GUÍA DE PROGRAMACIÓN EN UN APOCALIPSIS ZOMBI

JOSÉ ANTONIO FUENTES SANTIAGO

Guía de Programación en un Apocalipsis Zombi (Demo)

José Antonio Fuentes Santiago

Bloque 1. Fundamentos de programación

Ahora...
Primero, fue un aviso en los móviles.
Luego, los gritos en la calle.
Después, el silencio.

En apenas unas horas, el mundo que conocías dejó de funcionar. Las calles están vacías, los teléfonos no suenan, y algo se mueve al otro lado de la ciudad… algo que no camina como tú. Algo que no piensa como tú. Algo que nunca se detiene…

El aviso era claro, había que abandonar las zonas más pobladas y buscar un refugio lejos de las multitudes.

Aún estás en tu casa. Echas un vistazo a tu alrededor y ves comida enlatada, tus viejos apuntes, una linterna medio descargada y un portátil antiguo que, milagrosamente, sigue funcionando. Es ahora o nunca. Vas a necesitar conocimiento, ingenio, y una mochila bien preparada.

No sabes cuánto tiempo estarás ahí fuera. Pero sabes una cosa: sobrevivirás. Y para eso, necesitas entender cómo funciona este nuevo mundo.

Hoy, aprenderás a revisar cada rincón de tu mochila: desde cómo anotar lo importante, clasificar los suministros y organizar tus recursos, hasta interpretar señales del entorno y tomar tus primeras decisiones difíciles. Incluso habrá tiempo para reagrupar tus herramientas y dar los últimos retoques antes de salir por la puerta.

En este bloque...
Verás como puedes dejar preparada la mochila para abandonar la seguridad de tu casa… y enfrentarte al mundo exterior.

- **Capítulo 1. Los Comentarios**: Aprende a documentar tu código como si fuera tu diario de supervivencia.
- **Capítulo 2. Variables y constantes**: Descubre cómo almacenar y organizar información en tu programa.
- **Capítulo 3. Tipos de datos primitivos**: Conoce los diferentes tipos de datos básicos y cómo usarlos.
- **Capítulo 4. Operadores**: Domina las operaciones básicas para manipular datos y tomar decisiones.

- **Capítulo 5. Condicionales**: Implementa la lógica para tomar decisiones en tu código.
- **Capítulo 6. Bucles**: Aprende a repetir acciones de forma eficiente.
- **Desafío 1. Simulador de supervivencia de 7 Días**: Pon a prueba todo lo aprendido.
- **Resumen del bloque 1**: Repasa los conceptos clave.

Capítulo 1. Los comentarios. Anotando lo que necesitas

Antes de salir al exterior y enfrentarte a las criaturas que acechan en cada rincón de la ciudad, cualquier superviviente con un poco de sensatez prepara su libreta y comienza a escribir información vital: rutas de escape, cómo funcionan las herramientas que podrían salvarte la vida, patrones de comportamiento zombi que has observado, etc. Cada anotación podría marcar la diferencia entre sobrevivir o convertirte en una de esas cosas.

Los **comentarios** son pequeñas notas que te ayudan a saber qué hace cada parte del código, como si apuntaras recordatorios en la tapa de tu libreta de campo. Éstos, son ignorados por el transpilador, intérprete o compilador (según el tipo de lenguaje de programación como vimos antes), por lo que sólo te sirven a ti.

Comentarios de una línea

Los comentarios de una línea comienzan por `//`, por lo que todo lo que escribas hasta el final de ésta pertenecerá al comentario.

```
// Declaramos nuestros recursos iniciales
let totalRaciones: number = 24; // raciones totales disponibles
let racionesAlDia: number = 3; // consumo diario por persona

// Calcula el número de días con provisiones
let diasProvisiones: number = totalRaciones / racionesAlDia;
```

Comentarios multi-línea

Por otro lado, puedes añadir comentarios que ocupen varias líneas. Este tipo de comentarios comienza por `/*` y termina con `*/`. Todo el texto que incluyas entre la apertura y el cierre será tratado como parte del comentario.

```
/*
 * Función encargada de comprobar si una zona es segura.
 * - Recibe las coordenadas de la zona a comprobar.
 * - Devuelve un indicador de si la zona es segura (`true`) o no (`false`).
 */
function esZonaSegura(coordenadas: string): boolean {
    // Simulación: zonas que empiecen con "A" son seguras
    return coordenadas.startsWith("A");
}
```

Comentarios JSDoc

Hay una serie de comentarios especiales que puedes usar con TypeScript y JavaScript, que se muestran como ayuda contextual en la mayoría de editores de código y se exportan a distintos formatos de fichero de documentación. A este tipo de comentarios se les llama JSDoc y siguen una serie de normas que te ayudarán a documentar adecuadamente tu código.

Estos comentarios comienzan por `/**` y terminan como cualquier otro comentario de múltiples líneas (aunque también se pueden cerrar com `**/`).

```
/**
 * Función encargada de comprobar si una zona es segura.
 * @param {string} coordenadas Coordenadas de la zona a comprobar.
 * @return {boolean} Indicador de si la zona es segura o no.
 **/
function esZonaSegura(coordenadas: string): boolean {
    // Simulación: zonas que empiecen con "A" son seguras
    return coordenadas.startsWith("A");
}
```

🪓 Consejo de superviviente

Si tienes todo bien organizado y etiquetado, no necesitas mirar dentro de cada bolsa para saber lo que hay. Lo mismo pasa con la programación: si eliges bien los nombres, las estructuras y el código es lo suficientemente claro, apenas necesitarás comentarios. Sin embargo, hay ocasiones en las que los comentarios también te serán útiles para explicar decisiones complejas o advertir sobre trampas ocultas en el código.

