Universidad de Buenos Aires
Facultad de Ingeniería

Trabajo Práctico

TypeScript

[75.31]
Teoría de Lenguaje

Grupo Auchi:
- **Ciriani**, Chiara (107644)
- **Guglielmi**, Nicolás (107687)
- **Magnani**, Elian (107413)

1er Cuatrimestre de 2023

Índice

1. Introducción

En el siguiente informe se presenta un trabajo de investigación del lenguaje de programación **TypeScript**. Se abarcarán todos los puntos más característicos del lenguaje analizado, explicando la sintaxis y sus características tanto básicas como avanzadas. Luego de esta investigación, quedará claro bajo qué situaciones es conveniente o no el uso del lenguaje analizado.

Además, posteriormente al trabajo de investigación, se desarrollará un ejemplo integrador guiado.

Se ha tomado la decisión de elegir TypeScript como objeto de investigación a partir del análisis de la industria y las tendencias del mercado de los tiempos que corren. Uno de los recursos que aportó a la toma de esta decisión fue la "Stack Overflow Developer Survey", una encuesta anual realizada por el sitio web de preguntas y respuestas para programadores, Stack Overflow. La misma busca recopilar cierta información del campo informático como lo son habilidades, conocimientos, herramientas, tecnologías, salarios, satisfacción laboral y otros aspectos relacionados a los programadores.

TypeScript se posicionó en el 2022 como el cuarto lenguaje de programación más amado en la industria y como el tercero con más porcentaje de desarrolladores que no lo utilizan pero que han expresado interés en aprenderlo. Por lo tanto, esto nos motivó a elegirlo como lenguaje de investigación ya que ninguno había programado en este lenguaje y estos resultados nos dieron ganas de aprenderlo. Nos gustaría investigar si realmente es un lenguaje de programación que provee tantos beneficios como para ser el cuarto lenguaje de programación más amado en la industria.

1.1 ¿Qué es TypeScript?

Una vez elegido el lenguaje de programación a investigar, nos surge la duda: ¿qué es TypeScript?

Sabemos que es un lenguaje de programación pero, ¿qué lo diferencia del resto? ¿por qué elegiríamos este lenguaje por encima de otros?

TypeScript es un lenguaje de programación de código abierto desarrollado por Microsoft que se basa en JavaScript. Es un superset de JavaScript, lo que significa que todo el código JavaScript válido también es código TypeScript válido. Sin embargo, TypeScript agrega características adicionales a JavaScript, como el tipado estático opcional y la adición de características orientadas a objetos.

La principal característica distintiva de TypeScript es su capacidad para agregar **tipado estático opcional** al código JavaScript. Esto significa que en lugar de inferir los tipos en tiempo de ejecución, se pueden agregar anotaciones de tipo al código TypeScript para ayudar al compilador a detectar posibles errores de tipos durante la fase de compilación. Estas anotaciones de tipo proporcionan una mayor seguridad y confiabilidad en el código, permitiendo a los desarrolladores atrapar errores antes de que ocurran en tiempo de ejecución.

TypeScript también ofrece otras características como **clases, interfaces, herencia, genéricos, decoradores y módulos**, que amplían la capacidad de JavaScript para desarrollar aplicaciones más escalables y estructuradas. Además, TypeScript proporciona una amplia compatibilidad con las últimas características de ECMAScript (el estándar en el que se basa JavaScript), lo que permite a los desarrolladores utilizar características de JavaScript moderno incluso si no son completamente compatibles en todos los navegadores.

Una vez que el código TypeScript se escribe y se anotan los tipos, debe ser compilado a JavaScript para que pueda ser ejecutado en un entorno de navegador o en un servidor Node.js. El compilador de TypeScript toma el código fuente TypeScript y lo transpila a código JavaScript equivalente, manteniendo la estructura y las anotaciones de tipo definidas en TypeScript.

TypeScript se ha vuelto muy popular y es ampliamente utilizado en proyectos de desarrollo web y aplicaciones empresariales debido a su capacidad para mejorar la seguridad y la productividad en el desarrollo de aplicaciones JavaScript. Además, cuenta con una gran comunidad de desarrolladores que contribuyen a su desarrollo y proporcionan bibliotecas y herramientas adicionales para facilitar el desarrollo con TypeScript. Por lo tanto, así como todas estas razones motivan a distintos desarrolladores a utilizar este lenguaje en sus proyectos, también nos motivaron a nosotros a desarrollar nuestra propia investigación y aplicación.

2. Origen

El lenguaje de programación TypeScript tuvo su aparición de manera pública por primera vez en su versión 0.8 en octubre del 2012, después de dos años de desarrollo por parte de Microsoft.

Está claro que, como el nombre indica, Microsoft se tomó muy en serio la idea de llevar el tipado estático a JavaScript. Pero, es interesante mencionar cómo surgió la idea de este nuevo lenguaje y cuál fue la motivación de los creadores. Resulta que, según

Steve Lucco, el arquitecto jefe para el tiempo de ejecución y las herramientas de Javascript, una de las grandes motivaciones para el desarrollo de Typescript fue la experiencia de otros equipos de Microsoft que intentaban desarrollar y mantener productos de Microsoft en JavaScript. En su discurso en una entrevista que realizó citó al equipo de Bing Maps como uno de los que ha tenido dificultades para escalar JavaScript. También, mencionó que el equipo de TypeScript ha trabajado con el equipo de Office 365 (es probable que esté relacionado con el desarrollo de Office Web Apps).

Sin embargo, la historia del origen de este lenguaje es un poco más amplia y entraremos en algunos detalles para entender mejor su actual funcionamiento.

El nacimiento de TypeScript

2010-2012: El comienzo (2010-2012)

TypeScript fue anunciado por primera vez por Microsoft en octubre del 2012, con Anders Hejlsberg, el creador de C# y Turbo Pascal, a la cabeza del proyecto. La motivación inicial detrás de TypeScript fue abordar las deficiencias de JavaScript, particularmente en aplicaciones a gran escala. JavaScript, al ser un lenguaje de escritura dinámica, era propenso a errores de tiempo de ejecución debido a la falta de verificación de tipos. TypeScript tenía como objetivo proporcionar funciones opcionales de escritura estática y programación orientada a objetos, lo que facilita a los desarrolladores la creación y el mantenimiento de aplicaciones complejas.

Un ejemplo simple de las anotaciones de tipo de TypeScript:

```
function greet(name: string): string {
 return "Hello, " + name;
}
```

Como TypeScript es un superconjunto estricto de JavaScript, conserva la compatibilidad con el código JavaScript existente. El compilador de TypeScript (tsc) transpila el código de TypeScript a JavaScript, lo que le permite ejecutarse en cualquier entorno que admita JavaScript. Esto aseguró que los desarrolladores pudieran adoptar gradualmente TypeScript en sus proyectos sin tener que volver a escribir su base de código completa.

Adopción temprana y respuesta de la comunidad

2012–2015: TypeScript gana tracción

Los primeros años de TypeScript vieron un crecimiento constante en la adopción entre la comunidad de desarrolladores. El lanzamiento de Angular 2 en 2014 marcó un hito importante para TypeScript, ya que el popular marco de interfaz de usuario eligió

TypeScript como su lenguaje predeterminado. Esta decisión ayudó a TypeScript a ganar credibilidad y provocó un mayor interés de otros proyectos y organizaciones.

Durante este período, TypeScript experimentó varios lanzamientos importantes, que introdujeron nuevas funciones y mejoras en el lenguaje. La respuesta de la comunidad fue en gran medida positiva, ya que los desarrolladores reconocieron el valor que aportaba TypeScript en términos de seguridad de tipos, mejores herramientas y mejor capacidad de mantenimiento del código.

Integración con las principales bibliotecas y Frameworks
2015–2018: TypeScript se generaliza

A medida que crecía la popularidad de TypeScript, las principales bibliotecas y marcos comenzaron a ofrecer soporte de primera clase para el lenguaje. React, Vue y otros proyectos populares agregaron declaraciones de TypeScript a sus paquetes, lo que permitió a los desarrolladores beneficiarse de las funciones de autocompletado y verificación de tipos de TypeScript. Esto, a su vez, alentó a más desarrolladores a adoptar TypeScript en sus proyectos, lo que generó un ciclo de retroalimentación positiva que impulsó aún más el crecimiento de TypeScript.

Durante este período, TypeScript también experimentó mejoras significativas en su sistema de tipos, con la introducción de funciones como tipos de unión, tipos de intersección y tipos asignados. Estas mejoras permitieron a los desarrolladores expresar relaciones de tipo complejas, lo que hizo que TypeScript fuera aún más potente y flexible.

Ejemplo de tipos de unión e intersección en TypeScript:

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

type User = {
 role: "user";
 username: string;
};

