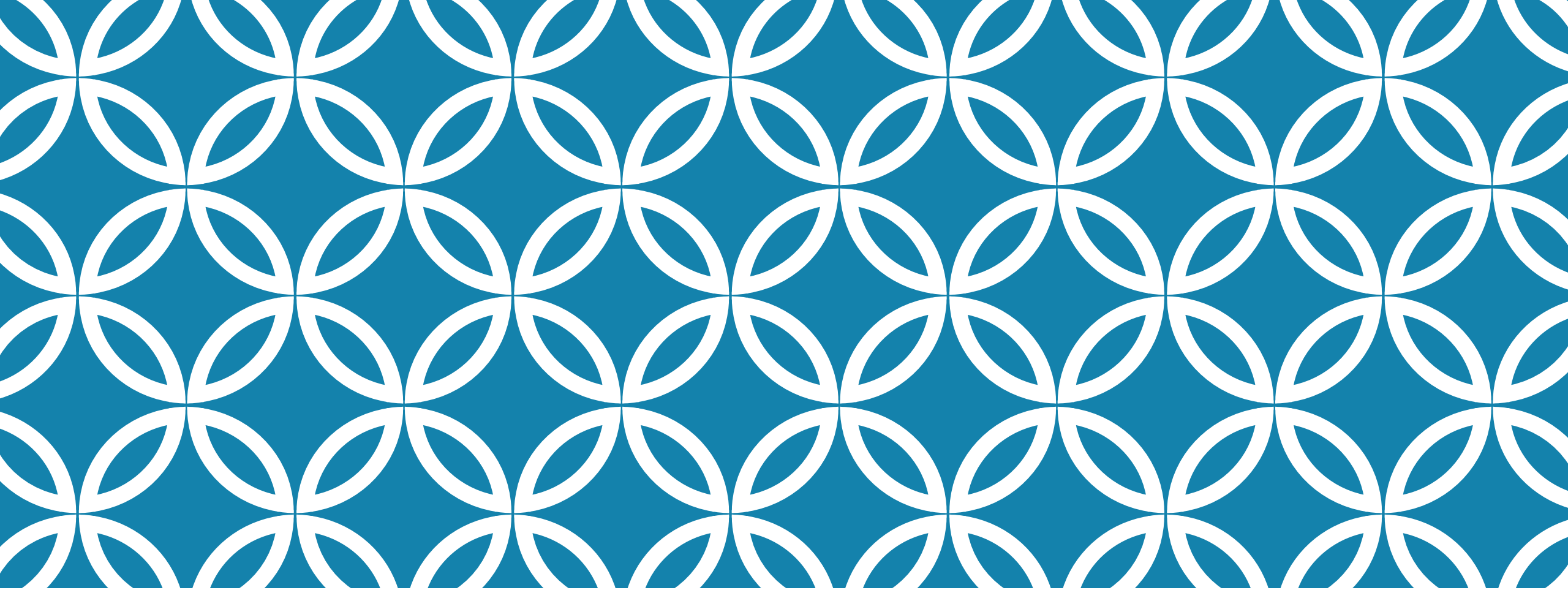


FUNCTION

何謂函數

將一堆程式碼合在一起並且給個名字，使用時呼叫該名字就執行那一堆程式碼

無回傳值的函數

```
function hello() {
    console.log('Hello, World!')
}
```

有回傳值的函數

```
function hello() {
    return 'Hello, World!'
}
```

參數

讓呼叫函數時可以傳一些資料進去，讓函數可以根據傳進去的值做一些判斷或運算

```
function hello(name) {
    return `Hello, ${name}`
}
console.log(hello('David'))
```

丟出錯誤

函數執行過程可以透過 throw 丟出錯誤，然後終止函數執行

```
function div(a, b) {
  if (b == 0) {
    throw '分母不可為零'
  }
  return a + b
}

try {
  let result = div(5, 0)
  console.log(result)
} catch (e) {
  console.log('ERROR: ' + e)
}
```

參數預設值

當參數有預設值時，呼叫時該參數就可以省略不輸入

```
function hello(name='Anonymous') {
  return `Hello, ${name}`
}

console.log(hello())
console.log(hello('David'))
```

函數中的函數

當函數中需要一個僅需在內部使用的函數時，將該函數定義在內部即可

```
function arithmetic(a, op, b) {
  function add(a, b) {
    return a + b
  }
  switch (op) {
    case '+':
      return add(a, b)
    default:
      return undefined
  }
}
console.log(arithmetic(5, '+', 3))
console.log(arithmetic(5, '*', 2))
```

