Validar tarjeta de crédito Máster Card

¿Cuántas veces te has puesto a escribir el número de tu tarjeta de crédito y te ha bailado una cifra? ¿Sabes que al igual que con el NIF existe una validación para capturar esos fallos tontos?

Si hablamos de tarjetas de crédito podemos pensar en varios tipos de validaciones:

1. Chequear si el usuario ha introducido el número de tarjeta en el formato válido, o transformarlo al que nos haga falta: aquí pasa como con el NIF, igual le ha puesto separadores con espacios o guiones, o les faltan o sobran números... esto lo controlaremos más adelante en el capítulo de expresiones regulares.
2. Chequear si el número de tarjeta lo ha tecleado bien el usuario, para ello utilizamos el algoritmo de Luhn (en el caso de MasterCard y otras tarjetas).
3. Chequear si la tarjeta de verdad es "buena", es decir si no es una tarjeta robada, está activa y también validar si la compra no es fraudulenta (para esto ya requieres interacción con un servidor).

Vamos a centrarnos en el caso 2, vamos a ver si el usuario ha tecleado bien la tarjeta.

Enunciado

Para empezar, vamos a ver el algoritmo de Luhn, que es el que se utiliza para validar las tarjetas de crédito de MasterCard y otras tarjetas. Las tarjetas de crédito tienen un número de 16 dígitos, pero para explicar como funciona este algoritmo, simplificamos y nos quedamos con un número de 8 dígitos, para que sea más fácil, el numero en concreto que vamos a probar es este.

12345674

De esos dígitos los siete primeros son los números de la tarjeta y el último es el dígito de control (cómo pasaba con el NIF y la letra correspondiente), así que, lo primero que tenemos que hacer es separar el número de la tarjeta del dígito de control, y nos queda el siguiente número:

1234567

El último número (él 4) lo sacamos aparte, ya lo utilizaremos para comparar con el dígito que vamos a calcular, y comprobar así, si el número está bien formado.

Y seguimos los pasos:

- Empezamos por la derecha y uno sí uno no, multiplicamos cada dígito por 2.
- Sobre ese resultado, si uno de los números es mayor que 9, sumamos las decenas y las unidades.

Números	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Multiplico	*2		*2		*2		*2
Resultado	2	2	6	4	10	6	14
Si paso de 10 sumo unidades y decenas	2	2	6	4	(1+0=1)	6	(1+4=5)

Números	**1**	**2**	**3**	**4**	**5**		**6**	**7**
Números resultantes	2	2	6	4	1		6	5

Sobre ese resultado, sumo todos esos números:

2 + 2 + 6 + 4 + 1 + 6 + 5 = 26

Ahora calculo:

Flag = 10 - (suma % 10)

Es decir sería:

Flag = 10 - (26 % 10) = 10 - 6 = 4

> suma % 10 es el resto de dividir el valor que tiene la variable suma entre 10.

Ya lo que nos queda es comprobar que el flag coincide con el original (valor 4) coincide con este valor.

¿Oye y por qué tanto follón de multiplicar, sumar...? Aquí van mates detrás, fíjate que con un sólo dígito de control estamos controlando muchos casos, y para una tarjeta de crédito, lo normal es usar sólo números (es cómodo para por ejemplo un datáfono)

Aún así este algoritmo tiene sus debilidades:

- Hay ciertas permutaciones que no detecta, por ejemplo: 09 y 90
- Algunos errores dobles, por ejemplo, si en vez de 55 pongo 66

Firma y pruebas

Parámetros de entrada:

En este caso podemos elegir entre dos opciones:

- Podemos pasar el número de tarjeta como un numero (en teoría una master card no puede empezar por cero).
- Podemos pasar el número de tarjeta como un string.

Cada aproximación tiene sus pros y cons:

- Si usamos la primera (number) está muy bien porque limitamos a un número la entrada, pero después internamente tendríamos que pasarlo a string para iterar de forma fácil con cada dígito.
- Si usamos la segunda (string) la parte buena es que directamente podemos ir iterando por el array, la parte mala es que en un string podemos meter cualquier cosa.

¿Qué vamos a hacer?

Vamos a optar por tener un parámetro de tipo string, y en módulos posteriores ya veremos cómo hacer la comprobación de que la tarjeta cumple el formato esperado utilizando una expresión regular y si no, lanzaremos una excepción (*throw*).