type SuperUser = Admin & User;
```

TypeScript hoy: un elemento básico para el desarrollo moderno
2018-Presente: La maduración de TypeScript

En los últimos años, TypeScript se ha convertido en un elemento básico para el desarrollo web moderno. El lenguaje ha tenido una adopción generalizada entre desarrolladores y organizaciones por igual, gracias a su combinación de seguridad de tipos, herramientas mejoradas y compatibilidad con el ecosistema de JavaScript.

El equipo de TypeScript en Microsoft continúa iterando en el lenguaje, con lanzamientos regulares que introducen nuevas características y mejoras. Paralelamente, la comunidad de JavaScript más amplia ha adoptado TypeScript, con proyectos populares como Next.js, NestJS y GraphQL que brindan compatibilidad con TypeScript de primera clase desde el primer momento.

3. Usos

Para entender los usos del lenguaje y cuándo es conveniente utilizarlo y cuándo no, es importante primero ampliar en algunas de las ventajas que el mismo ofrece

Como mencionamos previamente, TypeScript es un lenguaje de programación de código abierto desarrollado por Microsoft que amplía JavaScript con características de tipado estático. Su principal objetivo es brindar a los desarrolladores una mayor seguridad y productividad en el desarrollo de aplicaciones JavaScript a gran escala.

El uso de TypeScript ofrece varias ventajas:

1. *Tipado estático:* TypeScript permite asignar tipos a las variables, parámetros y valores de retorno de las funciones, lo que ayuda a detectar errores en tiempo de compilación y mejora la calidad del código. Esto facilita la detección temprana de errores y proporciona un mejor soporte para herramientas de desarrollo.
2. *Autocompletado y herramientas de desarrollo:* Los IDE y editores de código modernos ofrecen un soporte avanzado para TypeScript, lo que incluye autocompletado inteligente, refactoring, navegación de código y documentación mejorada. Estas herramientas mejoran la productividad y facilitan el mantenimiento del código.
3. *Escalabilidad y mantenibilidad:* TypeScript es especialmente útil en proyectos grandes y complejos, ya que permite establecer una estructura más sólida y modular. El tipado estático y las características orientadas a objetos ayudan a mantener el código organizado, legible y fácilmente mantenible.
4. *Integración con librerías y frameworks:* TypeScript es ampliamente utilizado en el ecosistema de desarrollo web, y muchos frameworks y bibliotecas populares, como Angular, React y Vue.js, ofrecen soporte nativo para TypeScript. Esto facilita la integración y proporciona una experiencia de desarrollo más coherente.

3.1 ¿Para qué sirve?

Debido a estas ventajas, podemos afirmar que es conveniente usar TypeScript en los siguientes casos:

1. *Proyectos a gran escala:* Cuando se está desarrollando una aplicación grande y compleja, TypeScript puede ayudar a manejar mejor la complejidad del código. Como fue mencionado en las ventajas del lenguaje, el tipado estático permite detectar y corregir errores en tiempo de compilación, lo que reduce la posibilidad de errores costosos en tiempo de ejecución.
2. *Equipos de desarrollo grandes:* En entornos con múltiples desarrolladores trabajando en el mismo proyecto, TypeScript proporciona un conjunto de reglas y convenciones más estricto que ayuda a mantener un código limpio y consistente. El tipado estático también facilita la colaboración al proporcionar información clara sobre las interfaces y estructuras de datos utilizadas en el proyecto.
3. *Mantenibilidad y escalabilidad:* TypeScript es útil cuando se tiene la intención de mantener y escalar el código a largo plazo. El tipado estático y las características de programación orientada a objetos de TypeScript ayudan a mantener una base de código más estructurada y modular, facilitando la adición de nuevas características y la realización de cambios sin romper el funcionamiento existente.
4. *Integración con frameworks y bibliotecas:* Si un proyecto planea utilizar un framework o una biblioteca que tiene soporte nativo para TypeScript, como Angular, React o Vue.js, utilizar TypeScript permitirá aprovechar al máximo las características y herramientas específicas de ese ecosistema. Esto puede mejorar la productividad y la calidad del código.
5. *Mejora de la productividad en el desarrollo:* TypeScript ofrece una serie de características que mejoran la experiencia de desarrollo, como autocompletado inteligente, refactorización, navegación de código y documentación mejorada. Estas herramientas pueden ayudar a escribir y mantener el código más rápidamente, lo que se traduce en una mayor productividad.

En general, TypeScript es una buena elección cuando se valora **la seguridad, la escalabilidad, la mantenibilidad y la productividad en el desarrollo de aplicaciones grandes y complejas, especialmente en entornos de trabajo colaborativos**.

Ahora que sabemos cuándo TypeScript es buena elección para utilizar en el desarrollo de un proyecto, podríamos ampliar más en qué tipo de desarrollos se usa debido a estas ventajas presentadas.

TypeScript se utiliza principalmente para el **desarrollo de aplicaciones web y de backend**, aunque también se puede utilizar en otros contextos. Algunos casos de uso comunes son los siguientes:

1. *Desarrollo web:* TypeScript es ampliamente utilizado en el desarrollo de aplicaciones web, tanto del lado del cliente como del lado del servidor. En el lado del cliente, se utiliza con frecuencia junto con frameworks como Angular, React y Vue.js para crear interfaces de usuario interactivas y dinámicas. En el lado del servidor, se puede utilizar con Node.js y Express.js para desarrollar API y servicios web.
2. *Aplicaciones de una sola página (Single-Page Applications, SPAs):* Las SPAs son aplicaciones web que se cargan una vez y luego interactúan con el servidor mediante solicitudes AJAX. TypeScript se adapta bien a este tipo de aplicaciones, ya que proporciona un desarrollo estructurado y sólido para el manejo de la lógica del cliente.
3. *Desarrollo de aplicaciones de backend:* TypeScript también puede utilizarse en el desarrollo de aplicaciones de backend utilizando Node.js. Al igual que en el desarrollo del lado del cliente, el tipado estático y las características adicionales de TypeScript ayudan a construir aplicaciones backend más sólidas y fáciles de mantener.
4. *Proyectos de código abierto y bibliotecas:* Muchos proyectos de código abierto y bibliotecas populares utilizan TypeScript para proporcionar una mejor experiencia de desarrollo y una API más segura. Al utilizar TypeScript, los desarrolladores pueden aprovechar las ventajas del tipado estático y obtener una mejor documentación y autocompletado en sus proyectos.
5. *Desarrollo de herramientas y utilidades:* TypeScript también se puede utilizar para crear herramientas y utilidades de desarrollo. Al aprovechar las características del lenguaje, los desarrolladores pueden crear herramientas personalizadas, generadores de código, análisis estático y otras utilidades para mejorar la eficiencia y calidad del desarrollo.

En resumen, TypeScript se utiliza comúnmente en el **desarrollo web**, tanto en el lado del cliente como del servidor, así como en proyectos de código abierto, bibliotecas y desarrollo de herramientas. Ofrece ventajas en términos de seguridad, mantenibilidad, escalabilidad y productividad, lo que lo hace atractivo para una amplia gama de casos de uso en el desarrollo de software.

3.2 ¿Para qué no sirve?

Sin embargo, aunque TypeScript es útil en muchos escenarios de desarrollo, hay algunas situaciones en las que puede no ser tan conveniente:

1. *Proyectos pequeños y scripts simples:* Si se está desarrollando una aplicación pequeña o un script simple, los beneficios del tipado estático y las herramientas avanzadas pueden no ser necesarios. En estos casos, JavaScript puro puede ser más ligero y suficiente para cumplir con los requisitos del proyecto.
2. *Proyectos que requieren una entrega rápida de prototipos:* Si se está en una etapa temprana de desarrollo y se necesita crear prototipos rápidamente para validar ideas o conceptos, TypeScript puede agregar una capa adicional de complejidad y tiempo al proceso de desarrollo. En estos casos, es posible que se prefiera utilizar JavaScript para iterar rápidamente.
3. *Equipo de desarrollo con poca experiencia en TypeScript:* Si el equipo de desarrollo no está familiarizado o carece de experiencia en TypeScript, la curva de aprendizaje inicial puede ser un obstáculo. Esto puede resultar en una mayor cantidad de tiempo dedicado a aprender y comprender las características del lenguaje, lo que puede retrasar el desarrollo y la productividad del equipo.
4. *Librerías y frameworks sin soporte para TypeScript:* Aunque TypeScript es compatible con muchas bibliotecas y frameworks populares, algunas pueden no tener soporte nativo para TypeScript o pueden tener una documentación limitada. En estos casos, trabajar con JavaScript puede ser más conveniente, ya que no hay necesidad de realizar adaptaciones o conversiones para integrar las bibliotecas o frameworks en tu proyecto.
5. *Aplicaciones de rendimiento crítico:* En situaciones donde se busca la máxima eficiencia y rendimiento, el tiempo adicional necesario para la transpilación y la comprobación de tipos en tiempo de ejecución de TypeScript puede no ser deseable. En estos casos, es posible que se prefiera utilizar un lenguaje de programación de bajo nivel o con un enfoque más ligero.

Básicamente, aunque TypeScript es un lenguaje versátil y poderoso, puede no ser conveniente en proyectos pequeños o simples, durante las etapas de prototipado rápido, cuando el equipo de desarrollo no está familiarizado con el lenguaje, en entornos sin soporte para TypeScript o en aplicaciones de rendimiento crítico.

Si bien recién se mencionaron situaciones en las que puede no ser tan conveniente utilizar TypeScript en comparación de Javascript, también podemos mencionar situaciones en las que otros lenguajes de programación pueden resolver mejor ciertos problemas en comparación con TypeScript. Algunos ejemplos claros donde se indica que no es conveniente usar TypeScript son:

1. *Desarrollo de aplicaciones nativas para dispositivos móviles:* Si se está desarrollando una aplicación móvil nativa para iOS o Android, es posible que lenguajes como Swift (para iOS) o Kotlin (para Android) sean más adecuados. Estos lenguajes están optimizados para las plataformas móviles respectivas y

brindan acceso directo a las API y funcionalidades específicas del sistema operativo.

2. *Programación de sistemas de tiempo real:* En escenarios en los que se requiere un control preciso del tiempo y la latencia, lenguajes de programación de bajo nivel como C o C++ pueden ser más apropiados. Estos lenguajes ofrecen un control directo sobre los recursos del sistema y proporcionan mayor eficiencia y predictibilidad en el rendimiento.
3. *Aplicaciones científicas y de alto rendimiento:* Para aplicaciones que involucran cálculos científicos complejos o procesamiento intensivo, lenguajes como Python con bibliotecas especializadas (por ejemplo, NumPy o TensorFlow) o lenguajes como Julia pueden ser más adecuados. Estos lenguajes están optimizados para operaciones matemáticas y computacionales de alto rendimiento.
4. *Desarrollo de aplicaciones de tiempo de ejecución específico:* Si se está desarrollando aplicaciones que se ejecutan directamente en entornos específicos, como aplicaciones de script para Adobe Photoshop o Autodesk Maya, es posible que se tenga que utilizar lenguajes de scripting específicos proporcionados por esas aplicaciones.
5. *Requisitos de integración con sistemas existentes:* En algunos casos, si se necesita integrarse con sistemas heredados o infraestructuras específicas, puede ser más beneficioso utilizar lenguajes de programación que tienen una amplia compatibilidad o herramientas de interoperabilidad existentes para ese propósito específico.

De todas formas, en última instancia, la elección del lenguaje de programación depende del contexto, los requisitos del proyecto y las preferencias del equipo de desarrollo. Si bien TypeScript es adecuado para muchos casos de uso, existen situaciones donde otros lenguajes pueden brindar soluciones más óptimas para problemas específicos.

En resumen, TypeScript es una opción sólida para **proyectos grandes y complejos en los que la seguridad, la escalabilidad y el mantenimiento son aspectos importantes**. También es recomendable cuando se trabaja con **frameworks o bibliotecas** que ofrecen soporte nativo para TypeScript. Para proyectos más pequeños y simples, o para aquellos que requieren una curva de aprendizaje más suave, JavaScript puede ser suficiente.

En comparación a otros lenguajes de programación, se motiva su uso en el desarrollo web y frontend, proyectos a gran escala, equipos de desarrollo grandes, integración con frameworks y bibliotecas populares, y cuando se busca mejorar la productividad en el desarrollo. Sus características de tipado estático y herramientas avanzadas proporcionan beneficios significativos en términos de seguridad, mantenibilidad y escalabilidad del código.

4. Características básicas del lenguaje

Antes de entrar en detalle sobre algunas características específicas de TypeScript, mencionaremos algunas características básicamente del lenguaje en general como para tener un mayor contexto.

Algunas características avanzadas del lenguaje TypeScript:

1. *Tipado estático opcional:* TypeScript permite agregar anotaciones de tipo estáticas a las variables, parámetros de función, valores de retorno y más. Esto ayuda a detectar errores de tipos durante la fase de compilación y mejora la seguridad y la calidad del código.
2. *Inferencia de tipos:* TypeScript puede inferir automáticamente el tipo de una variable en función de su valor inicial. Esto evita la necesidad de anotar explícitamente el tipo en todos los casos y reduce la cantidad de código necesario.
3. *Soporte para ECMAScript:* TypeScript ofrece un amplio soporte para las características más recientes de ECMAScript, como las arrow functions, destructuring, spread operators y async/await, lo que permite utilizar las últimas funcionalidades de JavaScript incluso si no son completamente compatibles en todos los navegadores.
4. *Orientación a objetos:* TypeScript admite la programación orientada a objetos y proporciona características como clases, herencia, interfaces y modificadores de acceso. Esto facilita la organización y el mantenimiento del código, especialmente en proyectos más grandes.
5. *Modificadores de acceso:* TypeScript incluye modificadores de acceso como 'public', 'private' y 'protected', que permiten controlar la visibilidad y accesibilidad de las propiedades y métodos de una clase.
6. *Módulos y namespaces:* TypeScript admite la modularidad a través de módulos y namespaces, lo que facilita la organización y la reutilización de código en diferentes archivos y proyectos.
7. *Decoradores: L*os decoradores son una característica de TypeScript que permite agregar metadatos a clases, métodos y propiedades. Se utilizan comúnmente en frameworks como Angular para agregar funcionalidades adicionales a las clases.
8. *Funciones de orden superior:* TypeScript admite funciones de orden superior, lo que significa que se pueden pasar funciones como argumentos a otras funciones y devolver funciones como resultado de otras funciones.

Estas son solo algunas de las características básicas de TypeScript. El lenguaje también ofrece muchas más funcionalidades avanzadas y útiles que permiten un desarrollo más eficiente y seguro de aplicaciones web y empresariales.

Ahora, entraremos en detalle con algunas de estas características.

4.1 Paradigmas que soporta

En términos de paradigmas de programación, TypeScript es un l**enguaje multiparadigma** que admite *varios enfoques*.

1. *Paradigma Orientado a Objetos:* TypeScript es un lenguaje orientado a objetos y permite la programación orientada a objetos de manera similar a otros lenguajes como Java o C#. Permite la definición de clases, interfaces, herencia, polimorfismo y encapsulamiento. También incluye conceptos como clases abstractas, modificadores de acceso y métodos estáticos.
2. *Paradigma Funcional:* TypeScript también admite el paradigma funcional. Se puede escribir funciones puras, utilizar funciones de orden superior y aprovechar características como las expresiones lambda (funciones flecha). Además, TypeScript proporciona soporte para inferencia de tipos en las funciones, lo que facilita la escritura de código funcional de manera más segura y expresiva.
3. *Paradigma Declarativo:* Aunque TypeScript no es un lenguaje puramente declarativo, se puede escribir código de manera declarativa utilizando bibliotecas y marcos de trabajo que lo admiten. Por ejemplo, se pueden utilizar bibliotecas como React o Angular para desarrollar interfaces de usuario de manera declarativa, definiendo la estructura y el comportamiento de los componentes sin tener que preocuparnos demasiado por los detalles de implementación.

En resumen, TypeScript es principalmente un lenguaje orientado a objetos que proporciona características y sintaxis para la programación funcional. También, se pueden aprovechar enfoques declarativos al utilizar bibliotecas y marcos de trabajo compatibles. **La flexibilidad de TypeScript permite combinar múltiples paradigmas según las necesidades del proyecto**.

Comparación con Oz

Si comparamos con **Oz**, estos dos lenguajes difieren en los paradigmas de programación que soportan:

Por un lado, Oz:

- Oz es un lenguaje multiparadigma que combina características de *programación funcional, lógica y orientada a objetos*.

- Como *lenguaje funcional*, Oz permite la definición y manipulación de funciones como ciudadanos de primera clase, lo que facilita la programación basada en la composición de funciones.
- Como *lenguaje lógico*, Oz ofrece la capacidad de realizar razonamiento lógico y declarativo utilizando restricciones lógicas y programación lógica.
- Además, Oz también incluye elementos de *programación orientada a objetos*, permitiendo la definición de clases y objetos con estado y comportamiento.

Por otro lado, TypeScript:

- TypeScript se basa principalmente en el paradigma de *programación orientada a objetos*.
- TypeScript extiende el lenguaje JavaScript con características orientadas a objetos, como clases, herencia, interfaces y encapsulación.
- Además de la programación orientada a objetos, TypeScript también es compatible con elementos de *programación funcional*, como funciones de orden superior, lambdas y tipos de datos inmutables.
- TypeScript proporciona un sistema de tipos estático que ayuda a detectar errores en tiempo de compilación y mejora la capacidad de mantenimiento del código.

En resumen, Oz es un lenguaje multiparadigma que combina elementos de programación funcional, lógica y orientada a objetos, lo que permite una mayor flexibilidad en la forma de abordar los problemas de programación. Por otro lado, TypeScript se centra principalmente en el paradigma de programación orientada a objetos, con soporte adicional para conceptos de programación funcional.

4.2 Compilado / Interpretado

TypeScript es un lenguaje que se compila a JavaScript. Esto significa que, en primer lugar, es necesario compilar el código fuente de TypeScript a JavaScript antes de que pueda ser interpretado y ejecutado por un navegador web o un entorno de ejecución de JavaScript.

El proceso de compilación de TypeScript implica tomar el código fuente escrito en TypeScript, que incluye anotaciones de tipo y características adicionales, y traducirlo a código JavaScript equivalente. Durante este proceso, el compilador de TypeScript realiza varias tareas, como la verificación estática de tipos, la eliminación de las características específicas de TypeScript y la generación de código JavaScript compatible.

La compilación de TypeScript puede ser realizada mediante la línea de comandos utilizando el compilador de TypeScript (tsc) o mediante herramientas de construcción (build tools) como Webpack o Gulp, que pueden automatizar el proceso de compilación junto con otras tareas de construcción.

Una vez que el código fuente de TypeScript se ha compilado a JavaScript, el resultado es un archivo JavaScript estándar que puede ser interpretado y ejecutado por cualquier entorno de ejecución de JavaScript compatible, como un navegador web o un servidor Node.js. En este punto, el código JavaScript generado se comportará como cualquier otro código JavaScript, sin ninguna referencia directa a TypeScript.

La compilación a JavaScript es necesaria porque los navegadores y entornos de ejecución de JavaScript nativamente no entienden el código TypeScript. Al compilarlo a JavaScript, se asegura que el código TypeScript sea compatible con los entornos de ejecución existentes y se pueda ejecutar sin problemas.

Es importante tener en cuenta que TypeScript también admite la función de "compilación en tiempo real" o "compilación en el editor". Esto significa que algunos editores de código o IDEs tienen integraciones especiales con el compilador de TypeScript, lo que permite ver los errores de tipo y otros problemas de manera interactiva mientras se escribe el código TypeScript, sin tener que esperar a que se ejecute el proceso de compilación manualmente.

Comparación con Oz

Si comparamos con **Oz**, estos dos lenguajes difieren en cuanto a cómo se compilan o interpretan.

Por un lado, Oz:

- Oz es un lenguaje interpretado en su forma más común. Los programas escritos en Oz se ejecutan a través de un intérprete que procesa y ejecuta las instrucciones del código fuente directamente.
- El intérprete de Oz puede ser utilizado de manera interactiva, lo que permite ejecutar instrucciones y obtener resultados en tiempo real. También, es posible ejecutar programas completos almacenados en archivos de código fuente.

Por otro lado, TypeScript:

- TypeScript es un lenguaje compilado. Los programas escritos en TypeScript se compilan a JavaScript antes de su ejecución. El proceso de compilación de TypeScript implica la traducción del código fuente de TypeScript a código JavaScript equivalente.

- La compilación de TypeScript se realiza utilizando el compilador de TypeScript (tsc), que toma el código fuente escrito en TypeScript (con extensión .ts) y genera archivos JavaScript (con extensión .js) listos para ser ejecutados en un entorno compatible con JavaScript.

En resumen, Oz se interpreta directamente sin una etapa de compilación separada, mientras que TypeScript requiere una etapa de compilación previa que traduce el código TypeScript a JavaScript antes de su ejecución. Cabe destacar que, a pesar de que TypeScript se compila a JavaScript, sigue siendo interpretado por el entorno en el que se ejecuta (por ejemplo, el navegador o el entorno de ejecución de Node.js).

4.3 Tipado

Como bien indica su nombre, el tipado es una *característica fundamental* de TypeScript que lo distingue de JavaScript. TypeScript proporciona un **sistema de tipos estático opcional** que permite especificar y verificar los tipos de variables, parámetros, funciones y estructuras de datos en tiempo de compilación.

Algunos aspectos importantes sobre el tipado en TypeScript son los siguientes:

1. *Tipos estáticos opcionales:* TypeScript permite agregar anotaciones de tipo estáticas a las variables, parámetros de función, valores de retorno y otros elementos del código. Estas anotaciones de tipo son opcionales, lo que significa que se puede optar por especificar los tipos o dejar que TypeScript infiera automáticamente los tipos según el contexto.
2. *Tipos básicos:* TypeScript proporciona tipos básicos como *number*, *string*, *boolean*, *object*, *null*, *undefined*, *symbol*, etc. Estos tipos básicos ayudan a definir el tipo de datos de las variables y expresiones.
3. *Tipos personalizados:* Además de los tipos básicos, TypeScript permite definir tipos personalizados, como interfaces, clases, enumeraciones y alias de tipo.
    - Las interfaces se utilizan para describir la forma de los objetos.
    - Las clases permiten definir objetos con propiedades y métodos.
    - Las enumeraciones representan un conjunto de valores posibles.
    - Los alias de tipo permiten definir nombres alternativos para tipos existentes.
4. *Inferencia de tipos:* TypeScript tiene un sistema de inferencia de tipos sólido que puede deducir automáticamente los tipos según el contexto en el que se utiliza una expresión. Esto significa que no siempre es necesario especificar explícitamente los tipos, ya que TypeScript puede inferirlos basándose en el código circundante.

5. *Verificación estática de tipos:* Durante el proceso de compilación, el compilador de TypeScript realiza una verificación estática de tipos para detectar posibles errores y discrepancias de tipos en el código. Esto ayuda a capturar errores comunes antes de que se ejecute el código y mejora la calidad y robustez del programa.
6. *Tipado estructural:* TypeScript utiliza un sistema de tipado estructural en lugar de un tipado nominal. Esto significa que los tipos son compatibles si tienen una estructura similar, independientemente de su nombre. Esto permite realizar asignaciones más flexibles y facilita la interoperabilidad entre diferentes partes del código.

El tipado en TypeScript brinda ventajas como la **detección temprana de errores**, mejor mantenibilidad del código, autocompletado y documentación más precisa, y una mayor confianza en el comportamiento del programa. Sin embargo, es importante destacar que TypeScript no impone restricciones de tipos en tiempo de ejecución, ya que se compila a JavaScript, que es un lenguaje de tipado dinámico. **La verificación de tipos solo ocurre en tiempo de compilación**.

Comparación con Oz

Si comparamos con **Oz**, estos dos lenguajes difieren en cuanto a sus enfoques y características relacionadas con el tipado:

Por un lado, Oz:

- Oz es un *lenguaje dinámicamente tipado*, lo que significa que no se requiere declarar los tipos de datos de manera explícita en el código.
- En Oz, los valores y las variables pueden cambiar de tipo durante la ejecución del programa.
- El tipado en Oz se basa en el concepto de *pattern-matching*, donde los patrones se utilizan para verificar la estructura de los datos en tiempo de ejecución.

Por otro lado, TypeScript:

- TypeScript es un lenguaje *estáticamente tipado* que proporciona un sistema de tipos opcional.
- En TypeScript, los tipos de datos deben declararse explícitamente en el código, lo que brinda mayor claridad y documentación sobre los tipos de datos esperados.
- El sistema de tipos de TypeScript permite detectar errores de tipo durante la compilación, antes de la ejecución del programa, lo que ayuda a prevenir errores comunes y mejorar la calidad del código.

En resumen, Oz es dinámicamente tipado, lo que significa que los tipos de datos no se declaran explícitamente y pueden cambiar durante la ejecución del programa. Por otro lado, TypeScript es estáticamente tipado y requiere que los tipos de datos se declaren explícitamente en el código fuente. Esto permite detectar errores de tipo en tiempo de compilación y brinda una mayor seguridad y robustez al código.

4.4 Control de flujos

El control de flujo se refiere a las estructuras y mecanismos utilizados para dirigir la ejecución del código en diferentes direcciones, según ciertas condiciones o eventos. TypeScript proporciona varias construcciones de control de flujo que permiten tomar decisiones, ejecutar código repetidamente y manejar excepciones.

Algunas de las principales construcciones de control de flujo en TypeScript son las siguientes:

1. *Sentencia if/else:* La sentencia if/else permite ejecutar un bloque de código si se cumple una determinada condición, y un bloque alternativo (else) si la condición no se cumple. Por ejemplo:

```
if (condition) {
  // código a ejecutar si la condición es verdadera
} else {
// código a ejecutar si la condición es falsa
}
```

2. *Sentencia switch:* La sentencia switch se utiliza para seleccionar uno de varios bloques de código para su ejecución, según el valor de una expresión.

```
switch (expression) {
  case value1:
    // código a ejecutar si la expresión es igual a value1
    break;
  case value2:
    // código a ejecutar si la expresión es igual a value2
    break;
  // más casos...
  default:
     // código a ejecutar si ninguno de los casos anteriores se
cumple
}
```

3. *Bucles:* TypeScript ofrece diferentes tipos de bucles para repetir la ejecución de un bloque de código.
    - *Bucle for:* El bucle for se utiliza cuando se conoce el número exacto de iteraciones.

```
for (let i = 0; i < 5; i++) {
// código a ejecutar en cada iteración
}
```

- *Bucle while:* El bucle while se utiliza cuando la condición de finalización no se conoce de antemano.

```
while (condition) {
  // código a ejecutar mientras la condición sea verdadera
}
```

- *Bucle do-while:* Similar al bucle while, pero la condición se verifica después de cada iteración, asegurando que el bloque de código se ejecute al menos una vez.

```
do {
  // código a ejecutar al menos una vez
} while (condition);
```

- *Sentencia try/catch/finally:* La sentencia 'try/catch/finally' se utiliza para manejar excepciones y errores en el código.

```
try {
  // código que podría lanzar una excepción
} catch (error) {
  // código para manejar la excepción
} finally {
  // código a ejecutar siempre, independientemente de si se lanzó una excepción o no
}
```

Estas son solo algunas de las construcciones de control de flujo disponibles en TypeScript. Estas herramientas permiten controlar el flujo de ejecución del programa, tomar decisiones basadas en condiciones y manejar excepciones, lo que hace que el código sea más flexible y capaz de adaptarse a diferentes situaciones.

Excepciones

Las *excepciones* se pueden utilizar como control de flujo.
Una excepción es una manera de, a un bloque de código, hacerlo seguro a posibles errores no controlados. Por lo tanto, en caso que se lanze un error, uno podría evaluar y hacer algo. Además, una excepción, en general, interrumpe el stack. Si no tenemos ningún catch, esa excepción llegará al usuario. En cambio, si ponemos un catch, se ejecuta lo que pongamos dentro.

Como explicamos previamente, Typescript utiliza la sentencia 'try/catch/finally' para manejar excepciones y errores en el código.

Si comparamos con **Oz**, este tiene algo muy parecido. Oz hace uso del siguiente *semantic statement*: `(try <s1> catch <x> then <s2> end, E)`.

1) Se apila el st (catch <x> then <s2> end, E) en ST
2) Se apila (<s1>, E) en ST

Comparación con Oz

Si comparamos con **Oz**, estos dos lenguajes difieren en cuanto al control de flujo y las estructuras de control que ofrecen:

Por un lado, Oz:

- Oz ofrece estructuras de control de flujo típicas, como condicionales (if-then-else) y bucles (for, while).
- Además, Oz cuenta con un mecanismo poderoso llamado *"restricciones"* (constraints), que permite expresar y resolver problemas de forma declarativa. Estas restricciones permiten especificar relaciones lógicas entre variables y definir restricciones matemáticas.

Por otro lado, TypeScript:

- TypeScript también proporciona las estructuras de control de flujo comunes, como condicionales (if-else) y bucles (for, while). Estas estructuras son similares a las de otros lenguajes de programación basados en C.
- TypeScript también admite *operadores lógicos*, como el operador ternario (?:), que permite tomar decisiones basadas en condiciones en una sola línea de código.

Además de las estructuras de control convencionales, tanto Oz como TypeScript pueden hacer uso de funciones y expresiones lambda para expresar el control de flujo en formas más flexibles. Por ejemplo, pueden usarse *funciones de orden superior* como map, filter y reduce para realizar transformaciones y operaciones en colecciones de datos. Por ejemplo, ahora mostraremos el caso de cómo sería en Oz y de cómo sería en TypeScript.

En Oz:

```
declare
   fun {DoubleList list}
      case list
      of nil then nil
      [] X|Xs then X*2|{DoubleList Xs}
      end
   end
in
```

