Typescript

استاد درس: دکتر آرش شفیعی

پدید آورندگان: محمدامین صابری - ۹۹۳۶۲۳۰۲۶، ابوالفضل شیشه‌گر - ۹۹۳۶۲۳۰۲۵

دانشکده مهندسی کامپیوتر

دانشگاه اصفهان

پاییز ۱۴۰۳

فهرست عناوین

مقدمه

زبان برنامه‌نویسی TypeScript یک زبان رایگان و منبع باز است که توسط شرکت مایکروسافت توسعه داده شده و امکان اضافه کردن نوع‌های استاتیک با انوتیشن‌های اختیاری به JavaScript را فراهم می‌کند .این زبان برای توسعه برنامه‌های بزرگ طراحی شده و به JavaScript ترنسپایل می‌شود.

TypeScriptدر سال ۲۰۱۲ برای اولین بار به صورت عمومی معرفی شد، پس از دو سال توسعه داخلی در مایکروسافت اندرس هیلسبرگ، معمار اصلی #C و خالق Delphi وTurbo Pascal ، در توسعه TypeScript کار کرده است.

TypeScriptیک سوپرست از JavaScript است، یعنی همه برنامه‌های JavaScript از نظر نحوی TypeScript معتبر هستند، اما ممکن است به دلایل ایمنی در بررسی نوع شکست بخورند. TypeScript می‌تواند برای توسعه برنامه‌های JavaScript برای اجرای سمت کلاینت و سرور (مانند Node.js یا Deno) استفاده شود. گزینه‌های متعددی برای ترنسپایل وجود دارد .کامپایلر پیش‌فرض TypeScript می‌تواند استفاده شود، یا کامپایلر Babel می‌تواند برای تبدیل TypeScript به JavaScript فراخوانی شود.

TypeScriptاز فایل‌های تعریف پشتیبانی می‌کند که می‌توانند اطلاعات نوع مربوط به کتابخانه‌های JavaScript موجود را شامل شوند، مانند فایل‌های هدر C++ که می‌توانند ساختار فایل‌های شیء موجود را توصیف کنند. این امکان را می‌دهد که برنامه‌های دیگر از مقادیر تعریف شده در فایل‌ها استفاده کنند، انگار که موجودیت‌های TypeScript استاتیک باشند. فایل‌های هدر ثالثی برای کتابخانه‌های محبوب مانندjQuery ، MongoDBو D۳.js وجود دارند .هدرهای TypeScript برای ماژول‌های کتابخانه Node.jsنیز در دسترس هستند، که امکان توسعه برنامه‌های Node.js در TypeScript را فراهم می‌کنند.

کامپایلر TypeScript خودش به زبان TypeScript نوشته شده و به JavaScript ترنسپایل می‌شود .این کامپایلر تحت مجوز Apache License ۲,۰قرار دارد.

TypeScriptدر سال ۲۰۱۳، با انتشار نسخه ۰.۹، از جنریک‌ها پشتیبانی کرد نسخه ۱.۰ TypeScript در کنفرانس توسعه‌دهنده Buildمایکروسافت در سال ۲۰۱۴ منتشر شد Visual Studio ۲۰۱۳ Update ۲ از TypeScript به صورت داخلی پشتیبانی می‌کند.

بهبودهای بیشتری در ماه جولای ۲۰۱۴ انجام شد، زمانی که تیم توسعه اعلام کرد که یک کامپایلر جدید TypeScript را ارائه داده است، که ادعا می‌شود پنج برابر سریع‌تر از کامپایلر قبلی است .همزمان، کد منبع، که در ابتدا در CodePlex میزبانی می‌شد، به GitHub منتقل شد

TypeScriptهمواره با پیشنهادات ECMAScript فعلی و آینده هماهنگ بوده و برخی از ویژگی‌های آن به JavaScript اضافه شده‌اند .مثلاً، ویژگی‌هایی مانند کلاس‌ها، ماژول‌ها، توابع پیکانی، توابع مولد، توابع آسنکرون و انتظار، اپراتور اسپرد و رست و دیکوراتورها از TypeScript الهام گرفته شده‌اند.

TypeScriptهمچنین از ویژگی‌هایی پشتیبانی می‌کند که در JavaScript وجود ندارند، مانند انواع توپل، انواع متحد، انواع تقاطع، انواع لیترال، انواع متغیر، انواع متریک، انواع شرطی، انواع متریک شرطی، انواع متریک توزیع شده، انواع متریک توزیع شده شرطی، انواع متریک توزیع شده توپل، انواع متریک توزیع شده توپل شرطی. این انواع پیچیده به برنامه‌نویسان امکان می‌دهند که با دقت بیشتری نوع داده‌های خود را مشخص کنند و از خطاهای نوع در زمان اجرا جلوگیری کنند.

TypeScriptهمچنین از ویژگی‌هایی مانند ماژول‌هایES٦ ، دکوراتورهای متد و کلاس، انواع متغیر، انواع متریک، انواع شرطی، انواع تقاطع، انواع متحد، انواع توپل، انواع لیترال، انواع متریک شرطی، انواع متریک توزیع شده، انواع متریک توزیع شده شرطی، انواع متریک توزیع شده توپل، انواع متریک توزیع شده توپل شرطی پشتیبانی می‌کند.

TypeScriptدر سال‌های اخیر به یکی از زبان‌های برنامه‌نویسی محبوب برای توسعه وب تبدیل شده است. برخی از پروژه‌های معروفی که از TypeScript استفاده می‌کنند عبارتند ازAngular :، React، Vue، Nest، Ionic، Deno، RxJS، Next.js، Nuxt.js، Svelte، Gatsby، GraphQL، Jest، Cypressو بسیاری دیگر.

TypeScript در هر زمینه و حوزه‌ای که JavaScript کاربرد دارد، قابل استفاده است. TypeScript می‌تواند در سمت سرور با Node.js یا Deno، در سمت کاربر با مرورگرها، در برنامه‌های تحت وب با React، Angular، Vue و غیره، در برنامه های موبایل با React Native، Ionic، NativeScript و غیره، و در برنامه‌های رومیزی با Electron، NW.js و غیره استفاده شود. TypeScript به شما امکان می‌دهد تا برنامه‌های پویا، پاسخگو، قابل انعطاف و مقیاس پذیر را با JavaScript بسازید.

١. توسعه برنامه‌های وب
٢. توسعه برنامه‌های موبایل
٣. توسعه برنامه‌های دسکتاپ
٤. توسعه برنامه‌های بازی
٥. توسعه برنامه‌های IoT
٦. توسعه برنامه‌های کاربردی

در ابتدا، TypeScript به منظور افزایش کارایی و بهبود قابلیت‌های جاوا اسکریپت توسعه داده شد. اما با گذر زمان، این زبان برنامه‌نویسی به یکی از محبوب‌ترین زبان‌های برنامه‌نویسی تبدیل شده است.

این زبان برای رفع مشکلاتی مانند کاهش خطاها، افزایش قابلیت خواندن کد، افزایش سرعت توسعه، افزایش قابلیت نگهداری کد، افزایش قابلیت توسعه‌پذیری و افزایش قابلیت تست کد ابداع شده است. TypeScript به عنوان یک زبان تایپ شده، از قابلیت‌هایی مانند تعیین نوع داده‌ها، اعلان متغیرها، تعریف توابع، تعریف کلاس‌ها و ... پشتیبانی می‌کند. TypeScript بهبودهایی را برای JavaScript ایجاد کرده است.

TypeScript در مقایسه با جاوا اسکریپت، به دلیل داشتن تایپ ایستا، کدنویسی را سریع‌تر و آسان‌تر می‌کند و خطاهای نگارشی را کاهش می‌دهد. TypeScript همچنین از ویژگی‌هایی مانند کلاس‌ها و اینترفیس‌ها پشتیبانی می‌کند که کدنویسی را سازمان‌یافته‌تر می‌کند. در مقایسه با زبان‌های دیگری مانند C#، جاوا و...، TypeScript به دلیل داشتن تایپ ایستا، کدنویسی را سریع‌تر و آسان‌تر می‌کند. همچنین، TypeScript به شما امکان می‌دهد که به صورت خودکار مستندات برنامه‌ی خود را بسازید و درک بهتری از کد خود پیدا کنید.

این زبان برای توسعه‌ی برنامه‌های بزرگ و پیچیده مناسب است. برخی از ویژگی‌های TypeScript که آن را از زبان های مشابه خود متمایز می‌کند عبارتند از :

۱. استاتیک تایپینگ
۲. کلاس‌ها و اینترفیس‌ها
۳. سازگاری با کتابخانه‌های جاوااسکریپت
٤. پشتیبانی از ماژول‌ها
٥. پشتیبانی از ژنریک‌ها

به علاوه، TypeScript به شما امکان مدیریت بهتر کد را با استفاده از ویژگی‌های پیشرفته‌ی کامپایلر خود در حین کدنویسی می‌دهد.

۱. خوانایی:

TypeScript از ویژگی‌های مفیدی برای خوانایی کد برخوردار است. سینتکس TypeScript شبیه به JavaScript است و از ویژگی‌هایی مانند نوع‌دهی استفاده می‌کند که می‌تواند خوانایی و درک کد را بهبود بخشد. همچنین TypeScript از ویژگی‌هایی مانند تایپ‌های اختیاری و افزایش دهنده خوانایی کد برخوردار است.

۲. قابلیت اطمینان:

استفاده از TypeScript می‌تواند قابلیت اطمینان کد را افزایش دهد. با تایپ‌های استفاده شده، اشتباهات و ایرادات برخی از کدها را قبل از اجرا تشخیص می‌دهد. این کمک می‌کند که خطاهای مربوط به نوع داده‌ها کمتر شود و کد قابل اطمینان‌تری ارائه شود.

۳. هزینه (کارایی و بهره وری):

در مقایسه با JavaScript معمولی، استفاده از TypeScript ممکن است هزینه اضافی‌ای در زمان توسعه داشته باشد. اما این هزینه به دلیل کاهش خطاها، توسعه سریع‌تر و مدیریت بهتر کد، می‌تواند بهبود بهره وری بخشد. این موضوع به میزان پیچیدگی و اندازه پروژه نیز بستگی دارد.

٤. هزینه مورد نیاز برای یادگیری و برنامه‌نویسی:

یادگیری TypeScript برای برنامه‌نویسانی که با JavaScript آشنایی دارند، نسبتاً آسان است. با این حال، برای فهم کامل از ویژگی‌های TypeScript و استفاده بهینه از آن، ممکن است زمانی نیاز باشد. منابع آموزشی و ابزارهای متعددی برای یادگیری TypeScript وجود دارند.

٥. قابلیت جابجایی:

TypeScript به کد JavaScript ترجمه می‌شود، بنابراین کدهای تایپ‌شده TypeScript می‌توانند به سادگی به کدهای JavaScript تبدیل شوند. این قابلیت جابجایی باعث می‌شود که توسعه‌دهندگان بتوانند کد JavaScript خود را به تدریج به TypeScript منتقل کنند و یا از کد TypeScript به عنوان جایگزین برای کدهای JavaScript استفاده کنند.

به طور کلی، ارزیابی TypeScript باید با توجه به نیازها و معیارهای هر پروژه و تیم برنامه‌ریزی شود. استفاده از TypeScript می‌تواند به بهبود کیفیت کد و بهره وری کمک کند، اما هزینه‌هایی نظیر زمان و تلاش برای یادگیری و استفاده از آن نیز وجود دارد که باید در نظر گرفته شود.

TypeScript از یک کامپایلر استفاده می کند که TypeScript را به JavaScript تبدیل می‌کند. این کامپایلر توسط تیم TypeScript در شرکت مایکروسافت توسعه داده شده است. این کامپایلر قابلیت‌های زیر را دارد:

پشتیبانی از همه نسخه های TypeScript .JavaScript می‌تواند به هر نسخه ای از JavaScript که شما می‌خواهید تبدیل شود. برای مثال، شما می توانید TypeScript را به ES٥، ES٦، ES٢٠١٥ و غیره تبدیل کنید.

پشتیبانی از سورس مپ. TypeScript می‌تواند فایل‌های سورس مپ را برای خطایابی راحت تر در ویرایشگر یا مرورگر ایجاد کند. سورس مپ فایل‌هایی هستند که نشان می‌دهند که هر خط TypeScript به چه خط JavaScript تبدیل شده است.

پشتیبانی از TypeScript .JSX می‌تواند JSX را به JavaScript تبدیل کند. JSX یک سینتکس شبه HTML است که برای ساخت رابط کاربری با React و برخی دیگر از کتابخانه‌ها استفاده می‌شود.

