UNIVERSIDAD DE COLIMA

FACULTAD DE TELEMÁTICA

INTRODUCCIÓN A LA PROGRAMACIÓN

CARLOS ALBERTO FLORES CORTÉS

Antología digital con Ejercicios

CONTENIDO

Presentación

La presente antología de ejercicios ha sido diseñada como material de apoyo para la asignatura **Introducción a la Programación**, dirigida a estudiantes y personas que se inician en el aprendizaje del pensamiento computacional y el desarrollo de software.

El documento reúne una serie de ejercicios organizados de manera progresiva, que abarcan desde los conceptos más básicos de la programación, como la sintaxis elemental y el uso de operadores, hasta temas fundamentales como funciones, estructuras condicionales y ciclos. Cada ejercicio está compuesto por varios problemas que describen una función o algoritmo a resolver, con el objetivo de fortalecer la comprensión lógica y la capacidad de análisis del estudiante.

Los ejercicios han sido cuidadosamente diseñados para acompañar el proceso de aprendizaje de forma gradual: comienzan con algoritmos sencillos y aumentan paulatinamente en complejidad, llegando a plantear retos moderados que, sin ser difíciles, invitan al estudiante a reflexionar, practicar y consolidar los conocimientos adquiridos. De esta manera, la antología funciona como una guía práctica que permite poner en acción el avance teórico visto en clase.

Si bien la sintaxis y las descripciones de los problemas fueron pensadas originalmente para el lenguaje de programación **TypeScript**, los ejercicios no dependen de características exclusivas de dicho lenguaje. Por ello, pueden adaptarse y utilizarse sin dificultad para practicar con cualquier otro lenguaje de programación, lo que los convierte en un recurso flexible y reutilizable.

Adicionalmente, se recomienda el uso de comandos básicos de **Git** para la gestión y actualización de los ejercicios, tanto en repositorios locales como en plataformas remotas como GitHub. Esta práctica busca fomentar desde etapas tempranas el uso de herramientas fundamentales en el desarrollo de software profesional.

Finalmente, esta antología puede ser también un recurso valioso para instructores y docentes, ya que funciona como un banco de ejercicios que puede integrarse fácilmente a cursos, talleres o sesiones prácticas, facilitando la enseñanza y evaluación de los conceptos básicos de programación.

🥷 Ejercicio 01 - Hola, mundo

Los temas que se estudian al realizar este reto son:

1. Sintaxis básica de TypeScript: Aprenderás a utilizar la sintaxis de TypeScript para declarar variables y funciones, así como para implementar la lógica necesaria para obtener el resultado esperado.
2. Tipos de datos en TypeScript: Aprenderás a utilizar los tipos de datos básicos de TypeScript, incluyendo `string`, `number` y `boolean`, así como a declarar variables con tipos de datos personalizados.
3. Interpolación de cadenas de texto: Aprenderás a utilizar la interpolación de cadenas de texto para crear mensajes que incluyan valores de variables.
4. Funciones y valores de retorno: Los estudiantes aprenderán a crear funciones que reciban parámetros y regresen valores, así como a utilizar los valores de retorno en otras funciones.
5. Pruebas y demostración del funcionamiento: Aprenderás a utilizar la consola para ejecutar programas y mostrar resultados.
6. Trabajo con repositorios de control de versiones: Aprenderás a utilizar un repositorio de control de versiones para almacenar y entregar el código fuente de sus programas.

Estos temas abarcan los aspectos fundamentales de TypeScript, incluyendo la sintaxis, los tipos de datos, las funciones y las pruebas de programa. Al realizar este ejercicio, los estudiantes tendrán la oportunidad de consolidar su comprensión de estos conceptos y desarrollar habilidades prácticas en la implementación de código en TypeScript.

🎯 Objetivo de aprendizaje

1. Comprender la estructura básica de un programa escrito en TypeScript.
2. Familiarizarse con la sintaxis de TypeScript, incluyendo la declaración de variables, tipos de datos y funciones.
3. Practicar la creación de funciones con parámetros y valores de retorno.
4. Aprender a utilizar la consola para ejecutar programas y mostrar resultados.
5. Demostrar la capacidad para trabajar con código fuente y utilizar un repositorio para almacenar y entregar el código.
6. Adquirir habilidades de resolución de problemas y lógica al desarrollar soluciones para los requerimientos funcionales.

📋 Instrucciones

1. Utilizando `typescript` codifica las funciones que se indican en la sección `requerimientos funcionales` de este documento.
2. Las funciones deben ser codificadas en un archivo llamado `app.ts`.
3. Las funciones deben tener el nombre que se indica y el número y tipo de parámetros que se especifican en la sección `requerimientos funcionales`.
4. En el archivo `demo.ts` se deben incluir ejemplos de código que muestren el correcto funcionamiento de las funciones.
5. Las funciones deben ser probadas y ejecutadas utilizando la consola.
6. El código fuente final debe ser almacenado en este repositorio de GitHub.

📥 Entregables

- Código fuente de la solución a los requerimientos planteados en la sección `requerimientos funcionales` en este repositorio

👨‍💻 Requerimientos funcionales

Los requerimientos funcionales se refieren a las acciones específicas que las funciones deben realizar y los tipos de datos que deben recibir y retornar. Las firmas de las funciones muestran la sintaxis correcta para definir las funciones, especificando los nombres y tipos de los parámetros, así como el tipo de dato que retorna cada función.

1️⃣ sayHelloWithName

Escribe una función llamada `sayHelloWithName` que reciba como parámetro una cadena de texto (`String`) con el nombre de una persona y regrese otra cadena de texto (`String`) con el siguiente formato: `Hola, <nombre>!`, donde `<nombre>` es el nombre de la persona que se recibió como parámetro.

```
None
sayHelloWithName(name: string): string
```

2️⃣ add

Escribir una función llamada `add` que reciba dos parámetros de tipo numérico (`number`) y regrese como resultado la suma de los dos números.

```
None
add(a: number, b: number): number
```

🚀 Ejecución de programas

Para ejecutar un programa utilizar:

```
None
npx ts-node nombre-archivo
```

Por ejemplo:

```
None
npx ts-node demo
```

🚦 Ejecución de pruebas unitarias

Para ejecutar una prueba unitaria utilizar:

```
None
npx jest nombre-de-funcion
```

Por ejemplo:

```
None
npx jest add
```

Comandos de Git y GitHub

Actualizacón del repositorio local

Cada vez que se terminó e actualizar uno o más archivo utilizar, pasar los cambios a `staging` utilizando:

```
None
git add archivo.ext
```

Un `git add` por cada archivo que se actualizó

Una vez que se agregaron los archivo para la nueva versión, confirmar la nueva versión utilizando:

```
None
git commit -m "mensaje"
```

Si al hacer `commit` el linter detecta errores: 1. Corregir los errores, 2. Volver a hacer git add por cada archivo corregido 3. Volver a hacer el commit. Repetir estos 3 pasos hasta que no se obtengan errores por el linter.

Actualización del repositorio remoto

Para enviar las actualizaciones al repositorio remoto utilizar:

```
None
git push origin
```

📚 Recursos

Aquí tienes una lista de recursos recomendados que puedes utilizar como referencia y consulta:

- TypeScript
- TypeScript Deep Dive
- TypeScript in 5 minutes
- MDN JavaScript
- DevDocs
- TypeScript Playground
- Awesome TypeScript

🥷 Ejercicio 02: Sintaxis básica de Typescript

Los temas que se estudian al realizar este reto son:

1. Operadores aritméticos y de asignación: Aprenderás a utilizar los operadores aritméticos, como la suma, resta, multiplicación y división, para realizar operaciones matemáticas en TypeScript. Además, explorarás los operadores de asignación, como el operador de asignación simple y los operadores compuestos, para asignar valores a variables de forma eficiente.
2. Identificadores: Comprenderás qué son los identificadores en TypeScript y cómo utilizarlos para dar nombres significativos a variables, funciones y otros elementos en tu código. Aprenderás las convenciones y buenas prácticas para nombrar identificadores de manera clara y legible.

