chapter 2

function

بإعداد : سفيان يوسف الفازع .

* function هي في الواقع objects في java script .

* السمة المميزة للدالة (function) ما يميزها عن أي كائن (object) آخر هي وجود خاصية داخلية تسمى [call] .

* لا يمكن الوصول إلى الخصائص الداخلية عبر التعليمات البرمجية , بل تحدد سلوك الكود أثناء تنفيذه .

* يحدد (ECMAScript) الخصائص الداخلية المتعددة للكائنات في java script , وتتم الإشارة إلى هذه الخصائص الداخلية من خلال القوسين المربعين [] .

* خاصية [call] فريدة للدالة [function] .

* يناقش هذا الفصل الطرق المختلفة التي يتم بها تعريف الوظائف وتنفيذها في Javascript .

*Declaration VS Expressions

* يوجد نوعان من الدوال الحرفية :

* النوع الأول ---> إعلان الوظيفة function declaration :

يبدأ بالكلمة المفتاحية للوظيفة function وبعدها اسم الوظيفة مباشرة , محتويات الوظيفة محاطة بأقواس , كما موضح في التعبير :

```
function add(num1, num2) {

    return num1 + num2;
}
```

* النوع الثاني ---> التعبير الوظيفي function expression :

لا يتطلب اسماً بعد الوظيفة function .

تعتبر هذه الوظائف مجهولة لأن function object نفسه ليس له اسم , يشار عادة إليها عبر متغير أو خاصية , كما في التعبير :

يعين قيمة دالة للمتغير add .

```
var add = function(num1, num2) {
    return num1 + num2;
};
```

تنتهي expression بفاصلة منقوطة .

* يتطابق Expression مع Declaration باستثناء الإسم المفقود و الفاصلة المنقوطة في النهاية .
* على الرغم من أن الشكلين متشابهان , إلا أنهما يختلفان بطريقة مهمة جداً .
* يتم رفع الإعلان للوظيفة (function declarations) إلى أعلى , عند تنفيذ الكود .
* هذا يعني أنه يمكنك تحديد وظيفة (function) بعد استخدامها في الكود دون حدوث خطأ.
* على سبيل المثال :

```
var result = add(5, 5);
function add(num1, num2) {
 return num1 + num2;
}
```

قد يبدو هذا الكود أنه يتسبب في خطأ , لكنه يعمل بشكل جيد . لأن محرك جافا سكريبت يرفع الإعلان إلى الأعلى وينفذ الكود كما لو كان مكتوب بهذا الشكل .

```
// how the JavaScript engine interprets the code

function add(num1, num2) {
  return num1 + num2;
}
var result = add(5, 5);
```

* تحدث عملية الرفع فقط في طريقة declaration لأن اسم الوظيفة معروف مسبقاً .

ولكن لا يحدث الرفع في طريقة expression لأنه لا يمكن الإشارة إلى الوظائف (function) إلا من خلال متغير .

مثال هذا الكود سيسبب خطأ في التنفيذ :

```
// error!

var result = add(5, 5);
var add = function(num1, num2) {
 return num1 + num2;
 };
```

* لو أنك دائماً تقوم بتعريف الوظائف قبل استخدامها , يمكنك استخدام إحدى الطريقتين (declaration or expression) :

* Function as values :

* Java script يحتوي على وظائف من الدرجة الأولى , يمكنك استخدامها كما تفعل مع أي object أخرى .
* يمكنك تخصيصها للمتغيرات وإضافتها إلى الكائنات وتمريرها إلى وظائف أخرى وإعادتها من الوظائف .
* هذا يجعل وظائف java script قوية بشكل لايصدق .

```
1 - function sayHi() {
  console.log("Hi!");
}
sayHi();       // outputs "Hi!"
2 - var sayHi2 = sayHi;


sayHi2();      // outputs "Hi!"
```

1 - في هذا الكود , يوجد تصريح لدالة اسمها sayHi .

2 - ثم يتم إنشاء متغير اسمه SayHi2 وتعيين قيمته ل SayHi .