کامپایلرهایی که برای این زبان وجود دارند عبارتند از:

۱. TypeScript Compiler: کامپایلر رسمی Typescript توسط تیم Typescript توسعه داده شده است. از مزایای استفاده از این کامپایلر می‌توان به پشتیبانی از جدیدترین ویژگی‌های Typescript، سرعت بالا، امکان استفاده در پروژه‌های بزرگ و قابلیت انتخاب میزان خطاهایی که در هنگام کامپایل اعلام می‌شود، اشاره کرد.

۲. Babel: یک کامپایلر جاوااسکریپت است که از Typescript نیز پشتیبانی می‌کند. می‌توانید از آن برای کامپایل کردن کدهای Typescript خود استفاده کنید. Babel توسط James Kyle و Sebastian McKenzie توسعه داده شده است. از مزایای استفاده از Babel می‌توان به پشتیبانی از جدیدترین ویژگی‌های جاوااسکریپت، امکان استفاده در پروژه‌های بزرگ، امکان استفاده در محیط‌های مختلف و امکان تبدیل کدهای Typescript به کدهای ES۵ اشاره کرد.

۳. SWC: یک کامپایلر جاوااسکریپت و Typescript است که با سرعت بالا و کد منبع باز عرضه می‌شود. SWC توسط یک تیم توسعه‌دهنده با کد منبع باز توسعه داده شده است. از مزایای استفاده از SWC می‌توان به سرعت بالا، پشتیبانی از جدیدترین ویژگی‌های Typescript، پشتیبانی از کامپایل کردن کدهای جاوااسکریپت و Typescript به صورت همزمان و امکان استفاده در پروژه‌های بزرگ اشاره کرد.

نحو و معناشناسی

در اینجا فهرستی از کلمات کلیدی زبان برنامه نویسی TypeScript آورده شده است.

زبان برنامه نویسی TypeScript یک زبان برنامه نویسی تایپ شده است که از افزونه‌های استاندارد جاوااسکریپت بهره می‌برد. در زیر یک فهرست از کلمات کلیدی TypeScript همراه با توضیحات کوتاه و مثال‌ها آمده است:

۱. Interface (رابط) :

- توضیح: رابط‌ها یک مجموعه از تعریف‌های نوع برای اشیاء در TypeScript هستند.
- مثال:

```
interface Person{
 name: string;
 age: number;
}

function greet(person: Person): string {
 return `Hello, ${person.name}!`;
}
```

۲. Type (نوع) :

- توضیح: از `type` برای ایجاد یک نوع جدید استفاده می‌شود.
- مثال:

```
type Point = {
 x: number;
 y: number;
};
function getDistance(point1: Point, point2: Point): number {
```

اجرای الگوریتم محاسبه فاصله//

```
 return Math.sqrt((point2.x - point1.x)**2 + (point2.y - point1.y)**2);}
```

۳. Class (کلاس) :

- توضیح: از کلاس‌ها برای تعریف شیء ویژه‌ای با استفاده از ویژگی‌ها و روش‌ها استفاده می‌شود.
- مثال:

```
class Animal {
 name: string;

 constructor(name: string)}
  this.name = name;
   }
 makeSound(): void{
  console.log("Some generic sound");
   }
}
const myPet = new Animal("Fluffy");
myPet.makeSound();
```

۴. Generics (ژنریک) :

- توضیح: ژنریک‌ها این امکان را به برنامه نویسان می‌دهند که توعیت دینامیکی در توابع و کلاس‌ها اعمال کنند.
- مثال:

```
function identity<T>(arg: T): T {
 return arg;
}
let output = identity<string>("Hello");
```

۵. Enum (شماره‌گان) :

- توضیح: Enumها مجموعه‌ای از نمادهای نامی هستند که مقادیر عددی به آنها اختصاص می‌دهند.
- مثال:

```
enum Color{
 Red,
 Green,
 Blue,
}

let myColor: Color = Color.Green;
```

۶. Decorator (تزئینگی) :

- توضیح: تزئینگی‌ها کد را با استفاده از اطلاعات در زمان اجرا تغییر می‌دهند و معمولاً برای افزودن ویژگی‌های ویژه به کلاس‌ها یا توابع استفاده می‌شوند.
- مثال:

```
function log(target: any, key: string) {
 console.log(`${key} was called`);
}

class MyClass {
 @log
 myMethod(){
   //some logic
  }
}
```

۷. Module (ماژول) :

- توضیح: ماژول‌ها به برنامه‌نویسان اجازه می‌دهند که کد را به بخش‌های جداگانه تقسیم کنند و از آنها در جاهای دیگر استفاده کنند.
- مثال:

در فایل //math.ts

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

در فایل //main.ts

```
import { add } from "./math;"
let result = add(3, 5);
```

۸. Namespace (فضای نام) :

- توضیح: Namespaceها به برنامه‌نویسان امکان ارائه یک فضای نام برای تعداد زیادی از نمادها را می‌دهند تا از تداخل نام جلوگیری شود.
- مثال:

در فایل // shapes.ts

```
namespace Shapes {
 export class Circle {/*…*/}
 export class Square {/*…*/}
}
```

در فایل // main.ts

```
let myCircle = new Shapes.Circle();
```

۹. Async/Await :

- ❖ توضیح: کلیدواژه‌های `async` و `await` برای مدیریت عملیات ناهمگام و انتظار برای یک نتیجه استفاده می‌شوند.
- ❖ مثال:

```
async function fetchData() {
 let response = await fetch('https://api.example.com/data');
 let data = await response.json();
 return data;
}
```

۱۰. Union Types (انواع اتحادی) :

- ❖ توضیح: این اجازه را می‌دهد که یک متغیر یا پارامتر با چند نوع مختلف تعریف شود.
- ❖ مثال:

```
function printId(id: number | string): void {
 console.log(`ID is: ${id}`);
}

printId(123);
printId("ABC");
```

برای نوشتن گرامر برای زیرمجموعهTypeScript ، می‌توانیم قواعدهای BNF را به صورت زیر تعریف کنیم :

```
"Program" -> "StatementList"
"StatementList" -> "Statement"
 "StatementList" -> "Statement StatementList"
 "Statement" -> "VariableDeclaration"
 "Statement" -> "IfStatement"
```

"Statement" -> "WhileStatement"

"Statement" -> "FunctionDeclaration"

"Statement" -> "ExpressionStatement"

"Statement" -> "EqualityOperator"

"Statement" -> "RelationalOperator"

"Statement" -> "AdditiveOperator"

"Statement" -> "MultiplicativeOperator"

"VariableDeclaration" -> "let Identifier = Expression"

"IfStatement" -> "if (Expression) { StatementList }"

"IfStatement" -> "if (Expression) { StatementList } else { StatementList }"

"WhileStatement" -> "while (Expression) { StatementList }"

"FunctionDeclaration" -> "function Identifier(ParameterList) { StatementList}"

"ParameterList" -> "ε"

"ParameterList" -> "IdentifierList"

"IdentifierList" -> "Identifier"

"IdentifierList" -> "Identifier IdentifierList"

"ExpressionStatement" -> "Expression"

"Expression" -> "AssignmentExpression"

"AssignmentExpression" -> "LogicalOrExpression"

"AssignmentExpression" -> "LogicalOrExpression = AssignmentExpression"

"LogicalOrExpression" -> "LogicalAndExpression"

"LogicalOrExpression" -> "LogicalOrExpression || LogicalAndExpression"

"LogicalAndExpression" -> "EqualityExpression"

"LogicalAndExpression" -> "LogicalAndExpression && EqualityExpression"

"EqualityExpression" -> "RelationalExpression"

"EqualityExpression" -> "EqualityExpression EqualityOperator RelationalExpression"

"EqualityOperator" -> "=="

"EqualityOperator" -> "!="

"RelationalExpression" -> "AdditiveExpression"

"RelationalExpression" -> "RelationalExpression RelationalOperator AdditiveExpression"

"RelationalOperator" -> "<"

"RelationalOperator" -> "<="

"RelationalOperator" -> ">"

"RelationalOperator" -> ">="

"AdditiveExpression" -> "MultiplicativeExpression"

"AdditiveExpression" -> "AdditiveExpression AdditiveOperator MultiplicativeExpression"

"AdditiveOperator" -> "+"

"AdditiveOperator" -> "-"

"MultiplicativeExpression" -> "UnaryExpression"

"MultiplicativeExpression" -> "MultiplicativeExpression MultiplicativeOperator UnaryExpression"

"MultiplicativeOperator" -> "*"

"MultiplicativeOperator" -> "/"

"UnaryExpression" -> "PrimaryExpression"

"UnaryExpression" -> "! UnaryExpression"

"UnaryExpression" -> "- UnaryExpression"

"PrimaryExpression" -> "Literal"

"PrimaryExpression" -> "(AssignmentExpression)"

"PrimaryExpression" -> "Identifier"

"PrimaryExpression" -> "FunctionCall"

"Literal" -> "BooleanLiteral"

"Literal" -> "NumericLiteral"

"Literal" -> "StringLiteral"

"BooleanLiteral" -> "true"

"BooleanLiteral" -> "false"

"NumericLiteral" -> "[۰-۹]+"

"StringLiteral" -> "\"[^\"]*\""

"StringLiteral" -> "'[^']*'"

"Identifier" -> "[a-zA-Z_][a-zA-Z۰-۹_]*"

"FunctionCall" -> "Identifier(ArgumentList)"

"ArgumentList" -> "ε"

"ArgumentList" -> "AssignmentExpression"

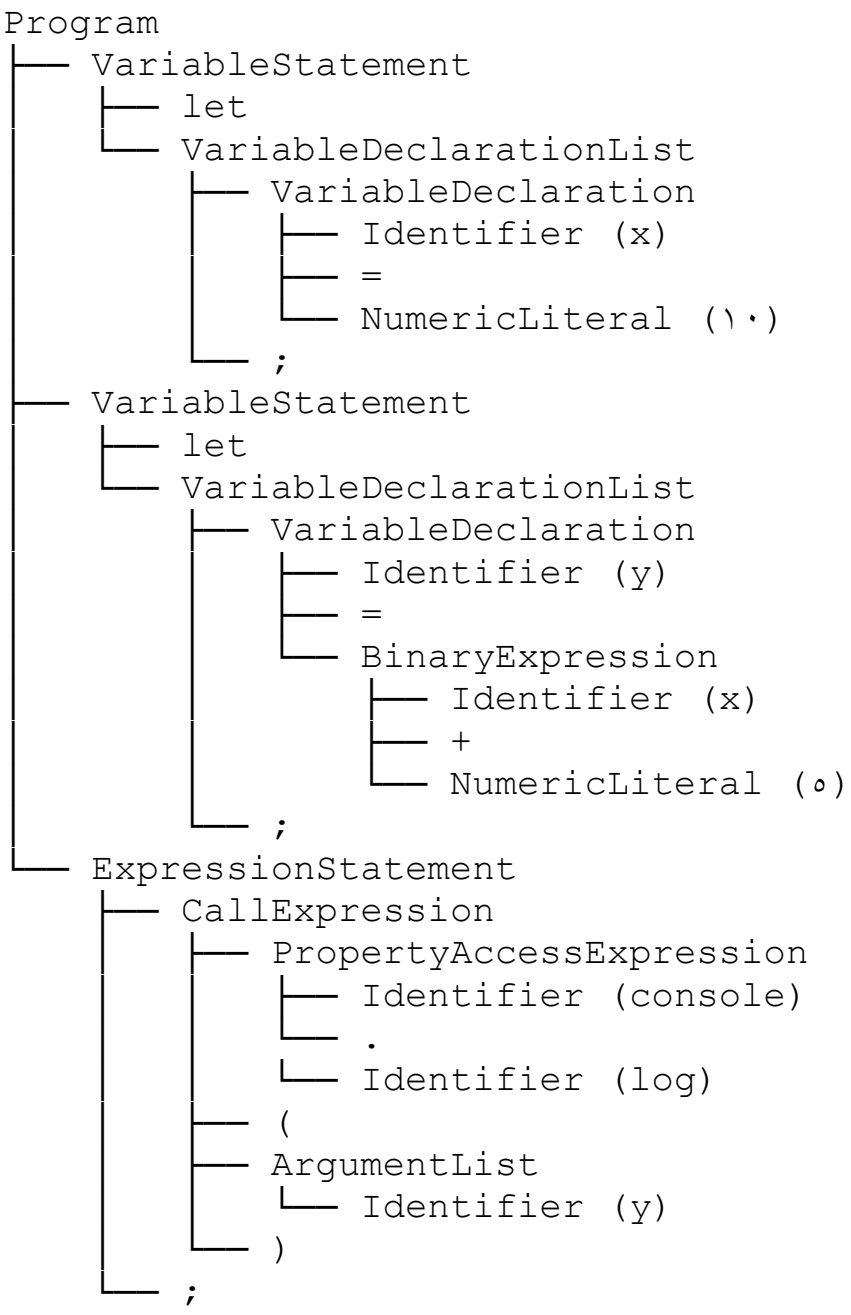
"ArgumentList" -> "AssignmentExpression ArgumentList"

این گرامر برای زیرمجموعه TypeScript فقط یک مثال است و ممکن است ناقص یا نادقیق باشد .