Ejercicios de supervivencia

Ejercicio 1.1: Documentando tu equipo

Llevas horas en tu casa, con las persianas bajadas y el silencio roto solo por gemidos distantes y pasos arrastrados que se acercan lentamente. Afuera, la ciudad ya no es segura: las criaturas rondan por las calles, y pronto tendrás que salir antes de que detecten tu ubicación. Antes de dar ese paso fatal, abres tu libreta de campo: es hora de registrar lo que llevas en la mochila. No basta con empaquetar, necesitas claridad absoluta. Cada objeto cuenta, cada decisión puede marcar la diferencia entre sobrevivir o convertirte en parte de la horda.

Tu misión: crea un archivo TypeScript (recuerda, con la extensión `.ts`) que documente tu equipo de supervivencia. Usa comentarios para explicar cada elemento:

```
let mochila = "Mochila táctica 40L";
let litrosAgua = 2.5;
let racionesComida = 8;
let linterna = true;
```

```
/*
 * Añade aquí un comentario de varias líneas
 * que explique tu estrategia de supervivencia
 */
```

Capítulo 2. Variables y constantes. Organizando tu mochila

El silencio de la casa se rompe con un gemido lejano que flota en el aire nocturno. No hay tiempo que perder: tu mochila de supervivencia debe estar perfectamente organizada antes de que esos sonidos se acerquen más. Cada recurso que guardes debe llevar una etiqueta clara con su nombre: “agua”, “comida”, “mapa”, “munición”. Así, cuando necesites algo urgentemente mientras huyes, sabrás exactamente dónde encontrarlo sin hacer ruido innecesario.

En programación, esas etiquetas son el nombre de las **variables** y **constantes** que te permiten organizar y acceder a tus recursos de forma eficiente.

Variables

Para comenzar a organizarte, usas tus pósits en los que escribes el nombre de algún suministro de tu mochila. Una variable sería el equivalente a esos pósits: puedes despegarlos y ponerlos en otro objeto cuando lo necesites. Por ejemplo, en una botella tienes el pósit “agua”, si la botella se agota podrás poner la etiqueta en otra botella de agua. Es decir, hemos cambiado el contenido, pero la etiqueta sigue siendo útil, siempre y cuando tenga el mismo tipo de dato.

El formato para declarar una variable es el siguiente:

```
// El símbolo "=" asignar un valor a la variable
let nombreVariable: tipo = valor;

// La asignación del valor inicial es totalmente opcional.
```

Por convenio, se suele usar lowerCamelCase para el nombre de las variables: los identificadores se escribirán con minúsculas y si hay varias palabras la primera letra de cada una irá en mayúsculas (excepto la letra inicial del nombre de la variable).

```
let litrosAguaRestante: number = 5;

// Esta variable no tiene un valor inicial
let diasComida: number;

// Puedes cambiar el valor de cualquier variable una vez declarada.
litrosAguaRestante = 4;
diasComida = 2;
```

Constantes

Imagina que escribes el nombre de un objeto, sobre éste, utilizando un bolígrafo permanente. Una vez que nombras el objeto, ese nombre y ese contenido no cambiarán durante toda la expedición. Así, evitas confusiones y te aseguras de que lo

importante siempre esté bien identificado. Pues algo similar son las constantes, una vez inicializadas ya no podrás modificar su valor durante toda el tiempo de vida de la aplicación.

El formato para las constantes es muy parecido al que usamos para declarar las variables, sólo que en este caso, es obligatorio asignar un valor inicial:

```
const NOMBRE_CONSTANTE: tipo = valor;
```

El nombre de las constantes se suele escribir con mayúsculas, y si se compone de varias palabras se utiliza _ como separador:

```
const LITROS_POR_DIA = 2;
const MENSAJE_AYUDA = "SOS";

// Si intentamos cambiar el valor de una constante, obtendremos un error.
// LITROS_POR_DIA = 3;
```

🪓 Consejo de superviviente

Usa solo caracteres del alfabeto inglés para los nombres de todo lo que uses en tu programa: variables, constantes, funciones, enumerados, clases, etc. Evita tildes, eñes y otros caracteres especiales (ñ, á, ü, etc.). Aunque algunos lenguajes los aceptan, pueden causar problemas de compatibilidad cuando compartas tu código con otros supervivientes o cuando uses herramientas de diferentes sistemas. Es como usar señales universales en lugar de dialectos locales: todos podrán entenderte sin importar de dónde vengan.

🗒️ Nota en la cabaña

También existe la palabra clave `var` para declarar variables en JavaScript. Sin embargo, su uso no se recomienda porque puede provocar comportamientos inesperados relacionados con el ámbito y la redeclaración de variables. Siempre que puedas, utiliza `let` o `const` para mayor seguridad y claridad en tu código.

Buenas prácticas para un código limpio

Así como en una expedición, la organización y la claridad pueden marcar la diferencia entre sobrevivir y perderse, en programación seguir buenas prácticas te ayudará a mantener tu código legible y funcional:

- Nombra claramente tus variables y constantes. Usa nombres descriptivos y evita abreviaturas confusas.
- Mantén la coherencia en el estilo de nombres: lowerCamelCase para variables, MAYÚSCULAS_CON_GUIONES para constantes.
- No reutilices variables para propósitos distintos, cada etiqueta debe tener un solo uso. Es decir, `diasConAgua` no debería pasar a guardar información del número de baterías.