提前返回

有效範圍 / 生命週期

下列三種狀況，輸出內容為何？

```
let a = 10
function test() {
  a = 20
}
test()
console.log(a)
```

```
let a = 10
function test() {
  let a = 20
}
test()
console.log(a)
```

```
let a = 10
function test(a) {
  a = 20
}
test(a)
console.log(a)
```

全域與區域重疊時，區域優先

修改的是複製品還是本尊？

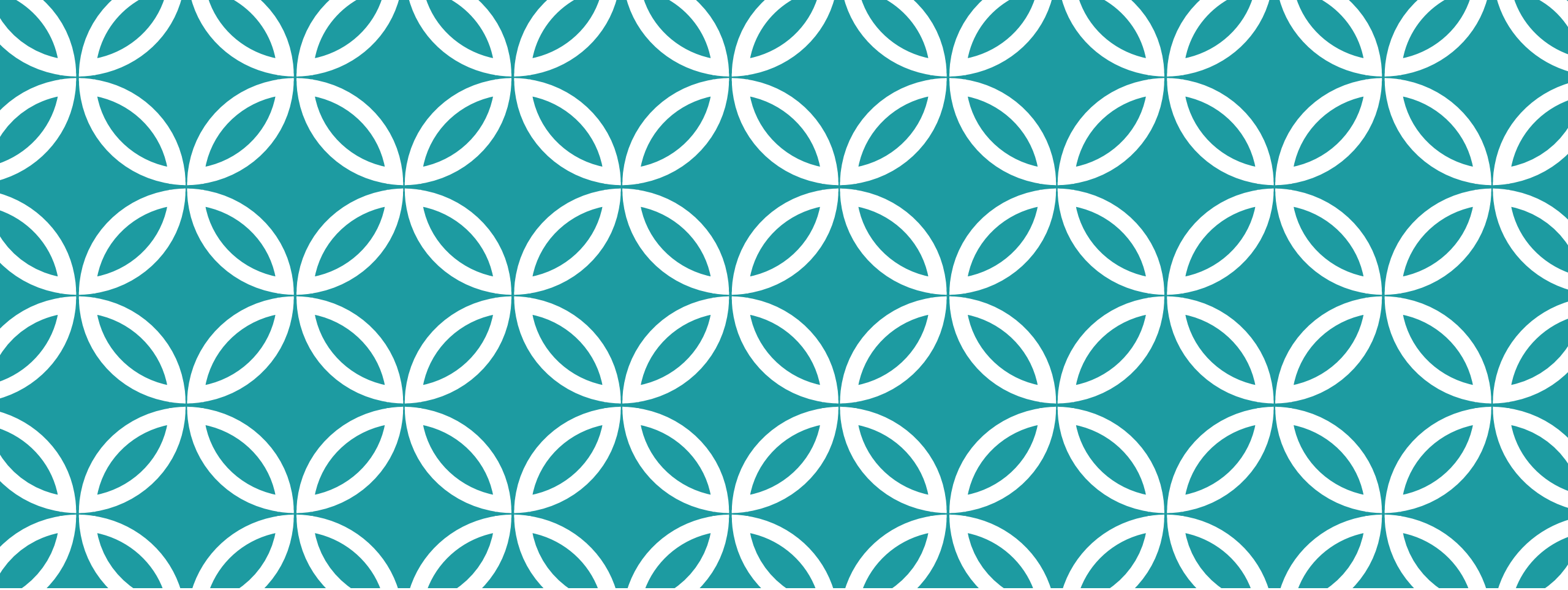
複製品（call-by-value）

```
function test(a) {
  a = 20
}

let a = 10
test(a)
console.log(a) // Prints 10
```

本尊（call-by-address / sharing）

```
function test(a) {
  a[0] = 20
}

let a = [10]
test(a)
console.log(a)  // Prints [20]
```

修改物件內容都是改到本尊



CLOSURE

何謂 CLOSURE

Closure 又稱為匿名函數，意思是該函數沒有名字。有名字的稱為具名函數。

具名函數

```
function add(a, b) {
    return a + b
}
```

匿名函數。若使用 typeof() 檢查 add 資料型態，會得到 function

```
const add = function(a, b) {
    return a + b
}
let n = add(5, 3)
```

參數的型態為FUNCTION

除了變數或常數的內容可以是函數外，函數的參數也可以是函數

```
function add(a, b, complete) {
    complete(a + b)
}
```