3. Distinción entre mayúsculas y minúsculas: Entenderás la importancia de la distinción entre mayúsculas y minúsculas en TypeScript y cómo afecta la forma en que se interpretan los identificadores y las palabras clave. Aprenderás a seguir las reglas de distinción de casos y evitar errores comunes relacionados con la sensibilidad de mayúsculas y minúsculas.
4. Sentencias: Aprenderás qué son las sentencias en TypeScript y cómo utilizarlas para construir la lógica de tus programas. Explorarás diferentes tipos de sentencias, como sentencias condicionales (if-else), bucles (for, while) y sentencias de control de flujo (break, continue), para controlar el flujo de ejecución de tu código.
5. Comentarios: Aprenderás a utilizar comentarios en TypeScript para documentar tu código y hacerlo más comprensible para ti y otros desarrolladores. Conocerás los diferentes tipos de comentarios, como los comentarios de una línea y los comentarios de múltiples líneas, y aprenderás cómo y cuándo utilizarlos adecuadamente.
6. Punto y coma: Comprenderás la importancia del punto y coma en TypeScript como separador de instrucciones y aprenderás a utilizarlo correctamente. Explorarás las convenciones y buenas prácticas relacionadas con el uso de punto y coma en diferentes situaciones, como al finalizar una instrucción o en la declaración de variables.
7. Interpolación: Aprenderás a utilizar la interpolación de cadenas de texto en TypeScript para combinar valores de variables con texto de una manera más concisa y legible. Conocerás la sintaxis y las mejores prácticas para realizar la interpolación de cadenas y crear mensajes dinámicos en tus programas.

🎯 Objetivo de aprendizaje

1. Comprender la estructura básica de un programa escrito en TypeScript.
2. Familiarizarse con la sintaxis de TypeScript, incluyendo la declaración de variables, tipos de datos y funciones.
3. Practicar la creación de funciones con parámetros y valores de retorno.
4. Aprender a utilizar la consola para ejecutar programas y mostrar resultados.
5. Demostrar la capacidad para trabajar con código fuente y utilizar un repositorio para almacenar y entregar el código.
6. Adquirir habilidades de resolución de problemas y lógica al desarrollar soluciones para los requerimientos funcionales.

📋 Instrucciones

1. Utilizando `typescript` codifica las funciones que se indican en la sección `requerimientos funcionales` de este documento.
2. Las funciones deben ser codificadas en un archivo llamado `app.ts`.

3. Las funciones deben tener el nombre que se indica y el número y tipo de parámetros que se especifican en la sección `requerimientos funcionales`.
4. En el archivo `demo.ts` se deben incluir ejemplos de código que muestren el correcto funcionamiento de las funciones.
5. Las funciones deben ser probadas y ejecutadas utilizando la consola.
6. El código fuente final debe ser almacenado en este repositorio de GitHub.

📥 Entregables

- Código fuente de la solución a los requerimientos planteados en la sección `requerimientos funcionales` en este repositorio

Requerimientos funcionales

En estos requerimientos funcionales se especifican las acciones que deben realizar las funciones y los tipos de datos que deben recibir y retornar. Las firmas de las funciones indican la sintaxis correcta para definir las funciones, mostrando los nombres y tipos de los parámetros, así como el tipo de dato que retorna cada función.

1️⃣ sum

Escribir una función que sume dos números.

```
sum(numberA: number, numberB): number;
```

2️⃣ subtract

Escribir una función que reste dos números.

```
subtract(numberA: number, numberB): number
```

3️⃣ multiply

Escribir una función que multiplique dos números.

```
multiply(numberA: number, numberB): number
```

4️⃣ divide

Escribir una función que divida dos números.

```
divide(numberA: number, numberB): number
```

5️⃣ remainder

Escribir una función que obtenga el residuo de una división entera entre dos números.

Firma de la función: `remainder(numberA: number, numberB): number`

🚀 Ejecución de programas

Para ejecutar un programa utilizar:

```
None
npx ts-node nombre-archivo
```

Por ejemplo:

```
None
npx ts-node demo
```

🚦 Ejecución de pruebas unitarias

Para ejecutar una prueba unitaria utilizar:

```
None
npx jest nombre-de-funcion
```

Por ejemplo:

```
None
npx jest add
```

Comandos de Git y GitHub

Actualizacón del repositorio local

Cada vez que se terminó e actualizar uno o más archivo utilizar, pasar los cambios a `staging` utilizando:

```
None
git add archivo.ext
```

Un `git add` por cada archivo que se actualizó

Una vez que se agregaron los archivo para la nueva versión, confirmar la nueva versión utilizando:

```
None
git commit -m "mensaje"
```

Si al hacer `commit` el linter detecta errores: 1. Corregir los errores, 2. Volver a hacer git add por cada archivo corregido 3. Volver a hacer el commit. Repetir estos 3 pasos hasta que no se obtengan errores por el linter.

Actualización del repositorio remoto

Para enviar las actualizaciones al repositorio remoto utilizar:

```
None
git push origin
```

📚 Recursos

Aquí tienes una lista de recursos recomendados que puedes utilizar como referencia y consulta:

- TypeScript
- TypeScript Deep Dive
- TypeScript in 5 minutes
- MDN JavaScript
- DevDocs
- TypeScript Playground
- Awesome TypeScript

🥷 Ejercicio 03: Primeras líneas con Typescript

Los temas que se estudian al realizar este reto son:

1. Operadores aritméticos y de asignación: Aprenderás a utilizar los operadores aritméticos, como la suma, resta, multiplicación y división, para realizar operaciones matemáticas en TypeScript. Además, explorarás los operadores de asignación, como el operador de asignación simple y los operadores compuestos, para asignar valores a variables de forma eficiente.
2. Identificadores: Comprenderás qué son los identificadores en TypeScript y cómo utilizarlos para dar nombres significativos a variables, funciones y otros elementos en tu código. Aprenderás las convenciones y buenas prácticas para nombrar identificadores de manera clara y legible.
3. Distinción entre mayúsculas y minúsculas: Entenderás la importancia de la distinción entre mayúsculas y minúsculas en TypeScript y cómo afecta la forma en que se interpretan los identificadores y las palabras clave. Aprenderás a seguir las reglas de distinción de casos y evitar errores comunes relacionados con la sensibilidad de mayúsculas y minúsculas.
4. Sentencias: Aprenderás qué son las sentencias en TypeScript y cómo utilizarlas para construir la lógica de tus programas. Explorarás diferentes tipos de sentencias, como sentencias condicionales (if-else), bucles (for, while) y sentencias de control de flujo (break, continue), para controlar el flujo de ejecución de tu código.
5. Comentarios: Aprenderás a utilizar comentarios en TypeScript para documentar tu código y hacerlo más comprensible para ti y otros desarrolladores. Conocerás los diferentes tipos de comentarios, como los comentarios de una línea y los comentarios de múltiples líneas, y aprenderás cómo y cuándo utilizarlos adecuadamente.
6. Punto y coma: Comprenderás la importancia del punto y coma en TypeScript como separador de instrucciones y aprenderás a utilizarlo correctamente. Explorarás las convenciones y buenas prácticas relacionadas con el uso de punto y coma en diferentes situaciones, como al finalizar una instrucción o en la declaración de variables.
7. Interpolación: Aprenderás a utilizar la interpolación de cadenas de texto en TypeScript para combinar valores de variables con texto de una manera más concisa y legible. Conocerás la sintaxis y las mejores prácticas para realizar la interpolación de cadenas y crear mensajes dinámicos en tus programas.

🎯 Objetivo de aprendizaje

1. Comprender la estructura básica de un programa escrito en TypeScript.
2. Familiarizarse con la sintaxis de TypeScript, incluyendo la declaración de variables, tipos de datos y funciones.

3. Practicar la creación de funciones con parámetros y valores de retorno.
4. Aprender a utilizar la consola para ejecutar programas y mostrar resultados.
5. Demostrar la capacidad para trabajar con código fuente y utilizar un repositorio para almacenar y entregar el código.
6. Adquirir habilidades de resolución de problemas y lógica al desarrollar soluciones para los requerimientos funcionales.