Un ejemplo, de nombres muy poco descriptivos sería el siguiente:

```
const l = 10;
let c = a / b;
```

Lo anterior, puede significar cualquier cosa y te hará perder un tiempo valioso para entenderlo. Sin embargo, el siguiente código es comprensible y te permite saber rápidamente qué representa cada variable:

```
const LITROS_POR_DIA: number = 10;
let diasConAgua: number = litrosAguaRestante / LITROS_POR_DIA;
```

Evita los números mágicos

Cuando escribes código, a veces aparecen números que funcionan perfectamente pero cuyo significado no es evidente. Por ejemplo, imagina que calculas el peso de tu mochila así: `let pesoTotal = 8.2 + 3.5 + 1.8;`. Funciona, pero si regresas a ese código días después, ¿recordarás qué representa cada número? ¿Son kilogramos de agua, comida, munición? ¿Por qué esos valores específicos?

Estos valores literales sin contexto se conocen como **números mágicos**: números que aparecen directamente en el código sin explicación de su significado. Dificultan la lectura y el mantenimiento porque quien lee el código (incluyéndote a ti mismo en el futuro) debe adivinar qué representan.

Observa estos dos ejemplos del mismo cálculo:

Código con números mágicos:

```
// pesoMochilla llega de otros cálculos
const capacidadRestante = 15 - pesoMochila;

console.log(`Puedes cargar ${capacidadRestante}kg más`);
```

Código sin números mágicos:

```
const CAPACIDAD_MAXIMA_MOCHILA_KG = 15;

// pesoMochilla llega de otros cálculos
const capacidadRestante = CAPACIDAD_MAXIMA_MOCHILA_KG - pesoMochila;

console.log(`Puedes cargar ${capacidadRestante}kg más`);
```

La diferencia es clara: en el segundo caso, cuando vuelvas a leer tu código comprenderás inmediatamente qué representa el valor que antes era 15. Además, si necesitas cambiar la capacidad de la mochila, solo modificas el valor de la constante en un lugar, y todos los cálculos que la usen se actualizarán automáticamente.

🪓 Consejo de superviviente

No todos los números son números mágicos. Valores obviamente comprensibles como 0, 1, 2 (para duplicar), 100 (para porcentajes), o -1 (para indicar “no encon-

trado") suelen ser claros por sí mismos. Usa tu criterio: si un número necesita contexto para entenderse, conviértelo en una constante con nombre descriptivo.

Ejercicios de supervivencia

Ejercicio 2.1: Organizando tu inventario

Los gruñidos se escuchan más cerca ahora, y sabes que no puedes permitirte ningún error. Abres la mochila sobre la mesa con manos temblorosas y observas que tiene un espacio definido y un peso máximo que no se puede superar sin romperla. Pero dentro de esos márgenes críticos, sabes que a futuro tu inventario siempre cambiará: un día llevarás más agua, otro menos comida, y el peso variará con cada objeto que añadas o consumas mientras huyes de las hordas zombis.

En tu libreta de campo, con la linterna iluminando las páginas, decides registrar qué cosas son inmutables y cuáles cambian con el tiempo. Así sabrás en todo momento cómo organizar tus recursos para no quedarte indefenso en medio de la huida cuando la horda aparezca.

Tu misión: declara variables y constantes para organizar tu inventario de supervivencia.

- Define constantes con:
  - La capacidad de la mochila que será de 40 litros
  - El peso máximo de la mochila, que será 15 kilogramos
- Declara variables para recursos que pueden cambiar:
  - El número de litros de agua con un valor inicial de 3.5 litros.
  - El número de raciones de comida actuales y luego asígnale un valor de 12 raciones.
  - El peso actual con un valor inicial de 8.2 kilogramos.

Capítulo 3. Tipos de datos primitivos. Separando los suministros

No todos los suministros son iguales. Algunos son líquidos, otros sólidos; unos sirven para alimentarte, otros para orientarte. En programación ocurre lo mismo: cada dato tiene un **tipo**, y como superviviente, debes conocerlos tan bien como tu brújula o tu cuchillo.

Para declarar un tipo de dato, usaremos la siguiente estructura `: tipo_de_dato`. Por ejemplo:

```
let zombis: number = 14;
let nombre: string = "Pedro";
```

Tipos numéricos

Los tipos de datos numéricos representan tus recursos cuantificables. Te dicen cuánto tienes, cuánto necesitas y cuánto puedes arriesgar.

Los **números enteros** son ideales para contar objetos, pasos o zombis.

```
// Número exacto. No guardas baterías partidas
let baterias: number = 24;
```

Los **números decimales** son cruciales para medidas más precisas: distancia, tiempo, temperatura, peso.

```
// Aquí la precisión es vital.
let temperatura: number = -15.7;
```

> 🪓 **Consejo de superviviente**
>
> En los códigos de ejemplo verás que se usa el punto (`.`) para los valores decimales, así que tendremos `2.5` en lugar de `2,5`. Esto es debido a que la mayoría de lenguajes de programación (incluido TypeScript) reconocen el punto como separador decimal estándar.

Aunque aquí se ha explicado este tipo de datos por separado, en TypeScript no hay distinción para los valores enteros y reales. Es por ello que sólo existe el tipo `number` para representar ambos.

Recuerda escoger adecuadamente el tipo de número correcto. Si haces cálculos con enteros cuando deberías usar decimales, podrías acabar con medio litro de agua "fantasma".

> 🗒️ **Nota en la cabaña**
>
> Al operar con valores numéricos, puede aparecer un valor misterioso: `NaN` (Not a Number). Es como encontrar una lata en tu mochila que parece comida, pero al abrirla... ¡está vacía o tiene algo incomible! `NaN` surge cuando intentas hacer operaciones matemáticas imposibles, como dividir 0 entre 0, o convertir una palabra en número. Si ves `NaN`, no confíes en ese recurso: es señal de que algo salió mal en tus cuentas y podrías quedarte sin provisiones cuando más lo necesitas. Revisa siempre tus cálculos y asegúrate de que no haya latas vacías en tu inventario.

