Creación de tipos a partir de otros tipos

En esta parte, vamos a cubrir algunas técnicas para adaptar un tipo a otro, como los tipos utilitarios, tipos de unión y tipos de intersección. Comenzaremos mostrando algunas funciones utilitarias útiles para este trabajo en TypeScript.

Tipos utilitarios

Los tipos utilitarios son un conjunto de tipos proporcionados por el lenguaje para transformar tipos existentes en nuevas versiones modificadas.

Ofrecen una forma conveniente de alterar propiedades de un tipo, haciéndolas opcionales, de solo lectura o excluyéndolas, entre otras transformaciones. Vamos a cubrir algunos de ellos aquí:

Pick<Type, Keys>: Pick construye un nuevo tipo seleccionando un subconjunto de propiedades de un tipo existente. Pick se usa cuando necesitas un tipo con solo propiedades específicas de un tipo principal. Esto es útil para crear tipos más enfocados y menos voluminosos. El siguiente es un ejemplo de uso de Pick:

```
interface IUser {
  id: number;
  name: string;
  email: string;
}
type UserPreview = Pick<IUser, 'id' | 'name'>;
const userPreview: UserPreview = {id: '1', name: 'John'};
```

Aquí, UserPreview contiene solo id y name de IUser.

Más ejemplos:

Ejemplo 1: Selección de propiedades para simplificar un tipo

Imagina que tienes una interfaz IProducto que tiene varias propiedades, pero en una parte de tu aplicación solo necesitas algunas de ellas.

```
interface IProducto {
   id: number;
   nombre: string;
```

```
  descripcion: string;
  precio: number;
  stock: number;
  categoria: string;
}

type ProductoResumen = Pick<IProducto, 'id' | 'nombre' | 'precio'>;

const productoResumen: ProductoResumen = {
  id: 101,
  nombre: 'Laptop',
  precio: 1500
};

console.log(productoResumen); // { id: 101, nombre: 'Laptop', precio: 1500 }
```

En este ejemplo:

- ProductoResumen contiene solo las propiedades id, nombre, y precio de la interfaz IProducto.
- Esto es útil si solo necesitas una vista previa o un resumen del producto en lugar de todas las propiedades detalladas.

Ejemplo 2: Uso de Pick para permisos limitados

Supón que tienes una interfaz de usuario completa, pero en algunas áreas de tu aplicación quieres limitar el acceso a ciertas propiedades, por ejemplo, cuando un usuario con menos permisos solo puede ver ciertos datos.

```
interface IUsuarioCompleto {
  id: number;
  nombre: string;
  email: string;
  direccion: string;
  telefono: string;
  rol: 'admin' | 'user';
}

type UsuarioBasico = Pick<IUsuarioCompleto, 'id' | 'nombre' | 'rol'>;

const usuarioBasico: UsuarioBasico = {
  id: 1,
```

```
  nombre: 'Maria',
  rol: 'user'
};

console.log(usuarioBasico); // { id: 1, nombre: 'Maria', rol: 'user' }
```

En este ejemplo:

- UsuarioBasico incluye solo las propiedades id, nombre, y rol.
- Esto es útil cuando deseas exponer solo un subconjunto de datos para ciertos usuarios, manteniendo otras propiedades como email, direccion, y telefono restringidas.

Ejemplo 3: Definir tipos para formularios o APIs

Si estás trabajando con formularios o API endpoints, a menudo solo necesitas un subconjunto de las propiedades de un objeto.

```
interface IPerfilCompleto {
  usuarioId: number;
  nombreCompleto: string;
  email: string;
  fechaNacimiento: Date;
  genero: string;
  biografia: string;
}

type PerfilEdicion = Pick<IPerfilCompleto, 'nombreCompleto' | 'email' | 'biografia'>;

const perfilEdicion: PerfilEdicion = {
  nombreCompleto: 'Carlos Pérez',
  email: 'carlos@example.com',
  biografia: 'Desarrollador de software'
};

console.log(perfilEdicion);
// { nombreCompleto: 'Carlos Pérez', email: 'carlos@example.com', biografia: 'Desarrollador de software' }
```

En este ejemplo:

- PerfilEdicion se utiliza para editar solo ciertas propiedades del perfil (nombreCompleto, email, y biografia).
- Esto puede ser útil para representar el formulario de edición del perfil del usuario en una aplicación, donde el usuario puede actualizar solo ciertos campos.

Ejemplo 4: Reducir tipos para propósitos de Log

A veces solo necesitas un conjunto mínimo de información para fines de registro (logging).

```
interface ITransaccion {
  transaccionId: string;
  fecha: Date;
  monto: number;
  metodoPago: string;
  estado: string;
  detalles: string;
}

type LogTransaccion = Pick<ITransaccion, 'transaccionId' | 'monto' | 'estado'>;

const logTransaccion: LogTransaccion = {
  transaccionId: 'TX12345',
  monto: 250,
  estado: 'completada'
};

console.log(logTransaccion);
// { transaccionId: 'TX12345', monto: 250, estado: 'completada' }
```

En este ejemplo:

- LogTransaccion contiene solo las propiedades transaccionId, monto, y estado.
- Este tipo reducido puede ser útil para propósitos de registro (logging), donde solo quieres registrar información esencial.

Ejemplo 5: Seleccionar propiedades para vistas o componentes específicos

Cuando trabajas con componentes en una aplicación de front-end, puede que solo necesites pasar ciertas propiedades a un componente específico.

```
interface IArticulo {
  id: number;
  titulo: string;
  contenido: string;
  autor: string;
  fechaPublicacion: Date;
  etiquetas: string[];
}

type ArticuloResumen = Pick<IArticulo, 'id' | 'titulo' | 'fechaPublicacion'>;

const articuloResumen: ArticuloResumen = {
  id: 789,
  titulo: 'Introducción a TypeScript',
  fechaPublicacion: new Date('2024-09-19')
};

console.log(articuloResumen);
// { id: 789, titulo: 'Introducción a TypeScript', fechaPublicacion: 2024-09-19T00:00:00.000Z }
```

En este ejemplo:

- ArticuloResumen contiene solo id, titulo, y fechaPublicacion.
- Este tipo podría ser útil para un componente que solo muestra una vista previa del artículo, sin necesidad de detalles adicionales como contenido o etiquetas.

Record<Keys, Type>: Record genera un tipo con un conjunto de claves y asigna un tipo uniforme a los valores de estas claves. Es ideal para crear objetos donde las claves tienen un tipo de valor común, a menudo utilizado para propósitos de mapeo o búsqueda. Aquí hay un ejemplo de uso de Record:

```
type UserNamesById = Record<UserId, string>;
const userNamesById: UserNamesById = {'1': 'John', '2': 'Alice'};
```

UserNamesById es un objeto diccionario que asigna claves UserId a nombres de tipo string.

Más ejemplos:

Ejemplo 1: Mapeo de estados con mensajes

Usando Record para crear un mapeo de estados de una aplicación a mensajes de estado específicos.

```
type Estados = 'loading' | 'success' | 'error';
type MensajesPorEstado = Record<Estados, string>;

const mensajes: MensajesPorEstado = {
  loading: 'Cargando...',
  success: 'Operación completada con éxito.',
  error: 'Ha ocurrido un error.'
};

console.log(mensajes.loading); // "Cargando..."
console.log(mensajes.error);   // "Ha ocurrido un error."
```

En este ejemplo:

- Estados es un tipo que define las posibles claves ('loading', 'success', 'error').
- MensajesPorEstado es un Record que asigna a cada estado un mensaje de tipo string.
- mensajes es un objeto que contiene un mensaje para cada estado.