📋 Instrucciones

1. Utilizando `typescript` codifica las funciones que se indican en la sección `requerimientos funcionales` de este documento.
2. Las funciones deben ser codificadas en un archivo llamado `app.ts`.
3. Las funciones deben tener el nombre que se indica y el número y tipo de parámetros que se especifican en la sección `requerimientos funcionales`.
4. En el archivo `demo.ts` se deben incluir ejemplos de código que muestren el correcto funcionamiento de las funciones.
5. Las funciones deben ser probadas y ejecutadas utilizando la consola.
6. El código fuente final debe ser almacenado en este repositorio de GitHub.

📥 Entregables

- Código fuente de la solución a los requerimientos planteados en la sección `requerimientos funcionales` en este repositorio

Requerimientos funcionales

En estos requerimientos funcionales se especifican las acciones que deben realizar las funciones y los tipos de datos que deben recibir y retornar. Las firmas de las funciones indican la sintaxis correcta para definir las funciones, mostrando los nombres y tipos de los parámetros, así como el tipo de dato que retorna cada función.

1️⃣ getDollars

Escribe una función llamada `getDollars` que tome un valor en pesos mexicanos como parámetro y lo convierta a dólares estadounidenses. La función debe considerar un tipo de cambio dado como un segundo parámetro. Devuelve el valor convertido a dólares.

Parámetros:

- `pesos`: Un número que representa la cantidad en pesos mexicanos.

- `tipoCambio`: Un número que representa el tipo de cambio de pesos a dólares.

Resultado:

- Un número que representa la cantidad convertida en dólares.

```
getDollars(pesos: number, tipoCambio: number): number
```

2️⃣ getAreaCircle

Escribe una función llamada `getAreaCircle` que calcule el área de un círculo. La función debe tomar como parámetro el radio del círculo y devolver el área. Utiliza el valor de π (pi) para realizar el cálculo.

Parámetros:

- `radio`: Un número que representa el radio del círculo.

Resultado:

- Un número que representa el área del círculo.

```
getAreaCircle(radio: number): number
```

3️⃣ getFahrenheit

Escribe una función llamada `getFahrenheit` que convierta una temperatura dada en grados Celsius a grados Fahrenheit. La función debe tomar como parámetro la temperatura en grados Celsius y devolver el valor equivalente en grados Fahrenheit.

Parámetros:

- `gradosCelsius`: Un número que representa la temperatura en grados Celsius.

Resultado:

- Un número que representa la temperatura convertida en grados Fahrenheit.

```
getFahrenheit(gradosCelsius: number): number
```

4️⃣ getAreaTrapezoid

Escribe una función llamada `getAreaTrapezoid` que calcule el área de un trapecio. La función debe tomar como parámetros las longitudes de las dos bases y la altura del trapecio. Devuelve el área calculada.

Parámetros:

- `baseMayor`: Un número que representa la longitud de la base mayor del trapecio.
- `baseMenor`: Un número que representa la longitud de la base menor del trapecio.
- `altura`: Un número que representa la altura del trapecio.

Resultado:

- Un número que representa el área del trapecio.

```
getAreaTrapezoid(baseMayor: number, baseMenor: number, altura: number): number
```

🚀 Ejecución de programas

Para ejecutar un programa utilizar:

```
None
npx ts-node nombre-archivo
```

Por ejemplo:

```
None
npx ts-node demo
```

🚦 Ejecución de pruebas unitarias

Para ejecutar una prueba unitaria utilizar:

```
None
npx jest nombre-de-funcion
```

Por ejemplo:

```
None
npx jest add
```

Comandos de Git y GitHub

Actualizacón del repositorio local

Cada vez que se terminó e actualizar uno o más archivo utilizar, pasar los cambios a `staging` utilizando:

```
None
git add archivo.ext
```

Un `git add` por cada archivo que se actualizó

Una vez que se agregaron los archivo para la nueva versión, confirmar la nueva versión utilizando:

```
None
git commit -m "mensaje"
```

Si al hacer `commit` el linter detecta errores: 1. Corregir los errores, 2. Volver a hacer git add por cada archivo corregido 3. Volver a hacer el commit. Repetir estos 3 pasos hasta que no se obtengan errores por el linter.

Actualización del repositorio remoto

Para enviar las actualizaciones al repositorio remoto utilizar:

```
None
git push origin
```

📚 Recursos

Aquí tienes una lista de recursos recomendados que puedes utilizar como referencia y consulta:

- TypeScript
- TypeScript Deep Dive
- TypeScript in 5 minutes
- MDN JavaScript
- DevDocs
- TypeScript Playground
- Awesome TypeScript

🥷 Ejercicio 04: Introducción a funciones

Los temas que se estudian al realizar este reto son:

1. Sintaxis básica de una función. Aprenderás a definir funciones en TypeScript, entendiendo la estructura básica que incluye el nombre de la función, los parámetros, y el cuerpo de la función donde se ejecuta el código.
2. Parámetros y valores de retorno. Descubrirás cómo pasar datos a las funciones mediante parámetros y cómo obtener un resultado mediante valores de retorno, comprendiendo la importancia de los tipos en ambos casos.

🎯 Objetivos de aprendizaje

1. Comprender la sintaxis básica de una función en TypeScript, incluyendo la declaración de funciones, el nombre de la función, los parámetros y el cuerpo de la función.
2. Ser capaz de declarar y definir una función simple en TypeScript, utilizando la palabra clave "function", y asignarle un nombre descriptivo.
3. Aprender a declarar y utilizar parámetros en una función, comprendiendo cómo se definen y cómo se accede a sus valores dentro del cuerpo de la función.
4. Entender la importancia de los tipos de datos en los parámetros de una función y cómo TypeScript permite especificar los tipos de datos de los parámetros para mejorar la seguridad y la legibilidad del código.
5. Ser capaz de escribir una función que devuelve un valor específico y entender cómo se especifica el tipo de valor de retorno en la declaración de la función.
6. Practicar la creación de funciones simples que aceptan parámetros, realizan operaciones en ellos y devuelven un resultado coherente.

📋 Instrucciones

1. Utilizando `typescript` codifica las funciones que se indican en la sección `requerimientos funcionales` de este documento.
2. Las funciones deben ser codificadas en un archivo llamado `app.ts`.
3. Las funciones deben tener el nombre que se indica y el número y tipo de parámetros que se especifican en la sección `requerimientos funcionales`.
4. En el archivo `demo.ts` se deben incluir ejemplos de código que muestren el correcto funcionamiento de las funciones.
5. Las funciones deben ser probadas y ejecutadas utilizando la consola.
6. El código fuente final debe ser almacenado en este repositorio de GitHub.

📥 Entregables

- Código fuente de la solución a los requerimientos planteados en la sección `requerimientos funcionales` en este repositorio

👨‍💻 Requerimientos funcionales

Los requerimientos funcionales se refieren a las acciones específicas que las funciones deben realizar y los tipos de datos que deben recibir y retornar. Las firmas de las funciones muestran la sintaxis correcta para definir las funciones, especificando los nombres y tipos de los parámetros, así como el tipo de dato que retorna cada función.