呼叫

```
add(5, 3, function(result) {
    console.log(result)
})
```

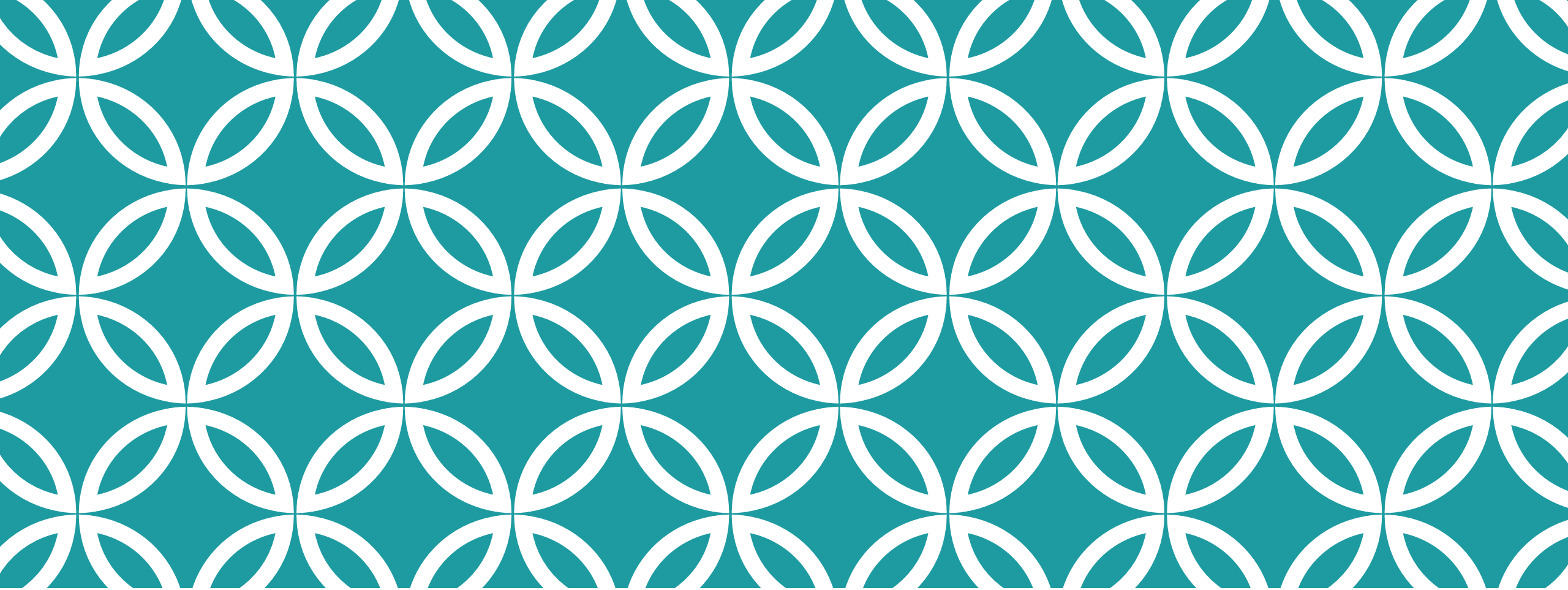
ARROW FUNCTION

箭頭函數

```
const x10 = (number) => {
   return number * 10
}
```

一般函數

```
const x10 = function(number) {
   return number * 10
}
```



非同步函數

非同步呼叫

延遲兩秒執行

```
function test() {
  console.log('Hello, World')
}

setTimeout(() => {
  test()
}, 2000)

console.log('done')
```

單位：毫秒

非同步處理

使用 closure 語法，可以讓執行比較久的函數，改採非同步方式傳回資料

```
function add(a, b, complete) {
    setTimeout(function() {
        complete(a + b)
    }, 0);
}

add(5, 3, function(result) {
    console.log(result)
})

console.log('done')
```

看看先印出結果還是先印出 done

練習

將下列 add() 函數改為 async 的非同步語法

```
function add(a, b, complete) {
    setTimeout(function() {
        complete(a + b)
    }, 0);
}
```

ASYNC

使用 setTime() 函數太麻煩，改使用 async 函數

async 會讓回傳值變成 Promise 型態

```
async function test() {
  return 'Hello, World';
}

test()
.then(result => {
  console.log(result)
})

console.log('done')
```

