UT04.1: Funciones

Contenido

1. Introducción

Las *funciones* son un tipo de objeto en JavaScript. Vamos a trabajar el uso de funciones para hacer el código más mantenible. Algunos de los contenidos que vamos a tratar ya han sido trabajados previamente, pero en el documento ofreceremos una visión más avanzada del concepto de función.

Algunos de los puntos que vamos a tratar son:

- **Closure** o cerramientos. Combinación entre funciones y contexto.
- Funciones de tipo Arrow.
- Métodos `apply()`, `call()` y `bind()`.
- Parámetros *Rest*.

1.1. Concepto de función

Una función es un bloque de código organizado y reutilizable que es usado para realizar una acción relacionada de forma sencilla. Se utilizan de forma básica para establecer estructuras o realizar cálculos.

Una función consta de un **nombre de función** que nos permite identificarla. No podemos tener dos funciones con el mismo nombre en el mismo ámbito. Normalmente son declaradas en el ámbito global, pero también pueden ser definidas dentro de otra función. Este tipo de funciones son las **funciones internas**. Entre llaves tenemos el **cuerpo de la función** que es el código que se ejecutará cuando la función sea invocada.

En la declaración de una función, debemos definir los **parámetros** que espera cuando se ejecute, los cuales serán asignados en el momento de la invocación. La **invocación de la función** se realiza mediante su nombre y el operador **()**, en los cuales indicaremos los valores que serán asignados a los parámetros de la función. No es obligatorio asignar valor a todos los parámetros de una función.

Por último, una función siembre devuelve un valor de retorno. Como vemos en el ejemplo, la función devuelve un valor `undefined`.

```
function greetingsV1(name) {
  $$result.log(`Hello ${name}`); // Hello Pablo
}
$$result.log(`Hello ${greetingsV1('Pablo')}`); // Hello undefined
```

Si queremos personalizar el valor que devuelve la función, debemos utilizar la sentencia return. En el siguiente ejemplo la función devuelve una cadena de texto con el saludo y es mostrado en la página desde fuera de la función.

```
function greetingsV2(name) {
  return (`Hello ${name}`); // Retorna "Hello Pablo"
}
$$result.log(greetingsV2('Pablo')); // Hello Pablo
```

1.2. Argumentos vs Parámetros

Un **argumento** es el valor que es pasado a la función cuando es invocada.

Un **parámetro** es la variable que hemos declarado como parte de la definición de una función.

En la primera invocación de la función del ejemplo, pasamos los argumentos *5* y *3* y se asignan a los parámetros `num1` y `num2`.

En la segunda invocación, vemos como no es necesario invocar la función con la lista completa de argumentos. Los argumentos son asignados a los parámetros en orden. Si un parámetro no recibe un argumento automáticamente queda definido con *undifined*, por esa razón, el valor de retorno es `NaN`.

```
function sum(num1, num2) {
  return num1 + num2;
}
$$result.log(sum(2, 3)); // 5
$$result.log(sum(2)); // NaN
```

1.3. El objeto `arguments`

Es una variable local disponible en todas las funciones menos en las de tipo Arrow. Esta variable nos permite referenciar los argumentos de una función como si fuera un *array*.

La ventaja de este objeto es que podemos invocar una función con un **número de argumentos indeterminados**. En este ejemplo vemos como a través de un bucle for, podemos recorrer el objeto mientras no excedamos del número total de argumentos.

```
function printAllV1() {
  for (let i = 0; i < arguments.length; i++) {
    $$result.log(arguments[i]);
  }
}
printAllV1(1, 2, 3, 4, 5); // 1 2 3 4 5
```

Aunque funciona como un *array* realmente no lo es ya que no dispone de los métodos alrededor del objeto *array*. Lo que podemos hacer es transformar `arguments` en un array.

En la versión 2 del ejemplo, estamos transformando `arguments` en un *array* para poder ordenarlo y mostrar la lista de argumentos en la página en orden creciente.

```
function printAllV2() {
  const values = Array.from(arguments);
  values.sort((a, b) => a - b);
  for (const v of values) {
    $$result.log(v);
  }
}
printAllV2(4, 3, 2, 5, 1); // 1 2 3 4 5
```

1.4. Parámetros predeterminados

Parámetros predeterminados de una función permiten asignar valores por defecto a los parámetros que no reciben valor como argumento o son *undifined*.

