Types in Prototypes

Viktor
Vershanskiy

wentout

JAVASCRIPT-ON-DON

RnD JS

JS

JAVASCRIPT-ON-DON

RnD JS

JS

Виктор

wentout

Bio

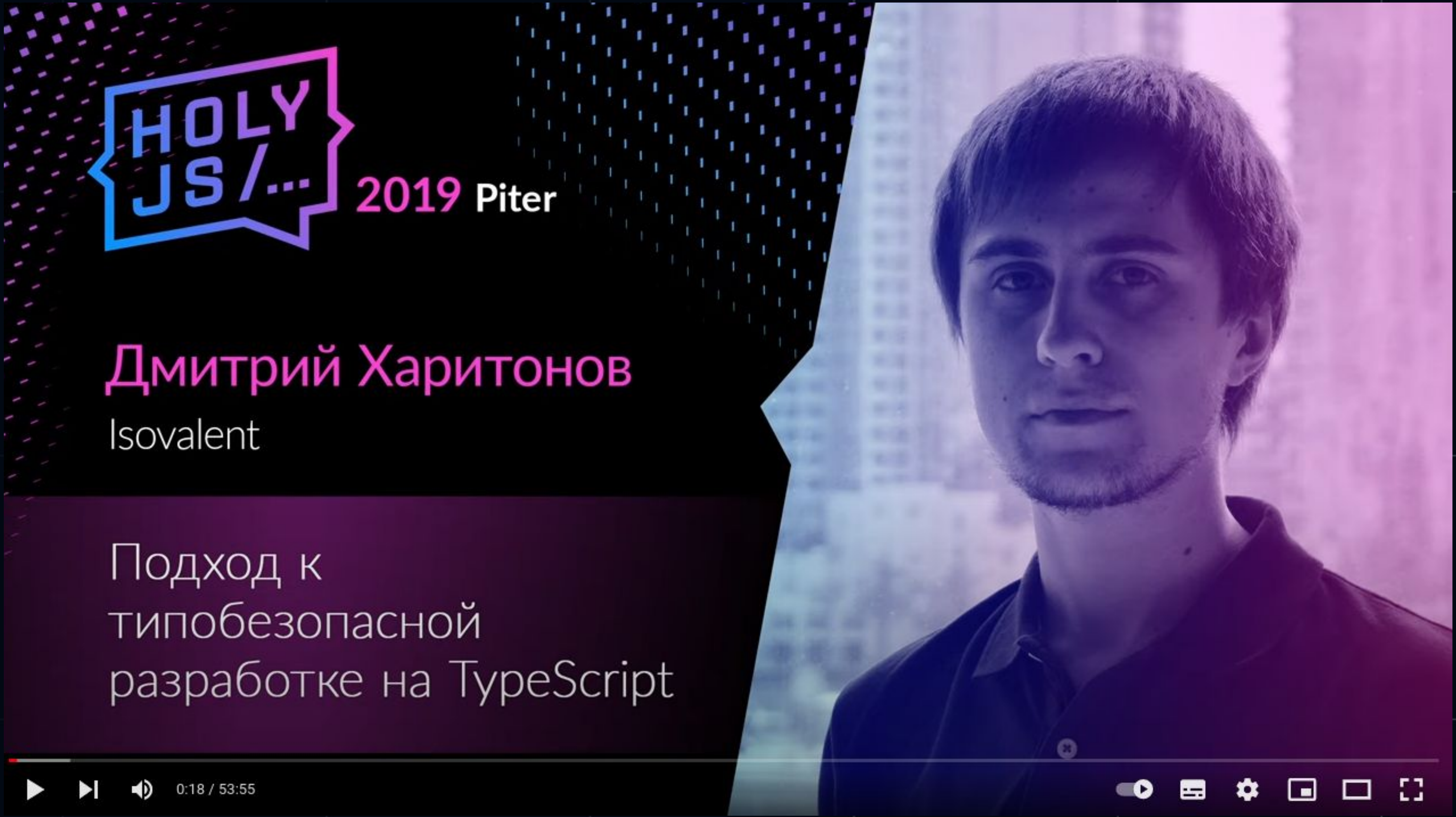
- JS production at 1999
- Back-End на JS в 2000
- Node.js с 2009
- Diagnostics Group
- BUGs Chrome & v8
- PMI PMBoK + Agile
- PhD in Economy of IT

Outline of the talk

- Known TypeScript of HolyJS talks last 3 yrs
- Previous talks of this Moonshine Spiritual Journey
- Class Based vs Functional Based Constructors
- Time Matters the Difference
- Real three types of Inheritance in JS ~ TS
- Optional Fields Definitions
- Identity as a Single Pattern of Chaining