何謂 PROMISE

Promise 是一種資料型態，以非同步方式回傳資料

透過連續的 then() 函數分層處理資料

錯誤透過 catch() 函數捕獲，並取得錯誤理由

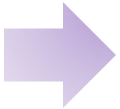
```
const div = (a, b) => {
  return new Promise((resolve, reject) => {
    if (b == 0) {
      reject('分母為零，無法運算')
    } else {
      resolve(a / b)
    }
  })
}

div(5, 3)
.then(result => parseInt(result))
.then(value => console.log(value))
.catch(reason => console.log('ERROR: ' + reason))

console.log('done')
```

這裡省略了 { reutrn … }

ASYNC 與 PROMISE

兩種寫法一樣，加上 async 的函數，JavaScript 會自動轉成回傳 Promise

```
async function div(a, b) {
  if (b == 0) {
    throw '分母不可為零'
  }
  return a / b
}
```



```
function div(a, b) {
  return new Promise((resolve, reject) => {
    if (b == 0) {
      reject('分母不可為零')
    } else {
      resolve(a / b)
    }
  })
}
```

AWAIT

await 用來等待非同步呼叫完成，相當於把非同步改為同步

await 一定要在 async 函數中才能使用

```
async function add(a, b) {
  return a + b
}

async function main() {
  let response = await add(5, 3)
  console.log(response)
}

main()
```

不使用 AWAIT

若有三個非同步函數，卻需要以同步方式呼叫時，程式碼非常難閱讀

```
async function getNumber() {
  return 10
}
async function tenTimes(number) {
  return number * 10
}
async function square(number) {
  return number * number
}
```

```
bn.onclick = () => {
  getNumber()
  .then(result => {
    tenTimes(result)
    .then(result => {
      square(result)
      .then(result => {
        console.log(result)
      })
    })
  })
}
```

使用 AWAIT

await 需在 async 函數中才能使用

程式碼非常容易閱讀

別忘記要加

```
bn.onclick = async () => {
  let result = await getNumber()
  result = await tenTimes(result)
  result = await square(result)
  console.log(result)
  console.log('done')
}
```