```
   {Browse {DoubleList [1 2 3 4 5]}}
end
```

En este ejemplo de Oz, la función 'DoubleList' recibe una lista 'list' y la duplica multiplicando cada elemento por 2. Utiliza la recursión para aplicar la multiplicación a cada elemento de la lista y devuelve una nueva lista con los elementos duplicados. El resultado se muestra con '{Browse}' para verlo en la salida.

En TypeScript:

```
const numbers = [1, 2, 3, 4, 5];

// Utilizando map para duplicar los elementos de la lista
const doubledNumbers = numbers.map((num) => num * 2);
console.log(doubledNumbers); // Resultado: [2, 4, 6, 8, 10]

// Utilizando filter para obtener solo los números pares de la lista
const evenNumbers = numbers.filter((num) => num % 2 === 0);
console.log(evenNumbers); // Resultado: [2, 4]

// Utilizando reduce para obtener la suma de todos los elementos de la
lista
const sum = numbers.reduce((accumulator, num) => accumulator + num, 0);
console.log(sum); // Resultado: 15
```

En este ejemplo de TypeScript, utilizamos las funciones de orden superior 'map', 'filter' y 'reduce' en la lista de números.

- 'map' se utiliza para duplicar cada elemento.
- 'filter' para obtener solo los números pares.
- 'reduce' para calcular la suma de todos los elementos.

Los resultados se imprimen en la consola.

Estos ejemplos ilustran cómo se pueden utilizar las *funciones de orden superior* para realizar transformaciones y operaciones en colecciones de datos tanto en Oz como en TypeScript.

En resumen, tanto Oz como TypeScript proporcionan estructuras de control de flujo estándar, como condicionales y bucles. Sin embargo, Oz ofrece un enfoque más declarativo a través del uso de restricciones, mientras que TypeScript se centra en una sintaxis más convencional.

4.5 TDAs

Los TDAs (*Tipos de Datos Abstractos*) son una parte fundamental de la programación y se refieren a estructuras de datos que encapsulan un conjunto de valores y las operaciones que se pueden realizar sobre esos valores. Los TDAs se definen

en términos de su comportamiento y funcionalidad, sin exponer los detalles internos de implementación.

En TypeScript, los TDAs se pueden implementar utilizando clases, interfaces y funciones. Algunos ejemplos comunes de TDAs son los siguientes:

1. *Listas:* Una lista es una colección ordenada de elementos donde se pueden realizar operaciones como agregar un elemento, eliminar un elemento, buscar un elemento, etc. En TypeScript, se puede implementar una lista utilizando una clase o una interfaz que defina las operaciones y propiedades asociadas con una lista.
2. *Pilas:* Una pila es una estructura de datos en la que el último elemento que se agrega es el primero en ser eliminado (LIFO - *Last In First Out*). Se puede implementar una pila utilizando una clase o una interfaz que defina las operaciones de apilar (push) y desapilar (pop).
3. *Colas*: Una cola es una estructura de datos en la que el primer elemento que se agrega es el primero en ser eliminado (FIFO - *First In First Out*). Se puede implementar una cola utilizando una clase o una interfaz que defina las operaciones de encolar (enqueue) y desencolar (dequeue).

4. *Conjuntos:* Un conjunto es una colección de elementos sin repetición y sin un orden específico. Se puede implementar un conjunto utilizando una clase o una interfaz que defina las operaciones de agregar un elemento, eliminar un elemento, verificar si un elemento existe, etc. Esto en TypeScript se llama Set.
5. *Diccionarios:* Un diccionario es una estructura de datos que almacena pares clave-valor, donde cada clave es única. Se puede implementar un diccionario utilizando una clase o una interfaz que defina las operaciones de agregar un par clave-valor, eliminar un par clave-valor, buscar un valor por clave, etc. Esto en TypeScript se llama Map.

Estos son solo ejemplos básicos de TDAs que se pueden implementar en TypeScript. La idea principal de los TDAs es proporcionar abstracciones que oculten los detalles internos de implementación y se centren en la funcionalidad y las operaciones que se pueden realizar sobre los datos. Esto facilita la reutilización del código, mejora la modularidad y ayuda a mantener un diseño claro y estructurado en los programas.

Comparación con Oz

Si comparamos con **Oz**, estos dos lenguajes difieren en cuanto a la forma en que abordan los *Tipos Abstractos de Datos (TDAs)*:

Por un lado, Oz:

- En Oz, no existe una construcción específica para definir Tipos Abstractos de Datos (TDA) de manera nativa en el lenguaje.
- Sin embargo, Oz proporciona un mecanismo flexible para la definición y manipulación de estructuras de datos utilizando registros, listas y otros tipos de datos básicos.
- Los programadores pueden definir sus propios TDAs mediante combinaciones de registros y otros tipos de datos, aplicando las restricciones lógicas de Oz para garantizar la integridad y el comportamiento deseado.

Por otro lado, TypeScript:

- TypeScript ofrece soporte nativo para la definición de Tipos Abstractos de Datos a través de las interfaces y las clases.
- Las interfaces en TypeScript permiten describir la forma y las propiedades que deben tener los objetos que implementen ese TDA. Proporcionan una abstracción para definir la estructura de datos y los métodos asociados.
- Las clases en TypeScript permiten definir tanto la estructura como el comportamiento de los objetos, encapsulando el estado y proporcionando métodos para interactuar con ellos.

En resumen, en Oz no hay una construcción específica para definir TDAs, pero se pueden utilizar registros y restricciones lógicas para modelar estructuras de datos más complejas. Por otro lado, TypeScript proporciona soporte nativo para definir TDAs a través de interfaces y clases, permitiendo una abstracción más clara y fuerte del comportamiento y la estructura de los objetos.

4.6 Parámetros o cualquier concepto básico particular que soporta el lenguaje

Algunos parámetros que son soportados por el lenguaje TypeScript:

1. *Parámetros de función:* En TypeScript, se pueden definir funciones que tomen parámetros. Además, se puede especificar los tipos de los parámetros para garantizar que se pasen los tipos correctos de argumentos a la función. Por ejemplo:

```
function greet(name: string) {
  console.log(`Hello, ${name}!`);
}

greet("John"); // Llama a la función greet pasando el argumento "John"
```

2. *Parámetros opcionales:* se pueden declarar parámetros de función como opcionales añadiendo un signo de interrogación (?) al final del nombre del parámetro. Los parámetros opcionales pueden tener valores predeterminados.

```
function greet(name?: string) {
  if (name) {
    console.log(`Hello, ${name}!`);
  } else {
    console.log("Hello, anonymous!");
  }
}

greet(); // Llama a la función greet sin pasar ningún argumento
```

3. *Parámetros por defecto:* se pueden asignar valores predeterminados a los parámetros de una función. En estos casos, si no se proporciona un valor para ese parámetro, se utilizará el valor predeterminado.

```
function greet(name: string = "anonymous") {
  console.log(`Hello, ${name}!`);
}

greet(); // Llama a la función greet sin pasar ningún argumento
```

4. *Parámetros de rest/spread:* TypeScript admite los parámetros de rest y spread, que te permiten trabajar con un número variable de argumentos. Los parámetros de rest se representan con tres puntos (...), mientras que los parámetros de spread permiten expandir un arreglo o objeto en argumentos individuales.

```
function sum(...numbers: number[]) {
  return numbers.reduce((acc, num) => acc + num, 0);
}

console.log(sum(1, 2, 3)); // Llama a la función sum con múltiples argumentos

const values = [1, 2, 3];
console.log(sum(...values)); // Utiliza el operador spread para pasar un arreglo como argumentos individuales
```

Estos son solo algunos ejemplos de **conceptos y parámetros básicos que se soportan en TypeScript**. El lenguaje también ofrece una amplia gama de características y opciones avanzadas para trabajar con parámetros y funciones, como parámetros de solo lectura, funciones de alto orden, funciones flecha y más.

Comparación con Oz

Si comparamos con **Oz**, difieren en varios conceptos básicos y la forma en que manejan los parámetros.

A continuación, daremos una comparación entre los dos lenguajes en relación con algunos conceptos básicos:

1. **Funciones y parámetros**

Por un lado, Oz:

- En Oz, las funciones se definen utilizando el keyword 'fun' y los parámetros se pasan mediante la coincidencia de patrones. Oz permite la descomposición de estructuras complejas utilizando patrones de concordancia para acceder a los valores de los parámetros.
- Ejemplo:

```
fun {Sum X Y}
   X + Y
end

{Browse {Sum 2 3}} % Resultado: 5
```

Por otro lado, TypeScript:

- En TypeScript, las funciones se definen utilizando el keyword 'function' y los parámetros se declaran dentro de paréntesis. TypeScript ofrece tipado estático, lo que significa que los tipos de los parámetros se pueden especificar para obtener comprobaciones de tipo en tiempo de compilación.
- Ejemplo:

```
function sum(x: number, y: number): number {
  return x + y;
}

console.log(sum(2, 3)); // Resultado: 5
```

2. **Valores predeterminados de los parámetros**

Por un lado, Oz:

- En Oz, los valores predeterminados de los parámetros no son directamente soportados. Sin embargo, se pueden utilizar patrones de concordancia para lograr un comportamiento similar.

Por otro lado, TypeScript:

- En TypeScript, se pueden asignar valores predeterminados a los parámetros de función utilizando la sintaxis de asignación. Si un argumento no se proporciona al llamar a la función, se utilizará el valor predeterminado especificado.
- Ejemplo:

```
function greet(name: string = "John"): void {
  console.log(`Hello, ${name}!`);
}

