```typescript
// Sitio web oficial de TypeScript: https://www.typescriptlang.org/ // Comentario de una línea

/*
Comentario de múltiples líneas
*/

// ** Declaración de una variable y una constante **
// TypeScript (con tipo explícito)
let nombre: string = 'Juan'
let edad: number = 30
const PI: number = 3.1416

// TypeScript (con inferencia de tipo)
let apellido = 'Pérez' // TypeScript infiere que 'apellido' es de tipo string
let anios = 35      // TypeScript infiere que 'anios' es de tipo number
const gravedad = 9.8 // TypeScript infiere que 'gravedad' es de tipo number

// ** Tipos de datos primitivos **
// string
let mensaje: string = "Hola mundo"
let saludo: string = `Bienvenido`

// number
let entero: number = 42
let decimal: number = 3.14
let hexadecimal: number = 0xf00d
let binario: number = 0b1010
let octal: number = 0o744

// boolean
let esVerdadero: boolean = true
let esFalso: boolean = false

// null y undefined
let nulo: null = null
let indefinido: undefined = undefined

// symbol (disponible desde ES6)
let simboloUnico: symbol = Symbol("id")

// bigint (para enteros muy grandes)
let numeroGrande: bigint = 9007199254740991n

// Imprimir en consola
console.log("¡Hola, desde TypeScript!")

// ** Operadores y estructuras de control **
let num1: number = 10
let num2: number = 5
let texto1: string = "Hola"
let texto2: string = "Mundo"
let verdadero: boolean = true
let falso: boolean = false

// Operadores aritméticos
let suma: number = num1 + num2
let resta: number = num1 - num2
let multiplicacion: number = num1 * num2
let division: number = num1 / num2
let modulo: number = num1 % num2

// Operadores de asignación
let contador: number = 0
contador += 1
contador++

// Operadores de comparación
let igual: boolean = num1 == num2
let diferente: boolean = num1 != num2
let mayorQue: boolean = num1 > num2
let menorQue: boolean = num1 < num2
let mayorIgualQue: boolean = num1 >= num2
let menorIgualQue: boolean = num1 <= num2
let estrictamenteIgual: boolean = num1 === num2 // Compara valor y tipo
let estrictamenteDiferente: boolean = num1 !== num2 // Compara valor y tipo

// Operadores lógicos
let andLogico: boolean = verdadero && falso
let orLogico: boolean = verdadero || falso
let notLogico: boolean = !verdadero

// Operador condicional (ternario)
let resultado: string = (num1 > num2) ? "Num1 es mayor" : "Num2 es mayor o igual"

// Operador typeof
let tipoDeDatoNum1: string = typeof num1 // Devuelve "number"
let tipoDeDatoTexto1: string = typeof texto1 // Devuelve "string"
let tipoDeDatoVerdadero: string = typeof verdadero // Devuelve "boolean"

// ** Estructuras de control **
let edadPersona: number = 18

if (edadPersona >= 18) {
  console.log("Es mayor de edad")
} else if (edadPersona >= 16) {
```

```
  console.log("Está cerca de ser mayor de edad")
} else {
  console.log("Es menor de edad")
}

// Switch
let diaSemana: number = 3 // 1: Lunes, 2: Martes, ...

switch (diaSemana) {
  case 1:
    console.log("Lunes")
    break
  case 2:
    console.log("Martes")
    break
  case 3:
    console.log("Miércoles")
    break
  default:
    console.log("Otro día")
}

// Bucles
// For
for (let i: number = 0; i < 5; i++) {
  console.log(`Iteración número ${i}`)
}

// While
let contadorWhile: number = 0
while (contadorWhile < 3) {
  console.log(`Contador while: ${contadorWhile}`)
  contadorWhile++
}

// Do While
let contadorDoWhile: number = 0
do {
  console.log(`Contador do-while: ${contadorDoWhile}`)
  contadorDoWhile++
} while (contadorDoWhile < 3)

// For...in
interface Auto {
  marca: string
  modelo: string
  anio: number
}

let miAuto: Auto = {
  marca: "Toyota",
  modelo: "Corolla",
  anio: 2020
}

for (let propiedad in miAuto) {
  console.log(`Propiedad: ${propiedad}, Valor: ${miAuto[propiedad]}`)
}

// For...of
let colores: string[] = ["rojo", "verde", "azul"]

for (let color of colores) {
  console.log(`Color: ${color}`)
}

// Break y continue
for (let i: number = 0; i < 10; i++) {
  if (i === 3) {
    break // Sale del bucle cuando i es 3
  }
  if (i % 2 === 0) {
    continue // Salta a la siguiente iteración si i es par
  }
  console.log(`Número impar: ${i}`)
}

// 1. Números entre 10 y 55 (pares, no 16, no múltiplos de 3):
function imprimirNumerosParesNo16NoMultiplosDe3(): void {
  console.log("Números pares entre 10 y 55 (no 16, no múltiplos de 3):")
  for (let numero: number = 10; numero <= 55; numero++) {
    if (numero % 2 === 0 && numero !== 16 && numero % 3 !== 0) {
      console.log(numero)
    }
  }
}

imprimirNumerosParesNo16NoMultiplosDe3()

/* Salida por consola
Números pares entre 10 y 55 (no 16, no múltiplos de 3):
10
14
20
22
26
```

```
28
32
34
38
40
44
46
50
52

*/

// 2. Números Primos del 1 al 1000:
function esPrimo(numero: number): boolean {
  if (numero <= 1) {
    return false // 1 y números menores no son primos
  }
  for (let i: number = 2; i <= Math.sqrt(numero); i++) {
    if (numero % i === 0) {
      return false // Si es divisible por algún número entre 2 y su raíz cuadrada, no es primo
    }
  }
  return true // Si no se encontró ningún divisor, es primo
}

function imprimirNumerosPrimosHasta1000(): void {
  console.log("Números primos del 1 al 1000:")
  for (let numero: number = 2; numero <= 1000; numero++) {
    if (esPrimo(numero)) {
      console.log(numero)
    }
  }
}

imprimirNumerosPrimosHasta1000()

/* Salida por consola
Números primos del 1 al 1000:
2
3
5
7
11...