Cadenas

Las cadenas de texto son como los mensajes que dejas escritos en las paredes o los datos que apuntas en tu libreta de supervivencia. Una cadena (`string`) representa un conjunto de caracteres: desde una simple palabra hasta párrafos completos de información crítica.

En el mundo post-apocalíptico, estas cadenas pueden contener:

- **Nombres de lugares**: “Refugio Norte”, “Hospital Abandonado”

- **Mensajes de emergencia**: “SOS - Necesitamos suministros médicos”
- **Coordenadas**: “Grid 47-B, Sector Industrial”
- **Inventarios**: “3x Latas, 2x Medicinas, 1x Radio”

```
let nombreDestinoSeguro: string = "Presa del Tablillas";
let mensaje: string = "Grupo hostil avistado al sur";
let coordenadas: string = "Lat: 40.7128, Lon: -74.0060";
```

Las cadenas son vitales para comunicarte con más supervivientes y registrar información que puede salvar vidas. Sin ellas, perderías la capacidad de documentar tus hallazgos o transmitir ubicaciones precisas por radio.

En TypeScript, hay tres formas de escribir una cadena:

```
// Usando comillas simples
let mensaje: string = '¡Cuidado! Hay zombis cerca.';

// Usando comillas dobles
let aviso: string = "Refugio seguro encontrado";

// Usando template literals (plantillas de cadena) con el símbolo `.
// Esto nos permite embeber variables, constantes y otro código mediante
// el uso del siguiente formato: ${códigoAIncluir}
let zona: string = "Centro Comercial";
let sector: number = 7;
let informe: string = `Explorando ${zona}, sector ${sector}`;
```

🪓 Consejo de superviviente

En la mayoría de lenguajes las cadenas van entre comillas. Asegúrate de cerrar bien dichas comillas, ya que si olvidas una, tu mensaje puede perderse en la jungla del código.

🗒️ Nota en la cabaña

A lo largo de la guía, por compatibilidad con otros lenguajes, usamos comillas dobles `""` cuando no vayamos a incluir el valor de una variable o constante.

Booleanos

El tipo `boolean` consta sólo de dos valores **verdadero** (`true`) y **falso** (`false`). Esto te permite tomar decisiones. ¿Marchar o quedarse? ¿Encender el fuego o esperar?

```
let esZonaSegura: boolean = true;
let tormentaAproximandose: boolean = false;
```

> 🪓 **Consejo de superviviente**
>
> Nunca asumas que algo es verdadero por defecto. Verifica tus condiciones, o podrías dar por segura una zona repleta de trampas.

Cuando no tenemos recursos: `null` y `undefined`

null

En la supervivencia, hay momentos en los que buscas en tu mochila y... ¡no queda nada! Ni agua, ni comida, ni munición. En programación, esos momentos se representan con `null`.

Tener un valor `null` es como mirar dentro de la mochila y ver que el compartimento está vacío, pero sabes que existe. Has guardado algo ahí antes, pero ahora no hay nada. Es una ausencia intencionada: tú mismo has decidido dejarlo vacío.

undefined

`undefined` representa la ausencia de valor porque la variable no ha sido inicializada. Es asignado por el motor de JavaScript cuando declaras una variable sin darle valor, o cuando accedes a una propiedad inexistente. Aunque otros lenguajes tienen conceptos similares, la forma en que `undefined` se comporta es propia de JavaScript, así que es importante conocer sus peculiaridades.

> 🪓 **Consejo de superviviente**
>
> Antes de salir a explorar, revisa bien tu mochila y asegúrate de que no tienes compartimentos vacíos (`null`) ni olvidados (`undefined`). Si intentas usar algo que no existe, podrías encontrarte en serios problemas cuando más lo necesites.

> 🗒️ **Nota en la cabaña**
>
> Recuerda que `null` y `undefined` son valores distintos. Usa `null` cuando quieras indicar que algo está vacío a propósito, y evita usar `undefined` ya que el propio lenguaje lo "asignará automáticamente" cuando una variable o constante ni siquiera se hayan inicializado. ¡No confundas la falta de recursos con el olvido de prepararlos!

El tipo `any`

Imagina que encuentras una caja cerrada durante una exploración. No sabes qué hay dentro: podría ser comida, medicinas, munición, o incluso algo peligroso. En TypeScript, existe un tipo especial llamado `any` que funciona como esa caja misteriosa: puede contener cualquier tipo de dato.

```
let cajaEncontrada: any = "Radio de emergencia";
console.log(`Contenido inicial: ${cajaEncontrada}`);

// Después descubres que también puede ser un número
cajaEncontrada = 42;
console.log(`Ahora contiene: ${cajaEncontrada} baterías`);

// O incluso un booleano
cajaEncontrada = true;
console.log(`¿Está funcional? ${cajaEncontrada}`);
```

El tipo any es útil cuando:

- Mientras estás trabajando con código antiguo que no tiene tipos definidos.
- Recibes datos de fuentes externas (como mensajes de radio) donde no sabes qué esperar.
- Necesitas una solución temporal mientras determinas el tipo exacto.

Aunque exista este tipo de datos, debes usarlo con mucha precaución. Es como llevar una caja sin etiquetar en tu mochila: funciona, pero pierdes la ventaja de saber exactamente qué tienes. TypeScript no podrá ayudarte a detectar errores si usas any en exceso. Siempre que puedas, especifica el tipo de tus datos: tu yo del futuro te lo agradecerá.