Ejemplo 2: Configuración de temas de estilo

Usando Record para definir un conjunto de propiedades de estilo para temas en una aplicación.

```
type Tema = 'light' | 'dark';
type ConfiguracionTema = Record<Tema, { backgroundColor: string; color: string }>;

const temas: ConfiguracionTema = {
  light: { backgroundColor: '#ffffff', color: '#000000' },
  dark: { backgroundColor: '#000000', color: '#ffffff' }
};

console.log(temas.light.backgroundColor); // "#ffffff"
console.log(temas.dark.color);          // "#ffffff"
```

En este ejemplo:

- Tema define las posibles claves ('light' y 'dark').
- ConfiguracionTema es un Record que mapea cada tema a un objeto que contiene las propiedades backgroundColor y color.
- temas contiene la configuración específica para cada tema.

Ejemplo 3: Mapeo de eventos a manejadores

Usando Record para mapear nombres de eventos a funciones manejadoras.

```
type Eventos = 'click' | 'hover' | 'submit';
type ManejadoresDeEventos = Record<Eventos, () => void>;

const manejadores: ManejadoresDeEventos = {
  click: () => console.log('Se hizo clic.'),
  hover: () => console.log('Se pasó el cursor.'),
  submit: () => console.log('Se envió el formulario.')
};

// Uso de los manejadores
manejadores.click();   // "Se hizo clic."
manejadores.hover();   // "Se pasó el cursor."
```

En este ejemplo:

- Eventos define un conjunto de nombres de eventos.
- ManejadoresDeEventos es un Record que mapea cada evento a una función sin parámetros (() => void).
- manejadores es un objeto que contiene una función manejadora para cada evento.

Ejemplo 4: Diccionario de traducciones

Usando Record para definir un diccionario de traducciones entre idiomas.

```
type Idiomas = 'es' | 'en' | 'fr';
type Traducciones = Record<Idiomas, string>;

const saludos: Traducciones = {
  es: 'Hola',
  en: 'Hello',
  fr: 'Bonjour'
};

console.log(saludos.es); // "Hola"
console.log(saludos.en); // "Hello"
console.log(saludos.fr); // "Bonjour"
```

En este ejemplo:

- Idiomas define las claves ('es', 'en', 'fr').
- Traducciones es un Record que mapea cada idioma a una cadena de saludo.
- saludos contiene un saludo para cada idioma.

Ejemplo 5: Configuración de roles y permisos

Usando Record para mapear roles de usuario a permisos específicos.

```
type Roles = 'admin' | 'user' | 'guest';
type Permisos = Record<Roles, { read: boolean; write: boolean; delete: boolean }>;

const permisos: Permisos = {
   admin: { read: true, write: true, delete: true },
   user: { read: true, write: true, delete: false },
   guest: { read: true, write: false, delete: false }
};

console.log(permisos.admin.write);  // true
console.log(permisos.guest.delete); // false
```

En este ejemplo:

- Roles define un conjunto de roles ('admin', 'user', 'guest').
- Permisos es un Record que mapea cada rol a un objeto que define sus permisos (read, write, delete).
- permisos es un objeto que contiene los permisos para cada rol.

Ejemplo 6: Mapeo de configuraciones de ruta en una aplicación

Usando Record para definir rutas de una aplicación y su configuración.

```
type Rutas = 'home' | 'about' | 'contact';
type ConfiguracionRuta = Record<Rutas, { path: string; componente: string }>;

const rutas: ConfiguracionRuta = {
   home: { path: '/', componente: 'HomeComponent' },
   about: { path: '/about', componente: 'AboutComponent' },
   contact: { path: '/contact', componente: 'ContactComponent' }
};
```

```
console.log(rutas.home.path); // "/"
console.log(rutas.contact.componente); // "ContactComponent"
```

En este ejemplo:

- Rutas define las claves ('home', 'about', 'contact').
- ConfiguracionRuta es un Record que mapea cada ruta a un objeto que define el path y el componente.
- rutas es un objeto que contiene la configuración para cada ruta de la aplicación.

Partial<Type>: Partial convierte todas las propiedades de un tipo dado en propiedades opcionales. Partial es útil en situaciones como la actualización de partes de un objeto, donde no necesitas proporcionar todas las propiedades. Proporciona flexibilidad en la creación de objetos. Así es como usarías Partial en código:

```
type PartialIUser = Partial<IUser>;
const partialUser: PartialIUser = {id: '1'};
```

PartialIUser permite cualquier combinación de propiedades de IUser, incluidos objetos incompletos, porque todas las propiedades son opcionales.
Más ejemplos:

Ejemplo 1: Actualización parcial de un objeto

Usando Partial para actualizar solo algunas propiedades de un objeto.

```
interface PerfilUsuario {
  id: number;
  nombre: string;
  email: string;
  edad: number;
}

function actualizarPerfil(id: number, actualizaciones: Partial<PerfilUsuario>): PerfilUsuario {
  const perfilExistente: PerfilUsuario = { id, nombre: 'Juan', email: 'juan@example.com', edad: 30 };

  return { ...perfilExistente, ...actualizaciones };
}
```

```
// Actualizar solo el nombre y la edad del usuario
const perfilActualizado = actualizarPerfil(1, { nombre: 'Carlos', edad: 35 });
console.log(perfilActualizado);
// { id: 1, nombre: 'Carlos', email: 'juan@example.com', edad: 35 }
```

En este ejemplo:

- Partial<PerfilUsuario> permite pasar solo las propiedades que deseas actualizar (nombre y edad en este caso).
- La función actualizarPerfil utiliza el operador de propagación (...) para combinar el objeto existente con las actualizaciones proporcionadas.

Ejemplo 2: Configuración de opciones en una función

Usando Partial para permitir que una función acepte una configuración parcial.

```
interface ConfiguracionServidor {
  host: string;
  puerto: number;
  ssl: boolean;
}

function iniciarServidor(config: Partial<ConfiguracionServidor>) {
  const configuracionPredeterminada: ConfiguracionServidor = {
    host: 'localhost',
    puerto: 8080,
    ssl: false
  };

  // Combina la configuración predeterminada con la configuración proporcionada
  const configuracionFinal = { ...configuracionPredeterminada, ...config };
  console.log(`Iniciando servidor en ${configuracionFinal.host}:${configuracionFinal.puerto} con SSL:
${configuracionFinal.ssl}`);
}

// Iniciar el servidor solo cambiando el puerto
iniciarServidor({ puerto: 3000 });
// "Iniciando servidor en localhost:3000 con SSL: false"
```

En este ejemplo:

- Partial<ConfiguracionServidor> permite pasar solo las opciones de configuración que deseas cambiar (puerto en este caso).
- Esto hace que la función iniciarServidor sea más flexible y fácil de usar.

Ejemplo 3: Formularios de actualización en una aplicación

Usando Partial para manejar formularios de actualización donde solo algunos campos pueden ser modificados.

```
interface DatosPerfil {
  nombre: string;
  email: string;
  telefono: string;
}

function manejarEnvioFormulario(datos: Partial<DatosPerfil>) {
  console.log('Actualizando los siguientes datos:', datos);
}

// Simula el envío del formulario solo con algunos campos modificados
manejarEnvioFormulario({ nombre: 'Ana' });
// "Actualizando los siguientes datos: { nombre: 'Ana' }"
```

En este ejemplo:

- Partial<DatosPerfil> permite que el formulario envíe solo los campos que han sido modificados (nombre en este caso).
- Esto es útil en formularios donde los usuarios pueden optar por no cambiar todos los campos.

