JavaScript 進階

林新德

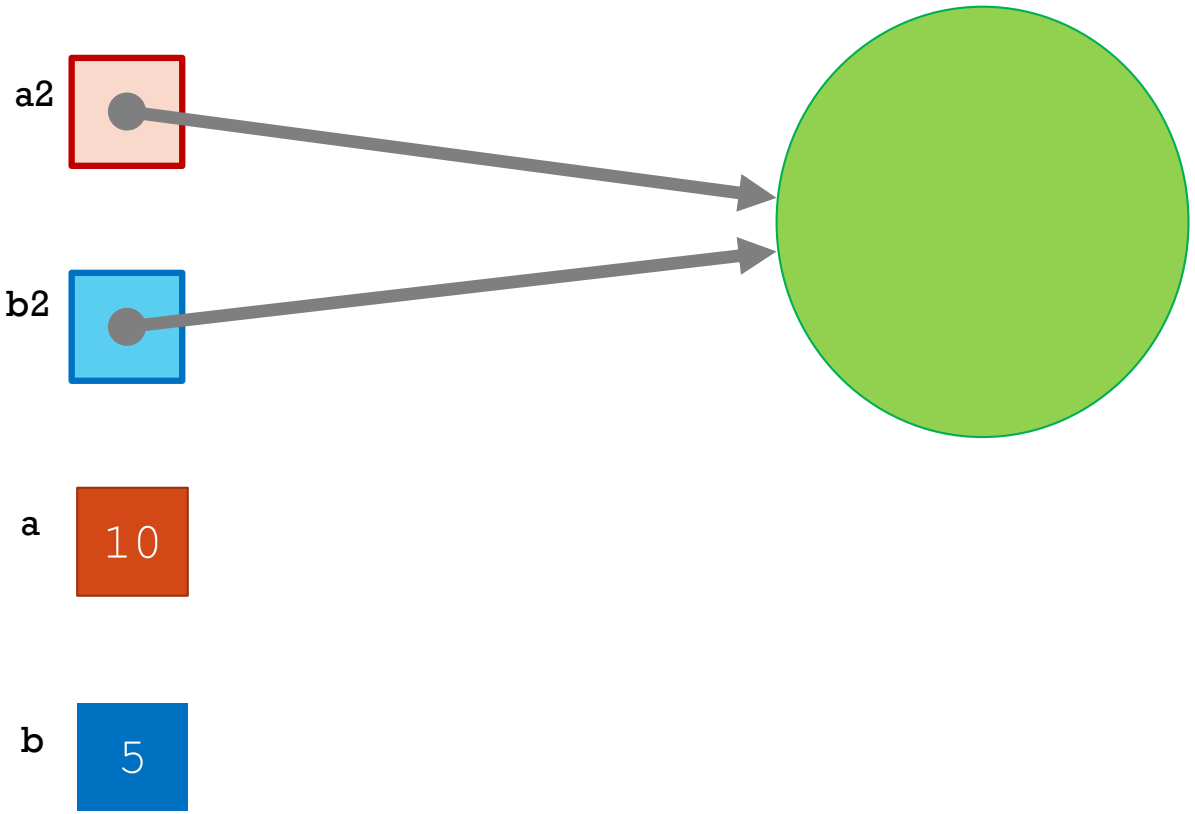
shinder.lin@gmail.com

參考專案 https://bitbucket.org/lsd0125/mfee45-js2

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



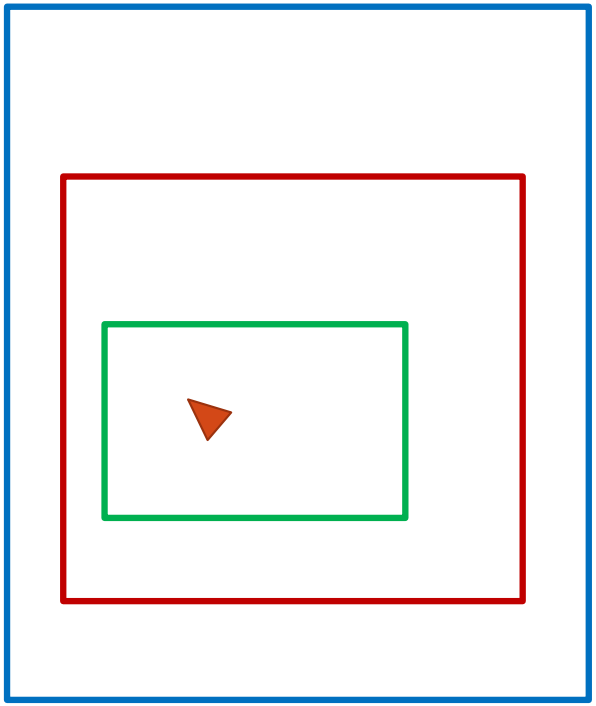
標籤的事件處理器	說明
onclick	單擊滑鼠左鍵。
ondblclick	雙擊滑鼠左鍵。
onmousedown	按下滑鼠左鍵時。
onmousemove	滑鼠在元素上移動時。
onmouseover	滑鼠移入元素時。
onmouseout	滑鼠移開元素時。
onmouseup	放開滑鼠左鍵時。
onkeydown	按下按鍵時。
onkeypress	按住按鍵時重複觸發。
onkeyup	放開按鍵時。
onload	內容載入後，使用於<body>。
onresize	文件大小改變時。
onscroll	文件捲動時。
onblur	失焦時，用於表單內的元素。