FETCH

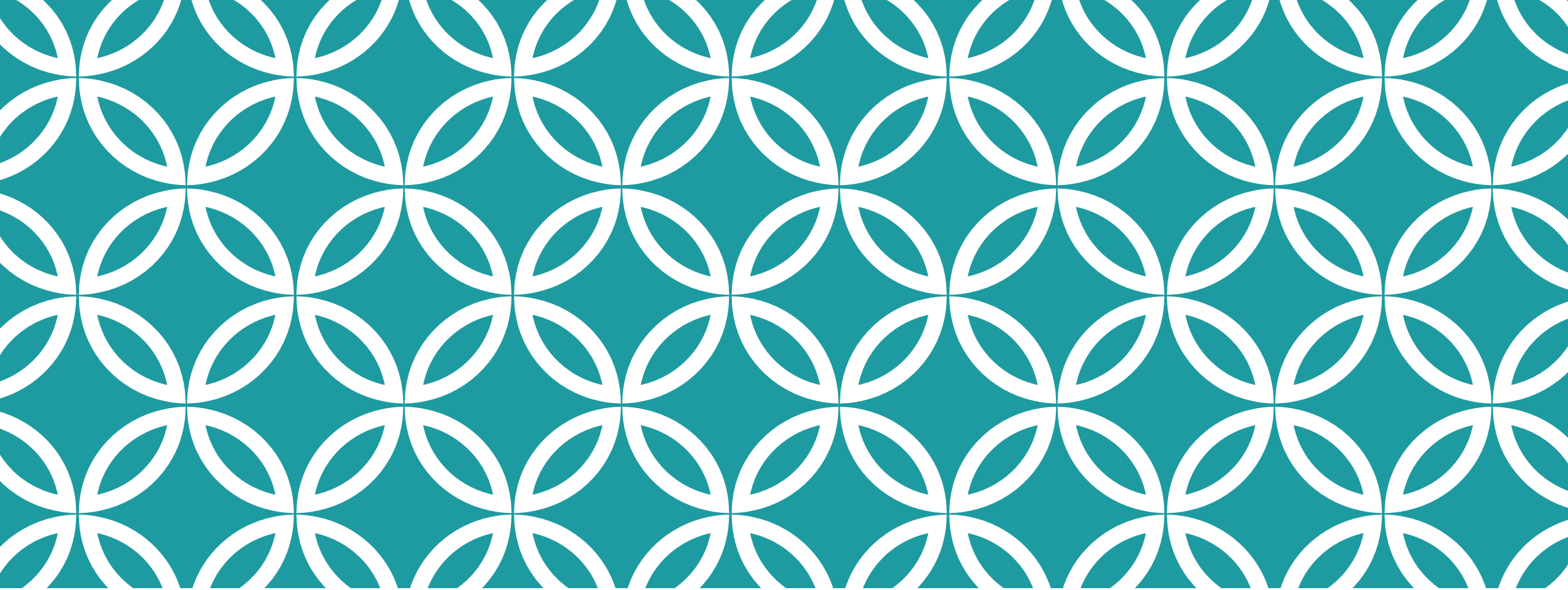
fetch() 函數用來非同步載入資料，資料來源可為檔案或網址

要注意網址不可為 file:// 或是出現 CORS （同源政策）問題

在不關閉瀏覽器的 CORS 功能外，可將資料傳至 GitHub，GitHub 的 raw 頁面允許非同源呼叫

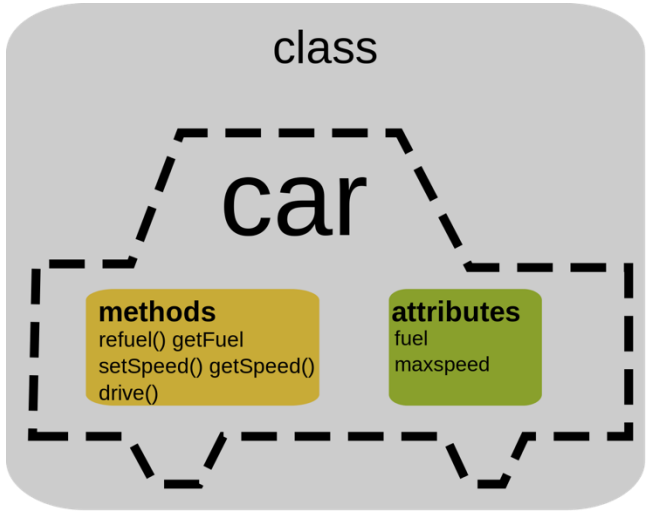
```
fetch('elephant.jpg')
.then(response => {
    return response.blob()
})
.then(async blob => {
    // const url = URL.createObjectURL(blob)
    // image.src = url

    const byteArray = new Uint8Array(await blob.arrayBuffer()) // [225, 216, ...]
    const base64 = btoa(String.fromCharCode(...byteArray))
    image.src = `data:${blob.type};base64,${base64}`
})
```

image/jpeg



物件導向

WHY 物件導向

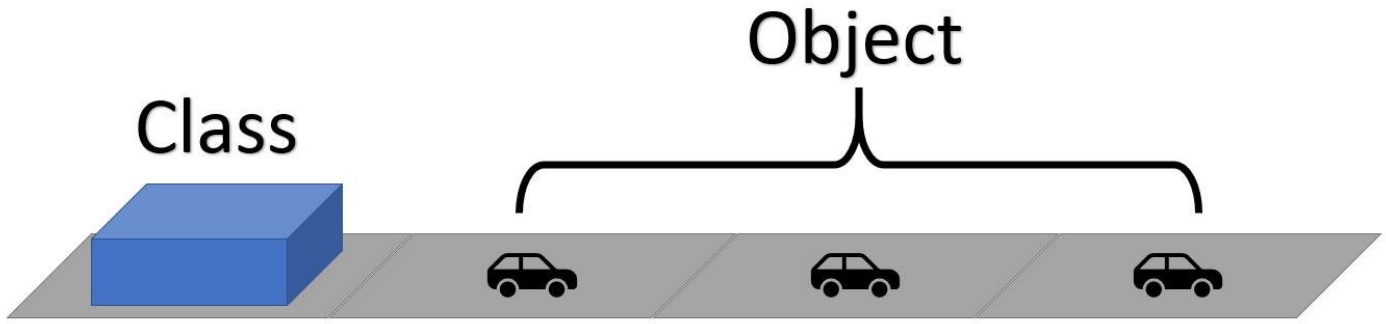


Reuse and duplicate（程式碼重複利用）

程式碼很容易與具體事物或觀念對映

結構與架構清楚且擴充性高，易於發展大型複雜系統

缺點：有些簡單的事情複雜化



物件導向四大特性

封裝
- 將屬於該物件特有的「東西」封裝在該物件中，包含特徵與操作方法
- 特徵即屬性，以變數實踐，操作方法以函數實踐

繼承
- 父類別有的「東西」會繼承到子類別
- 可透過存取控制來決定父類別哪些東西可以繼承，哪些不行

抽象
- 只定義物件的特徵與操作方法，但實行細節並未說明，由子類別去實做
- 抽象類別不可以實體化

多型
- 同一個物件，經由型別轉換後呈現不同的樣貌，呼叫不同的操作方法

函數即是物件

當物件用時建議大寫，
有些前端框架強制大寫

```
function Person(name, age) {
  this.name = name
  this.age = age
}

var david = new Person('David', 30)
david.hairColor = 'black'

console.log(david.name)
console.log(david.age)
console.log(david.hairColor)
```