🗒️ Nota en la cabaña

Cuando encuentres datos misteriosos en tu código, puedes usar el operador `typeof` como una linterna para inspeccionar qué tipo de suministro tienes realmente. Es especialmente útil cuando trabajas con variables de tipo any o cuando recibes datos de fuentes externas.

```
let cajaDesconocida: any = "Medicinas";
console.log(typeof cajaDesconocida); // "string"

cajaDesconocida = 25;
console.log(typeof cajaDesconocida); // "number"

cajaDesconocida = true;
console.log(typeof cajaDesconocida); // "boolean"

cajaDesconocida = null;
// ¡Cuidado con esta trampa!
console.log(typeof cajaDesconocida); // "object"
```

El operador `typeof` devuelve una cadena que identifica el tipo: `"string"`, `"number"`, `"boolean"`, `"undefined"`, `"object"`, `"function"` o `"symbol"`. Ten en cuenta que `typeof null` devuelve `"object"` por razones históricas de JavaScript, así que no te confíes solo de esta inspección cuando busques valores nulos.

Ejercicios de supervivencia

Ejercicio 3.1: Clasificando suministros

El primer refugio que has encontrado apenas es más que una caseta de mantenimiento olvidada en las afueras. No hay mucho espacio, pero es lo suficientemente sólido como para pasar la noche y evitar los grupos de zombis que caminan sin descanso.

Con tu linterna inspeccionas cada rincón: un poco de comida enlatada, una vieja radio que tal vez funcione, y huellas recientes que no sabes si son humanas o no.

En tu libreta de campo decides clasificar los datos más importantes: el nombre del refugio, la temperatura, si es seguro, y el último mensaje de radio recibido. Si no lo registras con claridad, podrías olvidar detalles vitales cuando llegue el momento de decidir.

Tu misión: crea variables con diferentes tipos de datos para tu inventario:

- Nombre del refugio: cadena
- Temperatura actual: número decimal
- ¿Es zona segura?: booleano
- Último mensaje de radio: string o null si no hay contacto

Tras esto, imprime un mensaje con el estado del refugio.

```
// Ejemplo de cómo empezar:
let nombreRefugio: string = "Refugio Alpha";
// Continúa con el resto...

// TODO: Muestra toda la información usando console.log
console.log("=== ESTADO DEL REFUGIO ===");
```

Capítulo 4. Operadores. Cómo leer las señales del entorno

Cuando estás en una situación extrema, tu cerebro analiza constantemente el entorno: calcula cuánto te queda, compara riesgos y combina señales para tomar decisiones. En programación, eso lo haces con **operadores**.

Éstos son símbolos que permiten **transformar y evaluar datos**: sumar, restar, comparar, analizar condiciones... En otras palabras, son como tus sentidos, tus conocimientos y tu juicio trabajando juntos para sobrevivir.

Operadores aritméticos

Cuando necesitas saber cuántos días puedes resistir, qué distancia te queda o si los suministros alcanzan para toda la semana, estás haciendo **operaciones aritméticas**.

```
let comidaTotal: number = 50; // raciones disponibles
let consumoDiario: number = 3; // raciones por día
let metrosTotalesDestino: number = 15000; // metros hasta el refugio
let metrosRecorridos: number = 8500; // metros ya caminados
```

```ts
const diasSupervivencia: number = comidaTotal / consumoDiario;
const distanciaRestante: number = metrosTotalesDestino - metrosRecorridos;

console.log(`Puedes sobrevivir ${diasSupervivencia} días`);
console.log(`Te quedan ${distanciaRestante} metros por recorrer`);
```

En la mayoría de lenguajes de programación estos son los operadores aritméticos básicos:

Operador	Significado	Prioridad	Ejemplo
**	Potencia	Más Alta	`deteccion = zombis ** nivelRuido`
*	Multiplicación	Alta	`energiaTotal = pasos * caloriasPorPaso`
/	División	Alta	`dias = raciones / consumoDiario`
+	Suma	Media	`total = litrosAgua + litrosTe`
-	Resta	Media	`faltan = objetivo - avance`
%	Módulo (resto)	Baja	`sobran = balas % capacidadCargador`

Cuando calcules tus recursos, recuerda que la prioridad de los operadores puede cambiar el resultado. Para evitarlo, usa paréntesis para priorizar las operaciones. Podrías pensar que tienes más comida o agua de la que realmente hay, y esa confusión puede ser fatal en una situación crítica. ¡No dejes que un error de cálculo te deje sin provisiones!

Por ejemplo:

- Esta operación `3 + 4 * 12`, da como resultado 51:
  - Por prioridad, primero se resuelve la multiplicación `4 * 12 = 48`.
  - Luego se realiza la suma `48 + 3 = 51`.
- Al usar paréntesis `(3 + 4) * 12`, el resultado será 84:
  - Los paréntesis dan prioridad a la suma `3 + 4 = 7`.
  - A continuación, se lleva a cabo la multiplicación `7 * 12 = 84`.

Operadores de cadena

Puedes usar el operador + cuando trabajas con cadenas de texto. En este contexto no realiza la operación de suma sino que une o **concatena** los valores:

```ts
let nombreSuperviviente: string = "Alex";
let mensajeSOS: string = "SOS enviado por " + nombreSuperviviente;
// Resultado: "SOS enviado por Alex"

let distancia: number = 500;
let reporte: string = "Refugio a " + distancia + " metros";
// Resultado: "Refugio a 500 metros"
```

Si mezclas números y cadenas, TypeScript convertirá todo a texto:

```ts
let suministros: number = 15;
let estado: string = "Tengo " + suministros + " raciones";
```

```
// Resultado: "Tengo 15 raciones"

// ¡Cuidado con este error común!
let resultado = "5" + 3; // "53" (cadena), no 8 (número)
```

> 🗒️ **Nota en la cabaña**
>
> En JavaScript se permiten otros operadores aritméticos con las cadenas que, si van acompañados de algún valor numérico normalmente, terminarán convirtiendo el valor de la cadena a un número para así operar con él. Evita esta clase de conversiones automáticas porque pueden provocar que obtengas el temido **NaN**. Si por algo usamos TypeScript es para controlar que usas los tipos correctos. Recuerda que un cálculo erróneo pueda hacer que termines en medio de una horda de zombis.