- **برنامه و درخت تجزیه .TypeScript** یک برنامه TypeScript یک مجموعه از جملات است که هر جمله یک عمل خاص را انجام می دهد. برای تجزیه و تحلیل ساختار و منطق یک برنامه TypeScript ، معمولا از یک درخت تجزیه استفاده می شود .درخت تجزیه یک ساختار درختی است که رئوس آن عبارتند از عبارات و عملگرهای زبان و برگ های آن عبارتند از شناسه ها و مقادیر ثابت. برای مثال ، برای برنامه TypeScript زیر:

```
let x = 10;
let y = x + 5;
console.log(y);
```

درخت تجزیه به صورت زیر خواهد بود:



معناشناسی عملیاتی به توصیف روند اجرای دستورات و عملیات در سطح پایین می‌پردازد. در اینجا، ساختارهای معناشناسی TypeScript به زبان C (بسیار ساده شده) ترجمه می‌شوند:

۱. تعریف یک کلاس در TypeScript:

```
class Person {
    private name: string;
    private age: number;

    constructor(name: string, age: number) {
        this.name = name;
        this.age = age;
    }

    greet(): void {
        console.log(`Hello, my name is ${this.name} and I'm
${this.age} years old.`);
    }
}
```

ساده‌سازی به زبان C:

```
struct Person {
    char name[50];
    int age;
};

void Person_init(struct Person *this, char *name, int age) {
    strcpy(this->name, name);
    this->age = age;
}
```

```
void Person_greet(struct Person *this) {
    printf("Hello, my name is %s and I'm %d years old.\n", this->name, this->age);
}
```

۲. استفاده از شیء از یک کلاس در TypeScript:

```
const person1 = new Person("Alice", 30);
person1.greet();
```

ساده‌سازی به زبان C:

```
int main() {
    struct Person person1;
    Person_init(&person1, "Alice", 30);
    Person_greet(&person1);
    return 0;
}
```

این ترجمه‌ها تلاشی برای نمایش ساختارهای TypeScript در یک زبان سطح پایین‌تر (C) می‌باشند.

در این قسمت به ترسیم درخت گرامر نوشته و نحوه رسم این درخت می‌پردازیم :

برای تبدیل گرامر نوشته شده به درخت تجزیه و هم چنین تصویرسازی آن به ترتیب نیاز به کارکردن با دو کتابخانه به نام های "anytree" و "graphviz" داریم.

anytree یک کتابخانه در زبان برنامه‌نویسی Python است که برای کار با ساختارهای درختی استفاده می‌شود. این کتابخانه امکانات زیادی برای ایجاد و مدیریت درخت‌ها فراهم می‌کند و بسیار مناسب برای استفاده در حوزه‌هایی مانند پردازش زبان‌های طبیعی، تحلیل داده‌ها، و یادگیری ماشین است.

برای نصب آن میتوان از دستور pip install anytree استفاده کرد.

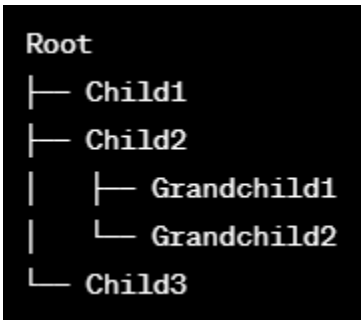
بیایید یک مثال ساده از ساختار درختی با **anytree** بررسی کنیم:

```
from anytree import Node, RenderTree

# ایجاد گره‌ها

root = Node("Root")

child ۱= Node("Child۱", parent=root)

child ۲= Node("Child۲", parent=root)

child ۳= Node("Child۳", parent=root)

grandchild ۱= Node("Grandchild۱", parent=child۲)

grandchild ۲= Node("Grandchild۲", parent=child۲)

# نمایش درخت

for pre, fill, node in RenderTree(root):

    print("%s%s" % (pre, node.name))
```

در این مثال:

- گره **Root** به عنوان گره اصلی ایجاد شده است.
- سه گره به نام‌های **Child۱**، **Child۲**، و **Child۳** به عنوان فرزندان گره اصلی اضافه شده‌اند.
- دو گره به نام‌های **Grandchild۱** و **Grandchild۲** به عنوان فرزندان گره **Child۲** ایجاد شده‌اند.

با استفاده از **RenderTree** می‌توانیم درخت را به صورت ساختاری نمایش دهیم:



در قطعه کدی که برای گرامر استفاده کردیم، در ابتدا قوانین گرامر را در لیستی ذخیره کردیم. این کد یک درخت نمایش دهنده‌ی ساختار گرامری برنامه‌های کامپایلر بر اساس یک گرامر نوشته شده به زبان EBNF (Extended Backus-Naur Form) ایجاد می‌کند. این گرامر به عنوان ورودی برای ایجاد درخت ساختاری از کتابخانه **anytree** استفاده می‌شود.

```
from anytree import Node, RenderTree
from anytree.exporter import DotExporter

# Define the grammar rules
grammar_rules = {
    "Program": ["StatementList"],
    "StatementList": ["Statement", "Statement ; StatementList"],
    "Statement": [
        "VariableDeclaration",
        "IfStatement",
        "WhileStatement",
        "FunctionDeclaration",
        "ExpressionStatement",
    ],
    "VariableDeclaration": ["let Identifier = Expression"],
    "IfStatement": [
        "if ( Expression ) { StatementList }",
```

```
    "if ( Expression ) { StatementList } else { StatementList }",
  ],
  "WhileStatement": ["while ( Expression ) { StatementList }"],
  "FunctionDeclaration": ["function Identifier ( ParameterList ) { StatementList }"],
  "ParameterList": ["ε", "IdentifierList"],
  "IdentifierList": ["Identifier", "Identifier, IdentifierList"],
  "ExpressionStatement": ["Expression"],
  "Expression": ["AssignmentExpression"],
        "AssignmentExpression": ["LogicalOrExpression", "LogicalOrExpression = AssignmentExpression"],
        "LogicalOrExpression": ["LogicalAndExpression", "LogicalOrExpression || LogicalAndExpression"],
       "LogicalAndExpression": ["EqualityExpression", "LogicalAndExpression && EqualityExpression"],
    "EqualityExpression": ["RelationalExpression", "EqualityExpression EqualityOperator RelationalExpression"],
  "EqualityOperator": ["==", "!="],
    "RelationalExpression": ["AdditiveExpression", "RelationalExpression RelationalOperator AdditiveExpression"],
  "RelationalOperator": ["<", "<=", ">", ">="],
    "AdditiveExpression": ["MultiplicativeExpression", "AdditiveExpression AdditiveOperator MultiplicativeExpression"],
  "AdditiveOperator": ["+", "-"],
        "MultiplicativeExpression": ["UnaryExpression", "MultiplicativeExpression MultiplicativeOperator UnaryExpression"],
  "MultiplicativeOperator": ["*", "/"],
    "UnaryExpression": ["PrimaryExpression", "! UnaryExpression", "UnaryOperator UnaryExpression"],
  "UnaryOperator": ["-", "+"],
```

```
    "PrimaryExpression": ["Literal", "( AssignmentExpression )", "Identifier", "FunctionCall"],
    "Literal": ["BooleanLiteral", "NumericLiteral", "StringLiteral"],
    "BooleanLiteral": ["true", "false"],
    "NumericLiteral": ["[٠-٩]+"],
    "StringLiteral": ['"[^"]*"', "'[^']*'"],
    "Identifier": ["[a-zA-Z_][a-zA-Z٠-٩_]*"],
    "FunctionCall": ["Identifier ( ArgumentList )"],
    "ArgumentList": ["ε", "AssignmentExpression", "AssignmentExpression, ArgumentList"],
}

# Create a tree based on the grammar
def create_tree(node, rule):
    for token in reversed(grammar_rules[rule]):
        child = Node(token, parent=node)
        if token in grammar_rules:
            create_tree(child, token)

# Create the root node and build the tree
root = Node("Program")
create_tree(root, "Program")

# Print the tree using anytree's RenderTree
for pre, fill, node in RenderTree(root):
    print(f"{pre}{node.name}")
```

در ادامه توضیحاتی در مورد کد آورده شده داریم:

۱. **تعریف گرامر:**

- یک دیکشنری به نام **grammar_rules** تعریف شده است که قوانین گرامر EBNF را برای برنامه‌های کامپایلر مشخص می‌کند.

۲. **ایجاد درخت:**

- تابع **create_tree** به عنوان ورودی یک گره و یک قاعده گرامر می‌گیرد و درخت را ایجاد می‌کند.
- این تابع از تعریف گرامر **grammar_rules** برای ساختارهای بازگشتی استفاده می‌کند و با استفاده از توابع **Node**از کتابخانه **anytree** گره‌ها را ایجاد می‌کند.

۳. **ساخت درخت اصلی:**

- یک گره اصلی به نام "Program" ایجاد شده و به عنوان ریشه درخت تعیین شده است.
- سپس تابع **create_tree** با ورودی "Program" فراخوانی می‌شود تا درخت را بسازد.

٤. **نمایش درخت:**

- با استفاده از تابع **RenderTree** از کتابخانه **anytree**، درخت نمایش داده می‌شود.

خروجی قطعه کد بالا به شرح زیر است (درخت گرامر):


Program
StatementList
Statement ; StatementList
Statement
ExpressionStatement
Expression
AssignmentExpression
LogicalOrExpression = AssignmentExpression
LogicalOrExpression
LogicalOrExpression || LogicalAndExpression
LogicalAndExpression
LogicalAndExpression && EqualityExpression
EqualityExpression
EqualityExpression EqualityOperator RelationalExpression
RelationalExpression
RelationalExpression RelationalOperator AdditiveExpression
AdditiveExpression
AdditiveExpression AdditiveOperator MultiplicativeExpression
MultiplicativeExpression
MultiplicativeExpression MultiplicativeOperator UnaryExpression
UnaryExpression
UnaryOperator UnaryExpression
! UnaryExpression
PrimaryExpression
FunctionCall
Identifier (ArgumentList)
Identifier
[a-zA-Z_][a-zA-Z0-9_]*
(AssignmentExpression)
Literal
StringLiteral
'[^']*'
"[^"]*"
NumericLiteral
[0-9]+
BooleanLiteral
false
true
FunctionDeclaration
function Identifier (ParameterList) { StatementList }
WhileStatement
while (Expression) { StatementList }
IfStatement
if (Expression) { StatementList } else { StatementList }
if (Expression) { StatementList }
VariableDeclaration
let Identifier = Expression

کتابخانه graphviz یک ابزار نرم‌افزاری متن‌باز است که به تولید نمودارهای گرافیکی و گرافهای ترتیبی (درختی) از داده‌ها و روابط بین آن‌ها کمک می‌کند.

در ادامه، ویژگی‌های کلیدی و مفاهیم مهم کتابخانه graphviz را مرور می‌کنیم:

۱. **زبان DOT** :

- graphviz از زبان DOT برای توصیف گراف‌ها و نمودارها استفاده می‌کند.DOT یک زبان توصیفی ساده است که اجزای گراف (گره‌ها، یال‌ها و ویژگی‌های دیگر) را توصیف می‌کند.

۲. **نوع‌های گراف:**

- graphviz قابلیت ساخت گراف‌های مختلف را فراهم می‌کند، از جمله گرافهای جهت‌دار و بدون جهت، گرافهای وزن‌دار و بدون وزن، گرافهای ترتیبی، و گرافهای همگن و ناهمگن.

۳. **الگوریتم‌های ترتیب‌دهی:**

- graphviz دارای الگوریتم‌های مختلفی برای ترتیب‌دهی گرافها است. این الگوریتم‌ها باعث ترتیب صحیح و بهینه گره‌ها در نمودارها می‌شوند.