onchange	內容改變時，用於<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



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>
```

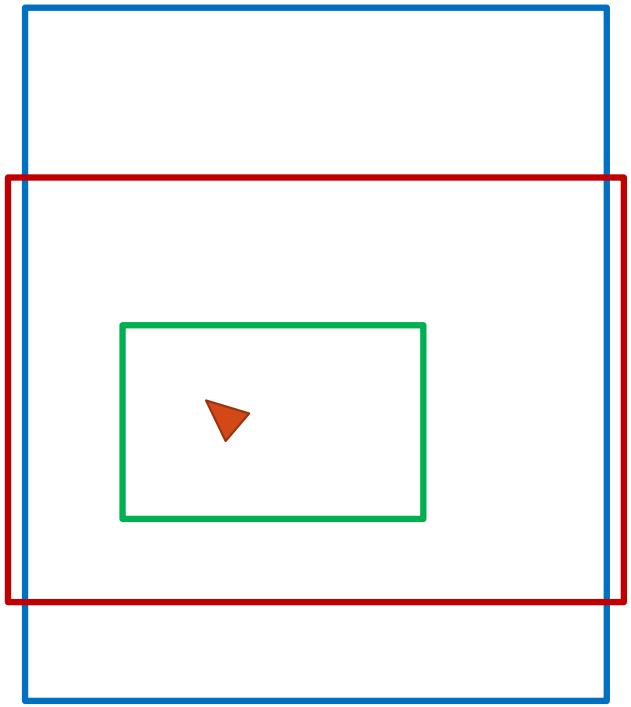
CAPTURING and AT TARGET

```
<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);  // useCapture
    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(); // 阻斷事件的傳遞
}, true); // useCapture
```

滑鼠事件座標



內容頁 `event.pageX` `event.pageY`

滑鼠在頁面的位置

顯示區 `event.clientX` `event.clientY`

滑鼠在顯示區的位置

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

滑鼠在 e.target 座標系裡的位置