*/

// 3. Serie de Fibonacci:
function imprimirSerieFibonacci(cantidadNumeros: number): void {
  if (cantidadNumeros <= 0) {
    return // No hacer nada si se piden 0 o menos números
  }

  let a: number = 0
  let b: number = 1

  console.log("Serie de Fibonacci:")
  console.log(a) // Imprime el primer número (0)

  if (cantidadNumeros > 1) {
    console.log(b) // Imprime el segundo número (1)
  }

  for (let i: number = 2; i < cantidadNumeros; i++) {
    let siguiente: number = a + b
    console.log(siguiente)
    a = b // Actualiza 'a' al valor anterior de 'b'
    b = siguiente // Actualiza 'b' al nuevo valor 'siguiente'
  }
}

// Imprimir los primeros 10 números de la serie de Fibonacci
imprimirSerieFibonacci(10)

/* Salida por consola
Serie de Fibonacci:
0
1
1
2
3
5
8
13
21
34
*/

// ** Funciones **
// Función con tipos en TypeScript
function sumarTS(a: number, b: number): number {
  return a + b
}

let resultadoSuma: number = sumarTS(5, 3) // resultadoSuma será de tipo number
// sumarTS(5, "tres"); // Error! Argumento de tipo string no asignable al parámetro de tipo number

// Expresión de función con tipos
```

```typescript
let multiplicarTS: (a: number, b: number) => number = function (a: number, b: number): number {
  return a * b
}

let resultadoMultiplicacion: number = multiplicarTS(4, 6)

// Con parámetros y argumentos
function saludar(nombre: string, saludoOpcional?: string): string { // 'saludoOpcional' es opcional
  let saludo = saludoOpcional ? saludoOpcional : "Hola"
  return `${saludo}, ${nombre}!`
}

console.log(saludar("Ana")) // Hola, Ana!
console.log(saludar("Pedro", "Buen día")) // Buen día, Pedro!


function sumarVarios(...numeros: number[]): number { // 'numeros' es un array de números (rest parameter)
  let total: number = 0
  for (let num of numeros) {
    total += num
  }
  return total
}

console.log(sumarVarios(1, 2, 3, 4, 5)) // 15

// Valor de retorno
function mostrarMensaje(mensaje: string): void {
  console.log(mensaje)
  // No retorna nada explícitamente
}

// Función de flecha
let restarTS = (a: number, b: number): number => a - b

let resultadoResta: number = restarTS(10, 2)

// Funciones dentro de funciones
function funcionExterna() {
  function funcionInterna() {
    console.log("Soy una función interna")
  }
  funcionInterna() // Llamada a la función interna
}

// Ejercicio opcional:
function funcionCadenaNumero(cadena1: string, cadena2: string): number {
  let contadorNumeros: number = 0

  for (let i: number = 1; i <= 100; i++) {
    let mensaje: string | number = i // Inicializamos con el número por defecto
    let multiploDeTres: boolean = i % 3 === 0
    let multiploDeCinco: boolean = i % 5 === 0

    if (multiploDeTres && multiploDeCinco) {
      mensaje = cadena1 + cadena2
    } else if (multiploDeTres) {
      mensaje = cadena1
    } else if (multiploDeCinco) {
      mensaje = cadena2
    } else {
      contadorNumeros++ // Incrementa el contador si se imprime el número
    }

    console.log(mensaje)
  }

  return contadorNumeros
}

// Ejemplo de uso de la función
let texto1: string = "Múltiplo de Tres"
let texto2: string = "Múltiplo de Cinco"
let cantidadNumerosImpresos: number = funcionCadenaNumero(texto1, texto2)

console.log(`\nCantidad de números impresos en lugar de textos: ${cantidadNumerosImpresos}`)

/* Salida por consola
Múltiplo de Tres
Múltiplo de Cinco
Múltiplo de Tres
Múltiplo de Tres
Múltiplo de Cinco...
*/

// ** Estructuras de datos **
// 1) Arreglos (Arrays)
// 1. Array literal (la forma más común)
let numeros: number[] = [1, 2, 3, 4, 5]
let nombres: string[] = ['Ana', 'Carlos', 'Sofía']
let booleanos: boolean[] = [true, false, true]
let mixto: any[] = [1, 'texto', true, null] // Usar 'any' con moderación

// * Metodos de insercion *
let frutas: string[] = ['manzana', 'banana']

```

```typescript
// push(): Añade al final del array
frutas.push('naranja')
console.log("Después de push:", frutas) // Salida: ['manzana', 'banana', 'naranja']

// unshift(): Añade al inicio del array
frutas.unshift('kiwi')
console.log("Después de unshift:", frutas) // Salida: ['kiwi', 'manzana', 'banana', 'naranja']

// splice(): Añade en una posición específica (y puede eliminar elementos)
frutas.splice(2, 0, 'mango', 'piña') // En la posición 2, elimina 0 elementos, e inserta 'mango' y 'piña'
console.log("Después de splice (inserción):", frutas) // Salida: ['kiwi', 'manzana', 'mango', 'piña', 'banana', 'naranja']

// Acceder por índice y asignar (en realidad es actualización si ya existe el índice, inserción si es al final o más allá, pero puede dejar hu
frutas[frutas.length] = 'fresa' // Añade al final (como push, pero menos común directamente así)
console.log("Después de asignar por índice al final:", frutas) // Salida: ['kiwi', 'manzana', 'mango', 'piña', 'banana', 'naranja', 'fresa']

// * Métodos de eliminación *
let colores: string[] = ['rojo', 'verde', 'azul', 'amarillo', 'morado']

// pop(): Elimina el último elemento y lo retorna
let ultimoColor = colores.pop()
console.log("Después de pop:", colores) // Salida: ['rojo', 'verde', 'azul', 'amarillo']
console.log("Elemento eliminado con pop:", ultimoColor) // Salida: morado

// shift(): Elimina el primer elemento y lo retorna
let primerColor = colores.shift()
console.log("Después de shift:", colores) // Salida: ['verde', 'azul', 'amarillo']
console.log("Elemento eliminado con shift:", primerColor) // Salida: rojo

// splice(): Elimina elementos desde una posición específica
let coloresBorrados = colores.splice(1, 2) // Desde la posición 1, elimina 2 elementos ('azul', 'amarillo')
console.log("Después de splice (borrado):", colores) // Salida: ['verde']
console.log("Elementos borrados con splice:", coloresBorrados) // Salida: ['azul', 'amarillo']

// delete: Elimina un elemento por índice, pero deja un hueco 'empty' (undefined)
delete colores[0]
console.log("Después de delete colores[0]:", colores) // Salida: [empty] (o [undefined] dependiendo del entorno de ejecución)
console.log("Colores[0] después de delete:", colores[0]) // Salida: undefined

// * Métodos de actualización *
let animales: string[] = ['perro', 'gato', 'hamster']

// Acceder por índice y reasignar
animales[1] = 'conejo' // Reemplaza 'gato' con 'conejo'
console.log("Después de actualizar animales[1]:", animales) // Salida: ['perro', 'conejo', 'hamster']

// slice() y assign/spread para actualizar una porción (no es una actualización directa, crea un nuevo array con la modificación)
let parte1 = animales.slice(0, 1) // ['perro']
let parte2 = ['loro', 'canario']
let parte3 = animales.slice(2) // ['hamster']
animales = [...parte1, ...parte2, ...parte3] // Reconstruye 'animales' con la parte modificada
console.log("Después de actualizar con slice y spread:", animales) // Salida: ['perro', 'loro', 'canario', 'hamster']

// * Métodos de ordenación *
let numerosDesordenados: number[] = [5, 2, 8, 1, 9, 4]
let nombresDesordenados: string[] = ['Carlos', 'Ana', 'Beatriz', 'David']

// sort(): Ordena el array
// Por defecto, sort() ordena como strings (unicode point order), así que para números hay que pasar una función de comparación.
