氏數列 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 ...

$$F_0 = 0$$

$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}$$

```
function fib(n) {  
  if (n <= 0) return 0;  
  if (n === 1) return 1;  
  return fib(n - 1) + fib(n - 2);  
}  
  
for (let i = 1; i <= 12; i++) {  
  console.log(fib(i));  
}
```



```
// 沒有使用快取
function fib(n) {
  if (n <= 0) return 0;
  if (n === 1) return 1;
  return fib(n - 1) + fib(n - 2);
}

const startTime = Date.now();
for (let i = 1; i <= 40; i++) {
  console.log(fib(i));
}

console.log(`${Date.now() - startTime} ms`); // 102334155, 2518 ms
```



```
// 使用快取
const cache = [0, 1];

function fib(n) {
  if (cache[n]) return cache[n];
  if (n <= 0) return 0;
  if (n === 1) return 1;
  const v = fib(n - 1) + fib(n - 2);
  cache[n] = v;
  return v;
}

const startTime = Date.now();
for (let i = 1; i <= 40; i++) {
  console.log(fib(i));
}

console.log(`${Date.now() - startTime} ms`); // 102334155, 1 ms
```

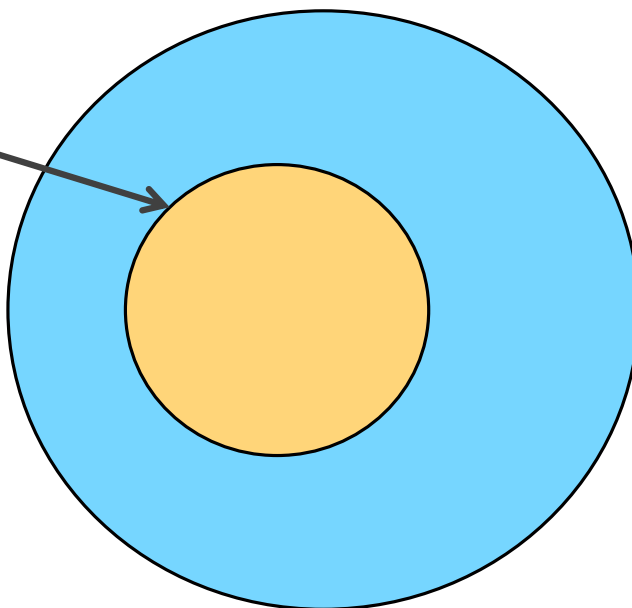
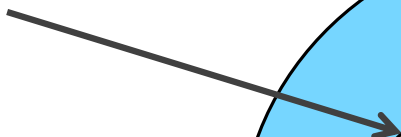


# Closure ( 閉包 )

```
const f = (function () {  
  let n = 3;  
  return function (a) {  
    n--;  
    if (n >= 0) {  
      return a * a;  
    } else {  
      return null;  
    }  
  };  
})();  
  
console.log(f(6));  
console.log(f(7));  
console.log(f(8));  
console.log(f(9));
```



f



## Arrow functions 和傳統函式差異

```
function f1(name, age) {  
}  
  
let f2 = (name, age) => {  
};  
  
const f3 = ()=>{  
};  
  
const f4 = n => n*n;  
  
const f5 = function(n){  
    return n*n;  
};
```



```

const obj = {
  name: 'shinder',
  tranditional: function () { return this; },
  tranditional2() { return this; },
  arrow: () => this,
}

for (let i in obj) {
  if (obj[i] instanceof Function) {
    console.log('1st: ', obj[i].name, obj[i]());
  }
}

console.log('-----');
for (let i in obj) {
  if (obj[i] instanceof Function) {
    const f = obj[i];
    console.log('2nd: ', f.name, f());
  }
}

```

```

1st:  tranditional  obj
1st:  tranditional2 obj
1st:  arrow         window
-----
2nd:  tranditional  window
2nd:  tranditional2 window
2nd:  arrow         window