Ejemplo 4: Creación de objetos con inicialización parcial

Usando Partial para crear objetos que serán inicializados en diferentes etapas.

```
interface Pedido {
  id: number;
  producto: string;
  cantidad: number;
  precioTotal: number;
}

let nuevoPedido: Partial<Pedido> = {};
```

```
nuevoPedido.producto = 'Laptop';
nuevoPedido.cantidad = 2;

// Inicialización completa del objeto
nuevoPedido = {
  ...nuevoPedido,
  id: 101,
  precioTotal: 1500
};

console.log(nuevoPedido);
// { producto: 'Laptop', cantidad: 2, id: 101, precioTotal: 1500 }
```

En este ejemplo:

- Partial<Pedido> permite que nuevoPedido se construya en varias etapas, comenzando con un objeto vacío y agregando propiedades según sea necesario.
- Esto es útil para objetos que se construyen progresivamente.

Ejemplo 5: Función para limpiar propiedades opcionales

Usando Partial en una función que limpia las propiedades opcionales de un objeto.

```
interface Config {
  tema: string;
  mostrarBarraNavegacion: boolean;
  mostrarBarraLateral: boolean;
}

function limpiarPropiedades(config: Partial<Config>): Config {
  const configuracionPorDefecto: Config = {
    tema: 'claro',
    mostrarBarraNavegacion: true,
    mostrarBarraLateral: true
  };

  // Remueve las propiedades no definidas
  return { ...configuracionPorDefecto, ...config };
}

// Llamando a la función sin cambiar el tema
```

```
const configuracionLimpia = limpiarPropiedades({ mostrarBarraLateral: false });
console.log(configuracionLimpia);
// { tema: 'claro', mostrarBarraNavegacion: true, mostrarBarraLateral: false }
```

En este ejemplo:

- Partial<Config> permite pasar solo algunas propiedades a la función limpiarPropiedades.
- Esto hace que sea posible limpiar u omitir propiedades no definidas y rellenar con valores predeterminados.

Ejemplo 6: Construcción de objetos de configuración

Usando Partial para permitir la construcción dinámica de un objeto de configuración a lo largo del tiempo.

```
interface Opciones {
  modo: 'auto' | 'manual';
  intervalo: number;
  activo: boolean;
}

let opcionesParciales: Partial<Opciones> = {};
opcionesParciales.modo = 'auto';

// Configurando más tarde
opcionesParciales.intervalo = 3000;
opcionesParciales.activo = true;

console.log(opcionesParciales);
// { modo: 'auto', intervalo: 3000, activo: true }
```

En este ejemplo:

- Partial<Opciones> permite que opcionesParciales se configure en pasos separados.
- Esto es útil cuando la configuración se recopila de diferentes fuentes o etapas.

Required<Type>: Required transforma todas las propiedades opcionales de un tipo en obligatorias. Required es lo opuesto a Partial. Impone que todas las propiedades del tipo deben ser proporcionadas, asegurando definiciones completas de objetos. El siguiente es un ejemplo de uso de Required:

```
type RequiredIUser = Required<PartialIUser>;
const requiredUser: RequiredIUser = {id: '1', name: 'John', email: 'john@example.com'};
```

RequiredIUser exige que todas las propiedades, incluso aquellas opcionales en PartialIUser, deben estar presentes.

Más ejemplos:

Ejemplo 1: Convertir propiedades opcionales en obligatorias

Supongamos que tienes un tipo que contiene propiedades opcionales, y deseas asegurarte de que todas esas propiedades sean obligatorias.

```
interface ConfiguracionOpcional {
  host?: string;
  puerto?: number;
  ssl?: boolean;
}

// Convertir todas las propiedades opcionales en obligatorias
type ConfiguracionCompleta = Required<ConfiguracionOpcional>;

const configuracion: ConfiguracionCompleta = {
  host: 'localhost',
  puerto: 8080,
  ssl: true
};

console.log(configuracion);
// { host: 'localhost', puerto: 8080, ssl: true }
```

En este ejemplo:

- ConfiguracionOpcional tiene todas sus propiedades opcionales (host, puerto, ssl).
- ConfiguracionCompleta usa Required para convertir todas las propiedades en obligatorias.
- configuracion ahora requiere que todas las propiedades estén presentes al crear el objeto.

Ejemplo 2: Usar required para asegurar la completitud en objetos

Este ejemplo muestra cómo usar Required para garantizar que todas las propiedades estén definidas en un objeto antes de su uso.

```
interface UsuarioParcial {
  id: string;
  nombre?: string;
  email?: string;
}

type UsuarioCompleto = Required<UsuarioParcial>;

function mostrarDetallesUsuario(usuario: UsuarioCompleto) {
  console.log(`ID: ${usuario.id}`);
  console.log(`Nombre: ${usuario.nombre}`);
  console.log(`Email: ${usuario.email}`);
}

// `Required` obliga a incluir todas las propiedades opcionales
const usuario: UsuarioCompleto = {
  id: '123',
  nombre: 'Carlos',
  email: 'carlos@example.com'
};

mostrarDetallesUsuario(usuario);
// ID: 123
// Nombre: Carlos
// Email: carlos@example.com
```

En este ejemplo:

- UsuarioParcial tiene propiedades opcionales (nombre y email).
- UsuarioCompleto hace que todas las propiedades sean obligatorias usando Required.
- Al llamar a mostrarDetallesUsuario, se garantiza que todas las propiedades están presentes.

Ejemplo 3: Asegurar la Configuración Completa en un Componente

En situaciones donde es crucial tener una configuración completa, como en un componente UI, Required puede asegurar que todos los valores están presentes.

```
interface OpcionesBoton {
  texto?: string;
  color?: string;
  deshabilitado?: boolean;
}

type OpcionesBotonCompleto = Required<OpcionesBoton>;

function crearBoton(opciones: OpcionesBotonCompleto) {
  console.log(`Creando botón: [Texto: ${opciones.texto}, Color: ${opciones.color}, Deshabilitado:
${opciones.deshabilitado}]`);
}

// Debe proporcionar todas las propiedades
const botonOpciones: OpcionesBotonCompleto = {
  texto: 'Enviar',
  color: 'azul',
  deshabilitado: false
};

crearBoton(botonOpciones);
// Creando botón: [Texto: Enviar, Color: azul, Deshabilitado: false]
```

En este ejemplo:

- OpcionesBoton tiene todas sus propiedades opcionales (texto, color, deshabilitado).
- OpcionesBotonCompleto convierte todas esas propiedades en obligatorias usando Required.
- La función crearBoton requiere que todas las opciones sean proporcionadas.

Ejemplo 4: Garantizar la compleción de los datos en una función

Required puede ser útil en funciones que procesan datos, asegurando que los datos estén completos antes de usarlos.

```
interface DatosPersonales {
  nombre?: string;
```

```
  apellido?: string;
  edad?: number;
}

type DatosCompletos = Required<DatosPersonales>;

function procesarDatos(datos: DatosCompletos) {
  console.log(`Nombre completo: ${datos.nombre} ${datos.apellido}, Edad: ${datos.edad}`);
}

// Crear un objeto con todas las propiedades necesarias
const datos: DatosCompletos = {
  nombre: 'Ana',
  apellido: 'García',
  edad: 28
};

procesarDatos(datos);
// Nombre completo: Ana García, Edad: 28
```

En este ejemplo:

- DatosPersonales contiene propiedades opcionales (nombre, apellido, edad).
- DatosCompletos convierte todas esas propiedades en obligatorias usando Required.
- La función procesarDatos puede ahora confiar en que todas las propiedades están presentes.