La firmaría quedaría tal que así

```
export const validaTarjetaMasterCard = (numeroTarjeta: string): boolean => {
  // TODO
};
```

Una vez que hayamos completado la implementación recibiendo el parámetro como string, te queda como *te toca* para después, implementarlo aceptando como parámetro un numero.

Parámetro de salida:

- Un booleano que diga si la tarjeta está bien formada o no.

Evaluando el problema

Vamos a diseccionar este problema en pequeñas partes (las podemos llamar pruebas de concepto), así cuando tengamos todos los trozos, los unimos y tenemos el problema resuelto.

Antes de nada, creamos un archivo nuevo que se va a llamar *master-card.helpers.ts*, y en este archivo vamos a poner todos los métodos de ayuda que vamos a necesitar para resolver el problema.

Eliminando el dígito de control

Consisten en qué si tengo string, devolver otro string con un elemento menos (el del final quitado), es decir si tengo:

"12345674"

Me tiene que devolver:

"1234567"

La función se podría llamar

```
export const eliminaUltimoDigito = (numeroTarjeta: string): string => {
  // TODO
};
```

Te toca, intenta implementarlo, ya que estás añade pruebas unitarias 😃.

Hay varias formas de implementar esto:

- Podrías utilizar un método como slice.
- Podrías usar un bucle hasta el último elemento menos uno.
- ...

Vamos a añadir unas pruebas para ver que lo estamos haciendo bien.

Creamos un archivo para las pruebas *master-card.helpers.spec.ts* y añadimos las pruebas unitarias:

master-card.helpers.spec.ts

```
describe("eliminaUltimoDigito", () => {
  it("debería devolver un throw si la entrada es undefined", () => {
    // Arrange
    const cadena: any = undefined;

    // Act
    const result = () => eliminaUltimoDigito(cadena);

    // Assert
    expect(result).toThrowError("No se ha introducido una cadena");
  });

  it("debería devolver un throw si la entrada es null", () => {
    // Arrange
    const cadena: any = null;

    // Act
    const result = () => eliminaUltimoDigito(cadena);

    // Assert
    expect(result).toThrowError("No se ha introducido una cadena");
  });

  it("debería devolver un throw si la entrada es un string vacío", () => {
    // Arrange
    const cadena = "";

    // Act
    const result = () => eliminaUltimoDigito(cadena);

    // Assert
    expect(result).toThrowError("No se ha introducido una cadena");
  });

  it("debería devolver un string sin el último dígito", () => {
    // Arrange
    const cadena = "123456789";

    // Act
    const result = eliminaUltimoDigito(cadena);

    // Assert
    expect(result).toBe("12345678");
  });
});
```

Y para la implementación vamos a elegir, la opción de spread operator.

master-card.helpers.ts

```
export const eliminaUltimoDigito = (cadena: string): string => {
  if (!cadena) {
    throw new Error("No se ha introducido una cadena");
  }
  return cadena.slice(0, cadena.length - 1);
};
```

Te toca, prueba a implementarlo con un bucle *For* ¿Cuál es tu favorita?

Obteniendo el último dígito

Ahora vamos a obtener el último dígito de un string, por ejemplo:

"12345674"

Me tiene que devolver:

"4"

La función se podría llamar

```
export const obtenerUltimoDigito = (cadena: string): number => {
  // TODO
};
```

Te toca, intenta implementarlo, ya que estás añade pruebas unitarias 😀.

Aquí tienes varias opciones:

- Puedes usar *length* y *cadena[]*
- Puedes usar slice.

Ojo hay que tener en cuenta casos "arista" como string vacío, null o undefined (eso lo podrás ver en las pruebas unitarias)

Vamos a añadir unas pruebas para ver que lo estamos haciendo bien.

master-card.helpers.spec.ts

```
describe("obtenerUltimoDigito", () => {
  it("debería devolver un throw si la entrada es undefined", () => {
    // Arrange
    const cadena: any = undefined;

    // Act
    const result = () => obtenerUltimoDigito(cadena);
```

```
    // Assert
    expect(result).toThrowError("No se ha introducido una cadena");
  });

  it("debería devolver un throw si la entrada es null", () => {
    // Arrange
    const cadena: any = null;

    // Act
    const result = () => obtenerUltimoDigito(cadena);

    // Assert

    expect(result).toThrowError("No se ha introducido una cadena");
  });

  it("debería devolver un throw si la entrada es un string vacío", () => {
    // Arrange
    const cadena = "";

    // Act

    const result = () => obtenerUltimoDigito(cadena);

    // Assert

    expect(result).toThrowError("No se ha introducido una cadena");
  });

  it("debería devolver el último número de la cadena convertido a número", () => {
    // Arrange
    const cadena = "123456789";

    // Act
    const result = obtenerUltimoDigito(cadena);

    // Assert
    expect(result).toBe(9);
  });
});
```