1️⃣ getAverage

Escribe una función que tome tres números como parámetros y calcule el promedio de esos tres valores. Asegúrate de que la función retorne el resultado como un número.

```
getAverage(number1: number, number2: number, number3: number): number
```

2️⃣ getSquarePerimeter

Escribe una función que calcule el perímetro de un cuadrado. La función debe recibir como parámetro la longitud de un lado del cuadrado y devolver el valor del perímetro.

```
getSquarePerimeter(sideLength: number): number
```

3️⃣ getMilesToKilometers

Diseña una función que convierta una distancia de millas a kilómetros. La función debe recibir la distancia en millas como argumento y devolver el equivalente en kilómetros, utilizando la conversión estándar donde 1 milla es igual a 1.60934 kilómetros.

```
getMilesToKilometers(miles: number): number
```

4️⃣ getDoubleNumber

Implementa una función que calcule y devuelva el doble de un número. La función debe aceptar un número como parámetro y retornar el resultado de multiplicar ese número por dos.

```
getDoubleNumber(number: number): number
```

5️⃣ getTriangleArea

Crea una función que calcule el área de un triángulo. La función debe aceptar la base y la altura del triángulo como parámetros, y devolver el área.

```
getTriangleArea(base: number, height: number): number
```

6️⃣ getSphereVolume

debe recibir el radio de la esfera como parámetro y devolver el volumen, utilizando la fórmula 4/3 x Pi x radio^3.

```
getSphereVolume(radius: number): number
```

🚀 Ejecución de programas

Para ejecutar un programa utilizar:

```
None
npx ts-node nombre-archivo
```

Por ejemplo:

```
None
npx ts-node demo
```

🚦 Ejecución de pruebas unitarias

Para ejecutar una prueba unitaria utilizar:

```
None
npx jest nombre-de-funcion
```

Por ejemplo:

```
None
npx jest add
```

Comandos de Git y GitHub

Actualización del repositorio local

Cada vez que se terminó e actualizar uno o más archivo utilizar, pasar los cambios a `staging` utilizando:

```
None
git add archivo.ext
```

Un `git add` por cada archivo que se actualizó

Una vez que se agregaron los archivo para la nueva versión, confirmar la nueva versión utilizando:

```
None
git commit -m "mensaje"
```

Si al hacer `commit` el linter detecta errores: 1. Corregir los errores, 2. Volver a hacer git add por cada archivo corregido 3. Volver a hacer el commit. Repetir estos 3 pasos hasta que no se obtengan errores por el linter.

Actualización del repositorio remoto

Para enviar las actualizaciones al repositorio remoto utilizar:

```
None
git push origin
```

📚 Recursos

Aquí tienes una lista de recursos recomendados que puedes utilizar como referencia y consulta:

- TypeScript
- TypeScript Deep Dive
- TypeScript in 5 minutes
- MDN JavaScript
- DevDocs
- TypeScript Playground
- Awesome TypeScript

🥷 Ejercicio 05: Programación de funciones

Los temas que se estudian al realizar este reto son:

1. Sintaxis básica de una función. Aprenderás a definir funciones en TypeScript, entendiendo la estructura básica que incluye el nombre de la función, los parámetros, y el cuerpo de la función donde se ejecuta el código.
2. Parámetros y valores de retorno. Descubrirás cómo pasar datos a las funciones mediante parámetros y cómo obtener un resultado mediante valores de retorno, comprendiendo la importancia de los tipos en ambos casos.

🎯 Objetivos de aprendizaje

1. Comprender la sintaxis básica de una función en TypeScript, incluyendo la declaración de funciones, el nombre de la función, los parámetros y el cuerpo de la función.
2. Ser capaz de declarar y definir una función simple en TypeScript, utilizando la palabra clave "function", y asignarle un nombre descriptivo.
3. Aprender a declarar y utilizar parámetros en una función, comprendiendo cómo se definen y cómo se accede a sus valores dentro del cuerpo de la función.
4. Entender la importancia de los tipos de datos en los parámetros de una función y cómo TypeScript permite especificar los tipos de datos de los parámetros para mejorar la seguridad y la legibilidad del código.
5. Ser capaz de escribir una función que devuelve un valor específico y entender cómo se especifica el tipo de valor de retorno en la declaración de la función.
6. Practicar la creación de funciones simples que aceptan parámetros, realizan operaciones en ellos y devuelven un resultado coherente.

📋 Instrucciones

1. Utilizando `typescript` codifica las funciones que se indican en la sección `requerimientos funcionales` de este documento.
2. Las funciones deben ser codificadas en un archivo llamado `app.ts`.
3. Las funciones deben tener el nombre que se indica y el número y tipo de parámetros que se especifican en la sección `requerimientos funcionales`.
4. En el archivo `demo.ts` se deben incluir ejemplos de código que muestren el correcto funcionamiento de las funciones.
5. Las funciones deben ser probadas y ejecutadas utilizando la consola.
6. El código fuente final debe ser almacenado en este repositorio de GitHub.

📥 Entregables

- Código fuente de la solución a los requerimientos planteados en la sección `requerimientos funcionales` en este repositorio

👨‍💻 Requerimientos funcionales

Los requerimientos funcionales se refieren a las acciones específicas que las funciones deben realizar y los tipos de datos que deben recibir y retornar. Las firmas de las funciones muestran la sintaxis correcta para definir las funciones, especificando los nombres y tipos de los parámetros, así como el tipo de dato que retorna cada función.

1️⃣ getHypotenuse

Escribe una función que calcule la longitud de la hipotenusa de un triángulo rectángulo. La función debe recibir las longitudes de los dos catetos como parámetros y devolver la longitud de la hipotenusa.

```
getHypotenuse(side1: number, side2: number): number
```

2️⃣ geSeconds

Escribe una función que convierta horas a segundos. La función debe recibir la cantidad de horas como parámetro y devolver la cantidad de segundos equivalente, donde 1 hora es igual a 3600 segundos.

```
getSeconds(hours: number): number
```

3️⃣ getMiles

Implementa una función que convierta distancias en kilómetros a millas. La función debe recibir la distancia en kilómetros como argumento y devolver la distancia equivalente en millas, donde 1 kilómetro es igual a 0.621371 millas.

```
getMiles(kilometers: number): number
```

4️⃣ getLitres

Crea una función que convierta pies cúbicos a litros. La función debe recibir el volumen en pies cúbicos como argumento y devolver el volumen equivalente en litros, donde 1 pie cúbico es igual a 28.3168 litros.

```
getLitres(cubicFeet: number): number
```

5️⃣ getCylinderSurfaceArea

Crea una función que calcule el área de la superficie lateral de un cilindro. La función debe recibir el radio y la altura del cilindro como parámetros y devolver el área de la superficie lateral, donde el área de la superficie lateral de un cilindro se calcula como 2 x π x radio x altura.

```
getCylinderSurfaceArea(radius: number, height: number): number
```

🚀 Ejecución de programas

Para ejecutar un programa utilizar:

```
None
npx ts-node nombre-archivo
```

Por ejemplo:

```
None
npx ts-node demo
```

🚦 Ejecución de pruebas unitarias

Para ejecutar una prueba unitaria utilizar:

```
None
npx jest nombre-de-funcion
```

Por ejemplo:

```
None
npx jest add
```

Comandos de Git y GitHub

Actualizacón del repositorio local

Cada vez que se terminó e actualizar uno o más archivo utilizar, pasar los cambios a `staging` utilizando:

```
None
git add archivo.ext
```

Un `git add` por cada archivo que se actualizó

Una vez que se agregaron los archivo para la nueva versión, confirmar la nueva versión utilizando:

```
None
git commit -m "mensaje"
```

Si al hacer `commit` el linter detecta errores: 1. Corregir los errores, 2. Volver a hacer git add por cada archivo corregido 3. Volver a hacer el commit. Repetir estos 3 pasos hasta que no se obtengan errores por el linter.

Actualización del repositorio remoto

Para enviar las actualizaciones al repositorio remoto utilizar:

```
None
git push origin
```

📚 Recursos

Aquí tienes una lista de recursos recomendados que puedes utilizar como referencia y consulta:

- TypeScript
- TypeScript Deep Dive
- TypeScript in 5 minutes
- MDN JavaScript
- DevDocs
- TypeScript Playground
- Awesome TypeScript

🥷 Ejercicio 06: Introducción a Sentencias condicionales

Los temas que se estudian al realizar este reto son:

- Sintaxis de las estructuras condicionales en TypeScript.
- El operador ternario como una forma concisa de expresar condiciones.
- La sentencia if y su uso en la toma de decisiones.
- La sentencia if...else para manejar casos alternativos.
- La sentencia if...else if para evaluar múltiples condiciones en orden.
- Cómo controlar el flujo de un programa con condicionales.