> 🗒️ **Nota en la cabaña**
>
> Para evitar confusiones, usa **template literals** (plantillas de cadena) cuando combines texto y variables: ‘`Tengo ${suministros} raciones`‘. Es más claro y menos propenso a errores que la concatenación.
> Eso sí, a lo largo de la guía verás que, cuando el texto es muy largo, usamos la concatenación para permitir que pueda verse correctamente.

Operadores de comparación

Saber si estás mejor o peor que ayer, si tienes lo mínimo necesario, o si has llegado al límite, es parte del día a día de la supervivencia. Cada decisión cuenta: comparar distancias, recursos o niveles de amenaza puede significar la diferencia entre resistir una noche más o perderlo todo. Tu mente analiza constantemente: ¿hay más zombis que balas?, ¿la temperatura ha bajado desde el amanecer?, ¿el refugio sigue siendo seguro o ha quedado comprometido?

```
let temperatura: number = 8;
let horaActual: number = 19; // 19:00 horas
let horaPuestaSol: number = 18; // 18:00 horas
let suministros: number = 15;
let nivelCritico: number = 20;

// ¿Es seguro continuar?
let haceFrio: boolean = temperatura < 12;
let necesitoRefugio: boolean = horaActual >= horaPuestaSol;
let debesEnviarSocorro: boolean = suministros < nivelCritico;

console.log(`Hace frío: ${haceFrio}`);
console.log(`Necesito refugio: ${necesitoRefugio}`);
console.log(`Debes enviar socorro: ${debesEnviarSocorro}`);
```

Estos operadores devuelven **valores booleanos**: `verdadero` o `falso`.

Operador	Significado	Ejemplo
==	Igual a (con conversión)	`clima == "despejado"`
===	Igual a (sin conversión)	`agua === 5`
!=	Distinto de (con conversión)	`zona != "segura"`
!==	Distinto de (sin conversión)	`estado !== "peligro"`
>	Mayor que	`litros_agua > 2`
<	Menor que	`zombis < municion`
>=	Mayor o igual que	`hora_actual >= hora_puesta_sol`
<=	Menor o igual que	`temperatura <= 0`

🪓 Consejo de superviviente

No confundas = con ==. El primer operador guarda un valor, el segundo lo compara. Si cometes el error de acampar (`acampar = true`) cuando realmente querías comprobar si podías (`acampar == true`), puede costarte muy caro.

En JavaScript existen dos tipos de operadores de igualdad y desigualdad que funcionan de manera diferente. Como superviviente, necesitas conocer la diferencia para evitar errores.

Operadores con conversión automática (== y !=)

```
"1" == 1      // true: la cadena "1" se convierte al número 1
"3" != 3      // false: se convierte y compara 3 != 3
0 == false    // true: el valor false se convierte a 0
```

Operadores estrictos (=== y !==)

```
"1" === 1     // false: diferentes tipos (string vs number)
0 === false   // false: diferentes tipos (number vs boolean)
```

Los operadores estrictos comparan tanto el **valor** como el **tipo** de datos, sin conversiones automáticas. Esto es más predecible y seguro.

🗒️ Nota en la cabaña

Como recomendación, usa siempre === y !== para evitar comparaciones inesperadas.

Operadores lógicos

Una vez que sabes comparar, lo natural es empezar a combinar esas comparaciones para tomar decisiones más complejas. Y es que rara vez una sola señal es suficiente para tomar decisiones. Usas múltiples factores: clima, energía, terreno… y los combinas mentalmente. Eso es lo que hacen los **operadores lógicos**.

Operador	Significado	Prioridad	Ejemplo
!	No (invierte el valor booleano)	Alta	`!zonaPeligrosa`
&&	Y (ambas condiciones deben cumplirse)	Media	`aguaDisponible && terrenoSeguro`
||	O (basta con una condición)	Baja	climaBueno || refugioCercano

```
// Estado inicial del campamento
let interruptor: boolean = true;
let climaBueno: boolean = false;
let refugioCercano: boolean = true;

// Al pulsar el interruptor cambiamos su estado. Eso lo podemos hacer con !
let luzEncendida: boolean = !interruptor;
// Hemos apagado la luz: ahora es false

// Podremos continuar la marcha si el clima es bueno o tenemos un refugio
// cercano. Si cualquiera de las dos se cumple nos sirve para seguir
// caminando.
// Lo siguiene dará true puesto que hay un refugio cercano
let continuarMarcha: boolean = climaBueno || refugioCercano;
// Continuar marcha (OR): true
console.log(`Continuar marcha (OR): ${continuarMarcha}`);

// Podremos continuar la marcha cuando el clima sea bueno y tengamos un
// refugio cercano. Sólo podremos continuar cuando las dos condiciones se
// cumplan.
// Lo siguiene dará false puesto que no hay clima bueno.
continuarMarcha = climaBueno && refugioCercano;
// Continuar marcha (AND): false
console.log(`Continuar marcha (AND): ${continuarMarcha}`);
```

Para entender cómo se comporta cada operación booleana, usamos una tabla de verdad. En ella se muestran todas las combinaciones posibles de entradas y el resultado que obtendremos en cada caso.