٤. **فرمت‌های خروجی:**

- graphviz قادر به ایجاد نمودارهای در قالب‌های مختلف مانندPNG ، SVG، PDFو ... است. این امکان به کاربران این امکان را می‌دهد که نمودارها را با فرمت‌های مختلف در پروژه‌ها یا اسناد متنی خود استفاده کنند.

٥. **پشتیبانی از زبان‌های برنامه‌نویسی:**

- graphviz به عنوان یک ابزار مستقل ارائه می‌شود ولی ابزارها و کتابخانه‌های برنامه‌نویسی مختلفی برای اتصال به آن وجود دارند. این امکان به برنامه‌نویسان این امکان را می‌دهد که از ویژگی‌های graphviz در داخل برنامه‌ها و اسکریپت‌های خود استفاده کنند.

قبل از هر چیز باید با اجرای دستور زیر در ترمینال، این کتابخانه را نصب کنیم.

```
pip install graphviz
```

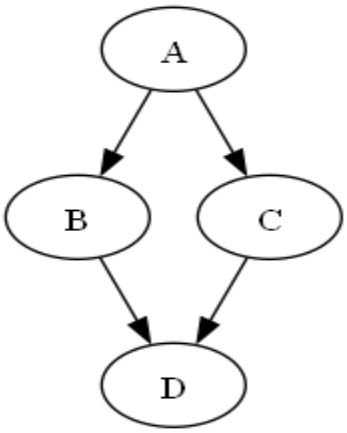
برای استفاده از این کتابخانه، باید کد خود را به کد DOT بنویسیم. با یک مثال ساده میتوان طریقه کار زبان DOT را توضیح داد:

```
digraph G {
   A -> B;
   A -> C;
   B -> D;
   C -> D;
}
```

در اینجا ما یک گراف جهتدار (directed graph) به نام G داریم. این گراف شامل دو یال از طرف نود A به نود های B و C و دو یال از طرف نود های B و C به D است. این قطعه کد را باید به نام فایلی با پسوند DOT ذخیره کنیم. حال برای تبدیل این قطعه کد به تصویر، دستور به قالب زیر را داخل ترمینال اجرا میکنیم:

```
dot -Tpng input.dot -o output.png
```

این دستور تصویر فایل input.dot را در outpt.png ذخیره میکند.



لازم به ذکر است که این زبان ویژگی های زیاد و پیشرفته ای دارد ولی برای این پروژه، همین مقدار توضیحات کافی است. قطعه کد تبدیل شده به زبان DOT به شرح زیر است:

```
digraph G {
"  Program” -> “StatementList;"
"  StatementList” -> “Statement;"
"  StatementList” -> “Statement ; StatementList;"
```

" Statement” -> “VariableDeclaration;"

" Statement” -> “IfStatement;"

" Statement” -> “WhileStatement;"

" Statement” -> “FunctionDeclaration;"

" Statement” -> “ExpressionStatement;"

" VariableDeclaration” -> “let Identifier = Expression;"

" IfStatement” -> “if (Expression) { StatementList };"

" IfStatement” -> “if (Expression) { StatementList } else { StatementList };"

" WhileStatement” -> “while (Expression) { StatementList };"

" FunctionDeclaration” -> “function Identifier(ParameterList) { StatementList};"

" ParameterList” -> “ε;"

" ParameterList” -> “IdentifierList;"

" IdentifierList” -> “Identifier;"

" IdentifierList” -> “Identifier, IdentifierList;"

" ExpressionStatement” -> “Expression;"

" Expression” -> “AssignmentExpression;"

" AssignmentExpression” -> “LogicalOrExpression;"

" AssignmentExpression” -> “LogicalOrExpression = AssignmentExpression;"

" LogicalOrExpression” -> “LogicalAndExpression;"

" LogicalOrExpression” -> “LogicalOrExpression || LogicalAndExpression;"

" LogicalAndExpression” -> “EqualityExpression;"

" LogicalAndExpression” -> “LogicalAndExpression && EqualityExpression;"

" EqualityExpression” -> “RelationalExpression;"

" EqualityExpression” -> “EqualityExpression EqualityOperator RelationalExpression;"

" EqualityOperator;"==" <– "

" EqualityOperator;"=!" <– "

" RelationalExpression” -> “AdditiveExpression;"

" RelationalExpression” -> “RelationalExpression RelationalOperator AdditiveExpression;"

" RelationalOperator;">" <– "

" RelationalOperator;"=>" <– "

" RelationalOperator;"<" <– "

" RelationalOperator;"=<" <– "

" AdditiveExpression” -> “MultiplicativeExpression;"

" AdditiveExpression” -> “AdditiveExpression AdditiveOperator MultiplicativeExpression;"

" AdditiveOperator;"+" <– "

" AdditiveOperator;"–" <– "

" MultiplicativeExpression” -> “UnaryExpression;"

" MultiplicativeExpression” -> “MultiplicativeExpression MultiplicativeOperator UnaryExpression;"

" MultiplicativeOperator;"*" <– "

" MultiplicativeOperator;"/" <– "

" UnaryExpression” -> “PrimaryExpression;"

" UnaryExpression” -> “! UnaryExpression;"

" UnaryExpression” -> “- UnaryExpression;"

" PrimaryExpression” -> “Literal;"

" PrimaryExpression” -> “(AssignmentExpression);"

" PrimaryExpression” -> “Identifier;"

" PrimaryExpression” -> “FunctionCall;"

" Literal” -> “BooleanLiteral;"

" Literal” -> “NumericLiteral;"

" Literal” -> “StringLiteral;"

" BooleanLiteral” -> “true;"

" BooleanLiteral” -> “false;"

" NumericLiteral;"+[۰-۹]" <- "

" StringLiteral;""*["\^]"\" <- "

" StringLiteral;"'*['^]'" <- "

" Identifier” -> “[a-zA-Z_][a-zA-Z۰-۹_];"*

" FunctionCall” -> “Identifier(ArgumentList);"

" ArgumentList” -> “ε;"

" ArgumentList” -> “AssignmentExpression;"

" ArgumentList” -> “AssignmentExpression, ArgumentList;"

}

با توجه به پیچیدگی و همچنین بزرگ بودن تصویر درخت، امکان نمایش آن در این سند وجود ندارد و فایل این تصویر با نام output.png به پیوست الحاق میشود.

متغیرها و نوع‌های داده‌ای

درTypeScript ، انقیاد نوع و مقدار به دو صورت انجام می‌شود: به صورت **صریح** و به صورت **ضمنی**

- انقیاد نوع و مقدار به صورت صریح در TypeScript با استفاده از نوع داده‌ها انجام می‌شود. برای مثال، در کد زیر، نوع داده‌ی پارامتر x به صورت صریح تعریف شده است:

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

- انقیاد نوع و مقدار به صورت ضمنی در TypeScript با استفاده از Type Inference انجام می‌شود. برای مثال، در کد زیر، نوع داده‌ی پارامتر xبه صورت ضمنی تعریف شده است:

```
function add(x, y) {
  return x + y;
}
```

در این حالت، TypeScriptبا استفاده ازType Inference ، نوع داده‌ی پارامتر xرا به صورت خودکار تشخیص داده و انقیاد نوع و مقدار را به صورت ضمنی انجام می‌دهد.

در زبان برنامه نویسی TypeScript، انقیاد نوع و مقدار در زمان کامپایل صورت می‌گیرد. به عبارت دیگر، در زمان کامپایل، نوع و مقدار متغیرها تعیین می‌شود و پس از آن قابل تغییر نیستند.

- **متغیرهای ایستا به طور صریح** : درTypeScript ، شما می‌توانید متغیرهای ایستا را با استفاده از کلیدواژه const تعریف کنید. برای مثال:

```
const pi = 3.14;
```

- **متغیرهای پویا در پشتهبه طور صریح** : درTypeScript ، شما می‌توانید متغیرهای پویا در پشته را با استفاده از کلیدواژه letتعریف کنید. برای مثال:

```
let x = 10;
```

- **متغیرهای پویا در هیپ به طور صریح** :درTypeScript ، شما می‌توانید متغیرهای پویا در هیپ را با استفاده از کلاس Arrayتعریف کنید. برای مثال:

```
let arr: number[] = [1, 2, 3];
```

- **متغیرهای پویا در هیپ به طور ضمنی** :درTypeScript ، شما می‌توانید متغیرهای پویا در هیپ را با استفاده از Type Inferenceبه صورت implicit تعریف کنید. برای مثال:

```
let arr = [1, 2, 3];
```

حوزه تعریف در زبان TypeScript ایستا است یا Static Scope دارد. این بدان معنی است که محدوده یک متغیر در زمان کامپایل مشخص می‌شود و بستگی به محل تعریف آن دارد. به عبارت دیگر، متغیرها در TypeScript فقط در دامنه‌ای که تعریف شده‌اند یا در دامنه‌های زیرمجموعه‌ی آن قابل دسترسی هستند. برای مثال، در کد زیر:

```
let x = 10; // متغیر x در دامنه سراسری تعریف شده است


function foo() {
  let y = 20; // متغیر y در دامنه تابع foo تعریف شده است
  console.log(x); // متغیر x در اینجا قابل دسترسی است چون دامنه سراسری دامنه‌ی بالاتری از دامنه تابع foo است
```

```
  console.log(y); // متغیر y در اینجا قابل دسترسی است چون در همان دامنه تعریف شده است
}

function bar() {
  let z = 30; // متغیر z در دامنه تابع bar تعریف شده است
  console.log(x); // متغیر x در اینجا قابل دسترسی است چون دامنه سراسری دامنه‌ی بالاتری از دامنه تابع bar است
  console.log(y); // متغیر y در اینجا قابل دسترسی نیست چون در دامنه تابع foo تعریف شده است که دامنه‌ی مجزایی از دامنه تابع bar است
  console.log(z); // متغیر z در اینجا قابل دسترسی است چون در همان دامنه تعریف شده است
}

foo(); // این خروجی را می‌دهد: ۱۰, ۲۰
bar(); // این خروجی را می‌دهد: ۱۰ ReferenceError: y is not defined, 30
```

می‌توانید ببینید که متغیرها فقط در دامنه‌هایی که تعریف شده‌اند یا در دامنه‌های زیرمجموعه‌ی آن‌ها قابل دسترسی هستند و این محدوده در زمان کامپایل مشخص می‌شود. این ویژگی باعث می‌شود که کد نویسی در TypeScript ساده‌تر و خطای کمتری داشته باشد.

برای اینکه به زبان TypeScript امکان استفاده از حوزه تعریف پویا (Dynamic Scope) را بیافزاییم، باید چند تغییر در زبان ایجاد کنیم. حوزه تعریف پویا بدین معنی است که محدوده یک متغیر در زمان اجرا مشخص می‌شود و بستگی به محل فراخوانی آن دارد. به عبارت دیگر، متغیرها در حوزه تعریف پویا در هر دامنه‌ای که از آن استفاده شوند قابل دسترسی هستند. برای مثال، در کد زیر:

```
let x = 10; // متغیر x در دامنه سراسری تعریف شده است

function foo() {
  let y = 20; // متغیر y در دامنه تابع foo تعریف شده است
  bar(); // تابع bar را از داخل تابع foo فراخوانی می‌کنیم
```

```
}

function bar() {
  console.log(x); // متغیر x در اینجا قابل دسترسی است چون در دامنه سراسری تعریف شده است
  console.log(y); // متغیر y در اینجا قابل دسترسی است چون از داخل دامنه تابع foo فراخوانی شده است
}

foo(); // این خروجی را می‌دهد: ۱۰, ۲۰
```

می‌توانید ببینید که متغیر y در تابع bar قابل دسترسی است چون از داخل دامنه تابع foo که متغیر y را تعریف کرده است فراخوانی شده است. این ویژگی باعث می‌شود که کد نویسی در حوزه تعریف پویا پیچیده‌تر و خطای بیشتری داشته باشد.