numerosDesordenados.sort((a, b) => a - b) // Orden ascendente numérico
console.log("Números ordenados ascendente:", numerosDesordenados) // Salida: [1, 2, 4, 5, 8, 9]

nombresDesordenados.sort() // Orden alfabético (por defecto sort() para strings)
console.log("Nombres ordenados alfabéticamente:", nombresDesordenados) // Salida: ['Ana', 'Beatriz', 'Carlos', 'David']

nombresDesordenados.sort((a, b) => b.localeCompare(a)) // Orden alfabético inverso (con localeCompare para considerar acentos, etc. si fuera n
console.log("Nombres ordenados alfabéticamente inverso:", nombresDesordenados) // Salida: ['David', 'Carlos', 'Beatriz', 'Ana']

// reverse(): Invierte el orden del array
numerosDesordenados.reverse()
console.log("Números invertidos (reverse):", numerosDesordenados) // Salida: [1, 2, 4, 5, 8, 9] (después de ordenar ascendente y luego reverse

// 2) Objectos
// 1. Objeto literal (la forma más común)
let persona: { nombre: string, edad: number, profesion?: string } = {
  nombre: 'Elena',
  edad: 28,
  profesion: 'Desarrolladora' // 'profesion' es opcional según la definición (usando ?)
}

let coche: { marca: string, modelo: string, anio: number } = {
  marca: 'Toyota',
  modelo: 'Corolla',
  anio: 2022
}

let objetoVacio: {} = {} // Objeto vacío

// 3. Clases (para crear objetos con una estructura definida y métodos - más avanzado, pero importante en OOP con TypeScript)
class Animal {
  nombre: string
  tipo: string

  constructor(nombre: string, tipo: string) {
    this.nombre = nombre
    this.tipo = tipo
```

```
  }

  hacerSonido(): void {
    console.log("Sonido genérico de animal")
  }
}

let miAnimal = new Animal('León', 'Mamífero')
console.log("Objeto creado con clase:", miAnimal)
miAnimal.hacerSonido() // Salida: Sonido genérico de animal

// * Inserción *
let libro: { titulo: string, autor: string } = {
  titulo: 'Cien años de soledad',
  autor: 'Gabriel García Márquez'
}

// Dot notation (notación de punto)
libro.genero = 'Realismo mágico' // Añade la propiedad 'genero'
console.log("Después de añadir con dot notation:", libro)
// Salida: {titulo: 'Cien años de soledad', autor: 'Gabriel García Márquez', genero: 'Realismo mágico'}

let propiedadDinamica = 'editorial'
libro[propiedadDinamica] = 'Sudamericana' // Añade propiedad dinámicamente
console.log("Después de añadir propiedad dinámica:", libro)
// Salida: {titulo: 'Cien años de soledad', autor: 'Gabriel García Márquez', genero: 'Realismo mágico', paginas: 496, editorial: 'Sudamericana

// * Métodos de eliminación *
let producto: { nombre: string, precio: number, disponible: boolean } = {
  nombre: 'Laptop',
  precio: 1200,
  disponible: true
}

// delete keyword
delete producto.disponible // Elimina la propiedad 'disponible'
console.log("Después de delete producto.disponible:", producto) // Salida: {nombre: 'Laptop', precio: 1200}

// Asignar 'undefined' -  técnicamente no borra la propiedad, sino que le asigna el valor 'undefined'. La propiedad sigue existiendo, pero sin
producto.nombre = undefined
console.log("Después de asignar producto.nombre = undefined:", producto) // Salida: {nombre: undefined} - propiedad 'nombre' sigue ahí, pero co

// * Métodos de actualización *
let usuario: { nombreUsuario: string, email: string, activo: boolean } = {
  nombreUsuario: 'usuario123',
  email: 'usuario@example.com',
  activo: false
}

// Dot notation
usuario.activo = true // Actualiza la propiedad 'activo'
console.log("Después de actualizar usuario.activo:", usuario) // Salida: {nombreUsuario: 'usuario123', email: 'usuario@example.com', activo: tr

// * Métodos de ordenación *
// Los objectos en JavaScript no tienen un orden específico, pero se pueden ordenar por propiedades al convertirlos a arrays o al usar métodos
// Object.keys(), Object.values() o Object.entries() para obtener las propiedades y luego ordenarlas.

// 3) Mapas (Map)
// 1. Constructor Map - vacío
let mapaVacio = new Map()
console.log("Mapa vacío:", mapaVacio) // Salida: Map(0) {}

// 2. Constructor Map - con inicialización (array de arrays clave-valor)
let mapaInicial = new Map([
  ['nombre', 'Ricardo'],
  ['edad', 42],
  [true, 'activo'],
  [100, 'valor numerico'] // Claves pueden ser de diferentes tipos
])
console.log("Mapa inicializado:", mapaInicial)
// Salida (aproximada, el orden de salida es el de inserción):
// Map(4) {
//   'nombre' => 'Ricardo',
//   'edad' => 42,
//   true => 'activo',
//   100 => 'valor numerico'
// }

// * Métodos de inserción *
let paisesCapitales = new Map()

// set(clave, valor): Inserta o actualiza un par clave-valor
paisesCapitales.set('España', 'Madrid')
console.log("Después de set('España', 'Madrid'):", paisesCapitales) // Salida: Map(1) { 'España' => 'Madrid' }

// * Métodos de eliminación *
let agendaContactos = new Map([
  ['Ana', '123-456-7890'],
  ['Pedro', '987-654-3210'],
  ['Lucía', '555-123-9876']
])

// delete(clave): Elimina la entrada con la clave especificada. Retorna true si la clave existía y fue eliminada, false si no.