Tabla de la verdad de la operación AND
(solo devuelve true cuando ambas condiciones lo son)

Entrada 1	Entrada 2	Resultado
false	false	false
false	true	false
true	false	false
true	true	true

Tabla de la verdad de la operación OR
(devuelve true si al menos una condición se cumple)

Entrada 1	Entrada 2	Resultado
false	false	false
false	true	true
true	false	true
true	true	true

Tabla de la verdad de la operación NOT
(invierte el valor: lo que era true pasa a false, y viceversa)

Entrada	Resultado
false	true
true	false

Cuando trabajes con operadores lógicos, no siempre usarás valores booleanos. Puedes utilizar cualquier otro tipo de dato que terminará, o transformándose en un valor `false` (a estos tipos de datos los llamamos valores *falsy*), o en un valor true (en este caso les llamaremos valores *truthy*).

- Valores falsy: `null`, `undefined`, `false`, `NaN`, `0`, `-0`, `""` (cadena vacía).
- Valores truthy: `true`, cualquier objeto, cualquier array (aunque esté vacío), cualquier número distinto de 0, cualquier cadena no vacía y los valores `Infinity` y `-Infinity`.

Dicho esto, cuando no uses valores estrictamente booleanos en los operadores AND y OR, la operación realmente devolverá uno de esos valores:

AND

- Si ambos valores son truthy, devuelve el segundo.
- Si no, se devuelve el primer valor falsy que haya.

```
1 && "Zombi" // "Zombi" => Se devuelve el segundo valor truthy
0 && "Zombi" // 0 => El primer valor falsy encontrado
"Zombi" && 0 // 0 => El primer valor falsy encontrado
null && 0    // null => El primer valor falsy encontrado
```

OR

- Se devuelve el primer valor truthy encontrado.
- Si ambos valores son falsy, se devuelve el segundo.

```
1 || "Zombi" // 1 => Se devuelve el primer valor truthy
0 || "Zombi" // "Zombi" => Se devuelve el primer valor truthy
"Zombi" || 0 // "Zombi" => Se devuelve el primer valor truthy
null || 0    // 0 => Todos falsy. Se devuelve el último valor
```

🗒️ Nota en la cabaña

El operador `!` por norma general invierte un valor booleano, pero puede actuar con otro tipo de valores:

```
!""          // true  => Valor falsy negado
!"SOS"       // false => Valor truthy negado
!0           // true  => Valor falsy negado
!12          // false => Valor truthy negado
!null        // true  => Valor falsy negado
```

Al igual que ocurre con los operadores aritméticos, debes tener en cuenta la prioridad de los operadores lógicos. Así que recuerda que && se evalúa antes que `||`. Por lo que, en el caso de querer priorizar una operación sobre otra, deberás usar paréntesis para dejar claro el orden y evitar sorpresas en tus decisiones.

Y es que no es lo mismo:

```
continuarMarcha = (climaBueno || refugioCercano) && energiaSuficiente;
```

que te permitirá continuar la marcha sólo cuando haya energía suficiente y se cumpla al menos una de las otras dos condiciones, que lo siguiente:

```
continuarMarcha = climaBueno || refugioCercano && energiaSuficiente;
```

en cuyo caso, te permitirá continuar la marcha cuando haya un clima bueno, o exista un refugio cerca y a su vez tengas energía suficiente. Ya que el operador && tiene preferencia sobre `||`.

Con la primera opción, siempre será necesario tener energía suficiente. Sin embargo, con la segunda opción, simplemente con que haya un clima adecuado podrías continuar la marcha aunque te hayas agotado.

🗒️ Nota en la cabaña

En ES2020 se introdujo un operador lógico con un nombre un poco extraño: **operador de coalescencia nula** (nullish coalescing en inglés), representado por el símbolo `??`. Funciona como un OR, pero omitiendo la mayoría de los valores `falsy` y sólo teniendo en cuenta los valores `null` y `undefined`. De forma que si el primer elemento tiene uno de esos valores, se devolverá el valor que hay tras `??`:

```
let combustibleDisponible = 0;
combustibleDisponible ?? 50 // Devolverá 0. No es null ni undefined
combustibleDisponible || 50 // Devolverá 50. 0 se "convierte" a false
```

Este operador te permite asegurarte de que sólo en el caso de que el primer valor no esté definido o tenga valor nulo se utilice el segundo valor. Es muy útil sobre todo con valores numéricos.

Operadores unarios

En situaciones límite, a veces necesitas tomar decisiones rápidas con un solo recurso: sumar una unidad de energía, invertir una señal, o comprobar si tienes algo en la mochila. Los **operadores unarios** te permiten modificar o consultar el valor de una variable usando un solo operando, como si fuera un gesto instintivo de supervivencia.

Los operadores de **incremento** (`++`) y/o **decremento** (`--`) te permiten sumar o restar uno a un valor. Es decir, te permiten dar un paso adelante o atrás en el contador.

```
let latasRefresco = 5;
latasRefresco++; // Has encontrado una lata más: ahora tienes 6
latasRefresco--; // Te has bebido una lata: ahora tienes 5
```

El operador de **negación** (`!`), que ya conoces de los operadores lógicos, también cuenta como unario.