```





```

const obj = {
  tranditional: function () {
    return this;
  },
  arrow: () => this,
}
const obj2 = {};
obj2.t_bind = obj.tranditional.bind(obj2);
obj2.a_bind = obj.arrow.bind(obj2);

for (let i in obj2) {
  console.log('1st: ', obj2[i].name, obj2[i]());
}
console.log('-----');
for (let i in obj2) {
  const f = obj2[i];
  console.log('2nd: ', f.name, f());
}

```

1st:	bound tranditional	obj2
1st:	bound arrow	window
-----		
2nd:	bound tranditional	obj2
2nd:	bound arrow	window



```
// 箭頭函式建立時，立即綁定 context
const obj = {
  name: 'shinder',
  arrowInTrand: function () {
    return () => this;
  },
}

console.log(obj.arrowInTrand());
const f = obj.arrowInTrand();
console.log(f());
const f2 = obj.arrowInTrand().bind({name: 'dephne'});
console.log(f2());
```





## \*\* 陣列的高階方法

- 排序是以字串（unicode字碼）為順序。

```
const str = '到底需要日曬多久才能幫助人體獲得足夠的維生素D';  
const ar = str.split('');  
ar.sort();  
console.log(ar);  
const br = [35, 6, 78, 12, 54, 9];  
br.sort();  
console.log(br);
```



- 自訂排序規則，數值由小到大：

```
const br = [35, 6, 78, 12, 54, 9];  
br.sort(function (a, b) {  
    return a - b;    // 負值對調  
});  
console.log(br);
```

- 自訂排序規則：

```
let people = [  
    {name: "David", age: 25},  
    {name: "Carl", age: 28},  
    {name: "Bill", age: 31},  
];  
people.sort(function (a, b) {  
    return b.age - a.age;  
});  
console.log(people);
```



## ▪ `forEach()` 的用法

```
<div id="rect"></div>
<script>
  const rect = document.querySelector('#rect');
  const people = [
    { name: 'david', age: 25, id: 'A006' },
    { name: 'bill', age: 27, id: 'A009' },
    { name: 'peter', age: 23, id: 'A011' },
  ];
  people.forEach(function(val, ind){
    rect.innerHTML += ind + ': ' + val.name + '<br>';
  });
</script>
```



▪ filter() 和 map() 的用法

```
const ar = [12,1,3,5,2,6,8,9,12];

let br = ar.filter(function(val){
    return val%2===0; //過濾條件
});
rect.innerHTML = br.toString() + '<br>';

let cr = ar.map(function(val){
    return val*val;
});
rect.innerHTML += cr.toString() + '<br>';

let dr = ar.map(function(val){
    var className = val%2 ? '' : 'red';
    return `<span class="${className}">${val}</span>`;
});
rect.innerHTML += dr.toString() + '<br>';
```



## ▪ reduce() 的用法

```
<pre class="rect"></pre>

<script>
  const rect = document.querySelector(".rect");
  const ar = [2, 7, 34, 17, 19, 23, 84];

  const t = ar.reduce(function (a, v, i) {
    rect.innerHTML += `${a}, ${v}, ${i}\n`;
    return a + v;
  });
  console.log(t);
</script>
```







# \*\* 非同步程式設計

## Promise ( ES6 )

- 用來**改善** `callback functions` 的問題。
- 使用於 `non-blocking IO` 時，非同步（異步，`Asynchronous`）的情況。
- `Promise` 物件建立時會進入 `pending`（等待期），等著 `resolve` 或 `reject` 被呼叫。



- 測試 Promise 用法
- [https://developer.mozilla.org/zh-TW/docs/Web/JavaScript/Guide/Using\\_promises](https://developer.mozilla.org/zh-TW/docs/Web/JavaScript/Guide/Using_promises)

```
// catch() 內的 callback function 會在發生駁回時呼叫
let promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    Math.random() > .5 ? resolve('ok') : reject('fail');
  }, 500)
});

promise.then(result => {
  console.log('result:', result);
})
.catch(ex => {
  // 只捕捉在此之前的駁回狀況 (* catch() 通常放最後面)
  console.log('ex:', ex);
})
.then(() => {
  console.log('3rd');
});
```