TypeScript on HolyJS

Known TypeScript on HolyJS



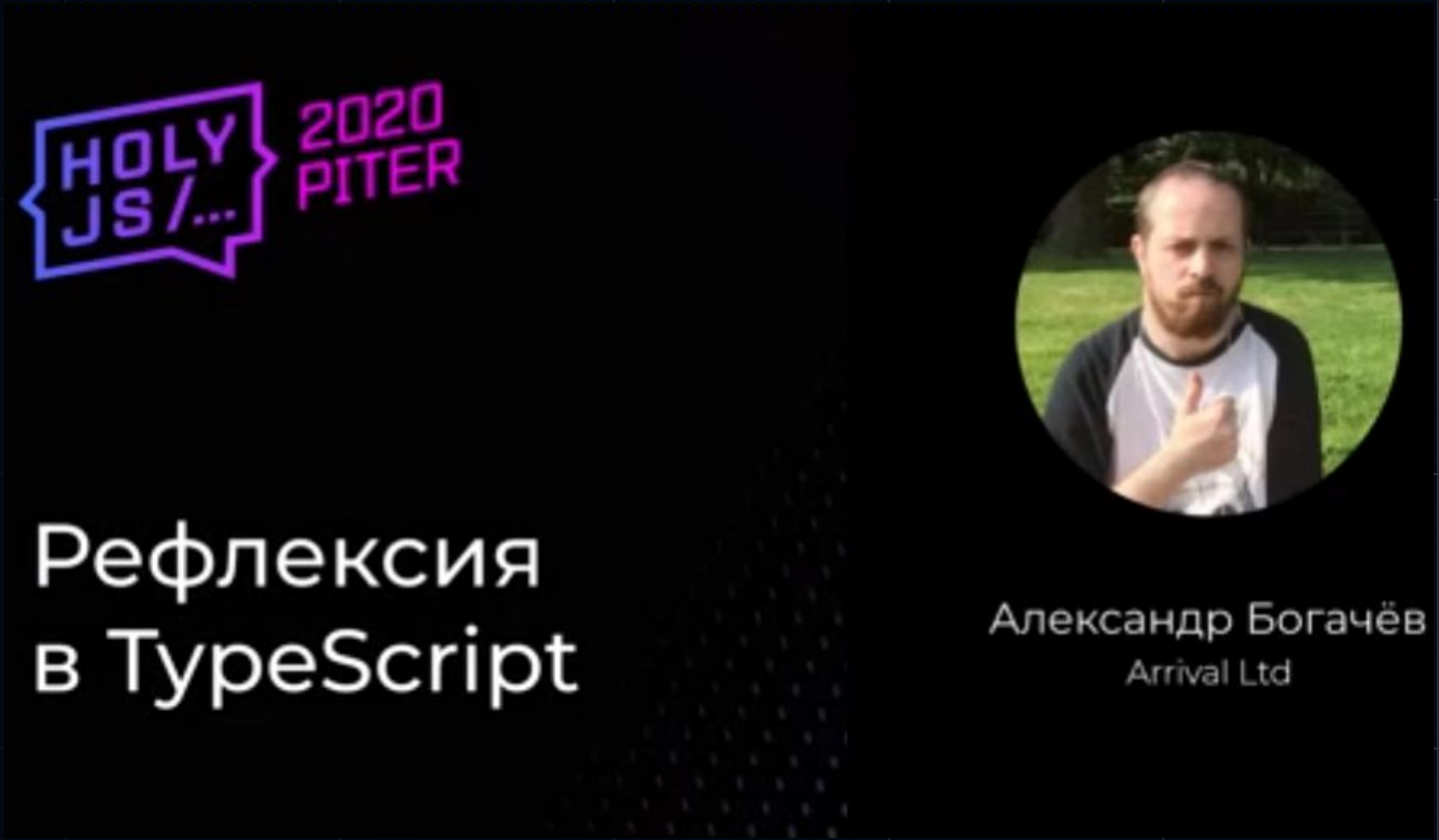
HOLY JS /... 2019 Piter

Дмитрий Харитонов

Isovalent

Подход к типобезопасной разработке на TypeScript

Known TypeScript on HolyJS

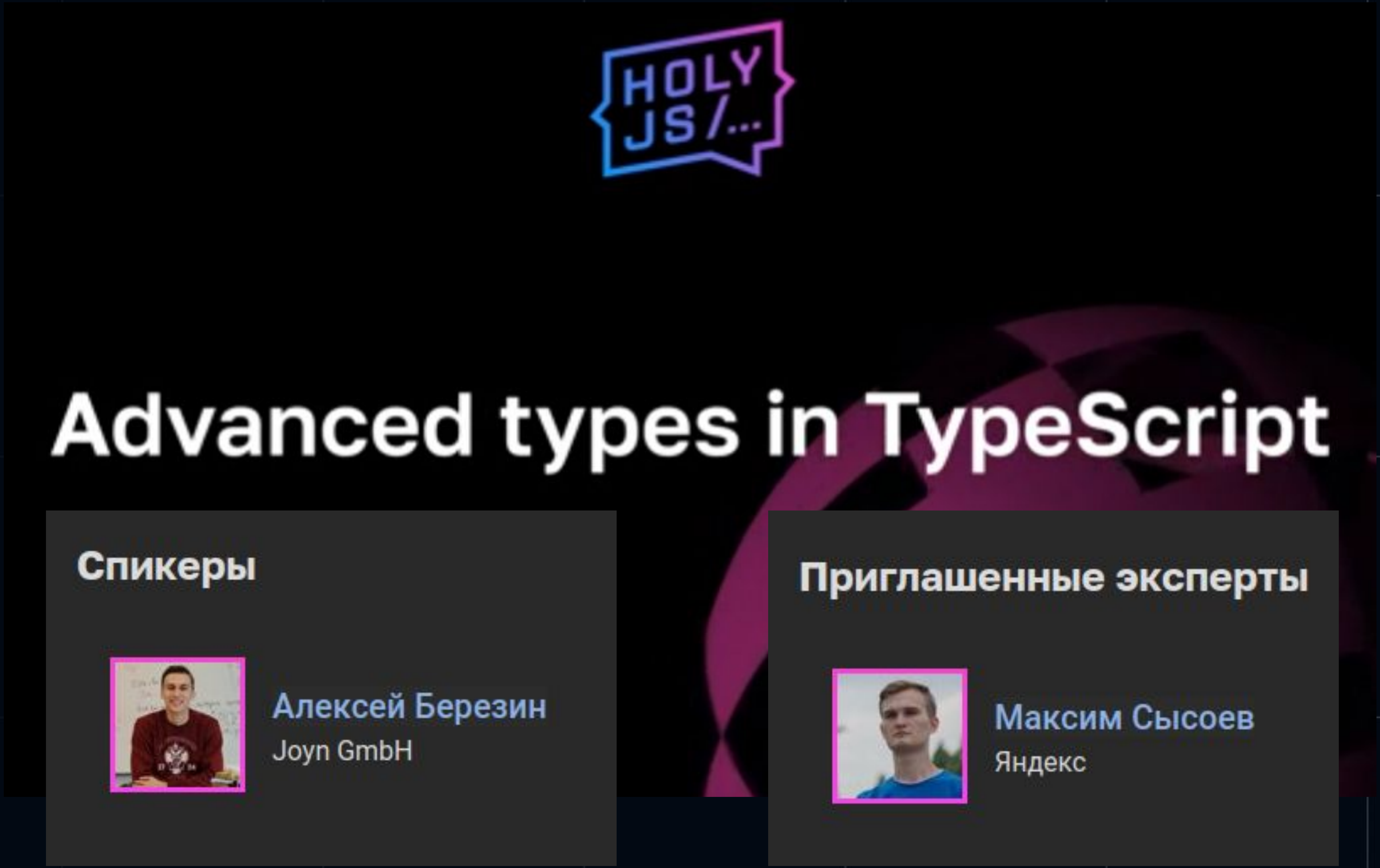


HOLY JS/... 2020 PITER

Рефлексия в TypeScript

Александр Богачёв
Arrival Ltd

Known TypeScript on HolyJS



Advanced types in TypeScript

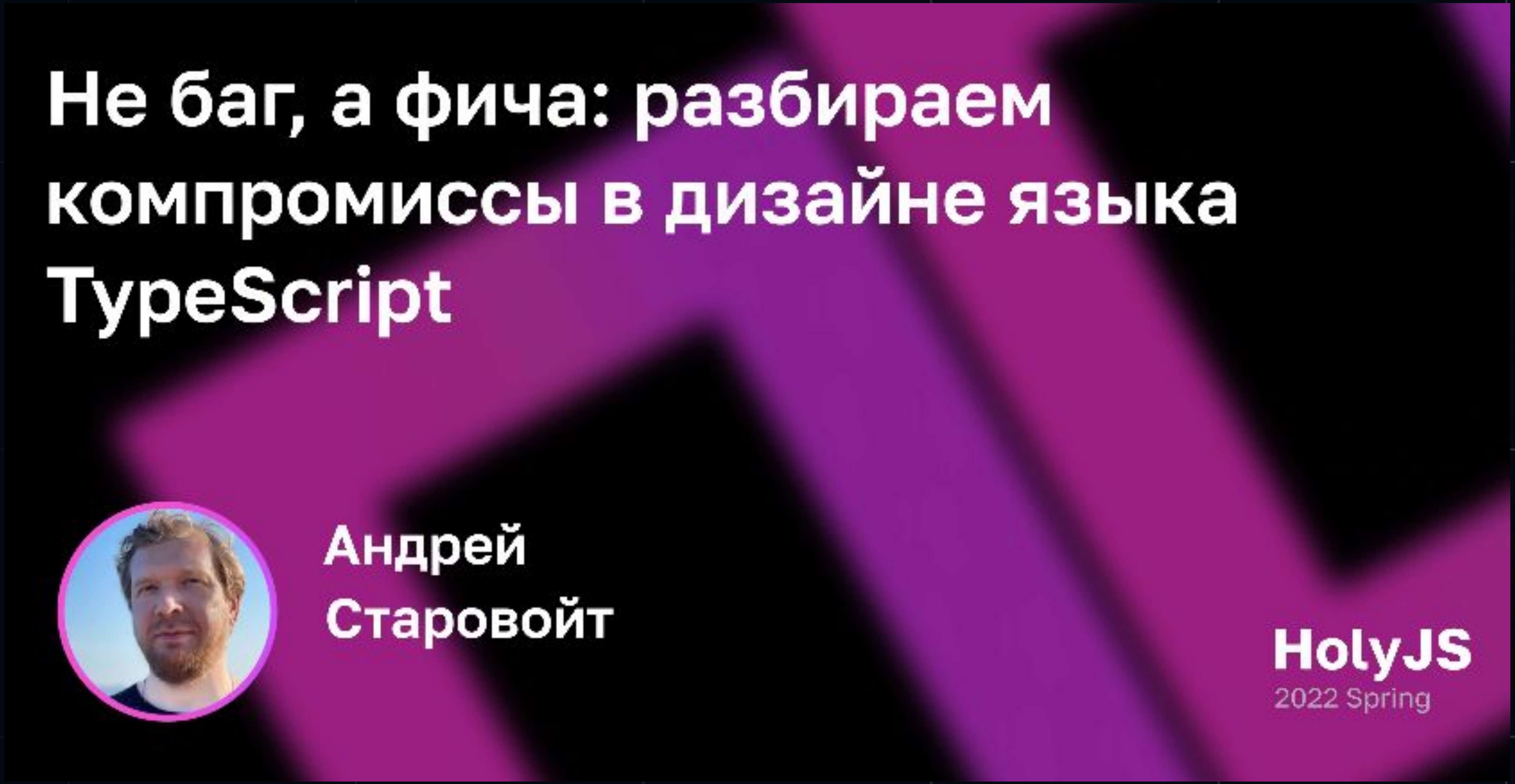
Спикеры

Алексей Березин
Joyn GmbH

Приглашенные эксперты

Максим Сысоев
Яндекс

Known TypeScript on HolyJS

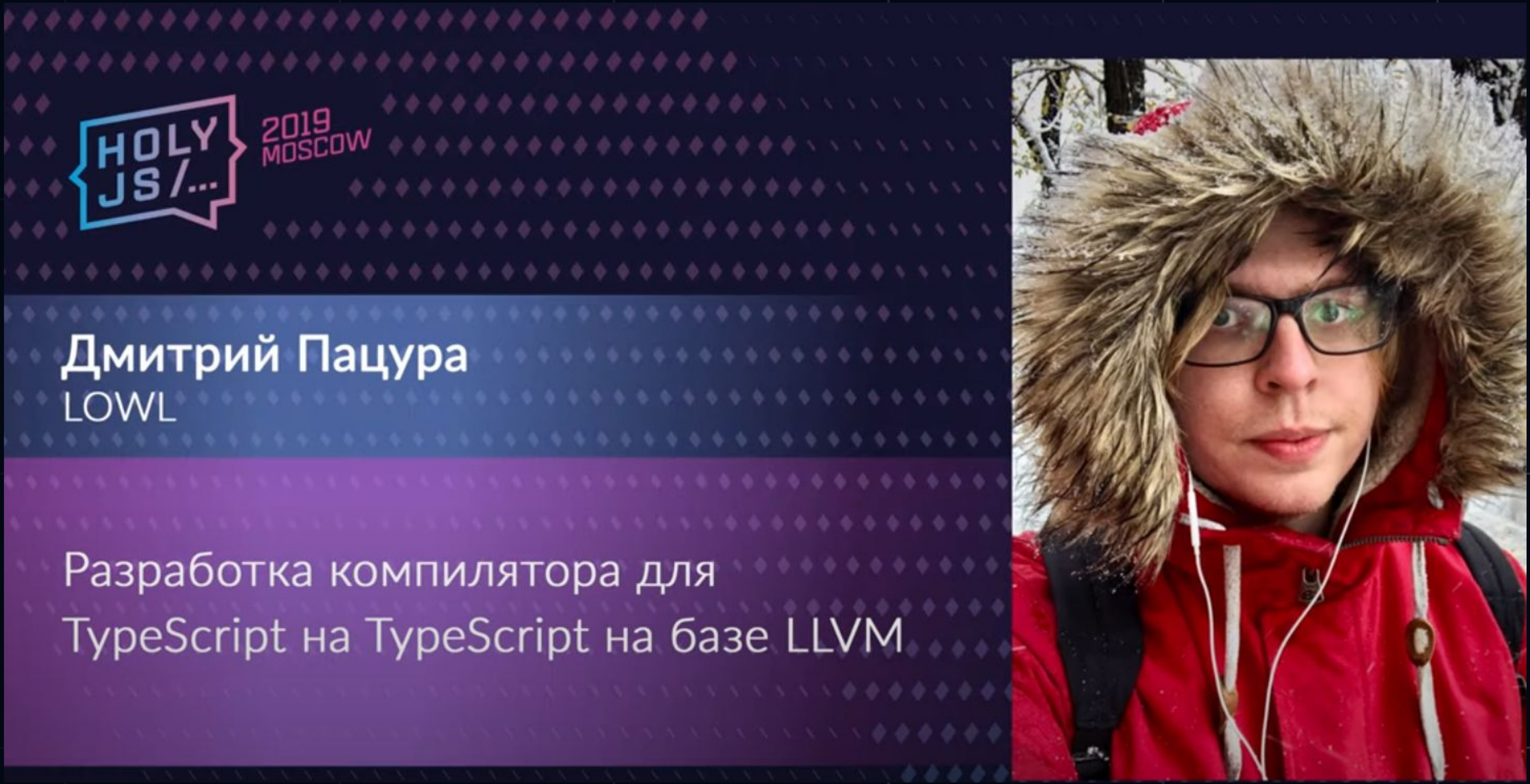


Не баг, а фича: разбираем компромиссы в дизайне языка TypeScript

Андрей Старовойт