```
<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 = `
        page:   ${event.pageX},   ${event.pageY},
        client: ${event.clientX}, ${event.clientY},
        offset: ${event.offsetX}, ${event.offsetY},
        `;
    });
</script>
```

DIV 塗鴉 1

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

  rect.addEventListener("mousemove", (e) => {
    const b = document.createElement("div");
    b.className = "ball";
    b.style.left = e.offsetX - 10 + "px";
    b.style.top = e.offsetY - 10 + "px";
    rect.append(b);
  });
```

```
<style>
  .rect {
    position: relative;
    width: 800px;
    height: 600px;
    background-color: #e7f5f7;
    border: 1px solid #cccccc;
  }
  .ball {
    position: absolute;
    width: 20px;
    height: 20px;
    border-radius: 50%;
    background-color: red;
    border: 1px solid black;
    /* 不要讓元素感應到滑鼠 */
    pointer-events: none;
  }
</style>
```

DIV 塗鴉 2

```
const rect = document.querySelector(".rect");
function getRandomRGB() {
  let s = Math.floor(16777216 * Math.random()).toString(16);
  s = s.padStart(6, "0");
  return "#" + s;
}

const mDown = (e) => rect.addEventListener("mousemove", mMove);
const mUp = (e) => rect.removeEventListener("mousemove", mMove);
const mMove = (e) => {
  const size = 10 + Math.floor(Math.random() * 21);
  const b = document.createElement("div");
  b.className = "ball";
  b.style.left = e.offsetX - size / 2 + "px";
  b.style.top = e.offsetY - size / 2 + "px";
  b.style.backgroundColor = getRandomRGB();
  b.style.width = size + "px";
  b.style.height = size + "px";
  rect.append(b);
};

rect.addEventListener("mousedown", mDown);
window.addEventListener("mouseup", mUp);
```

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: #fafffb;
  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>
```

鍵盤事件

```
<input type="text" id="input1" /><br />
<input type="text" id="input2" />
<script>
  const [input1, input2] = document.querySelectorAll("input");
  const handler = (e) => {
    const {key, keyCode, which, code, isComposing} = e;
    console.log(e.type, {key, keyCode, which, code, isComposing});
    // event.code: 明確的按鍵名稱
    // isComposing: 是否使用中文輸入法拼字中 (keyup 才有效)
  };

  input1.addEventListener("keydown", handler);
  input1.addEventListener("keypress", handler); // 有按出文字內容才會觸發
  input1.addEventListener("keyup", handler);
</script>
```

```
<div class="rect">
  <div class="ball" style="left: 0px; top: 0px"></div>
</div>
<script>
  const ball = document.querySelector(".ball");
  const handler = (e) => {
    e.preventDefault(); // 避免內容捲動的預設行為
    const x = parseInt(ball.style.left);
    const y = parseInt(ball.style.top);
    switch (e.code) {
      case "ArrowRight":
        ball.style.left = x + 10 + "px"; break;
      case "ArrowLeft":
        ball.style.left = x - 10 + "px"; break;
      case "ArrowUp":
        ball.style.top = y - 10 + "px";  break;
      case "ArrowDown":
        ball.style.top = y + 10 + "px";  break;
    }
  };

  window.addEventListener("keydown", handler);
</script>
```

鍵盤方向鍵控制 div 移動：例一

鍵盤方向鍵控制 div 移動：例二

```
const keyStates = {}; // 代表按鍵狀態
const ball = document.querySelector(".ball");
const downHandler = (e) => {
  e.preventDefault();
  console.log(e.code);
  keyStates[e.code] = true; // 表示開始按下按鍵

  let cx = parseInt(ball.style.left);
  let cy = parseInt(ball.style.top);

  if (keyStates.ArrowRight) cx += 10;
  if (keyStates.ArrowLeft)  cx -= 10;
  if (keyStates.ArrowUp)    cy -= 10;
  if (keyStates.ArrowDown)  cy += 10;

  ball.style.left = cx + "px";
  ball.style.top = cy + "px";
};
const upHandler = (e) => {
  e.preventDefault();
  delete keyStates[e.code]; // 表示放開按鍵
};
window.addEventListener("keydown", downHandler);
window.addEventListener("keyup", upHandler);
```

** 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