3 - يشير كل من SayHi , SayHi2 الآن إلى نفس الوظيفة , لذلك نتيجة التنفيذ نفسها للاثنين .

* نفس الكود السابق معاد باستخدام function constructor :

```
var sayHi = new Function("console.log(\"Hi!\");");

sayHi();        // outputs "Hi!"

var sayHi2 = sayHi;

sayHi2();       // outputs "Hi!"
```

constructor يجعل الأمر أكثر وضوحاً أن sayHi كإنه كائن object ويمكن تمريره مثل أي object آخر .

* دائماً ضع في تفكيرك أن (functions are objects) ,, ليكون السلوك behavior له معنى مفهوم .

على سبيل المثال ، يمكنك تمرير دالة إلى دالة أخرى كوسيطة argument.

مثال على تمرير الدوال كطريقة sort () .....

* تقبل طريقة sort () الموجودة في مصفوفات JavaScript دالة مقارنة كمعامل اختياري .

يتم استدعاء دالة المقارنة عندما يجب مقارنة قيمتين في المصفوفة.

> إذا كانت القيمة الأولى أصغر من الثانية ، يجب أن تُرجع دالة المقارنة رقمًا سالبًا.

> إذا كانت القيمة الأولى أكبر من الثانية ، يجب أن تُرجع الدالة رقمًا موجبًا.

> إذا كانت القيمتان متساويتان ، يجب أن ترجع الدالة صفرًا.

بشكل افتراضي ، يحول sort () كل عنصر في مصفوفة إلى سلسلة string ثم يجري مقارنة.

هذا يعني أنه لا يمكنك فرز مصفوفة من الأرقام بدقة دون تحديد دالة مقارنة.

على سبيل المثال ، تحتاج إلى تضمين دالة مقارنة لفرز مجموعة من الأرقام بدقة ، مثل:

```
var numbers = [ 1, 5, 8, 4, 7, 10, 2, 6 ];
1- numbers.sort(function(first, second) {
      return first - second;
});
console.log(numbers);      // "[1, 2, 4, 5, 6, 7, 8, 10]"

2- numbers.sort();
console.log(numbers);      // "[1, 10, 2, 4, 5, 6, 7, 8]"
```

في هذا المثال ، function 1 للمقارنة التي تم تمريرها إلى sort () هي في الواقع function expression.
لاحظ أنه لا يوجد اسم للدالة ؛

يؤدي طرح القيمتين إلى إرجاع النتيجة الصحيحة من دالة المقارنة, كما هو موضح الترتيب الصحيح.

2 - sort () ، لا يستخدم دالة مقارنة.
يختلف ترتيب المصفوفة عن المتوقع ، حيث أن الرقم 1 يتبعه 10. وذلك لأن المقارنة الافتراضية لدالة sort() تحول جميع القيم إلى (سلاسل string) قبل مقارنتها.

Parameters

* جانب آخر فريد من وظائف JavaScript هو أنه يمكنك تمرير أي عدد من parameters إلى أي وظيفة function دون التسبب في خطأ .

وذلك لأن function parameters يتم تخزينها فعليًا كهيكل يشبه المصفوفة يسمى arguments .

تعتبر مثل مصفوفة Javascript العادية ، يمكن أن تنمو arguments لتحتوي على أي عدد من القيم .

تتم الإشارة إلى القيم عبر مؤشرات رقمية ، وهناك خاصية طول لتحديد عدد القيم الموجودة .

ملاحظة :

(The arguments object is not an instance of Array)

وبالتالي لا يحتوي على نفس العمليات مثل المصفوفة ;

Array.isArray(arguments) -----> always returns false.

* JavaScript لا يتجاهل parameter المبعوثة والمسماة للدالة أيضًا .

* يتم تخزين عدد الوسائط arguments التي تتوقعها الدالة في خاصية طول الدالة .

* وبما أن function هي مجرد object , لذلك يمكن أن يكون لها خصائص .

* تشير خاصية length إلى طبيعة الوظيفة ، أو عدد المعلمات التي تتوقعها .