Versiones anteriores a ES6, para declarar valores por defecto para los parámetros teníamos que comprobar su valor en el cuerpo de la función. El siguiente ejemplo utiliza los operandos lógicos para implementar una asignación en forma de cortocircuito. En el ejemplo asignamos valores de los parámetros a *1*.

```
function multiply (a, b){
  a = a || 1;
  b = b || 1;
  return a * b;
}
$$result.log(multiply()); // 1
```

Uno de los problemas que ofrece esta solución es que los valores que se convierten a booleanos como falsos, por ejemplo, el caso del valor 0, tendríamos que hacer una comprobación especial para ellos.

Los parámetros predeterminados son declarados en la definición de la función, asignando un valor a cada parámetro.

```
function multiply (a = 1, b = 1){
  return a * b;
}
$$result.log(multiply()); // 1
$$result.log(multiply(0)); // 0
$$result.log(multiply(5)); // 5
$$result.log(multiply(5, 0)); // 0
```

Una vez que hemos asignado el valor a un parámetro, podemos reutilizarlo en los siguientes parámetros predeterminados. Como vemos en la definición, el valor de *b* es obtenido a partir del valor de *a*.

```
function multiply (a = 1, b = a + 1){
  return a * b;
}
$$result.log(multiply()); // 2
$$result.log(multiply(0)); // 0
$$result.log(multiply(5)); // 30
$$result.log(multiply(5, 0)); // 0
```

También podemos hacer uso de funciones. Hemos creado una función para generar un número aleatorio entre 0 y 10, el cual asignamos de forma predeterminada al parámetro.

```
function getRandom(){
  return Math.floor(Math.random() * 10);
}
function multiply (a = 1, b = getRandom()){
  return a * b;
}
$$result.log(multiply());
$$result.log(multiply(0));
$$result.log(multiply(5));
$$result.log(multiply(5, 0));
```

Para finalizar, tenemos que tener en cuenta que a diferencia de otros lenguajes, en JavaScript el argumento predeterminado se evalúa en el momento de la llamada, es decir, se crea un nuevo objeto cada vez que se llama a la función. En este ejemplo definimos como valor predeterminado un *array*, el cual es creado en cada invocación de la función.

```
function append(value, array = []) {
  array.push(value)
  return array
}
$$result.log(append(1)); // [1]
$$result.log(append(2)); // [2], no [1, 2]
```

2. Closure

2.1. Ámbito de funciones

El ámbito de una función refleja el alcance de las variables que hemos declarado en su cuerpo, es decir, define desde dónde están accesibles sus variables.

No podemos acceder a las variables definidas dentro de una función desde cualquier lugar fuera de la función, porque la variable se define solo en el ámbito de la función. En este ejemplo vemos como la variable `message` está salvaguardada por el ámbito de la función, por lo que el intérprete genera un error.

```
function greeting() {
  let message = "Hello";
}
greeting();
console.log(message); // ReferenceError: message is not defined
```

Podemos tener funciones definidas en el ámbito local de una función, son las denominadas **funciones internas**. Estas funciones internas su ámbito incluye, además del ámbito local, el ámbito de la **función principal**, es decir, las funciones internas tienen acceso a sus propias variables y a cualquier variable que tenga acceso la función principal. Sin embargo, la función principal no tiene acceso a las variables declaradas en el ámbito local de la función interna.

La siguiente función contiene un función interna denominada `hi()`, la cual es tiene acceso a las variables de la función principal.

```
function greetingV3() {
  let message = "Hello";
  let sayHi = function hi() {
    $$result.log(message);
  };
  sayHi(); // Hello
}
greetingV3();
```

Como vemos a continuación, en la función interna declaramos una variable local a la cual queremos acceder desde la función principal. El intérprete genera un error porque desde la función principal no podemos acceder a una variable local de la función interna.

```
function greetingV4() {
  let sayHi = function hi() {
    let text = "Hi";
  };
  sayHi();
  try {
    $$result.log(text);
  } catch (error) {
    $$result.log(error.message); //text is not defined
  }
}
greetingV4();
```