// 依序執行兩段非同步的程式片段

```
new Promise((resolve, reject) => {  
  setTimeout(() => {  
    resolve('123');  
  }, 1000);  
})  
  .then(result => {  
    console.log(result);  
    return new Promise((resolve, reject) => {  
      setTimeout(() => {  
        resolve('456');  
      }, 1500);  
    });  
  })  
  .then(result => {  
    console.log(result);  
  });
```



```
// Promise 以 function 包裝使用

const myPromise = (pid = 1) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      Math.random() > .5 ? resolve('000-' + pid) : reject('XXX-' + pid);
    }, 500 * Math.random())
  });
};

for (let i = 0; i < 10; i++) {
  // 在迴圈內，可能不會依照順序執行（若要依續執行應該使用 async/await）
  myPromise(i)
    .then(result => console.log({result}))
    .catch(ex => console.log({ex}));
}
```



## 使用 Promise 靜態方法

```
const startTime = Date.now();  
// 預先定義的測試函式  
  
const myPromise1 = (n) => {  
  const rnd = 100 + Math.floor(Math.random() * 500);  
  console.log({n, rnd});  
  return new Promise((resolve, reject) => {  
    setTimeout(() => resolve(`0:${n}-${rnd}`), rnd);  
  })  
};  
  
const myPromise2 = (n) => {  
  const rnd = 100 + Math.floor(Math.random() * 500);  
  console.log({n, rnd});  
  return new Promise((resolve, reject) => {  
    setTimeout(() => (  
      Math.random() > .5 ? resolve(`0:${n}-${rnd}`) : reject(`X:${n}-${rnd}`)  
    ), rnd);  
  })  
}
```



```
const pArr = [];  
for (let i = 1; i <= 10; i++) {  
  pArr.push(myPromise1(i));  
  // pArr.push(Promise.resolve(i)); // 立即完成的 promise 物件  
}  
  
Promise.all(pArr)  
  .then(r => {  
    // 所有 promise 物件都完成，才會進入這裡  
    console.log(r);  
    console.log(Date.now() - startTime);  
  }).catch(ex => {  
    // 只要有一個 promise 駁回，就會進入這裡  
    console.log(ex)  
  })
```



```
const pArr = [];  
for (let i = 1; i <= 10; i++) {  
  pArr.push(myPromise2(i));  
}  
  
Promise.any(pArr)  
  .then(r => {  
    // 等待第一個 promise 完成，就會進入這裡  
    console.log(r);  
    console.log(Date.now() - startTime);  
  }).catch(ex => {  
    // 全部的 promise 都駁回，會進入這裡  
    console.log(ex)  
  })
```





```
const pArr = [];  
for (let i = 1; i <= 10; i++) {  
  pArr.push(myPromise2(i));  
}  
  
Promise.race(pArr)  
  .then(r => {  
    // 只要有一個 promise 完成，就會進入這裡  
    console.log(r);  
    console.log(Date.now() - startTime);  
  }).catch(ex => {  
    // 只要有一個 promise 駁回，就會進入這裡  
    console.log(ex)  
  })
```



## async/await ( ES7 )

- 用來改善 Promise 。
- await 修飾的方法呼叫結果，必須是回傳 Promise 物件 。
- await 只能用在 async 宣告的方法內 。
- async 宣告的方法中，使用 await 的呼叫，有類似依序執行的效果 。
- 除錯應該使用 try/catch 結構 。



```
async function doSomething() {  
  let str = '';  
  for (let i = 0; i < 10; i++) {  
    str += await myPromise2(10 - i)  
      .then(result => {  
        console.log('result:', result);  
        return result + '\n';  
      })  
      .catch(ex => {  
        console.log('ex:', ex);  
        return '\n';  
      });  
  }  
  console.log(str);  
}  
  
doSomething();
```





# \*\* 物件導向程式設計

## 舊的物件導向

- 自訂類型

```
function Person(name, age) {  
    this.name = name;  
    this.age = age;  
    this.getInfo = function() {  
        return this.name + ':' + this.age;  
    }  
}  
  
var b = new Person('Bill', 32);  
console.log( b.getInfo() );  
console.log( b.name );
```



