

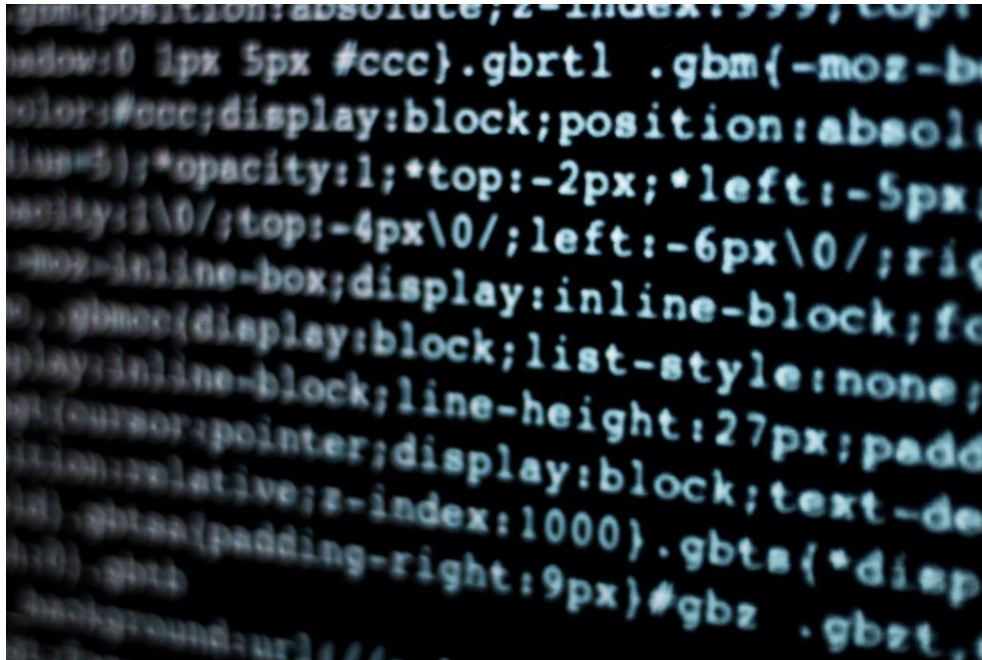
FUNDAMENTOS DEL DESARROLLO WEB EN ENTORNO CLIENTE: TYPESCRIPT Y PROGRAMACIÓN ORIENTADA A OBJETOS

Conceptos clave para crear
aplicaciones web modernas y
eficientes



INTRODUCCIÓN A TYPESCRIPT

¿QUÉ ES TYPESCRIPT Y SUS VENTAJAS FRENTE A JAVASCRIPT?



Superconjunto de JavaScript

TypeScript amplía JavaScript con características adicionales para mejorar la seguridad y mantenibilidad del código.

Tipado estático y detección de errores

Permite definir tipos de datos y detectar errores antes de la ejecución, evitando fallos en tiempo real.