```
const s1 = Symbol("abc");
const obj1 = { name: "bill", age: 23 };

obj1[s1] = 345; // Symbol 可以當做 key, 相當於隱藏的屬性

console.log(obj1);
console.log(JSON.stringify(obj1));

for (let k in obj1) {
  console.log(k, obj1[k]);
}

// Object.getOwnPropertySymbols(obj1) // 查看物件裡有哪些 symbol 的屬性
```

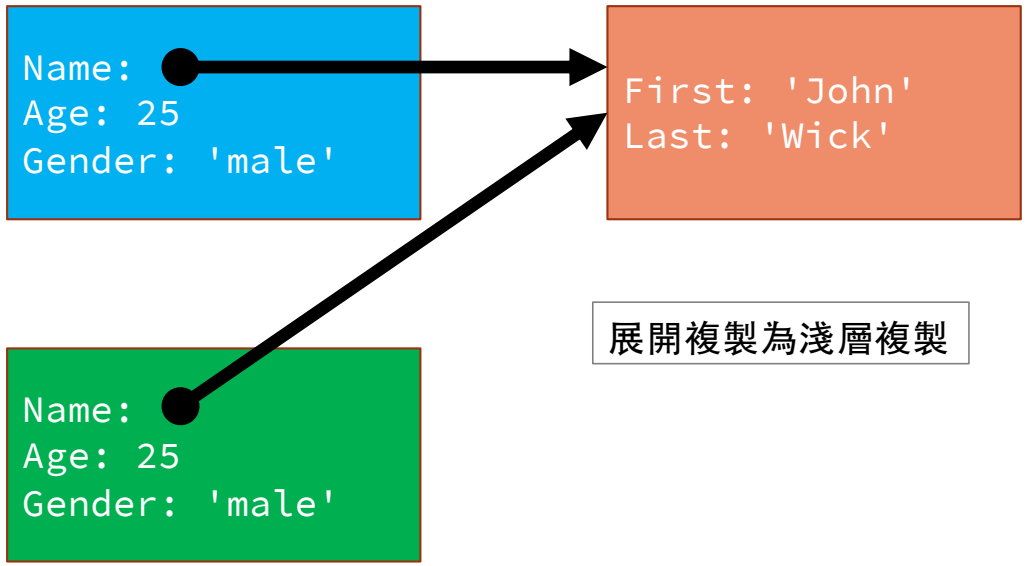
```
<div class="mydiv">123</div>
<div class="mydiv">456</div>
<script>
  const ar = [7, 9, 3, 5];
  const obj = {name: 'shinder'};
  const divs = document.querySelectorAll('.mydiv');

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

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

  console.log(divs[Symbol.iterator]);
  // ƒ 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 p1 = {
  name: {
    first: "John",
    last: "Wick",
  },
  age: 25,
  gender: "male",
};

const p2 = { ...p1 };   // 淺層複製

p1.age = 30;
p1.name.last = "Doe";


console.log({p1});
console.log({p2});
```

展開複製為淺層複製

- 深層複製 （舊的用法透過 JSON 轉換）

```
const p1 = {
  name: {
    first: "John",
    last: "Wick",
  },
  age: 25,
  gender: "male",
};
const p2 = JSON.parse(JSON.stringify(p1)); // 深層複製 (傳統的作法)
p1.age = 30;
p1.name.last = "Doe";

console.log({p1});
console.log({p2});
```

- 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" };
const p2 = { name: "Alex" };
const p3 = { name: "David" };
p1.next = p2;
p2.next = p3;
p3.next = p1;
try {
  const str = JSON.stringify(p1);
} catch (ex) {
  console.log(ex);
}
const q1 = structuredClone(p1);
p1.name = "Flora";
console.log(q1);
```