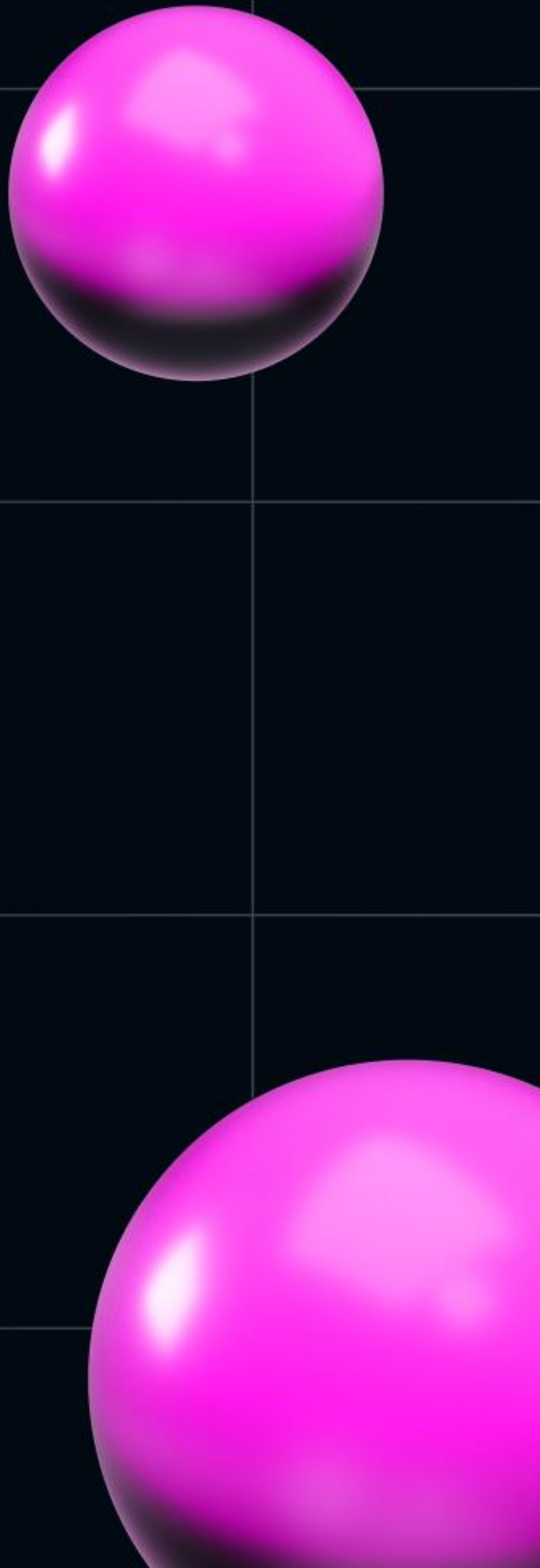
HolyJS 2022 Spring

Known TypeScript on HolyJS



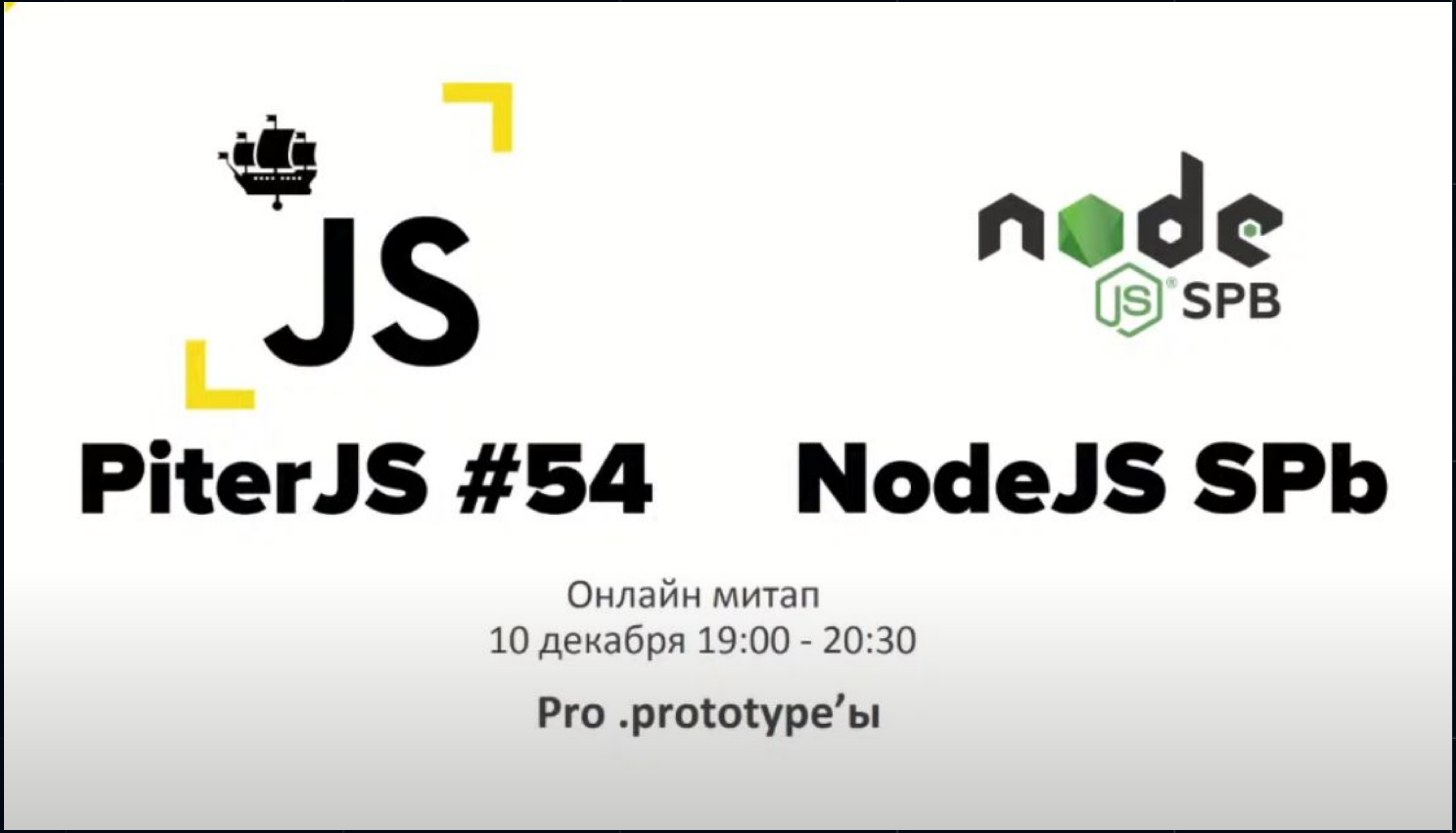
Дмитрий Пацура
LOWL

Разработка компилятора для TypeScript на TypeScript на базе LLVM



Moonshine Spiritual

Moonshine Spiritual talks

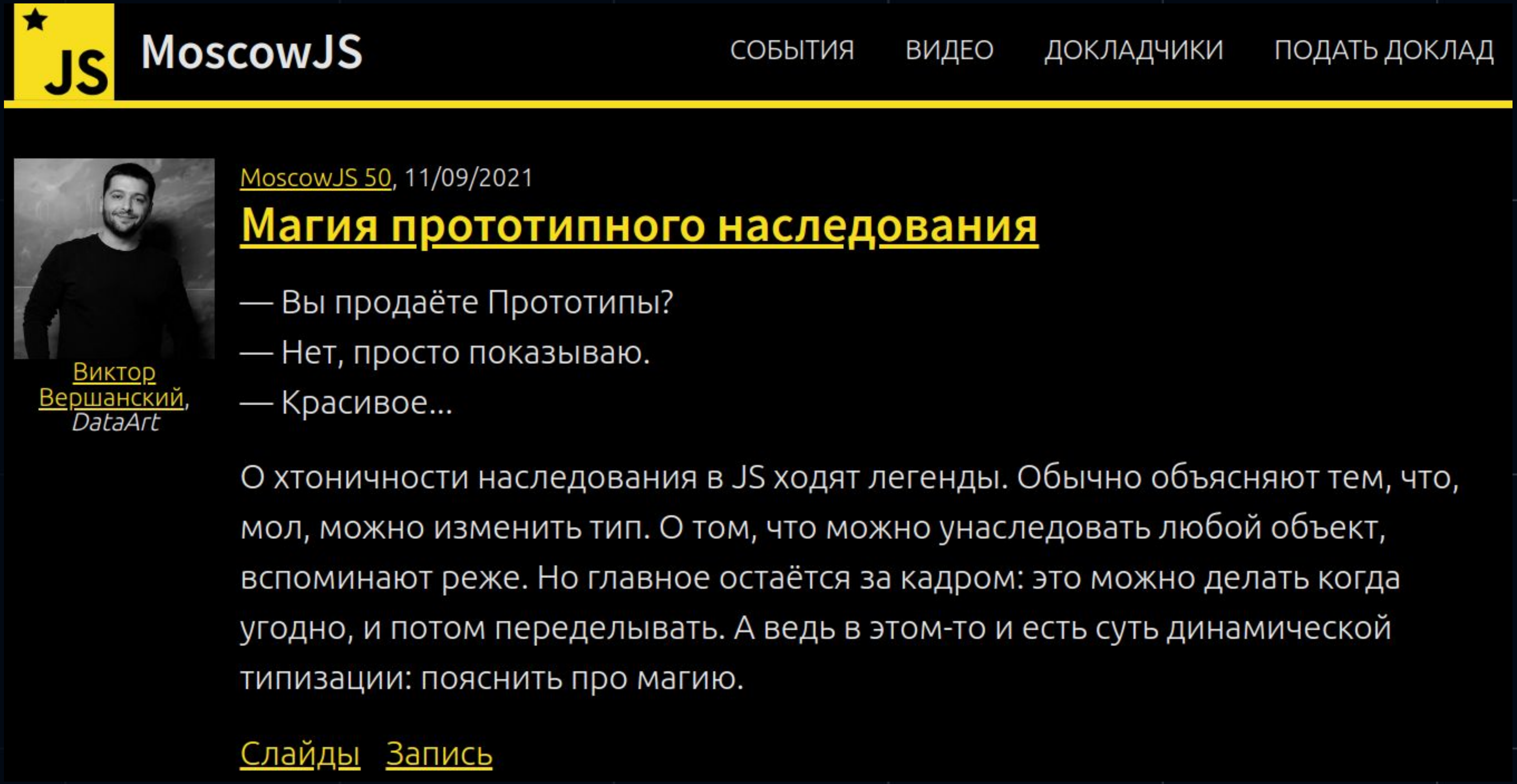


PiterJS #54 NodeJS SPb

Онлайн митап
10 декабря 19:00 - 20:30

Pro .prototype'ы

Moonshine Spiritual talks



Виктор Вершанский, *DataArt*

MoscowJS 50, 11/09/2021

Магия прототипного наследования

— Вы продаёте Прототипы?
— Нет, просто показываю.
— Красивое...