let borradoPedro = agendaContactos.delete('Pedro')
console.log("Después de delete('Pedro'):", agendaContactos) // Salida: Map(2) { 'Ana' => '123-456-7890', 'Lucía' => '555-123-9876' }
console.log("¿Se borró Pedro?", borradoPedro) // Salida: true
```

```

// clear(): Elimina todas las entradas del mapa
agendaContactos.clear()
console.log("Después de clear():", agendaContactos) // Salida: Map(0) {} (mapa vacío)

// * Métodos de actualización *
// se realiza con set(clave, nuevoValor) para una clave existente. Si la clave ya existe, set reemplaza el valor asociado a esa clave.

// * Métodos de ordenación *
// Los mapas mantienen el orden de inserción, así que al iterar sobre ellos se respetará el orden en que se añadieron las entradas.

// 4) Conjuntos (Set)
// 1. Constructor Set - vacío
let conjuntoVacio = new Set()
console.log("Conjunto vacío:", conjuntoVacio) // Salida: Set(0) {}

// 2. Constructor Set - con inicialización (iterable, por ejemplo, un array)
let conjuntoInicial = new Set([10, 20, 30, 20, 10]) // Intentamos añadir duplicados
console.log("Conjunto inicializado:", conjuntoInicial) // Salida: Set(3) { 10, 20, 30 } - ¡Duplicados se eliminan!

let conjuntoStrings = new Set('hola mundo') // Iterable es string, añade cada caracter como elemento único
console.log("Conjunto de un string:", conjuntoStrings) // Salida: Set(8) { 'h', 'o', 'l', 'a', ' ', 'm', 'u', 'n', 'd' }

// * Métodos de inserción *
let numerosUnicos = new Set()

// add(valor): Añade un valor al conjunto. Si el valor ya existe, no hace nada (los Sets solo tienen valores únicos)
numerosUnicos.add(10)
console.log("Después de add(10):", numerosUnicos) // Salida: Set(1) { 10 }

numerosUnicos.add(20).add(30).add(20) // Encadeamiento de add.  Intento de añadir 20 de nuevo, no tendrá efecto.
console.log("Después de añadir 20, 30, 20:", numerosUnicos) // Salida: Set(3) { 10, 20, 30 } - Sólo una instancia de cada valor

numerosUnicos.add(NaN)
numerosUnicos.add(NaN) // Intentar añadir NaN duplicado -  Set considera NaN === NaN para la unicidad
console.log("Después de añadir NaN duplicado:", numerosUnicos) // Salida: Set(4) { 10, 20, 30, NaN } -  ¡Solo una instancia de NaN!

// * Métodos de eliminación *
let letrasUnicas = new Set(['a', 'b', 'c', 'd', 'e'])

// delete(valor): Elimina el valor del conjunto. Retorna true si el valor existía y fue eliminado, false si no.
let borradoC = letrasUnicas.delete('c')
console.log("Después de delete('c'):", letrasUnicas) // Salida: Set(4) { 'a', 'b', 'd', 'e' }
console.log("¿Se borró 'c'?", borradoC) // Salida: true

// clear(): Elimina todos los elementos del conjunto
letrasUnicas.clear()
console.log("Después de clear():", letrasUnicas) // Salida: Set(0) {} (conjunto vacío)

// * Métodos de actualización *
// Se realiza con add(valor) para añadir un nuevo valor o actualizar uno existente (si no es único, no se añade)

// * Métodos de ordenación *
// Los conjuntos no tienen un orden específico, pero se pueden convertir a arrays y luego ordenar si es necesario.

// Ejercicio opcional:
import * as readline from 'readline'

interface Contacto {
  nombre: string
  telefono: string
}

const agenda: Map<string, string> = new Map() // Usamos un Map para almacenar contactos (nombre: telefono)

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
})

function mostrarMenu(): void {
  console.log("\n--- Agenda de Contactos ---")
  console.log("1. Buscar contacto")
  console.log("2. Insertar contacto")
  console.log("3. Actualizar contacto")
  console.log("4. Eliminar contacto")
  console.log("5. Listar todos los contactos")
  console.log("6. Finalizar programa")
  console.log("---------------------------")
}

function obtenerInput(pregunta: string): Promise<string> {
  return new Promise((resolve) => {
    rl.question(pregunta, (respuesta) => {
      resolve(respuesta.trim())
    })
  })
}

function validarTelefono(telefono: string): boolean {
  const numeroTelefonoRegex = /^[0-9]+$/ // Regex para solo números
  return numeroTelefonoRegex.test(telefono) && telefono.length <= 11
}

async function buscarContacto(): Promise<void> {
  const nombreBuscar = await obtenerInput("Introduce el nombre del contacto a buscar: ")
```

```
  const telefono = agenda.get(nombreBuscar)
  if (telefono) {
    console.log(`\nContacto encontrado:`)
    console.log(`Nombre: ${nombreBuscar}, Teléfono: ${telefono}`)
  } else {
    console.log(`\nContacto "${nombreBuscar}" no encontrado en la agenda.`)
  }
}

async function insertarContacto(): Promise<void> {
  const nombre = await obtenerInput("Introduce el nombre del nuevo contacto: ")
  let telefono = await obtenerInput("Introduce el teléfono del nuevo contacto (máximo 11 dígitos numéricos): ")

  if (!validarTelefono(telefono)) {
    console.log("Error: Teléfono no válido. Debe ser numérico y de máximo 11 dígitos.")
    return
  }

  if (agenda.has(nombre)) {
    console.log(`Error: El contacto "${nombre}" ya existe en la agenda. Utiliza 'Actualizar' para modificar el teléfono.`)
    return
  }

  agenda.set(nombre, telefono)
  console.log(`\nContacto "${nombre}" insertado correctamente en la agenda.`)
}

async function actualizarContacto(): Promise<void> {
  const nombreActualizar = await obtenerInput("Introduce el nombre del contacto a actualizar: ")

  if (!agenda.has(nombreActualizar)) {
    console.log(`Error: El contacto "${nombreActualizar}" no existe en la agenda.`)
    return
  }

  let nuevoTelefono = await obtenerInput(`Introduce el nuevo teléfono para "${nombreActualizar}" (máximo 11 dígitos numéricos): `)
  if (!validarTelefono(nuevoTelefono)) {
    console.log("Error: Teléfono no válido. Debe ser numérico y de máximo 11 dígitos.")