2.2. Patrón IIFE (Immediately-Invoked Function Expression)

En ocasiones vamos a necesitar que una función solamente se ejecute una única vez en nuestro código, por lo que debemos asegurarnos de que no pueda volver a invocarse desde ningún tipo de la aplicación. El patrón de diseño IIFE permite invocar una función justo en el momento de ser declarada, y que sea imposible volver a realizar esta acción. El patrón esta basado en dos conceptos:

- **Funciones anónimas**: Son funciones que no tienen asignado un nombre, por tanto, solo pueden ser invocadas en el momento de su declaración.
- **Expresiones de función**: Que permita realizar la invocación.

En el ejemplo vemos la estructura del patrón IIFE. Tenemos una función anónima definida dentro de un conjunto de paréntesis, y a continuación utilizamos el operador `()` para generar la expresión de invocación de la función anónima.

```
(function () {
  $$result.log("Hello"); // Hello
})();
```

Las funciones anónimas también pueden definir parámetros en su declaración. En este caso, hemos rediseñado el ejemplo anterior para aceptar un parámetro de entrada y que sea utilizado en el cuerpo de la función.

```
(function (name) {
  $$result.log("Hello " + name); // Hello Pablo
})("Pablo");
```

Por último, también la función anónima puede devolver un valor y ser asignado a una variable, para que pueda ser reutilizado. Vemos como la variable `result` recibe el valor resulto por IIFE.

```
let result = (function () {
  let name = "Pablo";
  return "Hello " + name;
})();
$$result.log(result); // Hello Pablo
```

2.3. Closure

La variable `name` del último ejemplo del punto anterior, al ser declarada en el ámbito de la función anónima, no tendremos acceso a ella desde fuera de la función. Este concepto nos va a servir para introducir la siguiente estructura del lenguaje, los **closures**.

Un *closure* permite acceder al ámbito de una función principal desde una función interna. En JavaScript, el *closure* se crea cada vez una función es declarada, esta característica la podemos utilizar para definir ciertas estructuras de código.

Podemos definir un *closure* como una función interna la cual siempre tiene acceso a las variables y parámetros de la función principal que la contiene, incluso cuando la ejecución de la función principal ha finalizado.

En este ejemplo tenemos una función principal `mainFunction()` en cuyo cuerpo declaramos una variable `name` y una función interna `greeting()`, la cual devolvemos como valor de retorno. Al invocar la función principal, asignamos el valor devuelto a una variable `myGreeting` que contendrá la función interna. Como el *closure* se ha creado en la declaración, aunque la función principal a terminado de ejecutarse, la función interna mantiene una referencia a sus variables locales, de esta forma, al invocar la función interna desde la variable `myGreeting` seguimos accediendo a la variables definidas en su ámbito, las cuales incluyen las variables declaradas en la función principal, `mainFunction()`.

```
function mainFunction() {
  let name = "Pablo";
  function greting() {
    $$result.log("Hello " + name);
  }
  return greting;
}
let myGreeting = mainFunction();
myGreeting(); // Hello Pablo
```

Mediante los *closures* podemos implementar el encapsulamiento de información para evitar que sea modificada sin seguir las reglas de nuestra lógica de negocio. Queremos implementar un contador que nos permita llevar un recuento de algún elemento de nuestra aplicación.

La primera aproximación podría ser declarar una función con una variable que mantenga el contador. El problema es que la variable se reinicia en cada invocación de la función.

```
function counter(){
  let count = 0;
  return ++count;
}
$$result.log(counter()); //1
$$result.log(counter()); //1
```

En un segundo intento podríamos utilizar una variable global, el problema es que puede ser modificada desde cualquier parte de nuestro código.

```
let count = 0;
function counter(){
  return ++count;
}
$$result.log(counter()); //1
$$result.log(counter()); //2
```