🎯 Objetivos de aprendizaje

- Comprender la sintaxis de las estructuras condicionales en TypeScript y su importancia en la programación.
- Utilizar el operador ternario de manera efectiva para expresar condiciones de forma concisa en sus programas.
- Dominar el uso de la sentencia if para tomar decisiones basadas en condiciones específicas.
- Aplicar la sentencia if...else para manejar casos alternativos y ejecutar diferentes bloques de código según la evaluación de una condición.
- Utilizar la sentencia if...else if para evaluar múltiples condiciones en orden y ejecutar el bloque de código correspondiente al primer caso verdadero.
- Aprender a controlar el flujo de un programa mediante la implementación de estructuras condicionales.

📋 Instrucciones

1. Utilizando `typescript` codifica las funciones que se indican en la sección `requerimientos funcionales` de este documento.
2. Las funciones deben ser codificadas en un archivo llamado `app.ts`.
3. Las funciones deben tener el nombre que se indica y el número y tipo de parámetros que se especifican en la sección `requerimientos funcionales`.
4. En el archivo `demo.ts` se deben incluir ejemplos de código que muestren el correcto funcionamiento de las funciones.
5. Las funciones deben ser probadas y ejecutadas utilizando la consola.
6. El código fuente final debe ser almacenado en este repositorio de GitHub.

📥 Entregables

- Código fuente de la solución a los requerimientos planteados en la sección `requerimientos funcionales` en este repositorio

👨‍💻 Requerimientos funcionales

Los requerimientos funcionales se refieren a las acciones específicas que las funciones deben realizar y los tipos de datos que deben recibir y retornar. Las firmas de las funciones muestran la sintaxis correcta para definir las funciones, especificando los nombres y tipos de los parámetros, así como el tipo de dato que retorna cada función.

1️⃣ isAdult

Escribir una función que reciba como parámetro un número con el valor de la edad de una persona y regrese `true` si es mayor de edad o `false` si no es.

```
isAdult(age: number): boolean
```

2️⃣ toTitle

Escribir una función que reciba como parámetros un `string` con un mensaje y un `booleano` para indicar si es un título con `true` o si no lo es con `false`. Si el valor del booleano es `true`, regresará el mismo mensaje pero con todas las letras en mayúsculas.

```
toTitle(message: string, isTitle: boolean): string
```

3️⃣ sayHello*

Escribir una función que reciba como parámetro un `string` con el nombre de una persona y un `booleano` para indicar si esta persona es un administrador o no. Si la persona es un administrador, la función regresará un saludo y la clave de acceso del sistema, que es "1234", por ejemplo "Hola, Juan. Tu clave es 1234". Si la persona no es un administrador, la función regresará un saludo simple, por ejemplo "Hola, Juan".

```
sayHello(name: string, isAdmin: boolean): string
```

4️⃣ totalCost

Escribir una función que reciba como parámetros el costo del producto y la cantidad de productos comprados. La función deberá regresar el costo total. Si el costo total excede de $1000, deberá aplicar un descuento del 15%

```
totalCost(productCost: number, quantity: number): number
```

5️⃣ getDiscount*

Escribir una función que recibe como parámetro el total de la venta y regrese el porcentaje de descuento que le corresponde. Si la venta es mayor que $1000 el descuento es del 25% de lo contrario el descuento es del 10%

```
getDiscount(total: number): number
```

6️⃣ getCinemaCost

Escribir una función que reciba como parámetros la fecha de la función de cine y el número de boletos a comprar. La función deberá calcular el costo total, tomando en cuenta que el costo de cada boleto es de $100, que los jueves hay promoción de 3x2 y que el resto de los días se aplica un 10% de descuento.

```
getCinemaCost(date: Date, tickets: number): number
```

7️⃣ grade

Escribir una función que reciba como parámetro un número entero con una calificación y regrese una cadena de texto según la siguiente escala:

- Si la calificación es mayor a 90, la función debe devolver "Champion!".
- Si la calificación es mayor a 80, la función debe devolver "Good".
- Si la calificación es mayor a 60, la función debe devolver "Not bad".
- Para cualquier otro caso, la función debe devolver "Try again".

```
grade(score: number): string
```

8️⃣ hasAccess

Escribe una función que determine si una persona tiene acceso según el color de su camisa y el color de sus zapatos. La función debe recibir dos cadenas de texto como parámetros, que representan el color de la camisa y el color de los zapatos. Si alguno de los dos es de color blanco, la función debe devolver `true`; en caso contrario, debe devolver `false`.

```
hasAccess(shirtColor: string, shoesColor: string): boolean
```

9️⃣ isStudent

Escribir una función que reciba como parámetros si tiene o no identificación de estudiante y la edad. Para ser estudiante es necesario tener identificación y ser mayor de 18. La función debe regresar `true` si es estudiante o `false` si no lo es.

```
isStudent(hasID: boolean, age: number): boolean
```

🚀 Ejecución de programas

Para ejecutar un programa utilizar:

```
None
npx ts-node nombre-archivo
```

Por ejemplo:

```
None
npx ts-node demo
```

🚦 Ejecución de pruebas unitarias

Para ejecutar una prueba unitaria utilizar:

```
None
npx jest nombre-de-funcion
```

Por ejemplo:

```
None
npx jest add
```

Comandos de Git y GitHub

Actualizacón del repositorio local

Cada vez que se terminó e actualizar uno o más archivo utilizar, pasar los cambios a `staging` utilizando:

```
None
git add archivo.ext
```

Un `git add` por cada archivo que se actualizó

Una vez que se agregaron los archivo para la nueva versión, confirmar la nueva versión utilizando:

```
None
git commit -m "mensaje"
```

Si al hacer `commit` el linter detecta errores: 1. Corregir los errores, 2. Volver a hacer git add por cada archivo corregido 3. Volver a hacer el commit. Repetir estos 3 pasos hasta que no se obtengan errores por el linter.

Actualización del repositorio remoto

Para enviar las actualizaciones al repositorio remoto utilizar:

```
None
git push origin
```

📚 Recursos

Aquí tienes una lista de recursos recomendados que puedes utilizar como referencia y consulta:

- TypeScript
- TypeScript Deep Dive
- TypeScript in 5 minutes
- MDN JavaScript
- DevDocs
- TypeScript Playground
- Awesome TypeScript

🥷 Ejercicio 07 Sentencias condicionales

Los temas que se estudian al realizar este reto son:

- Sintaxis de las estructuras condicionales en TypeScript.
- El operador ternario como una forma concisa de expresar condiciones.
- La sentencia if y su uso en la toma de decisiones.
- La sentencia if...else para manejar casos alternativos.
- La sentencia if...else if para evaluar múltiples condiciones en orden.
- Cómo controlar el flujo de un programa con condicionales.

🎯 Objetivos de aprendizaje

- Comprender la sintaxis de las estructuras condicionales en TypeScript y su importancia en la programación.
- Utilizar el operador ternario de manera efectiva para expresar condiciones de forma concisa en sus programas.
- Dominar el uso de la sentencia if para tomar decisiones basadas en condiciones específicas.
- Aplicar la sentencia if...else para manejar casos alternativos y ejecutar diferentes bloques de código según la evaluación de una condición.

- Utilizar la sentencia if...else if para evaluar múltiples condiciones en orden y ejecutar el bloque de código correspondiente al primer caso verdadero.
- Aprender a controlar el flujo de un programa mediante la implementación de estructuras condicionales.

📋 Instrucciones

1. Utilizando `typescript` codifica las funciones que se indican en la sección `requerimientos funcionales` de este documento.
2. Las funciones deben ser codificadas en un archivo llamado `app.ts`.
3. Las funciones deben tener el nombre que se indica y el número y tipo de parámetros que se especifican en la sección `requerimientos funcionales`.
4. En el archivo `demo.ts` se deben incluir ejemplos de código que muestren el correcto funcionamiento de las funciones.
5. Las funciones deben ser probadas y ejecutadas utilizando la consola.
6. El código fuente final debe ser almacenado en este repositorio de GitHub.