О хтоничности наследования в JS ходят легенды. Обычно объясняют тем, что, мол, можно изменить тип. О том, что можно унаследовать любой объект, вспоминают реже. Но главное остаётся за кадром: это можно делать когда угодно, и потом переделывать. А ведь в этом-то и есть суть динамической типизации: пояснить про магию.

Слайды Запись

Moonshine Spiritual talks

Chronotope: await Eloquent.Errors

Приглашенные эксперты

Дмитрий Махнёв
JetBrains

Спикеры

Виктор Вершанский

Moonshine Spiritual talks

Time Script

incremental computation

Moonshine Spiritual talks

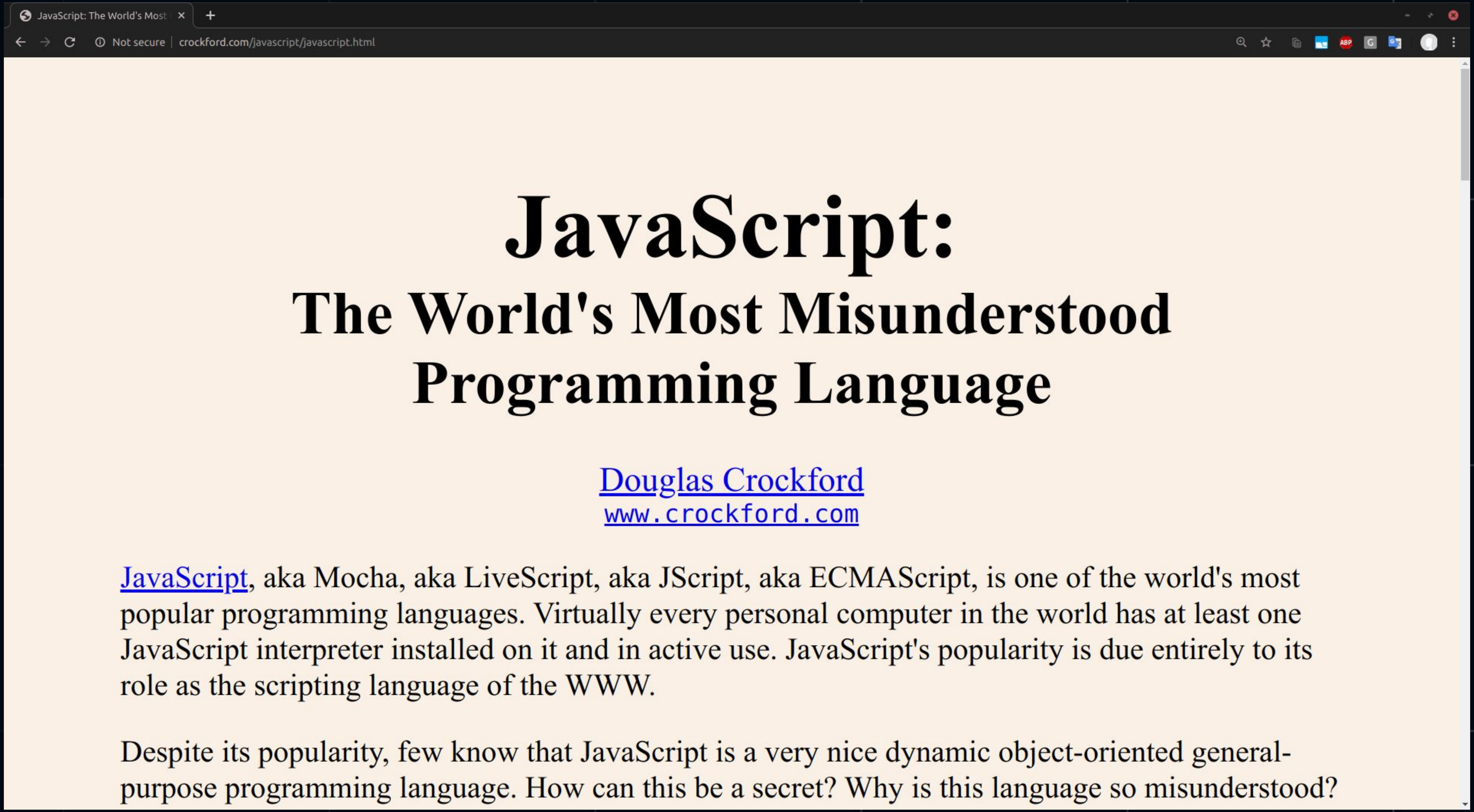
Moonshine Spiritual talks

Class Based

vs

Functional Based

Class vs Function


JavaScript: The World's Most
Not secure | crockford.com/javascript/javascript.html

