Essential TypeScript

Personal Summary

Patrick Bucher

2024-12-25

Contents

This document is a personal and partial summary of Essential TypeScript (3rd Edition) by Adam Freeman. Some examples have been retained, some have been modified, and some have been made up.

1 Understanding TypeScript

TypeScript is a superset of JavaScript. It adds static typing to produce safe and predictable code. It also provides convenience features such as a concise class constructor syntax, and access control keywords. Some TypeScript features are implemented entirely by the compiler,leaving no trace of TypeScript in the resulting JavaScript code.

Working effectively with TypeScript requires understanding JavaScript and its type system. TypeScript allows using modern language features in code that is transformed for execution in older JavaScript runtimes, although not every new feature can be translated to older runtimes.

2 Your First TypeScript Application

The TypeScript compiler `tsc` can be installed using the Node Package Manager (NPM), which is distributed with Node.js. Install it from nodejs.org and make sure to run Node version 18 and NPM version 8 or later:

```
$ node --version
v23.1.0
$ npm --version
10.9.0
```

The TypeScript compiler can be installed (globally, i.e. for all of the current user's projects) in a specific version, e.g. 5.6.3 as follows:

```
$ npm install --global typescript@5.6.3
$ tsc --version
Version 5.6.3
```

It's also recommended to install Git from git-scm.org and a programmer's text editor such as Visual Studio Code from code.visualstudio.com.

To create a new project (e.g. `race`), create a folder, navigate into it, and initialize a Node.js project with default settings:

```
$ mkdir race
$ cd race
$ npm init --yes
```

A file `package.json` has been created, which keeps track of the project's settings and package dependencies.

Tye TypeScript code shall be placed in the `src/` subfolder and be copiled to pure JavaScript code into the `dist/` subfolder. Create a file `tsconfig.json` for this purpose:

```
{
  "compilerOptions": {
    "target": "ES2023",
    "outDir": "./dist",
    "rootDir": "./src",
    "module": "CommonJS"
  }
}
```

This configuration produces JavaScript code according to the ECMAScript2023 specification. The `module` setting defines which module system is being used in the *emitted* JavaScript code, not the module system being used in the TypeScript code! CommonJS is used here, because the resulting code shall be run in Node.js.

The main code file is commonly named `index.ts`, which is to be placed into the `src/` folder:

```
console.clear();
console.log("Hello, World!");
```

To compile the program, run the `tsc` command:

```
$ tsc
```

This produces a file `dist/index.js` that can be run by Node.js:

```
$ node dist/index.js
Hello, World!
```

2.1 Demo Application: Drivers in a Race

As an example application, multiple drivers entering a race shall be modeled in TypeScript. The driver is represented as a class called `Driver`, which is defined in `driver.ts`:

```
export class Driver {
  public id: number;
  public name: string;
  public retired: boolean = false;

  public constructor(id: number, name: string, retired: boolean = false) {
    this.id = id;
    this.name = name;
    this.retired = retired;
  }

  public describe(): string {
    return `${this.id}\t${this.name}\t${this.retired ? "[r]" : "[ ]"}`;
  }
}
```

This code uses features that are not available in JavaScript:

- Fields (`id`, `name`, `retired`) and methods (`describe`) are annotated with a type (`number`, `string`, `boolean`), which is written as a suffix after the colon `:`.

- Fields and methods also have access modifiers (`public`).

TypeScript provides a shorter syntax to initialize the fields of a class that are passed to its constructor. The `Driver` class can be writter more concisely without loosing any functionality as follows:

```
export class Driver {
  // no additional field declaration required

  public constructor(
    public id: number,
    public name: string,
    public retired: boolean = false,
  ) {
    // no manual initialization required
  }

  public describe(): string {
    return `${this.id}\t${this.name}\t${this.retired ? "[r]" : "[ ]"}`;
  }
}
```

The field declaration and initialization is automatically handled by the constructor, whose parameters now also come with access modifiers. The default access modifier is `public`, but here it must be defined for the constructor parameters so that the compiler detects that concise constructor syntax is being used.

Multiple drivers shall be put together in a container class called `Race`, which is defined in `race.ts`:

```
import { Driver } from "./driver";

export class Race {
  private nextId: number = 0;

  constructor(
    public track: string,
    public laps: number,
    public drivers: Driver[] = [],
  ) {}

  addDriver(name: string, retired: boolean = false): number {
    const maxId = Math.max(...this.drivers.map((d) => d.id));
    const newId = maxId + 1;
    this.drivers.push(new Driver(newId, name, retired));
    this.nextId = newId + 1;
```

```
    return newId;
  }

  getDriverById(id: number): Driver {
    return this.drivers.find((d) => d.id === id);
  }

  markRetired(id: number, retired: boolean) {
    const driver = this.getDriverById(id);
    if (driver) {
      driver.retired = retired;
    }
  }
}
```

The drivers are stored in an array that is annotated with the type `Driver[]`—an array of drivers.

Those two classes—`Driver` and `Race`—can interact together as shown in `index.ts`:

```
import { Driver } from "./driver";
import { Race } from "./race";

const drivers = [
  new Driver(1, "Freddie Fuel"),
  new Driver(2, "Eddie Engine"),
  new Driver(3, "Walter Wheel"),
];
const race = new Race("Detroit City Speedwary", 48, drivers);

const lateEntrantId = race.addDriver("Tommy Tardy");
const lateEntrant = race.getDriverById(lateEntrantId);

const crashed = race.getDriverById(2);
race.markRetired(crashed.id, true);

for (const driver of race.drivers) {
  console.log(driver.describe());
}
```

Which produces the following output:

```
$ tsc && node dist/index.js
1   Freddie Fuel    [ ]
2   Eddie Engine    [r]
3   Walter Wheel    [ ]
```

```
4   Tommy Tardy     [ ]
```

There are no type annotations used in the code of `index.ts`. The TypeScript compiler is able to infer the proper types by the context. However, the code can be made easier to read by providing additional type annotations. The programmer can decide to what extent type annotations shall be used to give both the compiler and other programmers hints on the types being used.

The code of `index.ts` can be rewritten as follows with additional type annotations:

```
import { Driver } from "./driver";
import { Race } from "./race";

const drivers: Driver[] = [
  new Driver(1, "Freddie Fuel"),
  new Driver(2, "Eddie Engine"),
  new Driver(3, "Walter Wheel"),
];
const race: Race = new Race("Detroit City Speedwary", 48, drivers);

const lateEntrantId: number = race.addDriver("Tommy Tardy");
const lateEntrant: Driver = race.getDriverById(lateEntrantId);

const crashed: Driver = race.getDriverById(2);
race.markRetired(crashed.id, true);

for (const driver of race.drivers) {
  console.log(driver.describe());
}
```

Instead of storing the drivers in an array and looking them up by filtering for their `id`, storing them in a `Map` will make the lookup easier and faster. The values of `driverMap` can be filtered like an array (`race.ts`):

```
import { Driver } from "./driver";

export class Race {
  private driverMap = new Map<number, Driver>();

  constructor(
    public track: string,
    public laps: number,
    drivers: Driver[] = [],
  ) {
    drivers.forEach((d) => this.driverMap.set(d.id, d));
  }
```

```
  addDriver(name: string, retired: boolean = false): number {
    const maxId = Math.max(...this.driverMap.keys());
    const newId = maxId + 1;
    this.driverMap.set(newId, new Driver(newId, name, retired));
    return newId;
  }

  getDriverById(id: number): Driver {
    return this.driverMap.get(id);
  }

  markRetired(id: number, retired: boolean) {
    const driver = this.getDriverById(id);
    if (driver) {
      driver.retired = retired;
    }
  }

  getDrivers(includeRetired: boolean): Driver[] {
    return [...this.driverMap.values()].filter(
      (d) => !d.retired || includeRetired,
    );
  }
}
```

The otherwise dynamic types of a JavaScript `Map` are restricted for `driverMap` with the type parameter `number` for keys and `Driver` for values, which are defined in angle brackets. Those type hints allow the compiler to know and check the value type of the `Map`: `Driver`, which has a field `retired` of type `boolean`.

Retired drivers can be removed from the map by adding the method `removeRetired` (`race.ts`):

```
removeRetired() {
  this.driverMap.forEach((d) => {
    if (d.retired) {
      this.driverMap.delete(d.id);
    }
  });
}
```

The types of object literals can be described using an object's *shape*: a combination of the property names and types, which can be declared as a new `type` alias:

```
type DriverCounts = {
  total: number;
  active: number;
};
```

A method to return those counts can be annotated with `DriverCounts` as its return type:

```
getDriverCounts(): DriverCounts {
  return {
    total: this.driverMap.size,
    active: this.getDrivers(false).length,
  };
}
```

Those new features can be used as follows (`index.ts`):

```
import { Driver } from "./driver";
import { Race } from "./race";

const drivers: Driver[] = [
  new Driver(1, "Freddie Fuel"),
  new Driver(2, "Eddie Engine"),
  new Driver(3, "Walter Wheel"),
];
const race: Race = new Race("Detroit City Speedwary", 48, drivers);

const lateEntrantId: number = race.addDriver("Tommy Tardy");
const lateEntrant: Driver = race.getDriverById(lateEntrantId);

const crashed: Driver = race.getDriverById(2);
race.markRetired(crashed.id, true);

race.getDrivers(false).forEach((d) => console.log(d.describe()));
console.log(race.getDriverCounts(), "before removing retired");
race.removeRetired();
console.log(race.getDriverCounts(), "after removing retired");
```

Output:

```
1   Freddie Fuel    [ ]
3   Walter Wheel    [ ]
4   Tommy Tardy     [ ]
{ total: 4, active: 3 } before removing retired
{ total: 3, active: 3 } after removing retired
```

2.2 Using a Third-Party Package

TypeScript allows using any JavaScript package with additional static type support. In order to make use of ECMAScript modules, which is the common standard for modules by now, the Node.js project configuration file `package.json` has to be extended by the following setting:

```
"type": "module"
```

The TypeScript compiler settings in `tsconfig.json` have to be modified accordingly, so that the module system defined by the Node.js project is considered:

```
"module": "Node16"
```

The way in wh ich Node.js implements ECMAScript modules requires to use the `.js` (*not* `.ts`!) extension for imports, e.g. in `race.ts`:

```
import { Driver } from "./driver.js";
```

This has to be understood as a reference to the compiled JavaScript file as opposed to the original TypeScript source code file.

The existing application shall be extended by the Inquirer.js library to provide interactive menus. Pure JavaScript libraries can be installed into a TypeScript project as in a plain JavaScript project:

```
npm install inquirer@9.1.4
```

The library can be imported as follows in `index.js`:

```
import inquirer from "inquirer";
```

Since the Inquirer.js project doesn't provide any type information, the TypeScript compiler cannot check its proper usage concerning data types. There are, however, two ways in which such type information can be provided still: First, to describe the types yourself; and second, to use existing type declarations from the Typed project—a repository of type declarations for many JavaScript packages. Such type declarations can be installed as a development dependency as follows (in a slightly different version, though):

```
npm install --save-dev @types/inquirer@9.0.5
```

The Inquirer.js package shall be used to implement an interactive menu. This menu provides different options using different kinds of prompts, e.g. to enter additional drivers to a race, or to mark drivers as retired, which then can be purged from the list.

Each command is represented as an entry in an `enum`, which is a TypeScript feature to group related constants together:

```
enum Commands {
  Add = "Add New Driver",
  Retire = "Retire a Driver",
  Toggle = "Show/Hide Retired Drivers",
  Purge = "Remove Retired Drivers",
  Quit = "Quit",
}
```

The menu is run by the `promptUser` function, which dispatches individual commands to their respective prompt function—or performs the action on its own:

```
function promptUser(): void {
  console.clear();
  displayDrivers();
  inquirer
    .prompt({
      type: "list",
      name: "command",
      message: "Choose Option",
      choices: Object.values(Commands),
    })
    .then((answers) => {
      switch (answers["command"]) {
        case Commands.Toggle:
          showRetired = !showRetired;
          promptUser();
          break;
        case Commands.Add:
          promptAdd();
          break;
        case Commands.Retire:
          if (race.getDriverCounts().active > 0) {
            promptRetire();
          } else {
            promptUser();
          }
          break;
        case Commands.Purge:
          race.removeRetired();
          promptUser();
          break;
      }
    });
}
```

The `inquirer.prompt` function shows a prompt that is configured using a JavaScript object with the following properties:

- `type: "list"`: shows the provided `choices` (see below) as an interactive menu
- `name: "commands"`: assigns a name to the property that will hold the user's choice
- `message: "Choose Option"`: is the actual prompt being shown to the user
- `choices: Object.values(Commnds)`: provides the options for the user to select from a list

The other prompt functions use a different `type` together with different options for their user interaction:

```
function promptAdd(): void {
  console.clear();
  inquirer
    .prompt({
      type: "input",
      name: "add",
      message: "Enter Driver: ",
    })
    .then((answers) => {
      if (answers["add"] !== "") {
        race.addDriver(answers["add"]);
      }
      promptUser();
    });
}

function promptRetire(): void {
  console.clear();
  inquirer
    .prompt({
      type: "checkbox",
      name: "retired",
      message: "Retire Driver: ",
      choices: race
        .getDrivers(showRetired)
        .map((driver) => ({
          name: driver.name,
          value: driver.id,
          checked: driver.retired,
        })),
    })
    .then((answers) => {
      let retiredDrivers = answers["retired"] as number[];
```

```
      race
        .getDrivers(true)
        .forEach((driver) =>
          race.markRetired(
            driver.id,
            retiredDrivers.find((id) => id === driver.id) !== undefined,
          ),
        );
      promptUser();
    });
}
```

Since the compiler cannot figure out the type of `answers["retired"]`, the *type assertion* `as number[]` is used to explicitly tell the compiler that an array of numbers is being used.