greet(); // Resultado: Hello, John!
greet("Alice"); // Resultado: Hello, Alice!
```

En resumen, Oz y TypeScript tienen enfoques diferentes en cuanto a la definición y el manejo de funciones y parámetros. Oz utiliza la coincidencia de patrones para pasar y descomponer los parámetros, mientras que TypeScript utiliza la sintaxis más tradicional de parámetros declarados dentro de paréntesis. TypeScript también ofrece la capacidad de especificar tipos y asignar valores predeterminados a los parámetros.

5. Características avanzadas del lenguaje

Antes de entrar en detalle sobre algunas características específicas de TypeScript, mencionaremos algunas características avanzadas del lenguaje en general como para tener un mayor contexto.

Algunas características avanzadas del lenguaje TypeScript:

1. *Anotaciones de tipos avanzadas:* TypeScript permite anotaciones de tipos más complejas, como tipos de función, tipos genéricos, tipos condicionales, tipos literales y tipos de tuplas. Estas anotaciones de tipos avanzadas permiten expresar estructuras de datos más complejas y proporcionar un mayor nivel de precisión en la verificación estática de tipos.
2. *Inferencia de tipos condicionales:* TypeScript tiene la capacidad de inferir tipos basados en condiciones. Esto significa que el tipo de una variable puede cambiar dependiendo de ciertas comprobaciones condicionales en el código. Esto es especialmente útil cuando se trabaja con operadores de tipo condicional, como 'typeof' o 'keyof'.
3. *Clases y herencia:* TypeScript es un lenguaje orientado a objetos que admite la creación de clases y la herencia de propiedades y métodos. Se puede utilizar la sintaxis de clase para definir clases, aplicar modificadores de acceso ('public', 'private', 'protected'), implementar interfaces y utilizar herencia para extender la funcionalidad de una clase base.

4. *Decoradores:* Los decoradores son una característica de TypeScript que permite agregar metadatos y modificar el comportamiento de las clases, métodos, propiedades y otros elementos del código. Los decoradores se aplican utilizando la sintaxis '@nombreDecorador' y proporcionan una forma flexible de agregar funcionalidades adicionales a través de transformaciones en tiempo de compilación.
5. *Módulos y espacios de nombres:* TypeScript proporciona un sistema de módulos para organizar y reutilizar código. Puedes utilizar la palabra clave 'import' para importar elementos desde otros módulos y 'export' para exportar elementos de un módulo. Además, TypeScript también admite los espacios de nombres ('namespace') como una forma de agrupar lógicamente código relacionado.
6. *Tipos de declaración y ambientales:* TypeScript permite declarar tipos para librerías y entornos existentes mediante archivos de declaración ('.d.ts'). Estos archivos contienen definiciones de tipos para bibliotecas JavaScript y proporcionan información de tipo para el compilador de TypeScript. También, se puede utilizar las declaraciones ambientales para definir tipos para variables y objetos globales en entornos como el navegador o Node.js.
7. *Tipos condicionales:* TypeScript admite tipos condicionales ('conditional types') que permiten realizar inferencias de tipos basadas en condiciones. Esto permite escribir código que se adapte de manera inteligente según los tipos de los valores.
8. *Operadores y utilidades de tipo:* TypeScript proporciona varios operadores y utilidades de tipo que permiten realizar transformaciones y operaciones en tipos. Algunos ejemplos incluyen 'keyof' para obtener las claves de un tipo, 'typeof' para obtener el tipo de una expresión, 'Pick' y 'Omit' para seleccionar u omitir propiedades de un tipo, y muchos más.

Estas son solo algunas de las características más avanzadas que TypeScript ofrece. El lenguaje continúa evolucionando y añadiendo nuevas funcionalidades con cada versión, lo que lo convierte en una herramienta poderosa para desarrollar aplicaciones escalables y mantenibles.

5.1 Manejo de memoria

En TypeScript, el manejo de memoria es una preocupación que recae en el entorno en el que se ejecuta el código, como un navegador web o un entorno de ejecución de Node.js. TypeScript, al ser un lenguaje de alto nivel, se ejecuta en una máquina virtual o en un intérprete que se encarga del manejo de memoria subyacente.

A continuación, se presentan algunas consideraciones generales sobre el manejo de memoria en entornos comunes:

1. *Recolección de basura (Garbage Collection):* La mayoría de los entornos de ejecución modernos utilizan un recolector de basura para administrar automáticamente la memoria asignada a los objetos y liberarla cuando ya no se necesita. El recolector de basura realiza un seguimiento de los objetos que están en uso y libera automáticamente la memoria de los objetos que ya no son accesibles. Esto permite que los desarrolladores se enfoquen en la lógica de la aplicación sin preocuparse demasiado por la gestión manual de la memoria.
2. *Ciclos de vida de los objetos:* Es importante tener en cuenta los ciclos de vida de los objetos en TypeScript. Al crear objetos, es necesario asegurarse de liberar los recursos asociados una vez que ya no sean necesarios. Por ejemplo, cerrar conexiones de red, liberar recursos de archivos, cancelar suscripciones a eventos, etc. Esto se logra utilizando patrones como la liberación explícita de recursos o aprovechando los mecanismos de recolección de basura mencionados anteriormente.
3. *Gestión de memoria manual:* Aunque TypeScript no proporciona características específicas para la gestión manual de memoria, en entornos donde se requiere un control más fino sobre los recursos, como en desarrollo de juegos o aplicaciones intensivas en recursos, se pueden utilizar técnicas adicionales específicas del entorno para administrar la memoria de manera más precisa. Estas técnicas pueden incluir el uso de memoria compartida, administración de almacenamiento en caché o técnicas de administración de memoria personalizadas.
4. *Optimización de rendimiento:* Aunque el manejo de memoria está en gran medida automatizado en TypeScript, aún se pueden realizar optimizaciones de rendimiento para reducir la cantidad de memoria utilizada por una aplicación. Esto puede incluir técnicas como el uso de estructuras de datos eficientes, limitar el alcance de los objetos, evitar fugas de memoria y optimizar la forma en que se asignan y liberan los recursos.

Es importante tener en cuenta que **el manejo de memoria puede variar según el entorno de ejecución específico en el que se ejecute el código TypeScript**. Cada entorno puede tener sus propias consideraciones y mejores prácticas para garantizar un uso eficiente y seguro de la memoria. Por lo tanto, es recomendable consultar las documentaciones y pautas específicas del entorno que se esté utilizando para obtener más información sobre el manejo de memoria en ese contexto particular.

Comparación con Oz

Si comparamos con **Oz**, estos dos lenguajes no difieren tanto en cuanto al *manejo de memoria*:

Por un lado, Oz:

- En Oz, el manejo de memoria es realizado de manera transparente por el entorno de ejecución. El programador no tiene que preocuparse por la asignación y liberación de memoria explícitamente.
- Oz utiliza un sistema de recolección de basura (garbage collector) para liberar automáticamente la memoria ocupada por los objetos que ya no son accesibles.
- El *recolector de basura* en Oz realiza un seguimiento de los objetos utilizados en el programa y periódicamente libera la memoria de los objetos que no tienen referencias válidas, lo que simplifica la administración de la memoria para el programador.

Por otro lado, TypeScript:

- TypeScript se ejecuta en entornos que utilizan el motor de JavaScript subyacente, que generalmente implementa un *recolector de basura*.
- Al igual que en Oz, el manejo de memoria en TypeScript también es automático y está gestionado por el recolector de basura del motor JavaScript subyacente.
- Los objetos que ya no son accesibles se identifican y se liberan automáticamente cuando el recolector de basura determina que ya no hay referencias válidas a ellos.

En resumen, tanto Oz como TypeScript utilizan un sistema de recolección de basura para el manejo de memoria. En Oz, la recolección de basura es manejada internamente por el entorno de ejecución de Oz, mientras que en TypeScript se beneficia del recolector de basura del motor JavaScript subyacente. Esto permite que los programadores se enfoquen en la lógica del programa sin tener que preocuparse por la asignación y liberación de memoria de forma explícita.

5.2 Manejo de concurrencia

El manejo de concurrencia se refiere a cómo un programa administra y coordina la ejecución de múltiples tareas o procesos simultáneamente. En TypeScript, el manejo de concurrencia puede ser abordado de varias formas dependiendo del entorno de ejecución y de los requisitos específicos de la aplicación.

A continuación, se presentan algunas técnicas y conceptos comunes relacionados con el manejo de concurrencia:

1. *Hilos (Threads)*: Los hilos son unidades de ejecución individuales que pueden ejecutarse de forma concurrente dentro de un proceso. Sin embargo, TypeScript en sí mismo no ofrece soporte directo para la programación de hilos. En entornos de Node.js, se pueden aprovechar los módulos nativos como 'worker_threads' para crear hilos y realizar operaciones concurrentes. En el caso de aplicaciones web en el navegador, se puede utilizar Web Workers para ejecutar tareas en segundo plano.
2. *Event Loop:* El modelo de concurrencia principal en JavaScript y TypeScript se basa en el Event Loop. El Event Loop es una estructura que permite manejar tareas asíncronas y eventos de forma eficiente sin bloquear el hilo principal de ejecución. La mayoría de las operaciones asíncronas en TypeScript, como operaciones de red, acceso a bases de datos y solicitudes HTTP, se basan en devoluciones de llamada (callbacks), promesas o async/await, lo que permite que el **código se ejecute de manera concurrente sin bloqueos**.
3. *Promesas y async/await:* **Las promesas son un mecanismo para gestionar operaciones asíncronas en TypeScript**. Permiten encadenar tareas y manejar de manera efectiva el flujo asíncrono del código. La sintaxis 'async/await' es una adición más reciente a JavaScript/TypeScript que simplifica aún más el manejo de tareas asíncronas al permitir escribir código asíncrono de manera más similar a código síncrono secuencial.
4. *Mutex y Semáforos:* En escenarios donde se requiere la sincronización de acceso a recursos compartidos entre hilos o tareas, se pueden utilizar mecanismos de sincronización como mutex y semáforos. Estos permiten establecer bloqueos y asegurar que solo un hilo o tarea tenga acceso a un recurso compartido a la vez, evitando condiciones de carrera y conflictos de datos.
5. *Programación Reactiva:* La programación reactiva se centra en el flujo de eventos y las notificaciones de cambios en lugar de esperar activamente por resultados. Se pueden utilizar bibliotecas y patrones de programación reactiva como RxJS para manejar concurrencia y eventos de manera más eficiente y componer flujos de datos asincrónicos.

Es importante tener en cuenta que el manejo de concurrencia puede ser complejo y debe ser abordado con cuidado para evitar problemas como condiciones de carrera, bloqueos y cuellos de botella de rendimiento. La elección de la técnica y el enfoque de concurrencia adecuados dependerá de los requisitos específicos de la aplicación y del entorno en el que se ejecute. Es recomendable estudiar y comprender las mejores prácticas y herramientas disponibles para el manejo de concurrencia en el entorno específico en el que se esté trabajando.

Concurrencia declarativa

La *concurrencia declarativa* es la concurrencia que vive bajo el paradigma declarativo. Es determinístico lo cual significa que ante un mismo input, ocurre el mismo output.

Ahora realizaremos una comparación con **Oz**.

Por ejemplo:

```
local X0 X1 X2 X3 in
  thread X1 = 1 + X0 end
  thread X3 = X1 + X2 end
  X0 = 4
  X2 = 2
  { Browse [X0 X1 X2 X3] }
end
```

Lo que se quiere demostrar con este ejemplo es que hay veces que no sabemos cuándo vamos a tener cierto valor. La ventaja de los threads es que son bloqueantes. Por lo tanto, se podría bloquear hasta que tengamos el valor de una variable.
En este ejemplo, podemos observar que Oz intentará ejecutar el primer thread. Si X0 no tiene valor todavía, se bloqueará. Recién cuando tengamos el valor, se desbloqueará el statement. En cambio, en TypeScript se maneja de forma distinta. Oz lo maneja sólo pero en este otro lenguaje deberíamos poner `X1 = 1 + await X0`.

En resumen, la concurrencia determinística implica que no importa cuál se ejecuta primero porque se van a ejecutar los dos de todas formas, no nos importa el orden. Pero, por supuesto, en el código hay cosas que se ejecutan en un orden para que tenga sentido.

- Oz: lo maneja sólo
- TypeScript: se usa el recurso 'await' para manejar este caso.

Comparación con Oz

Si comparamos con **Oz**, estos dos lenguajes difieren bastante en cuanto al *manejo de concurrencia*:

Por un lado, Oz:

- Oz está diseñado desde el principio para brindar soporte nativo y eficiente para la concurrencia y el paralelismo.
- En Oz, la concurrencia se basa en el modelo de cálculo llamado *"computación por restricciones"* (constraint-based computation), que permite expresar y resolver problemas concurrentes de manera declarativa.

- Oz proporciona primitivas de concurrencia como los *"actores"* (actors), que son unidades de ejecución independientes que se comunican entre sí mediante el envío de mensajes.
- Los actores en Oz ejecutan concurrentemente y pueden compartir información a través de canales de comunicación, lo que permite la construcción de sistemas concurrentes y paralelos de manera modular y segura.

Por otro lado, TypeScript:

- TypeScript, al ser un superconjunto de JavaScript, hereda su modelo de concurrencia basado en el *bucle de eventos* (event loop) y las operaciones asíncronas no bloqueantes.
- TypeScript aprovecha las características de JavaScript, como las *promesas* (promises) y las *funciones asíncronas* (async/await), para manejar la concurrencia y las operaciones asíncronas.
- La concurrencia en TypeScript se logra mediante la ejecución de tareas en segundo plano (background threads) o mediante la ejecución asincrónica de operaciones en el hilo principal, lo que evita bloquear la ejecución del programa mientras se espera la finalización de una operación.

Para ilustrar las diferencias en el manejo de concurrencia entre Oz y TypeScript vamos a mostrar dos ejemplos.

Ejemplo en Oz:

```
functor
import Thread
import System

declare
   thread
      proc {Worker}
         {Browse "Worker starting..."}
         System.sleep(1000) -- Simula una tarea de larga duración
         {Browse "Worker finished!"}
      end
   end
in
   {Browse "Main thread starting..."}
   Thread.fork(Worker)
   {Browse "Main thread continuing..."}
    System.sleep(2000) -- Espera para que el Worker tenga tiempo de
ejecución
   {Browse "Main thread finished!"}
end
```

En este ejemplo de Oz, se crea un *hilo* (thread) utilizando la función 'Thread.fork' para ejecutar una tarea en segundo plano. Mientras tanto, el hilo principal continúa ejecutándose y muestra mensajes en la salida. Luego, se utiliza 'System.sleep' para simular una espera de 2 segundos, permitiendo que el hilo Worker tenga tiempo de ejecución. Al ejecutar este programa, se observará que los mensajes del hilo principal y del hilo Worker se muestran de forma intercalada en la salida, demostrando la concurrencia.

Ejemplo en TypeScript:

```
function asyncFunction() {
   return new Promise((resolve) => {
      setTimeout(() => {
         console.log("Async function finished!");
         resolve();
      }, 2000);
   });
}

async function main() {
   console.log("Main function starting...");
   await asyncFunction();
   console.log("Main function finished!");
}

main();
```

En este ejemplo de TypeScript, se utiliza una función asíncrona ('async') junto con una promesa ('Promise') y 'setTimeout' para simular una operación asíncrona que demora 2 segundos en completarse. Dentro de la función principal ('main'), se muestra un mensaje antes y después de esperar a que la función asíncrona termine. Al ejecutar este código, se observará que el mensaje *"Main function finished!"* se muestra después del mensaje *"Async function finished!"*, ya que la función 'await' pausa la ejecución hasta que la promesa se resuelva.

Estos ejemplos ilustran cómo Oz y TypeScript abordan la concurrencia de manera diferente. En Oz, se utilizan hilos y actores para realizar tareas concurrentemente, mientras que en TypeScript se utilizan funciones asíncronas y promesas para manejar operaciones asíncronas de manera no bloqueante.

En resumen, Oz y TypeScript abordan la concurrencia de manera diferente. Oz tiene un enfoque intrínseco de concurrencia y paralelismo, proporcionando primitivas como actores para la comunicación entre unidades de ejecución independientes. TypeScript se basa en el modelo de concurrencia de JavaScript, utilizando características

como promesas y funciones asíncronas para manejar operaciones asíncronas y evitar bloqueos en la ejecución del programa.

5.3 Manejo de errores

El manejo de errores es una parte importante del desarrollo de software, ya que permite detectar y gestionar de manera adecuada las situaciones excepcionales o inesperadas que puedan ocurrir durante la ejecución de un programa. En TypeScript, existen varias técnicas y mecanismos para manejar y gestionar errores.

A continuación, se presentan algunas de ellas:

1. *Bloques try-catch-finally:* se puede utilizar bloques 'try-catch-finally' para capturar y manejar errores específicos. Dentro del bloque 'try', se coloca el código que puede generar una excepción. Si se produce una excepción, el bloque 'catch' se ejecuta, permitiendo manejar el error de manera apropiada. El bloque 'finally' opcional se ejecuta siempre, ya sea que se produzca una excepción o no, y se utiliza para realizar acciones de limpieza.