- 使用 **prototype** 擴充

```
function Person(name, age) {  
    this.name = name;  
    this.age = age;  
    this.getInfo = function() {  
        return this.name + ':' + this.age;  
    }  
}  
Person.prototype.toString = function() {  
    return JSON.stringify( this );  
};  
var b = new Person('Bill', 32);  
console.log( b.getInfo() );  
console.log( '' + b );
```



# ES6 的物件導向

```
class Person {  
  gender = 'male'; // 屬性預設值，屬性可以不用宣告  
  constructor(name = 'Shin', age = 20) {  
    this.name = name;  
    this.age = age;  
  }  
  getInfo() {  
    return `${this.name}: ${this.age}`;  
  }  
  toString = () => JSON.stringify(this)  
}  
  
const p = new Person('Shinder', 32);  
console.log(p.getInfo()); // Shinder: 32  
console.log('' + p);      // {"gender":"male","name":"Shinder","age":32}
```



```
// Employee 繼承 Person 的功能
class Employee extends Person {
  #id=''; // 私有屬性
  constructor(id, name, age) {
    super(name, age);
    this.#id = id;
  }
}

const p2 = new Employee('B007', 'David', 28);
console.log('' + p2); // {"gender":"male","name":"David","age":28}
// console.log(p2.#id); // 發生錯誤，不能使用私有屬性
```





```
class Employee2 extends Employee {
  #id; // 私有屬性不會被繼承，需要宣告
  constructor(id, name, age) {
    super(id, name, age);
    this.#id = id;
  }
  // getter
  get id () {
    return this.#id;
  }
  // setter
  set id(id) {
    this.#id = id;
  }
}

const p3 = new Employee2('C005', 'Joe', 27);
console.log('' + p3);
console.log(p3.id);
p3.id = 'D009';
```





## \*\* ESM 模組操作 ( import, export )

```
/* *** 檔案: tools/my-js01.js *** */  
console.log(import.meta.url); // 取得此 js 檔的路徑  
  
// 預設匯出  
export default function a(n) {  
    return n * n;  
}  
  
// 使用 const 或 let 時不能使用 export default  
export const b = (n) => n * n * n;  
export const c = 125;  
  
const d = 333;  
console.log({d}); // 注意輸出次數
```



- 以往 ECMAScript module (ESM) 必須經由 `babel.js` 等工具轉換才能執行。
- 目前瀏覽器已經支援 ECMAScript module 的功能。
- 在 HTML 檔的 `<script>` 加入 `type="module"` 屬性，才能使用 ESM 功能。

```
<script type="module">
  import defaultItem from './tools/my-js01.js';
  import {b as otherName, c} from './tools/my-js01.js';
  // import defaultItem, {b as otherName, c} from './tools/my-js01.js';

  console.log(defaultItem(10));
  console.log(otherName(3));
  console.log(c);
</script>
```



```
/* *** 檔案: tools/person.js *** */  
  
export default class Person {  
  constructor(name = 'Shinder', age = 28) {  
    this.name = name;  
    this.age = age;  
  }  
  
  toString = () => JSON.stringify(this)  
}  
const data = [1, 3, 5, 7, 9];  
  
export {data}; // 不是預設匯出時，可以用物件包裹
```



```
<script type="module">
  // 一般用法
  import Person, {data} from './tools/person.js';

  import * as all from './tools/person.js';

  console.log(all);    // Module {Symbol(Symbol.toStringTag): 'Module'}
  const p2 = new all.default('Victor', 35);
  all.data.reverse();
  console.log(p2.toString()); // {"name":"Victor","age":35}
  console.log(all.data);    // [9, 7, 5, 3, 1]
</script>
```



```
/* *** 檔案: tools/index.js *** */  
// 通常是一個檔案放一個類別或一個函式  
import Person, {data} from './person.js';  
import a, {b, c} from './my-js01.js';  
  
export { a, b, c, Person, data };
```

```
/* *** 檔案: tools/index.js *** */  
  
export {default as Person, data} from './person.js';  
export {default as a, b, c} from './my-js01.js';
```

```
<script type="module">  
  import {Person} from './tools';  
  
  const p = new Person('Bill', 32);  
  console.log('' + p);  
  
</script>
```







