TypeScript: Desarrollo Profesional

1. Introducción a TypeScript

TypeScript es un superconjunto de JavaScript que añade tipado estático opcional. El código TypeScript se compila a JavaScript.

Beneficios de TypeScript:
- Detección de errores en tiempo de compilación
- Mejor autocompletado e IntelliSense en IDEs
- Código más mantenible y documentado
- Refactoring más seguro
- Soporte para características modernas de JavaScript

Instalación:
```
npm install -D typescript
npx tsc --init  # Crear tsconfig.json
```

Tipos básicos:
```
let nombre: string = "María";
let edad: number = 28;
let activo: boolean = true;
let lista: number[] = [1, 2, 3];
let tupla: [string, number] = ["edad", 28];
```

Inferencia de tipos:
```
let mensaje = "Hola";  // TypeScript infiere string
mensaje = 123;  // Error: Type 'number' is not assignable to type 'string'
```

El tipo 'any' desactiva el checking (evitar su uso):
```
let dato: any = "texto";
dato = 123;  // Sin error, pero perdemos beneficios
```

El tipo 'unknown' es más seguro que any:
```
let valor: unknown = "texto";
if (typeof valor === "string") {
  console.log(valor.toUpperCase());  // OK, TypeScript sabe que es string
}
```

2. Interfaces y Types

Las interfaces y types definen la forma de los objetos.

Interfaces:
```
interface Usuario {
  id: number;
  nombre: string;
  email: string;
```

```
  edad?: number;  // Propiedad opcional
  readonly createdAt: Date;  // Solo lectura
}

const usuario: Usuario = {
  id: 1,
  nombre: "Ana",
  email: "ana@ejemplo.com",
  createdAt: new Date()
};
```

Type aliases:

```
type ID = string | number;
type Punto = { x: number; y: number };

type Estado = "pendiente" | "activo" | "completado";  // Union literal
let estado: Estado = "activo";
```

Diferencias principales:
- Las interfaces pueden extenderse y fusionarse
- Los types son más flexibles para unions y tipos complejos

Extensión de interfaces:

```
interface Persona {
  nombre: string;
}

interface Empleado extends Persona {
  puesto: string;
  salario: number;
}
```

Intersection types:

```
type EmpleadoCompleto = Persona & {
  puesto: string;
  departamento: string;
};
```

Interfaces para funciones:

```
interface Operacion {
  (a: number, b: number): number;
}

const sumar: Operacion = (a, b) => a + b;
```

3. Funciones Tipadas

TypeScript permite tipar parámetros, valores de retorno y funciones completas.

Funciones básicas:

```
function saludar(nombre: string): string {
  return "Hola, " + nombre;
}
```

```
// Arrow function
const multiplicar = (a: number, b: number): number => a * b;
```

Parámetros opcionales y por defecto:

```
function crearUsuario(
  nombre: string,
  email: string,
  edad?: number,  // Opcional
  rol: string = "usuario"  // Por defecto
): Usuario {
  return { nombre, email, edad, rol };
}
```

Rest parameters:

```
function sumarTodos(...numeros: number[]): number {
  return numeros.reduce((acc, n) => acc + n, 0);
}
```

Sobrecarga de funciones:

```
function procesar(valor: string): string;
function procesar(valor: number): number;
function procesar(valor: string | number): string | number {
  if (typeof valor === "string") {
    return valor.toUpperCase();
  }
  return valor * 2;
}
```

Funciones como tipos:

```
type Callback = (error: Error | null, resultado?: string) => void;

function operacionAsincrona(callback: Callback): void {
  // ...
}
```

Funciones genéricas:

```
function primero<T>(array: T[]): T | undefined {
  return array[0];
}

const num = primero([1, 2, 3]);  // number | undefined
const str = primero(["a", "b"]);  // string | undefined
```

4. Genéricos

Los genéricos permiten crear componentes reutilizables que trabajan con múltiples tipos.