Finalmente, el operador de **cambio de signo** (`-`) cambia un valor positivo a negativo y viceversa.

```
let temperatura = 10;
// Cambiamos el signo: ahora es -10 ¡cuidado con la hipotermia!
temperatura = -temperatura;
// Al aplicar el cambio de signo a -10 ahora el valor vuelve a ser 10
temperatura = -temperatura;
```

Operador	Significado	Ejemplo
`++`	Suma uno al valor. Incremento.	`latasRefresco++`
`--`	Resta uno al valor. Decremento.	`latasRefresco--`
`!`	Invierte el valor booleano	`!linternaEncendida`
`-`	Cambia el signo de un número	`-temperatura`

Usar operadores unarios te permite reaccionar rápido, pero hay que tener cautela si usas los operadores `++` y `--` como prefijos o sufijos cuando asignas los valores. Ya que si se antepone el operador, primero se hace la operación de incrementar o decrementar el valor y luego la asignación:

```
let latas: number = 6;
let latasRestantes: number = --latas;
// latas y latasRestantes tienen el valor 5
```

Sin embargo, si el operador se sitúa tras la variable, primero se realizará la asignación y luego se incrementará o decrementará el valor:

```
let latas: number = 6;
let latasRestantes: number = latas--;
// latasRestantes tiene el valor 6 y latas el valor 5.
```

🗒️ **Nota en la cabaña**

Cuando necesites actualizar el valor de una variable usando su propio valor anterior, puedes simplificar tu código usando los operadores de asignación compuesta. Así evitas repetir el nombre de la variable y tu código será más limpio y fácil de leer. Solo tienes que poner el operador antes del signo igual (=):

```
latas = latas + 3; // Se simplifica a:
latas += 3;

suministros = suministros * 2; // Se simplifica a:
suministros *= 2;

esSeguro = esSeguro && hayRefugio; // Se simplifica a:
esSeguro &&= hayRefugio;
```

A partir de ES2021, se introdujeron los operadores compuestos como &&=, ||= y ??=.

Ejercicios de supervivencia

Ejercicio 4.1: Calculadora de recursos

Han pasado tres días desde que encontraste un refugio abandonado. Tus recursos se agotan y debes calcular si lo que tienes alcanzará para completar la misión.

Mientras organizas los suministros en el sótano del refugio, tu detector de metales emite pitidos constantes: hay algo enterrado cerca que podría cambiar tu suerte. Pero antes de agarrar la pala, necesitas saber exactamente cuánto tiempo puedes resistir con lo que tienes. La excavación completa te llevará ocho días, y no hay margen para errores de cálculo.

Tu misión: realiza los siguientes cálculos en tu vieja libreta de campo:

- Calcula el consumo diario de agua (2 litros).
- Calcula el consumo diario de comida (1.5 raciones).
- Calcula cuántos días puedes sobrevivir con los recursos actuales.
- Calcula cuántos recursos adicionales necesitas para completar la misión.
- Como extra, calcula el porcentaje de recursos que se consumirá cada día.

```
// Estado inicial del campamento tras tres días de organización
let litrosAguaTotal: number = 45;
let racionesComidaTotal: number = 30;
let diasMision: number = 8;

console.log("=== ANÁLISIS DE RECURSOS DEL REFUGIO ===");
```

Ejercicio 4.2: Sistema de alerta temprana

La tormenta llegó sin avisar. Mientras dormías en el refugio, el viento comenzó a aullar entre las grietas de las paredes, y la temperatura descendió bruscamente. Un

fuerte golpe te despierta, debes averiguar qué lo ha producido y evaluar los riesgos inmediatamente.

Has instalado varios sensores alrededor del refugio durante los últimos días, y ahora tu supervivencia depende de interpretar correctamente esas señales. El generador ha empezado a fallar intermitentemente, el combustible escasea, y la visibilidad se ha reducido considerablemente. Cada decisión que tomes en las próximas horas podría determinar si puedes sobrevivir a esta noche.

Tu misión: define un sistema de alerta que debe evaluar si las condiciones superan los umbrales críticos:

- Temperatura peligrosa: por debajo de -10º C
- Viento extremo: superior a 40 km/h
- Visibilidad crítica: menos de 100 metros
- Combustible en reserva: menos del 20%
- Horario de alto riesgo: después de las 21:00

```
// Lecturas actuales de los sensores durante la tormenta
let temperatura: number = -12;
let vientoKmh: number = 45;
let visibilidadMetros: number = 50;
let nivelCombustible: number = 18;
let horaActual: number = 22;

console.log("=== SISTEMA DE ALERTAS DE EMERGENCIA ===");
```

Ejercicio 4.3: Lógica de supervivencia

La tormenta ha pasado, pero sus consecuencias se sienten en cada rincón del refugio. Durante la noche, el estrés del temporal ha puesto a prueba todos los sistemas: el generador funciona de forma intermitente y los suministros se agotan más rápido de lo previsto.

Esta mañana, mientras revisas los daños con tu libreta de campo en la mano, la radio capta una transmisión entrecortada. No sabes si proviene de supervivientes, de una trampa o de simples interferencias, pero el mensaje sugiere que alguien podría necesitar ayuda.

Tu misión: tu experiencia te ha enseñado que en situaciones así no puedes basarte en una sola señal. Por lo que necesitas evaluar múltiples condiciones simultáneamente:

- Determina si es seguro salir del refugio. Para ello debes comprobar que el perímetro sea seguro y al menos que o el tiempo sea estable o el sistema de comunicaciones esté operativo.
- Determina si el refugio está en modo crítico. Verifica si el generador no está funcionando o hay suministros insuficientes.
- Determina si merece la pena salir a investigar la señal de auxilio.
  - Analiza que haya suministros suficientes para llevar contigo.
  - Al llegar al lugar de origen de la señal, deberás analizar si la zona es segura.
  - Por otro lado, deberás asegurarte de que en tu ausencia el generador tendrá que seguir funcionando según el combustible que queda.

```ts
// Condiciones actuales
let generadorFuncional: boolean = true;
let suministrosSuficientes: boolean = false;
let comunicacionActiva: boolean = true;
let perimetroSeguro: boolean = true;
let climaEstable: boolean = false;

console.log("=== ANÁLISIS DE SEÑAL DE AUXILIO ===");
```

💡 Pista

Recuerda las prioridades en los distintos operadores que uses, y si es necesario utiliza paréntesis para clarificar el orden.