## \*\* 不刷頁面變換網址

- 1. 使用 # 和 `hashchange` 事件。
- `hash` 的彈性不佳，只能保留一個字串，資料沒有結構性。
- `hash` 的 SEO 效果很差。
- 2. 使用 `history.pushState()` 和 `popstate` 事件。



```
<a href="#abc">abc</a><br>
<a href="#def">def</a><br>
<a href="#123">123</a><br>
<div id="info"></div>
<script>
    window.addEventListener('hashchange', function () {
        document.querySelector('#info').innerHTML = location.hash;
    });
</script>
```



```
<button onclick="pushState()">click me</button>
<script>
  let count = 1;

  function pushState() {
    count++;
    history.pushState({count}, '', `/hello/${count}?id=${count}`);
  }

  window.addEventListener('popstate', function (event) {
    console.log(event.state);
    const {pathname, search} = location;
    console.log({pathname, search, state: history.state});
  });
</script>
```





## \*\* 正規表示法（補充資料）

- 正規表示法（**regular expression**）的目的是做文字的比對和尋找，在文字處理上非常重要，它是從善長文字處理的程式語言 **Perl** 上推廣而來。
- 現在，**JavaScript** 和其它許多程式語言也都支援正規表示法。**JavaScript** 裡使用的是 **RegExp** 物件。
- **RegExp** 物件可以搭配 **String** 物件的 **match**、**replace**、**search** 和 **split** 方法一起使用。
- **RegExp** 物件可以直接使用「/」包裹的方式定義。
- 練習場：<https://regex101.com/>

```
var re1 = /\sbe\s/i;  
var re2 = new RegExp('\\sbe\\s', 'i');
```



```
var str = "b be bEAch bead Beaker BEAN bee being abbey abet";  
var re = /\sbe/ig; // remove 'g' and try again  
console.log(str.search(re));  
console.log(str.match(re));  
console.log(str.replace(re, "**"));  
console.log(str.split(re));
```



# 單一字元表示法

表示法	說明	範例
\d	數字0~9	/\d\d/ 符合者為 '22' ； '2c' 則不符合
\D	「非」數字	/\D\D/ 符合者為 'ac' ； '2c'則不符合
\s	一個空白（space）	/a\sbar/ 符合者為 'a bar' ； 'abar' 則不符合
\S	「非」空白	/a\Sbar/ 符合者為 'a-bar' ； 'abar' 和 'a bar' 不符合
\w	字母、數字或底線（_）	/c\w/ 符合者為 'c7' ； 'c#' 和 'c-' 不符合
\W	「非」字母、數字或底線	/c\W/ 符合者為 'c%' ； 'ca' 和 'c_' 不符合
.	任何字元（不包含換行）	/a../ 符合者可為 'a12' 、 'ap+' 、 'a##'
[]	中括號中任一字元	/b[ae]d/ 符合者可為 'bad' 、 'bed'
[^]	不包含中括號中任一字元	/b[^ae]d/ 符合者可為 'b-d' 、 'bod' ； 'bad' 和 'bed' 不符合



# 多字元表示法

表示法	說明	範例
*	重複0次或多次	/lo*p/ 符合者可為 'lp'、'lop'、'loop'、'looop'
?	重複0次或1次	/lo?p/ 符合者為 'lp'、'lop'
+	重複1次或多次	/lo+p/ 符合者可為 'lop'、'loop'、'looop'
{n}	重複n次	/ba{2}d/ 符合者為 'baad'
{n, }	重複n次或以上	/ba{2,}d/ 符合者可為 'baad'、'baaad'
{n, m}	重複n次至m次之間	/ba{1,2}d/ 符合者為 'bad'、'baad'

- 上表裡的表示符號又稱為「貪婪計量子 (Greedy quantifiers)」，會儘量找尋較長的字串。例如表示式為「lo\*」，當搜尋的對象為 "looop" 時，搜尋到的會是 "looo"，而不是 "loo"、"lo" 或 "l"。
- 「貪婪計量子」後面接個「?」時，會變成「自閉計量子 (Reluctant quantifiers)」儘量找尋較短的字串。例如表示式為「lo+?」，當搜尋的對象為 "looop" 時，搜尋到的會是 "lo"，而不是 "loo" 或 "looo"。