** 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));
}
```

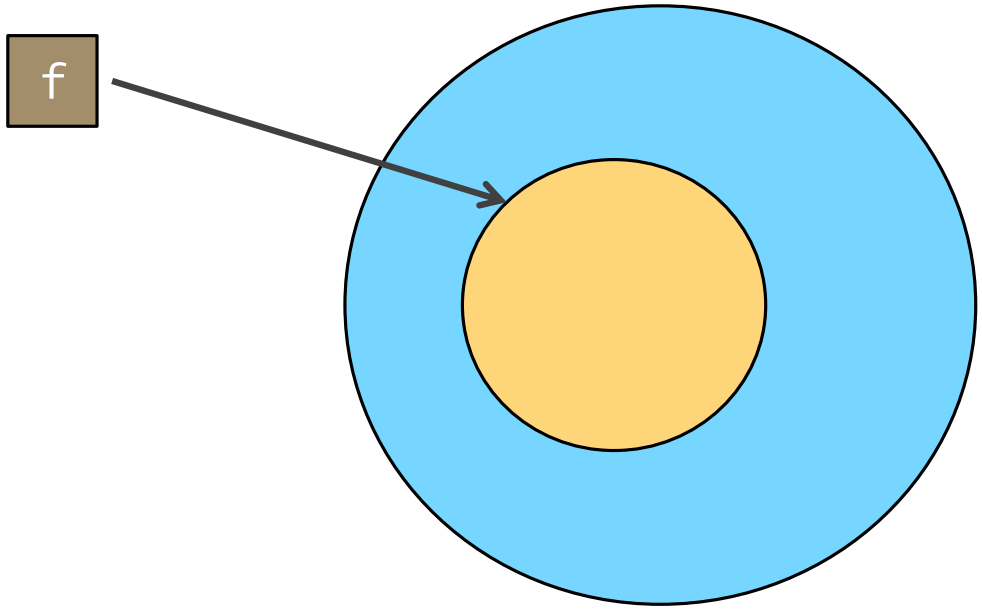
```
// 沒有使用快取
function fib(n) {
  if (n <= 0) return 0;
  if (n === 1) return 1;
  return fib(n - 1) + fib(n - 2);
}

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
```

```
<pre id="info"></pre>

<script>
  // 使用迴圈計算費氏數列
  const cache = [0, 1]; // 快取

  const beginTime = Date.now(); // 取得當下的 timestamp (毫秒)

  for (let n = 2; n <= 40; n++) {
    cache[n] = cache[n - 1] + cache[n - 2];
  }
  const endTime = Date.now(); // 取得當下的 timestamp (毫秒)
  info.innerHTML = cache.toString();
  info.innerHTML += "\n\n" + (endTime - beginTime); // 相差多少毫秒
</script>
```

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

```
const f = function () {
  let n = 3;

  return function (a) {
    n--;
    if (n >= 0) return a * a;
    return null;
  };
};

const f2 = f();
console.log(f2(7));
console.log(f2(7));
console.log(f2(7));
console.log(f2(7));

/*
// 錯誤的 closure 使用方式
console.log(f()(7));
console.log(f()(7));
console.log(f()(7));
console.log(f()(7));
*/
```

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;
};
```

```
function f1() {
  console.log("f1:", this);
}
const f2 = function () {
  console.log("f2:", this);
};
// 箭頭函式在定義時，就會綁定 this (context)
const f3 = () => {
  console.log("f3:", this);
};

const objA = {
  name: "objA", f1, f2, f3,
};
const objB = {
  name: "objB", f1, f2, f3,
};
```

```
f1();
f2();
f3();

objA.f1();
objA.f2();
objA.f3();
objB.f1();
objB.f2();
objB.f3();
```

```
function f1(x, y) {
  console.log("f1:", this);
  console.log({ x, y });
}
// 箭頭函式在定義時，就會綁定 this（context）
const f3 = (x, y) => {
  console.log("f3:", this);
  console.log({ x, y });
};

const objA = {
  name: "objA",
};

// 第一個參數表示，裡面的 this 要綁定誰
f1.call(objA, 1, 2);
f3.call(objA, 3, 4); // 箭頭函式使用 call() 是沒有作用的

f1.apply(objA, [1, 2]);
f3.apply(objA, [3, 4]);
```

```
// 綁定後取得新的函式
const f1_bound = f1.bind(objA);
f1_bound(10, 20);
const objB = {
  name: "objB",
  f1_bound,
};
objB.f1_bound(30, 40);
```

```
const obj = {
  name: "obj",
  myThis: this,
  f1: function () {
    return this;
  },
  f2() {
    return this;
  },
  f3: () => this,
  f4() {
    return () => this;
  },
};
console.log(obj.myThis);
console.log(obj.f1());
console.log(obj.f2());
console.log(obj.f3());
console.log(obj.f4()());
```