The list of drivers is shown using the `displayDrivers` function:

```
function displayDrivers(): void {
  console.log(
    `Race at ${race.track} over ${race.laps} laps. ` +
      `${race.getDriverCounts().active} drivers active.`,
  );
  race
    .getDrivers(showRetired)
    .forEach((driver) => console.log(driver.describe()));
}
```

2.3 Adding Persistence

The application shall be extended with persistent storage of the data. For this purpose the Lowdb package shall be used, which stores data in JSON files. Even though this is a pure JavaScript package, it supplies type information:

```
npm install lowdb@5.1.0
```

Persistence shall be added by the means of inheritance: The `Race` class—with its various operations that manipulate the data—is extended to synchronize the changes with the JSON database on the disk. Currently, `driverMap` is declared as `private`, preventing sub-classes from accessig it. To allow this access, `driverMap` has to be declared as `protected` (`race.ts`):

```
protected driverMap = new Map<number, Driver>();
```

The new sub-class `PersistentRace` then can be implemented as follows (`persistentRace.ts`):

```ts
import { Driver } from "./driver.js";
import { Race } from "./race.js";
import { LowSync } from "lowdb";
import { JSONFileSync } from "lowdb/node";

type schemaType = {
  drivers: {
    id: number;
    name: string;
    retired: boolean;
  }[];
};

export class PersistentRace extends Race {
  private database: LowSync<schemaType>;

  constructor(
    public track: string,
    public laps: number,
    drivers: Driver[] = [],
  ) {
    super(track, laps, []);
    this.database = new LowSync(new JSONFileSync("race.json"));
    this.database.read();
    if (this.database.data == null) {
      this.database.data = { drivers: drivers };
      this.database.write();
      drivers.forEach((driver) => this.driverMap.set(driver.id, driver));
    } else {
      this.database.data.drivers.forEach((driver) =>
        this.driverMap.set(
          driver.id,
          new Driver(driver.id, driver.name, driver.retired),
        ),
      );
    }
  }

  addDriver(name: string, retired: boolean = false): number {
    let result = super.addDriver(name, retired);
    this.storeDrivers();
    return result;
  }
```

```
  markRetired(id: number, retired: boolean): void {
    super.markRetired(id, retired);
    this.storeDrivers();
  }

  removeRetired(): void {
    super.removeRetired();
    this.storeDrivers();
  }

  private storeDrivers() {
    this.database.data.drivers = [...this.driverMap.values()];
    this.database.write();
  }
}
```

The schema used for storing the data persistently is defined using a `type` called `schemaType`, which describes the shape of the object to be stored: an array of drivers.

The extended implementation can be used as follows (`index.ts`):

```
import { PersistentRace } from "./persistentRace.js";

const race: PersistentRace = new PersistentRace("Detroit City Speedwary", 48, drivers);
```

The data is then stored persistently in a file called `race.json`.

3 JavaScript Primer, Part 1

To understand the benefits provided by TypeScript, one has to understand what JavaScript issues it addresses.

For the sake of convenient demonstration with automatic execution of a script upon saving it, the `nodemon` package can be used in a new project called `primer`:

```
mkdir primer
cd primer
npm init --yes
npm install nodemon@3.1.7
touch index.js
npx nodemon index.js
```

JavaScript is similar to many other programming languages in many ways, but it confuses its users with some of its—peculiar, but well-defined—behaviour:

```
let penPrice = 5;
let paperPrice = "5";
if (penPrice == paperPrice) {
  console.log("pen and paper cost the same");
}
console.log(`total price: ${penPrice + paperPrice}`);
```

Output:

```
pen and paper cost the same
total price: 55
```

The operators == and + deal differently with the same types.

In JavaScript, variables are untyped, but values have a type. There are the following built-in types in JavaScript:

- `number`: numeric values, both floating-point and integer values
- `string`: text data
- `boolean`: the values `true` and `false`
- `symbol`: unique constant values
- `null`: a non-existant or invalid reference with the only possible value of `null`
- `undefined`: the type of variables that are uninitialized
- `object`: the type of compound values, made up of primitive and/or other compound values

The type of an expression can be determined by using the `typeof` operator on it, which returns the type name as a `string`:

```
> typeof 5
'number'
> typeof "5"
'string'
> typeof (typeof 5)
'string'
> typeof null
'object'
```

Notice the last example: The type of `null` is `object` instead of `null`. This behaviour is inconsistent, but cannot be changed because a lot of code depends on this (mis)behaviour.

3.1 Type Coercion

An operator being applied to two values of different types needs to coerce one value to the type of the other value. Different operators apply different rules for this process, which is called *type coercion*: The `==` operator applied to a `number` and a `string` converts the `string` to a `number` and then compares the two `number` values:

```
> 3 == "5"
false
> 3 == "3"
true
```

However, the `+` operator applied to a `string` and a value of any other type will first convert the other value to a `string` and then concatenate both `string` values:

```
> "3" + 4
'34'
> 4 + "3"
'43'
```

When aplied to a `number` value and `undefined`, the latter is converted to `NaN`, resulting in an addition that results in `NaN` itself:

```
> 3 + undefined
NaN
```

Such behaviour is erratic, but well-defined and documented. To solve problems arising from those rules, the strict equality operator `===` can be used instead of `==`. Here, both value and type must match:

```
> 3 == "3"
true
> 3 === "3"
false
```

To make sure that numbers are added instead of concatenated, use explicit type conversion:

```
> let x = 3;
> let y = 4;
> let z = "5";
> const addNumbers = (a, b) => Number(a) + Number(b);
> addNumbers(x, y);
7
> addNumbers(y, z);
9
```

Type coercion can be very useful, too: The or-operator `||` converts `null` and `undefined` to `false`, which allows for defining fallback values:

```
> let userChoice = userInput || "Hamburger";
> userChoice
'Hamburger'
```

Unfortunately, not only `null` and `undefined` are coerced into `false`, but also the empty string `""`, the number `0`, and `NaN`:

```
> let defaultInterestRate = 1.5;
> let explicitInterestRate = 0;
> let actualInterestRate = explicitInterestRate || defaultInterestRate;
> actualInterestRate
1.5
```

Here, the fallback to the default value is not desired, because `0` is a perfectly fine value in this context. The *nullish coalescing operator* `??`, which is a rather recent addition to JavaScript, only converts `null` and `undefined` to `false`:

```
> let defaultInterestRate = 1.5;
> let explicitInterestRate = 0;
> let actualInterestRate = explicitInterestRate ?? defaultInterestRate;
> actualInterestRate
0
```

3.2 Functions

Function parameters are untyped, and argument values will be coerced as needed based on the operators being applied to them. Parameters can be given default values so that they don't end up being `undefined` when the function is called with fewer arguments than specified:

```
> const formatCurrency = (currency, amount = 0.0) => `${currency} ${amount}`;
> formatCurrency("CHF", 3.5);
'CHF 3.5'
> formatCurrency("CHF");
'CHF 0'
```

To deal with a variable number of arguments, rest parameters can be used:

```
> const sum = (...xs) => xs.reduce((acc, x) => acc + x, 0);
> sum(3, 4, 5);
12
```

Values such as `undefined` and `NaN` being passed explicitly have to be dealt with programmatically:

```
function mean(...numbers) {
  let actualNumbers = numbers.map((x) => (Number.isNaN(x) ? 0 : Number(x)));
  let sum = actualNumbers.reduce((acc, x) => acc + x, 0);
  return sum / actualNumbers.length;
}
```

3.3 Arrays

JavaScript arrays are dynamically sized and can take up elements of different types. They support various operations:

- operations on single values
  - `push(item)`: adds `item` at the end
  - `pop()`: removes and returns the last item
  - `unshift(item)`: adds `item` at the beginning
  - `shift()`: removes and returns the first item
- operations on the entire array or parts of it
  - `concat(others)`: returns a new array consisting of the original elements and the elements of the passed arrays
  - `join(separator)`: joins the array elements to a string with `separator` in between
  - `sort(compare)`: returns a new array with the original elements in ascending order; an optional `compare` function can be passed
  - `reverse()`: returns a new array with the original elements in reversed order
  - `slice(start, end)`: returns a section of the array from start (inlcusive) to end (exclusive)
  - `splice(index, count)`: removes `count` elements starting from `index`

  - `includes(value)`: returns `true` if the array contains `value`; `false` otherwise

- higher-order functions
  - `every(predicate)`: returns `true` if `predicate` returns `true` for all elements
  - `some(predicate)`: returns `true` if `predicate` returns `true` for at least one element
  - `filter(predicate)`: returns an array consisting of the elements for which `predicate` returns `true`
  - `find(predicate)`: returns the first value for which `predicate` returns `true`
  - `findIndex(predicate)`: returns the first index of the value for which `predicate` returns `true`
  - `forEach(callback)`: calls the `callback` function on each element
  - `map(callback)`: returns an array consisting of the return values of `callback` called on every element
  - `reduce(callback, start)`: combines the array elements using the `callback` function and an optional `start` value

Examples:

```
> let numbers = [1, 2, 3]
> numbers.push(4)
> numbers.unshift(0)
> numbers.pop()
4
> numbers.shift()
0

> [1, 2, 3].concat([4, 5, 6], [7, 8, 9])
[ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
> [1, 2, 3, 4, 5].join(" < ")
'1 < 2 < 3 < 4 < 5'
> [5, 2, 3, 4, 1].sort()
[ 1, 2, 3, 4, 5 ]
> [1, 2, 3, 4, 5].reverse()
[ 5, 4, 3, 2, 1 ]
> [2, 4, 6, 8].slice(2, 4)
[ 6, 8 ]
> [1, 2, 3, 4, 5].splice(2, 3)
[ 3, 4, 5 ]
> [2, 4, 6, 8].includes(5)
false

> [1, 1, 2, 3, 5, 8, 13, 21].every(x => x % 2 == 0)
false
> [1, 1, 2, 3, 5, 8, 13, 21].some(x => x % 2 == 0)
```

```
true
> [1, 1, 2, 3, 5, 8, 13, 21].filter(x => x % 2 == 0)
[ 2, 8 ]
> [1, 1, 2, 3, 5, 8, 13, 21].find(x => x % 2 == 0)
2
> [1, 1, 2, 3, 5, 8, 13, 21].findIndex(x => x % 2 == 0)
2
> [1, 1, 2, 3, 5, 8, 13, 21].forEach(x => console.log(`x=${x}`))
x=1
x=1
x=2
x=3
x=5
x=8
x=13
x=21
> [1, 1, 2, 3, 5, 8, 13, 21].map(x => x * 2)
[ 2,  2,  4,  6, 10, 16, 26, 42 ]
> [1, 1, 2, 3, 5, 8, 13, 21].reduce((acc, x) => acc + x, 0)
54
```

An array can be passed to a function expecting rest parameters by applying the spread operator `...` to it:

```
function sumUp(...numbers) {
  return numbers.reduce((acc, x) => acc + x, 0);
}

const numbers = [1, 1, 2, 3, 5, 8];
console.log(sumUp(...numbers)); // 20
```

The spread operator can also be used to concatenate arrays:

```
> let xs = [2, 4, 6, 8, 10];
> let ys = [3, 6, 9, 12, 15];
> let zs = [...xs, ...ys];
> zs
[ 2, 4, 6,  8, 10, 3, 6, 9, 12, 15 ]
```

Arrays can be unpacked by applying destructuring assignment to them:

```
let words = ["read", "write", "think", "morning"];
let [first, second] = words;
```

```
let [, , third] = words;
let [, , ...lastTwo] = words;
console.log(`first: ${first}`);
console.log(`second: ${second}`);
console.log(`third: ${third}`);
console.log(`lastTwo: ${lastTwo}`);
```

Output:

```
first: read
second: write
third: think
lastTwo: think,morning
```

3.4 Objects

JavaScript objects are collections of properties, which have a name and a value. Objects can be expressed using a literal syntax:

```
> let alice = { name: "Alice", age: 52 };
> let bob = { name: "Bob", age: 46 };
```

Properties can be accessed using the dot operator, added by assignment, and removed using the `delete` keyword:

```
> let aliceAge = alice.age;
> alice.place = "Sweden";
> delete alice.name;
> alice
{ age: 52, place: 'Sweden' }
```

Reading a property that doesn't exist returns `undefined`. The *optional chaining operator* `?.` will stop the evaluation once `null` or `undefined` is reached. This is especially useful in combination with the `??` operator:

```
> bob?.place?.population ?? 0;
0
```

The spread operator can be applied to objects for destructuring:

```
> let { name, age } = bob;
> name
'Bob'
> age
46

> let olderBob = {...bob, age: 60, disease: "Diabetes" };
> olderBob
{ name: 'Bob', age: 60, disease: 'Diabetes' }

> let { bobsName, ...otherProperties } = olderBob;
> otherProperties
{ name: 'Bob', age: 60, disease: 'Diabetes' }
```

An object can be turned into its JSON representation using `JSON.stringify`:

```
> JSON.stringify(bob);
'{"name":"Bob","age":46}'
> JSON.stringify(alice);
'{"age":52,"place":"Sweden"}'
```

JavaScript objects support getters and setters:

```
let stock = {
  item: "Beer",
  _price: 1.25,
  _quantity: 100,

  set price(newPrice) {
    this._price = newPrice;
  },

  get price() {
    return this._price;
  },

  set quantity(newQuantity) {
    this._quantity = newQuantity;
  },

  get quantity() {
    return this._quantity;
  },
```

```
  get worth() {
    return this._price * this._quantity;
  },
};

console.log(`stock worth before: ${stock.worth}`);
stock.price *= 1.03; // inflation
stock.quantity += 100; // hoarding
console.log(`stock worth after: ${stock.worth}`);
```

Output:

```
stock worth before: 125
stock worth after: 257.5
```

- The properties `price` and `quanitity` have getter and setter methods; values can be read from them and be assigned to them. They are backed internal properties `_price` and `_quantity`.
- The property `worth` is a computed property that only has a `get` method and therefore cannot be overwritten.

3.4.1 this

`this` refers to different objects depending on how a function or method using it is called. Consider a function that outputs a label and a value:

```
function output(value) {
  console.log(`${this.label}=${value}`);
}

label = "x";
output(13);
```

Output:

```
x=13
```

The `this` object refers to the *global object* by default. Properties can be set to the global object by simple assignment (as `label = "x"` above) without using the `var`, `let`, or `const` keyword—except in strict mode!

