
TS

Decorator

Metaprogramming

- Metaprogramming is nothing less than the magic in programming!
- As per Wikipedia
  - Metaprogramming is a programming technique in which computer programs have the ability to treat other programs as their data
  - This means that a program can be designed to read, generate, analyze, or transform other programs, and even modify itself while running
- Simply put, Metaprogramming involves writing code that can
  - Generate code
  - Manipulate language constructs at the run time. This phenomenon is known as Reflective Metaprogramming or Reflection.

Reflection in Metaprogramming

- Reflection is a branch of Metaprogramming
- Reflection has three sub-branches:
  - **Introspection**
    - Code is able to inspect itself
    - It is used to access the internal properties such that we can get the low-level information of our code
  - **Self-Modification**
    - As the name suggests, code is able to modify itself
  - **Intercession**
    - The literal meaning of intercession is, acting on behalf of somebody else
    - In metaprogramming, the intercession does exactly the same using the concepts like, wrapping, trapping, intercepting
- ES6 gives us the Reflect object (aka the Reflect API) to achieve Introspection
- The Proxy object of ES6 helps us with Intercession

Decorator

- Decorators in TypeScript provide a way of programmatically tapping into the process of defining class
- Decorators allow us to inject code into the actual definition of a class
- Decorators can be used on class definitions, class properties, class functions, and even method parameters
- Decorators use the form @expression, where expression must evaluate to a function that will be called at runtime with information about the decorated declaration
- The concept of decorators exists in other programming languages, and are called attributes in C#, or annotations in Java or decorators in python

Decorator syntax

- A decorator is simply a function that is called with a specific set of parameters
- These parameters are automatically populated by the JavaScript runtime, and contain information about the class to which the decorator has been applied
- Number of parameters and types of these parameters determine where a decorator can be applied

```
function simpleDecorator(constructor : Function) {
  console.log('simpleDecorator called.')
}
```

Class Decorator

- A Class Decorator is declared just before a class declaration
- The class decorator is applied to the constructor of the class and can be used to observe, modify, or replace a class definition
- The expression for the class decorator will be called as a function at runtime, with the constructor of the decorated class as its only argument
- If the class decorator returns a value, it will replace the class declaration with the provided constructor function

```
function Logger(constuctor: Function) {
  console.log('logging enabled')
}

@Logger
class Person {
  constructor(
    private name: string) {
    console.log('creating object')
  }
}

const person = new Person('person1')
```

Decorator Factories

- If we want to customize how a decorator is applied to a declaration, we can write a decorator factory
- A Decorator Factory is simply a function that returns the expression that will be called by the decorator at runtime

```
function Logger() {
  return function (constuctor: Function) {
    console.log('inside Logger')
  }
}

@Logger()
class Person {
  constructor(
    private name: string) {
    console.log('creating object')
  }
}

const person = new Person('person1')
```

Method Decorators

- A Method Decorator is declared just before a method declaration
- The can be used to observe, modify, or replace a method definition
- The expression for the method decorator will be called as a function at runtime, with the following three arguments:
  - Either the constructor function of the class for a static member, or the prototype of the class for an instance member.
  - The name of the member.
  - The Property Descriptor for the member.
- If the method decorator returns a value, it will be used as the Property Descriptor for the method

```
function Logger() {
  return function (
    target: any,
    propertyKey: string,
    descriptor: PropertyDescriptor
  ) {
    console.log('inside Logger')
  }
}

class Person {

  @Logger()
  hello() {
    console.log('hello hello')
  }
}
```

Property Decorator

- A Property Decorator is declared just before a property declaration
- The expression for the property decorator will be called as a function at runtime, with the following two arguments:
  - Either the constructor function of the class for a static member, or the prototype of the class for an instance member
  - The name of the member

```
function Logger() {
  return function (
    target: any,
    propertyName: string) {
    console.log('inside Logger')
  }
}

class Person {
  @Logger()
  name: string
}
```

Parameter Decorator

- A Parameter Decorator is declared just before a parameter declaration
- The parameter decorator is applied to the function for a class constructor or method declaration
- The expression for the parameter decorator will be called as a function at runtime, with the following three arguments:
  - Either the constructor function of the class for a static member, or the prototype of the class for an instance member
  - The name of the member
  - The ordinal index of the parameter in the function's parameter list

```
function Logger() {
  return function (
    target: any,
    propertyName: string,
    index: number) {
    console.log('inside Logger')
  }
}

class Person {
  sayHello(@Logger() message: string) {
    console.log('message: ', message)
  }
}
```