برای اضافه کردن حوزه تعریف پویا به زبانTypeScript ، باید چند گام را طی کنیم:

- اولاً باید یک کلمه کلیدی جدید مانند dynamic ایجاد کنیم که بتوانیم با آن متغیرهایی را تعریف کنیم که حوزه تعریف پویا داشته باشند. به عنوان مثال:

```
dynamic z = 30; // متغیر z با حوزه تعریف پویا تعریف شده است
```

- دوماً باید مکانیزمی را برای ذخیره و بازیابی مقدار متغیرهای حوزه تعریف پویا در زمان اجرا طراحی کنیم. برای این منظور می‌توانیم از یک ساختار داده مانند یک پشته (Stack) استفاده کنیم که هر بار که یک تابع فراخوانی می‌شود، مقدار متغیرهای حوزه تعریف پویا را در آن ذخیره کنیم و هر بار که یک تابع به پایان برسد، مقدار متغیرهای حوزه تعریف پویا را از آن برداریم. به این ترتیب، هر تابع می‌تواند به مقدار متغیرهای حوزه تعریف پویا که در توابع فراخواننده آن‌ها تعریف شده‌اند دسترسی داشته باشد. به عنوان مثال:

```
let stack = []; // یک پشته خالی ایجاد می‌کنیم

function foo() {
  dynamic y = 20; // متغیر y با حوزه تعریف پویا تعریف می‌کنیم
  stack.push(y); // مقدار y را در پشته ذخیره می‌کنیم
  bar(); // تابع bar را فراخوانی می‌کنیم
```

```
  stack.pop(); // مقدار y را از پشته برمی‌داریم
}

function bar() {
  console.log(stack[stack.length - 1]); // مقدار y را از بالای پشته می‌خوانیم
}

foo(); // این خروجی را می‌دهد: ۲۰
```

- سوماً باید قواعد نحوی و معنایی زبان را برای پشتیبانی از حوزه تعریف پویا تغییر دهیم. برای این منظور می‌توانیم از یک کامپایلر یا مفسر جدید برای تفسیر و اجرای کدهای TypeScript که حاوی متغیرهای حوزه تعریف پویا هستند استفاده کنیم. این کامپایلر یا مفسر باید بتواند کلمه کلیدی dynamic را تشخیص دهد و برای مدیریت مقدار متغیرهای حوزه تعریف پویا از پشته استفاده کند.

برای نوشتن یک قطعه کد که در آن حوزه تعریف پویا استفاده شود، می‌توانیم از کد زیر الهام بگیریم:

```
  let stack = []; // یک پشته خالی ایجاد می‌کنیم

  function foo() {
    dynamic x = 10; // با حوزه تعریف پویا تعریف می‌کنیم x متغیر
    stack.push(x); // را در پشته ذخیره می‌کنیم x مقدار
    console.log(x); // را چاپ می‌کنیم x مقدار
    bar(); // را فراخوانی می‌کنیم bar تابع
    stack.pop(); // را از پشته برمی‌داریم x مقدار
  }

  function bar() {
    dynamic x = 20; // با حوزه تعریف پویا تعریف می‌کنیم x متغیر
```

```
  stack.push(x); // مقدار x را در پشته ذخیره می‌کنیم
  console.log(x); // مقدار x را چاپ می‌کنیم
  baz(); // تابع baz را فراخوانی می‌کنیم
  stack.pop(); // مقدار x را از پشته برمی‌داریم
}

function baz() {
  console.log(stack[stack.length - 1]); // مقدار x را از بالای پشته می‌خوانیم
}

foo(); // این خروجی را می‌دهد: ۱۰, ۲۰, ۲۰
```

می‌توانید ببینید که در این کد، متغیر x در هر تابعی که از آن استفاده شود قابل دسترسی است و مقدار آن بستگی به محل فراخوانی آن دارد. این کد نشان می‌دهد که چگونه می‌توانیم از حوزه تعریف پویا در زبان TypeScript استفاده کنیم.

بلوک‌ها در زبان TypeScript به همان شکلی که در جاوااسکریپت تعریف می‌شوند، با استفاده از علامت‌های { و } ایجاد می‌شوند. بلوک‌ها می‌توانند بخشی از یک تابع، یک شرط، یک حلقه یا هر عبارتی باشند که نیاز به یک بلوک دارد. بلوک‌ها محدوده‌ای برای متغیرها و توابع ایجاد می‌کنند که به آن‌ها scope می‌گوییم. متغیرها و توابعی که در یک بلوک تعریف شوند، فقط در آن بلوک قابل دسترسی هستند، مگر اینکه با کلمه‌کلیدی var تعریف شده باشند که در این صورت به scope بالاتر منتقل می‌شوند. برای جلوگیری از این رفتار، می‌توان از کلمه‌کلیدی let یا const برای تعریف متغیرها استفاده کرد که محدوده‌شان به بلوک محدود می‌شود. برای مثال، در کد زیر، متغیر x با var تعریف شده و در خارج از بلوک قابل دسترسی است، اما متغیر y با let تعریف شده و فقط در داخل بلوک قابل دسترسی است:

```
if (true) {
  var x = 10;
  let y = 20;
}
console.log(x); // 10
console.log(y); // Error: y is not defined
```

در اینجا چند نمونه از تعریف بلوک‌ها در TypeScript آورده شده است:

۱. تعریف بلوک در یک تابع:

```
function exampleFunction(){
// این یک بلوک است
    let x = 10;
    let y = 20;
    let result = x + y;
    console.log(result);
}
```

۲. تعریف بلوک در یک شرط:

```
let num = 5;
if (num > 0) {
// این یک بلوک درون شرط است
    console.log;("عدد مثبت است");
} else {
    console.log("عدد منفی یا صفر است");
}
```

۳. تعریف بلوک در یک حلقه:

```
for (let i = 0; i < 5; i++) {
// این یک بلوک درون حلقه است
    console.log(i);
}
```

۴. تعریف بلوک در یک شیء (Object):

```
let person = {
  name: "John,"
  age: 30,
  displayInfo: function(){
// این یک بلوک درون شیء است
    console.log(`Name: ${this.name}, Age: ${this.age}`);
  }
};

person.displayInfo();
```

در هر یک از موارد فوق، آکولادها (`{}`) برای تعریف بلوک‌ها استفاده شده‌اند و دستورات داخلی هر بلوک بین این آکولادها قرار دارند.

همانطور که پیش تر اشاره شد، در زبان TypeScript کلمات کلیدی ویژه‌ای وجود دارند که می‌توانند حوزه‌ی تعریف متغیرها را تغییر دهند. این کلمات کلیدی عبارتند از:

- var : این کلمه کلیدی برای تعریف متغیرهایی استفاده می‌شود که محدوده‌شان به تابع یا فایل محدود نمی‌شود و در هر scopeقابل دسترسی هستند. این کلمه کلیدی همانند جاوااسکریپت عمل می‌کند و ممکن است باعث ایجاد مشکلاتی در کد شود.
- let : این کلمه کلیدی برای تعریف متغیرهایی استفاده می‌شود که محدوده‌شان به بلوک یا حلقه محدود می‌شود و در خارج از آن قابل دسترسی نیستند. این کلمه کلیدی بهتر از var است و می‌تواند جلوی برخی از خطاهای رایج را بگیرد.
- const : این کلمه کلیدی برای تعریف متغیرهایی استفاده می‌شود که مقدار آن‌ها قابل تغییر نیست و باید در زمان تعریف مقداردهی شوند. محدوده‌ی این متغیرها همانند let است و می‌توانند برای ایجاد ثابت‌ها یا مقادیر ثابت استفاده شوند.

برای مثال، در کد زیر، متغیر x با var تعریف شده و در خارج از بلوک قابل دسترسی است، اما متغیر y با let تعریف شده و فقط در داخل بلوک قابل دسترسی است. همچنین متغیر z با const تعریف شده و مقدار آن قابل تغییر نیست:

```
if (true) {
  var x = 10;
```

```
  let y = 20;
  const z = 30;
}
console.log(x); // 10
console.log(y); // Error: y is not defined
console.log(z); // Error: z is not defined
z = 40; // Error: Assignment to constant variable
```

نوع‌های داده‌ای در زبان TypeScript به دو دسته تقسیم می‌شوند: نوع‌های اولیه و نوع‌های پیچیده. نوع‌های اولیه شامل مقادیری هستند که به صورت مستقیم در حافظه ذخیره می‌شوند و می‌توانند از نوع عددی، رشته‌ای، بولین، null یا undefinedباشند. نوع‌های پیچیده شامل مقادیری هستند که به صورت غیرمستقیم در حافظه ذخیره می‌شوند و می‌توانند از نوع آرایه، شیء، تابع، کلاس، اینترفیس، ژنریک، اتحاد، تقاطع یا هر نوع سفارشی دیگری باشند. در ادامه به توضیح هر یک از این نوع‌ها می‌پردازیم.

- **نوع عددی :(number)** این نوع برای نمایش اعداد صحیح یا اعشاری استفاده می‌شود. تایپ‌اسکریپت از اعداد مبنای ۱۰، ۱۶، ۲ و ۸ پشتیبانی می‌کند. برای تعریف یک متغیر از نوع عددی، می‌توان از کلمه‌کلیدی number استفاده کرد. مثال:

```
let decimal: number = 6; // عدد مبنای ۱۰
let hex: number = 0xf00d; // عدد مبنای ۱۶
let binary: number = 0b1010; // عدد مبنای ۲
let octal: number = 0o744; // عدد مبنای ۸
```

- **نوع رشته‌ای :(string)** این نوع برای نمایش متن‌ها یا رشته‌های حرفی استفاده می‌شود. تایپ‌اسکریپت از نوشتن رشته‌ها با استفاده از علامت‌های " و ' پشتیبانی می‌کند. همچنین از نوشتن رشته‌ها با استفاده از علامت ` نیز پشتیبانی می‌کند که به آن‌ها رشته‌های الگویی (template strings) می‌گویند. این رشته‌ها می‌توانند چند خطی باشند و می‌توان در آن‌ها از عبارات متغیری با استفاده از نشانه ${expr} استفاده کرد. برای تعریف یک متغیر از نوع رشته‌ای، می‌توان از کلمه‌کلیدی stringاستفاده کرد. مثال:

```
let color: string = "blue"; // رشته با علامت "
```

```
color = 'red'; // رشته با علامت '

let name: string = `John`; // رشته الگویی

let age: number = 25;

let sentence: string = `Hello, my name is ${name}. I'll be ${age + ۱} □□□□□ □□□ □□□□ □□□□□.□□ // رشته الگویی با عبارت متغیری
```

- **نوع بولین :(boolean)** این نوع برای نمایش مقادیر درست یا غلط استفاده می‌شود. تایپ‌اسکریپت از مقادیر true و falseبرای این نوع پشتیبانی می‌کند. برای تعریف یک متغیر از نوع بولین، می‌توان از کلمه‌کلیدی boolean استفاده کرد. مثال:

```
let isDone: boolean = false; // مقدار غلط

isDone = true; // مقدار درست
```

- **نوع null و :undefined** این دو نوع برای نمایش مقادیر خالی یا تعریف نشده استفاده می‌شوند. تایپ‌اسکریپت از مقادیر nullو undefined برای این نوع‌ها پشتیبانی می‌کند. این دو نوع به صورت پیش‌فرض زیرمجموعه‌ای از همه نوع‌ها هستند و می‌توان آن‌ها را به هر نوع دیگری اختصاص داد. اما اگر از گزینه‌ی strictNullChecks در فایل tsconfig.jsonاستفاده کنیم، این رفتار تغییر می‌کند و فقط می‌توان این دو نوع را به خودشان یا نوع any اختصاص داد. مثال:

```
let n: null = null; // مقدار خالی

let u: undefined = undefined; // مقدار تعریف نشده

let num: number = null; // اگر گزینه‌ی strictNullChecks فعال نباشد، این امکان وجود دارد

let str: string = undefined; // اگر گزینه‌ی strictNullChecks فعال نباشد، این امکان وجود دارد

let x: any = null; // این همیشه امکان‌پذیر است

let y: any = undefined; // این همیشه امکان‌پذیر است
```

- **نوع هر چیز :(any)** این نوع برای نمایش مقادیری استفاده می‌شود که نوع آن‌ها مشخص نیست یا می‌خواهیم از بررسی نوع آن‌ها توسط کامپایلر تایپ‌اسکریپت صرف‌نظر کنیم. تایپ‌اسکریپت از هر مقداری برای این نوع پشتیبانی می‌کند و هیچ

خطایی در مورد نوع آن برنمی‌گرداند. برای تعریف یک متغیر از نوع هر چیز، می‌توان از کلمه‌کلیدی any استفاده کرد. مثال:

```
let notSure: any = 4; // مقدار عددی
notSure = "maybe a string"; // مقدار رشته‌ای
notSure = false; // مقدار بولین
notSure.ifItExists(); // هیچ خطایی نمی‌دهد
notSure.toFixed(); // هیچ خطایی نمی‌دهد
```