** 陣列的高階方法

```
const ar1 = [..."依商業登記資料當初設立以及第二年加入兩支新球隊"];
const ar2 = [12, 3, 67, 23, 9, 35];

ar1.sort(); // 中文排序是以字串（unicode字碼）為順序
ar2.sort(); // 預設看成字串排序

console.log(ar1);
console.log(ar2);

ar2.sort((a, b) => {
  console.log({ a, b });
  return a - b; // 排序規則, 負值才對調 (** 不要用布林值)
});
console.log(ar2);
```

- sort() 排序

- 亂數排序：

```
const ar1 = [];
for (let i = 1; i <= 42; i++) {
  ar1.push(i);
}
ar1.sort(() => Math.random() - 0.5);
console.log(ar1);
```

- 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

```
<script src="./data/products.js"></script>
<script>
  const tbody = document.querySelector("tbody");
  let str = "";
  products.forEach((element, index, array) => {
    str += `
    <tr>
      <td>${element.bookname}</td>
      <td>${element.price}</td>
    </tr>
    `;
  });
  tbody.innerHTML = str;
</script>
```

```
<table border="1">
  <thead>
    <tr>
      <td>書名</td>
      <td>價格</td>
    </tr>
  </thead>
  <tbody></tbody>
</table>
```

```
<script src="./data/products.js"></script>
<script>
  const tbody = document.querySelector("tbody");

  const itemTpl = ({ bookname, price }) => {
    return `
    <tr>
      <td>${bookname}</td>
      <td>${price}</td>
    </tr>
    `;
  };

  let str = "";
  products.forEach((el) => {
    str += itemTpl(el);
  });

  tbody.innerHTML = str;
</script>
```

```
const ar1 = [2, 3, 4, 5, 6, 7, 8, 9];

// filter() 篩選
const ar2 = ar1.filter((el, i) => {
  // 回傳 true 為你要挑選的項目
  return el % 2;
});
console.log({ ar1, ar2 });

// map() 以對應的格式輸出
const ar3 = ar1.map((el, i) => {
  return el * el;
});
console.log({ ar1, ar3 });

const ar4 = ar1.map((el, i) => {
  return `<b>${el}</b>`;
});
console.log({ ar1, ar4 });
```

- filter() 和 map() 的用法

- reduce() 的用法

```
const ar1 = [2, 3, 4, 5, 6, 7, 8, 9];

const t = ar1.reduce((previousValue, el, i) => {
  console.log({previousValue, el, i});
  return previousValue + el;
}, 0)

console.log({t});
```

```
<script src="./data/products.js"></script>
<script>
  const tbody = document.querySelector("tbody");
  const itemTpl = ({ bookname, price }) => {
    return `
    <tr>
      <td>${bookname}</td>
      <td>${price}</td>
    </tr>
    `;
  };

  tbody.innerHTML = products
    .filter((el) => {
      el.bookname += ' -- 不好看'; // 不要這樣做, 會變更到原本的資料
      return el.price >= 550;
    })
    .map((el, i) => {
      return itemTpl(el);
    })
    .join("");
</script>
```

- filter() 和 map() 的注意事項

- 練習一：在上頁範例中加入「搜尋欄位」，可搜尋商品名稱。
- 練習二：在上頁範例中加入選單（combobox），可選擇排序方式。
- 練習三：合併練習一和練習二的功能。