Async Programming

Synchronous vs Asynchronous

- In programming, synchronous operations block instructions until the task is completed, while asynchronous operations can execute without blocking other operations
- An asynchronous program can make the system capable of doing more work increasing the number of requests that can be handled at the same time with the same resources than the synchronous one, thus improving throughput
- Asynchronous operations are generally completed by firing an event or by calling a provided callback function

Asynchronous Programming

- TypeScript provides following ways for async programming
  - Callbacks
  - Promises
  - Async/Await

Callbacks

- Callback is the simplest way to create async code
- Callback is a parameter of type function, which is executed when the job is done
- Usually developer writes this function but never calls it explicitly

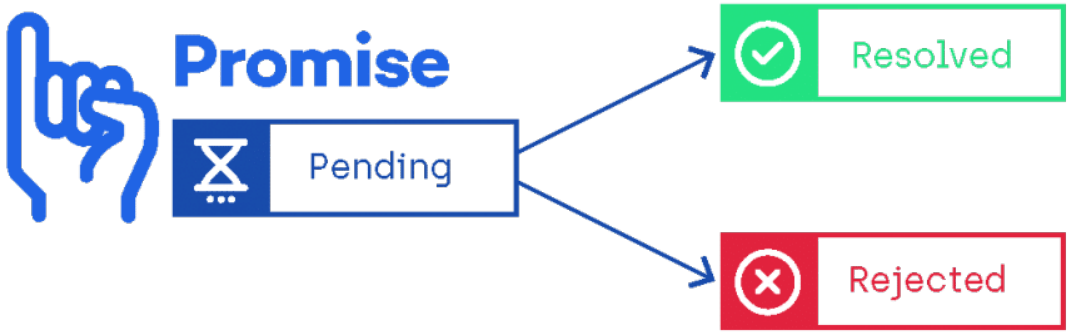
```
function makeTea(serve: Function) {
  getHotWater(() => {
    getTeaLeaves(() => {
      wait(() => {
        serve()
      })
    })
  })
}

makeTea(() => {
  console.log('servering tea')
})
```

Promise

- A promise in JavaScript is similar to a promise in real life
- When we make a promise in real life, it is a guarantee that we are going to do something in the future
- A promise has 2 possible outcomes
  - it will either be kept when the time comes
  - or it won't
- This is also the same for promises in TypeScript. When we define a promise in TypeScript, it will be resolved when the time comes, or it will get rejected

Promise

- A promise is an object that may produce a single value some time in the future
  - either a resolved value, or
  - reason that it's not resolved
- A promise is an object which can be returned synchronously from an asynchronous function
- It will be in one of 3 possible states:
  - **Pending:** Initial State, before the Promise succeeds or fails
  - **Resolved:** Completed Promise
  - **Rejected:** Failed Promise
- A promise is **settled** if it's not pending (it has been resolved or rejected)
- You can consume promise by calling then(), catch() and , finally() methods on the promise



Promise – then()

- The Then is called when a promise is resolved
- Note that, it takes two parameters:
  - The resolve callback is called if the promise is fulfilled.
  - The reject callback is called when the promise is rejected

```
function doPromise(finish) {
  return new Promise(function (resolve, reject) {
    setTimeout(() => {
      if (finish) {
        resolve("Your 🚗 has repaired")
      } else {
        reject("Your 🚗 has not repaired yet")
      }
    }, 1000);
  });
}

const res = doPromise(true)
res.then(
  success => console.log(success),
  reason => console.log(reason)
)
```

Promise – catch()

- The catch is called when a promise is rejected
- Internally, the catch() method invokes the then(undefined, reject) method

```
const res = doPromise(false)
res.then(
  success => {
    console.log('success')
    console.log(success)
  }
).catch((reason) => {
  console.log('catch')
  console.log(reason)
})
```

Promise – finally()