La solución pasa por hacer un *closure*. En la función principal declaramos la variable que mantendrá el valor del contador, la cual podemos inicializar mediante un parámetro, y devolvemos una función interna que incremente el contador. La función interna tendrá acceso a la variable de la principal independientemente de que la ejecución de la principal haya finalizado. Esto nos permite mantener el estado del contador, y encapsular su valor para que solamente a través de la función pueda ser modificado.

```
function setupCounter(val) {
  let count = val || 0;
  return function counter() {
    return ++count;
  }
}
let counter = setupCounter(5);
$$result.log(counter()); //6
$$result.log(counter()); //7
```

2.3.1. *Closure* y IIFE

La forma más habitual de utilizar los *closure* es en combinación del patrón IIFE en sustitución de un objeto de un solo método. En el siguiente ejemplo, `app` es un objeto global que le vamos a asignar el valor de retorno de una función anónima. En la función anónima declaramos una variable `computerId` que contendrá un identificador pasado por parámetro, así como una función interna que devolverá el valor del identificador. Por último, tenemos que la función anónima devuelve un objeto literal con una única propiedad con la referencia a la función interna, en la cual se ha definido en *closure*.

```
// app es un objeto global con la referencia a un objeto del IIFE.
let app = (function (id) { // Función principal anónima
  let computerId = id; // Variable con ámbito local solo accesible desde
la función anónima y por tanto encapsulada.
  let getId = function () { // función que permite cambiar el contenido d
e la variable local.
    return computerId;
  };
  return {  // Objeto devuelto por función anónima y asignado a app.
    getId: getId // propiedad que hace referencia a la función local.
  };
})(123); // Invocación de la función anónima. Se hace justo en el momento
 de su declaración.
// El objeto global puede acceder al ámbito de la función anónima a travé
s de sus funciones internas.
$$result.log(app.getId());  //123
```

2.3.2. *Closure* como objeto

Vamos a rediseñar el contador para que podamos incrementar y decrementar el contador. Necesitamos definir las dos funciones internas que realicen ambas operaciones, y devolverlas utilizando un objeto literal. Invocaremos las funciones utilizando las propiedades igual que si fueran un método.

```
function setupCounter(val) {
  let count = val || 0;
  function increment() {
    return ++count;
  }
  function decrement() {
    return --count;
  }
  return {
    increment: increment,
    decrement: decrement
  }
}
let counter = setupCounter(5);
$$result.log(counter.increment()); //6
$$result.log(counter.increment()); //7
$$result.log(counter.decrement()); //6
$$result.log(counter.decrement()); //5
```

3. Funciones *arrow*

Es una alternativa que permite compactar la expresión de una función tradicional, pero tiene ciertas limitaciones que restringen su uso en algunas situaciones. Algunas diferencias con las funciones tradicionales son:

- No disponen de enlace a `this`, por lo que no las hacen compatibles con métodos de un objeto.
- No tienen acceso al objeto `arguments`.
- No son aptas para usarse con métodos `call`, `apply` o `bind` que veremos posteriormente.
- No pueden ser usadas como constructor de objetos.

Por su simplicidad son las candidatas perfectas para ser utilizadas como funciones de *callback*.

3.1. Sintaxis de funciones *arrow*

Veamos cómo podemos crear una función *arrow*.

3.1.1. Funciones sin parámetros

Una función sin parámetro de entrada queda definida así.

```
function greeting(){
  return "Hello World";
}
$$result.log(greeting()); // Hello World
```

Para transformarla en una función *arrow* tenemos que utilizar una expresión, por ejemplo, en una asignación de variable. El nombre no es necesario y solamente el operador `()` vacío porque no tenemos argumentos. A continuación, utilizamos el operador de la función `=>` y por último indicamos el valor de retorno de forma implícita, ya que no es necesario utilizar la sentencia `return`. Es importante que los parámetros y el operador *arrow* estén en la misma línea. Por último, invocamos la función desde la variable para este caso concreto.

```
let arrow = () => "Hello World";
$$result.log(arrow()); // Hello World
```

3.1.2. Funciones con parámetros

Tenemos dos funciones con parámetros de entrada.

```
function greeting(name){
  return "Hello " + name;
}
$$result.log(greeting('Pablo')); // Hello Pablo

function multiply (a, b){
  return a * b;
}
$$result.log(multiply(3, 5)); // 15
```