** 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

```
// 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');
  }, 200);
})
  .then(result => {
    console.log(result);
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        resolve('456');
      }, 300);
    });
  })
  .then(result => {
    console.log(result);
  });
```

```
function myPromise1(n) {
  return new Promise((resolve, reject) => {
    const ms = 200 + Math.floor(Math.random() * 500);
    setTimeout(() => {
      resolve({ n, ms });
    }, ms);
  });
}

myPromise1(2)
  .then((r) => {
    console.log(r);
    return myPromise1(3);
  })
  .then((r) => {
    console.log(r);
    return myPromise1(4);
  })
  .then((r) => {
    console.log(r);
  });
```

🐱 Promise 以 function 包裝使用

```
const myPromise2 = (n) => {
  const ms = 200 + Math.floor(Math.random() * 500);
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      return Math.random() > .5 ? resolve({n, ms}) : reject({n,
ms});
    }, ms);
  })
}

for (let i = 0; i < 10; i++) {
  myPromise2(i)
    .then(result => console.log(JSON.stringify({result})))
    .catch(ex => console.log(JSON.stringify({ex})));
}
```

** 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）

使用 Promise 靜態方法

```
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);
  }).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);
  }).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);
  }).catch(ex => {
    // 只要有一個 promise 駁回，就會進入這裡
    console.log(ex)
  })
```

```
const promises = [];

for (i = 0; i < 10; i++) {
  promises.push(myPromise1(i + 1));
  // promises.push(myPromise1(i + 1).then((r) => console.log(r)));
  /*
  promises.push(myPromise1(i + 1).then((r) =>{
    console.log(r);
    return r.a;
  }));
  */
}

Promise.all(promises).then((result) => {
  console.log(result);
});
```

** 注意：Promise 物件後若有接 then()，則結果會是 then() 裡 callback 的回傳值。

```
function getBase64Img(file) {
  if (!(file instanceof File)) {
    return null;
  }
  return new Promise((resolve, reject) => {
    const reader = new FileReader();
    reader.onload = function () {
      resolve(reader.result);
    };
    reader.readAsDataURL(file);
  });
}
```

* Promise 應用

```
<form onsubmit="return false">
  <input type="file" onchange="inputChange()" id="inp" />
</form>
<img src="" alt="" id="myimg" />
<script>
  const inp = document.querySelector("#inp");
  const myimg = document.querySelector("#myimg");
  async function inputChange() {
    const f = inp.files[0];
    myimg.src = await getBase64Img(f);
  }
</script>
```

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 敘述結構。

* async 修飾的函式回傳值為 Promise 物件。

```
async function a(){
  const r1 = await myPromise1(3);
  console.log(r1);
  const r2 = await myPromise1(5);
  console.log(r2);
}
console.log('A:', new Date());
a().then(()=>{
  console.log('B:', new Date());
});
console.log('C:', new Date());
```

* 不同 async 修飾的函式為不同的 Promise 物件，各自執行。

```
(async () => {
  for (let i = 1; i <= 10; i++) {
    const r1 = await myPromise1('x'+i);
    console.log(r1);
  }
})();

(async () => {
  for (let i = 1; i <= 10; i++) {
    const r1 = await myPromise1('y'+i);
    console.log(r1);
  }
})();
```

** 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）

```
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	/^pos/ 符合者可為 'pose' ; 'apos' 不符合
$	字尾	/ring$/ 符合者為 'spring' ; 'ringer' 不符合
|	或	/jpg|png/
()	子表示法	/img\.(jpg|png)/

- 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。

```
const str = "b bEAch bead Beaker";
const re = /\sbe/ig;
let 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

```
const 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>
```