- In every situation want to execute the same piece of code whether the promise is resolved or rejected
- finally() will be called in both resolved and rejected state

```
const res = doPromise(false)
res.then(
  success => {
    console.log(success)
  }).catch((reason) => {
    console.log(reason)
  }).finally(() => {
    console.log('finally called')
  })
```

Promise

- The promised version of the same program

```
function makeTea(serve: Function) {
  getHotWater()
    .then(() => getTeaLeaves())
    .then(() => wait())
    .then(() => serve())
}

makeTea(() => {
  console.log('servering tea')
})
```

Async/Await

- The pyramid of doom was significantly mitigated with the introduction of Promises
- Promises paved the way to one of the coolest improvements in JavaScript.
- ECMAScript 2017 brought in syntactic sugar on top of Promises in JavaScript in the form of async and await statements.
- They allow us to write Promise-based code as if it were synchronous, but without blocking the main thread
- In short, async/await feature is just an extension to promises

Async

- Async functions enable us to write promise based code as if it were synchronous, but without blocking the execution thread
- It operates asynchronously via the event-loop
- Async functions will always return a value
- Using async simply implies that a promise will be returned, and if a promise is not returned, JavaScript automatically wraps it in a resolved promise with its value

```
async function test() {
  return 20
}

test()
  .then(value => {
    console.log(value)
  })
```

Await

- The await operator is used to wait for a Promise
- It can be used inside an Async block only
- The keyword Await makes TypeScript wait until the promise returns a result
- It has to be noted that it only makes the async function block wait and not the whole program execution

```
function sleep() {
  console.log('sleeping now')
  return new Promise(resolve => {
    setTimeout(resolve, 1000)
  })
}

async function test() {
  await sleep()
  console.log('called after sleep')
}

test()
```

Modules

Introduction

- Because TypeScript compiles to and interoperates with JavaScript, it has to support the various module standards that JavaScript programmers use
- Modules are a design pattern used to improve the structure, readability, and testability of code
- You can do the following with Modules:
  - Structure code in self-contained **blocks**
  - Encapsulate code (since modules are self-contained)
  - Define public APIs: Modules expose what they want to define their interface with the outside world
  - Create isolated namespaces: Modules have their own namespaces and they don't pollute other modules
  - Reuse existing code by loading and using third-party libraries, other modules, and much more
  - Define your dependencies: Explicitly define the list of modules that the current one requires to function

History

- When JS launched, it was not having any support for module which made developer's life difiicult
- To ease the pain, many developers started using object literals and naming conventions to try and avoid name clashes (known as Revealing Module Pattern) e.g. jQuery
- JS now supports following standards
  - **Asynchronous Module Definition** (**AMD**)
  - **CommonJS** (**CJS**)
  - **Universal Module Definition** (**UMD**)
- With ES2015, ECMAScript introduced official language support for modules. Hence it is now possible to define modules using a standard module format called **ES Modules** (**ESM**)

Module Loaders

- **Module loaders** take care of loading modules at runtime
- Different loaders support different combinations of module formats
- They are useful as soon as your target environment cannot load all the modules that you intend to make use of
- Following are popular module loaders
  - SystemJS: https://github.com/systemjs/systemjs
  - jspm.io: https://jspm.io
  - StealJS: https://stealjs.com
  - RequireJS: https://requirejs.org

Bundlers

- **Bundlers** take care of bundling or *grouping* assets together, including modules at build time
- The main responsibility of bundlers is to understand the dependency graph of your code and to bundle/group everything together
- Following are the responsibilities
  - **Splitting code**: Splitting your code into different bundles that you can load lazily (that is, on demand/when needed)
  - **Transpiling code** (for example, TypeScript to JavaScript)
  - **Minification and uglification**: Compressing the code for production
  - **Processing CSS** (for example, using SASS, Less, PostCSS, and many others)
  - **Linting code**: Checking the code quality
  - **Testing code**: Executing tests automatically
  - **Generating source maps**
  - **Copying and transforming assets**: Copying images, fonts, and much more and transforming them if needed (for example, compressing images)

Bundlers

- Here are a few examples of bundlers:
  - Webpack: https://webpack.js.org
  - Parcel: https://parceljs.org
  - Rollup: https://rollupjs.org/guide/en
  - StealJS: https://stealjs.com
  - FuseBox: https://fuse-box.org
  - Babel: https://babeljs.io
  - Browserify: http://browserify.org