- **نوع آرایه :(array)** این نوع برای نمایش مجموعه‌ای از مقادیر مشابه استفاده می‌شود که می‌توان به آن‌ها با استفاده از اندیس دسترسی داشت. تایپ‌اسکریپت از دو روش برای تعریف نوع آرایه پشتیبانی می‌کند. روش اول این است که بعد از نوع مقادیر آرایه، علامت [ ] را قرار دهیم. روش دوم این است که از نوع جنریک Array<elemType> استفاده کنیم که elemType نوع مقادیر آرایه است. برای تعریف یک متغیر از نوع آرایه، می‌توان از هر یک از این روش‌ها استفاده کرد. مثال:

```
let list: number[] = [1, 2, 3]; // آرایه از نوع عددی با روش اول
let list: Array<number> = [1, 2, 3]; // آرایه از نوع عددی با روش دوم
let names: string[] = ["Ali", "Reza", "Sara"]; // آرایه از نوع رشته‌ای با روش اول
let names: Array<string> = ["Ali", "Reza", "Sara"]; // آرایه از نوع رشته‌ای با روش دوم
```

- **نوع شیء :(object)** این نوع برای نمایش مقادیری استفاده می‌شود که دارای خصوصیات و روش‌هایی هستند که می‌توان با استفاده از نقطه به آن‌ها دسترسی داشت. تایپ‌اسکریپت از نوشتن شیء‌ها با استفاده از علامت‌های { و } پشتیبانی می‌کند. برای تعریف یک متغیر از نوع شیء، می‌توان از کلمه‌کلیدی object استفاده کرد. مثال:

```
let person: object = {
  name: "John",
  age: 25,
  greet: function() {
    console.log("Hello, " + this.name);
  }
```

```
}; // شیء از نوع object

person.name; // خصوصیت name

person.age; // خصوصیت age

person.greet(); // روش greet
```

- **نوع تابع :(function)** این نوع برای نمایش مقادیری استفاده می‌شود که قابل فراخوانی هستند و می‌توانند پارامترها و خروجی‌هایی داشته باشند. تایپ‌اسکریپت از نوشتن توابع با استفاده از کلمه‌کلیدی function پشتیبانی می‌کند. برای تعریف یک متغیر از نوع تابع، می‌توان از نوع جنریک (...args: any[]) => any استفاده کرد که args... نشان‌دهنده‌ی پارامترهای تابع و any نشان‌دهنده‌ی خروجی تابع است. مثال:

```
let add: (...args: any[]) => any = function(x: number, y: number) {
  return x + y;
}; // تابع از نوع تابع

add(1, 2); // فراخوانی تابع با پارامترهای ۱ و ۲
```

- **نوع کلاس :(class)** این نوع برای نمایش مقادیری استفاده می‌شود که دارای خصوصیات و روش‌هایی هستند که می‌توان با استفاده از نقطه به آن‌ها دسترسی داشت. تایپ‌اسکریپت از نوشتن کلاس‌ها با استفاده از کلمه‌کلیدی class پشتیبانی می‌کند. برای تعریف یک متغیر از نوع کلاس، می‌توان از نام کلاس استفاده کرد. مثال:

```
class Animal {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
  move(distance: number) {
    console.log(`${this.name} moved ${distance} meters.`);
  }
} // کلاس Animal

let dog: Animal = new Animal("Rex"); // شیء از نوع کلاس Animal
```

```
dog.name; // خصوصیت name
dog.move(10); // روش move
```

- **نوع اینترفیس :(interface)** این نوع برای تعریف قراردادی برای ساختار یک شیء استفاده می‌شود. تایپ‌اسکریپت از نوشتن اینترفیس‌ها با استفاده از کلمه‌کلیدی interface پشتیبانی می‌کند. برای تعریف یک متغیر از نوع اینترفیس، می‌توان از نام اینترفیس استفاده کرد. مثال:

```
interface Person {
  name: string;
  age: number;
  greet(): void;
} // اینترفیس Person
let john: Person = {
  name: "John",
  age: 25,
  greet: function() {
    console.log("Hello, " + this.name);
  }
}; // شیء از نوع اینترفیس Person
john.name; // خصوصیت name
john.age; // خصوصیت age
john.greet(); // روش greet
```

- **نوع ژنریک :(generic)** این نوع برای تعریف نوع‌هایی استفاده می‌شود که وابسته به یک یا چند پارامتر نوعی هستند. تایپ‌اسکریپت از نوشتن نوع‌های ژنریک با استفاده از علامت‌های < و > پشتیبانی می‌کند. برای تعریف یک متغیر از نوع ژنریک، می‌توان از نام نوع ژنریک با پارامترهای نوعی مورد نظر استفاده کرد. مثال:

```
function identity<T>(arg: T): T {
  return arg;
} // تابع ژنریک با پارامتر نوعی T

let output: string = identity<string>("myString"); // متغیر از نوع ژنریک با پارامتر نوعی string

let output: number = identity<number>(42); // متغیر از نوع ژنریک
```

- **نوع اتحاد :(union)** این نوع برای تعریف نوع‌هایی استفاده می‌شود که می‌توانند یکی از چند نوع مشخص شده باشند. تایپ‌اسکریپت از استفاده از علامت | برای ایجاد نوع اتحاد پشتیبانی می‌کند. برای تعریف یک متغیر از نوع اتحاد، می‌توان از نوع‌های مورد نظر با علامت | جدا شده استفاده کرد. مثال:

```
let value: number | string; // متغیر از نوع اتحاد

value = 10; // مقدار عددی

value = "hello"; // مقدار رشته‌ای

value = true; // خطا: مقدار بولین
```

- **نوع تقاطع :(intersection)** این نوع برای تعریف نوع‌هایی استفاده می‌شود که شامل تمام خصوصیات و روش‌های چند نوع مشخص شده هستند. تایپ‌اسکریپت از استفاده از علامت & برای ایجاد نوع تقاطع پشتیبانی می‌کند. برای تعریف یک متغیر از نوع تقاطع، می‌توان از نوع‌های مورد نظر با علامت & جدا شده استفاده کرد. مثال:

```
interface Shape {
  area(): number;
}
interface Color {
  color: string;
}
let square: Shape & Color; // متغیر از نوع تقاطع
```

```
square = {
  area: function() {
    return 100;
  },
  color: "red"
}; // شیء از نوع تقاطع
square.area(); // روش area
square.color; // خصوصیت color
```

- **نوع سفارشی (custom):** این نوع برای تعریف نوع‌هایی استفاده می‌شود که با استفاده از کلمه‌کلیدی `type` ایجاد شده‌اند. تایپ‌اسکریپت از استفاده از کلمه‌کلیدی `type` برای ایجاد نوع سفارشی پشتیبانی می‌کند. برای تعریف یک متغیر از نوع سفارشی، می‌توان از نام نوع سفارشی استفاده کرد. مثال:

```
type ID = number | string; // نوع سفارشی
let userId: ID; // متغیر از نوع سفارشی
userId = 123; // مقدار عددی
userId = "abc"; // مقدار رشته‌ای
userId = true; // خطا: مقدار بولین
```

این‌ها برخی از نوع‌های داده‌ای در زبان `TypeScript` هستند که می‌توانند برای تعریف متغیرها و توابع استفاده شوند.

در ادامه درباره نحوه پیاده سازی هر یک از این متغیر ها و نحوه تخصیص آن‌ها در حافظه توضیح خواهیم داد.

- **نوع عددی (number):** این نوع داده در حافظه به صورت ۶۴ بیتی با فرمت `IEEE 754` ذخیره می‌شود. این فرمت شامل یک بیت برای علامت، ۱۱ بیت برای اندیس و ۵۲ بیت برای مقدار اعشاری است. برای پیاده‌سازی این نوع داده، تایپ‌اسکریپت از کلاس `Number` استفاده می‌کند که دارای خصوصیات و روش‌هایی برای کار با اعداد است. مثال:

```
let num: number = 3.14; // مقدار عددی
```

```
console.log(num.toString(2)); // تبدیل به رشته با مبنای ۲
console.log(Number.isInteger(num)); // بررسی که آیا عدد صحیح است
console.log(Number.MAX_VALUE); // بزرگترین مقدار قابل نمایش با نوع عددی
```

- **نوع رشته‌ای (string):** این نوع داده در حافظه به صورت یک آرایه از کاراکترها ذخیره می‌شود. هر کاراکتر ۱۶ بیت فضا اشغال می‌کند و با استفاده از کدگذاری UTF-16 نمایش داده می‌شود. برای پیاده‌سازی این نوع داده، تایپ‌اسکریپت از کلاس String استفاده می‌کند که دارای خصوصیات و روش‌هایی برای کار با رشته‌ها است. مثال:

```
let str: string = "Hello"; // مقدار رشته‌ای
console.log(str.length); // طول رشته
console.log(str[0]); // کاراکتر اول رشته
console.log(str.toUpperCase()); // تبدیل به حروف بزرگ
console.log(str.concat(" World")); // الحاق رشته دیگر
```

- **نوع بولین (boolean):** این نوع داده در حافظه به صورت یک بیت ذخیره می‌شود که می‌تواند مقدار ۰ یا ۱ داشته باشد. برای پیاده‌سازی این نوع داده، تایپ‌اسکریپت از کلاس Boolean استفاده می‌کند که دارای خصوصیات و روش‌هایی برای کار با مقادیر بولین است. مثال:

```
let flag: boolean = true; // مقدار بولین
console.log(flag.valueOf()); // مقدار واقعی
console.log(flag.toString()); // تبدیل به رشته
console.log(!flag); // عملگر منطقی نقیض
```

- **نوع null و undefined:** این دو نوع داده در حافظه به صورت یک مقدار خاص ذخیره می‌شوند که نشان‌دهنده‌ی عدم وجود یا تعریف یک متغیر هستند. برای پیاده‌سازی این دو نوع داده، تایپ‌اسکریپت از مقادیر null و undefined استفاده می‌کند که دارای خصوصیات و روش‌هایی برای کار با آن‌ها هستند. مثال:

```
let x: null = null; // مقدار خالی
let y: undefined = undefined; // مقدار تعریف نشده
console.log(x == y); // برابری با تبدیل نوع
```

```
console.log(x === y); // برابری بدون تبدیل نوع
console.log(typeof x); // نوع داده
console.log(typeof y); // نوع داده
```

- **نوع هر چیز (any):** این نوع داده در حافظه به صورت هر نوع داده‌ای که مقدار متغیر دارد ذخیره می‌شود. برای پیاده‌سازی این نوع داده، تایپ‌اسکریپت از مقادیر هر نوع داده‌ای استفاده می‌کند و هیچ بررسی نوعی را بر روی آن‌ها انجام نمی‌دهد. مثال:

```
let z: any = 10; // مقدار عددی
z = "hello"; // مقدار رشته‌ای
z = true; // مقدار بولین
z.ifItExists(); // هیچ خطایی نمی‌دهد
z.toFixed(); // هیچ خطایی نمی‌دهد
```

- **نوع آرایه (array):** این نوع داده در حافظه به صورت یک مجموعه از مقادیر مشابه که در مکان‌های متوالی قرار دارند ذخیره می‌شود. برای پیاده‌سازی این نوع داده، تایپ‌اسکریپت از کلاس `Array` استفاده می‌کند که دارای خصوصیات و روش‌هایی برای کار با آرایه‌ها است. مثال:

```
let arr: number[] = [1, 2, 3]; // آرایه از نوع عددی
console.log(arr.length); // طول آرایه
console.log(arr[0]); // عنصر اول آرایه
console.log(arr.push(4)); // اضافه کردن عنصر به آخر آرایه
console.log(arr.pop()); // حذف کردن عنصر از آخر آرایه
```

- **نوع شیء (object):** این نوع داده در حافظه به صورت یک مجموعه از خصوصیات و روش‌هایی که با استفاده از نقطه قابل دسترسی هستند ذخیره می‌شود. برای پیاده‌سازی این نوع داده، تایپ‌اسکریپت از کلاس `Object` استفاده می‌کند که دارای خصوصیات و روش‌هایی برای کار با شیءها است. مثال:

```
let obj: object = {
  name: "John",
  age: 25,
```

```
  greet: function() {
    console.log("Hello, " + this.name);
  }
}; // شیء از نوع object

console.log(obj.name); // خصوصیت name

console.log(obj.age); // خصوصیت age
```