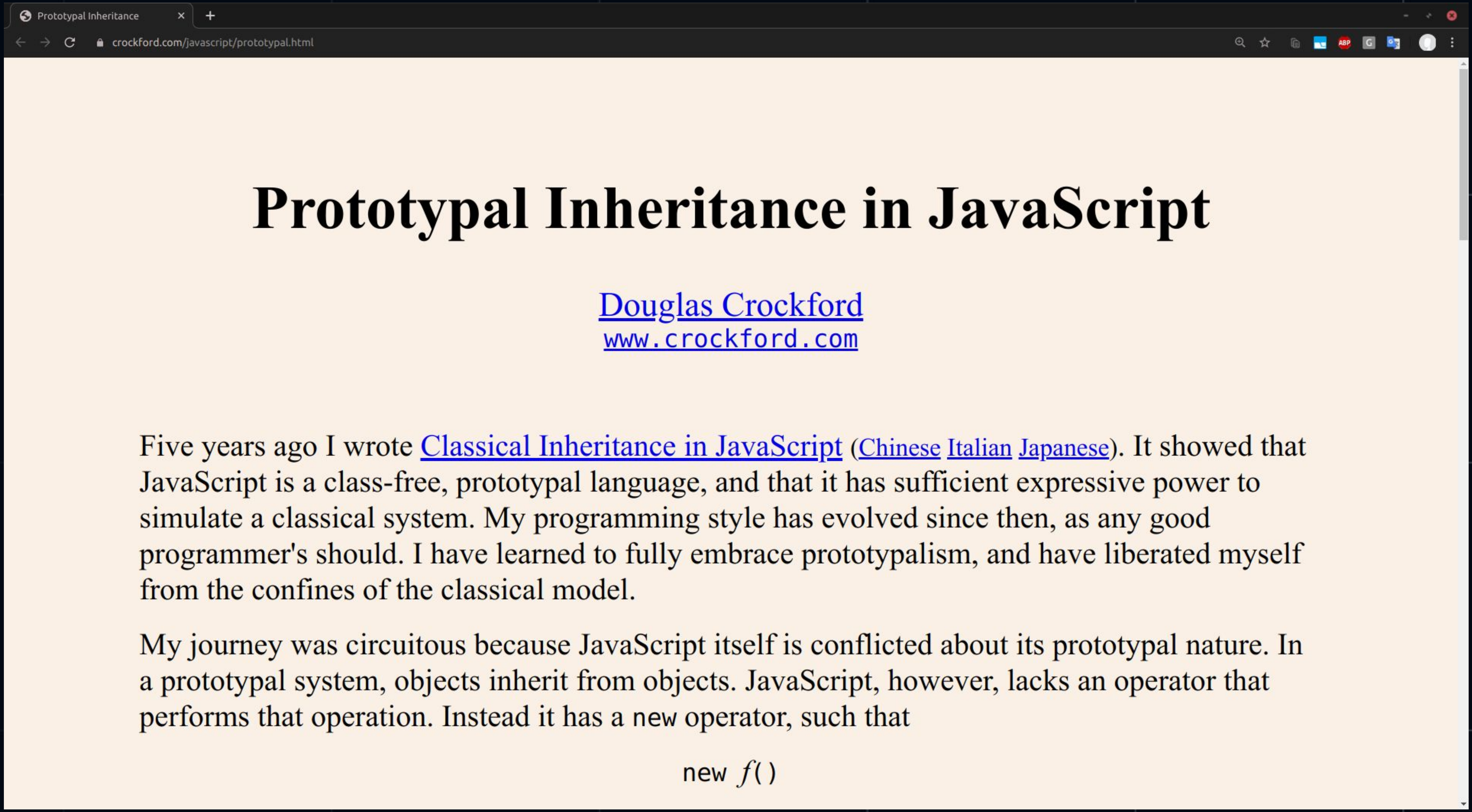
JavaScript:
The World's Most Misunderstood Programming Language

Douglas Crockford
www.crockford.com

JavaScript, aka Mocha, aka LiveScript, aka JScript, aka ECMAScript, is one of the world's most popular programming languages. Virtually every personal computer in the world has at least one JavaScript interpreter installed on it and in active use. JavaScript's popularity is due entirely to its role as the scripting language of the WWW.

Despite its popularity, few know that JavaScript is a very nice dynamic object-oriented general-purpose programming language. How can this be a secret? Why is this language so misunderstood?

Class vs Function



Prototypal Inheritance

crockford.com/javascript/prototypal.html

Prototypal Inheritance in JavaScript

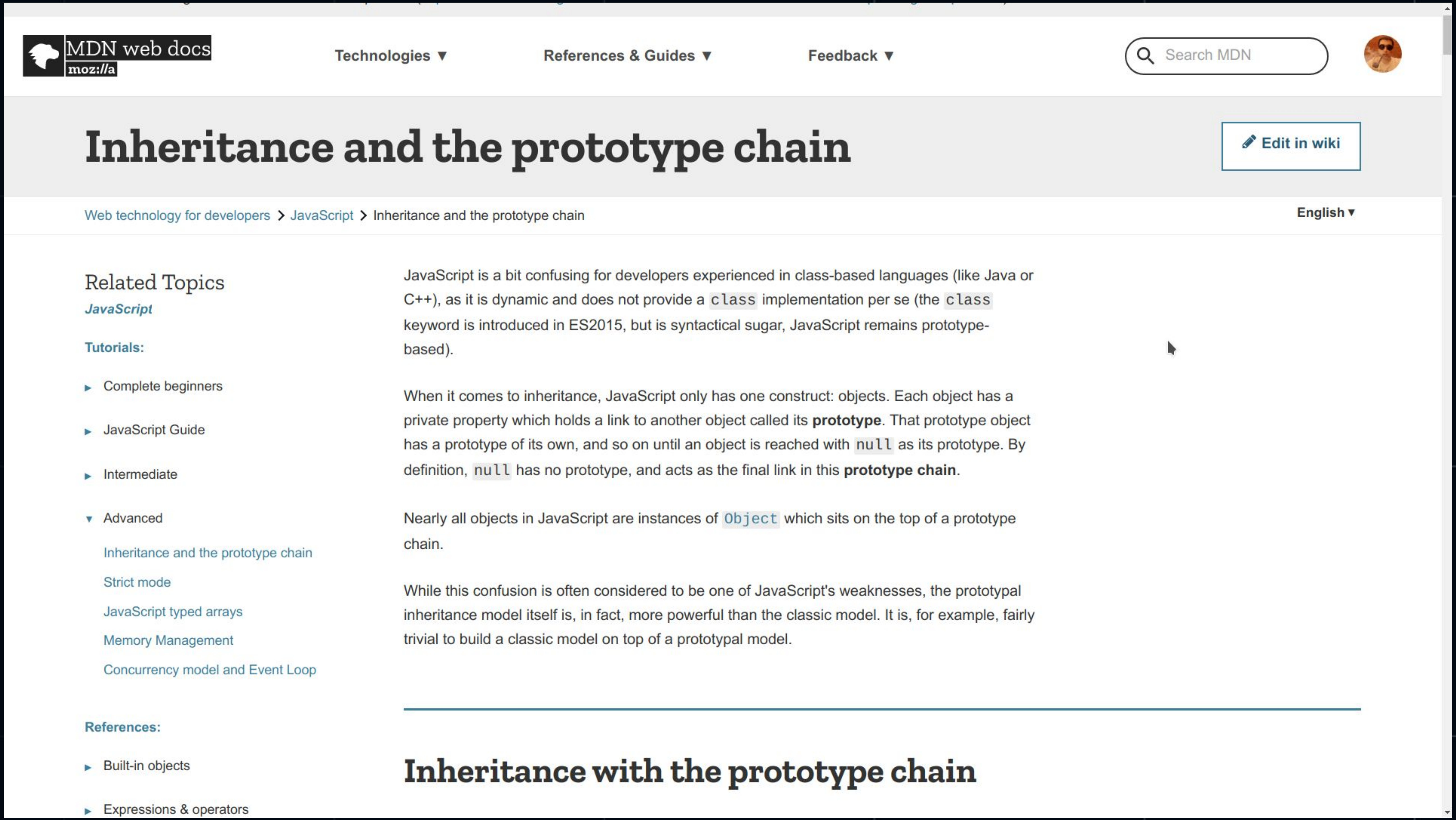
Douglas Crockford
www.crockford.com

Five years ago I wrote Classical Inheritance in JavaScript (Chinese Italian Japanese). It showed that JavaScript is a class-free, prototypal language, and that it has sufficient expressive power to simulate a classical system. My programming style has evolved since then, as any good programmer's should. I have learned to fully embrace prototypalism, and have liberated myself from the confines of the classical model.

My journey was circuitous because JavaScript itself is conflicted about its prototypal nature. In a prototypal system, objects inherit from objects. JavaScript, however, lacks an operator that performs that operation. Instead it has a `new` operator, such that

`new` *f*()

Class vs Function



Inheritance and the prototype chain

Web technology for developers > JavaScript > Inheritance and the prototype chain

Related Topics
JavaScript

Tutorials:

- Complete beginners
- JavaScript Guide
- Intermediate
- Advanced
  - Inheritance and the prototype chain
  - Strict mode
  - JavaScript typed arrays
  - Memory Management
  - Concurrency model and Event Loop

References:

- Built-in objects
- Expressions & operators

JavaScript is a bit confusing for developers experienced in class-based languages (like Java or C++), as it is dynamic and does not provide a `class` implementation per se (the `class` keyword is introduced in ES2015, but is syntactical sugar, JavaScript remains prototype-based).