## 位置及其它表示法

表示法	說明	範例
^	字首	<code>/^pos/</code> 符合者可為 'pose' ； 'apos' 不符合
\$	字尾	<code>/ring\$/</code> 符合者為 'spring' ； 'ringer' 不符合
	或	<code>/jpg png/</code>
()	子表示法	<code>/img\.(jpg png)/</code>



- **RegExp** 物件有兩個方法 **exec** 和 **test** 。
- **exec** 方法通常是用來搜尋字串中符合字模的子字串。
- **test** 方法是用來測試字串是否符合字模。

```
var str = "b bEAch bead Beaker";  
var re = /\sbe/ig;  
var obj;  
while ( obj = re.exec(str) ) {  
    console.log( obj );  
    console.log(re.lastIndex + ' -----');  
}
```





## \*\* 其他補充資料

### 表示時間點的物件 **Date**

```
var d = new Date();  
console.log( d );  
console.log( d.getFullYear() );  
console.log( d.getMonth() ); // from 0 to 11, 索引  
console.log( d.getDate() );  
console.log( d.getDay() );  
console.log( d.getHours() );  
console.log( d.getMinutes() );  
console.log( d.getSeconds() );  
console.log( d.getTime() ); // 1970年至今的毫秒數  
console.log( Date.now() ); // 1970年至今的毫秒數
```



## ■ 秒針

```
<style>
  .clock {
    position: relative;
    width: 600px;
    height: 600px;
    border-radius: 50%;
    background-color: lightcyan;
    border: 1px solid black;
  }
  .hand {
    position: absolute;
    left: 300px;
    top: 300px;
  }
  .hand-sec {
    position: absolute;
    width: 2px;
    height: 300px;
    left: -1px;
    top: -300px;
    background-color: red;
  }
</style>
```

```
<div class="clock">
  <div class="hand">
    <div class="hand-sec"></div>
  </div>
</div>
<script>
  const sec_hand = document.querySelector(".clock>.hand");
  //sec_hand.style.transform = "rotate(30deg)";
  const runClock = () => {
    const now = new Date();

    sec_hand.style.transform =
      `rotate(${now.getSeconds() * 6}deg)`;
    setTimeout(runClock, 1000);
  };
  runClock();
</script>
```