* تعد معرفة طبيعة الوظيفة أمرًا مهمًا في جافا سكريبت لأن الوظائف لن تؤدي إلى خطأ إذا قمت بتمرير عدد كبير جدًا من parameter أو القليل جدًا منها.

مثال بسيط باستخدام ---- > arguments and function arity

```
function reflect(value) {
 return value;
}
console.log(reflect("Hi!"));       // "Hi!"
console.log(reflect("Hi!", 25));   // "Hi!"
console.log(reflect.length);       // 1
```

الدالة reflect () يتم تعريفها باستخدام(معلمة parameter) مسماة واحدة ، ولكن لا يوجد خطأ عند تمرير معلمة ثانية إلى الوظيفة.
كما أن خاصية length هي 1 نظرًا لوجود معلمة واحدة مسماة.

يتم إعادة تعريف الدالة reflect بدون parameter مسماة .
تقوم بإرجاع ---> returns arguments[0]
وهي أول parameter يتم تمريره أو إرساله .

> يعمل هذا الإصدار الجديد من الوظيفة تمامًا مثل الإصدار السابق ، لكن طوله يساوي 0.

```
reflect = function() {
return arguments[0];
};

console.log(reflect("Hi!"));      // "Hi!"
console.log(reflect("Hi!", 25));  // "Hi!"
console.log(reflect.length);      // 0
```

ملاحظة : التطبيق الأول لـ reflect () أسهل في الفهم لأنه يستخدم parameter مسماة (كما تفعل في لغات أخرى).

ومع ذلك ، في بعض الأحيان ، يكون استخدام مثل هذا التطبيق باستخدام arguments أكثر فاعلية من parameter .
على سبيل المثال ، عند الحاجة لإنشاء دالة تقبل أي عدد من parameter وترجع مجموعها sum . لا يمكنك استخدام parameter المسماة لأنك لا تعرف العدد الذي ستحتاج إليه ، لذلك في هذه الحالة ، يعد استخدام argument هو الخيار الأفضل .

```
function sum() {
  var result = 0,
      i = 0,
      len = arguments.length;
  while (i < len) {
      result += arguments[i];
      i++;
  }
  return result;
}
 console.log(sum(1, 2));          // 3
 console.log(sum(3, 4, 5, 6));    // 18
 console.log(sum(50));            // 50
 console.log(sum());              // 0
```

تقبل الدالة sum () أي عدد من parameter وتجمعها معًا عن طريق تكرار القيم الموجودة في argument باستخدام حلقة while.

تعمل الوظيفة حتى في حالة عدم تمرير أي parameter ، لأنه يتم تهيئة النتيجة بقيمة 0.

Overloading

* تدعم معظم object oriented programming التحميل الزائد للوظائف function overloading ، وهي تعريف أكثر من دالة, لهم نفس الإسم , مختلفون في عدد أو نوع parameter .

الفكرة من الـ Overloading , هي تجهيز عدة دوال لهم نفس الإسم , هذه الدوال تكون متشابهة من حيث الوظيفة , مختلفة قليلاً في الأداء .

فعلياً , تكون كل دالة تحتوي على ميزات إضافية عن الدالة التي أنشأت قبلها .

كما ذكرنا سابقًا ، يمكن أن تقبل وظائف JavaScript أي عدد من parameters ، وأنواع parameters التي تتخذها الوظيفة غير محددة على الإطلاق .

مثال يبين ما يحدث عندما تحاول التصريح عن وظيفتين بنفس الاسم :

```
function sayMessage(message) {
 console.log(message);
}
function sayMessage() {
console.log("Default message");
}
sayMessage("Hello!");

  // outputs "Default message"
```

في JavaScript ، عندما تحدد وظائف متعددة بنفس الاسم ، فإن الوظيفة التي تظهر أخيرًا في التعليمات البرمجية الخاصة بك هي التي تفوز.
تتم إزالة تعريفات الوظائف السابقة بالكامل ، والأخيرة هي التي يتم استخدامها.

```
function sayMessage(message) {

    if (arguments.length === 0) {
        message = "Default message";
    }

  console.log(message);
}
sayMessage("Hello!");

  // outputs "Hello!"
```

في هذا المثال ، تتصرف وظيفة sayMessage () بشكل مختلف بناءً على عدد parameter التي تم تمريرها.