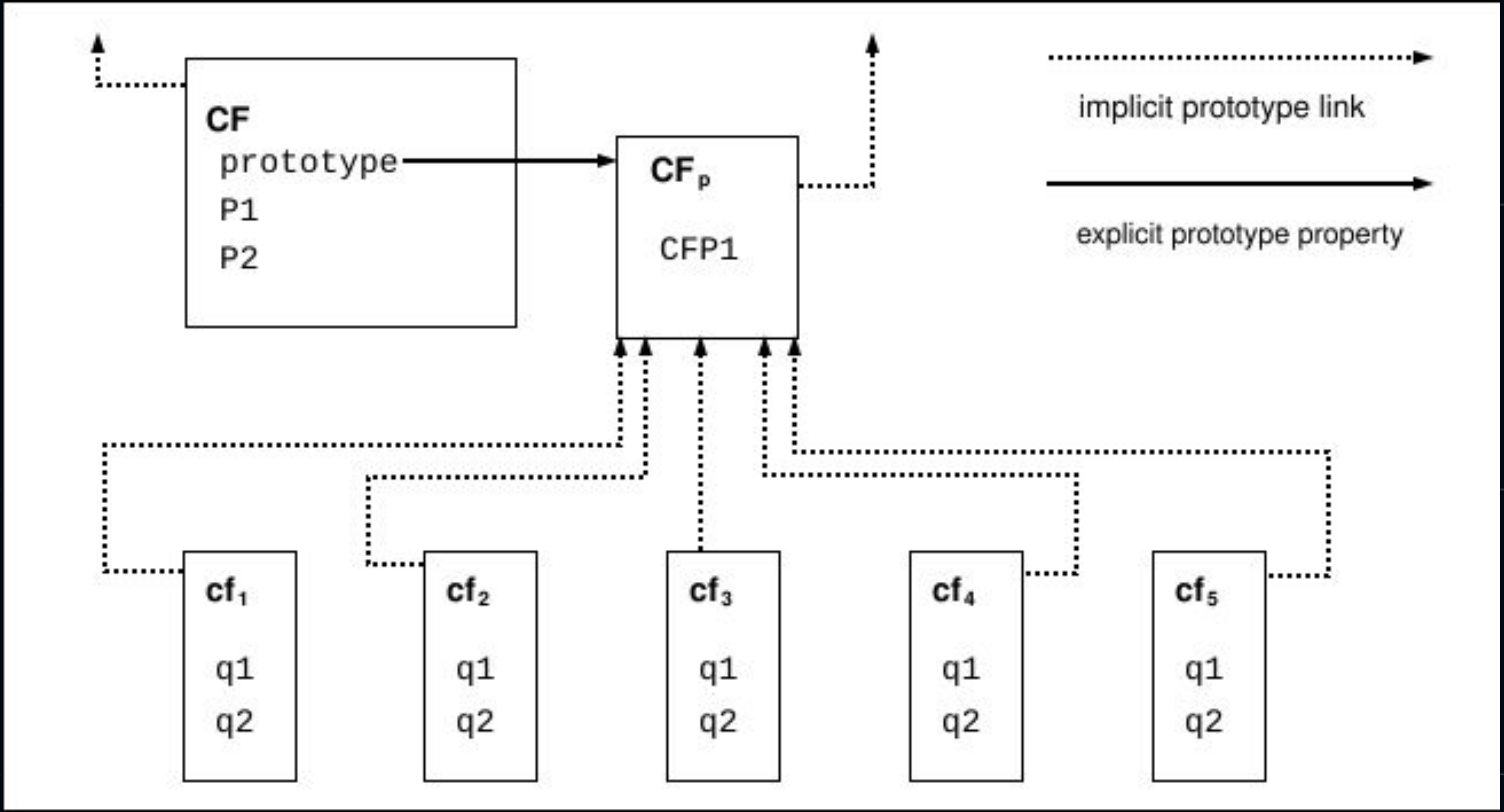
When it comes to inheritance, JavaScript only has one construct: objects. Each object has a private property which holds a link to another object called its **prototype**. That prototype object has a prototype of its own, and so on until an object is reached with `null` as its prototype. By definition, `null` has no prototype, and acts as the final link in this **prototype chain**.

Nearly all objects in JavaScript are instances of `Object` which sits on the top of a prototype chain.

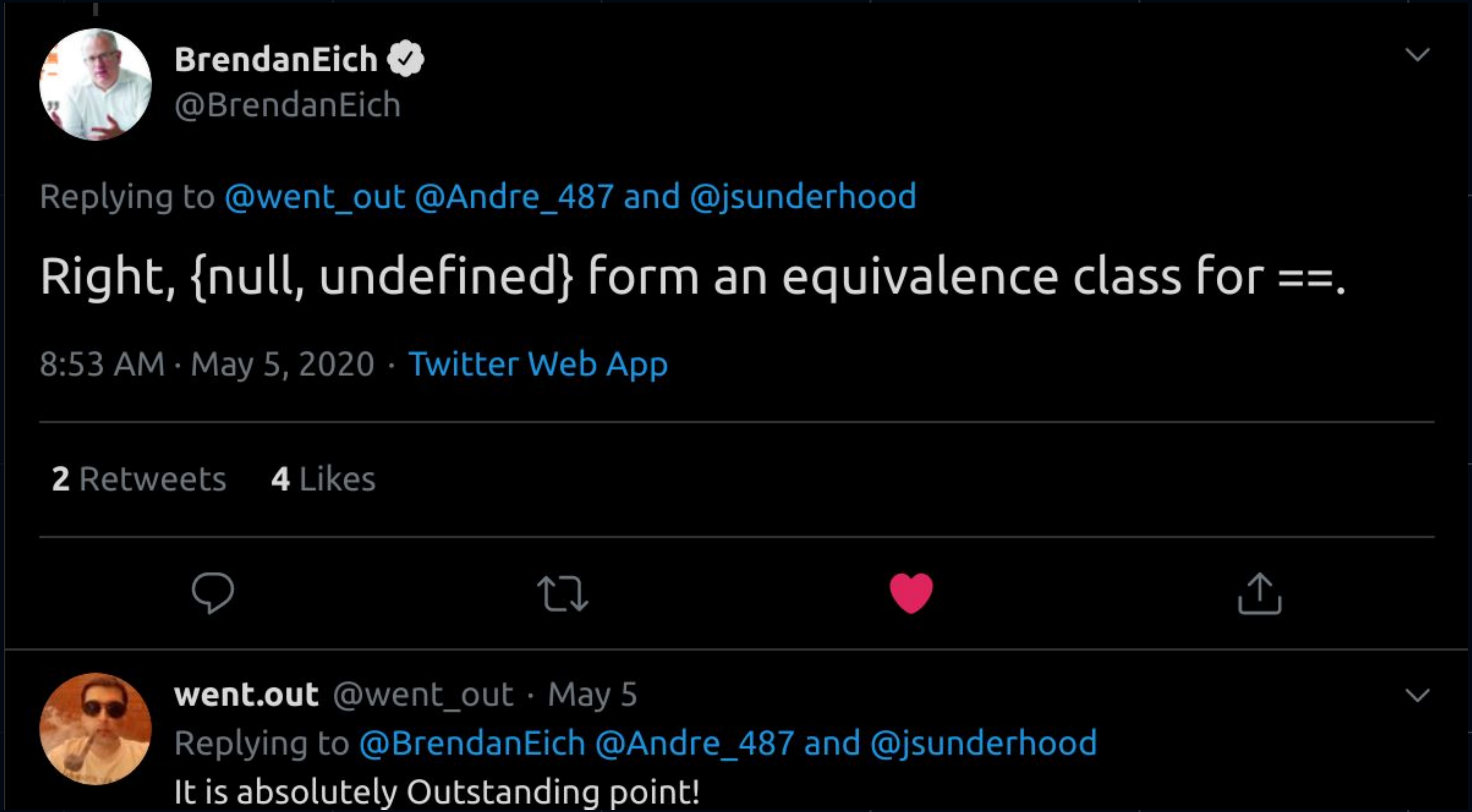
While this confusion is often considered to be one of JavaScript's weaknesses, the prototypal inheritance model itself is, in fact, more powerful than the classic model. It is, for example, fairly trivial to build a classic model on top of a prototypal model.