```
try {
  // Código que puede generar una excepción
} catch (error) {
  // Manejo del error
} finally {
  // Acciones de limpieza
}
```

   Como se mencionó previamente, si comparamos con **Oz**, este tiene algo muy parecido. Oz hace uso del siguiente *semantic statement*: `(try <s1> catch <x> then <s2> end, E)`.

   1) Se apila el st (catch <x> then <s2> end, E) en ST
   2) Se apila (<s1>, E) en ST

2. *Lanzamiento de excepciones:* se pueden lanzar excepciones manualmente utilizando la palabra clave 'throw'. Esto permite indicar que ha ocurrido un error en una determinada parte del código y detener la ejecución normal.

```
function divide(a: number, b: number): number {
  if (b === 0) {
    throw new Error("División por cero no permitida");
  }
  return a / b;
}
```

```
try {
  const result = divide(10, 0);
} catch (error) {
  console.log(error.message);
}
```

Si comparamos con **Oz**, este tiene algo muy parecido. Oz hace uso del siguiente *semantic statement*: `(raise <x> end, E)` en donde el raise puede ser implícito o explícito. Básicamente, lo que se hace es lo siguiente. Se empiezan a descartar elementos del ST hasta encontrar un catch. Si no hay un catch, se finaliza la ejecución con *"Uncaught exception"*. En cambio, si hay un catch apilado, supongamos `(catch <y> then <s> end, Ec)`, luego se apila (<s>, Ec + {<y> ->E(<x>)}).

3. *Tipos de errores personalizados:* se pueden crear propios tipos de errores personalizados extendiendo la clase 'Error' incorporada en TypeScript. Esto permite definir errores con información adicional y comportamiento específico.

```
class CustomError extends Error {
  constructor(message: string) {
    super(message);
    this.name = "CustomError";
  }
}

try {
  throw new CustomError("Ocurrió un error personalizado");
} catch (error) {
  console.log(error.name); // "CustomError"
  console.log(error.message); // "Ocurrió un error personalizado"
}
```

4. *Uso de devoluciones de llamada (callbacks):* en entornos asincrónicos, como Node.js o el manejo de eventos en el navegador, a menudo se utilizan devoluciones de llamada (callbacks) para manejar errores en operaciones asíncronas. Por lo general, se sigue una convención donde el primer parámetro de la devolución de llamada es un objeto de error, si ocurre alguno, y el segundo parámetro es el resultado o la respuesta exitosa.

```
function fetchData(callback: (error: Error | null, data?: any) =>
void) {
  // Simulación de operación asincrónica
  setTimeout(() => {
    const error = null; // O null o un objeto de error
    const data = { name: "John", age: 30 };
    callback(error, data);
```

```
  }, 2000);
}

fetchData((error, data) => {
  if (error) {
    console.log("Ocurrió un error:", error.message);
  } else {
    console.log("Datos recibidos:", data);
  }
});
```

Si comparamos con **Oz**, las funciones en Oz son ciudadanos de primera clase (*First Class*). Es decir, las funciones, como cualquier otra variable, pueden ser asignadas a una variable. Por ejemplo:

```
local Ints L in
  Ints = fun lazy {$ N}
    N | {Ints N+1}
end
L = {Ints 1}
```

En este ejemplo, a la función Ints se le asigna una variable.

Poder pasar funciones como parámetro a otras funciones se conoce como programación de alto orden (*High Order Programming*). Si a su vez esa función recibe otra función, es de segundo orden. Es decir, podemos ir apilando funciones.

Por lo tanto, esto es lo que utiliza TypeScript para hacer **callbacks**.

Estas son solo algunas técnicas básicas para manejar errores en TypeScript. Sin embargo, hay que tener en cuenta que el manejo de errores puede ser un tema complejo y puede haber diferentes enfoques según el contexto y los requisitos específicos de la aplicación. Es importante tener una estrategia coherente para el manejo de errores y considerar la posibilidad de registrar y notificar los errores, así como de proporcionar mensajes de error claros y útiles para facilitar la depuración y el mantenimiento del código.

Comparación con Oz

Si comparamos con **Oz**, estos dos lenguajes difieren en cuanto al *manejo de errores*:

Por un lado, Oz:

- En Oz, el manejo de errores se basa en el concepto de *"falla silenciosa"* (silent failure). Esto significa que, por defecto, los errores no se propagan automáticamente y no interrumpen la ejecución del programa.

- En lugar de lanzar excepciones, Oz utiliza valores especiales, como 'nil' o 'fail', para indicar condiciones de error.
- Los programadores de Oz deben utilizar construcciones de control de flujo, como la concordancia de patrones y los operadores lógicos, para manejar explícitamente las condiciones de error y tomar acciones adecuadas.
- Además, Oz ofrece primitivas como 'try', 'catch', 'raise' y 'fail' para controlar los errores de forma más explícita cuando sea necesario.

Por otro lado, TypeScript:

- En TypeScript, el manejo de errores se basa en el uso de *excepciones*, siguiendo el modelo de excepciones de JavaScript.
- Las excepciones se lanzan cuando ocurre un error y pueden ser capturadas y manejadas utilizando bloques 'try-catch'.
- Los programadores de TypeScript pueden utilizar la declaración 'throw' para lanzar excepciones y utilizar bloques 'try-catch' para capturar y manejar las excepciones lanzadas.
- TypeScript también permite especificar tipos de excepciones que pueden ser lanzadas por una función utilizando la sintaxis de tipo de retorno de la función.

Para ilustrar el manejo de errores tanto en Oz como en TypeScript vamos a mostrar dos ejemplos.

Ejemplo en Oz:

```
declare
   proc {Divide x y}
      if y == 0 then
         raise divideByZero
      else
         x/y
      end
   end

   proc {Main}
      try
         {Browse {Divide 10 0}}
      catch divideByZero then
         {Browse "Error: Division by zero"}
      end
   end
in
   {Main}
end
```

En este ejemplo de Oz, se define una función 'Divide' que intenta realizar una división. Si el divisor ('y') es igual a cero, se lanza una excepción personalizada llamada

'divideByZero'. Luego, en el procedimiento 'Main', se realiza una llamada a 'Divide' dentro de un bloque 'try-catch'. Si se lanza la excepción 'divideByZero', se captura en el bloque 'catch' y se muestra un mensaje de error. Al ejecutar este programa, se mostrará el mensaje *"Error: Division by zero"*, ya que se ha capturado la excepción.

Ejemplo en TypeScript:

```
function divide(x: number, y: number): number {
   if (y === 0) {
      throw new Error("Division by zero");
   }
   return x / y;
}

function main() {
   try {
      console.log(divide(10, 0));
   } catch (error) {
      console.log("Error:", error.message);
   }
}

main();
```

En este ejemplo de TypeScript, se define una función 'divide' que intenta realizar una división. Si el divisor ('y') es igual a cero, se lanza una excepción de tipo 'Error' con un mensaje específico. Luego, en la función 'main', se realiza una llamada a 'divide' dentro de un bloque 'try-catch'. Si se lanza una excepción durante la división, se captura en el bloque 'catch' y se muestra un mensaje de error. Al ejecutar este código, se mostrará el mensaje *"Error: Division by zero"*, ya que se ha capturado la excepción.

Estos ejemplos ilustran cómo Oz y TypeScript manejan las excepciones de manera similar. En Oz, se utilizan construcciones de control de flujo, como 'try-catch', para manejar explícitamente las condiciones de error y tomar acciones adecuadas. Por el otro lado, en TypeScript, también se utilizan bloques 'try-catch' para capturar y manejar las excepciones lanzadas. Además, en TypeScript se pueden lanzar excepciones de cualquier tipo, incluyendo las personalizadas, utilizando la instrucción 'throw new Error()'.

En resumen, Oz y TypeScript son similares en algunos aspectos pero también difieren en su enfoque para el manejo de errores. En Oz, los errores no se propagan automáticamente y se utilizan valores especiales para indicar condiciones de error, mientras que en TypeScript se utilizan excepciones y bloques 'try-catch' para manejar los errores de manera más tradicional.

5.4 Paralelismo

El paralelismo es la capacidad de realizar múltiples tareas simultáneamente, lo que puede conducir a una mejora en el rendimiento y la eficiencia de un programa. Es decir, es el mecanismo en el cual podemos hacer ejecuciones que no sean en un único hilo. Hay que tener cuidado ya que que dos operaciones se ejecuten de forma independiente no significa que se ejecuten todas al mismo tiempo. Siempre hay un lenguaje que no permite esto y otros que sí.

Por ejemplo, en TypeScript, casi que ni tiene paralelismo. En este lenguaje, el paralelismo puede lograrse utilizando diferentes enfoques según el entorno de ejecución y los requisitos específicos de la aplicación.

A continuación, se presentan algunas técnicas comunes para lograr el paralelismo:

1. *Hilos (Threads):* Los hilos son unidades de ejecución independientes que pueden ejecutarse simultáneamente dentro de un programa. Aunque TypeScript en sí mismo no ofrece soporte directo para la programación de hilos, en entornos como Node.js, se pueden aprovechar módulos nativos como 'worker_threads' para crear hilos y realizar tareas concurrentes.

   En **Oz**, existe la palabra reservada 'thread' que significa que se ejecuta en un *hilo independiente*. Por ejemplo:

```
local A B C in
  thread
    A = B + C
  end
  thread
    C = 4
  end
  thread
    B = 10
  end
end
```

   En este ejemplo, hay tres hilos que corren de manera independiente y cada uno hace algo.
   Una comparación que se puede hacer entre Oz y TypeScript es qué sucede en el caso de si comentamos los threads. En este caso, B y C son variables ligadas, es decir, no tienen valor que nosotros le hayamos declarado. Por lo tanto, pueden pasar distintas cosas dependiendo del lenguaje.
   - TypeScript: le asigna un valor por defecto (undefined)
   - Oz: la decisión es lockearse. Esto se llama *Dataflow*. Básicamente, lo que sucede en Oz es que se lockea hasta que las variables B y C tengan valor y, de esta forma, así destrabar el hilo que intenta asignarle valor a A.

2. *Procesos:* Un proceso es una instancia de un programa en ejecución que tiene su propio espacio de memoria y recursos asignados. En entornos como Node.js, se puede aprovechar el módulo 'child_process' para crear procesos independientes y realizar tareas en paralelo. Los procesos pueden comunicarse entre sí mediante canales de comunicación como tuberías o mensajes.
3. *Programación asincrónica:* TypeScript se basa en gran medida en la **programación asincrónica para lograr el paralelismo.** Se pueden aprovechar las devoluciones de llamada (callbacks), las promesas y la sintaxis 'async/await' para ejecutar tareas de manera asincrónica y aprovechar al máximo los recursos de la CPU mientras se esperan respuestas de operaciones de entrada/salida (por ejemplo, solicitudes de red, acceso a bases de datos).
4. *Event Loop:* El modelo de concurrencia basado en el Event Loop de JavaScript y TypeScript permite manejar múltiples tareas de forma concurrente sin bloquear el hilo principal de ejecución. Mediante la programación de devoluciones de llamada y el uso de funciones asíncronas, se puede lograr la ejecución paralela de tareas sin tener que crear y administrar hilos o procesos manualmente.
5. *Bibliotecas y marcos de trabajo:* En el ecosistema de TypeScript, existen bibliotecas y marcos de trabajo que facilitan la implementación del paralelismo. Algunas bibliotecas populares incluyen Worker Threads, Async/Await Parallel, Parallel.js y RxJS, entre otras. Estas bibliotecas proporcionan abstracciones y utilidades para simplificar la programación paralela y el manejo de tareas concurrentes.

Es importante tener en cuenta que el paralelismo puede tener implicaciones en la sincronización y la concurrencia de los datos. Debemos asegurarnos de que las operaciones paralelas no generen condiciones de carrera ni conflictos de datos. Además, hay que tener en cuenta que el rendimiento y los beneficios del paralelismo pueden variar según el entorno de ejecución y la capacidad del hardware en el que se ejecute la aplicación.

Es recomendable estudiar y comprender las mejores prácticas y técnicas específicas del entorno en el que se esté trabajando para aprovechar al máximo las capacidades de paralelismo disponibles y optimizar el rendimiento de la aplicación.

Comparación con Oz

Si comparamos con **Oz**, estos dos lenguajes difieren en cuanto al *paralelismo*:

Por un lado, Oz:

- Oz está diseñado para facilitar el paralelismo y la ejecución concurrente de manera nativa.
- En Oz, se utilizan los *actores* (actors) como primitivas de concurrencia para lograr el paralelismo.
- Los actores en Oz son unidades de ejecución independientes que se comunican entre sí mediante el envío de mensajes. Cada actor tiene su propia ejecución secuencial y su propio estado, lo que permite la ejecución paralela de múltiples actores.
- Los actores en Oz pueden ejecutarse en múltiples procesadores o núcleos de CPU, aprovechando así el paralelismo a nivel de hardware.

Por otro lado, TypeScript:

- TypeScript, al ser un lenguaje basado en JavaScript, no ofrece soporte nativo para el paralelismo de alto nivel.
- Sin embargo, TypeScript se beneficia de las capacidades de JavaScript para el paralelismo a nivel de *hilos* (Web Workers) y *operaciones asíncronas*.
- Los *Web Workers* en JavaScript permiten ejecutar tareas en segundo plano utilizando hilos separados, lo que puede proporcionar cierto nivel de paralelismo.
- TypeScript también puede aprovechar las *operaciones asíncronas*, como promesas y funciones asíncronas, para ejecutar tareas en paralelo, aunque de forma limitada, ya que estas operaciones están limitadas al contexto de ejecución del motor JavaScript subyacente.

Para ilustrar las diferencias en el manejo del paralelismo entre Oz y TypeScript vamos a mostrar dos ejemplos.

Ejemplo en Oz:

```
functor
import Thread
import System

declare
   thread
      proc {Worker}
         {Browse "Worker starting..."}
         System.sleep(1000) -- Simula una tarea de larga duración
         {Browse "Worker finished!"}
      end
   end
in
   {Browse "Main thread starting..."}
   Thread.fork(Worker)
```

```
   {Browse "Main thread continuing..."}
    System.sleep(2000) -- Espera para que el Worker tenga tiempo de
ejecución
   {Browse "Main thread finished!"}
end
```

En este ejemplo de Oz, se crea un *hilo* (thread) utilizando la función 'Thread.fork' para ejecutar una tarea en segundo plano. Mientras tanto, el hilo principal continúa ejecutándose y muestra mensajes en la salida. Luego, se utiliza 'System.sleep' para simular una espera de 2 segundos, permitiendo que el hilo Worker tenga tiempo de ejecución. Al ejecutar este programa, se observará que los mensajes del hilo principal y del hilo Worker se muestran de forma intercalada en la salida, demostrando el paralelismo.

Ejemplo en TypeScript: (utilizando Web Workers)

```
// main.ts
const worker = new Worker('worker.js');

console.log('Main thread starting...');

worker.onmessage = (event) => {
  console.log(`Received message from worker: ${event.data}`);
};

worker.postMessage('Start task');

console.log('Main thread continuing...');

// worker.js
console.log('Worker thread starting...');

self.onmessage = (event) => {
  console.log(`Received message from main thread: ${event.data}`);
  setTimeout(() => {
    console.log('Task finished!');
    self.postMessage('Task completed');
  }, 1000);
};
```

En este ejemplo de TypeScript, se utiliza un Web Worker para lograr un nivel básico de paralelismo. El archivo 'main.ts' crea un nuevo Web Worker a partir del archivo 'worker.js'. El hilo principal muestra mensajes antes y después de enviar un mensaje al Web Worker. El Web Worker, definido en el archivo 'worker.js', muestra mensajes cuando recibe un mensaje del hilo principal y realiza una tarea simulada utilizando 'setTimeout'. Luego, envía un mensaje de vuelta al hilo principal para indicar que la tarea ha finalizado. Al ejecutar este código en un entorno que admita Web Workers, se

observará que los mensajes del hilo principal y del Web Worker se muestran de forma intercalada en la salida, lo que demuestra el paralelismo a nivel de hilos.

Estos ejemplos ilustran cómo Oz y TypeScript abordan el paralelismo de manera diferente. En Oz, se utilizan actores para lograr el paralelismo y la ejecución concurrente, mientras que en TypeScript, se puede aprovechar el paralelismo utilizando Web Workers para ejecutar tareas en segundo plano en hilos separados.

En resumen, Oz y TypeScript tienen enfoques diferentes en cuanto al paralelismo. Oz proporciona primitivas de concurrencia, como los actores, para lograr el paralelismo y la ejecución concurrente de manera nativa. En cambio, TypeScript aprovecha las capacidades de JavaScript, como los Web Workers y las operaciones asíncronas, para lograr cierto nivel de paralelismo.

5.5 Cualquier concepto básico particular que soporta el lenguaje

A continuación, presentamos algunos conceptos básicos particulares que son compatibles con el lenguaje TypeScript:

1. *Decoradores:* los decoradores son una característica especial de TypeScript que permite agregar metadatos a clases, métodos, propiedades y otros elementos del código. Los decoradores se definen utilizando la sintaxis '@nombreDelDecorador' y se colocan justo encima del elemento que se desea decorar. Los decoradores son ampliamente utilizados en marcos de trabajo como Angular para agregar funcionalidades adicionales a las clases.
2. *Tipos de datos avanzados:* además de los tipos de datos básicos como *'number'*, *'string'* y *'boolean'*, TypeScript admite tipos de datos más avanzados como *'tuple'*, *'enum'*, *'any'*, *'void'*, *'null'*, *'undefined'*, *'never'* y *'unknown'*. Estos tipos proporcionan mayor flexibilidad y precisión al trabajar con datos en TypeScript.
3. *Union Types y Intersection Types:* TypeScript permite combinar múltiples tipos en un solo tipo utilizando Union Types ('|') e Intersection Types ('&').
    - Union Types permiten definir una variable o parámetro que puede aceptar uno de varios tipos diferentes.
    - Por otro lado, Intersection Types permiten combinar varios tipos en un solo tipo, donde la variable o parámetro debe satisfacer todas las propiedades y requisitos de cada tipo.
4. *Clases abstractas:* TypeScript admite la definición de clases abstractas, que son clases que no se pueden instanciar directamente, sino que se utilizan como base para otras clases que heredan de ellas. Las clases abstractas pueden contener métodos abstractos, que deben ser implementados por las clases hijas. Las clases

abstractas son útiles para definir comportamientos comunes y establecer una estructura jerárquica en tu código.

5. *Genéricos (Generics):* los genéricos son una característica poderosa de TypeScript que permiten crear componentes y funciones que pueden trabajar con varios tipos de datos de manera flexible. Los genéricos te permiten parametrizar tipos y reutilizar código de manera más generalizada.
6. *Módulos y espacios de nombres (Namespaces):* TypeScript proporciona la capacidad de organizar tu código en módulos y espacios de nombres. Los módulos te permiten estructurar tu código en archivos separados y exportar/importar funcionalidades entre ellos, mientras que los espacios de nombres (namespaces) te permiten agrupar lógicamente clases, interfaces y funciones relacionadas bajo un mismo nombre.

Estos son solo algunos de los conceptos básicos particulares que TypeScript soporta. El lenguaje tiene muchas otras características y funcionalidades que se pueden explorar para desarrollar aplicaciones más robustas y escalables. Es recomendable consultar la documentación oficial de TypeScript para obtener más detalles y ejemplos sobre estas características.

Para cerrar esta sección, mencionaremos algunas características avanzadas y únicas de Typescript.

Algunas son las siguientes:

1. *Tipos condicionales (Conditional Types):* TypeScript ofrece tipos condicionales que permiten realizar inferencias y transformaciones de tipos basadas en condiciones. Estos tipos condicionales son especialmente útiles en casos complejos donde se necesita inferir o transformar tipos en función de ciertas condiciones lógicas. Esto brinda una mayor flexibilidad y capacidad de expresión en el sistema de tipos de TypeScript.
2. *Tipos mapped (Mapped Types):* Los tipos mapeados en TypeScript permiten crear nuevos tipos basados en la transformación de propiedades existentes en otro tipo. Esto es útil cuando se desea generar un nuevo tipo con propiedades modificadas o adicionales basadas en un tipo existente. Se puede utilizar los tipos mapped para crear tipos inmutables, realizar operaciones de filtro o incluso crear tipos de acciones para manejar el estado de la aplicación de manera más segura.
3. *Tipos literales y discriminantes de unión (Literal Types and Union Discriminants):* TypeScript permite trabajar con tipos literales, lo que significa que se puede especificar un conjunto específico de valores para una variable o una propiedad. Esto es útil cuando se desea limitar los valores que se pueden

asignar a una variable o utilizar tipos literales en discriminantes de unión para crear tipos que actúen como una especie de enumeración.

4. *Comprobación exhaustiva de patrones (Exhaustive Pattern Checking):* TypeScript ofrece una comprobación exhaustiva de patrones en los bloques switch para asegurarse de que todos los casos posibles estén cubiertos y evitar omisiones accidentales. Si falta algún caso en el bloque switch, TypeScript emitirá un error de compilación para informarte sobre la falta de cobertura.
5. *Declaraciones de espacios de nombres ambientales (Ambient Namespace Declarations):* TypeScript permite declarar espacios de nombres ambientales que permiten extender tipos y funcionalidades de bibliotecas o módulos existentes. Esto es especialmente útil cuando se está trabajando con bibliotecas JavaScript que no tienen declaraciones de tipos nativas o cuando deseas agregar funcionalidades adicionales a una biblioteca existente.
6. *Modificadores de acceso más estrictos:* TypeScript ofrece modificadores de acceso más estrictos en comparación con JavaScript. Además de los modificadores 'public', 'private' y 'protected', TypeScript agrega el modificador 'readonly', que permite declarar propiedades de solo lectura. Esto promueve la inmutabilidad y ayuda a prevenir cambios accidentales en las propiedades.

Estas son algunas de las características avanzadas y únicas de TypeScript que van más allá de las funcionalidades básicas de otros lenguajes. Estas características permiten un mayor control sobre el sistema de tipos, la capacidad de expresión y la seguridad en el desarrollo de aplicaciones TypeScript.

Comparación con Oz

Antes de terminar, una comparación interesante entre **Oz** y TypeScript es el soporte que ofrecen para la programación lógica y la programación orientada a objetos, respectivamente.

Por un lado, Oz:

- Oz es un lenguaje que brinda un sólido soporte para la programación lógica y la programación concurrente.
- En la programación lógica, Oz utiliza el paradigma de la programación por restricciones, donde los problemas se modelan en términos de restricciones lógicas y se resuelven mediante un proceso de búsqueda y unificación.
- Oz proporciona primitivas como la *concordancia de patrones*, los *procedimientos* (procedures) y los *actores* (actors) para expresar lógica y concurrencia de manera declarativa y concurrente.

- También ofrece características como la reactividad, la propagación de restricciones y la coordinación de actores para modelar problemas complejos y sistemas concurrentes.

Por otro lado, TypeScript:

- TypeScript es un lenguaje que se basa en JavaScript y ofrece un conjunto de características adicionales, incluido un sistema de tipos estático opcional.
- El enfoque principal de TypeScript es la programación orientada a objetos y el desarrollo de aplicaciones robustas y mantenibles.
- TypeScript proporciona soporte para clases, herencia, interfaces, polimorfismo y encapsulación, siguiendo los principios de la programación orientada a objetos.
- Además, TypeScript ofrece el sistema de tipos estático que permite detectar errores en tiempo de compilación y proporciona autocompletado, refactorización y verificación de tipos durante el desarrollo.
- TypeScript se compila a JavaScript, lo que permite ejecutar el código en cualquier entorno compatible con JavaScript.

En resumen, mientras Oz se enfoca en la programación lógica y la programación concurrente, TypeScript se centra en la programación orientada a objetos y el desarrollo de aplicaciones robustas. Ambos lenguajes brindan características distintivas y se adaptan a diferentes tipos de problemas y paradigmas de programación.

6. Estadísticas

Para entender la importancia de TypeScript y la popularidad que obtuvo en estos últimos años, podemos observar las estadísticas de su uso y evolución que demuestran la ganancia de popularidad que obtuvo este lenguaje.

Algunas estadísticas de su popularidad y adopción:

- TypeScript ha experimentado un crecimiento significativo en popularidad en los últimos años. Según la encuesta de *Stack Overflow Developer Survey 2021*, TypeScript se ubicó como el tercer lenguaje de programación más amado y el cuarto lenguaje más popular entre los desarrolladores.
- Pero, es importante mencionar que, en la edición del 2022, TypeScript se ubicó como quinto lenguaje más utilizado. Mientras que JavaScript fue el lenguaje más usado por décimo año consecutivo.
- De acuerdo a esta misma encuesta, es decir la *Stack Overflow Developer Survey 2022*, TypeScript salió cuarto en el lenguaje más amado (aquellos que los desarrolladores usan y quieren seguir usando) y tercero en el lenguaje más querido (aquellos que no usan, pero les gustaría tener ocasión de usar).

- Lo importante es que, en 2022, TypeScript completó los cincos lenguajes de programación más utilizados en todo el mundo.
- En el *Índice de Popularidad de Lenguajes de TIOBE* de junio de 2021, TypeScript ocupó el puesto número 15 entre los lenguajes de programación más populares.
- En el *índice de clasificación Redmonk* de junio de 2021, TypeScript se ubicó en el puesto 12 entre los lenguajes de programación más populares, basado en la cantidad de preguntas etiquetadas en Stack Overflow y la cantidad de proyectos etiquetados en GitHub.
- Javascript y Typescript dominan el mercado laboral de los desarrolladores en estos momentos, y representan un total del 31% de las ofertas de empleo que requerían explícitamente un lenguaje de programación. Eso significa que casi 1 de cada 3 ofertas de trabajo requiere conocimientos de Javascript o Typescript.
- En general, el lenguaje de programación más utilizado por los desarrolladores de software de todo el mundo en los últimos 12 meses es Javascript. Además, muchos de los desarrolladores de software también tiene previsto adoptar o migrar a JavaScript. Si bien no es TypeScript, TypeScript es un superset de JavaScript. Esto significa que los programas de JavaScript son programas válidos de TypeScript, a pesar de que TypeScript sea otro lenguaje de programación.

Fuentes:

- Statista: “Most used programming languages among developers worldwide as of 2022”, citado en febrero de 2023. (Fuente)
- Statista: “Most wanted programming languages among developers worldwide, as of 2022”, citado en febrero de 2023. (Fuente)
- Statista: “Most demanded programming languages by recruiters worldwide in 2022”, citado en febrero de 2023. (Fuente)
- Halo Lab: “The Best Programming Languages In 2022, citado en febrero de 2023. (Fuente)
- Analytics Insight: “Top 10 Programming Languages Recruiters are Looking For in 2022”, citado en febrero de 2023. (Fuente)

Uso en proyectos y empresas:

- Grandes empresas y proyectos de renombre utilizan TypeScript. Algunas de ellas incluyen Microsoft, Google, Slack, Asana, Angular, NestJS, entre otras. Esto demuestra la confianza y la adopción del lenguaje en diversos sectores de la industria.

Frameworks y bibliotecas:

- *Angular*: Es uno de los frameworks web más populares y está completamente construido en TypeScript. Ofrece características poderosas para la creación de aplicaciones web escalables y mantenibles.

- *React:* Aunque React es principalmente un framework de JavaScript, también es ampliamente utilizado con TypeScript. TypeScript brinda beneficios adicionales en términos de verificación de tipos y desarrollo seguro.
- *Node.js:* TypeScript se utiliza cada vez más en el desarrollo de aplicaciones del lado del servidor con Node.js. La combinación de TypeScript y Node.js permite compartir código entre el lado del cliente y el lado del servidor, lo que mejora la eficiencia del desarrollo.
- Según la encuesta de *Stack Overflow Developer Survey 2022*, respecto a los frameworks web, Node.js y React.js han sido, con diferencia, las dos tecnologías web más utilizadas.

Evolución:
- TypeScript ha tenido varias versiones importantes desde su lanzamiento inicial. Cada versión trae nuevas características, mejoras de rendimiento y correcciones de errores. Es recomendable mantenerse actualizado con las últimas versiones para aprovechar al máximo las mejoras y las nuevas funcionalidades del lenguaje.

7. Comparación con otros lenguajes similares

A continuación, realizaremos una comparación entre TypeScript y algunos lenguajes similares, destacando las diferencias y características distintivas de TypeScript en relación con estos lenguajes.

1. Javascript

TypeScript es un superset de JavaScript, lo que significa que todo el código JavaScript válido también es código TypeScript válido. Sin embargo, la principal diferencia radica en la capacidad de TypeScript para agregar tipado estático opcional. A diferencia de JavaScript, donde el tipo de una variable se infiere en tiempo de ejecución, en TypeScript se pueden agregar anotaciones de tipo que permiten detectar errores en tiempo de compilación y brindan un mayor nivel de seguridad y confiabilidad. Esto hace que TypeScript sea más adecuado para proyectos de gran escala y equipos de desarrollo colaborativos.

Ejemplo de diferencia*:*

JavaScript:
```
function sum(a, b) {
  return a + b;
```