Y para la implementación vamos a elegir, la opción de *slice*.

master-card.helpers.ts

```
export const obtenerUltimoDigito = (cadena: string): number => {
  if (!cadena) {
    throw new Error("La cadena es null o undefined");
  }
  return parseInt(cadena.slice(-1));
};
```

Cómo ves, en este método hemos usado *parseInt*, que nos permite convertir un string a un número.

Recorriendo el string del revés y multiplicando por 2 y saltando uno

Vamos a por la siguiente prueba de concepto, recorrer el string del revés y empezando por el final el primero elemento que nos encontremos, multiplicar por 2, saltarnos uno y vuelta a empezar, podemos devolver un array de números

La firma del método sería:

```
export const multiplicaPorDosSaltandoUno = (cadena: string): number[] => {
  // TODO
};
```

Este método empieza a tener un poco de lógica, vamos a definir una batería de pruebas unitarias, te toca, piensa primero que pruebas podrías definir y después contrastarlas con las que hemos implementado.

master-card.helpers.spec.ts

```
describe("multiplicaPorDosSaltandoUno", () => {
  it("debería devolver un throw si la entrada es undefined", () => {
    // Arrange
    const cadena: any = undefined;

    // Act
    const result = () => multiplicaPorDosSaltandoUno(cadena);

    // Assert
    expect(result).toThrowError("No se ha introducido una cadena");
  });

  it("debería devolver un throw si la entrada es null", () => {
    // Arrange
    const cadena: any = null;

    // Act
    const result = () => multiplicaPorDosSaltandoUno(cadena);

    // Assert
    expect(result).toThrowError("No se ha introducido una cadena");
  });

  it("debería devolver un throw si la entrada es un string vacío", () => {
    // Arrange
    const cadena = "";

    // Act
    const result = () => multiplicaPorDosSaltandoUno(cadena);
```

```
    // Assert
    expect(result).toThrowError("No se ha introducido una cadena");
  });

  it("debería devolver un array con los números separados y si leemos el array de
atrás para delante, los números que están en posición impar multiplicarlos por
dos", () => {
    // Arrange
    const cadena = "123456789";

    // Act
    const resultado = multiplicaPorDosSaltandoUno(cadena);

    // Assert
    expect(resultado).toEqual([2, 2, 6, 4, 10, 6, 14, 8, 18]);
  });

  // TODO: (Te Toca) añade más casos, 16 digitos, 1 digito, 2 digitos, 3 digitos
});
```

Y ahora la implementación, te toca ¿Cómo implementarías esto? No te agobies que esto puede ser un poco duro de hacer, lee las pistas si tienes dudas, y si no te contamos la solución paso a paso.

Aquí van unas pistas:

- Tienes que recorrer el array, esta vez del final al principio, aquí puede merecer la pena usar un bucle for pero empiezas desde el final (length -1) y terminas en cero, y el iterador va decrementando (i--).
- Por otro lado, puedes definir encima del for una variable booleana, que se llame *pordos*, de primeras está a true y en cada iteración del false cambiamos su valor *pordos = !pordos*.
- Partimos de un array de números vacíos, y en cada iteración del for, si *pordos* es true, multiplicamos por dos el número y lo añadimos al array, si *pordos* es false, añadimos el número tal cual al array.

Aquí va la solución:

master-card.helpers.ts

```
export const multiplicaPorDosSaltandoUno = (cadena: string): number[] => {
  if (!cadena) {
    throw new Error("No se ha introducido una cadena");
  }

  let resultado: number[] = [];
  let porDos = true;

  for (let i = cadena.length - 1; i >= 0; i--) {
    const nuevoNumero = porDos ? parseInt(cadena[i]) * 2 : parseInt(cadena[i]);
    resultado = [nuevoNumero, ...resultado];
    porDos = !porDos;
  }
```

```
  return resultado;
};
```

Hay otra forma de convertir un string a un número, que es usando +, por ejemplo:

```
const numero = +"123";
```

Pero esto no se considera una buena práctica, ya que no es muy legible.