Ejemplo 5: Inicialización completa de configuraciones

Imagina que tienes una configuración opcional para inicializar una aplicación, pero antes de usarla, deseas asegurar que todas las propiedades estén presentes.

```
interface ConfigApp {
  modo?: 'desarrollo' | 'produccion';
  version?: string;
  mostrarErrores?: boolean;
}

type ConfigAppCompleta = Required<ConfigApp>;

function inicializarApp(config: ConfigAppCompleta) {
```

```
  console.log(`Inicializando aplicación en modo: ${config.modo}, versión: ${config.version}, mostrar errores: ${config.mostrarErrores}`);
}

// La configuración ahora es obligatoria y completa
const configApp: ConfigAppCompleta = {
  modo: 'desarrollo',
  version: '1.0.0',
  mostrarErrores: true
};

inicializarApp(configApp);
// Inicializando aplicación en modo: desarrollo, versión: 1.0.0, mostrar errores: true
```

En este ejemplo:

- ConfigApp tiene propiedades opcionales (modo, version, mostrarErrores).
- ConfigAppCompleta usa Required para hacer que todas las propiedades sean obligatorias.
- La función inicializarApp requiere ahora una configuración completa.

Omit<Type, Keys>: Omit crea un nuevo tipo omitiendo propiedades específicas de un tipo existente. Omit es útil para crear un tipo que excluya ciertas propiedades de otro tipo, lo que es particularmente útil para excluir propiedades sensibles o innecesarias. Un ejemplo de Omit es el siguiente:

```
type UserWithoutEmail = Omit<IUser, 'email'>;
const userWithoutEmail: UserWithoutEmail = {id: '2', name: 'Alice'};
```

UserWithoutEmail es un tipo similar a IUser pero sin la propiedad email.

El tipo Omit<Type, Keys> en TypeScript permite crear un nuevo tipo que excluye propiedades específicas de un tipo existente. Esto es útil cuando necesitas trabajar con un subconjunto de las propiedades de un tipo, excluyendo aquellas que no son necesarias o que deben ser omitidas por razones de seguridad, privacidad, o simplicidad. A continuación, se presentan más ejemplos que ilustran cómo Omit se puede utilizar en diferentes escenarios:

Ejemplo 1: Excluir propiedades sensibles de un tipo

Supongamos que tienes una interfaz que representa un usuario y quieres omitir información sensible como la contraseña.

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

// Crear un tipo que omite la propiedad 'password'
type UsuarioSinPassword = Omit<Usuario, 'password'>;

const usuario: UsuarioSinPassword = {
  id: '1',
  nombre: 'Juan',
  email: 'juan@example.com'
};

console.log(usuario);
// { id: '1', nombre: 'Juan', email: 'juan@example.com' }
```

En este ejemplo:

- UsuarioSinPassword es un nuevo tipo que omite la propiedad password de la interfaz Usuario.
- Esto puede ser útil cuando deseas enviar información de usuario a la interfaz sin exponer datos sensibles como la contraseña.

Ejemplo 2: Crear un tipo de vista previa

Usando Omit para crear un tipo que excluye propiedades no necesarias para una vista previa.

```
interface Producto {
  id: number;
  nombre: string;
  descripcion: string;
  precio: number;
  stock: number;
}

// Crear un tipo de vista previa del producto, omitiendo 'descripcion' y 'stock'
type ProductoPreview = Omit<Producto, 'descripcion' | 'stock'>;

const producto: ProductoPreview = {
```

```
  id: 101,
  nombre: 'Laptop',
  precio: 1500
};

console.log(producto);
// { id: 101, nombre: 'Laptop', precio: 1500 }
```

En este ejemplo:

- ProductoPreview es un nuevo tipo que omite las propiedades descripcion y stock de la interfaz Producto.
- Este tipo puede ser útil cuando solo necesitas mostrar una vista previa o un resumen del producto sin incluir detalles adicionales.

Ejemplo 3: Crear un tipo de solo lectura para la interfaz de usuario

Imagina que quieres mostrar los detalles de un usuario en la interfaz, pero deseas excluir propiedades que no deberían ser editables o visibles.

```
interface UsuarioCompleto {
  id: string;
  nombre: string;
  email: string;
  direccion: string;
  telefono: string;
}

// Crear un tipo de solo lectura para la interfaz, omitiendo 'direccion' y 'telefono'
type UsuarioSoloLectura = Omit<UsuarioCompleto, 'direccion' | 'telefono'>;

const usuarioSoloLectura: UsuarioSoloLectura = {
  id: '2',
  nombre: 'Maria',
  email: 'maria@example.com'
};

console.log(usuarioSoloLectura);
// { id: '2', nombre: 'Maria', email: 'maria@example.com' }
```

En este ejemplo:

- UsuarioSoloLectura es un nuevo tipo que omite las propiedades direccion y telefono de UsuarioCompleto.
- Esto puede ser útil para crear una vista de solo lectura que no incluya información privada o no editable.

Ejemplo 4: Excluir propiedades para configuración de componentes

Usando Omit para simplificar la configuración de un componente excluyendo propiedades no necesarias.

```
interface ConfiguracionComponente {
  width: number;
  height: number;
  backgroundColor: string;
  borderRadius: number;
  shadow: boolean;
}

// Crear un tipo de configuración simplificada sin 'borderRadius' y 'shadow'
type ConfiguracionSimple = Omit<ConfiguracionComponente, 'borderRadius' | 'shadow'>;

const config: ConfiguracionSimple = {
  width: 300,
  height: 200,
  backgroundColor: '#fff'
};

console.log(config);
// { width: 300, height: 200, backgroundColor: '#fff' }
```

En este ejemplo:

- ConfiguracionSimple omite borderRadius y shadow de ConfiguracionComponente.
- Esto puede ser útil para configurar un componente donde solo necesitas las dimensiones y el color de fondo.

Ejemplo 5: Excluir propiedades para validación de formularios

Usando Omit para excluir propiedades no necesarias en la validación de un formulario.

```
interface DatosFormulario {
  nombre: string;
  email: string;
  password: string;
  confirmPassword: string;
}

// Crear un tipo que excluye 'confirmPassword' para la validación
type DatosFormularioValidacion = Omit<DatosFormulario, 'confirmPassword'>;

const datosParaValidacion: DatosFormularioValidacion = {
  nombre: 'Carlos',
  email: 'carlos@example.com',
  password: 'secreto'
};

console.log(datosParaValidacion);
// { nombre: 'Carlos', email: 'carlos@example.com', password: 'secreto' }
```

En este ejemplo:

- DatosFormularioValidacion omite confirmPassword de DatosFormulario.
- Esto puede ser útil cuando solo necesitas validar los campos principales, excluyendo aquellos que son secundarios o que se manejan de manera diferente.

Ejemplo 6: Creación de tipos para registro de Logs

Usando Omit para excluir datos sensibles o innecesarios al crear registros de logs.