- **نوع تابع (function):** این نوع داده در حافظه به صورت یک کد قابل فراخوانی ذخیره می‌شود. برای پیاده‌سازی این نوع داده، تایپ‌اسکریپت از کلاس Function استفاده می‌کند که دارای خصوصیات و روش‌هایی برای کار با توابع است. مثال:

```
let add: (...args: any[]) => any = function(x: number, y: number) {
  return x + y;
}; // تابع از نوع تابع

console.log(add(1, 2)); // فراخوانی تابع با پارامترهای ۱ و ۲

console.log(add.name); // نام تابع

console.log(add.length); // تعداد پارامترهای تابع

console.log(add.call(null, 3, 4)); // فراخوانی تابع با تغییر مقدار this
```

- **نوع کلاس (class):** این نوع داده در حافظه به صورت یک قالب برای ساخت شیءها ذخیره می‌شود. برای پیاده‌سازی این نوع داده، تایپ‌اسکریپت از کلاس‌های جاوااسکریپت استفاده می‌کند که دارای خصوصیات و روش‌هایی برای کار با کلاس‌ها است. مثال:

```
class Animal {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
```

```
  move(distance: number) {
    console.log(`${this.name} moved ${distance} meters.`);
  }
} // کلاس Animal

let dog: Animal = new Animal("Rex"); // شیء از نوع کلاس Animal

console.log(dog.name); // خصوصیت name

console.log(dog.move(10)); // روش move

console.log(dog instanceof Animal); // بررسی که آیا شیء از نوع کلاس Animal است

console.log(Animal.prototype); // پروتوتایپ کلاس Animal
```

- **نوع اینترفیس (interface):** این نوع داده در حافظه به صورت یک قرارداد برای ساختار شیءها ذخیره می‌شود. برای پیاده‌سازی این نوع داده، تایپ‌اسکریپت از اینترفیس‌های جاوااسکریپت استفاده می‌کند که دارای خصوصیات و روش‌هایی برای کار با اینترفیس‌ها است. مثال:

```
interface Person {
  name: string;
  age: number;
  greet(): void;
} // اینترفیس Person

let john: Person = {
  name: "John",
  age: 25,
  greet: function() {
    console.log("Hello, " + this.name);
  }
}; // شیء از نوع اینترفیس Person
```

```
console.log(john.name); // خصوصیت name
console.log(john.age); // خصوصیت age
console.log(john.greet()); // روش greet
console.log(john instanceof Person); // خطا: اینترفیس‌ها در زمان اجرا وجود ندارند
console.log(Person.prototype); // خطا: اینترفیس‌ها در زمان اجرا وجود ندارند
```

- **نوع ژنریک (generic):** این نوع داده در حافظه به صورت یک نوع وابسته به پارامترهای نوعی ذخیره می‌شود. برای پیاده‌سازی این نوع داده، تایپ‌اسکریپت از نوع‌های ژنریک جاوااسکریپت استفاده می‌کند که دارای خصوصیات و روش‌هایی برای کار با نوع‌های ژنریک است. مثال:

```
function identity<T>(arg: T): T {
  return arg;
} // تابع ژنریک با پارامتر نوعی T
let output: string = identity<string>("myString"); // متغیر از نوع ژنریک با پارامتر نوعی string
let output: number = identity<number>(42); // متغیر از نوع ژنریک با پارامتر نوعی number
console.log(output); // مقدار متغیر
console.log(typeof output); // نوع داده
console.log(identity.prototype); // پروتوتایپ تابع
```

- **نوع اتحاد (union):** این نوع داده در حافظه به صورت یک نوع که می‌تواند یکی از چند نوع مشخص شده باشد ذخیره می‌شود. برای پیاده‌سازی این نوع داده، تایپ‌اسکریپت از نوع‌های اتحاد جاوااسکریپت استفاده می‌کند که دارای خصوصیات و روش‌هایی برای کار با نوع‌های اتحاد است. مثال:

```
let value: number | string; // متغیر از نوع اتحاد
value = 10; // مقدار عددی
value = "hello"; // مقدار رشته‌ای
```

```
value = true; // خطا: مقدار بولین
console.log(value); // مقدار متغیر
console.log(typeof value); // نوع داده
console.log(value.length); // خصوصیت length فقط برای رشته‌ها قابل دسترسی است
```

- **نوع تقاطع (intersection):** این نوع داده در حافظه به صورت یک نوع که شامل تمام خصوصیات و روش‌های چند نوع مشخص شده است ذخیره می‌شود. برای پیاده‌سازی این نوع داده، تایپ‌اسکریپت از نوع‌های تقاطع جاوااسکریپت استفاده می‌کند که دارای خصوصیات و روش‌هایی برای کار با نوع‌های تقاطع است. مثال:

```
interface Shape {
  area(): number;
}
interface Color {
  color: string;
}
let square: Shape & Color; // متغیر از نوع تقاطع
square = {
  area: function() {
    return 100;
  },
  color: "red"
}; // شیء از نوع تقاطع
console.log(square.area()); // روش area
console.log(square.color); // خصوصیت color
console.log(square instanceof Shape); // خطا: اینترفیس‌ها در زمان اجرا وجود ندارند
console.log(square instanceof Color); // خطا: اینترفیس
```

- **نوع سفارشی (custom):** این نوع داده در حافظه به صورت یک نوع که با استفاده از کلمه‌کلیدی `type` ایجاد شده است ذخیره می‌شود. برای پیاده‌سازی این نوع داده، تایپ‌اسکریپت از نوع‌های سفارشی جاوااسکریپت استفاده می‌کند که دارای خصوصیات و روش‌هایی برای کار با نوع‌های سفارشی است. مثال:

```
type ID = number | string; // نوع سفارشی

let userId: ID; // متغیر از نوع سفارشی

userId = 123; // مقدار عددی

userId = "abc"; // مقدار رشته‌ای

userId = true; // خطا: مقدار بولین

console.log(userId); // مقدار متغیر

console.log(typeof userId); // نوع داده
```

این‌ها برخی از نوع‌های داده‌ای در زبان `TypeScript` هستند که می‌توانند برای تعریف متغیرها و توابع استفاده شوند.

- **نوع عددی (number):** برای این نوع داده، عملگرهای حسابی، رابطه‌ای، منطقی، بیتی و انتسابی تعریف شده‌اند. مثال:

```
let x: number = 10; // مقدار عددی

let y: number = 5; // مقدار عددی

console.log(x + y); // عملگر حسابی جمع

console.log(x > y); // عملگر رابطه‌ای بزرگتر از

console.log(x && y); // عملگر منطقی و

console.log(x | y); // عملگر بیتی یا

console.log(x = y); // عملگر انتسابی
```

- **نوع رشته‌ای (string):** برای این نوع داده، عملگرهای رابطه‌ای، منطقی، الحاق و انتسابی تعریف شده‌اند. مثال:

```
let s: string = "Hello"; // مقدار رشته‌ای

let t: string = "World"; // مقدار رشته‌ای
```

```
console.log(s == t); // عملگر رابطه‌ای برابری
console.log(s || t); // عملگر منطقی یا
console.log(s + t); // عملگر الحاق
console.log(s = t); // عملگر انتسابی
```

- **نوع بولین (boolean):** برای این نوع داده، عملگرهای رابطه‌ای، منطقی، بیتی و انتسابی تعریف شده‌اند. مثال:

```
let a: boolean = true; // مقدار بولین
let b: boolean = false; // مقدار بولین
console.log(a == b); // عملگر رابطه‌ای برابری
console.log(a && b); // عملگر منطقی و
console.log(a & b); // عملگر بیتی و
console.log(a = b); // عملگر انتسابی
```

- **نوع null و undefined:** برای این دو نوع داده، عملگرهای رابطه‌ای، منطقی و انتسابی تعریف شده‌اند. مثال:

```
let n: null = null; // مقدار خالی
let u: undefined = undefined; // مقدار تعریف نشده
console.log(n == u); // عملگر رابطه‌ای برابری با تبدیل نوع
console.log(n === u); // عملگر رابطه‌ای برابری بدون تبدیل نوع
console.log(n || u); // عملگر منطقی یا
console.log(n = u); // عملگر انتسابی
```

- **نوع هر چیز (any):** برای این نوع داده، همه عملگرهای ممکن برای هر نوع داده‌ای که مقدار متغیر دارد تعریف شده‌اند. مثال:

```
let z: any = 10; // مقدار عددی
z = "hello"; // مقدار رشته‌ای
```

```
z = true; // مقدار بولین
console.log(z + z); // عملگر حسابی یا الحاق
console.log(z > z); // عملگر رابطه‌ای
console.log(z && z); // عملگر منطقی
console.log(z | z); // عملگر بیتی
console.log(z = z); // عملگر انتسابی
```

- **نوع آرایه (array):** برای این نوع داده، عملگرهای رابطه‌ای، منطقی، اندیس‌گذاری و انتسابی تعریف شده‌اند. مثال:

```
let arr: number[] = [1, 2, 3]; // آرایه از نوع عددی
let arr2: number[] = [4, 5, 6]; // آرایه از نوع عددی
console.log(arr == arr2); // عملگر رابطه‌ای برابری
console.log(arr || arr2); // عملگر منطقی یا
console.log(arr[0]); // عملگر اندیس‌گذاری
console.log(arr = arr2); // عملگر انتسابی
```

- **نوع شیء (object):** برای این نوع داده، عملگرهای رابطه‌ای، منطقی، نقطه، اشاره‌گر به عضو و انتسابی تعریف شده‌اند. مثال:

```
let obj: object = {
  name: "John",
  age: 25,
  greet: function() {
    console.log("Hello, " + this.name);
  }
}; // شیء از نوع object
```

```
let obj2: object = {
  name: "Mary",
  age: 20,
  greet: function() {
    console.log("Hi, " + this.name);
  }
}; // شیء از نوع object
console.log(obj == obj2); // عملگر رابطه‌ای برابری
console.log(obj || obj2); // عملگر منطقی یا
console.log(obj.name); // عملگر نقطه
console.log(obj->greet()); // عملگر اشاره‌گر به عضو
console.log(obj = obj2); // عملگر انتسابی
```

- **نوع تابع :(function)** برای این نوع داده، عملگرهای رابطه‌ای، منطقی، پرانتز، نقطه و انتسابی تعریف شده‌اند.
مثال:

```
let add: (...args: any[]) => any = function(x: number, y: number) {
  return x + y;
}; // تابع از نوع تابع
let sub: (...args: any[]) => any = function(x: number, y: number) {
  return x - y;
}; // تابع از نوع تابع
console.log(add == sub); // عملگر رابطه‌ای برابری
console.log(add || sub); // عملگر منطقی یا
console.log(add(1, 2)); // عملگر پرانتز
```

```
console.log(add.name); // عملگر نقطه
console.log(add = sub); // عملگر انتسابی
```

- **نوع کلاس :(class)** برای این نوع داده، عملگرهای رابطه‌ای، منطقی، پرانتز، نقطه، اشاره‌گر به عضو و انتسابی تعریف شده‌اند. مثال:

```
class Animal {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
  move(distance: number) {
    console.log(`${this.name
```


- **نوع کلاس :(class)** برای این نوع داده، عملگرهای رابطه‌ای، منطقی، پرانتز، نقطه، اشاره‌گر به عضو و انتسابی تعریف شده‌اند. مثال:


```
class Animal {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
  move(distance: number) {
    console.log(`${this.name} moved ${distance} meters.`);
  }
} // کلاس Animal
let dog: Animal = new Animal("Rex"); // شیء از نوع کلاس Animal
console.log(dog.name); // خصوصیت name
console.log(dog.move(10)); // روش move
```

```
console.log(dog instanceof Animal); // عملگر رابطه‌ای نمونه‌ی
console.log(Animal.prototype); // عملگر نقطه
console.log(dog->name); // عملگر اشاره‌گر به عضو
console.log(dog = new Animal("Max")); // عملگر انتسابی
```

- **نوع اینترفیس (interface):** این نوع داده در حافظه به صورت یک قرارداد برای ساختار شیءها ذخیره می‌شود. برای پیاده‌سازی این نوع داده، تایپ‌اسکریپت از اینترفیس‌های جاوااسکریپت استفاده می‌کند که دارای خصوصیات و روش‌هایی برای کار با اینترفیس‌ها است. مثال:

```
interface Person {
  name: string;
  age: number;
  greet(): void;
} // اینترفیس Person
let john: Person = {
  name: "John",
  age: 25,
  greet: function() {
    console.log("Hello, " + this.name);
  }
}; // شیء از نوع اینترفیس Person
console.log(john.name); // خصوصیت name
console.log(john.age); // خصوصیت age
console.log(john.greet()); // روش greet
console.log(john instanceof Person); // خطا: اینترفیس‌ها در زمان اجرا وجود ندارند
console.log(Person.prototype); // خطا: اینترفیس‌ها در زمان اجرا وجود ندارند
```