Para transformalas en funciones *arrow* debemos indicar el listado de parámetros en el operador `()` y utilizarlos en el cuerpo. Nuevamente el retorno se hace de forma implícita.

```
let greet = (name) => "Hello " + name;
$$result.log(greet('Pablo')); // Hello Pablo

let multi = (a, b) => a * b;
$$result.log(multi(3, 5)); // 15
```

3.1.3. Funciones con varias líneas

Las funciones *arrow* deberían resumirse a una única línea, pero en ocasiones necesitaremos utilizar más de una. Para este tipo de funciones es obligatorio definir los límites de la función mediante llaves, los retornos implícitos no se pueden realizar por lo que necesitamos la sentencia `return`. A continuación, hemos rediseñado las funciones del caso anterior utilizando una notación de bloque.

```
let greet = (name) => {
  return "Hello " + name;
}
$$result.log(greet('Pablo')); // Hello Pablo

let multi = (a, b) => {
  return a * b;
}
$$result.log(multi(3, 5)); // 15
```

3.1.4. Retorno de objetos literales

Si necesitamos retornar un objeto literal en una función *arrow*, el cuerpo de la función debe estar entre paréntesis.

```
let greet = () => ({name: "Pablo"});
$$result.log(greet().name); // Pablo
```

3.1.5. Parámetros predeterminados

También admite el uso de parámetros predeterminados.

```
let multi = (a = 1, b = 1) => a * b;
$$result.log(multi(3)); // 3
```

3.2. Ejercicios de funciones fechas con arrays

Partimos de un array de objetos literales.

```
let computers = [
    {
        computerID: 134,
        brand: 'HP',
        model: 'EliteBook',
        memory: 16,
    },
    {
        computerID: 14,
        brand: 'HP',
        model: 'EliteBook',
        memory: 32,
    },
    {
        computerID: 456,
        brand: 'HP',
        model: 'Pavilion',
        memory: 16,
    },
];
```

1 Ordena el array utilizando la propiedad *model* de los objetos.

```
function sortComputersByModel(){
  let localComputers = [...computers];
  localComputers.sort((elemA,elemB) => elemA.model.localeCompare(elemB.mo
del));
  //EliteBook,EliteBook,Pavilion
  localComputers.forEach((elem) => $$result.log(elem.model));
}
```

2 Encuentra el elemento con la memoria mayor de 16.

```
function findHighPerformanceComputer(){
  let computer = computers.find((elem) => elem.memory > 16);
  $$result.log("ID: " + computer.computerID +
    " Brand: " + computer.brand +
    " Memory: " + computer.memory);//ID: 14 Brand: HP Memory: 32
}
```

3 Genera un array con la memoria menor o igual a 16.

```
function findHighPerformanceComputer(){
  let lowPerformanceComputer = computers.filter((elem) => elem.memory <=
16);
  lowPerformanceComputer.forEach((elem) => $$result.log(elem.computerID))
;
}
```

4 Reduce el array a una cadena de caracteres con la marca y el modelo.

```
function ArrayToString(){
  $$result.log(
    computers.reduce((str, elem) => (str += " " + elem.brand + " " + elem
.model + "."), "")
  )
}
```

5 Dado un array de enteros con números pares e impares. Separa los valores pares en un array y los impares en otro.

```
function evenAndOddNumbers(){
  let numbers = [32, -5, 66, 32, 23, 14, 32, 16];
  let evenNumbers = numbers.filter((elem) => !(elem % 2));
  let oddNumbers = numbers.filter((elem) => elem % 2);

  $$result.log(evenNumbers);//32,66,32,14,32,16
  $$result.log(oddNumbers);//-5,23
}
```

4. Objeto `function`

Las funciones son un tipo de objeto en JavaScript, y por tanto tienen métodos y propiedades. Vamos a utilizar un objeto literal para revisar los métodos y las propiedades de una función.

```
let computer = {
  computerID: 134,
  brand: 'HP',
  model: 'EliteBook',
  memory: 16,
```

```
}
```

4.1. Propiedades de `function`

Tenemos dos propiedades disponibles:

- `name`: Con el nombre de la función.
- `length`: Número de parámetros que espera la función.

```
function functionProperties(arg1, arg2, arg3) {
  $$result.log(functionProperties.name); // functionProperties
  $$result.log(functionProperties.length); // 3
}
```

4.2. Contexto de una función

El contexto es una característica de las funciones que indica sobre qué objeto se está ejecutando la función. El contexto es referenciado utilizando la palabra reservada `this`, pero a diferencia de los lenguajes de programación orientados a objetos puros como **Java**, donde `this` apunta a una instancia de clase, en JavaScript el contexto viene determinado por cómo es invocada la función.