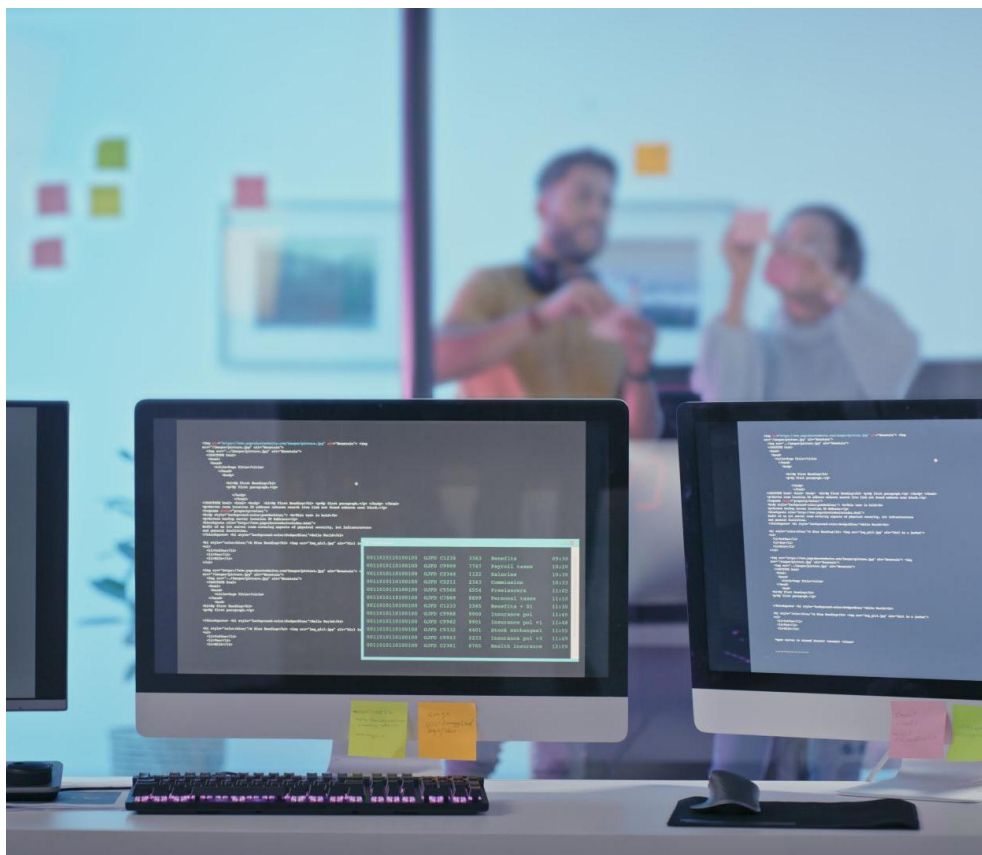
Mejora en la mantenibilidad

Uso de interfaces, clases y tipado para mejorar la estructura y claridad del código en proyectos colaborativos.

Compatibilidad y programación orientada a objetos

Compatible con JavaScript, permite usar librerías existentes y aplicar herencia y modificadores de acceso.

INSTALACIÓN, CONFIGURACIÓN Y USO CON FRAMEWORKS



Instalación y compilación básica

TypeScript se instala global o localmente, y sus archivos .ts se compilan a .js con el comando tsc.

Configuración con tsconfig.json

El archivo tsconfig.json configura opciones clave como versión de JavaScript, módulos y comprobación estricta.

Estructura del proyecto

La estructura típica incluye carpetas src para código fuente y dist para archivos generados por TypeScript.

Integración con frameworks

TypeScript se integra con Node.js, Angular, React y Vue.js, mejorando la calidad del código y robustez.

TIPOS BÁSICOS, ESTRUCTURAS DE DATOS Y FUNCIONES EN TYPESCRIPT

TIPOS PRIMITIVOS Y ESPECIALES

Tipos primitivos básicos

String, number y boolean son tipos primitivos que representan texto, números y valores lógicos respectivamente y son inmutables.

Tipos especiales null y undefined

Null y undefined indican ausencia de valor y su manejo puede configurarse con strictNullChecks para mayor seguridad.

Tipos any y unknown

Any desactiva el chequeo de tipos, mientras unknown acepta cualquier valor pero requiere verificación de tipo antes de usar.