Una duda que te puede surgir es ¿Por qué tenemos que pasar a array de números en vez de devolver un sólo *string*? Esto es porque algunas multiplicaciones nos pueden devolver número de dos dígitos.

Sumando decenas y unidades

El siguiente paso es recorrer el array de números y los números que tengan dos cifras, sumar decenas y unidades, con lo que volvemos a tener números de una cifra.

La firma del método sería:

```
export const sumaDecenasUnidades = (numeros: number[]): number[] => {
  // TODO
};
```

Te toca ¿Piensas en unas pruebas unitarias acordes?

Aquí va la propuesta de pruebas unitarias:

master-card.helpers.spec.ts

```
describe("sumaDecenasUnidades", () => {
  it("debería devolver un throw si la entrada es undefined", () => {
    // Arrange
    const numero: any = undefined;

    // Act
    const result = () => sumaDecenasUnidadesColeccion(numero);

    // Assert
    expect(result).toThrowError("No se ha introducido un número");
  });

  it("debería devolver un throw si la entrada es null", () => {
    // Arrange
    const numero: any = null;

    // Act
    const result = () => sumaDecenasUnidadesColeccion(numero);
```

```
    // Assert
    expect(result).toThrowError("No se ha introducido un número");
  });

  it("debería devolver un array con la suma de las decenas y las unidades de los
números de la entrada", () => {
    // Arrange
    const input: number[] = [1, 2, 18, 4, 5, 12, 7, 14, 9];

    // Act
    const result = sumaDecenasUnidadesColeccion(input);

    // Assert
    expect(result).toEqual([1, 2, 9, 4, 5, 3, 7, 5, 9]);
  });
});
```

Vamos ahora a implementar el método... no te lo vas a creer, TE TOCA ! 😃, prueba y si no estás seguro, aquí van unas pistas:

- Esta vez tienes que recorrer el array y da igual el orden, y además vas a devolver un array con el mismo número de elementos (el resultado de aplicar la operación), así pues, un *map* de javascript puede ir perfecto.
- Si el número es menor que 10, lo pasas tal cual, si el número es mayor que nueve, lo que tienes que hacer es sumar las decenas y las unidades, para ello puedes utilizar el operador módulo (%) y la división entera (*Math.floor*), esto lo podrías separar en un método para un sólo item, o lo puedes implementar directamente en el map (si tienes dudas de como se usan estos operadores, pruébalos aparte, y chequea la documentación y si tienes dudas a por tu mentor).

Aquí va la solución:

master-card.helpers.ts

```
const sumaDecenasUnidades = (numero: number): number => {
  if (numero < 10) return numero;

  const unidades = numero % 10;
  const decenas = Math.floor(numero / 10);
  return unidades + decenas;
};

export const sumaDecenasUnidadesColeccion = (numeros: number[]): number[] => {
  if (!numeros) {
    throw new Error("No se ha introducido un número");
  }

  return numeros.map(sumaDecenasUnidades);
};
```

En este caso hemos separado la lógica de sumar decenas y unidades en un método aparte, para que sea más fácil de ver cómo funciona.

Sumando los dígitos

Para sumar los números resultantes, lo tenemos claro, sólo tenemos que utilizar un reduce, la firma del método:

```
export const sumaDigitos = (numeros: number[]): number => {
  // TODO
};
```

Este método es bien sencillo, por si acaso añadimos pruebas unitarias.

master-card.helpers.spec.ts

```
describe("sumaDigitos", () => {
  it("debería devolver un throw si la entrada es undefined", () => {
    // Arrange
    const numero: any = undefined;

    // Act
    const result = () => sumaDigitos(numero);

    // Assert
    expect(result).toThrowError("No se ha introducido un número");
  });

  it("debería devolver un throw si la entrada es null", () => {
    // Arrange
    const numero: any = null;

    // Act
    const result = () => sumaDigitos(numero);

    // Assert
    expect(result).toThrowError("No se ha introducido un número");
  });

  it("debería de devolver la suma de los dígitos de un array de números", () => {
    // Arrange
    const numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9];

    // Act
    const result = sumaDigitos(numeros);

    // Assert
    expect(result).toBe(45);
  });
});
```

Y vamos ahora a implementarlo, ¿Te animas a implementarlo y que pase las pruebas? Mírate en el módulo de loops como funcionaba reduce y verás que es muy fácil (también podrías implementarlo usando un bucle for y comparar las dos soluciones).