    return
  }

  agenda.set(nombreActualizar, nuevoTelefono)
  console.log(`\nContacto "${nombreActualizar}" actualizado correctamente.`)
}

async function eliminarContacto(): Promise<void> {
  const nombreEliminar = await obtenerInput("Introduce el nombre del contacto a eliminar: ")

  if (agenda.has(nombreEliminar)) {
    agenda.delete(nombreEliminar)
    console.log(`\nContacto "${nombreEliminar}" eliminado correctamente de la agenda.`)
  } else {
    console.log(`\nContacto "${nombreEliminar}" no encontrado en la agenda.`)
  }
}

async function listarContactos(): Promise<void> {
  if (agenda.size === 0) {
    console.log("\nLa agenda está vacía.")
  } else {
    console.log("\n--- Lista de Contactos ---")
    agenda.forEach((telefono, nombre) => {
      console.log(`Nombre: ${nombre}, Teléfono: ${telefono}`)
    })
    console.log("--------------------------")
  }
}

async function main(): Promise<void> {
  let continuar = true

  while (continuar) {
    mostrarMenu()
    const opcion = await obtenerInput("Selecciona una opción (1-6): ")

    const opciones: { [key: string]: () => Promise<void> } = {
      '1': buscarContacto,
      '2': insertarContacto,
      '3': actualizarContacto,
      '4': eliminarContacto,
      '5': listarContactos,
      '6': async () => {
        console.log("Finalizando programa. ¡Hasta luego!")
        continuar = false
        rl.close()
      },
    }

    if (opciones[opcion]) {
      await opciones[opcion]()
    } else {
      console.log("Opción no válida. Por favor, selecciona una opción del 1 al 6.")
    }
  }
}
```

```
main()

// 5) Cadena de caracteres
let nombreUsuario: string = "Alice"
let saludoPersonalizado: string = `Hola, ${nombreUsuario}! Bienvenida.` // Interpolación de variable
let operacion: string = `2 + 2 = ${2 + 2}` // Interpolación de expresión
console.log(saludoPersonalizado) // Imprime: Hola, Alice! Bienvenida.
console.log(operacion) // Imprime: 2 + 2 = 4

// 6) Valor y Referencia
// * primitivos *
let nombreUsuario: string = "Alice"
let saludoPersonalizado: string = `Hola, ${nombreUsuario}! Bienvenida.` // Interpolación de variable
let operacion: string = `2 + 2 = ${2 + 2}` // Interpolación de expresión
console.log(saludoPersonalizado) // Imprime: Hola, Alice! Bienvenida.
console.log(operacion) // Imprime: 2 + 2 = 4
// * objetos *
let objeto1: { valor: number } = { valor: 5 }
let objeto2: { valor: number } = objeto1 // objeto2 recibe una COPIA de la REFERENCIA a objeto1

objeto2.valor = 10 // Cambiar objeto2.valor SÍ afecta a objeto1.valor, ¡porque ambos apuntan al mismo objeto!

console.log("objeto1.valor:", objeto1.valor) // Imprime: objeto1.valor: 10 (¡CAMBIADO!)
console.log("objeto2.valor:", objeto2.valor) // Imprime: objeto2.valor: 10

let array1: number[] = [1, 2, 3]
let array2: number[] = array1 // array2 recibe una COPIA de la REFERENCIA a array1

array2.push(4) // Modificar array2 SÍ afecta a array1, ¡ambos refieren al mismo array!

console.log("array1:", array1) // Imprime: array1: [ 1, 2, 3, 4 ] (¡CAMBIADO!)
console.log("array2:", array2) // Imprime: array2: [ 1, 2, 3, 4 ]

// 7)  Recursividad
function factorialRecursivo(n: number): number {
  if (n === 0) { // Caso base: factorial de 0 es 1 (condición de parada para evitar bucle infinito)
    return 1
  } else { // Paso recursivo: factorial de n es n * factorial de (n-1)
    return n * factorialRecursivo(n - 1) // La función se llama a sí misma con un problema más pequeño (n-1)
  }
}

console.log("Factorial de 5:", factorialRecursivo(5)) // Imprime: Factorial de 5: 120 (5*4*3*2*1)
console.log("Factorial de 0:", factorialRecursivo(0)) // Imprime: Factorial de 0: 1

// 8)  Pilas y colas
class Pila<T> { // Usamos un genérico <T> para que la pila pueda ser de cualquier tipo
  private elementos: T[] = []; // Usamos un array privado para almacenar los elementos

  push(elemento: T): void {
    this.elementos.push(elemento) // Añadir al final del array (cima de la pila)
  }

  pop(): T | undefined {
    return this.elementos.pop() // Remover y retornar el último elemento (cima de la pila)
  }

  peek(): T | undefined {
    return this.elementos[this.elementos.length - 1] // Ver el último elemento sin removerlo
  }

  isEmpty(): boolean {
    return this.elementos.length === 0
  }

  size(): number {
    return this.elementos.length
  }
}

// Ejemplo de uso de la Pila
let pilaNumeros = new Pila<number>()

pilaNumeros.push(10)
pilaNumeros.push(20)
pilaNumeros.push(30)

console.log("Pila después de push:", pilaNumeros) // No se imprime la pila directamente de forma bonita, pero se puede ver su estado en la cons
console.log("Tamaño de la pila:", pilaNumeros.size()) // 3
console.log("Cima de la pila (peek):", pilaNumeros.peek()) // 30
console.log("Pop:", pilaNumeros.pop()) // 30 (y lo remueve)
console.log("¿Está vacía?:", pilaNumeros.isEmpty()) // false

while (!pilaNumeros.isEmpty()) {
  console.log("Desapilando:", pilaNumeros.pop()) // Desapila 20, luego 10
}

console.log("¿Está vacía después de desapilar todo?:", pilaNumeros.isEmpty()) // true

// 8) Clases
class Animal {
  nombre: string // Propiedad pública 'nombre'
  private edad: number // Propiedad privada 'edad' (solo accesible dentro de la clase)
  protected especie: string // Propiedad protegida 'especie' (accesible en la clase y subclases)

  constructor(nombre: string, edad: number, especie: string) { // Constructor: se ejecuta al crear una nueva instancia
```

```
    this.nombre = nombre
    this.edad = edad
    this.especie = especie
  }

  hacerSonido(): void { // Método público
    console.log("Sonido genérico de animal")
  }

  getEdad(): number { // Método público para acceder a la propiedad privada 'edad' (getter)
    return this.edad
  }

  // Método estático (se asocia a la clase, no a las instancias)
  static crearAnimalDomestico(nombre: string, especie: string): Animal {
    return new Animal(nombre, 0, especie) // Edad inicial 0 para domésticos por ejemplo
  }
}

// 10) Herencia y Polimorfismo
// Clase base (superclase) Animal (ya definida anteriormente)
class Animal { /* ... (definición de la clase Animal como antes) ... */ }

// Subclase Perro que hereda de Animal
class Perro extends Animal { // 'extends Animal' indica herencia
  raza: string // Propiedad adicional específica de Perro

  constructor(nombre: string, edad: number, raza: string) {
    super(nombre, edad, "Canino") // Llamada a super() para invocar el constructor de la superclase (Animal) y pasarle nombre, edad, especie="
    this.raza = raza // Inicializar la propiedad específica de Perro
  }

  ladrar(): void { // Método específico de Perro
    console.log("¡Guau Guau!")