إذا لم يتم تمرير أي parameter في (arguments.length === 0) ، فسيتم استخدام {Default message} .

غير ذلك ، يتم استخدام parameter الأولى كرسالة .

object methods

```
var person = {

    name: "Nicholas",          //property
    sayName: function() {      //method
        console.log(person.name);
    }
};
person.sayName();       // outputs "Nicholas"
```

بناء الجملة الخاص بخاصية البيانات والطريقة متطابقان تمامًا - معرف متبوعًا بنقطتين والقيمة .

يمكنك إضافة وإزالة الخصائص من الكائنات objects في أي وقت .

عندما تكون قيمة الخاصية عبارة عن function ، تعتبر الخاصية method .

يمكنك إضافة طريقة method إلى كائن بنفس الطريقة التي تضيف بها خاصية property.

في الكود التالي ،
يتم تخصيص object للمتغير person
property بخاصية name
و method طريقة تسمى sayName.

The this Object

في المثال السابق .....
تشير طريقة sayName () إلى person.name مباشرةً ، مما يؤدي إلى إنشاء اقتران وثيق بين method و object.
هذا يمثل مشكلة لعدد من الأسباب :

أولاً ، إذا قمت بتغيير اسم المتغير ، فعليك أيضًا أن تتذكر تغيير المرجع إلى هذا الاسم في method .
ثانيًا ، هذا النوع من الاقتران الضيق يجعل من الصعب استخدام نفس الوظيفة ل objects مختلفة.

الحل لهذا المشكلة هو object this يمثل object استدعاء الوظيفة.

عندما يتم استدعاء function أثناء إرفاقها ب object ، فإن قيمة {this} تساوي هذا object افتراضيًا.

لذلك ، بدلاً من الإشارة مباشرة إلى object داخل method ، تستخدم this .

يمكنك إعادة كتابة الكود من المثال السابق لاستخدام هذا:

```
var person = {

    name: "Nicholas",          //property
    sayName: function() {      //method
        console.log(this.name);
    }
};
person.sayName();

  // outputs "Nicholas"
```

```
function sayNameForAll() {
    console.log(this.name);
}
var person1= {
    name: "Nicholas",
    sayName: sayNameForAll
};
var person2= {
    name: "Greg",
    sayName: sayNameForAll
};
var name = "Michael";   //a global variable
person1.sayName();      // outputs "Nicholas"
person2.sayName();      // outputs "Greg"
sayNameForAll();        // outputs "Michael"
```

يتم إنشاء كائنين حرفيين يعينان sayName ليكون مساويًا للدالة sayNameForAll.

عندما يتم استدعاء sayName() على person1 ، فإنه ينتج "Nicholas" ؛
عندما يتم استدعاؤها على person2 ، فإنه ينتج "Greg".

عندما يتم استدعاء sayNameForAll () مباشرة ، فإنه ينتج "Michael" لأن global variable يعتبر خاصية ل object العام.

The call() Method

```
function sayNameForAll(label) {

console.log(label + ":" + this.name);

}

var person1 = {

    name: "Nicholas"

};

var person2 = {

    name: "Greg"

};

var name = "Michael";

sayNameForAll.call(this, "global");          // outputs "global:Michael"
sayNameForAll.call(person1, "person1");      // outputs "person1:Nicholas"
sayNameForAll.call(person2, "person2");      // outputs "person2:Greg"
```

يقبل sayNameForAll () معلمة واحدة تُستخدم (label) لقيمة الإخراج.

الدالة call تأخذ عدة مدخلات "parameters" وأهمهم هو الـ parameter الأول، فالـ parameter الأول هو الكائن object الذي ستشير إليه كلمة this في الدالة التي نعمل لها invocation بواسطة الدالة call ، وباقي الـ parameters تمرر إلى الدالة .