Inheritance with the prototype chain

Class vs Function



Class vs Function


BrendanEich
@BrendanEich

Replying to @went_out @Andre_487 and @jsunderhood

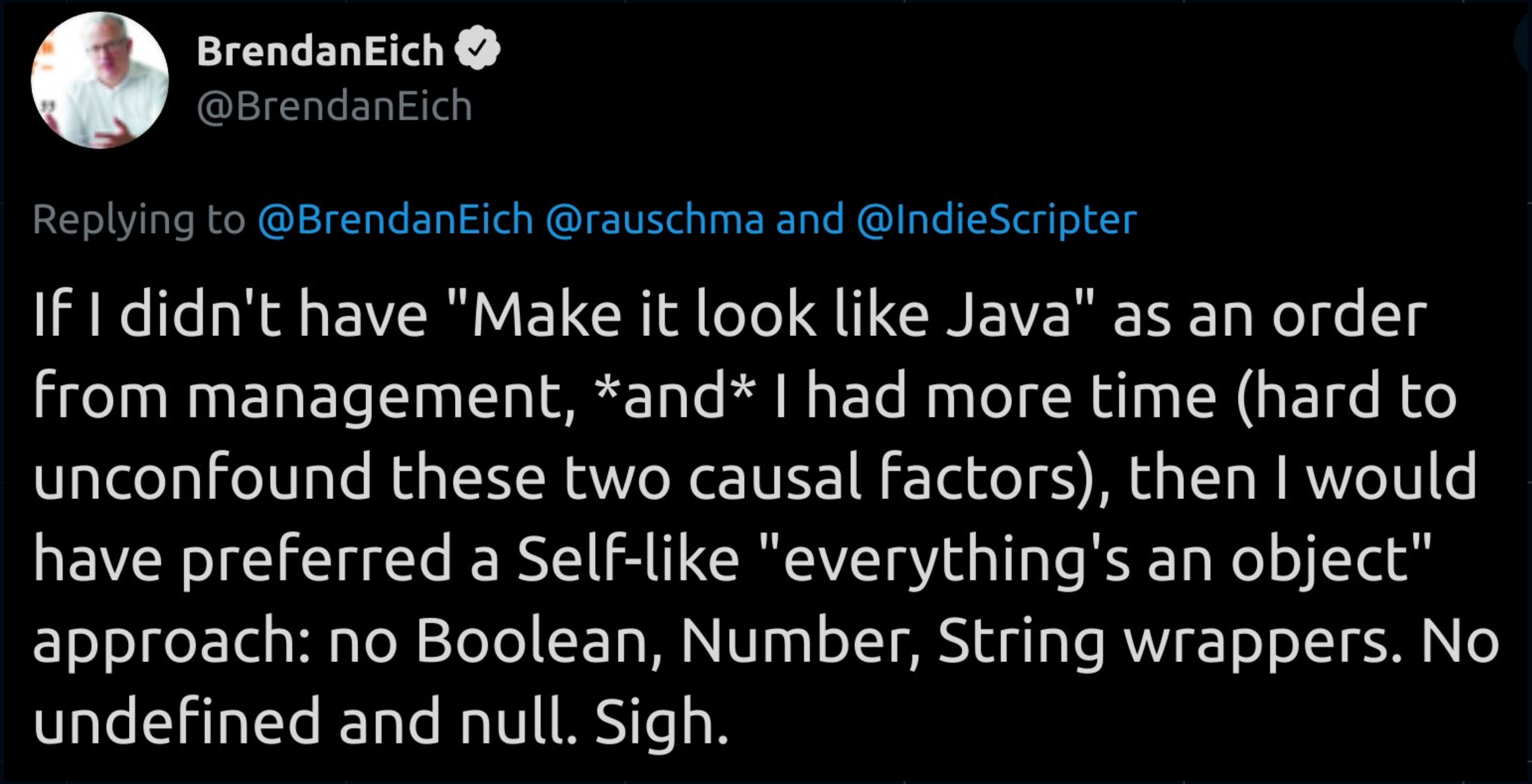
Right, {null, undefined} form an equivalence class for ==.

8:53 AM · May 5, 2020 · Twitter Web App

2 Retweets 4 Likes

went.out @went_out · May 5
Replying to @BrendanEich @Andre_487 and @jsunderhood
It is absolutely Outstanding point!

Class vs Function

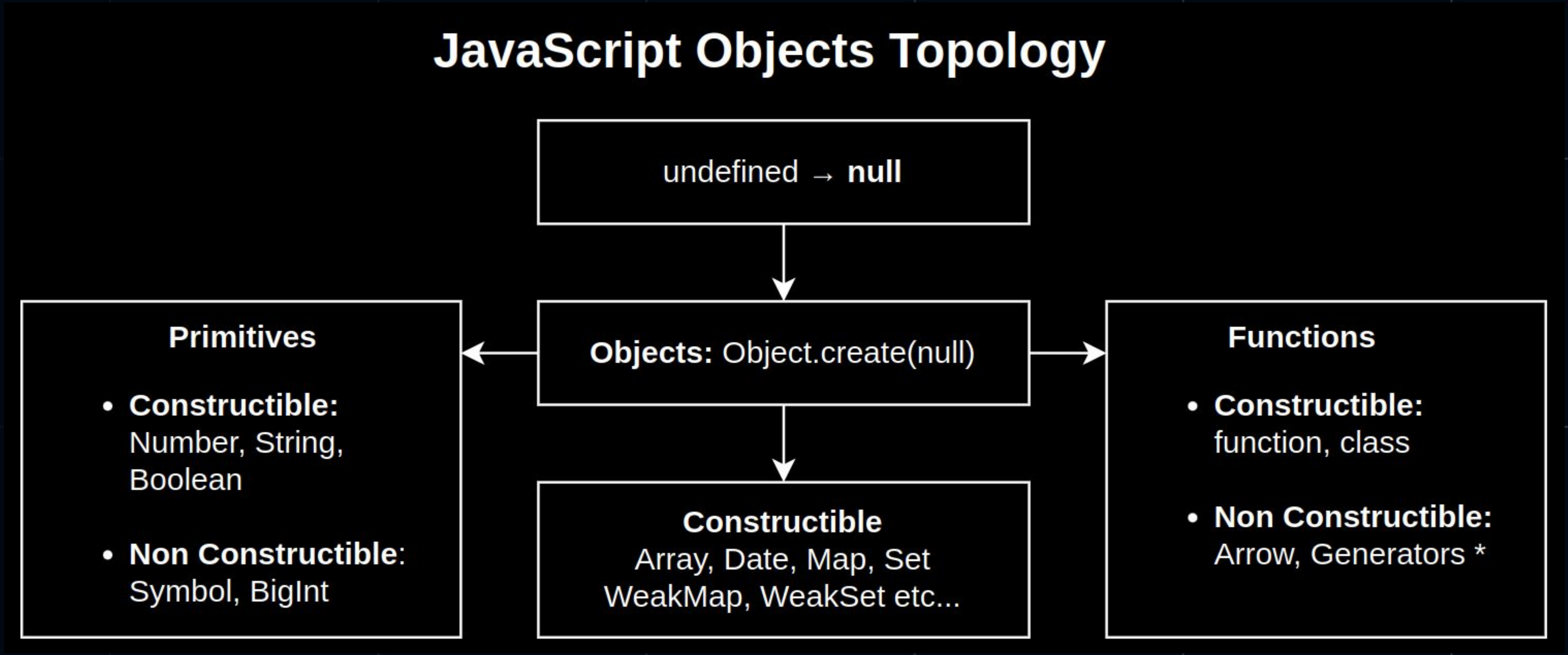


BrendanEich
@BrendanEich

Replying to @BrendanEich @rauschma and @IndieScripter

If I didn't have "Make it look like Java" as an order from management, *and* I had more time (hard to unconfound these two causal factors), then I would have preferred a Self-like "everything's an object" approach: no Boolean, Number, String wrappers. No undefined and null. Sigh.