Tipos void y never

Void es para funciones sin retorno y never para funciones que nunca terminan o lanzan errores, mejorando la predictibilidad.

```
100 m_fms = m_fms \ (ro \ (1-ro))
101 m_fms = m_fms \ (ro \ (1-ro))
102 m_fms = m_fms \ (ro \ (1-ro))
103 m_fms = m_fms \ (ro \ (1-ro))
104 m_fms = m_fms \ (ro \ (1-ro))
105 m_fms = m_fms \ (ro \ (1-ro))
106 m_fms = m_fms \ (ro \ (1-ro))
107 m_fms = m_fms \ (ro \ (1-ro))
108 m_fms = m_fms \ (ro \ (1-ro))
109 m_fms = m_fms \ (ro \ (1-ro))
110 m_fms = m_fms \ (ro \ (1-ro))
111 m_fms = m_fms \ (ro \ (1-ro))
112 m_fms = m_fms \ (ro \ (1-ro))
113 m_fms = m_fms \ (ro \ (1-ro))
114 m_fms = m_fms \ (ro \ (1-ro))
115 m_fms = m_fms \ (ro \ (1-ro))
116 m_fms = m_fms \ (ro \ (1-ro))
117 m_fms = m_fms \ (ro \ (1-ro))
118 m_fms = m_fms \ (ro \ (1-ro))
119 m_fms = m_fms \ (ro \ (1-ro))
120 m_fms = m_fms \ (ro \ (1-ro))
121 m_fms = m_fms \ (ro \ (1-ro))
122 m_fms = m_fms \ (ro \ (1-ro))
123 m_fms = m_fms \ (ro \ (1-ro))
124 m_fms = m_fms \ (ro \ (1-ro))
125 m_fms = m_fms \ (ro \ (1-ro))
126 m_fms = m_fms \ (ro \ (1-ro))
127 m_fms = m_fms \ (ro \ (1-ro))
128 m_fms = m_fms \ (ro \ (1-ro))
129 m_fms = m_fms \ (ro \ (1-ro))
130 m_fms = m_fms \ (ro \ (1-ro))
131 m_fms = m_fms \ (ro \ (1-ro))
132 m_fms = m_fms \ (ro \ (1-ro))
133 m_fms = m_fms \ (ro \ (1-ro))
134 m_fms = m_fms \ (ro \ (1-ro))
135 m_fms = m_fms \ (ro \ (1-ro))
136 m_fms = m_fms \ (ro \ (1-ro))
137 m_fms = m_fms \ (ro \ (1-ro))
138 m_fms = m_fms \ (ro \ (1-ro))
139 m_fms = m_fms \ (ro \ (1-ro))
140 m_fms = m_fms \ (ro \ (1-ro))
141 m_fms = m_fms \ (ro \ (1-ro))
142 m_fms = m_fms \ (ro \ (1-ro))
143 m_fms = m_fms \ (ro \ (1-ro))
144 m_fms = m_fms \ (ro \ (1-ro))
145 m_fms = m_fms \ (ro \ (1-ro))
146 m_fms = m_fms \ (ro \ (1-ro))
147 m_fms = m_fms \ (ro \ (1-ro))
148 m_fms = m_fms \ (ro \ (1-ro))
149 m_fms = m_fms \ (ro \ (1-ro))
150 m_fms = m_fms \ (ro \ (1-ro))
151 m_fms = m_fms \ (ro \ (1-ro))
152 m_fms = m_fms \ (ro \ (1-ro))
153 m_fms = m_fms \ (ro \ (1-ro))
154 m_fms = m_fms \ (ro \ (1-ro))
155 m_fms = m_fms \ (ro \ (1-ro))
156 m_fms = m_fms \ (ro \ (1-ro))
157 m_fms = m_fms \ (ro \ (1-ro))
158 m_fms = m_fms \ (ro \ (1-ro))
159 m_fms = m_fms \ (ro \ (1-ro))
160 m_fms = m_fms \ (ro \ (1-ro))
161 m_fms = m_fms \ (ro \ (1-ro))
162 m_fms = m_fms \ (ro \ (1-ro))
163 m_fms = m_fms \ (ro \ (1-ro))
164 m_fms = m_fms \ (ro \ (1-ro))
165 m_fms = m_fms \ (ro \ (1-ro))
166 m_fms = m_fms \ (ro \ (1-ro))
167 m_fms = m_fms \ (ro \ (1-ro))
168 m_fms = m_fms \ (ro \ (1-ro))
169 m_fms = m_fms \ (ro \ (1-ro))
170 m_fms = m_fms \ (ro \ (1-ro))
171 m_fms = m_fms \ (ro \ (1-ro))
172 m_fms = m_fms \ (ro \ (1-ro))
173 m_fms = m_fms \ (ro \ (1-ro))
174 m_fms = m_fms \ (ro \ (1-ro))
175 m_fms = m_fms \ (ro \ (1-ro))
176 m_fms = m_fms \ (ro \ (1-ro))
177 m_fms = m_fms \ (ro \ (1-ro))
178 m_fms = m_fms \ (ro \ (1-ro))
179 m_fms = m_fms \ (ro \ (1-ro))
180 m_fms = m_fms \ (ro \ (1-ro))
181 m_fms = m_fms \ (ro \ (1-ro))
182 m_fms = m_fms \ (ro \ (1-ro))
183 m_fms = m_fms \ (ro \ (1-ro))
184 m_fms = m_fms \ (ro \ (1-ro))
185 m_fms = m_fms \ (ro \ (1-ro))
186 m_fms = m_fms \ (ro \ (1-ro))
187 m_fms = m_fms \ (ro \ (1-ro))
188 m_fms = m_fms \ (ro \ (1-ro))
189 m_fms = m_fms \ (ro \ (1-ro))
190 m_fms = m_fms \ (ro \ (1-ro))
191 m_fms = m_fms \ (ro \ (1-ro))
192 m_fms = m_fms \ (ro \ (1-ro))
193 m_fms = m_fms \ (ro \ (1-ro))
194 m_fms = m_fms \ (ro \ (1-ro))
195 m_fms = m_fms \ (ro \ (1-ro))
196 m_fms = m_fms \ (ro \ (1-ro))
197 m_fms = m_fms \ (ro \ (1-ro))
198 m_fms = m_fms \ (ro \ (1-ro))
199 m_fms = m_fms \ (ro \ (1-ro))
200 m_fms = m_fms \ (ro \ (1-ro))
```



ESTRUCTURAS DE DATOS Y FUNCIONES

Estructuras de datos en TypeScript

TypeScript soporta arrays y tuplas para almacenar colecciones con tipos precisos y ordenados.