  }

  // Override del método hacerSonido() de la superclase Animal (Polimorfismo en acción)
  hacerSonido(): void {
    console.log("¡Guau! Soy un perro llamado " + this.nombre) // Personalizar el sonido para Perro
  }
}

// Subclase Gato que hereda de Animal
class Gato extends Animal {
  tipoPelo: string

  constructor(nombre: string, edad: number, tipoPelo: string) {
    super(nombre, edad, "Felino") // Llamada a super() para constructor de Animal
    this.tipoPelo = tipoPelo
  }

  maullar(): void {
    console.log("¡Miau Miau!")
  }

  // Override de hacerSonido() para Gato
  hacerSonido(): void {
    console.log(`¡Miau! Soy un gato llamado ${this.nombre}`)
  }
}

// Crear instancias de las subclases
let miPerroHerencia = new Perro("Buddy", 5, "Labrador")
let miGatoHerencia = new Gato("Whiskers", 2, "Persa")

console.log("Perro Nombre:", miPerroHerencia.nombre) // Hereda 'nombre' de Animal
console.log("Perro Raza:", miPerroHerencia.raza) // Propiedad propia de Perro
miPerroHerencia.ladrar() // Método propio de Perro
miPerroHerencia.hacerSonido() // Método overridden de Animal, comportamiento de Perro
console.log("Gato Especie (heredada):", miGatoHerencia.especie) // Hereda 'especie' de Animal
miGatoHerencia.maullar() // Método propio de Gato
miGatoHerencia.hacerSonido() // Método overridden de Animal, comportamiento de Gato

// 11) Excepciones
function dividir(a: number, b: number): number {
  if (b === 0) {
    throw new Error("¡División por cero!") // Lanzar una excepción (Error) si b es 0
  }
  return a / b
}

try { // Bloque try: código que puede lanzar una excepción
  let resultadoDivision = dividir(10, 2)
  console.log("Resultado de la división:", resultadoDivision) // Si no hay excepción, se ejecuta esto

  let divisionPorCero = dividir(5, 0) // ¡Esto lanzará una excepción!
  console.log("Esto NO se imprimirá si hay excepción") // No se llega a ejecutar si la línea anterior lanza excepción

} catch (error: any) { // Bloque catch: se ejecuta SOLO si ocurre una excepción en el bloque try
  console.error("¡Ocurrió un error al dividir!", error.message) // Captura la excepción y maneja el error
  // 'error' es la excepción capturada, y 'error.message' (típicamente) contiene un mensaje de error descriptivo
} finally { // Bloque finally (opcional): se ejecuta SIEMPRE, haya o no excepción en el try, y después del catch si hubo excepción
  console.log("Bloque finally: Siempre se ejecuta, independientemente de si hubo error o no.")
}

console.log("El programa continúa después del bloque try...catch...finally.") // El programa sigue ejecutándose
```

```

// 12)  Manejo ficheros
import * as fs from 'fs' // Importar el módulo 'fs' de Node.js

const nombreArchivo = 'mi-archivo.txt'
const contenido = 'Este es el contenido que quiero escribir en el archivo.\n¡Segunda línea!'

// 1. Escritura de un archivo (asíncrono - no bloqueante)
fs.writeFile(nombreArchivo, contenido, (error) => {
  if (error) {
    console.error("Error al escribir el archivo:", error)
  } else {
    console.log(`Archivo "${nombreArchivo}" escrito correctamente.`)

    // 2. Lectura del archivo (asíncrona) después de escribir (dentro del callback de writeFile)
    fs.readFile(nombreArchivo, 'utf8', (errorLectura, datos) => {
      if (errorLectura) {
        console.error("Error al leer el archivo:", errorLectura)
      } else {
        console.log("\nContenido del archivo leído:")
        console.log(datos) // Imprime el contenido del archivo
      }
    })
  }
})

console.log("Esta línea se ejecuta ANTES de que se complete la escritura/lectura del archivo (asíncrono).")