📥 Entregables

- Código fuente de la solución a los requerimientos planteados en la sección `requerimientos funcionales` en este repositorio

👨‍💻 Requerimientos funcionales

Los requerimientos funcionales se refieren a las acciones específicas que las funciones deben realizar y los tipos de datos que deben recibir y retornar. Las firmas de las funciones muestran la sintaxis correcta para definir las funciones, especificando los nombres y tipos de los parámetros, así como el tipo de dato que retorna cada función.

1️⃣ isPair

Esta función debe recibir un número entero como parámetro y devolver `true` si el número es par o `false` si es impar. Debes utilizar el operador ternario para implementar la función.

```
isPair(number: number): boolean
```

2️⃣ startsWithVowel

Crea una función que determine si una palabra comienza con una vocal. La función debe recibir una cadena de texto como parámetro y devolver `true` si la palabra comienza con una vocal o `false` si no comienza con una vocal.

```
startsWithVowel(word: string): boolean
```

3️⃣ getLongestWord

Escribe una función que reciba como parámetros dos palabras y devuelva la palabra más larga. Si ambas palabras tienen la misma longitud, la función debe devolver la primera palabra.

```
getLongestWord(word1: string, word2: string): string
```

4️⃣ getSeason

Crea una función que determine la estación del año en función del mes. La función debe recibir una fecha en formato Date como parámetro y devolver una cadena de texto con el nombre de la estación correspondiente:

- Enero, febrero y marzo: "Invierno"
- Abril, mayo y junio: "Primavera"
- Julio, agosto y septiembre: "Verano"
- Octubre, noviembre y diciembre: "Otoño"

```
getSeason(date: Date): string
```

5️⃣ calculateShippingCost

Crea una función que calcule el costo de envío de un paquete en función de su peso. Si el peso es menor o igual a 5 kg, el costo es $10; de lo contrario, el costo es de $15.

```
calculateShippingCost(weight: number): number
```

6️⃣ convertGradeToLetter

Crea una función que convierta una calificación numérica en una letra de nota. La función debe recibir un número entero como parámetro y devolver la letra de nota correspondiente según la siguiente escala:

- 90 a 100: "A"
- 80 a 89: "B"
- 70 a 79: "C"
- 0 a 70: "F"

- Si la calificación no está en el rango de 0 a 100, la función debe devolver "Nota inválida".

```
convertGradeToLetter(grade: number): string
```

7️⃣ classifyNumber

Crea una función que clasifique un número en función de su signo. La función debe recibir un número entero como parámetro y devolver una cadena de texto con la clasificación del número:

- Si el número es mayor que cero, la función debe devolver "Positivo".
- Si el número es menor que cero, la función debe devolver "Negativo".
- Si el número es igual a cero, la función debe devolver "Cero".

```
classifyNumber(number: number): string
```

8️⃣ classifyTriangle

Crea una función que clasifique un triángulo en función de la longitud de sus lados. La función debe recibir tres números enteros como parámetros que representan las longitudes de los lados del triángulo y devolver una cadena de texto con la clasificación del triángulo:

- Si todos los lados son iguales, la función debe devolver "Equilátero".
- Si exactamente dos lados son iguales, la función debe devolver "Isósceles".
- Si todos los lados son diferentes, la función debe devolver "Escaleno".

```
classifyTriangle(side1: number, side2: number, side3: number): string
```

9️⃣ classifyAngle

Crea una función que clasifique un ángulo en función de su medida. La función debe recibir un número entero como parámetro que representa la medida del ángulo en grados y devolver una cadena de texto con la clasificación del ángulo:

- Si el ángulo es agudo (menos de 90 grados), la función debe devolver "Ángulo Agudo".
- Si el ángulo es recto (exactamente 90 grados), la función debe devolver "Ángulo Recto".
- Si el ángulo es obtuso (entre 90 y 180 grados), la función debe devolver "Ángulo Obtuso".
- Si el ángulo no está en ninguno de los rangos anteriores, 0 o menor que cero y igual o mayor a 180, la función debe devolver "Ángulo Inválido".

```
classifyAngle(angle: number): string
```

10 calculateDiscount

Crea una función que calcule el descuento aplicado a una compra en función del monto total. La función debe recibir un número entero como parámetro que representa el monto total de la compra y devolver el monto con el descuento aplicado según las siguientes condiciones:

- Si el monto es mayor o igual a $100, se aplica un 10% de descuento.
- Si el monto es mayor o igual a $50 pero menor a $100, se aplica un 5% de descuento.
- Si el monto es menor a $50, no se aplica descuento.

```
calculateDiscount(total: number): number
```

🚀 Ejecución de programas

Para ejecutar un programa utilizar:

```
None
npx ts-node nombre-archivo
```

Por ejemplo:

```
None
npx ts-node demo
```

🚦 Ejecución de pruebas unitarias

Para ejecutar una prueba unitaria utilizar:

```
None
npx jest nombre-de-funcion
```

Por ejemplo:

```
None
npx jest add
```

Comandos de Git y GitHub

Actualizacón del repositorio local

Cada vez que se terminó e actualizar uno o más archivo utilizar, pasar los cambios a `staging` utilizando:

```
None
git add archivo.ext
```

Un `git add` por cada archivo que se actualizó

Una vez que se agregaron los archivo para la nueva versión, confirmar la nueva versión utilizando:

```
None
git commit -m "mensaje"
```

Si al hacer `commit` el linter detecta errores: 1. Corregir los errores, 2. Volver a hacer git add por cada archivo corregido 3. Volver a hacer el commit. Repetir estos 3 pasos hasta que no se obtengan errores por el linter.

Actualización del repositorio remoto

Para enviar las actualizaciones al repositorio remoto utilizar:

```
None
git push origin
```

Recursos

Aquí tienes una lista de recursos recomendados que puedes utilizar como referencia y consulta:

- TypeScript
- TypeScript Deep Dive
- TypeScript in 5 minutes
- MDN JavaScript
- DevDocs
- TypeScript Playground

- Awesome TypeScript

🥷 Ejercicio 08: Sentencias condicionales

Los temas que se estudian al realizar este reto son:

- Sintaxis de las estructuras condicionales en TypeScript.
- El operador ternario como una forma concisa de expresar condiciones.
- La sentencia if y su uso en la toma de decisiones.
- La sentencia if...else para manejar casos alternativos.
- La sentencia if...else if para evaluar múltiples condiciones en orden.
- Cómo controlar el flujo de un programa con condicionales.

🎯 Objetivos de aprendizaje

- Comprender la sintaxis de las estructuras condicionales en TypeScript y su importancia en la programación.
- Utilizar el operador ternario de manera efectiva para expresar condiciones de forma concisa en sus programas.
- Dominar el uso de la sentencia if para tomar decisiones basadas en condiciones específicas.
- Aplicar la sentencia if...else para manejar casos alternativos y ejecutar diferentes bloques de código según la evaluación de una condición.
- Utilizar la sentencia if...else if para evaluar múltiples condiciones en orden y ejecutar el bloque de código correspondiente al primer caso verdadero.
- Aprender a controlar el flujo de un programa mediante la implementación de estructuras condicionales.

📋 Instrucciones

1. Utilizando `typescript` codifica las funciones que se indican en la sección `requerimientos funcionales` de este documento.
2. Las funciones deben ser codificadas en un archivo llamado `app.ts`.
3. Las funciones deben tener el nombre que se indica y el número y tipo de parámetros que se especifican en la sección `requerimientos funcionales`.
4. En el archivo `demo.ts` se deben incluir ejemplos de código que muestren el correcto funcionamiento de las funciones.
5. Las funciones deben ser probadas y ejecutadas utilizando la consola.
6. El código fuente final debe ser almacenado en este repositorio de GitHub.

📥 Entregables

- Código fuente de la solución a los requerimientos planteados en la sección `requerimientos funcionales` en este repositorio

👨‍💻 Requerimientos funcionales

Los requerimientos funcionales se refieren a las acciones específicas que las funciones deben realizar y los tipos de datos que deben recibir y retornar. Las firmas de las funciones muestran la sintaxis correcta para definir las funciones, especificando los nombres y tipos de los parámetros, así como el tipo de dato que retorna cada función.