الدالة call هي دالة معرفة مسبقا في لغة javascript ، وهي دالة تتبع الكائن Function وبالتالي أي دالة تقوم بتعريفها تأخذ كل خصائص ووظائف الكائن Function ، فكما تعرف أن الدوال في javascript هي بالأساس الأول عبارة objects، وبالتالي أي دالة يمكن أن تستخدم معها الخاصية call.

فماذا تفعل الدالة call هذه ؟؟

بكل بساطة تقوم بعمل invocation للدوال في javascript.

syntax الخاص الدالة call:-

function.call(contextualObj, param1, param2, ...)

أول استدعاء للدالة يستخدم global this ويمرر في المعامل "global" لإخراج "global: Michael".

The Apply() Method

```
function sayNameForAll(label) {

console.log(label + ":" + this.name);

}

var person1 = {

    name: "Nicholas"

};

var person2 = {

    name: "Greg"

};

var name = "Michael";

sayNameForAll.Apply(this,[ "global"]);          // outputs "global:Michael"
sayNameForAll.Apply(person1, ["person1"]);      // outputs "person1:Nicholas"
sayNameForAll.Apply(person2, ["person2"]);      // outputs "person2:Greg"
```

الدالة apply تشبه تماماً الدالة call ، فكل ما قلناه سابقاً ينطبق على الدالة apply ، الاختلاف المحوري بين الدالتين يقع فيه الـ syntax، حيث أن الدالة apply تأخذ اثنين parameters فقط .

الأول ؛ هو الكائن object الذي يعد السياق الجديد كما في الدالة call ، والـ parameter الثاني ؛ هو عبارة عن مصفوفة array يتم تمرير عناصرها إلى الدالة التي نغير سياقها أثناء التنفيذ ، الـ syntax الخاص بالدالة apply:-

function.apply(contextualObj, paramsArray)

إذا كان لديك بالفعل مجموعة من البيانات ، فاستخدم apply () ؛
إذا كان لديك فقط متغيرات فردية ، فاستخدم call () .

The bind() Method

```
function sayNameForAll(label) {
console.log(label + ":" + this.name);
}
var person1 = {
    name: "Nicholas"
};
var person2 = {
    name: "Greg"
};
// create a function just for person1
1- var sayNameForPerson1 = sayNameForAll.bind(person1);
sayNameForPerson1("person1");       // outputs "person1:Nicholas"
 // create a function just for person2
 2- var sayNameForPerson2 = sayNameForAll.bind(person2, "person2");
 sayNameForPerson2();               // outputs "person2:Greg"
// attaching a method to an object doesn't change 'this'
3- person2.sayName = sayNameForPerson1;
person2.sayName("person2");         // outputs "person2:Nicholas"
```

الدالة bind تشبه إلى حد كبير جدا الدالة call ، لكن الاختلاف الجوهري يكمن في توقيت التنفيذ ، فمع الدالة call يتم التنفيذ فوري ، بمجرد ما تستدعي الدالة call يتم التنفيذ ، على عكس الدالة bind ، فالدالة bind تقوم بإرجاع دالة جديدة "bounded function" يتم تنفيذها في وقت لاحق ،

الـ syntax الخاص بالدالة bind:-

```
boundedFunc =
function.bind(contextualObj, param1,
param2, ...)
```

Summary

* تعتبر وظائف JavaScript فريدة من نوعها من حيث أنها أيضًا object ، مما يعني أنه يمكن الوصول إليها ونسخها والكتابة فوقها ومعاملتها بشكل عام تمامًا مثل أي object .

* عامل typeof يبحث عن الخواص الداخلية داخل object ويرجع " function " إذا وجدها .

* هناك نوعان من الأشكال الحرفية للوظائف : declarations and expressions .

* تحتوي إعلانات الوظيفة " declaration " على اسم الوظيفة على يمين الكلمة الأساسية للوظيفة ويتم رفعها إلى أعلى السياق الذي تم تعريفها فيه.

* يتم استخدام تعبيرات الوظائف " expressions " حيث يمكن أيضًا استخدام قيم أخرى ، مثل تعبيرات التعيين أو معلمات الوظيفة أو قيمة الإرجاع return لدالة أخرى .