// 13)  Json y Xml
// * JSON *
const jsonString = `
{
  "nombre": "Producto JSON",
  "precio": 19.95,
  "enStock": true
}
`

try {
  const objetoJSON = JSON.parse(jsonString) // Parsear la cadena JSON a un objeto JavaScript
  console.log("Objeto JSON parseado:", objetoJSON)
  console.log("Nombre del producto:", objetoJSON.nombre) // Acceder a propiedades del objeto
  console.log("Precio:", objetoJSON.precio)
  console.log("¿En stock?:", objetoJSON.enStock)

  // TypeScript inferirá el tipo del objeto como 'any' por defecto, pero podemos tiparlo más si conocemos la estructura
  interface ProductoJSON {
    nombre: string
    precio: number
    enStock: boolean
  }

  const objetoJSONTipado = JSON.parse(jsonString) as ProductoJSON // Parsear y asertar tipo
  console.log("Objeto JSON parseado y tipado:", objetoJSONTipado)
  console.log("Nombre del producto (tipado):", objetoJSONTipado.nombre) // Acceso tipado
} catch (error) {
  console.error("Error al parsear JSON:", error) // Manejar errores de parsing
}
// * XML *
import * as xmlParserLib from 'xml-parser-library'

const xmlString = `
<producto>
  <nombre>Producto XML Ejemplo</nombre>
  <precio>29.99</precio>
</producto>
`

// Parsear XML a Objeto
try {
  const objetoXML = xmlParserLib.parse(xmlString) // Parsear cadena XML a objeto JavaScript
  console.log("Objeto XML parseado:", objetoXML)
  console.log("Nombre del producto XML:", objetoXML.producto.nombre) // Acceder a datos (estructura depende de librería)
} catch (error) {
  console.error("Error al parsear XML:", error)
}

// Convertir Objeto a XML string
const objetoParaXML = {
  producto: {
    nombre: "Otro Producto",
    precio: 45.50
  }
}

try {
  const cadenaXMLResultante = xmlParserLib.stringify(objetoParaXML, { indent: '  ' }) // Convertir objeto a XML string (opciones de formato)
  console.log("\nCadena XML creada:")
  console.log(cadenaXMLResultante)
} catch (error) {
  console.error("Error al convertir a XML string:", error)
}

// 14) Pruebas Unitarias
// calculadora.ts
function sumar(a: number, b: number): number {
```

```
  return a + b
}

function multiplicar(a: number, b: number): number {
  return a * b
}

describe('Calculadora', () => { // 'describe' define un grupo de pruebas (suite)

  test('sumar dos números positivos', () => { // 'test' define un caso de prueba individual
    expect(sumar(2, 3)).toBe(5) // 'expect' es una aserción: esperamos que sumar(2, 3) sea igual a 5 ('toBe' es el matcher de igualdad)
  })

  test('sumar un positivo y un negativo', () => {
    expect(sumar(5, -2)).toBe(3)
  })

  test('multiplicar dos números', () => {
    expect(multiplicar(4, 3)).toBe(12)
  })

  test('multiplicar por cero', () => {
    expect(multiplicar(10, 0)).toBe(0)
  })
})

// 15) fechas
// 1. Fecha y hora actual
let ahora: Date = new Date() // Crea un objeto Date con la fecha y hora actual del sistema
console.log("Fecha y hora actual:", ahora)

// 2. Fecha específica (año, mes, día - ¡meses en JS son base 0: 0=Enero, 1=Febrero, ..., 11=Diciembre!)
let fechaEspecifica: Date = new Date(2024, 2, 15) // 15 de Marzo de 2024 (mes 2 = Marzo)
console.log("Fecha específica (Y, M, D):", fechaEspecifica)

// 3. Fecha y hora específica (año, mes, día, horas, minutos, segundos, milisegundos)
let fechaHoraEspecifica: Date = new Date(2024, 2, 15, 10, 30, 0, 0) // 15 de Marzo de 2024, 10:30:00
console.log("Fecha y hora específica:", fechaHoraEspecifica)

// 4. Fecha a partir de timestamp (milisegundos desde 1 de Enero de 1970 UTC)
let timestamp: number = 1708000000000 // Ejemplo de timestamp
let fechaDesdeTimestamp: Date = new Date(timestamp)
console.log("Fecha desde timestamp:", fechaDesdeTimestamp)

// 5. Fecha a partir de cadena ISO 8601 (formato estándar para fechas en strings)
let fechaISOString: string = "2024-03-15T10:30:00Z" // Formato ISO 8601 en UTC
let fechaDesdeISO: Date = new Date(fechaISOString)
console.log("Fecha desde ISO string:", fechaDesdeISO)

// 16) Asincronia
console.log("Inicio del programa")

// 1. Ejemplo con setTimeout y callback (más tradicional)
setTimeout(function () { // setTimeout es una función asíncrona.  El callback se ejecutará DESPUÉS de 2 segundos
  console.log("--- Callback de setTimeout ejecutado DESPUÉS de 2 segundos ---")
}, 2000) // 2000 milisegundos = 2 segundos

console.log("Esta línea se imprime INMEDIATAMENTE después del inicio de setTimeout, NO espera a los 2 segundos (asincronía con callback).")

// 2. Ejemplo con Promise (mejora la asincronía con callbacks)
function esperarConPromesa(ms: number): Promise<void> {
  return new Promise(resolve => { // 'resolve' es una función que debes llamar para indicar que la promesa se completó exitosamente
    setTimeout(() => {
      console.log(`--- Promesa resuelta DESPUÉS de ${ms / 1000} segundos ---`)
      resolve() // Resolver la promesa cuando el temporizador termine
    }, ms)
  })
}

esperarConPromesa(3000).then(() => { // '.then()' se ejecuta cuando la promesa se resuelve (éxito)
  console.log("Código después de que la promesa se resolvió (con .then())")
})

console.log("Esta línea también se imprime INMEDIATAMENTE después de iniciar esperarConPromesa (asincronía con Promise).")

// 3. Ejemplo con async/await (sintaxis aún más clara para asincronía con Promises)
async function ejecutarConAwait() { // 'async' indica que esta función maneja operaciones asíncronas
  console.log("Función async 'ejecutarConAwait' iniciada.")
  await esperarConPromesa(4000) // 'await' PAUSA la ejecución de ESTA FUNCIÓN hasta que la promesa esperarConPromesa(4000) se resuelva
  console.log("--- 'await' en 'ejecutarConAwait' terminó, y ahora se ejecuta esta línea DESPUÉS de 4 segundos ---")
  console.log("Función async 'ejecutarConAwait' finalizada.")
}

ejecutarConAwait() // Llamar a la función async

console.log("Esta línea se imprime INMEDIATAMENTE después de llamar a ejecutarConAwait (pero la función async sigue ejecutándose en 'segundo pl

console.log("Fin del programa principal (síncrono).  Las operaciones asíncronas continúan ejecutándose en 'segundo plano'.")