Aquí va la solución:

master-card.helpers.ts

```
export const sumaDigitos = (numeros: number[]): number => {
  if (!numeros) {
    throw new Error("No se ha introducido un número");
  }

  return numeros.reduce(
    (acumulador, numeroActual) => acumulador + numeroActual,
    0
  );
};
```

Calculando el flag

Vamos ahora a calcular el flag en base a la suma anterior, para ello tenemos que hacer lo siguiente:

- Calcular el módulo de 10 de la suma total.
- Restar 10 menos módulo de 10 que hemos calculado en el paso anterior.
- Devolver el resultado.

Es decir la formula era:

Flag = 10 - (suma % 10)

Un ejemplo

2 + 2 + 6 + 4 + 1 + 6 + 5 = 26

Ahora calculo:

Flag = 10 - (suma % 10)

Es decir sería:

Flag = 10 - (26 % 10) = 10 - 6 = 4

La firma del método:

```
export const calculaFlagDeSumaTotal = (sumaTotal: number): number => {
  // TODO
};
```

Vamos a añadir unas pruebas unitarias:

master-card.helpers.spec.ts

```
describe("calculaFlagDeSumaTotal", () => {
  it("debería devolver un throw si la entrada es undefined", () => {
    // Arrange
    const sumaTotal: any = undefined;

    // Act
    const result = () => calculaFlagDeSumaTotal(sumaTotal);

    // Assert
    expect(result).toThrowError("No se ha introducido un número");
  });

  it("debería devolver un throw si la entrada es null", () => {
    // Arrange
    const sumaTotal: any = null;

    // Act
    const result = () => calculaFlagDeSumaTotal(sumaTotal);

    // Assert
    expect(result).toThrowError("No se ha introducido un número");
  });

  it("debería devolver 7 si la suma total es 73", () => {
    // Arrange
    const sumaTotal = 73;

    // Act
    const result = calculaFlagDeSumaTotal(sumaTotal);

    // Assert
    expect(result).toBe(7);
  });

  it("debería devolver 3 si la suma total es 67", () => {
    // Arrange
    const sumaTotal = 67;

    // Act
    const result = calculaFlagDeSumaTotal(sumaTotal);

    // Assert
    expect(result).toBe(3);
  });
});
```

Y ahora la implementación, te toca, aquí van unas pistas:

- Vamos a implementar la siguiente lógica: 10 - (sumaTotal % 10).

La solución:

master-card.helpers.ts

```
export const calculaFlagDeSumaTotal = (sumaTotal: number): number => {
  if (!sumaTotal) {
    throw new Error("No se ha introducido un número");
  }

  return 10 - (sumaTotal % 10);
};
```

Implementando

Bueno ya tenemos todas las piezas juntas, vamos a crear una función que las agrupe, pero primero de todo vamos a crearnos un nuevo archivo que se llame *master-card.validator.ts* y vamos a crear la función que nos devuelva un booleano, que nos diga si la tarjeta es válida o no.

```
export const validaTarjetaMasterCard = (numeroTarjeta: string): boolean => {
  // TODO
};
```

Vamos a buscar en internet números de Master Card válidos y también añadimos erróneos, aquí va una batería de pruebas:

Tarjetas válidas:

```
5506927427317625;
5553042241984105;
5555553753048194;
5555555555554444;
```

Tarjetas no válidas:

```
5506927627317626;
5553042241944106;
5525553753048195;
5554555555554445;
```

Y vamos a armar todas las piezas que habíamos implementado.