```
interface Transaccion {
  transaccionId: string;
  usuarioId: string;
  monto: number;
  metodoPago: string;
  numeroTarjeta: string;
  fecha: Date;
}

// Crear un tipo de registro de logs que excluye 'numeroTarjeta'
type LogTransaccion = Omit<Transaccion, 'numeroTarjeta'>;
```

```
const log: LogTransaccion = {
  transaccionId: 'TX12345',
  usuarioId: 'USR001',
  monto: 100,
  metodoPago: 'tarjeta',
  fecha: new Date()
};

console.log(log);
// { transaccionId: 'TX12345', usuarioId: 'USR001', monto: 100, metodoPago: 'tarjeta', fecha:
<fecha_actual> }
```

En este ejemplo:

- LogTransaccion omite numeroTarjeta de Transaccion para evitar almacenar datos sensibles en los registros de logs.
- Esto ayuda a mantener la privacidad y seguridad de la información de los usuarios.

Readonly<Type>: Readonly hace que todas las propiedades de un tipo sean inmutables después de la creación. Readonly se usa para crear tipos donde las propiedades de los objetos no deberían cambiarse después de que el objeto es creado, lo cual es importante para mantener la integridad en ciertos objetos:

```
type ReadonlyIUser = Readonly<IUser>;
const user: ReadonlyIUser = {id: '1', name: 'John', email: 'john@example.com'};
```

ReadonlyIUser garantiza que una vez que se crea un objeto IUser, sus propiedades no puedan modificarse, ya que no podemos asignar un nuevo campo posteriormente.

Más ejemplos:

Ejemplo 1: Proteger la configuración de una aplicación

Supongamos que tienes una configuración global para tu aplicación que no debe ser modificada después de ser establecida.

```
interface ConfiguracionApp {
  apiUrl: string;
  version: string;
  entorno: 'producción' | 'desarrollo';
```

```
}

type ConfiguracionAppInmutable = Readonly<ConfiguracionApp>;

const configuracion: ConfiguracionAppInmutable = {
  apiUrl: 'https://api.ejemplo.com',
  version: '1.0.0',
  entorno: 'producción'
};

// Intentar modificar la configuración resultará en un error de compilación
// configuracion.apiUrl = 'https://nueva-api.ejemplo.com'; // Error: Cannot assign to 'apiUrl' because it is a read-only property.
console.log(configuracion);
```

En este ejemplo:

- ConfiguracionAppInmutable es un tipo inmutable que impide la modificación de sus propiedades después de la creación.
- Esto asegura que la configuración global de la aplicación no pueda ser alterada accidentalmente.

Ejemplo 2: Proteger un objeto de configuración de estilos

En una aplicación UI, es posible que tengas una configuración de estilos que no deba ser modificada una vez definida.

```
interface Estilos {
  colorFondo: string;
  colorTexto: string;
  tamañoFuente: number;
}

const estilosPredeterminados: Readonly<Estilos> = {
  colorFondo: '#ffffff',
  colorTexto: '#000000',
  tamañoFuente: 16
};

// Intentar modificar las propiedades resultará en un error
// estilosPredeterminados.colorTexto = '#333333'; // Error: Cannot assign to 'colorTexto' because it is a read-only property.
```

```
console.log(estilosPredeterminados);
```

En este ejemplo:

- estilosPredeterminados es un objeto de configuración de estilos que se vuelve inmutable usando Readonly.
- Esto garantiza que los estilos predeterminados no sean alterados durante el ciclo de vida de la aplicación.

Ejemplo 3: Prevenir modificaciones a constantes de dominio

En muchas aplicaciones, hay objetos que representan constantes del dominio y no deben ser modificados.

```
interface Punto {
  x: number;
  y: number;
}

const origen: Readonly<Punto> = {
  x: 0,
  y: 0
};

// Intentar modificar las coordenadas resultará en un error
// origen.x = 5; // Error: Cannot assign to 'x' because it is a read-only property.
console.log(origen); // { x: 0, y: 0 }
```

En este ejemplo:

- origen es un objeto Punto que representa el punto de origen (0, 0).
- Usando Readonly, se garantiza que las coordenadas del origen no puedan ser modificadas.

Ejemplo 4: Uso de Readonly en configuraciones de ruta

En aplicaciones de enrutamiento, es posible que desees asegurarte de que las rutas sean constantes y no puedan ser modificadas.

```
interface Ruta {
  nombre: string;
  path: string;
```

```
}

const rutas: ReadonlyArray<Ruta> = [
  { nombre: 'inicio', path: '/' },
  { nombre: 'acerca', path: '/acerca' },
  { nombre: 'contacto', path: '/contacto' }
];

// Intentar modificar una ruta o agregar nuevas rutas resultará en un error
// rutas.push({ nombre: 'blog', path: '/blog' }); // Error: Property 'push' does not exist on type 'readonly
Ruta[]'.
console.log(rutas);
```

En este ejemplo:

- rutas es un arreglo de objetos Ruta que se define como ReadonlyArray.
- Esto impide cualquier modificación al arreglo, como agregar, eliminar o cambiar elementos.

Ejemplo 5: Proteger configuraciones sensibles

Cuando trabajas con configuraciones sensibles, como credenciales o claves de API, es importante proteger estos datos.

```
interface Credenciales {
  apiKey: string;
  secret: string;
}

const credenciales: Readonly<Credenciales> = {
  apiKey: '12345-abcde',
  secret: 's3cr3t'
};

// Intentar modificar las credenciales resultará en un error
// credenciales.apiKey = 'nueva-clave'; // Error: Cannot assign to 'apiKey' because it is a read-only
property.
console.log(credenciales);
```

En este ejemplo:

- credenciales es un objeto que contiene una apiKey y un secret.
- Usando Readonly, se asegura que estos valores no puedan ser alterados después de ser inicializados.

Ejemplo 6: Objetos anidados con readonly

Usando Readonly para hacer que un objeto con propiedades anidadas sea inmutable.

```
interface Direccion {
  calle: string;
  ciudad: string;
  pais: string;
}

interface Persona {
  nombre: string;
  direccion: Direccion;
}

const persona: Readonly<Persona> = {
  nombre: 'Carlos',
  direccion: {
    calle: 'Calle Principal',
    ciudad: 'Ciudad',
    pais: 'País'
  }
};

// Intentar modificar las propiedades del objeto resultará en un error
// persona.nombre = 'Ana'; // Error: Cannot assign to 'nombre' because it is a read-only property.
// persona.direccion.ciudad = 'Nueva Ciudad'; // Error: Cannot assign to 'ciudad' because it is a read-only property.
console.log(persona);
```

En este ejemplo:

- persona es un objeto con una propiedad anidada direccion.
- Usando Readonly, se garantiza que tanto las propiedades directas como las anidadas sean inmutables.

Tipos de unión

Los tipos de unión, representados por un signo de barra vertical (|), permiten que una variable contenga valores que son una combinación de dos o más tipos, ofreciendo flexibilidad para definir tipos que pueden aceptar múltiples tipos de valores específicos. Son esenciales en escenarios donde una variable o tipo de retorno de función no está confinada a un solo tipo.

Apliquemos este concepto a los ejemplos proporcionados y creemos un tipo para nuestra interfaz IMessage:

```
type MessageType = "user" | "system";
```

Aquí, MessageType es un tipo de unión, lo que significa que puede contener un valor "user" o "system". Ahora, agreguemos esto a nuestra interfaz, como se muestra aquí:

```
interface IMessage {
   type: MessageType;
   // otras propiedades
}
```

En la interfaz IMessage, la propiedad type debe ser "user" o "system", adheriéndose al tipo de unión MessageType. Sin embargo, los tipos de unión no tienen que ser valores primitivos. Veamos ahora cómo podemos manejar el retorno de un tipo de unión desde una función. La función getChatFromDb aquí ilustra un caso de uso común para los tipos de unión en los valores de retorno de funciones:

```
type DbChatSuccessResponse = {
   success: true;
   data: IChat;
};
type DbChatErrorResponse = {
   success: false;
   error: string;
};
function getChatFromDb(chatId: string): DbChatSuccessResponse | DbChatErrorResponse {
   const findChatById = (_: string) => ({} as IChat)
   const chat = findChatById(chatId);
   if (chat) {
      return {
         success: true,
         data: chat,
      };
```

```
  } else {
    return {
      success: false,
      error: "Chat not found in the database",
    };
  }
}
```

getChatFromDb puede devolver DbChatSuccessResponse, que representa una operación exitosa, o DbChatErrorResponse, que tiene propiedades para una respuesta fallida. Este enfoque es útil para el manejo de errores y la obtención de datos, donde el resultado puede diferir significativamente. Ahora, manejemos el resultado del tipo de unión de la función con reducción en el siguiente código:

```
const dbResponse = getChatFromDb("chat123");
if (dbResponse.success === true) {
  console.log("Chat data:", dbResponse.data);
} else {
  console.error("Error:", dbResponse.error);
}
```

En este fragmento, la respuesta de getChatFromDb es un objeto de éxito o un objeto de error. La verificación if dbResponse.success === true distingue efectivamente entre estos dos posibles tipos de retorno. Si success es true, TypeScript entiende que dbResponse se ajusta a DbChatSuccessResponse y permite el acceso a dbResponse.data. De lo contrario, trata a dbResponse como DbChatErrorResponse, exponiendo la propiedad error.

Ejemplo 1: Tipos de unión con parámetros de función

Puedes usar un tipo de unión para definir qué tipos puede aceptar un parámetro de función:

```
function formatInput(input: string | number): string {
  if (typeof input === "string") {
    return `Input is a string: ${input.toUpperCase()}`;
  } else {
    return `Input is a number: ${input.toFixed(2)}`;
  }
}

console.log(formatInput("hello"));  // Output: Input is a string: HELLO
console.log(formatInput(42.123));   // Output: Input is a number: 42.12
```

En este ejemplo, el parámetro input puede ser un **string** o un **number**, y el comportamiento de la función cambia según el tipo del argumento.

Ejemplo 2: Tipos de unión en propiedades de una interfaz

Los tipos de unión también pueden ser útiles para propiedades en una interfaz, permitiendo que un objeto tenga más flexibilidad en los tipos de sus propiedades.

```
interface Product {
   id: number;
   name: string;
   price: number | string; // Puede ser un número o una cadena (ej. "Gratuito")
}

const product1: Product = {
   id: 1,
   name: "Laptop",
   price: 1000
};

const product2: Product = {
   id: 2,
   name: "Software",
   price: "Gratuito"
};
```

Aquí, la propiedad price puede ser tanto un **number** como un **string**, permitiendo manejar diferentes representaciones de precio.

Ejemplo 3: Tipos de unión con tipos más complejos

El siguiente ejemplo ilustra cómo puedes utilizar tipos de unión con diferentes tipos complejos, como objetos y tipos primitivos:

```
type ApiResponse = { status: "success"; data: object } | { status: "error"; message: string };

function handleApiResponse(response: ApiResponse): void {
   if (response.status === "success") {
      console.log("Data received:", response.data);
   } else {
      console.error("Error:", response.message);
```

```
    }
}

const successResponse: ApiResponse = { status: "success", data: { id: 1, name: "Sample" } };
const errorResponse: ApiResponse = { status: "error", message: "Failed to fetch data" };

handleApiResponse(successResponse); // Output: Data received: { id: 1, name: 'Sample' }
handleApiResponse(errorResponse);    // Output: Error: Failed to fetch data
```

En este caso, la función handleApiResponse maneja un tipo de unión que puede ser un objeto de éxito o un objeto de error. Según el valor de status, el tratamiento del dato cambia.

Ejemplo 4: Tipos de unión con valores predefinidos

Este ejemplo muestra cómo los tipos de unión pueden combinarse con valores predefinidos para asegurar que una variable solo puede tener ciertos valores.

```
type Status = "active" | "inactive" | "pending";

function setStatus(status: Status): void {
    console.log(`User status set to: ${status}`);
}

setStatus("active");   // Correcto
setStatus("inactive"); // Correcto
// setStatus("disabled"); // Error: El valor "disabled" no es asignable a "Status"
```

Aquí, la variable status solo puede tomar los valores "active", "inactive" o "pending", garantizando una mayor seguridad tipada en la función.

Ejemplo 5: Tipos de unión con parámetros opcionales

Puedes usar tipos de unión con parámetros opcionales para permitir que una función acepte diferentes combinaciones de argumentos:

```
function sendMessage(user: string, message: string, priority?: "low" | "high"): void {
    const priorityMessage = priority ? ` (Priority: ${priority})` : "";
    console.log(`Message to ${user}: ${message}${priorityMessage}`);
}
```

```
sendMessage("Alice", "Hello!"); // Output: Message to Alice: Hello!
sendMessage("Bob", "Important update", "high"); // Output: Message to Bob: Important update (Priority: high)
```

En este ejemplo, priority es un parámetro opcional que puede tener el valor "low" o "high". La función ajusta el comportamiento en función de si se proporciona o no este argumento.

Intersección de tipos

La intersección de tipos es una característica que crea un nuevo tipo que incluye todas las propiedades de los tipos combinados. Está simbolizada por un signo de ampersand (&) y es particularmente útil para componer tipos complejos a partir de otros más simples. En el siguiente código, vamos a crear un tipo para una entidad de chat de base de datos que también debe tener un valor id:

```
type IDBEntityWithId = {
   id: number;
};
type IChatEntity = {
   name: string;
};
type IChatEntityWithId = IDBEntityWithId & IChatEntity;
const chatEntity: IChatEntityWithId = {
   id: 1,
   name: "Typescript tutor",
};
```

Aquí, chatEntity se declara con el tipo IChatEntityWithId, por lo que debe incluir tanto id de IDBEntityWithId como name de IChatEntity. Esto ilustra cómo los tipos de intersección imponen la presencia de todas las propiedades de los tipos combinados.

Más ejemplos:

Ejemplo 1: Combinación de dos tipos básicos

En este ejemplo, combinamos un tipo de usuario y un tipo de dirección, de manera que el objeto resultante debe tener todas las propiedades de ambos:

```
type IUser = {
   username: string;
   email: string;
};

type IAddress = {
   city: string;
   country: string;
};

type IUserWithAddress = IUser & IAddress;

const userWithAddress: IUserWithAddress = {
   username: "johndoe",
   email: "johndoe@example.com",
   city: "Lima",
   country: "Peru",
};

console.log(userWithAddress);
```

Aquí, el tipo IUserWithAddress incluye las propiedades tanto de IUser como de IAddress. El objeto resultante debe contener username, email, city y country.

Ejemplo 2: Intersección de tipos con funciones

Puedes usar la intersección de tipos para definir un objeto que también funcione como una función:

```
type Logger = {
   log: (message: string) => void;
};

type DateLogger = {
   logDate: () => void;
};

type FullLogger = Logger & DateLogger;

const logger: FullLogger = {
   log: (message: string) => {
```

```
    console.log(`Log message: ${message}`);
  },
  logDate: () => {
    console.log(`Log date: ${new Date().toISOString()}`);
  },
};

logger.log("This is a log message");
logger.logDate();
```

En este caso, FullLogger combina la funcionalidad de dos tipos: Logger, que tiene una función log, y DateLogger, que tiene una función logDate. Ambos métodos deben estar presentes en cualquier objeto que implemente FullLogger.

Ejemplo 3: Intersección de tipos con propiedades opcionales

Este ejemplo muestra cómo las propiedades opcionales funcionan cuando se usan con intersecciones de tipos:

```
type IPerson = {
  name: string;
  age?: number;
};

type IEmployee = {
  employeeId: number;
};

type IEmployeeWithPerson = IPerson & IEmployee;

const employee: IEmployeeWithPerson = {
  name: "Alice",
  employeeId: 12345,
  age: 30, // esta propiedad es opcional
};

console.log(employee);
```

Aquí, IEmployeeWithPerson es una intersección de IPerson (donde age es opcional) e IEmployee. El objeto employee tiene que incluir las propiedades name y employeeId, pero age sigue siendo opcional.

Ejemplo 4: Intersección con múltiples tipos complejos

En este ejemplo, combinamos varios tipos complejos, donde uno de ellos tiene propiedades opcionales y el otro tiene propiedades obligatorias:

```
type IUser = {
   username: string;
   email?: string;
};

type IAdmin = {
   role: "admin";
   permissions: string[];
};

type IUserAdmin = IUser & IAdmin;

const adminUser: IUserAdmin = {
   username: "adminUser1",
   role: "admin",
   permissions: ["read", "write", "delete"]
};

console.log(adminUser);
```

Aquí, IUserAdmin es la combinación de un usuario (IUser) y un administrador (IAdmin). El resultado es un tipo que incluye username, opcionalmente email, y además las propiedades role y permissions de un administrador.

Ejemplo 5: Intersección de tipos y discriminación de uniones

Puedes usar intersecciones para discriminar tipos en una función que procesa tipos complejos:

```
type ITextMessage = {
   type: "text";
   content: string;
};

type IImageMessage = {
   type: "image";
   url: string;
```

```
};

type IMessage = ITextMessage & IImageMessage;

function processMessage(message: IMessage) {
  if (message.type === "text") {
    console.log("Text message:", message.content);
  } else {
    console.log("Image message:", message.url);
  }
}

const message: IMessage = {
  type: "text",
  content: "Hello world!",
  url: "https://example.com/image.jpg", // Como es intersección, debe incluir ambas propiedades
};

processMessage(message);
```

En este caso, IMessage es una combinación de un mensaje de texto y un mensaje de imagen. Aunque esto puede parecer contradictorio, dado que incluye tanto content como url, muestra cómo los tipos de intersección pueden combinar varias formas de un tipo.

Interfaz y características de OOP

Tanto la programación orientada a objetos (OOP) como las interfaces cumplen con algunos propósitos muy importantes: ayudan a definir claramente la estructura de los objetos que se pasan, reutilizan el código y permiten escribir funcionalidades extensibles. Hemos visto las interfaces antes, pero ahora, hablemos de cómo podemos extender sus definiciones. Interfaces

Extender las interfaces permite la creación de nuevas interfaces que heredan propiedades de las existentes, mejorando así la reutilización y organización. Vamos a tomar un ejemplo que usamos antes, pero ahora utilizando interfaces extendidas:

```
interface IMessageWithType extends IMessage {
  type: MessageType;
}
```

```
const userMessage: IMessageWithType = {
  id: 10,
  chatId: 2,
  userId: 1,
  content: "Hello, world!",
  createdAt: new Date(),
  type: "user",
};
```

La interfaz IMessageWithType extiende la interfaz IMessage, lo que significa que incluye todas las propiedades de IMessage más cualquier propiedad adicional definida en IMessageWithType. Aquí, IMessageWithType agrega la propiedad type, del tipo MessageType, a la estructura existente. Al crear un objeto userMessage del tipo IMessageWithType, se requiere incluir todas las propiedades tanto de IMessage como de IMessageWithType.

Ahora, veamos las funcionalidades de OOP que existen en TypeScript. Funcionalidades de OOP OOP es un paradigma de programación que utiliza objetos y clases para crear modelos basados en el mundo real. TypeScript adopta los principios básicos de OOP, permitiendo a los desarrolladores utilizar polimorfismo, abstracción, herencia y encapsulación utilizando sintaxis nativa. Veamos estas funcionalidades en detalle aquí:

Polimorfismo: Esto permite que los objetos de las subclases se traten como objetos de una superclase común. Se trata de crear una estructura donde una función puede utilizar las subclases de la superclase de manera intercambiable. El polimorfismo se logra principalmente a través de interfaces y clases abstractas. Al definir una interfaz común o una clase abstracta, TypeScript permite que diferentes clases implementen la misma estructura o métodos, lo que permite que las funciones trabajen con objetos de estas diferentes clases como si estuvieran trabajando con la clase base.

Abstracción: TypeScript utiliza clases abstractas e interfaces para implementar la abstracción. Estas construcciones permiten definir una plantilla estándar o contrato que otras clases pueden implementar, encapsulando la lógica compleja y exponiendo solo las partes necesarias.

Herencia: Este es un mecanismo en el que una nueva clase extiende (hereda de) una clase existente, lo que permite la reutilización de código y la creación de una relación jerárquica entre las clases. En TypeScript, la herencia se implementa usando la palabra clave extends. Una clase puede extender otra clase, heredando sus propiedades y métodos.

Encapsulación: Esto implica agrupar datos y métodos que operan en los datos dentro de una unidad, a menudo una clase, y restringir el acceso a algunos de los componentes del objeto, lo que garantiza la integridad de los datos. La encapsulación en TypeScript se logra a través de modificadores de acceso como public, private y protected. Estos modificadores controlan la visibilidad y accesibilidad de los

miembros de la clase, asegurando que los detalles internos de una clase estén ocultos y solo se expongan a través de una interfaz definida. Aquí hay una descripción de los modificadores de acceso:

- public: Este es el nivel de acceso predeterminado para los miembros de la clase. Los miembros declarados como public se pueden acceder desde cualquier lugar, lo que significa que no hay restricción de acceso. Esto incluye el acceso desde dentro de la clase, desde instancias de la clase y desde subclases.
- private: Los miembros declarados como private solo se pueden acceder desde dentro de la clase misma. No son accesibles desde instancias de la clase o desde subclases. Este nivel de acceso se utiliza para ocultar el estado interno y la funcionalidad de la clase desde el exterior, reforzando la encapsulación.
- protected: Los miembros declarados como protected se pueden acceder desde dentro de la clase y también desde las subclases. Sin embargo, no son accesibles desde instancias de la clase (a menos que sea a través de métodos definidos dentro de la clase o subclase). Esto permite una forma más controlada de accesibilidad, útil para casos en los que la subclase necesita un conocimiento más íntimo de la superclase sin exponer miembros al público en general.

Veamos el siguiente ejemplo que combina estas técnicas. Definiremos una clase abstracta AbstractDatabaseResource con métodos comunes y un método abstracto. Una clase concreta InMemoryChatResource extenderá esta clase abstracta y proporcionará implementaciones específicas para almacenar el chat en la memoria:

```
abstract class AbstractDatabaseResource {
  constructor(protected resourceName: string) {
  }
  protected logResource(resource: { id: number }): void {
    console.log(`[${this.resourceName}] Resource logged:`, resource);
  }
  abstract get(id: number): { id: number } | null
  abstract getAll(): { id: number }[]
  abstract addResource(resource: { id: number }): void;
}

const inMemoryChatResource = new InMemoryChatResource();
const chat1: IChat = {
  id: 1,
  ownerId: 2,
  messages: []
};
inMemoryChatResource.addResource(chat1);
const retrievedChat1 = inMemoryChatResource.get(1);
```

Hablemos de las diversas técnicas que hemos utilizado aquí:

- **Abstracción**: La clase AbstractDatabaseResource proporciona métodos externos que se utilizan para gestionar elementos de la base de datos (métodos como get, getAll y addResource), mientras oculta la complejidad de la implementación específica. Los usuarios de la clase AbstractDatabaseResource solo necesitan preocuparse por la interfaz: qué métodos están disponibles y qué parámetros aceptan, no cómo se implementan estos métodos. Esta separación de preocupaciones (SoC) hace que el sistema sea más fácil de entender y usar.
- **Herencia**: InMemoryChatResource es una clase concreta que extiende AbstractDatabaseResource. Esto significa que hereda sus propiedades y métodos, pero también proporciona implementaciones específicas para los métodos abstractos definidos en la clase base.
- **Encapsulación**: En InMemoryChatResource, el array resources se marca como private, lo que significa que no se puede acceder directamente desde fuera de la clase. Esta encapsulación garantiza que la representación interna de los recursos de chat esté oculta para el uso externo. El método logResource en la clase abstracta está marcado como protected, lo que permite que se acceda a él dentro de la clase y sus subclases, pero no fuera.
- **Polimorfismo**: Aunque InMemoryChatResource tiene diferentes implementaciones de los métodos (como addResource y get), se pueden usar de manera intercambiable en contextos donde se espera una clase AbstractDatabaseResource. Esto es polimorfismo, donde los objetos de diferentes clases se pueden tratar como objetos de una superclase común.

Instanciación y uso: Creamos una instancia de InMemoryChatResource y la usamos para agregar y recuperar datos de chat. A pesar de la implementación subyacente específica (array en memoria), el código solo se basa en el uso de la definición de la estructura de la clase abstracta compartida para conocer los métodos y propiedades que puede recuperar.

Es importante utilizar las interfaces y las clases de manera adecuada. Como regla general, utiliza las interfaces para definir contratos y formas de datos, asegurando la coherencia entre las implementaciones y facilitando la refactorización. Utiliza las clases para encapsular datos y comportamiento, aprovechando la herencia y el polimorfismo para promover la reutilización y el mantenimiento del código, manteniendo las definiciones de las clases enfocadas y evitando jerarquías de herencia demasiado complejas. Las interfaces son buenas para definir la estructura abstracta con la que vas a operar en un parámetro de función, y las clases son excelentes para encapsular la complejidad y proporcionar solo una forma simple y directa de interactuar con la lógica de las clases.

Ejemplo 1: Uso de interfaces con clases

TypeScript permite que las clases implementen interfaces, lo que garantiza que las clases que implementan una interfaz contengan ciertas propiedades y métodos.

```
interface IAnimal {
  name: string;
  makeSound(): void;
}

class Dog implements IAnimal {
  constructor(public name: string) {}

  makeSound(): void {
    console.log(`${this.name} barks!`);
  }
}

class Cat implements IAnimal {
  constructor(public name: string) {}

  makeSound(): void {
    console.log(`${this.name} meows!`);
  }
}

const animals: IAnimal[] = [new Dog("Buddy"), new Cat("Whiskers")];
animals.forEach(animal => animal.makeSound());
```

Características OOP utilizadas:

- **Polimorfismo:** Ambos, Dog y Cat, implementan la misma interfaz IAnimal. Esto permite tratar los objetos de ambas clases de la misma forma, iterando sobre una lista de animales y llamando al método makeSound() en cada uno.
- **Encapsulación:** Cada clase tiene su propio comportamiento específico encapsulado en su método makeSound.

Ejemplo 2: Herencia y modificadores de acceso

Aprovechamos la herencia para crear una jerarquía de clases y mostramos cómo funcionan los modificadores de acceso public, private y protected.

```
class Vehicle {
  protected speed: number = 0;
```

```
  accelerate(amount: number): void {
    this.speed += amount;
  }

  getSpeed(): number {
    return this.speed;
  }
}

class Car extends Vehicle {
  private model: string;

  constructor(model: string) {
    super();
    this.model = model;
  }

  public displayDetails(): void {
    console.log(`${this.model} is going at ${this.getSpeed()} km/h`);
  }
}

const myCar = new Car("Tesla");
myCar.accelerate(50);
myCar.displayDetails(); // Tesla is going at 50 km/h
```

Características OOP utilizadas:

- **Herencia:** Car extiende la clase Vehicle, heredando sus métodos y propiedades.
- **Encapsulación:** La propiedad speed es protected, lo que significa que puede ser accedida solo dentro de la clase Vehicle y sus subclases, pero no desde fuera de ellas. La propiedad model es private y solo es accesible dentro de la clase Car.

Ejemplo 3: Clases abstractas y métodos abstractos

Las clases abstractas permiten definir una estructura común para varias subclases, que deben implementar los métodos abstractos definidos.

```
abstract class Shape {
  constructor(public color: string) {}
```

```
    abstract calculateArea(): number;

    displayColor(): void {
        console.log(`The shape is ${this.color}`);
    }
}

class Circle extends Shape {
    constructor(color: string, private radius: number) {
        super(color);
    }

    calculateArea(): number {
        return Math.PI * Math.pow(this.radius, 2);
    }
}

class Rectangle extends Shape {
    constructor(color: string, private width: number, private height: number) {
        super(color);
    }

    calculateArea(): number {
        return this.width * this.height;
    }
}

const shapes: Shape[] = [new Circle("red", 5), new Rectangle("blue", 10, 20)];
shapes.forEach(shape => {
    shape.displayColor();
    console.log(`Area: ${shape.calculateArea()}`);
});
```

Características OOP utilizadas:

- **Abstracción:** Shape es una clase abstracta que define un método abstracto calculateArea(), que debe ser implementado por sus subclases.
- **Polimorfismo:** A pesar de que Circle y Rectangle son clases diferentes, se pueden tratar de la misma manera a través del tipo base Shape.

Ejercicio 1: Implementar una interfaz con métodos opcionales

Define una interfaz IUser que tenga propiedades name y email. Luego, crea una clase que implemente esta interfaz y agregue un método adicional getRole() que pueda devolver "admin" o "user".

Ejercicio 2: Abstracción y clases concretas

Crea una clase abstracta Employee con un método abstracto calculateSalary(). Luego, define clases concretas FullTimeEmployee y PartTimeEmployee que extiendan de Employee y calculen el salario basándose en diferentes parámetros como horas trabajadas o salario mensual.

Ejercicio 3: Herencia múltiple con interfaces

Define dos interfaces Flyable y Swimmable, cada una con un método abstracto fly() y swim(). Luego, crea una clase Bird que implemente ambas interfaces y defina los métodos respectivos.

Ejercicio 4: Aplicar encapsulación y polimorfismo

Crea una clase BankAccount que tenga las propiedades accountNumber, balance, y métodos como deposit() y withdraw(). Luego, extiende esta clase para crear SavingsAccount y CheckingAccount, y usa polimorfismo para manejar estas cuentas de manera uniforme.