動態新增
hariColor屬性

加入方法

在物件中加上操作方法（method）

```
function Person(firstName, lastName, age) {
  this.firstName = firstName
  this.lastName = lastName
  this.age = age
  this.name = () => `${firstName} ${lastName}`
}

let david = new Person('David', 'Wang', 30)
console.log(david.name())
```

繼承

```
function Person(firstName, lastName, age) {
  this.firstName = firstName
  this.lastName = lastName
  this.age = age
  this.name = () => `${firstName} ${lastName}`
}

function Superman(firstName, lastName, age) {
    Person.call(this, firstName, lastName, age)
    this.flyTo = (city) => {
      this.at = city
    }
}

const superman = new Superman('Clark', 'Kent', 20)
superman.flyTo('New York')
console.log(`${superman.name()} is at ${superman.at}`)
```

物件的 PROTOTYPE

Prototype 用來擴充物件原有的功能

不可使用箭頭函數

儲存到另外一個檔案，例如 lib.js

```
Array.prototype.shuffle = function() {
  let tmpArray = this.slice()
  for(let i = 0; i < tmpArray.length * 5; i++) {
    let m = parseInt(Math.random() * tmpArray.length)
    let n = parseInt(Math.random() * tmpArray.length)
    let tmp = tmpArray[m]
    tmpArray[m] = tmpArray[n]
    tmpArray[n] = tmp
  }
  return tmpArray
}
let arr = [1, 2, 3, 4, 5, 6].shuffle()
console.log(arr)
```

CLASS

```
class Person {
  constructor(firstName, lastName, age) {
    this.firstName = firstName
    this.lastName = lastName
    this.age = age
  }

  name() {
    return `${this.firstName} ${this.lastName}`
  }
}

let david = new Person("David", "Wang", 30)
let mei = new Person("Mei", "Chen", 27)
```

不要加 function

繼承

```
class Person {
  constructor(firstName, lastName, age) {
    this.firstName = firstName
    this.lastName = lastName
    this.age = age
  }

  name() {
    return `${this.firstName} ${this.lastName}`
  }
}

class Superman extends Person {
  flyTo(city) {
    this.at = city
  }
}
```

存取等級

預設：public

加上 # 為 private

```
class Superman extends Person {
 #job = 'Reporter'

 #girlFriend() {
  return 'Lois'
 }

 flyTo(city) {
  this.at = city
 }
}
```

類別屬性與類別方法

在屬性與方法前加上 static 時，就整個 class 只有一份

不需產生實體即可呼叫使用

```
class Superman extends Person {
  static homePlanet = 'Krypton'
}

console.log(Superman.homePlanet)
```

單例 SINGLETON

單例為設計模式中的一種模式，用於只需要產生一個實體的時機

例如：假設有一個播放音樂的類別，內部需要一個播放清單記錄所有需要播放的音樂

```
class Music {
 constructor(musicName) {
  this.list = (musicName === undefined) ? [] : [musicName]
 }
 listAdd(musicName) {
  this.list = [...this.list, musicName]
 }
 getList() {
  return this.list
 }
}
```

設計播放器

由於音樂播放裝置只有一個，又不希望混音播放，因此將 play() 設計成 static

```
class Music {
  static play() {
    this.list.forEach(musicName => {
      console.log('PLAYING: ' + musicName)
    })
  }
}
```

static 方法（正式名稱為 class 方法）無法呼叫 instance 方法

```
let music = new Music()
music.listAdd('aaa')
music.listAdd('bbb')
Music.play() // ❌
```

使用單例產生一個實體

常見的單例屬性

- shared、getInstance、default、standard

```
class Music {
  static shared = new Music()
  play() {
    this.list.forEach(musicName => {
      console.log('PLAYING: ' + musicName)
    })
  }
}

let music = Music.shared // 單例
music.listAdd('aaa')
music.listAdd('bbb')
music.play()
```