# 數學物件

Math的常用方法	說明
<code>abs(x)</code>	求絕對值。
<code>atan2(y, x)</code>	三角函數反正切（垂直距離和水平距離求角度）。
<code>ceil(x)</code>	大於等於 <b>x</b> 的最小整數。
<code>cos(x)</code>	三角函數餘弦。
<code>floor(x)</code>	小於等於 <b>x</b> 的最大整數。
<code>max(x, y, z, ..., n)</code>	最大值。
<code>min(x, y, z, ..., n)</code>	最小值。
<code>pow(x, y)</code>	<b>x</b> 的 <b>y</b> 次方。
<code>random()</code>	0 到 1 之間的亂數（大於等於 0, 小於 1）。
<code>round(x)</code>	四捨五入求整數。
<code>sin(x)</code>	三角函數正弦。



```
<div class="rect">隨機圓點</div>
<script>
  const rect = document.querySelector(".rect");

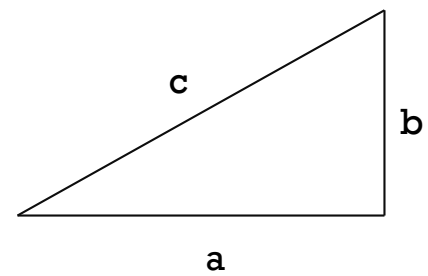
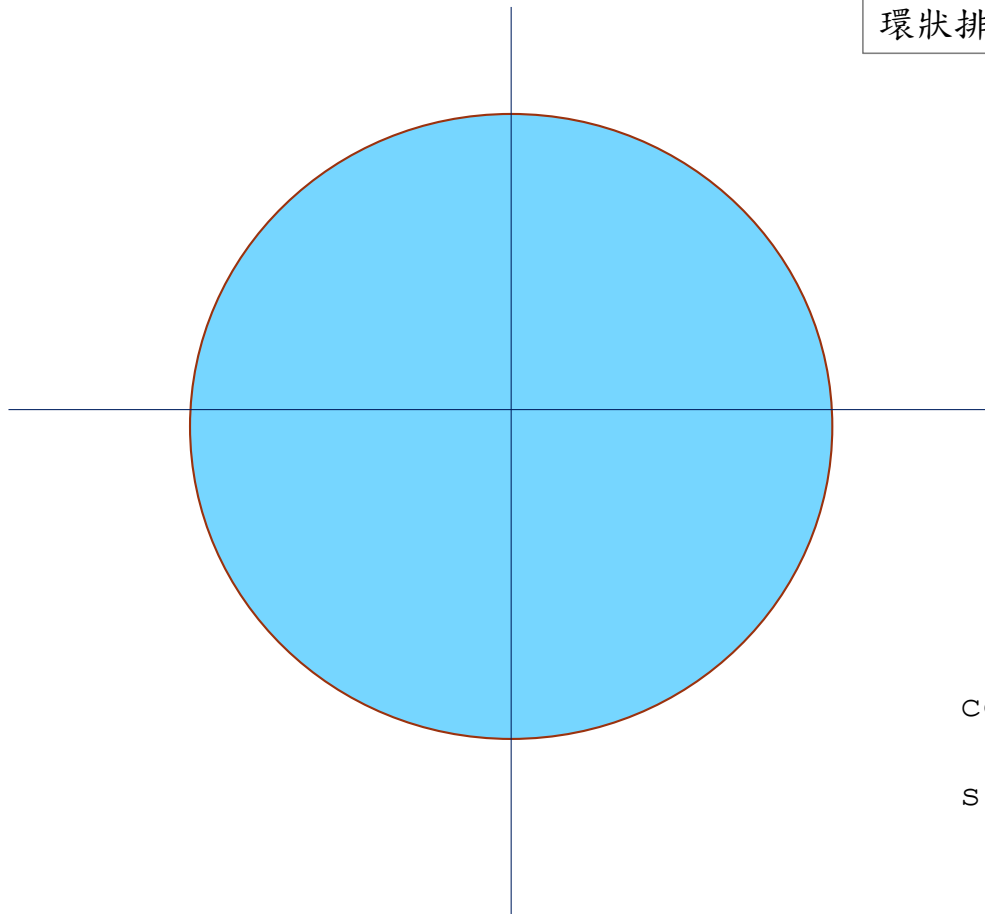
  for (let i = 0; i < 1000; i++) {
    const b = document.createElement("div");
    b.className = "ball";
    const size = 10 + Math.floor(Math.random() * 21);
    const x = Math.floor(Math.random() * 800);
    const y = Math.floor(Math.random() * 600);

    b.style.backgroundColor = `hsl(${bgc},100%,50%)`;
    b.style.left = x + "px";
    b.style.top = y + "px";
    b.style.height = b.style.width = size + "px";
    rect.appendChild(b);
  }
</script>
```

```
<style>
  .rect {
    position: relative;
    width: 800px;
    height: 600px;
    background-color: lightcyan;
    border: 1px solid black;
  }
  .ball {
    position: absolute;
    width: 20px;
    height: 20px;
    border-radius: 50%;
    background-color: red;
    text-align: center;
    border: 1px solid black;
  }
</style>
```



環狀排列



$$\cos \theta = ?$$

$$\sin \theta = ?$$





```
<div class="rect">環狀排列</div>
<script>
  const rect = document.querySelector(".rect");
  let b;
  const ballNum = 12;
  const angUnit = (Math.PI * 2) / ballNum;

  for (let i = 0; i < ballNum; i++) {
    b = document.createElement("div");
    b.className = "ball";
    b.innerHTML = i + 1;

    b.style.left =
      400 - 25 + Math.cos(i * angUnit - Math.PI / 3) * 260 + "px";
    b.style.top =
      300 - 25 + Math.sin(i * angUnit - Math.PI / 3) * 260 + "px";

    rect.appendChild(b);
  }
</script>
```







Thank you for listening