El **contexto por defecto** es el utilizado en una función ejecutada de forma tradicional. Tenemos dos tipos de casos:

- **Ejecutada sin el modo estricto**: El contexto es el objeto `window` que representa la ventana del navegador donde visualizamos la página.
- **Ejecutada en modo estricto**: El contexto es `undefined` ya que, en principio una función tradicional no tiene que ser ejecutada bajo ningún objeto.

En este ejemplo tenemos muestra el contexto por defecto utilizando el modo estricto declarado previamente.

```
function defaultContext() {
  (function () {
    $$result.log(this); // undefined
  }());
}
```

Podemos comprobar a eliminar la directiva del modo estricto para comprobar como obtenemos el objeto `window`.

El **contexto implícito** se utiliza cuando una función es utilizada como un método, por lo tanto, `this` es la instancia del objeto con el que estamos trabajando.

La función siguiente muestra un objeto literal que incluye un método `toString()`. El contexto en este caso es la instancia del objeto, por lo que el método tiene acceso al resto de propiedades del objeto.

```
function implicitContext() {
  const computer = {
    computerID: 134,
    brand: 'HP',
    model: 'EliteBook',
    memory: 16,
    toString() {
      return `${this.computerID}: ${this.brand} ${this.model}`;
    },
  };
  $$result.log(computer.toString()); // 134: HP EliteBook
}
```

Tenemos tres tipos de contextos más:

- Contexto utilizado en los métodos `call()` y `apply()`.
- Contexto utilizado en el método `bind()`.
- Contexto en constructores. Este contesto lo estudiaremos más adelante.

4.3. Método `call()`

El método `call()` del tipo de objeto `fuction` permite invocar a la función pero eligiendo el contexto sobre el que vamos a ejecutar la función, el cual pasaremos como primer argumento en la invocación. Como podemos ver en el ejemplo, estamos invocando la función `toString()` utilizando el método `call()` y pasando como contexto un objeto concreto sobre el que estamos extrayendo la información.

```
function toString() {
  return `${this.computerID}: ${this.brand} ${this.model}`;
}
const c = toString.call(computer);
$$result.log(c); // 134: HP EliteBook
```

Podemos utilizar el método para invocar una función anónima. En el ejemplo recorremos un array de objetos, invocando una función anónima pasando como contexto cada objeto del array, así como un segundo argumento con la posición que ocupa el objeto en el array, argumento que espera la función anónima para ser ejecutada.

```
for (let i = 0; i < computers.length; i++) {
  (function (num) {
    this.toString = function () {
      $$result.log(`#${num} ${this.computerID}: ${this.brand}
${this.model}`);
    };
    this.toString();
  }).call(computers[i], i);
}
```

Una función muy interesante de `call()` es su uso en la invocación de constructores de un objeto como veremos más adelante.

4.4. Método `apply()`

El método `apply()` funciona exactamente igual que `call()`, permitiendo invocar una función pasando por parámetro el contexto sobre el que se ejecutará el cuerpo de la función. La diferencia entre ambos métodos es que mientras a `call()` le pasamos una lista de argumentos separados por comas que necesita la función para ejecutarse, en `apply()` los argumentos están empaquetados en un *array*.

En el siguiente ejemplo disponemos de un *array* de enteros y pretendemos saber el mayor y el menos. La clase `Math` dispone de los métodos `max()` y `min()` que dado una lista de argumentos nos devuelve el mayor y el menor respectivamente. Con el método `apply()` podemos invocar los métodos `max()` y `min()` directamente desde el array. Para implementar esta funcionalidad no es necesario pasar ningún contexto, por lo que es asignado a `null`.

```
const numbers = [5, 6, 2, 3, 7];
const max = Math.max.apply(null, numbers);
const min = Math.min.apply(null, numbers);
$$result.log(max); // 7
$$result.log(min); // 2
```

4.5. Método `bind()`

El método `bind()` crea una nueva función "ligada" a otra función, es decir, la nueva función tendrá el mismo cuerpo, pero podemos asignar un contexto diferente. Este nuevo contexto no puede ser sobrescrito.