Functions and methods are objects in JavaScript, which have their own properties and methods in turn. The above function call is actually a convenience syntax for the following invocation:

```
output.call(global, 13);
```

The global object is called `global` in Node.js and `window` or `self` in a browser context; the latter having an object called `document` representing the DOM.

When a function belongs to an object and is invoked as a method, the `this` keyword refers to the surrounding object:

```
let object = {
  label: "y",
  output(value) {
    console.log(`${this.label}=${value}`);
  },
};

label = "x";
object.output(13); // same as: object.output.call(object, 13);
```

Output:

```
y=13
```

However, if the method is called outside of its object context, `this` refers to the global object:

```
let object = {
  label: "y",
  output(value) {
    console.log(`${this.label}=${value}`);
  },
};

label = "x";
let output = object.output;
output(13); // same as: output.call(global, 13);
```

Output:

```
x=13
```

The `this` keyword can be bound explitly and persistently to an object using the method's `bind` method:

```
let object = {
  label: "y",
  output(value) {
    console.log(`${this.label}=${value}`);
  },
};

label = "x";
object.output = object.output.bind(object);
object.output(13);
let output = object.output;
output(13);
```

Output:

```
y=13
y=13
```

Now `output` being called as a stand-alone function also uses `object` as its `this` reference: `this.label` has the value of `"y"` in both cases.

An arrow function returned from a method works differently in respect to its `this` reference. Consider a following example, in which the function creating the output is an arrow function being returned from a method:

```
let object = {
  label: "y",
  getOutput() {
    return (value) => console.log(`${this.label}=${value}`);
  },
};
```

Depending on how the function is called, `this` refers to different objects:

```
label = "x";

let outputReference = object.getOutput(); // invoked on object
outputReference(11);

let getOutput = object.getOutput;
let outputStandAlone = getOutput(); // invoked on global
outputStandAlone(11);
```

Output:

```
y=11
x=11
```

In the first usage, the `getOutput` is called in the context of `object`, binding `this` to `object`. In the second usage, the `getOutput` method is called in the global context, binding `this` to `global`.

An arrow functio has no `this` reference of its own! It instead works its way up the scope until it finds a `this` reference instead—either reaching the surrounding or the global object.

4 JavaScript Primer, Part 2

A JavaScript object inherits its properties and methods from another object known as its *prototype*. The links of prototypes form an inheritance chain. By default, an object defined by a literal has the prototype `Object`, which defines the following methods related to prototypes:

- `getPrototypeOf`: returns an object's prototype
- `setPrototypeOf`: sets an object's prototype
- `getOwnPropertyNames`: returns the names of the properties defined on an object itself (excluding inherited properties from its prototype)

The following example defines two objects and checks if they share the same prototype:

```
let alice = {
  name: "Alice",
  age: 52,
};

let bob = {
  name: "Bob",
  age: 47,
};

let aliceProto = Object.getPrototypeOf(alice);
let bobProto = Object.getPrototypeOf(bob);
console.log(`same prototype? ${aliceProto === bobProto}`);
console.log(`alice's properties: ${Object.getOwnPropertyNames(alice)}`);
console.log(`bob's properties: ${Object.getOwnPropertyNames(bob)}`);
```

Output:

```
same prototype? true
alice's properties: name,age
bob's properties: name,age
```

It is possible to define operations shared among objects directly on its default prototype `Object`:

```
let alice = {
  name: "Alice",
  age: 52,
};

let bob = {
  name: "Bob",
  age: 47,
};

let aliceProto = Object.getPrototypeOf(alice);

aliceProto.toString = function () {
  return `${this.name} is ${this.age} years old.`;
};

console.log(`alice: ${alice}`);
console.log(`bob: ${bob}`);

let product = {
  name: "Candy Bar",
  price: 1.25,
};

console.log(`product: ${product}`);
```

Output:

```
alice: Alice is 52 years old.
bob: Bob is 47 years old.
product: Candy Bar is undefined years old.
```

The method `toString` is overwritten directly on the prototype of the object `alice`, which is `Object`. Therefore, `toString` is also overwritten for the object `product`, which has no `age` property.

A better option is to define a common and custom prototype shared among the relevant objects explicitly, but leaving `Object` untouched:

```
let alice = {
  name: "Alice",
  age: 52,
};
```

```
let bob = {
  name: "Bob",
  age: 47,
};

let ProtoPerson = {
  toString: function () {
    return `${this.name} is ${this.age} years old.`;
  },
};

Object.setPrototypeOf(alice, ProtoPerson);
Object.setPrototypeOf(bob, ProtoPerson);

console.log(`alice: ${alice}`);
console.log(`bob: ${bob}`);

let product = {
  name: "Candy Bar",
  price: 1.25,
};

console.log(`product: ${product}`);
```

Output:

```
alice: Alice is 52 years old.
bob: Bob is 47 years old.
product: [object Object]
```

4.1 Constructor Functions and Prototype Chaining

Objects can not only be created using literal syntax, but also with *constructor functions*, which can apply additional logic upon the object's creation. Constructor functions have capitalized names by convention and are invoked using the `new` keyword, setting the `this` parameter to the newly instantiated object.

The constructor function's `prototype` property provides access to its prototype object, to which methods can be attached:

```
let Person = function (name, age) {
  this.name = name;
  this.age = age;
```

```
};

Person.prototype.toString = function () {
  return `${this.name} is ${this.age} years old.`;
};

let alice = new Person("Alice", 52);
let bob = new Person("Bob", 47);

console.log(alice.toString());
console.log(bob.toString());
```

Output:

```
Alice is 52 years old.
Bob is 47 years old.
```

Constructor functions can be chained by connecting their prototypes. A constructor further down the chain can invoke the constructor function higher up the chain using its `call` method:

```
let Person = function (name, age) {
  this.name = name;
  this.age = age;
};

Person.prototype.toString = function () {
  return `${this.name} is ${this.age} years old`;
};

let Employee = function (name, age, percentage, salary) {
  Person.call(this, name, age);
  this.percentage = percentage;
  this.salary = salary;
};

Employee.prototype.toString = function () {
  let salary = this.percentage * 0.01 * this.salary;
  return `${Person.prototype.toString.call(this)} and earns ${salary}`;
};

Object.setPrototypeOf(Employee.prototype, Person.prototype);

let alice = new Employee("Alice", 52, 75.0, 120000);
let bob = new Person("Bob", 47);
```

```
console.log(alice.toString());
console.log(bob.toString());
```

Output:

```
Alice is 52 years old and earns 90000
Bob is 47 years old
```

The `toString` method of the prototype further up the chain has to be invoked explicitly using `Person.prototype.toString.call`, passing it the `this` reference of the calling object.

The `instanceof` operator determines whether or not an object is part of a prototype chain. Using the example from above:

```
console.log(`is Alice a Person? ${alice instanceof Person}`);
console.log(`is Alice an Employee? ${alice instanceof Employee}`);
console.log(`is Bob a Person? ${bob instanceof Person}`);
console.log(`is Bob an Employee? ${bob instanceof Employee}`);
```

Output:

```
is Alice a Person? true
is Alice an Employee? true
is Bob a Person? true
is Bob an Employee? false
```

Static methods can be defined by assigning properties to the constructor function:

```
Person.output = function (...people) {
  people.forEach((p) => console.log(p.toString()));
};

Person.output(alice, bob);
```

4.2 Classes

Recent versions of JavaScript support classes, which are implemented using prototypes underneath. Keywords such `class`, `extends`, `constructor`, `super`, and `static` known from mainstream object-oriented languages such as C# and Java are mere syntactic sugar for the concepts described above with prototypes. Private members are created with a `#` prefix.

The example implementing an inheritance relationship between `Person` and `Employee` from before can be expressed using classes as follows:

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  toString() {
    return `${this.name} is ${this.age} years old`;
  }

  static output(...people) {
    people.forEach((p) => console.log(p.toString()));
  }
}

class Employee extends Person {
  constructor(name, age, percentage, salary) {
    super(name, age);
    this.percentage = percentage;
    this.salary = salary;
  }

  toString() {
    return `${super.toString()} and earns ${this.#salary()}`;
  }

  #salary() {
    return this.percentage * (this.salary / 100.0);
  }
}

let alice = new Person("Alice", 52);
let bob = new Employee("Bob", 47, 90, 120000);
Person.output(alice, bob);
```

Output:

```
Alice is 52 years old
Bob is 47 years old and earns 108000
```

Notice that for `bob` the `toString` method of `Employee` is called, even though the static `output` method is defined on `Person`.

4.3 Iterators and Generators

An iterator provides a function called `next`, which returns a sequence of objects containing a `value` and a `done` property, the latter indicating whether or not the sequence has been exhausted:

```
class Sequence {
  constructor(step, n) {
    this.step = step;
    this.n = n;
    this.value = 0;
    this.i = 0;
  }

  next() {
    this.value += this.step;
    this.i++;
    return {
      value: this.value,
      done: this.i > this.n,
    };
  }
}

let rowOfSix = new Sequence(6, 10);
let result = rowOfSix.next();
while (!result.done) {
  console.log(result.value);
  result = rowOfSix.next();
}
```

Output:

```
6
12
18
24
30
36
42
48
54
60
```

A generator is a function declared using an asterisk character (*) that returns an intermediate result using the `yield` keyword and l ater continues its execution upon the next call. The state is maintained implicitly by the runtime rather than explicitly by the programmer.

The number sequence from above can be expressed using a generator as follows:

```
function* createSequence(step, n) {
  let value = 0;
  for (let i = 0; i < n; i++) {
    value += step;
    yield value;
  }
}
```

Generators can be consumed in a `for`/`of` loop or using the spread operator:

```
for (let x of createSequence(6, 5)) {
  console.log(x);
}
[...createSequence(3, 4)].forEach((x) => console.log(x));
```

Output:

```
6
12
18
24
30
3
6
9
12
```

For objects that provide sequences of items, a special proprty called `Symbol.iterator` can be provided as a generator, allowing the object being used as a sequence in loops and with spread operations:

```
class Sequence {
  constructor(step, n) {
    this.step = step;
    this.n = n;
  }

  *[Symbol.iterator]() {
    let value = 0;
```

```
    for (let i = 0; i < this.n; i++) {
      value += this.step;
      yield value;
    }
  }
}

for (let x of new Sequence(7, 5)) {
  console.log(x);
}

[...new Sequence(10, 5)].forEach((x) => console.log(x));
```

Output:

```
7
14
21
28
35
10
20
30
40
50
```

4.4 Collections

An object's properties are key/value pairs. The keys and values of an object called `obj` can be obtained using the methods `Object.keys(obj)` and `Object.values(obj)`, respectively. Given an object and one of its keys, the value can be obtained using square bracket notation: `let value = obj[key]`.

```
let mouse = {
  name: "Pixie",
  legs: 4,
  food: "cheese",
};

for (let key of Object.keys(mouse)) {
  console.log(`${key}: ${mouse[key]}`);
}
```

Output:

```
name: Pixie
legs: 4
food: cheese
```

Objects only support strings for keys. The `Map` type is more general in as far as it allows for any types to be used as a key. A `Map` provides the following operations (among others):

- `set(key, value)`: stores the `value` under `key`
- `get(key)`: returns the value stored under `key`
- `keys()`: returns an iterator over the keys
- `values()`: returns an iterator over the values
- `entries()`: returns an iterator over `[key, value]` arrays

Using a `Symbol` as a key avoids collisions, which could happen when the keys are derived from the stored value:

```
class Person {
  constructor(name, age) {
    this.id = Symbol();
    this.name = name;
    this.age = age;
  }
}

let alice = new Person("Alice", 52);
let bob = new Person("Bob", 47);

let people = new Map();
people.set(alice.id, alice);
people.set(bob.id, bob);

for (let [id, { name, age, ..._ }] of people.entries()) {
  console.log(`${id.toString()}: ${name} (${age})`);
}
```

Note that a `Symbol` is rather abstract and not intended for output:

```
Symbol(): Alice (52)
Symbol(): Bob (47)
```

A `Set` stores unique values and supports, among others, the following operations:

- `add(value)`: adds a value to the set, if it is not already contained
- `entries()`: returns an iterator over the set's values in insertion order

- `has(value)`: returns true, if `value` is contained in the set, and `false` otherwise
- `size`: returns the number of elements in the set

```
let additions = [
  [3, 5],
  [1, 4],
  [4, 4],
  [2, 1],
];

let sums = new Set();

for (let [a, b] of additions) {
  sums.add(a + b);
}
sums.forEach((s) => console.log(s));
```

Both the additions of `[3, 5]` and `[4, 4]` result in the sum of `8`, but this particular result is stored only once in the set of sums.

Output:

```
8
5
3
```

4.5 Modules

Modules allow it to break an application into manageable junks. Most JavaScript projects use either one of the following module systems:

1. *ECMAScript modules* are the official standard built into recent runtimes.
2. *CommonJS modules* are provided by Node.js and used to be the de facto standard.

Since ECMAScript modules can deal with CommonJS modules, most projects targeting recent runtime versions should use ECMAScript modules. In Node.js, the type of module can be configured by convention or by configuration:

- by convention: using file name extensions
  - Use the `.mjs` extension for ECMAScript module files.
  - Use the `.cjs` extension for CommonJS module files.
- by configuration: using the `type` setting in `package.json`
  - Use the `"type": "module"` setting for ECMAScript modules.

– Use the `"type": "commonjs"` setting for CommonJS modules.

The following module defines functions for rounding values with different granularities (`round.js`):

```
export default function roundTo(value, granularity) {
  const factor = 1.0 / granularity;
  const scaledUp = value * factor;
  const rounded = Math.round(scaledUp);
  const scaledDown = rounded / factor;
  return scaledDown;
}

export function roundToNickels(value) {
  return roundTo(value, 0.05);
}

export function roundToDimes(value) {
  return roundTo(value, 0.1);
}
```

Functions defined in modules are private to the module by default and need to be made available to other modules using the `export` keyword. The `default` keyword denotes a single feature of the module that is imported by default without having to use an explicit name.

The `round.js` module can be used in `index.js` as follows:

```
import round, { roundToNickels, roundToDimes } from "./round.js";

console.log(round(10.0 / 3.0, 0.01));
console.log(roundToNickels(10.0 / 3.0));
console.log(roundToDimes(10.0 / 3.0));
```

The default feature of the module is imported using an alternative name: `round` instead of `roundTo`. The other two functions are imported using their names.

If the module code resides in the same project, relative paths are used, starting with `./` for modules located in the same directory, or starting with `../` for modules located in a directory higher up in the hierarchy.

If external modules are used, such as those located in `node_modules`, the import path starts with the module name: the name of the module directory located in `node_modules`.

5 Using the TypeScript Compiler

To demonstrate the usage of the TypeScript compiler, a new project called `tools` shall be created:

```
mkdir tools
cd tools
npm init --yes
```

Two dependencies—the TypeScript compiler and a tool for automatic compilation—shall be installed:

```
npm install --save-dev typescript@6.2.1
npm install --save-dev tsc-watch@5.6.3
```

This will store the specified dependencies in the `devDependencies` section of the `package.json` file—unlike dependencies installed without the `--save-dev` option, which are listed in the `dependencies` section.

A basic compiler configuration file `tsconfig.json` shall be created:

```
{
  "compilerOptions": {
    "target": "ES2022",
    "outDir": "./dist",
    "rootDir": "./src"
  }
}
```

TypeScript files located in the `src` folder will be compiled using the ECMAScript 2022 standard, and the resulting JavaScript code will be put into the `dist` folder.

A file `src/index.ts` shall be created:

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

greet("TypeScript");
```

The code can be compiled and executed as follows:

```
tsc
node dist/index.js
```

```
Output:
```

```
Hello, TypeScript!
```

Version numbers in `packge.json` (both in `dependencies` and `devDependencies`) can be specified using different rules:

- `6.2.1`: the exact version
- `*`: any version
- `>6.2.1`, `⩾6.2.1`, `<6.2.1`, `⩽6.2.1`: versions higher/lower (exclusive/inclusive) than the stated version number
- `~6.2.1`: same major and minor version, but allows for other patch version
- `^6.2.1`: same major version, but allows for other minor or patch version

The most commonly used NPM commands are:

- `npm install`: install the packages specified in `package.json`
- `npm install package@version`: install a single package with a specific version (saved to `dependencies` in `package.json`)
- `npm install --save-dev package@version`: install a single package with a specific version (saved to `devDependencies` in `package.json`)
- `npm install --global package@version`: install a single package with a specific version globally
- `npm list`: list local packages and their dependencies
- `npm run`: execute a script defined in `package.json`
- `npx package`: runs the code of a package

The `packag.json` and `package-lock.json` files shall be *included* in, the folder `node_modules` shall be *excluded* from version control

To control which files shall be compiled when running `tsc`, different options can be specified in `tsconfig.json`:

- `files`: overrides the standard behaviour by specifying a set of files to be compiled
- `include`/`exclude`: define files by patterns to be included/excluded
- `compileOnSave`: hints to the text editor that `tsc` shall be run upon saving a file, if set to `true`

To figure out which files are considered for compilation, run:

- globally: `tsc --listFiles`
- locally: `npx tsc --listFiles`

By default, the TypeScript compiler emits JavaScript code even when it encounters errors. This resulting code contains potential errors. To demonstrate this problematic behaviour, extend `index.ts` with the following function call:

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

greet("TypeScript");
greet(100);
```

Compilation will fail:

```
$ tsc
src/index.ts:6:7 - error TS2345: Argument of type 'number' is not assignable to parameter of type 'string'
```

But the following JavaScript code *was* emitted:

```
function greet(whom) {
    console.log(`Hello, ${whom}!`);
}
greet("TypeScript");
greet(100);
```

In this case, the resulting code is unproblematic, because numbers can be printed just as strings. However, this defeats the purpose of using TypeScript. This behaviour can be changed by setting the `noEmitOnError` setting to `true` in `tsconfig.json`:

```
{
  "compilerOptions": {
    "target": "ES2022",
    "outDir": "./dist",
    "rootDir": "./src",
    "noEmitOnError": true
  }
}
```

For automatic recompilation, run `tsc` with the `--watch` flag:

```
tsc --watch
```

However, the emitted JavaScript code still needs to be executed manually. Use the `tsc-watch` package instealled earlier for automatic execution after compilation:

```
npx tsc-watch --onsuccess 'node dist/index.js'
```

To avoid typing this command again at the beginning of the next session, encode it as a script in `package.json`:

```
{
  "name": "tools",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "start": "tsc-watch --onsuccess 'node dist/index.js'"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": "",
  "devDependencies": {
    "tsc-watch": "^6.2.1",
    "typescript": "^5.6.3"
  }
}
```

Which then can be run as follows:

```
npm start
```

To specify the targeted version of the JavaScript language being emitted by the compiler, set the `target` option in `tsconfig.json`. By default `ES5` is used. See the documentation of the `target` option for all allowed values. The other compiler options are documented on the TSConfig Reference page.

If a feature is used in TypeScript code that isn't available in the targeted JavaScript runtime, the compiler will report an error. The problem can be resolved by either targeting a later JavaScript standard or by changing the type definitions using the `lib` setting in `tsconfig.json`, which is an array of library names.

To target different type of module implementations, use the `module` setting in `tsconfig.json`.

See the list of compiler options to further control the compilation process.

6 Testing and Debugging TypeScript

Because the written TypeScript code and the emitted JavaScript code do not correlate on a line-by-line basis, *source maps* have to be generated in order to use a debugger on the TypeScript code. Set the `sourceMap` setting to `true` in `tsconfig.json` in order to create `.map` files alongside the `.js` files in the `dist` folder.

To use a debugger on TypeScript code, check your editor's documentation, or use the `debugger` keyword together with Node.js:

```
node inspect dist/index.js
```

See the Debugging Node.js documentation for further instructions.

A linter inspects the code for style issues and emits warnings. Install the following packages to lint your code:

```
npm install --save-dev eslint@<9.0.0
npm install --save-dev @typescript-eslint/parser@8.15.0
npm install --save-dev @typescript-eslint/eslint-plugin@8.15.0
```

Create a configuration file `.eslintrc` to the project's root folder:

```
{
  "root": true,
  "ignorePatterns": ["node_modules", "dist"],
  "parser": "@typescript-eslint/parser",
  "parserOptions": {
    "project": "./tsconfig.json"
  },
  "plugins": ["@typescript-eslint"],
  "extends": [
    "eslint:recommended",
    "plugin:@typescript-eslint/eslint-recommended",
    "plugin:@typescript-eslint/recommended"
  ]
}
```

The linter can be run using the following command:

```
npx eslint .
```

Change the code in `src/index.ts` to use a variable using a `let` statement:

```
function greet(whom: string): void {
  let message = `Hello, ${whom}!`;
  console.log(message);
}

greet("TypeScript");
```

The linter will warn you about using `let` where `const` would be more appropriate:

```
2:7  error  'message' is never reassigned. Use 'const' instead  prefer-const
```

Once `let` is replaced by `const`, the message won't be shown again when linting the code.

The rightmost information of the error message (`prefer-const`) ist the name of the rule, which can be disabled by adding in to the `rules` section of the `eslintrc` file:

```
{
  ...
  "rules": {
    "prefer-const": 0
  }
}
```

See the ESLint documentation for further configuration options.

6.1 Unit Testing

When writing TypeScript, both the test and the production code are compiled to JavaScript. It's the duty of the TypeScript compiler to verify the use of TypeScript features; the unit tests only verify JavaScript code.

Install the Jest test framework as follows:

```
npm install --save-dev jest@29.7.0
npm install --save-dev ts-jest@29.2.5
npm install --save-dev @types/jest@29.5.14
```

The `ts-jest` package automatically compiles TypeScript files before the tests are executed. The `@types/jest` package contains TypeScript definitions for the Jest API.

Create a test concifugartion file `jest.config.js` in the project's root folder with the following content:

```
module.exports = {
  roots: ["src"],
  transform: { "^.+\\.tsx?$": "ts-jest" },
};
```

The source code will be looked up in the `src` folder. Files with the `.ts` and `.tsx` extension should be processed by `ts-jest`.

For a file called `foo.ts`, a unit test is defined in a file called `foo.test.ts`.

Create a new file `src/rounding.ts` with the following code to be tested:

```
export function roundTo(value: number, granularity: number): number {
  const factor = 1.0 / granularity;
  const scaledUp = value * factor;
  const rounded = Math.round(scaledUp);
  const scaledDown = rounded / factor;
  return scaledDown;
}
```

Create a new file `src/rounding.test.ts` with the following test code:

```
import { roundTo } from "./rounding";

test("check round to cents", () => {
  expect(roundTo(10.0 / 3.0, 0.01)).toBe(3.33);
});

test("check round to nickels", () => {
  expect(roundTo(10.0 / 3.0, 0.05)).toBe(3.35);
});

test("check round to dimes", () => {
  expect(roundTo(10.0 / 3.0, 0.1)).toBe(3.3);
});
```

The `test` function provided by the Jest framework expects both a test description as a string and a function performing the actual test. The `expect` function expects a function result, which than can be further processed by a *matcher function* such as `toBe`.

The `import` statement does *not* require a `.js` extension, because internally `CommonJS` is used as the module system.

The tests can be run as follows:

```
npx jest
```

An interactive mode of test running is supported as well:

```
npx jest --watchAll
```

This allows the tests to be quickly executed again (e.g. by pressing `f` to run failed tests, or by pressing `[Enter]` to run all tests again).

A complete list of Jest matcher functions can be found in the Jest documentation.

7 Understanding Static Types

The following code examples can be run in a project called `types`, similar to the `tools` project from the above section.

In JavaScript, variables have no types, but values do. JavaScript's `typeof` keyword returns the type of an expression's resulting value:

```
console.log(`${typeof "hello"}`);
console.log(`${typeof 12.5}`);
console.log(`${typeof true}`);
console.log(`${typeof { some: "thing" }}`);
```

Output:

```
string
number
boolean
object
```

To restrict variables, function parameters, and function return values to certain types, TypeScript supports *type annotations*:

```
function discount(amount: number, percentage: number): number {
  const factor: number = (100 - percentage) / 100.0;
  const discounted: number = amount * factor;
  return discounted;
}

console.log(discount(50, 2.5));
```

The TypeScript compiler is able to infer the types of expressions. In this example, the result is converted to a `string`:

```
function discount(amount: number, percentage: number) {
  const factor: number = (100 - percentage) / 100.0;
  const discounted: number = amount * factor;
  return discounted.toFixed(2);
}

const originalPrice = 50;
const discountRate = 1.25;
const discountedPrice = discount(originalPrice, discountRate);
console.log(discountedPrice);
```

The `toFixed` method turns the `number` into a `string`, which is returned from the `discount` function. Since no variable nor return types have been annotated, the compiler has to figure out the types.

Extend the `tsconfig.json` file by adding the `declaration` setting:

```
{
  "compilerOptions": {
    …
    "declaration": true
  }
}
```

This will create a file called `index.d.ts` alongside `index.js` in the dist folder:

```
declare function discount(amount: number, percentage: number): string;
declare const originalPrice = 50;
declare const discountRate = 1.25;
declare const discountedPrice: string;
```

This is helpful to reveal what types the compiler has inferred.

The `any` type can be used if *all* types are allowed. In this case, the programmer must take care that the types are used correctly, and the compiler won't help. If no type parameters are provided, the TypeScript compiler fills in `any` implicitly.

This implicit usage of `any` can be disabled using the `noImplicitAny` compiler option in `tsconfig.json`:

```
{
  "compilerOptions": {
    …
    "noImplicitAny": true
  }
}
```

If different types are acceptable for a variable or function argument, the allowed types can be narrowed down using a *type union* rather than just allowing `any` type.

A type union is a list of types, separated by a bar, e.g. `string | number` to allow both strings and numeric values.

The discount example from before is extended using a flag indicating whether or not the result shall be formatted as a currency string, either returning a `string` or a `number`.

```
function discount(
  amount: number,
  percentage: number,
  format: boolean,
```

```
): string | number {
  const factor: number = (100 - percentage) / 100.0;
  const discounted: number = amount * factor;
  if (format) {
    return `$${discounted.toFixed(2)}`;
  }
  return discounted;
}

const originalPrice = 50;
const discountRate = 1.25;
const discountedPrice = discount(originalPrice, discountRate, true);
console.log(discountedPrice);
```

The operations allowed on the `discountedPrice` variable are the intersection of operations allowed on the `string` and on the `number` type, i.e. only `toString()` is allowed.

If the programmer knows more than the compiler, a *type assertion* using the `as` keyword can be used to make sure TypeScript treats a variable as having a specific type. Using the `discount` function from above:

```
const discountedPriceFormatted: string = discount(99.9, 5.0, true) as string;
const discountedPriceRaw: number = discount(99.9, 5.0, false) as number;
console.log(discountedPriceFormatted, discountedPriceRaw.toFixed(2));
```

Notice that no conversion is performed by using the `as` keyword; instead, TypeScript uses this information to determine which operations are allowed on the variables.

A different approach is to use *type guards* using the `typeof` JavaScript keyword:

```
const discountedPrice = discount(99.9, 5.0, false);
if (typeof discountedPrice === "string") {
  console.log(discountedPrice);
} else if (typeof discountedPrice === "number") {
  console.log(`$${discountedPrice.toFixed(2)}`);
}
```

The TypeScript compiler figures out that in the second branch, `discountedPrice` is a number, therefore permitting the usage of its `toFixed` method.

The `never` type can be used to ensure that type guards are being used exhaustively, i.e. that no type remains unhandled:

```
const discountedPrice = discount(99.9, 5.0, false);
if (typeof discountedPrice === "string") {
  console.log(discountedPrice);
} else if (typeof discountedPrice === "number") {
```

```
  console.log(`$${discountedPrice.toFixed(2)}`);
} else {
  let impossible: never = discountedPrice;
  console.log(`unexpected type for value ${impossible}`);
}
```

If the `discount` function were to be declared with the type `string | number | object`, the assignment of `discountedPrice` to `impossible` *would* actually happen, causing an error.

Unlike `any`, the type `unknown` can only be assigned to another type together with a type assertion. The following code using `any` compiles:

```
const discounted: any = discount(99.9, 5.0, false);
const discountedPrice: number = discounted;
console.log(discountedPrice);
```

When using `unknown` instead of `any` instead, this code would fail:

```
const discounted: unknown = discount(99.9, 5.0, false);
const discountedPrice: number = discounted;
console.log(discountedPrice);
```

Error:

```
error TS2322: Type 'unknown' is not assignable to type 'number'.
```

It works however together with a type assertion:

```
const discounted: unknown = discount(99.9, 5.0, false);
const discountedPrice: number = discounted as number;
console.log(discountedPrice);
```

Since the values `null` and `undefined` are legal values for all types, the TypeScript compiler won't complain when using them:

```
function discount(
  amount: number,
  percentage: number,
  format: boolean,
): string | number {
  if (amount == 0.0) {
    return null;
  }
  const factor: number = (100 - percentage) / 100.0;
  const discounted: number = amount * factor;
  if (format) {
```

```
    return `$${discounted.toFixed(2)}`;
  }
  return discounted;
}

const discountedPrice: string | number = discount(99.9, 5.0, false);
console.log(discountedPrice);
```

This behaviour can be changed by activating *strict null checks* in `tsconfig.json`:

```
{
  "compilerOptions": {
    …
    "strictNullChecks": true
  }
}
```

The code from above now cuases this error:

```
Type 'null' is not assignable to type 'string | number'.
```

The `null` type has to be included in the type union so that the program compiles and runs again:

```
function discount(
  amount: number,
  percentage: number,
  format: boolean,
): string | number | null {
  if (amount == 0.0) {
    return null;
  }
  const factor: number = (100 - percentage) / 100.0;
  const discounted: number = amount * factor;
  if (format) {
    return `$${discounted.toFixed(2)}`;
  }
  return discounted;
}

const discountedPrice: string | number | null = discount(99.9, 5.0, false);
console.log(discountedPrice);
```

Note that `typeof null` returns `"object"`, so the variable `discountedPrice` in the example above needs to be compared to the *value* `null`; a type guard won't help.

If the value `null` cannot be returned, i.e. by passing an `amount` argument not equal to `0.0` in the example above—the programmer knows more than the compiler—a *non-null assertion* can be applied, which is a `!` character after the expression producing a non-`null` value:

```
const discountedPrice: string | number = discount(99.9, 5.0, false)!;
console.log(discountedPrice);
```

Note that the type `null` is still part of the function's type union, but no longer of the variable's.

The also uses the `!` character, but right after the variable's name.

If the compiler cannot detect the assignment of a value to a variable, but the programmer is sure there *is* a value assigned, the *definitive assignment assertion* (also using the `!` character, but right after the variable's name) can be used to prevent compilation errors.

This code fails to compile:

```
let percentage: number;
eval("percentage = 5.0;");
const discountedPrice: string | number = discount(99.9, percentage, false)!;
console.log(discountedPrice);
```

Error:

```
Variable 'percentage' is used before being assigned.
```

The TypeScript compiler cannot peek inside the `eval` function, but the programmer can:

```
let percentage!: number;
eval("percentage = 5.0;");
const discountedPrice: string | number = discount(99.9, percentage, false)!;
console.log(discountedPrice);
```

8 Using Functions

Unlike many other languages, JavaScript does *not* support function overloading. In JavaScript, the last function defined with a name will be called. TypeScript, being more strict, throws an error:

```
function discount(amount: number, percentage: number): number {
  const factor = (100 - percentage) / 100.0;
  return amount * factor;
}

function discount(amount: number, value: number): number {
  return amount - value;
}
```

Output:

```
src/index.ts(1,10): error TS2393: Duplicate function implementation.
src/index.ts(6,10): error TS2393: Duplicate function implementation.
```

The issue can prevented using the following strategies:

1. Implement a single function that accepts both kinds of parameters. (Here: a percentage *and* an absolute value to be discounted.)
2. Implement two functions with distinct names. (Here: a `discountPercentage` and a `discountAbsolute` function.)

The first approach makes the code harder to read, whereas the second approach adds clarity by introducing more precise names.

In JavaScript, if too few arguments are passed to a function, the remainder parameters have the value `undefined`. If too many arguments are passed, they are collected in the `arguments` array.

TypeScript, however, is stricter than JavaScript and does *not* allow a function to be called with a different number of arguments.

```
function discount(amount: number, percentage: number): number {
  const factor = (100 - percentage) / 100.0;
  return amount * factor;
}

console.log(discount(100, 5.0)); // same number of arguments
console.log(discount(100)); // too few arguments
console.log(discount(100, 5.0, 1)); // too many arguments
```

Output:

```
src/index.ts(7,13): error TS2554: Expected 2 arguments, but got 1.
src/index.ts(8,32): error TS2554: Expected 2 arguments, but got 3.
```

The `noUnusedParameters` compiler option can be activated to produce an error if a function expects parameters that aren't actually used:

```
{
  "compilerOptions": {
    …
    "noUnusedParameters": true
  }
}
```

In this example, the `absolute` parameter is expected, but never used:

```
function discount(amount: number, percentage: number, absolute: number): number {
  const factor = (100 - percentage) / 100.0;
  return amount * factor;
}

console.log(discount(100, 5.0, 10.0));
```

Output:

```
src/index.ts(1,55): error TS6133: 'absolute' is declared but its value is never read.
```

Optional parameters must be declared after the required parameters with the suffix `?`:

```
function discount(
  amount: number,
  percentage: number,
  absolute?: number,
): number {
  const factor = (100 - percentage) / 100.0;
  if (absolute) {
    amount -= absolute;
  }
  return amount * factor;
}

console.log(discount(100, 5));
console.log(discount(100, 5, 10));
```

Output:

```
95
85.5
```

To avoid explicit checks whether or not an optional argument has been provided, a fallback value can be used:

```
function discount(
  amount: number,
  percentage: number,
  absolute?: number,
): number {
  const factor = (100 - percentage) / 100.0;
```

```
  return (amount - (absolute || 0)) * factor;
}
```

However, since a fallback value can be used, it can be provided using a *default-initialized* parameter, being declared without a question mark:

```
function discount(
  amount: number,
  percentage: number,
  absolute: number = 0,
): number {
  const factor = (100 - percentage) / 100.0;
  return (amount - absolute) * factor;
}

console.log(discount(100, 5));
console.log(discount(100, 5, 10));
```

Even though the parameter `absolute` was declared *without* a `?`, it sill is an optional parameter and therefore needs to be listed after the required parameters.

A variable number of arguments can be provided using a rest parameter, declared using an ellipsis (`...`), which collects zero, one, or multiple arguments provided after the optional parameters:

```
function discount(
  amount: number,
  percentage: number,
  absolute: number = 0,
  ...fees: number[]
): number {
  const factor = (100 - percentage) / 100.0;
  const totalFees = fees.reduce((acc, e) => acc + e, 0);
  return (amount - absolute + totalFees) * factor;
}

console.log(discount(100, 5));
console.log(discount(100, 5, 10));
console.log(discount(100, 5, 10, 1));
console.log(discount(100, 5, 10, 1, 2));
console.log(discount(100, 5, 10, 1, 2, 3));
```

Output:

```
95
85.5
```

```
86.45
88.35
91.19999999999999
```

If a function expecting a default-initialized parameter is called with the argument `null`, its default value is used instead, as if the function had be called without passing an argument for this default-initialized parameter.

The `strictNullChecks` compiler option disables the usage of `null` and `undefined` without declaring the parameters using an according type union, e.g. `number | null`. The possibility of `null` being passed then needs to be accounted for programmatically:

```
function discount(
  amount: number,
  percentage: number,
  absolute: number | null = 0,
): number {
  const factor = (100 - percentage) / 100.0;
  return (amount - absolute) * factor;
}

console.log(discount(100, 5, null));
```

Output:

```
error TS18047: 'absolute' is possibly 'null'.
```

Which can be fixed using an explicit `null` check:

```
function discount(
  amount: number,
  percentage: number,
  absolute: number | null = 0,
): number {
  const factor = (100 - percentage) / 100.0;
  if (absolute == null) {
    absolute = 0;
  }
  return (amount - absolute | 0) * factor;
}

console.log(discount(100, 5, null));
```

If no return type is annotated for a function that returns values of different types, the TypeScript compiler will automatically infer a type union for the function's return type. If the `declaration` setting is enabled, the inferred type unions can be seen in the according declarations file, e.g. in `dist/index.d.ts` for the code in `src/index.ts`. The `discount` function can either return a `number` or a `string`, depending on the value of the `format` parameter:

```
function discount(amount: number, percentage: number, format: boolean) {
  const factor = (100 - percentage) / 100.0;
  const result = amount * factor;
  if (format) {
    return result.toFixed(2);
  }
  return result;
}
```

The return type is inferred as `string | number` in `dist/index.d.ts`:

```
declare function discount(amount: number, percentage: number, format: boolean): string | number;
```

A JavaScript function that never explicitly returns a value using the `return` keyword will return `undefined` implicitly when called. Implicit returns can be prevented by enabling the `noImplicitReturns` compiler option in `tsconfig.json`:

```
{
  "compilerOptions": {
    …
    "noImplicitReturns": true
  }
}
```

When activated, each execution path of a function must return a value explicitly, otherwise the compiler throws an error:

```
function greet(whom: string): string {
  if (whom == "") {
    return `Hello, ${whom}!`;
  }
}
```

Output:

```
error TS2366: Function lacks ending return statement and return type does not include 'undefined'.
```

Once an explicit return is introduced for the case that `whom` is the empty string, the function will be compiled:

```
function greet(whom: string): string {
  if (whom !== "") {
    return `Hello, ${whom}!`;
  }
  return "";
}
```

A function's return type can depend on the type of its parameters. However, this relationship remains implicit and requires reading the code to be uncovered, as in the following function:

```
function multiply(x: number | string, y: number): number | string {
  if (typeof x === "number") {
    return x * y;
  }
  let result = "";
  for (let i = 0; i < y; i++) {
    result = `${result} ${x}`;
  }
  return result.trim();
}

console.log(multiply(3, 5));
console.log(multiply("oh", 5));
```

Output:

```
15
oh oh oh oh oh
```

A *type overload* describes such a relationship using different function declarations, expressing valid type combinations:

```
function multiply(x: number, y: number): number;
function multiply(x: string, y: number): string;
function multiply(x: number | string, y: number): number | string {
  if (typeof x === "number") {
    return x * y;
  }
  let result = "";
  for (let i = 0; i < y; i++) {
    result = `${result} ${x}`;
  }
  return result.trim();
}
```

```
let product: number = multiply(3, 5);
let output: string = multiply("oh", 5);
console.log(product);
console.log(output);
```

When the function `multiply` is called with an argument of type `number` for the parameter `x`, the compiler infers that the resulting value must be of type `number`, too; no type assertion is needed when assigning the return value to a variable of type `number`.

The `asserts` keyword indicates that a function performs a type assertion on a parameter value, which either passes or throws an exception. Consider the following function:

```
function increment(x: number | null): number {
  return x + 1;
}
```

Which cannot be compiled:

```
error TS18047: 'x' is possibly 'null'.
```

However, the code compiles when using an assert function:

```
function increment(x: number | null): number {
  assertNotNull(x);
  return x + 1;
}

function assertNotNull(x: any): asserts x {
  if (x == null) {
    throw new Error("x is null");
  }
}
```

Assertions can also be used for specific types, e.g. to make sure that a parameter is actually numeric:

```
function increment(x: number | string | null): number {
  assertIsNumeric(x);
  return x + 1;
}

function assertIsNumeric(x: any): asserts x is number {
  if (typeof x != "number") {
    throw new Error("x is not numeric");
```

```
  }
}
```

9 Using Arrays, Tuples, and Enums

TypeScript allows to restrict the types of the elements an array can cake up using type annotations:

```
const names: string[] = ["Dilbert", "Alice", "Wally"];
const ages: number[] = [42, 37, 53];
```

A type union must be placed within parentheses:

```
const measurements: (number | string)[] = [97, 37, "cold"];
```

Written as `number | string[]` (i.e. without parentheses), the type union would denote the type to be either a single number or an array of strings; hence the parentheses around the type union.

Note: Such a type union means that *each* element of an array can be either a `number` or a `string`, and not that *all* elements have to bei either numbers or strings. Therefore, values of different types can be mixed in the array.

The TypeScript compiler infers type unions for arrays based on the values being used upon initialization:

```
const values = [13, "hello", 42];
```

When the `declaration` compiler option is activated, the following declarations are produced (`dist/index.d.ts`):

```
declare const values: (string | number)[];
```

If an array is initialized empty, the type `any` is inferred:

```
const values = [];
```

Declarations:

```
declare const values: any[];
```

9.1 Tuples

Tuples are fixed-length arrays, in which every element can have a different type. Under the hood, tuples are implemented as regular JavaScript arrays:

```
let dilbert: [string, number] = ["Dilbert", 42];
let alice: [string, number] = ["Alice", 37];
let wally: [string, number] = ["Wally", 53];

let engineers: [string, number][] = [dilbert, alice, wally];
```

A type annotation is always required for tuples, otherwise they would be treated as regular arrays. Tuples can be used together with JavaScript's array features, such as accessing individual elements using square brackets and an index.

Optional elements (following non-optional elements) are supported using the `?` suffix, turning the type of such al element to a type union with `undefined`, e.g. `number?` becomes `number | undefined`.

Tuples can also use rest elements, making the number of elements more flexible—and thereby defeating the point of tuples.

Consider the following example in which an optional element denotes the best friend, and the rest parameters denote more friends:

```
let dilbert: [string, number, string?, ...string[]] = ["Dilbert", 42];
let alice: [string, number, string?, ...string[]] = ["Alice", 37, "Amy"];
let wally: [string, number, string?, ...string[]] = ["Wally", 53, "Ashok", "Boss", "Joe"];

let engineers: [string, number, string?, ...string[]][] = [dilbert, alice, wally];
```

The type declarations look as follows:

```
declare let dilbert: [string, number, string?, ...string[]];
declare let alice: [string, number, string?, ...string[]];
declare let wally: [string, number, string?, ...string[]];
declare let engineers: [string, number, string?, ...string[]][];
```

9.2 Enums

Enums are groups of fixed values that can be accessed by a name. An enum can be declared using the `enum` keyword:

```
enum Position {
  Engineer,
  Manager,
  Trainee,
}

let employees: [string, Position][] = [
  ["Dilbert", Position.Engineer],
  ["Pointy-Haired Boss", Position.Manager],
  ["Ashok", Position.Trainee],
];

employees.forEach((employee: [string, Position]) => {
  switch (employee[1]) {
    case Position.Engineer:
      console.log(`${employee[0]} works on the product.`);
      break;
    case Position.Manager:
      console.log(`${employee[0]} manages the team.`);
      break;
    case Position.Trainee:
      console.log(`${employee[0]} lends a helping hand.`);
      break;
  }
});
```

The label can be accessed as a `string` using square bracket notation:

```
enum Position {
  Engineer = 1,
  Manager = 2,
  Trainee = 3,
}

let position: Position = Position.Engineer;
let label: string = Position[position];
console.log(label);
```

Output:

```
Engineer
```

Enums are implemented by the TypeScript compiler, which automatically assigns a `number` value to every name, as shown in the declarations file:

```
declare enum Position {
  Engineer = 0,
  Manager = 1,
  Trainee = 2,
}
declare let employees: [string, Position][];
```

The numbers can be assigned manually. It's a good practice to either assign numbers manually to either all or none of the names to avoid duplications. The first enum `Position` defines numbers for all names, the second enum `State` defines numbers only for two names:

```
enum Position {
  Engineer = 1,
  Manager = 2,
  Trainee = 3,
}

enum State {
  Waiting,
  Running = 3,
  Sleeping,
  Starting,
  Stopping = 4,
}
```

The names with missing values are enumerated automatically, which fails in the second case, as the declaration shows:

```
declare enum Position {
    Engineer = 1,
    Manager = 2,
    Trainee = 3
}
declare enum State {
    Waiting = 0,
    Running = 3,
    Sleeping = 4,
    Starting = 5,
    Stopping = 4
}
```

Both `State.Sleeping` and `State.Stopping` have the value `4`, which creates a conflict:

```
console.log(State.Sleeping == State.Stopping);
```

Output:

```
true
```

This is clearly not what the programmer intended.

It's possible to use `string` instead of `number` values:

```
enum Country {
  Switzerland = "CH",
  Germany = "DE",
  France = "FR",
}

console.log(Country.Switzerland);
```

Output:

```
CH
```

Two distinct enums cannot be compared to one another, even if they share the underlying values:

```
enum Institution {
  Insurance,
  Bank,
  Court,
}

enum Furniture {
  Chair,
  Bank,
  Table,
}

console.log(Institution.Bank == Furniture.Bank);
```

Output:

```
error TS2367: This comparison appears to be unintentional because the types 'Institution' and 'Furniture' have
```

Enums are implemented as JavaScript objects, unless they are declared using the `const` keyword, in which case all the references to an enum are inlined. However, const enums cannot be accessed using their labels:

```
const enum State { Running, Sleeping, Waiting, Starting, Stopping }

let state: State = State.Waiting;
let label: string = State[state];
```

Output:

```
error TS2476: A const enum member can only be accessed using a string literal.
```

9.3 Literal Value Types

A literal value type defines a set of values that can be used in a certain place, e.g. to be assigned to a variable. Syntactically similar to type unions, literal values instead of types are used:

```
let trafficLight: "red" | "yellow" | "green" = "red";
trafficLight = "yellow";
trafficLight = "green";
trafficLight = "black";
```

The initial and the two subsequent assignments are valid, but the third is not:

```
error TS2322: Type '"black"' is not assignable to type '"red" | "yellow" | "green"'.
```

Literal value types are most useful when applied to function parameters:

```
function exp(value: number, power: 2 | 3): number {
  if (power == 2) {
    return value * value;
  }
  return value * value * value;
}
```

It's possible to assign overlapping values to variables that are restricted by different sets of values:

```
let twos: 2 | 4 | 6 | 8 = 4;
let threes: 3 | 6 | 9 | 12 = 9;

twos = 6;
threes = 6;
twos = 3;
```

The first two reassignments are valid, but the third is not:

```
error TS2322: Type '3' is not assignable to type '2 | 4 | 6 | 8'.
```

Literal value types can be mixed with actual types in type unions:

```
let value: string | 1 | 2 | 3 = "ok";
value = 1;
value = 2;
value = 3;
value = "whatever";
```

Type overrides can also be applied to literal value types:

```
function designate(rank: 1 | 2 | 4): string;
function designate(rank: 3 | 5): number;
function designate(rank: 1 | 2 | 3 | 4 | 5): number | string {
  switch (rank) {
    case 1:
      return "winner";
    case 2:
      return "first loser";
    case 4:
      return "missed the podium";
    default:
      return rank;
  }
}
```

It's also possible to use literal value types with string templates:

```
function greet(
  name: "Alice" | "Bob" | "Mallory",
): `Hello, ${"Alice" | "Bob" | "Mallory"}` {
  return `Hello, ${name}`;
}
```

9.4 Type Aliases

Type definitions are often used at multiple places, making it tedious to use and especially change them:

```
let dilbert: [string, number, boolean] = ["Dilbert", 42, true];
let alice: [string, number, boolean] = ["Alice", 37, true];
let wally: [string, number, boolean] = ["Wally", 52, false];
let employees: [string, number, boolean][] = [dilbert, alice, wally];
```

Using a *type alias*, the type can have a name assigned, which then can be re-used:

```
type employee = [string, number, boolean];
let dilbert: employee = ["Dilbert", 42, true];
let alice: employee = ["Alice", 37, true];
let wally: employee = ["Wally", 52, false];
let employees: employee[] = [dilbert, alice, wally];
```

10 Working with Objects

The object's *shape* is the combination of its property names and types. With the `declarations` compiler option activated, the inferred shapes of the used objects can be seen in the declarations file.

In the following example, all three objects share the `name` and `age` property:

```
let dilbert = { name: "Dilbert", age: 42, role: "Engineer" };
let alice = { name: "Alice", age: 37, role: "Engineer" };
let wally = { name: "Wally", age: 57, lazy: true };

let engineers = [dilbert, alice, wally];

for (let engineer of engineers) {
  console.log(`${engineer.name} is ${engineer.age} years old.`); // ok
  console.log(`${engineer.name} works as a ${engineer.role}`); // error
}
```

TypeScript infers that the `name` and `age` properties are available in all three objects, and therefore the first `console.log` statement is valid. However, the second isn't, because it tries to access the `role` property, which is only defined on two of the three objects:

```
error TS2339: Property 'role' does not exist on type …
```

Optional properties can be defined with the `?` suffix. In order to make use of optional properties, additional type guards using `in` have to be used to ensure that an optional property is actually available. For optional methods, it has to be checked that they are not `undefined` before being called:

```
type employee = {
  name: string;
  age: number;
  salary: number;
  role?: string;
```

```
  lazy?: boolean;
  calculateBonus?(salary: number): number;
};

let dilbert: employee = {
  name: "Dilbert",
  age: 42,
  salary: 100000,
  role: "Engineer",
  calculateBonus: (salary) => salary * 0.1,
};
let alice: employee = {
  name: "Alice",
  age: 37,
  salary: 90000,
  role: "Engineer",
  calculateBonus: (salary) => salary * 0.15,
};
let wally: employee = {
  name: "Wally",
  age: 57,
  salary: 80000,
  lazy: true,
  calculateBonus: (salary) => {
    if ("lazy" in this) {
      return 0;
    } else {
      return salary * 0.01;
    }
  },
};

let engineers: employee[] = [dilbert, alice, wally];

for (let e of engineers) {
  console.log(`${e.name} is ${e.age} years old.`);
  if ("role" in e) {
    console.log(`${e.name} works as a ${e.role}`);
  }
  if (e.calculateBonus) {
    console.log(`${e.name} gets a bonus of ${e.calculateBonus(e.salary)}.`);
  }
}
```

In the above example, both `role` and `lazy` are optional properties. Their access is surrounded by `in` checks. The optional method `calculateBonus` is checked to be defined before being invoked, otherwise the compiler would throw an error.

When using a union of shape types, the properties common to both shape types can be used without any further checks:

```
type Employee = {
  id: number;
  name: string;
  salary: number;
};

type Product = {
  id: number;
  name: string;
  price: number;
};

let dilbert: Employee = { id: 745, name: "Dilbert", salary: 100000 };
let alice: Employee = { id: 931, name: "Alice", salary: 90000 };
let stapler: Product = { id: 4529, name: "stapler", price: 8.35 };
let chair: Product = { id: 7826, name: "chair", price: 249.99 };

let assets: (Product | Employee)[] = [dilbert, stapler, alice, chair];

for (let asset of assets) {
  console.log(`${asset.id}: ${asset.name}`);
}
```

In order to make use of other properties, they can either be checked for availability using the `in` keyword—or by applying a *type predicate* to it, which are functions that assert that an expression is of a certain type:

```
type Employee = {
  id: number;
  name: string;
  salary: number;
};

type Product = {
  id: number;
  name: string;
  price: number;
};
```

```
let dilbert: Employee = { id: 745, name: "Dilbert", salary: 100000 };
let alice: Employee = { id: 931, name: "Alice", salary: 90000 };
let stapler: Product = { id: 4529, name: "stapler", price: 8.35 };
let chair: Product = { id: 7826, name: "chair", price: 249.99 };

let assets: (Product | Employee)[] = [dilbert, stapler, alice, chair];

function isEmployee(expr: any): expr is Employee {
  return "salary" in expr;
}

function isProduct(expr: any): expr is Product {
  return "price" in expr;
}

for (let asset of assets) {
  if (isEmployee(asset)) {
    console.log(`${asset.id}: ${asset.name} earns ${asset.salary}`);
  } else if (isProduct(asset)) {
    console.log(`${asset.id}: ${asset.name} costs ${asset.price}`);
  }
}
```

By convention, type predicates follow the name `is[Type]`.

10.1 Type Intersections

A type intersection combines two types to a new type that is only satisfied by values that conform to both original types. For object shapes, the intersection of two shapes requires that a value provides all the properties defined by either shape:

```
type Product = {
  id: number;
  name: string;
  price: number;
};

type Stock = {
  id: number;
  quantity: number;
};

type StockedProduct = Product & Stock;
```

```
let stock: StockedProduct[] = [
  { id: 1, name: "Stapler", price: 8.90, quantity: 17 },
  { id: 2, name: "Chair", price: 215.99, quantity: 3 },
  { id: 3, name: "Lamp", price: 89.95, quantity: 7 },
];

stock.forEach((i) => {
  const value: number = i.price * i.quantity;
  console.log(`${i.id}) ${i.name}:\t${value.toFixed(2)}`);
});
```

Output:

```
1) Stapler: 151.30
2) Chair:   647.97
3) Lamp:    629.65
```

The types `Product` and `Stock` have the `id` property in common; the `name` and `price` property of `Product` as well as the `quantity` property of `Stock` are only defined by one of the types.

To satisfy the `StockedProduct` type, which is an intersection of `Product` and `Stock`, objects need to define *all* four properties, which then can be safely used.

When multiple types of an intersection define the same property name, the intersected type uses an intersection of their types. If the types are identical, this particular type is used. If different types are involved, the intersection of that type is used. If no useful intersection can be found, e.g. because `number` and `string` have nothing in common, the `never` type is inferred, which makes it impossible to use the type. For methods, it is a good practice to consult the declarations file to figure out what implementation in terms of types has to be provided.

11 Working with Classes and Interfaces

In JavaScript, constructor functions can be used to create objects consistently. Unfortunately, constructor functions are treated in TypeScript like any other function, and the compiler cannot infer a type other than `any`:

```
const Employee = function (id: number, name: string) {
  this.id = id;
  this.name = name;
};

Employee.prototype.describe = function (): string {
```

```
  return `${this.id}: ${this.name}`;
};

let employees = [
  new Employee(1, "Dilbert"),
  new Employee(2, "Alice"),
  new Employee(3, "Wally"),
];
```

Output:

```
'new' expression, whose target lacks a construct signature, implicitly has an 'any' type.
```

TypeScript neglects support for constructor functions in favor of classes, with which the `Employee` type from above can be expressed as follows:

```
class Employee {
  id: number;
  name: string;

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

  describe(): string {
    return `${this.id}: ${this.name}`;
  }
}

let employees: Employee[] = [
  new Employee(1, "Dilbert"),
  new Employee(2, "Alice"),
  new Employee(3, "Wally"),
];
```

The objects in the `employees` array now have a proper type, which allows them to be tested using the `instanceof` keyword.

11.1 Access Control and Inheritance

In addition to JavaScripts `#` operator for private properties, TypeScript supports the access control keywords `public` (default), `private`, and `protected`. However, those are only enforced during compilation, but not in the emitted JavaScript code.

Properties defined as `readonly` can only be assigned in the constructor and only read afterwards.

Inheritance is implemented using the `extends` keyword. A subclass needs to invoke the constructor of its superclass using the `super` keyword.

The following example shows how those concepts are being used together:

```
class Person {
  private readonly id: number;
  public name: string;
  protected active: boolean;

  constructor(id: number, name: string, active: boolean) {
    this.id = id;
    this.name = name;
    this.active = active;
  }

  identify(): number {
    return this.id;
  }
}

class Employee extends Person {
  public position: string;

  constructor(id: number, name: string, active: boolean, position: string) {
    super(id, name, active);
    this.position = position;
  }
}

class Customer extends Person {
  private revenue: number;
  public segment: "b2b" | "b2c";

  constructor(
    id: number,
    name: string,
    active: boolean,
    segment: "b2b" | "b2c",
  ) {
    super(id, name, active);
    this.revenue = 0;
    this.segment = segment;
```