Primero vamos a añadir las pruebas unitarias, creamos un archivo nuevo *master-card.validator.spec.ts* y añadimos las pruebas unitarias:

master-card.validator.spec.ts

```
describe("esTarjetaMasterCardValida", () => {
  it("debería devolver un throw si la entrada es undefined", () => {
    // Arrange
    const numeroTarjeta: any = undefined;

    // Act
    const result = () => esTarjetaMasterCardValida(numeroTarjeta);

    // Assert
    expect(result).toThrowError(
      "No se ha introducido un número de tarjeta correcto"
    );
  });

  it("debería devolver un throw si la entrada es null", () => {
    // Arrange
    const numeroTarjeta: any = null;

    // Act
    const result = () => esTarjetaMasterCardValida(numeroTarjeta);

    // Assert
    expect(result).toThrowError(
      "No se ha introducido un número de tarjeta correcto"
    );
  });

  it("debería devolver un throw si la entrada es un string vacío", () => {
    // Arrange
    const numeroTarjeta = "";

    // Act
    const result = () => esTarjetaMasterCardValida(numeroTarjeta);

    // Assert
    expect(result).toThrowError(
      "No se ha introducido un número de tarjeta correcto"
    );
  });

  it("debería devolver un throw si la entrada no se puede convertir a un número",
() => {
    // Arrange
    const numeroTarjeta = "abalkajdañlkdjseo";

    // Act
    const result = () => esTarjetaMasterCardValida(numeroTarjeta);
```

```
    // Assert
    expect(result).toThrowError(
      "No se ha introducido un número de tarjeta correcto"
    );
  });

  it("debería devolver un throw si la entrada no tiene 16 dígitos", () => {
    // Arrange
    const numeroTarjeta = "123456789";

    // Act
    const result = () => esTarjetaMasterCardValida(numeroTarjeta);

    // Assert
    expect(result).toThrowError(
      "No se ha introducido un número de tarjeta correcto"
    );
  });

  it.each([
    ["5506927427317625", true],
    ["5553042241984105", true],
    ["5555553753048194", true],
    ["5506927627317626", false],
    ["5553042241944106", false],
    ["5525553753048195", false],
  ])(
    "si la entrada es %s debería devolver %s",
    (numeroTarjeta: string, expected: boolean) => {
      // Arrange

      // Act
      const result = esTarjetaMasterCardValida(numeroTarjeta);

      // Assert
      expect(result).toBe(expected);
    }
  );
});
```

Y ahora la implementación:

master-card.validator.ts

```
export const esTarjetaMasterCardValida = (numeroTarjeta: string): boolean => {
  if (
    !numeroTarjeta ||
    numeroTarjeta.length !== 16 ||
    isNaN(parseInt(numeroTarjeta))
  ) {
    throw new Error("No se ha introducido un número de tarjeta correcto");
```

```
  }
  const ultimoDigito = obtenerUltimoDigito(numeroTarjeta);
  const masterCardSinDigitoControl = eliminaUltimoDigito(numeroTarjeta);
  const masterCardMultiplicadaPorDos = multiplicaPorDosSaltandoUno(
    masterCardSinDigitoControl
  );
  const masterCardSumaDecenasUnidades = sumaDecenasUnidadesColeccion(
    masterCardMultiplicadaPorDos
  );
  const masterCardSumaTotal = sumaDigitos(masterCardSumaDecenasUnidades);

  const flagControlCalculado = calculaFlagDeSumaTotal(masterCardSumaTotal);

  return flagControlCalculado === ultimoDigito;
};
```

Ya tenemos esto en verde. Además de comprobar si *numeroTarjeta* es undefined, null o un string vacío, también hemos comprobado si se puede convertir a un número y si tiene 16 dígitos. Sino se cumplieran estas dos condiciones no tendría sentido seguir con la validación y lanzaría una excepción.

Refactorizando,

Ya que lo tenemos funcionando y con pruebas unitarias, vamos a refactorizar un poco.

Si te fijas este método cuesta un poco de leer, sería buena idea agrupar código por funcionalidad, una mejora clara sería separar la comprobación para saber si el número está bien formado:

```
+ const estaLaTarjetaBienFormada = (numeroTarjeta : string) => {
+  if (
+    !numeroTarjeta ||
+    numeroTarjeta.length !== 16 ||
+    isNaN(parseInt(numeroTarjeta))
+  ) {
+    throw new Error("No se ha introducido un número de tarjeta correcto");
+  }
+ }

export const esTarjetaMasterCardValida = (numeroTarjeta: string): boolean => {
-  if (
-    !numeroTarjeta ||
-    numeroTarjeta.length !== 16 ||
-    isNaN(parseInt(numeroTarjeta))
-  ) {
-    throw new Error("No se ha introducido un número de tarjeta correcto");
-  }
+ estaLaTarjetaBienFormada(numeroTarjeta);
+
  const ultimoDigito = obtenerUltimoDigito(numeroTarjeta);
  const masterCardSinDigitoControl = eliminaUltimoDigito(numeroTarjeta);
  const masterCardMultiplicadaPorDos = multiplicaPorDosSaltandoUno(
    masterCardSinDigitoControl
```

```
  );
  const masterCardSumaDecenasUnidades = sumaDecenasUnidadesColeccion(
    masterCardMultiplicadaPorDos
  );
  const masterCardSumaTotal = sumaDigitos(masterCardSumaDecenasUnidades);

  const flagControlCalculado = calculaFlagDeSumaTotal(masterCardSumaTotal);

  return flagControlCalculado === ultimoDigito;
};
```