```
}
```

TypeScript:

```
function sum(a: number, b: number): number {
  return a + b;
}
```

En este ejemplo, TypeScript especifica los tipos de los parámetros y el valor de retorno de la función 'sum', lo que permite detectar posibles errores de tipos durante la compilación.

2. Flow

Flow es otro lenguaje que proporciona tipado estático opcional para JavaScript. Al igual que TypeScript, Flow permite agregar anotaciones de tipo para verificar y analizar el código en busca de errores de tipos. Sin embargo, hay algunas diferencias entre TypeScript y Flow. TypeScript tiene una adopción más amplia y una comunidad más grande, lo que ha llevado a un ecosistema de herramientas y bibliotecas más maduro. Además, TypeScript tiene una integración más profunda con herramientas populares como Visual Studio Code y proporciona una mayor compatibilidad con proyectos Angular.

En términos de rendimiento y benchmarks, la comparación entre TypeScript y Flow puede variar dependiendo del proyecto y la configuración. En general, ambos ofrecen beneficios similares en cuanto a la detección de errores de tipos en tiempo de compilación.

3. Babel

A diferencia de TypeScript y Flow, Babel no es un lenguaje en sí mismo, sino un transpilador de JavaScript. Babel permite utilizar características de JavaScript que aún no son ampliamente compatibles en los navegadores actuales, al transpilar el código a una versión compatible. Si bien Babel no agrega tipado estático opcional como TypeScript y Flow, puede combinarse con estos para obtener las ventajas tanto del tipado estático como de las características modernas de JavaScript.

En cuanto a los benchmarks, es importante tener en cuenta que el rendimiento de un lenguaje de programación puede variar según el contexto y las implementaciones específicas. Los benchmarks y las comparaciones de rendimiento generalmente se enfocan en casos específicos y pueden diferir dependiendo del tipo de proyecto y las optimizaciones aplicadas.

4. ¿Cuál utilizar?

Es importante tener en cuenta que la elección entre TypeScript, Flow y Babel depende del contexto y los requisitos del proyecto.

- TypeScript es ampliamente utilizado en proyectos de gran escala y aplicaciones empresariales.
- Mientras que Flow se utiliza principalmente en proyectos de código abierto y en el ecosistema React.
- Babel se utiliza para transpilar código JavaScript moderno en una versión compatible con navegadores más antiguos.

La elección entre estos lenguajes y herramientas debe basarse en las necesidades y preferencias del proyecto, el equipo de desarrollo y el ecosistema de herramientas existente.

Comparación con Oz

Si bien **Oz** no es un lenguaje tan similar a TypeScript, por las diferencias que fuimos mencionando durante el informe, nos pareció interesante hacer una última comparación entre estos dos lenguajes.
Previamente, en la sección 4 y 5 del informe, mencionamos algunas características de TypeScript y distinguimos algunos elementos fundamentales de la programación como el manejo de la concurrencia o de la memoria explicando cómo la manejaba nuestro lenguaje de programación elegido.
Sin embargo, nos gustaría mencionar otros temas no vistos en estas secciones para terminar de cerrar esta comparación entre ambos lenguajes.

Popularidad y uso

- Oz: es menos conocido y utilizado, en comparación con otros lenguajes más populares. Se utiliza principalmente en entornos académicos y de investigación.
- TypeScript: ha ganado una gran popularidad en los últimos años, especialmente en el desarrollo web y de aplicaciones. Es ampliamente utilizado en proyectos de gran escala y tiene una gran comunidad de desarrolladores.

Integración con otras tecnologías

- Oz: la integración de Oz con otras tecnologías puede ser limitada debido a su enfoque más académico. Las bibliotecas y frameworks disponibles pueden ser menos numerosos en comparación con otros lenguajes más populares.
- TypeScript: se integra fácilmente con el ecosistema de JavaScript, lo que le brinda acceso a una amplia gama de bibliotecas y frameworks populares. También, es

ampliamente utilizado en combinación con tecnologías web como Angular y Node.js.

En resumen, según lo que estuvimos comparando durante el informe, podemos afirmar que **Oz y TypeScript son lenguajes muy diferentes** en términos de paradigmas de programación, tipado y enfoque. Oz está más orientado hacia la programación concurrente y se utiliza principalmente en entornos académicos, mientras que TypeScript es un lenguaje estáticamente tipado utilizado ampliamente en el desarrollo web y de aplicaciones. La elección entre ellos dependerá de las necesidades específicas y el contexto de desarrollo.

8. Casos de estudio

Ahora presentaremos algunos casos de estudio de proyectos reales en los que se ha utilizado TypeScript y los motivos por los que se eligió este lenguaje.

1. *Angular:* Angular es un popular framework de desarrollo de aplicaciones web que está construido completamente en TypeScript. Se eligió TypeScript debido a su capacidad de agregar tipado estático, lo que ayuda a detectar errores en tiempo de compilación y mejorar la productividad en proyectos grandes y complejos. El uso de TypeScript en Angular también facilita la construcción de aplicaciones escalables y mantenibles.
2. *Microsoft Office*: Microsoft ha utilizado TypeScript en varios proyectos de la suite de Microsoft Office, incluyendo aplicaciones como Outlook Web App (OWA) y SharePoint. El uso de TypeScript en estos proyectos ha permitido una mayor seguridad y confiabilidad del código, así como un desarrollo más ágil y colaborativo debido a las ventajas del tipado estático y las herramientas de autocompletado proporcionadas por TypeScript.
3. *Asana*: Asana, una popular herramienta de gestión de proyectos y tareas, utiliza TypeScript en su base de código. La elección de TypeScript se basó en la necesidad de desarrollar una aplicación web compleja y escalable, y aprovechar los beneficios del tipado estático para garantizar la calidad del código y reducir errores.
4. *Slack:* Slack, una plataforma de comunicación empresarial, también ha adoptado TypeScript en su desarrollo. El uso de TypeScript en Slack se ha centrado en mejorar la calidad y la mantenibilidad del código, así como en facilitar el trabajo en equipo y el desarrollo a gran escala.
5. *Jest:* Jest, un framework de pruebas unitarias para JavaScript, está escrito en TypeScript. La decisión de utilizar TypeScript en Jest se basó en la necesidad de

un código más robusto y fácil de mantener, así como en la capacidad de aprovechar el sistema de tipos de TypeScript para mejorar la calidad de las pruebas y detectar errores más rápidamente.

Estos son solo algunos ejemplos de casos reales en los que se ha utilizado TypeScript en proyectos de diferentes ámbitos. En general, las razones principales para elegir TypeScript incluyen la mejora de la seguridad y la calidad del código, la productividad en proyectos grandes y complejos, la facilidad de mantenimiento y la capacidad de aprovechar las herramientas y el ecosistema de TypeScript para desarrollar aplicaciones web escalables y colaborativas.

9. Conclusión

En conclusión, TypeScript es un lenguaje de programación poderoso y versátil que ofrece varias características y beneficios significativos para el desarrollo de aplicaciones web y empresariales.

La principal ventaja de TypeScript, como bien indica su nombre, es la posibilidad del tipado estático opcional. TypeScript permite agregar anotaciones de tipo estáticas, lo que mejora la seguridad del código y permite detectar errores de tipos durante la fase de compilación. Esto resulta en un código más robusto y confiable.

Sin embargo, como desarrollamos a lo largo de la investigación, Typescript es un lenguaje ampliamente utilizado porque ofrece muchas más ventajas. Por ejemplo, mejora la productividad. El tipado estático, junto con las herramientas de autocompletado y verificación de tipos, facilita el desarrollo y reduce los errores. Además, TypeScript ofrece características avanzadas como inferencia de tipos, funciones de orden superior y soporte para ECMAScript, lo que permite escribir código más conciso y expresivo.

Además, TypeScript proporciona características orientadas a objetos como clases, herencia e interfaces, lo que facilita la construcción de aplicaciones escalables y fáciles de mantener. También, ofrece soporte para módulos y namespaces, permitiendo una organización estructurada del código.

Más allá de las características técnicas, otra gran ventaja es su amplio ecosistema. TypeScript cuenta con una comunidad activa y un ecosistema maduro. Existen numerosas bibliotecas, frameworks y herramientas que admiten TypeScript, lo que facilita el desarrollo de aplicaciones y la integración con otras tecnologías.

Además, es importante mencionar la compatibilidad con Javascript. Como TypeScript es un superset de JavaScript, todo el código JavaScript existente es válido en TypeScript. Esto facilita la migración gradual de proyectos existentes a TypeScript y

permite a los desarrolladores aprovechar las ventajas de TypeScript en aplicaciones JavaScript existentes.

En general, TypeScript combina la flexibilidad y familiaridad de JavaScript con la seguridad y confiabilidad del tipado estático opcional. Esto lo convierte en una opción popular para el desarrollo de aplicaciones web y empresariales, especialmente en proyectos de gran escala y equipos de desarrollo colaborativos. Si se está buscando mejorar la calidad del código, aumentar la productividad y construir aplicaciones más robustas y mantenibles, TypeScript es una excelente elección.