Función genérica básica:

```
function identidad<T>(valor: T): T {
  return valor;
}
```

```
const numero = identidad<number>(42);
const texto = identidad("hola");  // Inferencia automática

Interfaces genéricas:
interface Respuesta<T> {
  data: T;
  status: number;
  mensaje: string;
}

interface Usuario { id: number; nombre: string; }

const respuesta: Respuesta<Usuario> = {
  data: { id: 1, nombre: "Ana" },
  status: 200,
  mensaje: "OK"
};

Restricciones con extends:
interface ConId {
  id: number;
}

function obtenerPorId<T extends ConId>(items: T[], id: number): T | undefined {
  return items.find(item => item.id === id);
}

Múltiples tipos genéricos:
function mapear<T, U>(array: T[], transformar: (item: T) => U): U[] {
  return array.map(transformar);
}

const numeros = [1, 2, 3];
const strings = mapear(numeros, n => n.toString());

Clases genéricas:
class Cola<T> {
  private items: T[] = [];

  agregar(item: T): void {
    this.items.push(item);
  }

  sacar(): T | undefined {
    return this.items.shift();
  }
}

const colaNumeros = new Cola<number>();
colaNumeros.agregar(1);
colaNumeros.agregar(2);
```

5. Utility Types

TypeScript incluye tipos de utilidad para transformar tipos existentes.

Partial<T> - Hace todas las propiedades opcionales:

```
interface Usuario {
  id: number;
  nombre: string;
  email: string;
}

function actualizarUsuario(id: number, cambios: Partial<Usuario>) {
  // cambios puede tener solo algunas propiedades
}

actualizarUsuario(1, { nombre: "Nuevo nombre" });
```

Required<T> - Hace todas las propiedades requeridas:

```
interface Config {
  debug?: boolean;
  timeout?: number;
}

const configCompleta: Required<Config> = {
  debug: true,
  timeout: 5000
};
```

Readonly<T> - Hace todas las propiedades de solo lectura:

```
const usuario: Readonly<Usuario> = { id: 1, nombre: "Ana", email: "ana@ej.com" };
usuario.nombre = "Otro";  // Error: Cannot assign to 'nombre'
```

Pick<T, K> - Selecciona propiedades específicas:

```
type UsuarioBasico = Pick<Usuario, "id" | "nombre">;
// { id: number; nombre: string }
```

Omit<T, K> - Excluye propiedades:

```
type SinEmail = Omit<Usuario, "email">;
// { id: number; nombre: string }
```

Record<K, T> - Crea tipo con claves K y valores T:

```
type Roles = "admin" | "usuario" | "invitado";
type Permisos = Record<Roles, string[]>;

const permisos: Permisos = {
  admin: ["leer", "escribir", "eliminar"],
  usuario: ["leer", "escribir"],
  invitado: ["leer"]
};
```

NonNullable<T> - Excluye null y undefined:

```
type MaybeString = string | null | undefined;
type DefinitelyString = NonNullable<MaybeString>;  // string
```

6. TypeScript con React

TypeScript mejora significativamente el desarrollo con React.

Componentes funcionales:

```
interface Props {
  nombre: string;
  edad?: number;
  onClick: () => void;
}

const Saludo: React.FC<Props> = ({ nombre, edad, onClick }) => {
  return (
    <div onClick={onClick}>
      Hola, {nombre}
      {edad && <span> ({edad} años)</span>}
    </div>
  );
};

// Alternativa sin React.FC (más recomendado):
function Saludo({ nombre, edad, onClick }: Props) {
  return <div>...</div>;
}
```

useState con tipos:

```
const [usuario, setUsuario] = useState<Usuario | null>(null);
const [items, setItems] = useState<string[]>([]);
```

useRef con tipos:

```
const inputRef = useRef<HTMLInputElement>(null);
const valorRef = useRef<number>(0);
```

Eventos tipados:

```
const handleChange = (e: React.ChangeEvent<HTMLInputElement>) => {
  console.log(e.target.value);
};

const handleSubmit = (e: React.FormEvent<HTMLFormElement>) => {
  e.preventDefault();
};

const handleClick = (e: React.MouseEvent<HTMLButtonElement>) => {
  // ...
};
```

Props con children:

```
interface LayoutProps {
  children: React.ReactNode;
  titulo?: string;
}

function Layout({ children, titulo }: LayoutProps) {
  return (
    <div>
      {titulo && <h1>{titulo}</h1>}
```

```
      {children}
    </div>
  );
}
```

Custom hooks tipados:

```
function useLocalStorage<T>(key: string, initialValue: T) {
  const [value, setValue] = useState<T>(() => {
    const stored = localStorage.getItem(key);
    return stored ? JSON.parse(stored) : initialValue;
  });

  const setStoredValue = (newValue: T) => {
    setValue(newValue);
    localStorage.setItem(key, JSON.stringify(newValue));
  };

  return [value, setStoredValue] as const;
}
```