Siguiente paso, hay una sección clara en el código en la que se extrae el ultimo dígito, y se devuelve tanto el numero sin digito de control, como el dígito de control, lo podemos agrupar:

```
+ interface TarjetaMasterCard {
+  masterCardSinDigitoControl: string;
+  ultimoDigito: number;
+ }
+
+ const separaDigitoControl = (numeroTarjeta : string) : TarjetaMasterCard => ({
+  ultimoDigito: obtenerUltimoDigito(numeroTarjeta),
+  masterCardSinDigitoControl: eliminaUltimoDigito(numeroTarjeta)
+ })

export const esTarjetaMasterCardValida = (numeroTarjeta: string): boolean => {
  estaLaTarjetaBienFormada(numeroTarjeta);

-  const ultimoDigito = obtenerUltimoDigito(numeroTarjeta);
-  const masterCardSinDigitoControl = eliminaUltimoDigito(numeroTarjeta);
+  const { ultimoDigito, masterCardSinDigitoControl } =
separaDigitoControl(numeroTarjeta);

  const masterCardMultiplicadaPorDos = multiplicaPorDosSaltandoUno(
    masterCardSinDigitoControl
  );
  const masterCardSumaDecenasUnidades = sumaDecenasUnidadesColeccion(
    masterCardMultiplicadaPorDos
  );
  const masterCardSumaTotal = sumaDigitos(masterCardSumaDecenasUnidades);

  const flagControlCalculado = calculaFlagDeSumaTotal(masterCardSumaTotal);

  return flagControlCalculado === ultimoDigito;
};
```

Ahora tenemos otro paso que podemos aislar en una función y es el cálculo del flag de la suma total:

```
+ const calculaFlagDeValidacion = (masterCardSinDigitoControl : string): number =>
{
```

```
+  const masterCardMultiplicadaPorDos = multiplicaPorDosSaltandoUno(
+    masterCardSinDigitoControl
+  );
+  const masterCardSumaDecenasUnidades = sumaDecenasUnidadesColeccion(
+    masterCardMultiplicadaPorDos
+  );
+  const masterCardSumaTotal = sumaDigitos(masterCardSumaDecenasUnidades);
+
+  return calculaFlagDeSumaTotal(masterCardSumaTotal);
+ }

export const esTarjetaMasterCardValida = (numeroTarjeta: string): boolean => {
  estaLaTarjetaBienFormada(numeroTarjeta);

  const { ultimoDigito, masterCardSinDigitoControl } =
separaDigitoControl(numeroTarjeta);

-  const masterCardMultiplicadaPorDos = multiplicaPorDosSaltandoUno(
-    masterCardSinDigitoControl
-  );
-  const masterCardSumaDecenasUnidades = sumaDecenasUnidadesColeccion(
-    masterCardMultiplicadaPorDos
-  );
-  const masterCardSumaTotal = sumaDigitos(masterCardSumaDecenasUnidades);
-
-  const flagControlCalculado = calculaFlagDeSumaTotal(masterCardSumaTotal);
+  const flagControlCalculado =
calculaFlagDeValidacion(masterCardSinDigitoControl);

  return flagControlCalculado === ultimoDigito;
};
```

Fíjate como se ha quedado la función, ¿Se entiende mejor ahora? Y por otro lado... como tenemos una batería de pruebas unitarias nos podemos quedar tranquilos de que funciona la refactorización.

Más información

Este algoritmo se usa para Visa, MasterCard, American Express...

Si quieres conocer más a fondo el algoritmo de Luhn:

https://en.wikipedia.org/wiki/Luhn_algorithm

https://www.dcode.fr/luhn-algorithm