// 17) Expresiones Regulares
// Buscar la palabra "hola" (case-sensitive)
const patronHola = /hola/

// Buscar "hola" o "Hola" o "HOLA" (case-insensitive) - flag 'i'
const patronHolaInsensible = /hola/i

```

```
// Buscar la palabra "mundo" al final de la cadena - anchor '<span class="math-inline">'
const patronMundoFin \= /mundo</span >/

// Buscar "número" seguido de uno o más dígitos - carácter especial '\d' (dígito), '+' (uno o más)
const patronNumeroSeguidoDeDigitos = /número\d+/

// Buscar cualquier caracter '.' (punto) - hay que escaparlo con '\.' porque '.' tiene significado especial en regex
const patronPuntoLiteral = /\./

// Buscar un espacio en blanco ' ' - caracter especial '\s' (whitespace)
const patronEspacioBlanco = /\s/

// Buscar una de varias opciones (o) - '|' (pipe)
const patronRojoVerdeAzul = /rojo|verde|azul/

// Agrupar partes del patrón con paréntesis '()'
const patronGrupo = /(ab)+/ // Busca una o más repeticiones de "ab"

// 18) Iteraciones
// * Iterador *
class ContadorPersonalizado {
  private valorInicial: number
  private valorFinal: number
  private valorActual: number

  constructor(inicio: number, fin: number) {
    this.valorInicial = inicio
    this.valorFinal = fin
    this.valorActual = inicio
  }

  // Implementar el método Symbol.iterator para hacer la clase iterable
  [Symbol.iterator]() {
    return this // El iterador es la propia instancia de ContadorPersonalizado en este caso
  }

  // Método next() requerido por el protocolo de iterador
  next(): IteratorResult<number> {
    if (this.valorActual <= this.valorFinal) {
      return { value: this.valorActual++, done: false } // Retornar el valor actual e indicar que NO está terminado
    } else {
      return { value: undefined, done: true } // Indicar que NO hay más valores (done: true)
    }
  }
}

// Usar el iterador personalizado con for...of
const contador = new ContadorPersonalizado(1, 5)

console.log("Iterando con for...of sobre ContadorPersonalizado:")
for (const numero of contador) { // for...of usa el iterador Symbol.iterator() de 'contador' para recorrerlo
  console.log(numero) // Imprime 1, 2, 3, 4, 5
}

// Usar el iterador explícitamente (menos común directamente así, pero para entender el concepto)
const iteradorContador = contador[Symbol.iterator]() // Obtener el iterador del objeto

let resultadoIteracion = iteradorContador.next()
while (!resultadoIteracion.done) { // Mientras 'done' sea false (hay más valores)
  console.log("Iterador next():", resultadoIteracion.value) // Imprimir el valor actual
  resultadoIteracion = iteradorContador.next() // Obtener el siguiente valor
}

// * Generador *
function* generadorNumerosParesHasta(limite: number): Generator<number, void, unknown> {
  for (let i = 0; i <= limite; i += 2) {
    if (i % 2 === 0) {
      yield i // 'yield' cede el valor 'i' y pausa la ejecución del generador
    }
  }
  // Al terminar el bucle, el generador termina y no cede más valores (retorna 'done: true' en next())
}

// Obtener el objeto generador llamando a la función generadora
const generadorPares = generadorNumerosParesHasta(10)

console.log("Usando for...of para iterar sobre el generador:")
for (const numeroPar of generadorPares) { // for...of itera sobre los valores cedidos por el generador
  console.log(numeroPar) // Imprime 0, 2, 4, 6, 8, 10
}

// Usar el generador explícitamente con next()
const otroGeneradorPares = generadorNumerosParesHasta(6)

console.log("\nUsando next() explícitamente para el generador:")
let resultadoGenerador = otroGeneradorPares.next() // Iniciar la ejecución del generador
while (!resultadoGenerador.done) { // Mientras 'done' sea false (el generador ha cedido un valor)
  console.log("Generador yield:", resultadoGenerador.value) // Imprimir el valor cedido
  resultadoGenerador = otroGeneradorPares.next() // Continuar la ejecución del generador hasta el siguiente 'yield'
}

// 19) Conjuntos
new Set([1, 2, 3, 4, 5]) // Crear un conjunto con valores únicos (sin duplicados)

// 20) Enumeraciones
// Enum numérico (por defecto, los valores se incrementan automáticamente desde 0)
```

```
enum EstadoPedido {
  Pendiente,   // Valor implícito 0
  EnProceso,  // Valor implícito 1
  Enviado,    // Valor implícito 2
  Entregado,  // Valor implícito 3
  Cancelado   // Valor implícito 4
}

// Enum numérico con valores iniciales personalizados
enum CodigosErrorHTTP {
  OK = 200,
  NotFound = 404,
  ServerError = 500,
  Unauthorized = 401
}

// Enum de cadena (string enums) - cada miembro debe ser inicializado con un valor string literal
enum DireccionesCardinales {
  Norte = "NORTE",
  Sur = "SUR",
  Este = "ESTE",
  Oeste = "OESTE"
}

// 21) Peticiones HTTP
async function obtenerDatosDeAPI() {
  const url = 'https://jsonplaceholder.typicode.com/todos/1' // Ejemplo de API REST pública para pruebas

  try {
    const respuesta = await fetch(url) // Realizar petición GET asíncrona a la URL. 'await' espera a que la promesa fetch se resuelva.

    if (!respuesta.ok) { // Verificar si la respuesta HTTP fue exitosa (código de estado 2xx)
      throw new Error(`Error HTTP: ${respuesta.status} ${respuesta.statusText}`) // Lanzar un error si la respuesta no es ok
    }

    const datosJSON = await respuesta.json() // Parsear el cuerpo de la respuesta como JSON (también es asíncrono, usa 'await')
    console.log("Datos recibidos de la API:")
    console.log(datosJSON) // Imprimir los datos JSON recibidos

    // Tipado de los datos JSON esperados (opcional, pero recomendado en TypeScript)
    interface TodoItem {
      userId: number
      id: number
      title: string
      completed: boolean
    }

    const datosTipados = datosJSON as TodoItem // Asertar el tipo de los datos JSON
    console.log("\nDatos tipados:")
    console.log(`Título de la tarea: ${datosTipados.title}`)
    console.log(`¿Completada?: ${datosTipados.completed}`)

  } catch (error: any) { // Capturar errores que puedan ocurrir durante la petición (ej., error de red, error al parsear JSON, error HTTP no-o
    console.error("Error al hacer la petición a la API:", error.message)
  }
}

obtenerDatosDeAPI() // Llamar a la función para realizar la petición HTTP

// 22) CallBacks
function operacionAsincrona(callback: (resultado: string) => void) {
  setTimeout(() => {
    const resultadoOp = "Operación asíncrona completada"
    callback(resultadoOp) // Llamar al callback cuando la operación asíncrona termina, pasando el resultado
  }, 1500) // Simular una operación que tarda 1.5 segundos
}

console.log("Iniciando operación asíncrona...")
operacionAsincrona(resultado => { // Pasar una función callback a operacionAsincrona
  console.log("Callback ejecutado con resultado:", resultado) // El callback se ejecuta DESPUÉS de 1.5 segundos
})
console.log("Esta línea se ejecuta inmediatamente después de llamar a operacionAsincrona (sin esperar al callback).")
```