Enumeraciones para constantes

Las enumeraciones permiten definir conjuntos de constantes con nombre, mejorando la legibilidad del código.

Funciones tipadas y seguras

Funciones con parámetros y retornos tipados facilitan la validación y reducen errores en desarrollo.

Tipos literales, uniones e inferencia

Tipos literales y uniones restringen valores; la inferencia deduce tipos automáticamente.

PROGRAMACIÓN ORIENTADA A OBJETOS

CLASES, CONSTRUCTORES Y MODIFICADORES DE ACCESO

Definición de Clase

Una clase es una plantilla que define atributos y métodos para crear objetos con características específicas.

Constructores en Clases

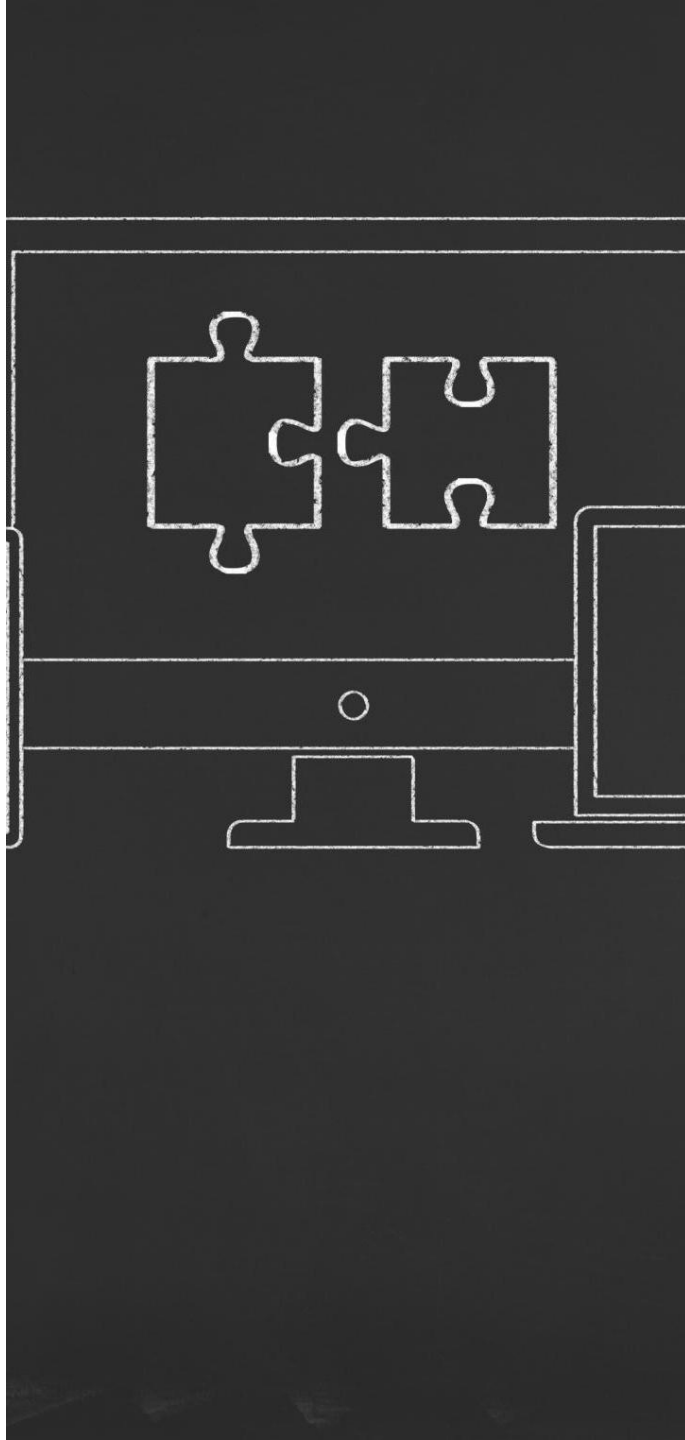
Los constructores inicializan objetos y permiten establecer valores iniciales, pudiendo ser sobrecargados.

Modificadores de Acceso

Public, private y protected controlan la visibilidad de atributos y métodos para fomentar la encapsulación.

Organización y Seguridad

Estas características facilitan código claro, seguro, reutilizable y fácil de mantener.