Como podemos ver en este ejemplo, tenemos dos objetos, uno que tienes un método `toString()`, y el otro no. Creamos una copia del método `toString()` del primero objeto, pero le asignamos el segundo objeto como contexto, por lo que al ejecutar la función copiada accedemos al contenido del segundo objeto.

```
const computer1 = {
  computerID: 134,
  brand: 'HP',
  model: 'EliteBook',
  memory: 16,
  toString() {
    return `${this.computerID}: ${this.brand} ${this.model}`;
  },
};
const computer2 = {
  computerID: 14,
  brand: 'HP',
  model: 'EliteBook',
  memory: 32,
};

const toStringCopy = computer1.toString.bind(computer2);
$$result.log(computer1.toString());
$$result.log(toStringCopy());
```

Este tipo de funcionalidad es muy útil para trabajar con frameworks de terceros, ya que nos permite extraer funciones de objetos para que las podamos utilizar con nuestros propios objetos.

El método también acepta parámetros predeterminados que antecederán al resto de los parámetros específicos cuando la función objetivo sea llamada.

Tenemos una función que devuelve un array a partir de los argumentos de entrada. Hemos creado una copia de la función sin contexto definido, para este caso particular no es necesario indicar ningún contexto, y le pasamos un argumento predeterminado, el cual precederá a los argumentos finales de la función ligada. Los argumentos predeterminados son enviados a la función objetivo seguidos de los argumentos de la función ligada en cada invocación. Como podemos observar, la invocación de la función ligada genera un *array* con los argumentos de entrada, pero en primero tenemos los argumentos predeterminados declarados en la función ligada, y después los definidos en la invocación.

```
function list() {
  return Array.from(arguments);
}
const listCopy = list.bind(undefined, 37);
$$result.log(list(1, 2, 3)); // [1, 2, 3]
$$result.log(listCopy()); // [37]
$$result.log(listCopy(1, 2, 3)); // [37, 1, 2, 3]
```

5. Parámetros *rest* y operador *spread*

5.1. Parámetros *rest*

Los parámetros *rest* nos permiten representan un número indeterminado de argumentos como si fueran un *array*, lo que nos permite trabajar con un número indefinido de argumentos.

En esta ocasión, la función `greet()` espera una lista de argumentos que recoge a través del parámetro *rest* denominado `names`. Como *array*, podemos iterar sobre `names` utilizando el método `forEach()`.

```
function greet(...names) {
  names.forEach((name) => $$result.log(name));
}
const names = ['Mary', 'John', 'James'];
greet(names); // Mary, John, James
```

Podemos utilizar los parámetros *rest* en combinación con parámetros normales, pero siempre teniendo en cuenta que los parámetros *rest* deben ser declarados en última posición. En el ejemplo siguiente hemos rediseñado la función de saludo para que muestre un mensaje personalizado a través de `message`, y el resto de los argumentos son empaquetados en el *array* del parámetro *rest*.

```
function greet(message = 'Hi', ...names) {
  $$result.log(`${message}: `);
  names.forEach((name) => $$result.log(name));
}
const names = ['Mary', 'John', 'James'];
```

```
greet('Welcome', names); // Welcome: Mary, John, James
```

5.2. Operador *spread*

Aunque ya lo hemos utilizado anteriormente, conviene recordad qué es el operador *spread*. Este operador toma el camino inverso a los parámetros *rest*. Partimos de un *array* para invocar una función que espera una serie de parámetros, el operador *spread* desestructura el *array* para poder invocar la función, pasando cada valor del *array* a su correspondiente parámetro en la función.

```
function greet(message = 'Hi', name1, name2) {
  $$result.log(`${message}: `);
  $$result.log(name1);
  $$result.log(name2);
}
const names = ['Mary', 'John'];
greet('Welcome', ...names); // Welcome: Mary, John
```