01 getCost

Escribe una función que determine el costo total de una compra de hamburgesas. La función debe recibir como parámetros la fecha de la compra y el número de hamburgesas compradas. El costo de cada hamburgesa es de $50 pesos. Para determninar el costo total se debe considerar lo siguiente:

1. En enero todos los lunes se aplica un descuento del 5%.
2. En agosto si compra más de 5 hamburgesas se aplica un descuento del 10%
3. En noviembre y diciembre, los jueves si compra más de 10 hamburgesas se aplica una promoción de 4x3.
4. En el resto de los meses si compra más de 3 hamburgesas se aplica un descuento del 5%.

```
getCost(date: Date, quantity: number): number
```

02 getSmallest

Escribe una función que reciba tres números enteros diferentes y regrese como resultado el menor de los tres números.

```
getSmallest(number1: number, number2: number, number3: number): number
```

03 isEligibleForDiscount

Escribe una función que determine si un cliente es elegible para un descuento en una tienda. La función debe recibir como parámetros el total de la compra y si el cliente es miembro de la tienda. Un cliente obtiene un descuento del 20% si el total de la compra es mayor a $1000 o si es miembro de la tienda. Si el total de la compra es menor o igual a $1000 y no es miembro, no tiene descuento.

```
isEligibleForDiscount(total: number, isMember: boolean): boolean
```

04 isLeapYear

Escribe una función que determine si un año es bisiesto. Un año es bisiesto si es divisible entre 4 y no es divisible entre 100, a menos que también sea divisible entre 400.

```
isLeapYear(year: number): boolean
```

05 isValidPassword

Escribe una función que determine si una contraseña es válida. Una contraseña es válida si tiene al menos 8 caracteres y contiene al menos una vocal.

```
isValidPassword(password: string): boolean
```

06 getDiscountAmount

Escribe una función que calcule el descuento aplicado a una compra. La función debe recibir como parámetros el total de la compra y un booleano que indique si el cliente es nuevo. Los clientes nuevos reciben un descuento del 15% o si el total de la compra es mayor a $2000, el descuento es del 20%. Si ninguna de estas condiciones se cumple, no hay descuento.

```
getDiscountAmount(total: number, isNewCustomer: boolean): number
```

📚 Recursos

Aquí tienes una lista de recursos recomendados que puedes utilizar como referencia y consulta:

- TypeScript
- TypeScript Deep Dive
- TypeScript in 5 minutes
- MDN JavaScript
- DevDocs
- TypeScript Playground
- Awesome TypeScript

🥷 Ejercicio 09: Introducción a sentencias repetitivas

Los temas que se estudian al realizar este reto son:

- Sintaxis de las estructuras repetitivas en TypeScript.
- La sentencia for y su uso.
- La sentencia while y su uso.
- La sentencia do...while y su uso.

🎯 Objetivos de aprendizaje

1. Comprender la sintaxis de las estructuras repetitivas en TypeScript y su importancia en la programación.
2. Aprender a utilizar la sentencia `for` para crear ciclos controlados por una variable de contador, permitiendo la ejecución repetida de bloques de código.
3. Dominar el uso de la sentencia `while` para ejecutar ciclos basados en una condición booleana, con la capacidad de repetir el bloque mientras la condición sea verdadera.
4. Utilizar la sentencia `do...while` en TypeScript para crear ciclos que garantizan que el bloque de código se ejecute al menos una vez antes de verificar la condición.
5. Aplicar de manera efectiva las estructuras repetitivas para automatizar tareas y procesar conjuntos de datos en programas TypeScript.
6. Entender cuándo y cómo usar cada tipo de estructura repetitiva según los requisitos específicos de un programa.

📋 Instrucciones

1. Utilizando `typescript` codifica las funciones que se indican en la sección `requerimientos funcionales` de este documento.
2. Las funciones deben ser codificadas en un archivo llamado `app.ts`.
3. Las funciones deben tener el nombre que se indica y el número y tipo de parámetros que se especifican en la sección `requerimientos funcionales`.
4. En el archivo `demo.ts` se deben incluir ejemplos de código que muestren el correcto funcionamiento de las funciones.
5. Las funciones deben ser probadas y ejecutadas utilizando la consola.
6. El código fuente final debe ser almacenado en este repositorio de GitHub.

📥 Entregables

- Código fuente de la solución a los requerimientos planteados en la sección `requerimientos funcionales` en este repositorio

👨‍💻 Requerimientos funcionales

Los requerimientos funcionales se refieren a las acciones específicas que las funciones deben realizar y los tipos de datos que deben recibir y retornar. Las firmas de las funciones muestran la sintaxis correcta para definir las funciones, especificando los nombres y tipos de los parámetros, así como el tipo de dato que retorna cada función.

01 printNumbers

Utilizando el ciclo `for`, escribir una función que imprima los números del 1 al 10.

```
printNumbers(): void
```

printOddNumbers

Utilizando el ciclo `for`, escribir una función que imprima los números impares que hay entre una número a y un número b.

```
printOddNumbers(a: number, b: number): void
```

02 invert

Utilizando el ciclo `for`, escribir una función que reciba dos números, `start` y `end`, y regrese una cadena con los números desde `end` hasta `start` en orden descendente, concatenados en una sola cadena. Por ejemplo, si `start` es 1 y `end` es 5, la función debería regresar "54321".

```
invert(start: number, end: number): string
```

03 countVowels

Utilizando el ciclo `for`, escribir una función que regrese un string que regrese cuantas vocales tiene. Pueden usar la función charAt()

```
countVowels(word: string): string
```

04 countToTen

Utilizando el ciclo `while`, escribir una función que imprima los números del 1 al 10.

```
countToTen(): void
```

05 getSumFrom100

Utilizando el ciclo `while`, escribir una función que regrese la suma de los números que hay entre 100 y n, donde n es siempre un número menor a 100. Por ejemplo si n es 98, la función debería regresar 100 + 99 + 98 = 297.

```
getSumFrom100(n: number): number
```

06 countLetter

Utilizando el ciclo `while`, escribir una función que reciba como parámetros un string y una letra. La función deberá regresar cuántas veces está presente esa letra dentro del string. Por ejemplo, si el string es "hola mundo" y la letra es "o", la función debería regresar 2.

```
countLetter(text: string, letter: string): number
```

07 printToFive

Utilizando el ciclo `do...while`, escribir una función que imprima los números del 1 al 5.

```
printToFive(): void
```

08 printFromAToB

Utilizando el ciclo `do...while`, escribir una función que reciba dos números, `a` y `b`, y regrese una cadena con los números desde `a` hasta `b` en orden ascendente, concatenados en una sola cadena. Por ejemplo, si `a` es 3 y `b` es 7, la función debería regresar "34567".

```
printFromAToB(a: number, b: number): string
```

09 getSum

Utilizando el ciclo `do...while`, escribir una función que regrese la sumatoria de los números que hay entre 1 y n. Por ejemplo, si n = 4 debe regresar 10. Porque 1+2+3+4 = 10

```
getSum(n: number): number
```

10 printMultiply

Utilizando el ciclo `do...while`, escribir una función que imprima la tabla de multiplicar de un número desde 1 hasta 12 como se muestra abajo:

```
None
Por ejemplo, si número = 5 debe imprimir:

5 x 1 = 5
5 x 2 = 10
```

```
5 x 3 = 15
...
5 x 12 = 60
```

```
printMultiply(number: number): void
```

11 fibonacci

Utilizando el ciclo `do...while`, escribir una función que regrese un string con los números de la serie de Fibonacci hasta n. Por ejemplo, si n es 5, la función debería regresar "0, 1, 1, 2, 3".

```
fibonacci(n: number): string
```

🚀 Ejecución de programas

Para ejecutar un programa utilizar:

```
None
npx ts-node nombre-archivo
```

Por ejemplo:

```
None
npx ts-node demo
```

🚦 Ejecución de pruebas unitarias

Para ejecutar una prueba unitaria utilizar:

```
None
npx jest nombre-de-funcion
```

Por ejemplo:

```
None
npx jest add
```

Comandos de Git y GitHub

Actualizacón del repositorio local

Cada vez que se terminó e actualizar uno o más archivo utilizar, pasar los cambios a `staging` utilizando:

```
None
git add archivo.ext
```

Un `git add` por cada archivo que se actualizó

Una vez que se agregaron los archivo para la nueva versión, confirmar la nueva versión utilizando:

```
None
git commit -m "mensaje"
```

Si al hacer `commit` el linter detecta errores: 1. Corregir los errores, 2. Volver a hacer git add por cada archivo corregido 3. Volver a hacer el commit. Repetir estos 3 pasos hasta que no se obtengan errores por el linter.