HERENCIA, POLIMORFISMO E INTERFACES

Concepto de Herencia

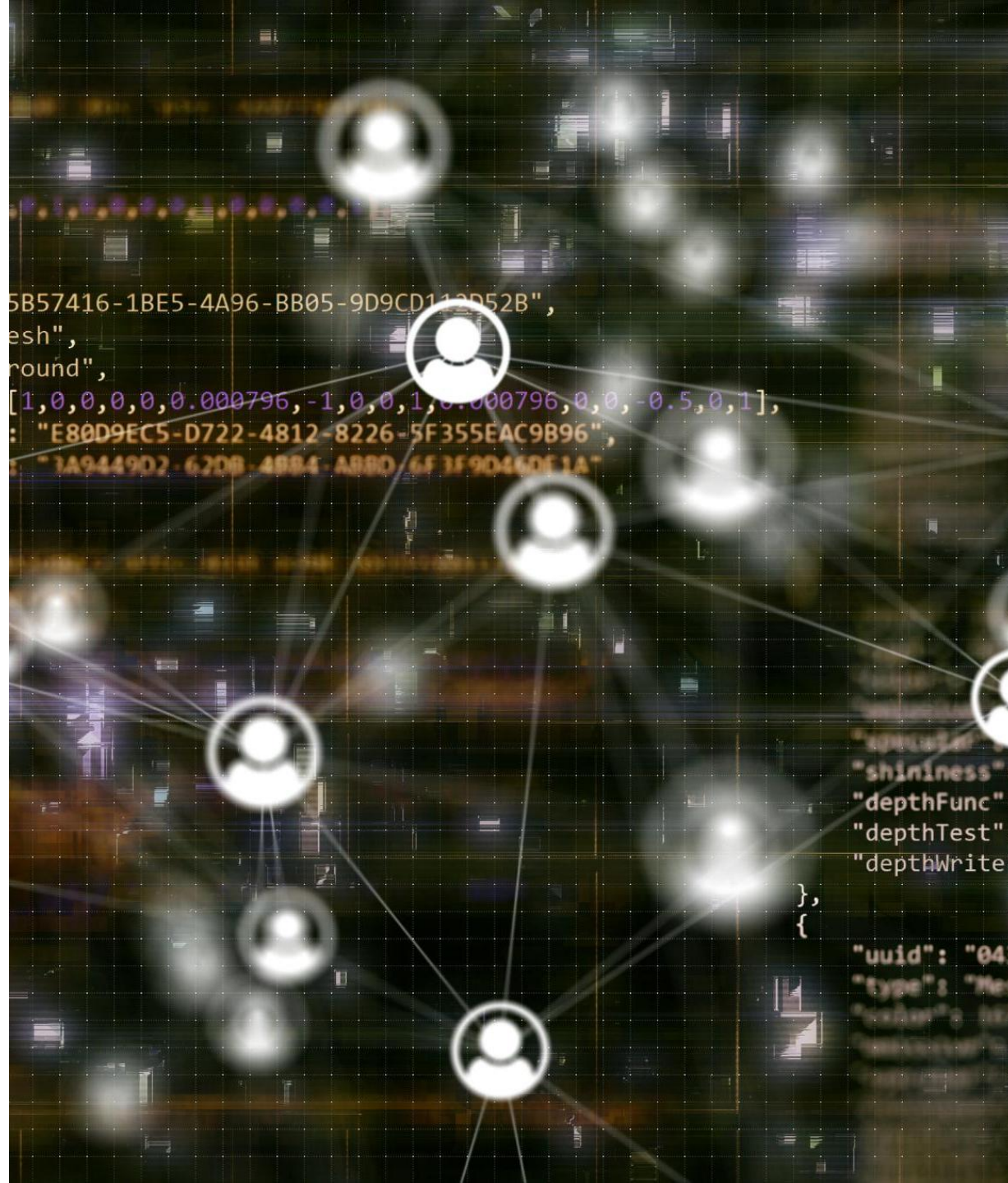
La herencia permite reutilizar atributos y métodos de una clase padre en una clase hija, creando jerarquías claras y eficientes.

Polimorfismo en programación

El polimorfismo permite que un método se comporte diferente según el objeto que lo utilice, con sobrecarga y sobrescritura.

Funciones de las Interfaces

Las interfaces definen contratos que obligan a implementar métodos, mejorando la flexibilidad y el desacoplamiento del código.



PROPIEDADES Y MÉTODOS ESTÁTICOS

Miembros estáticos de clase

Los miembros estáticos pertenecen a la clase y se comparten entre todas sus instancias sin necesidad de crear objetos.

Usos comunes

Se usan para definir constantes, contadores globales y funciones auxiliares accesibles sin instanciar la clase.

Limitaciones de métodos estáticos

Los métodos estáticos no pueden acceder directamente a atributos no estáticos, facilitando la organización y optimización del código.