```
  }

  book(revenue: number) {
    if (this.active) {
      this.revenue += revenue;
    } else {
      throw new Error("Cannot book revenue on inactive customer.");
    }
  }
}

let people = [
  new Person(1, "Patrick", true),
  new Employee(2, "John", true, "Engineer"),
  new Customer(3, "Jill", true, "b2c"),
];
```

The code required for constructing an object is usually rather repetitive:

- The class defines a couple of properties.
- The constructor accepts some of those properties as parameters.
- The property values are set to the arguments being passed to the constructor.

TypeScript extends the constructor syntax by allowing visibility keywords for constructor parameters. Those parameters are then automatically turned into properties, and their values will be assigned to those properties.

The example from above can be simplified as follows using extended constructor syntax:

```
class Person {
  constructor(
    private readonly id: number,
    public name: string,
    protected active: boolean,
  ) {}

  identify(): number {
    return this.id;
  }
}

class Employee extends Person {
  constructor(
    id: number,
    name: string,
```

```
    active: boolean,
    public position: string,
  ) {
    super(id, name, active);
  }
}

class Customer extends Person {
  private revenue: number;

  constructor(
    id: number,
    name: string,
    active: boolean,
    public segment: "b2b" | "b2c",
  ) {
    super(id, name, active);
    this.revenue = 0;
  }

  book(revenue: number) {
    if (this.active) {
      this.revenue += revenue;
    } else {
      throw new Error("Cannot book revenue on inactive customer.");
    }
  }
}

let people = [
  new Person(1, "Patrick", true),
  new Employee(2, "John", true, "Engineer"),
  new Customer(3, "Jill", true, "b2c"),
];
```

Notice that the constructor body remains empty for the class `Person`. The class `Employee`, however, still calls the constructor if its superclass explicitly. Since the `revenue` property of the `Customer` class cannot be passed to the constructor, the property remains explicitly defined and assigned.

Getter and setter methods can be defined using the `get` and `set` keyword. If only a `get` method is defined, the property becomes a read-only property. Usually, getter and setter methods have a *backing* field which stores the value being accessed. However, this is optional, and the value can be stored in a different way for a setter, or be calculated for a getter method.

The `accessor` keyword defines a property with an optional initial value, for which getter and setter

methods are automatically generated.

The following example demonstrates the usage of those features:

```
class Person {
  private jobs: string[] = new Array();

  constructor(
    private id: number,
    private firstName: string,
    private lastName: string,
  ) {}

  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  }

  set job(newJob: string) {
    this.jobs.push(newJob);
  }

  get job(): string {
    if (this.jobs.length == 0) {
      throw new Error(`${this.fullName} has no job.`);
    }
    return this.jobs[this.jobs.length - 1];
  }

  accessor active: boolean = true;

  get description(): string {
    return `${this.id}: ${this.fullName} [${this.active}]`;
  }
}

const johnSmith = new Person(1, "John", "Smith");
johnSmith.job = "System Administrator";
johnSmith.active = false;
console.log(johnSmith.description);
```

11.1.1 Abstract Classes

A class defined with the `abstract` keyword cannot be instantiated, but extended by other classes. Any method declared as `abstract` needs to be implemented by the subclass.

The concrete methods of an abstract class can call abstract methods, which need to be implemented by the subclasses so that the method call can be resolved.

The following example defines an abstract class `Animal`, which delegates species-specific operations to the concrete subclassess:

```
abstract class Animal {
  constructor(public name: string) {}

  describe(): string {
    return `I'm ${this.name} the ${this.species()} and I make «${this.noise()}».`;
  }

  abstract species(): string;
  abstract noise(): string;
}

class Horse extends Animal {
  constructor(name: string) {
    super(name);
  }

  species(): string {
    return "Horse";
  }

  noise(): string {
    return "Neigh";
  }
}

class Cat extends Animal {
  constructor(name: string) {
    super(name);
  }

  species(): string {
    return "Cat";
  }

  noise(): string {
    return "Meow";
  }
}
```

```
let animals = [new Horse("Betty"), new Cat("Dimka")];
animals.forEach((a) => console.log(a.describe()));
```

Output:

```
I'm Betty the Horse and I make «Neigh».
I'm Dimka the Cat and I make «Meow».
```

Notice that in the above example the TypeScript compiler infers the type union `Horse | Cat` for the `animals` array. Annotating `animals` with the type `Animal[]` makes sense here, because the compiler doesn't infer that only the API common to those two classes is used (via its abstract superclass).

11.1.2 Interfaces

The shapes of objects that are based on a class can be described using *interfaces*. They are very similar to shape types, but declared using the `interface` keyword. Interfaces can both define properties and methods.

A class uses the `implements` keyword to declare that it provides all the properties and methods an interface declares. Unlike inheritance, which only allows for one `extends` declaration, `implements` can list multiple interfaces.

Multiple `interface` declarations within the same file are merged into one interface.

The following example defines a hierarchy of classes and interfaces:

```
interface Shape {
  circumference(): number;
  area(): number;
}

interface Named {
  name: string;
}

interface Describable {
  color?: string;
  describe(): string;
}

abstract class Rectangular implements Shape, Named {
  public name: string;
```

```
  abstract circumference(): number;
  abstract area(): number;
}

class Square extends Rectangular {
  constructor(public side: number) {
    super();
  }

  circumference(): number {
    return 4 * this.side;
  }

  area(): number {
    return this.side * this.side;
  }
}

class Rectangle extends Square implements Describable {
  constructor(
    side: number,
    public otherSide: number,
    public color?: string,
  ) {
    super(side);
    this.name = "Rectangle";
  }

  circumference(): number {
    return 2 * this.side + 2 * this.otherSide;
  }

  area(): number {
    return this.side * this.otherSide;
  }

  describe(): string {
    return `${this.name} of ${this.side}x${this.otherSide}`;
  }
}

let shapes: Shape[] = [
  new Rectangle(3, 4),
  new Rectangle(2, 3, "green"),
```

```
  new Square(5),
];

shapes.forEach((s) => {
  let description: string;
  if ("describe" in s) {
    description = (s as Describable).describe();
  } else if ("name" in s && s.name) {
    description = (s as Named).name;
  } else {
    description = "Unknown shape";
  }
  console.log(
    `${description} with circumference of ${s.circumference()} and area of ${s.area()}`,
  );
});
```

- The `Shape` interface defines two methods: `circumference` and `area`, which both return a `number`.
- The `Named` interface defines a property: `name` of type string.
- The `Describable` interface defines an optional property `color` and a method `describe`; both of type `string`.
- The abstract class `Rectangular` implements both `Shape` and `Named` interface. It needs to implement the two methods of `Shape`, but only provides abstract methods, which must then be implemented by a concrete class.
- The `Square` class extends the abstract class `Rectangular` and therefore needs to implement both `circumference` and `area`.
- The `Rectangular` class extends `Square` and implements another interface: `Describable`. It, therefore, has to implement three methods: `circumference`, `area`, and `describe`.
- The `shapes` array (type `Shape[]`) uses the most common denominator of the three instances stored as its type.

Notice that the `instanceof` operator is of no use when testing for interfaces, because interfaces only exist up to compile time, but not in the JavaScript code actually executed. Therefore, type checks for interfaces have to be implemented in a clumsier way.

Output:

```
Rectangle of 3x4 with circumference of 14 and area of 12
Rectangle of 2x3 with circumference of 10 and area of 6
Unknown shape with circumference of 20 and area of 25
```

11.1.3 Dynamic Properties

JavaScript allows for properties to be used dynamically, whereas TypeScript restricts properties to those defined explicitly. An *index signature* allows for dynamic properties within TypeScript code with restricted types.

The property names can be of type `string` or `number`, but the property value can be of any type. An index signature is defined as follows, e.g. with `string` property names and `number` property values:

```
[propertyName: string]: number;
```

Notice the literal `propertyName` defined within square brackets.

The following example defines a class `Dimensions` that allows for dynamic properties:

```
class Product {
  dimensions: Dimensions;

  constructor(
    public name: string,
    public inStock: boolean,
  ) {
    this.dimensions = new Dimensions();
  }

  addDimension(name: string, value: number) {
    this.dimensions[name] = value;
  }
}

class Dimensions {
  [propertyName: string]: number;
}

let monitor: Product = new Product("Monitor", true);
monitor.addDimension("height", 30.5);
monitor.addDimension("width", 55.3);
monitor.addDimension("weight", 5.4);
monitor.addDimension("price", 399.99);

for (let property of Object.keys(monitor.dimensions)) {
  console.log(`${property}:\t${monitor.dimensions[property]}`);
}
```

To prevent access to undefined properties, activate the `noUncheckedIndexedAccess` compiler option.

12 Using Generic Types

Generics allow the use of type parameters in place of specific types. The specific type is filled in upon usage, allowing existing code to work with types that do not exist yet.

The `Item` class allows different types being used for its `id` property (`item.ts`):

```
export class Item<T> {
  constructor(
    public id: T,
    public name: string,
  ) {}

  describe(): string {
    return `${this.id}: ${this.name}`;
  }
}
```

The generic type parameter is written within angle brackets, and its naming conventionally starts with `T`.

Unlike the `name` property, which uses the specific type `string`, the `id` property can use any type `T`, which has to be filled in with a specific type when the class `Item` is being used:

```
import { Item } from "./item.js";

class Coordinates {
  constructor(
    public latitude: number,
    public longitude: number,
  ) {}

  toString(): string {
    return `${this.latitude};${this.longitude}`;
  }
}

const dilbert: Item<number> = new Item<number>(317, "Dilbert");
const stapler: Item<string> = new Item<string>("a3-v5-x7", "Stapler");
const chorweiler: Item<Coordinates> = new Item<Coordinates>(
  new Coordinates(51.028679, 6.89476),
```

```
  "Chorweiler",
);

console.log(dilbert.describe());
console.log(stapler.describe());
console.log(chorweiler.describe());
```

Output:

```
317: Dilbert
a3-v5-x7: Stapler
51.028679;6.89476: Chorweiler
```

The `Item` class not only supports `number`, `string`, or `Coordinates` as types for the `id` property, but any type that can be used with string interpolation, as used in its `describe` method (e.g. by providing a `toString` method).

12.1 Constraining Generic Types

The following example demonstrates the use of generics using geometry shapes (`shapes.ts`):

```
export class Rectangle {
  constructor(
    public width: number,
    public height: number,
  ) {}

  circumference(): number {
    return 2 * this.width + 2 * this.height;
  }

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

export class Square {
  constructor(public side: number) {}

  circumference(): number {
    return 4 * this.side;
  }
```

```
  area(): number {
    return this.side * this.side;
  }
}

export class Circle {
  constructor(public radius: number) {}

  circumference(): number {
    return 2 * this.radius * Math.PI;
  }

  area(): number {
    return this.radius * this.radius * Math.PI;
  }
}

export interface Shape {
  area(): number;
}

export class Body<T extends Shape> {
  constructor(
    public base: T,
    public height: number,
  ) {}

  volume(): number {
    return this.base.area() * this.height;
  }
}
```

The `Rectangle`, `Square`, and `Circle` class are unrelated, even though they implement the same methods (`circumference` and `area`), which could be grouped together by an interface. The class `Body` expects a generic type parameter `T`, which matches for any type that extends (or: implements, for that matter) the `Shape` interface. With an additional width, a shape is turned into a body:

- A `Rectangle` or a `Square` becomes a cuboid.
- A `Circle` becomes a cylinder.

Thanks to the type restriction on `T` (it must implement `Shape`), the `area` method can be used on the `base` object, without narrowing its type down to one specific type.

Those geometric objects can be used as follows:

```
import { Body, Circle, Rectangle, Square } from "./shapes.js";

let rectangle: Rectangle = new Rectangle(4, 3);
let square: Square = new Square(3);
let circle: Circle = new Circle(5);

console.log(
  `rectangle circumference: ${rectangle.circumference()};`,
  `rectangle area: ${rectangle.area()}`,
);
console.log(
  `square circumference: ${square.circumference()};`,
  `square area: ${square.area()}`,
);
console.log(
  `circle circumference: ${circle.circumference().toFixed(2)};`,
  `circle area: ${circle.area().toFixed(2)}`,
);

let cuboid: Body<Rectangle> = new Body(rectangle, 5);
let cube: Body<Square> = new Body(square, 4);
let cylinder: Body<Circle> = new Body(circle, 3);

console.log(`cuboid volume: ${cuboid.volume()}`);
console.log(`cube volume: ${cube.volume()}`);
console.log(`cylinder volume: ${cylinder.volume().toFixed(2)}`);
```

Output:

```
rectangle circumference: 14; rectangle area: 12
square circumference: 12; square area: 9
circle circumference: 31.42; circle area: 78.54
cuboid volume: 60
cube volume: 36
cylinder volume: 235.62
```

Generic types can also be restricted using a shape, so that only objects with certain properties or methods can be used. This is more flexible than restricting the type using a type union, which needs to be extended each time a new (compabible) type is additionally used.

The following example restricts its type parameter `T` using a shape type:

```
class NamedCollection<T extends { name: string }> {
  private items: T[];
```

```
  constructor() {
    this.items = new Array();
  }

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

  getNames(): string[] {
    return this.items.map((i) => i.name);
  }
}

type Named = { name: string };

class Dog {
  constructor(
    public name: string,
    public race: string,
  ) {}
}

class Company {
  constructor(
    public name: string,
    public revenue: number,
  ) {}
}

const myStuff: NamedCollection<Named> = new NamedCollection<Named>();
myStuff.add(new Dog("Doge", "Pitbull"));
myStuff.add(new Company("ACME Inc.", 1_359_725.39));
console.log(myStuff.getNames());
```

Output:

```
[ 'Doge', 'ACME Inc.' ]
```

12.2 Multiple Type Parameters

A class can make use of multiple type parameters, listed within angle brackets, separated by commas, e.g. `<T, U>`. The following class computes the volume based on an object that has an area,

and on another one that has a height:

```
type WithArea = { area(): number };
type WithHeight = { height: number };

class ShapeConverter<T extends WithArea, U extends WithHeight> {
  constructor(
    private withArea: T,
    private withHeight: U,
  ) {}

  volume(): number {
    return this.withArea.area() * this.withHeight.height;
  }
}

const rect = {
  a: 3,
  b: 4,
  area: function (): number {
    return this.a * this.b;
  },
};
const height = {
  height: 5,
};
const conv = new ShapeConverter(rect, height);
console.log(conv.volume());
```

Output:

```
60
```

The second type parameter `U` could also be moved to the `volume` method, making it generic:

```
type WithArea = { area(): number };
type WithHeight = { height: number };

class ShapeConverter<T extends WithArea> {
  constructor(private withArea: T) {}

  volume<U extends WithHeight>(withHeight: U): number {
    return this.withArea.area() * withHeight.height;
  }
```

```
}

const rect = {
  a: 3,
  b: 4,
  area: function (): number {
    return this.a * this.b;
  },
};
const height = {
  height: 5,
};
const conv = new ShapeConverter(rect);
console.log(conv.volume(height));
```

Output:

```
60
```

Notice that the type parameters haven't been stated explicitly, but have been inferred by the compiler. Check the declarations file to find out which types have been inferred.

12.3 Inheritance

When extending a class that expects a type parameter, the subclass has to provide a compatible type to its superclass:

```
class Item<T extends { identify(): string }> {
  private id: string;

  constructor(
    private name: string,
    thing: T,
  ) {
    this.id = thing.identify();
  }
}

class StockedItem<
  T extends { identify(): string; count(): number },
> extends Item<T> {
  private stock: number;
```

```
  constructor(name: string, thing: T) {
    super(name, thing);
    this.stock = thing.count();
  }
}

const orange = new Item("Orange", { identify: () => "ORNG" });
const oranges = new StockedItem("Oranges", {
  identify: () => "ORNG",
  count: () => 37,
});
```

A subclass can also fix the generic type of its superclass by replacing the type parameter with a specific type. A type parameter can also be further restricted in a subclass using a type union, as long as the type parameter in the subclass is more restrictive than the one of its superclass. (Values of the type parameter of the subclass must be assignable to the type parameter of the superclass.)

The `instanceof` operator cannot be used for checking generic type arguments, because the type information is only available during compilation time, but not during run time. A predicate function can be used instead.

The following code won't compile:

```
class Collection<T> {
  private items: T[] = [];

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

  filter<V extends T>(): V[] {
    return this.items.filter((i) => i instanceof V) as V[];
  }
}

class Car {
  constructor(
    public brand: string,
    public model: string,
  ) {}
}

class Drink {
  constructor(
```

```
    public brand: string,
    public name: string,
  ) {}
}

let stuff = new Collection<Car | Drink>();
stuff.add(new Car("Porsche", "911"));
stuff.add(new Drink("Jack Daniels", "Whiskey"));
let drinks = stuff.filter<Drink>();
console.log(drinks);
```

Error:

```
error TS2693: 'V' only refers to a type, but is being used as a value here.
```

However, the problem can be fixed using a predicate function, which asserts the type at compilation time:

```
class Collection<T> {
  private items: T[] = [];

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

  filter<V extends T>(predicate: (target: T) => target is V): V[] {
    return this.items.filter((i) => predicate(i)) as V[];
  }
}

class Car {
  constructor(
    public brand: string,
    public model: string,
  ) {}
}

class Drink {
  constructor(
    public brand: string,
    public name: string,
  ) {}
}
```

```
function isDrink(target: any): target is Drink {
  return target instanceof Drink;
}

let stuff = new Collection<Car | Drink>();
stuff.add(new Car("Porsche", "911"));
stuff.add(new Drink("Jack Daniels", "Whiskey"));
let drinks = stuff.filter<Drink>(isDrink);
console.log(drinks);
```

Static methods can accept their own type arguments, just as instance methods.

Generics can also be used for interfaces, which can further restrict generic types as subclasses can do.

13 Advanced Generic Types

JavaScript's collections can be used with generic type parametes, e.g. `Map<K, V>` with type `K` for keys and type `V` for values, and `Set<T>` with type `T` for the (unique) values:

```
let numbers: number[] = [1, 2, 3, 4, 1, 2, 3, 1, 2, 1];
let uniqueNumbers: Set<number> = new Set<number>();
let numbersCount: Map<number, number> = new Map<number, number>();

for (let n of numbers) {
  uniqueNumbers.add(n);
  if (numbersCount.has(n)) {
    let c: number = numbersCount.get(n) as number;
    numbersCount.set(n, c + 1);
  } else {
    numbersCount.set(n, 1);
  }
}

console.log(uniqueNumbers);
console.log(numbersCount);
```

TypeScript defines generic interfaces for iteration:

- `Iterator<T>`: an iterator with a `next` method returning `IteratorResult<T>` objects
- `IteratorResult<T>`: a result object with a `done` and a `value` property
- `Iterable<T>`: an object that defines the `Symbol.iterator` property for iteration
- `IterableIterator<T>`: a combination of `Iterable<T>` and `Iterator<T>`

The following example shows how the `Iterator` and `IteratorResult` types are used explicitly:

```
let menu: Map<string, number> = new Map<string, number>();
menu.set("Beer", 5.0);
menu.set("Coffee", 4.2);
menu.set("Water", 2.5);

let prices: Iterator<number> = menu.values();
let price: IteratorResult<number>;
let total: number = 0;
for (price = prices.next(); !price.done; price = prices.next()) {
  total += price.value;
}

console.log(`total price: ${total}`); // 11.7
```

Notice that iterators are only available since ES6. To target ES5 an earlier, the `downlevelIteration` compiler option needs to be set to `true`.

The `prices` iterator in the above example cannot directly be used in a `for/of` loop:

```
let total: number = 0;
for (let price of menu.prices()) {
  total += price;
}
```

```
error TS2488: Type 'Iterator<number, any, any>' must have a '[Symbol.iterator]()' method that returns an iterat
```

This is where the `IterableIterator<T>` interfaces is useful, which allows for objects to be directly iterated over by combining the `Iterator<T>` with the `Iterable<T>` interfaces:

```
class Menu {
  private menu: Map<string, number>;

  constructor() {
    this.menu = new Map<string, number>();
  }

  add(name: string, price: number) {
    this.menu.set(name, price);
  }

  prices(): IterableIterator<number> {
    return this.menu.values();
  }
```

```
}

let menu: Menu = new Menu();
menu.add("Beer", 5.0);
menu.add("Coffee", 4.2);
menu.add("Water", 2.5);

let total: number = 0;
for (let price of menu.prices()) {
  total += price;
}

console.log(`total price: ${total}`);
```

The method `Map.values` returns an iterator, which can be accessed over the `Menu.prices` method. To go one step further, the `Menu` class can be made iterable, saving the explicit method call:

```
class Menu implements Iterable<number> {
  private menu: Map<string, number>;

  constructor() {
    this.menu = new Map<string, number>();
  }

  add(name: string, price: number) {
    this.menu.set(name, price);
  }

  [Symbol.iterator](): Iterator<number> {
    return this.menu.values();
  }
}

let menu: Menu = new Menu();
menu.add("Beer", 5.0);
menu.add("Coffee", 4.2);
menu.add("Water", 2.5);

let total: number = 0;
for (let price of menu) {
  total += price;
}

console.log(`total price: ${total}`);
```

13.1 Index Types

Given a type `T`, the *index type query operator* `keyof` returns a union of the type's property names, which can be used as a type. This allows to constrain the dynamic access to properties:

```
class Accessor<T> {
  constructor(private object: T) {}

  get(property: keyof T): any {
    return this.object[property];
  }

  set(property: keyof T, value: any) {
    this.object[property] = value;
  }
}

class Person {
  constructor(
    public name: string,
    public age: number,
  ) {}
}

let dilbert: Person = new Person("Dilbert", 42);
let accessor: Accessor<Person> = new Accessor<Person>(dilbert);
accessor.set("age", 43);
console.log(`${accessor.get("name")} is ${accessor.get("age")} years old.`);
```

Output:

```
Dilbert is 43 years old.
```

Notice that the following operation fails:

```
accessor.set("dead", true);
```

```
error TS2345: Argument of type '"dead"' is not assignable to parameter of type 'keyof Person'.
```

The compiler detects that there is no such property `dead` on the type `Person` and refuses to compile the code.

The indexed access operator can be used together with the `type` keyword:

```
class Person {
  constructor(
    public name: string,
    public age: number,
  ) {}
}

let dilbert: Person = new Person("Dilbert", 42);

type allTypes = Person[keyof Person];

function get(obj: Person, prop: keyof Person): allTypes {
  return obj[prop];
}

console.log(get(dilbert, "name"), get(dilbert, "age"));
```

Output:

```
Dilbert 42
```

A type defined using the indexed access operator is known as a *lookup type*. Such types are most useful in conjunction with generic types, whose properties cannot be known beforehand.

13.2 Type Mappings

Index types can be used to map types, i.e. to programmatically create new types of existing types, thereby retaining or changing the original type's properties.

The following examples maps an existing type `Product` using a generic type mapping:

```
class Product {
  constructor(
    public id: number,
    public name: string,
    public stock: number,
  ) {}
}

type Mapped<T> = {
  [P in keyof T]: T[P];
};

let p: Mapped<Product> = { id: 3, name: "Beer", stock: 17 };
```

This feature becomes useful when it is used to change properties to change their access mode (optional/required) or readonly mode:

- suffix `?`: make optional
- suffix `-?`: make required
- prefix `readonly`: make readonly
- prefix `-readonly`: make read-write

Type mappings can be chained, as the following example shows:

```
class Product {
  constructor(
    public id: number,
    public name: string,
    public stock: number,
  ) {}
}

type Mapped<T> = {
  [P in keyof T]: T[P];
};

type MappedOptional<T> = {
  [P in keyof T]?: T[P];
};

type MappedRequired<T> = {
  [P in keyof T]-?: T[P];
};

type MappedReadonly<T> = {
  readonly [P in keyof T]: T[P];
};

type MappedReadWrite<T> = {
  -readonly [P in keyof T]: T[P];
};

type MappedProduct = Mapped<Product>;
type OptionalProduct = MappedOptional<MappedProduct>;
type RequiredProduct = MappedRequired<OptionalProduct>;
type ReadonlyProduct = MappedReadonly<RequiredProduct>;
type ReadWriteProduct = MappedReadWrite<ReadonlyProduct>;
```

```
let p: ReadWriteProduct = { id: 3, name: "Beer", stock: 17 };
p.name = `Lager ${p.name}`;
p.stock--;
console.log(p);
```

TypeScript provides some mapped types:

- `Partial<T>`: makes properties optional
- `Required<T>`: makes properties required
- `Readonly<T>`: makes properties readonly
- `Pick<T, K>`: selects specific properties
- `Omit<T, keys>`: omits specific properties
- `Record<T, K>`: creates a type without transformations

13.3 Conditional Types

A *conditional type* is a placeholder for a *result type* to be chosen as a generic type argument is used:

```
type parsedType<T extends boolean> = T extends true ? number : string;

let parsed: parsedType<true> = 123;
let unparsed: parsedType<false> = "123";

let mismatch: parsedType<true> = "123";
```

The type `parsedType` can be used with either `true` or `false`. In the first case, the type becomes `number`, in the second case, the type becomes `string`. The first two variables (`parsed`, `unparsed`) are correct assignments. The third variable (`mismatch`) causes an error:

```
error TS2322: Type 'string' is not assignable to type 'number'.
```

Using nested conditional types increases the risk of missing some possible combinations of valid types/values, which makes the code harder to understand, and creates holes in the type system.

Conditional types can be nested:

```
type state = "missing" | "available" | "raw" | "parsed";
type inputState<T extends state> = T extends "available"
  ? T extends "raw"
    ? string
    : number
  : Object;

let missing: inputState<"missing"> = {};
```

```
let unparsed: inputState<"raw"> = "123";
let parsed: inputState<"parsed"> = 123;
```

However, nested type expressions do not improve readability.

TypeScript defines some conditional types ready to be used:

- `Exclude<T, U>`: exclude the types that can be assigned to `U` from `T`
- `Extract<T, U>`: select the types that can be assigned to `U` from `T`
- `NonNullable<T>`: excludes `null` and `undefined` from `T`
- `Parameters<T>`: select types of function parameters
- `ReturnType<T>`: select return type of a function
- `ConstructorParameter<T>`: select types of constructor aprameters
- `InstanceType<T>`: select type of a constructor function

14 Using Decorators

Decorators are functions that transform classes, methods, fields, and accessors by replacing without otherwise modifying them. TypeScript 5 already supports decorators, and a future JavaScript specification will adopt them natively.

Decorators are applied using the `@` character in front of their name. The following decorator logs the time method invocations take:

```
function timed(method: any, ctx: ClassMethodDecoratorContext) {
  const name = String(ctx.name);
  return function (this: any, ...args: any[]) {
    const start = performance.now();
    const result = method.call(this, ...args);
    const finished = performance.now();
    const duration = (finished - start).toFixed(2);
    console.log(`${name}: ${duration}ms`);
    return result;
  };
}

class Factorial {
  constructor(private n: number) {}

  @timed
  calculate(): number {
    return this.doCalculate(this.n);
  }
```

```
  doCalculate(i: number): number {
    if (i == 0) {
      return 1;
    } else {
      return i * this.doCalculate(i - 1);
    }
  }
}

console.log(new Factorial(10).calculate());
```

The `ClassMethodDecoratorContext` defines various members:

- `kind`: the kind of element the decorator has been applied to (e.g. `"method"`)
- `name`: the name of the element the decorator has been applied to (string or symbol)
- `static`: whether or not the decorator has been applied to a static element
- `private`: whether or not the decorator has been applied to a private element

The `performance` API is used for timing purposes. Notice that the method is replaced by a function that performs the original task (i.e. calls the original method in its context) plus some additional task (measuring its timing).

For other kinds of decorators, other context types are used:

- Class: `ClassDecoratorContext`
- Methods: `ClassMethodDecoratorContext`
- Fields: `ClassFieldDecoratorContext`
- Accessors: `ClassGetterDecoratorContext` and `ClassSetterDecoratorContext`
- Auto-accessors: `ClassAccessorDecoratorContext`

See the documentation on decorators for more information.