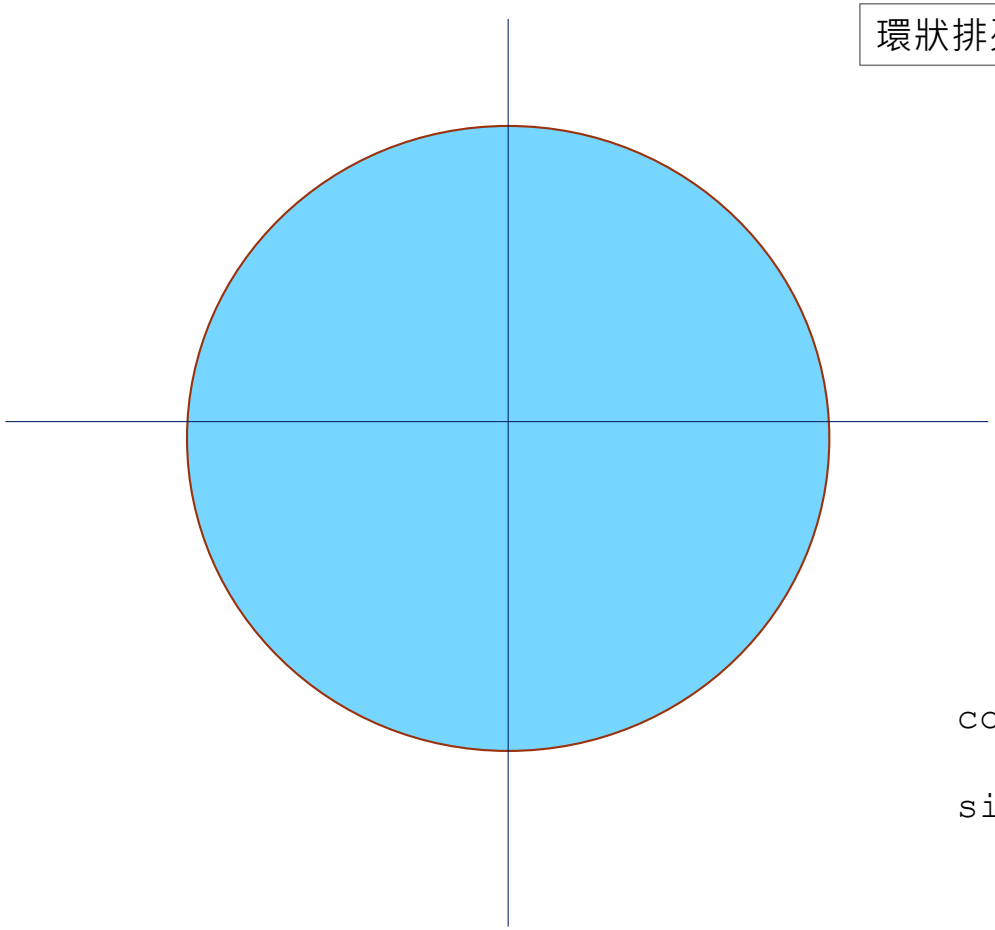
數學物件

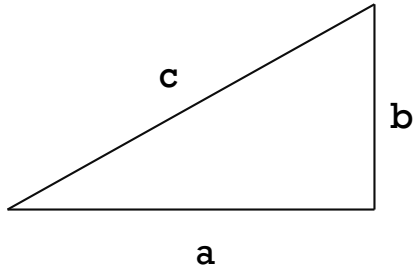
Math的常用方法	說明
abs(x)	求絕對值。
atan2(y, x)	三角函數反正切（垂直距離和水平距離求角度）。
ceil(x)	大於等於 x 的最小整數。
cos(x)	三角函數餘弦。
floor(x)	小於等於 x 的最大整數。
max(x, y, z, ..., n)	最大值。
min(x, y, z, ..., n)	最小值。
pow(x, y)	x 的 y 次方。
random()	0 到 1 之間的亂數（大於等於 0, 小於 1）。
round(x)	四捨五入求整數。
sin(x)	三角函數正弦。

```
<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 θ = ?

sin θ = ?

```
<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