- **نوع ژنریک (generic):** این نوع داده در حافظه به صورت یک نوع وابسته به پارامترهای نوعی ذخیره می‌شود. برای پیاده‌سازی این نوع داده، تایپ‌اسکریپت از نوع‌های ژنریک جاوااسکریپت استفاده می‌کند که دارای خصوصیات و روش‌هایی برای کار با نوع‌های ژنریک است. مثال:

```
function identity<T>(arg: T): T {
  return arg;
} // تابع ژنریک با پارامتر نوعی T

let output: string = identity<string>("myString"); // متغیر از نوع ژنریک با پارامتر نوعی string

let output: number = identity<number>(42); // متغیر از نوع ژنریک با پارامتر نوعی number

console.log(output); // مقدار متغیر

console.log(typeof output); // نوع داده

console.log(identity.prototype); // پروتوتایپ تابع
```

- **نوع اتحاد (union):** این نوع داده در حافظه به صورت یک نوع که می‌تواند یکی از چند نوع مشخص شده باشد ذخیره می‌شود. برای پیاده‌سازی این نوع داده، تایپ‌اسکریپت از نوع‌های اتحاد جاوااسکریپت استفاده می‌کند که دارای خصوصیات و روش‌هایی برای کار با نوع‌های اتحاد است. مثال:

```
let value: number | string; // متغیر از نوع اتحاد

value = 10; // مقدار عددی

value = "hello"; // مقدار رشته‌ای

value = true; // خطا: مقدار بولین

console.log(value); // مقدار متغیر

console.log(typeof value); // نوع داده

console.log(value.length); // خصوصیت length فقط برای رشته‌ها قابل دسترسی است
```

- **نوع تقاطع (intersection):** این نوع داده در حافظه به صورت یک نوع که شامل تمام خصوصیات و روش‌های چند نوع مشخص شده است ذخیره می‌شود. برای پیاده‌سازی این نوع داده، تایپ‌اسکریپت از نوع‌های تقاطع جاوااسکریپت استفاده می‌کند که دارای خصوصیات و روش‌هایی برای کار با نوع‌های تقاطع است. مثال:

```
interface Shape {
  area(): number;
}
interface Color {
  color: string;
}
let square: Shape & Color; // متغیر از نوع تقاطع
square = {
  area: function() {
    return 100;
  },
  color: "red"
}; // شیء از نوع تقاطع
console.log(square.area()); // روش area
console.log(square.color); // خصوصیت color
console.log(square instanceof Shape); // خطا: اینترفیس‌ها در زمان اجرا وجود ندارند
console.log(square instanceof Color); // خطا: اینترفیس‌ها در زمان اجرا وجود ندارند
```

- **نوع سفارشی (custom):** این نوع داده در حافظه به صورت یک نوع که با استفاده از کلمه‌کلیدی `type` ایجاد شده است ذخیره می‌شود. برای پیاده‌سازی این نوع داده، تایپ‌اسکریپت از نوع‌های سفارشی جاوااسکریپت استفاده می‌کند که دارای خصوصیات و روش‌هایی برای کار با نوع‌های سفارشی است. مثال:

```
type ID = number | string; // نوع سفارشی
let userId: ID; // متغیر از نوع سفارشی
userId = 123; // مقدار عددی
userId = "abc"; // مقدار رشته‌ای
userId = true; // خطا: مقدار بولین
```

```
console.log(userId); // مقدار متغیر

console.log(typeof userId); // نوع داده
```

لیست‌ها در TypeScript نوع داده‌ای هستند که می‌توانند مجموعه‌ای از عناصر با نوع داده یکسان یا متفاوت را نگهداری کنند. لیست‌ها در TypeScript با استفاده از علامت کروشه ([]) تعریف می‌شوند. برای مثال، لیست زیر شامل سه عنصر از نوعstring ، numberو boolean است:

```
let list = [“Hello”, 42, true];
```

رشته‌ها در TypeScript نوع داده‌ای هستند که می‌توانند مجموعه‌ای از کاراکترها را نمایش دهند. رشته‌ها در TypeScriptبا استفاده از علامت نقل قول تکی (‘’) یا دوتایی (“”) تعریف می‌شوند. برای مثال، رشته زیر شامل پنج کاراکتر از نوع string است:

```
let str = “Hello”;
```

آرایه‌های انجمنی در TypeScript نوع داده‌ای هستند که می‌توانند مجموعه‌ای از جفت‌های (کلید، مقدار) را ذخیره کنند. کلیدها می‌توانند از نوع string یا symbol باشند و مقادیر می‌توانند از هر نوع داده‌ای باشند. آرایه‌های انجمنی در TypeScriptبا استفاده از علامت آکولاد ({}) تعریف می‌شوند. برای مثال، آرایه انجمنی زیر شامل سه جفت از نوع (string, number)، (string, string)و (string, boolean) است:

```
let obj = {“name”: “Ali”, “age”: 25, “married”: false};
```

اشاره گرها و متغیرهای مرجع در زبان TypeScript به این صورت تعریف می‌شوند که یک متغیر را با استفاده از علامت = به یک متغیر دیگر اختصاص می‌دهیم. این باعث می‌شود که هر دو متغیر به یک شیء در حافظه اشاره کنند و هر تغییری که در یکی از آن‌ها ایجاد شود، بر روی دیگری نیز اعمال شود. برای مثال:

```
let a = {name: "Ali"};
let b = a; // b is a reference to a
a.age = 25; // add a new property to a
console.log(b.age); // 25
```

در این کد، متغیر b یک اشاره گر به متغیر a است و هر دو به یک شیء در حافظه اشاره می‌کنند. بنابراین، وقتی ما یک خاصیت جدید به a اضافه می‌کنیم، آن خاصیت در b نیز قابل دسترسی است. این رفتار با نوع‌های داده‌ای اولیه مانند رشته‌ها، اعداد و بولین‌ها متفاوت است. زیرا این نوع‌ها با ارزش (value) و نه با اشاره (reference) منتقل می‌شوند. برای مثال:

```
let x = "Hello";
let y = x; // y is a copy of x
x = x + " World"; // modify x
console.log(y); // Hello
```

در این کد، متغیر y یک کپی از مقدار x است و هر دو به رشته‌های متفاوتی در حافظه اشاره می‌کنند. بنابراین، وقتی ما مقدار xرا تغییر می‌دهیم، مقدار y تغییری نمی‌کند.

نشتی حافظه و اشاره گر معلق دو مشکل رایج در برنامه‌نویسی هستند که می‌توانند باعث کاهش عملکرد و پایداری برنامه شوند. در زبانTypeScript ، برای رفع این مشکلات، چندین سازوکار وجود دارد که در ادامه به آن‌ها اشاره می‌کنیم:

- استفاده از کلمه کلیدی let به جای var برای تعریف متغیرها. این کار باعث می‌شود که متغیرها فقط در محدوده‌ی بلوکی که تعریف شده‌اند، قابل دسترسی باشند و در صورت عدم استفاده، از حافظه پاک شوند. این کار می‌تواند جلوی نشتی حافظه را بگیرد. برای مثال:

```
function foo() {
  if (true) {
    let x = 10; // x is only accessible in this block
    console.log(x); // 10
  }
  console.log(x); // error: x is not defined
}
```

- استفاده از کلمه کلیدی const برای تعریف متغیرهایی که مقدار آن‌ها تغییر نمی‌کند. این کار باعث می‌شود که متغیرها ثابت شناخته شوند و نتوان آن‌ها را با اشاره گرهای دیگری اشتراک گذاشت. این کار می‌تواند جلوی اشاره گر معلق را بگیرد. برای مثال:

```
const y = {name: "Reza"}; // y is a constant reference to an object
y = {name: "Ali"}; // error: cannot assign to y
y.name = "Ali"; // ok: can modify the properties of the object
```

- استفاده از کلمه کلیدی null برای حذف اشاره گرهایی که دیگر نیازی به آن‌ها نداریم. این کار باعث می‌شود که اشاره گرها به هیچ شیء در حافظه اشاره نکنند و در نتیجه حافظه مربوط به شیء مورد نظر آزاد شود. این کار می‌تواند جلوی نشتی حافظه و اشاره گر معلق را بگیرد. برای مثال:

```
let z = {name: "Sara"}; // z is a reference to an object
z = null; // z is now null and the object is garbage collected
```

- استفاده از ابزارهای تحلیل کد و اشکال‌زدایی (debugging) برای شناسایی و رفع مشکلات نشتی حافظه و اشاره گر معلق. برخی از این ابزارها عبارتند از:
  - TSLint: یک ابزار تحلیل کد است که به شما کمک می‌کند تا کدهای TypeScript خود را با استانداردهای کیفیت و خوانایی مطابقت دهید. این ابزار می‌تواند برخی از مشکلات مربوط به اشاره گرها را نیز شناسایی و اصلاح کند.
  - Chrome DevTools: یک ابزار اشکال‌زدایی است که به شما امکان می‌دهد تا کدهای JavaScriptو TypeScript خود را در مرورگر کروم اجرا، مشاهده و ویرایش کنید. این ابزار می‌تواند به شما نشان دهد که چه مقدار حافظه توسط برنامه شما استفاده می‌شود و کجا ممکن است نشتی حافظه وجود داشته باشد.

در زبان برنامه نویسی TypeScript بازیافت کننده حافظه وجود دارد. این زبان بر پایه جاوا اسکریپت ساخته شده است و از همان مکانیزم بازیافت حافظه یا garbage collection که جاوا اسکریپت استفاده می‌کند بهره می‌برد. این مکانیزم به صورت خودکار اشیاء بدون استفاده را از حافظه حذف می‌کند و بهینه‌سازی حافظه را انجام می‌دهد. برای انجام این کار، یک سیستم GC به صورت مداوم حافظه برنامه را بررسی کرده و سعی می‌کند تا اشیایی که دیگر در برنامه استفاده نمی‌شوند را شناسایی و از حافظه حذف کند GC. در این کار از یک الگوریتم خاص استفاده می‌کند. الگوریتم GC به دو دسته تقسیم می‌شود: دسته اول الگوریتم‌های مبتنی بر نسل و دسته دوم الگوریتم‌های مبتنی بر مارک و سوییپ هستند.

الگوریتم‌های مبتنی بر نسل و الگوریتم‌های مبتنی بر مارک و سوییپ دو دسته از الگوریتم‌های فراابتکاری هستند که برای جستجوی پاسخ بهینه به کار می‌روند. این دو دسته از روش‌های مختلفی برای شناسایی و حذف اشیاء بدون استفاده از حافظه استفاده می‌کنند. در ادامه به توضیح مختصری درباره هر یک از آن‌ها می‌پردازیم:

- الگوریتم‌های مبتنی بر نسل: این دسته از الگوریتم‌ها اشیا حافظه را به چند نسل تقسیم می‌کنند. اشیاء جدید در نسل جوان قرار می‌گیرند، و اشیایی که بعد از چندین دوره GC هنوز هم در حافظه باقی مانده‌اند، در نسل قدیمی جای می‌گیرند. این الگوریتم با توجه به اینکه بیشتر اشیا کوتاه مدت هستند و در نسل جوان قرار می‌گیرند، برای جمع‌آوری زباله، این نسل را بیشتر بررسی می‌کند. این کار باعث می‌شود که زمان اجرای GC کاهش یابد و عملکرد برنامه بهبود یابد. الگوریتم‌های مبتنی بر نسل معمولاً از یک الگوریتم مارک و سوییپ برای جمع‌آوری اشیا در هر نسل استفاده می‌کنند. برای اطلاعات بیشتر می‌توانید به منبع ۱ مراجعه کنید.

- الگوریتم‌های مبتنی بر مارک و سوییپ: این دسته از الگوریتم‌ها ابتدا تمامی اشیا را علامت‌گذاری می‌کنند که در حافظه باقی مانده‌اند. سپس، تمامی اشیایی که علامت نخورده‌اند را حذف می‌کنند. این روش مناسب برای بررسی حافظه‌های بزرگ و پیچیده‌تر مناسب است. این الگوریتم مزیت داشتن سادگی و کارایی را دارد، اما مشکل اصلی آن این است که ممکن است باعث تکه‌تکه شدن حافظه شود و فضای خالی بین اشیا ایجاد کند. برای رفع این مشکل، می‌توان از الگوریتم‌هایی مانند مارک و جمع‌آوری (mark and compact) یا مارک و کپی (mark and copy) استفاده کرد.