Actualización del repositorio remoto

Para enviar las actualizaciones al repositorio remoto utilizar:

```
None
git push origin
```

Recursos

Aquí tienes una lista de recursos recomendados que puedes utilizar como referencia y consulta:

- TypeScript
- TypeScript Deep Dive
- TypeScript in 5 minutes
- MDN JavaScript
- DevDocs
- TypeScript Playground
- Awesome TypeScript

🥷 Ejercicio 10: Sentencias repetitivas

Los temas que se estudian al realizar este reto son:

- Sintaxis de las estructuras repetitivas en TypeScript.
- La sentencia for y su uso.
- La sentencia while y su uso.
- La sentencia do...while y su uso.

🎯 Objetivos de aprendizaje

- Comprender la sintaxis de las estructuras repetitivas en TypeScript y su importancia en la programación.
- Aprender a utilizar la sentencia `for` para crear ciclos controlados por una variable de contador, permitiendo la ejecución repetida de bloques de código.
- Dominar el uso de la sentencia `while` para ejecutar ciclos basados en una condición booleana, con la capacidad de repetir el bloque mientras la condición sea verdadera.
- Utilizar la sentencia `do...while` en TypeScript para crear ciclos que garantizan que el bloque de código se ejecute al menos una vez antes de verificar la condición.
- Aplicar de manera efectiva las estructuras repetitivas para automatizar tareas y procesar conjuntos de datos en programas TypeScript.
- Entender cuándo y cómo usar cada tipo de estructura repetitiva según los requisitos específicos de un programa.

📋 Instrucciones

1. Utilizando `typescript` codifica las funciones que se indican en la sección `requerimientos funcionales` de este documento.
2. Las funciones deben ser codificadas en un archivo llamado `app.ts`.
3. Las funciones deben tener el nombre que se indica y el número y tipo de parámetros que se especifican en la sección `requerimientos funcionales`.
4. En el archivo `demo.ts` se deben incluir ejemplos de código que muestren el correcto funcionamiento de las funciones.
5. Las funciones deben ser probadas y ejecutadas utilizando la consola.
6. El código fuente final debe ser almacenado en este repositorio de GitHub.

📥 Entregables

- Código fuente de la solución a los requerimientos planteados en la sección `requerimientos funcionales` en este repositorio

👨‍💻 Requerimientos funcionales

Los requerimientos funcionales se refieren a las acciones específicas que las funciones deben realizar y los tipos de datos que deben recibir y retornar. Las firmas de las funciones muestran la sintaxis correcta para definir las funciones, especificando los nombres y tipos de los parámetros, así como el tipo de dato que retorna cada función.

1️⃣ getSum

Escribe una función que calcule la suma de los números desde el número A hasta el número B utilizando un ciclo `for`. La función debe regresar la suma total. Por ejemplo, si `numberA = 1` y `numberB = 5`, la función debe regresar `15` (1 + 2 + 3 + 4 + 5).

```
getSum(numberA: number, numberB: number): number
```

2️⃣ getSequence

Escribe una función que genere un string con una secuencia de números descendente desde el número A hasta el número B utilizando un ciclo `for`. Si el número A es menor que el número B, la función debe regresar el string `"-1"`. Por ejemplo, si `numberA = 20` y `numberB = 15`, la función debe regresar el string `"20 19 18 17 16 15"`. Pero si `numberA = 15` y `numberB = 20`, la función debe regresar `"-1"`.

```
getSequence(numberA: number, numberB: number): string
```

3️⃣ getEvenSum

Escribe una función que calcule la suma de los números pares entre 1 y N utilizando un ciclo `while`. La función debe regresar la suma total. Por ejemplo, si `N = 10`, la función debe regresar `30` (2 + 4 + 6 + 8 + 10).

```
getEvenSum(N: number): number
```

4️⃣ count5and3

Escribe una función que cuente cuántos números múltiplos de 5 y 3 existen entre 1 y N utilizando un ciclo `while`. La función debe regresar la cantidad total. Por ejemplo, si `N = 30`, la función debe regresar `2` (15, 30).

```
count5and3(n: number): number
```

5️⃣ calculatePower

Escribe una función que tome dos números enteros, `base` y `exponent`, y calcule la potencia de `base` elevado a `exponent` utilizando un ciclo `do..while`. La función debe regresar el resultado de la potencia. Por ejemplo, si `base = 2` y `exponent = 3`, la función debe regresar `8` (2 x 2 x 2).

```
calculatePower(base: number, exponent: number): number
```

6️⃣ countVowels

Escribe una función que reciba una cadena de texto y cuente cuántas vocales (a, e, i, o, u) contiene utilizando un ciclo `do..while`. La función debe regresar la cantidad de vocales encontradas en la cadena. Por ejemplo, si `text = "hello"`, la función debe regresar `2`.

```
countVowels(text: string): number
```

7️⃣ countCharacters

Escribe una función que reciba una cadena de texto y cuente cuántos caracteres contiene, ignorando los espacios, puntos y comas. Usa un ciclo para recorrer la cadena y contar los caracteres. La función debe regresar el número total de caracteres. No está permitido utilizar la función `length` de JavaScript.

```
countCharacters(text: string): number
```

8️⃣ sumDigits

Escribe una función que reciba un número entero y utilice un ciclo para sumar los dígitos del número. Por ejemplo, si el número es `123`, la función debe regresar `6` (porque `1 + 2 + 3 = 6`).

```
sumDigits(number: number): number
```

9️⃣ reverseString

Escribe una función que reciba una cadena de texto y utilice un ciclo para invertir el orden de los caracteres. La función debe regresar la cadena invertida. Por ejemplo, si la cadena es `"hola"`, debe regresar `"aloh"`. No puedes utilizar el método `reverse` de JavaScript.

```
reverseString(text: string): string
```

🔟 factorial

Escribe una función que reciba un número entero positivo y utilice un ciclo para calcular el factorial de ese número. El factorial de un número es el producto de todos

los números enteros desde 1 hasta ese número. Por ejemplo, el factorial de `5` es `5 * 4 * 3 * 2 * 1 = 120`.

```
factorial(n: number): number
```

🚀 Ejecución de programas

Para ejecutar un programa utilizar:

```
None
npx ts-node nombre-archivo
```

Por ejemplo:

```
None
npx ts-node demo
```

🚦 Ejecución de pruebas unitarias

Para ejecutar una prueba unitaria utilizar:

```
None
npx jest nombre-de-funcion
```

Por ejemplo:

```
None
npx jest add
```

Comandos de Git y GitHub

Actualizacón del repositorio local

Cada vez que se terminó e actualizar uno o más archivo utilizar, pasar los cambios a `staging` utilizando:

```
None
git add archivo.ext
```

Un `git add` por cada archivo que se actualizó

Una vez que se agregaron los archivo para la nueva versión, confirmar la nueva versión utilizando:

```
None
git commit -m "mensaje"
```

Important

Si al hacer `commit` el linter detecta errores: 1. Corregir los errores, 2. Volver a hacer git add por cada archivo corregido 3. Volver a hacer el commit. Repetir estos 3 pasos hasta que no se obtengan errores por el linter.

Actualización del repositorio remoto

Para enviar las actualizaciones al repositorio remoto utilizar:

```
None
git push origin
```

📚 Recursos

Aquí tienes una lista de recursos recomendados que puedes utilizar como referencia y consulta:

- TypeScript
- TypeScript Deep Dive
- TypeScript in 5 minutes
- MDN JavaScript
- DevDocs
- TypeScript Playground
- Awesome TypeScript