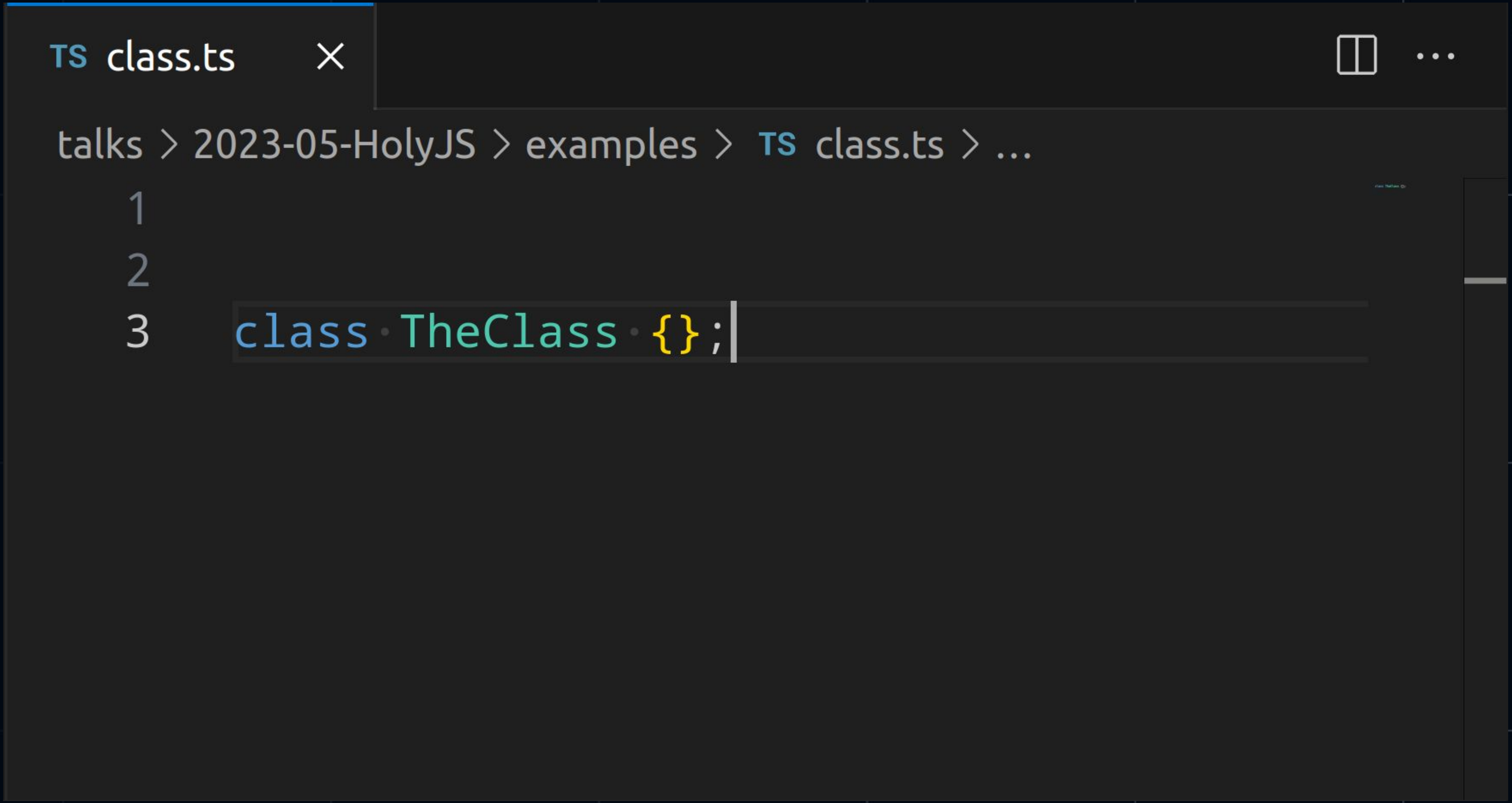
Class vs Function



JavaScript Objects Topology
undefined → **null**
Objects: Object.create(null)
Primitives
• **Constructible:** Number, String, Boolean
• **Non Constructible**: Symbol, BigInt
Constructible Array, Date, Map, Set WeakMap, WeakSet etc...
Functions
• **Constructible:** function, class
• **Non Constructible:** Arrow, Generators *

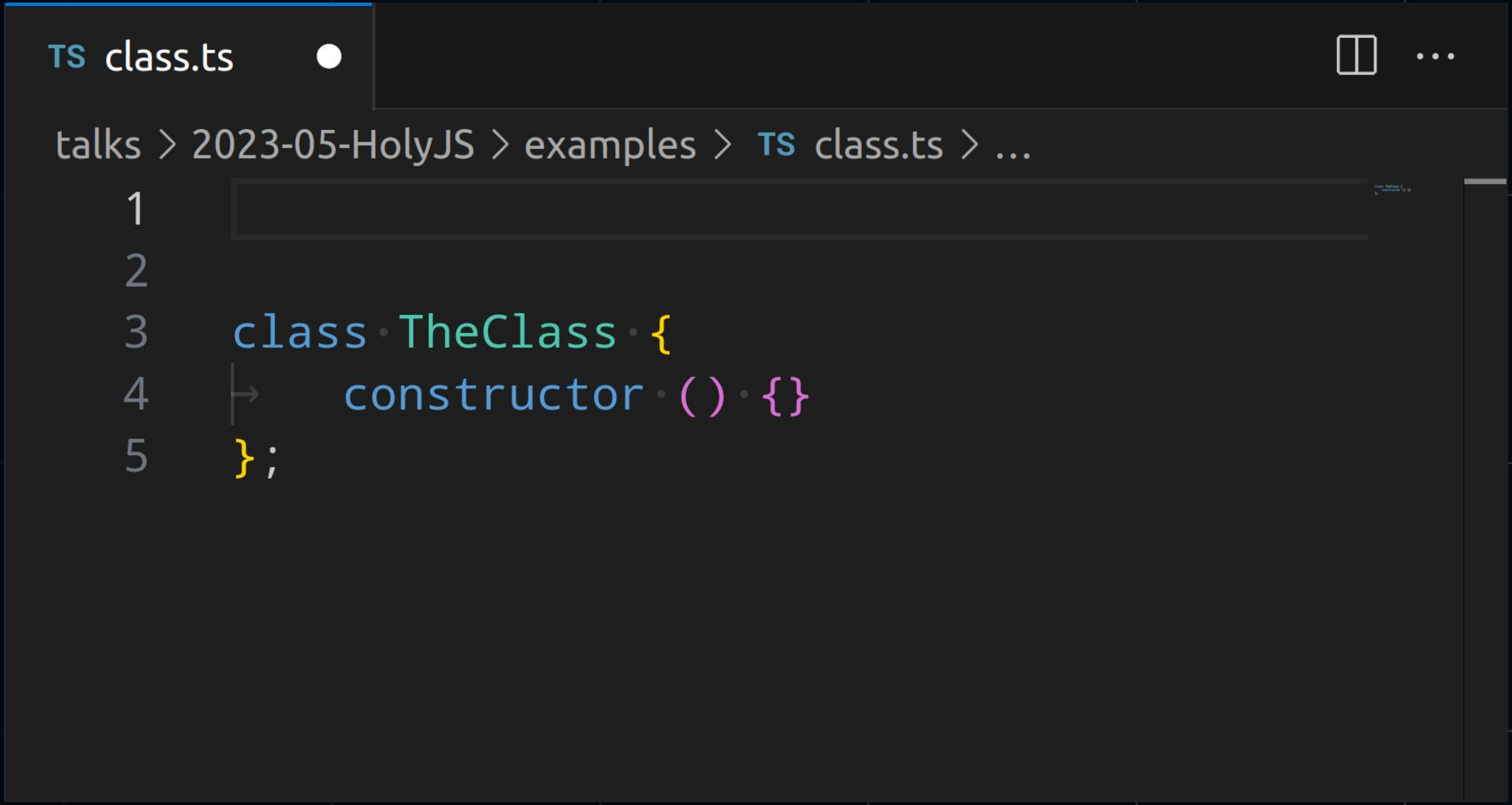
Class vs Function

Class vs Function



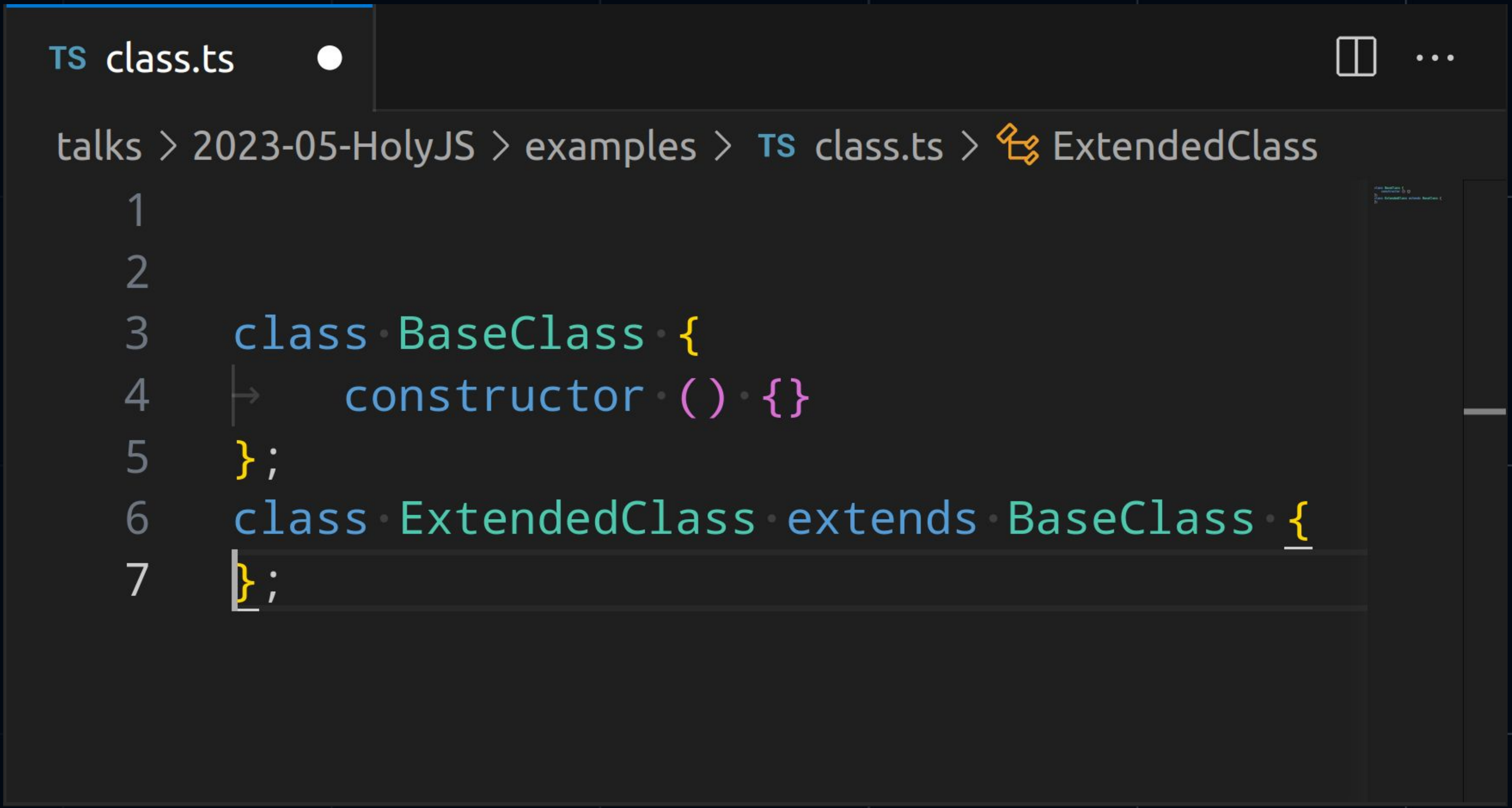
```ts


class TheClass {};
```

Class vs Function



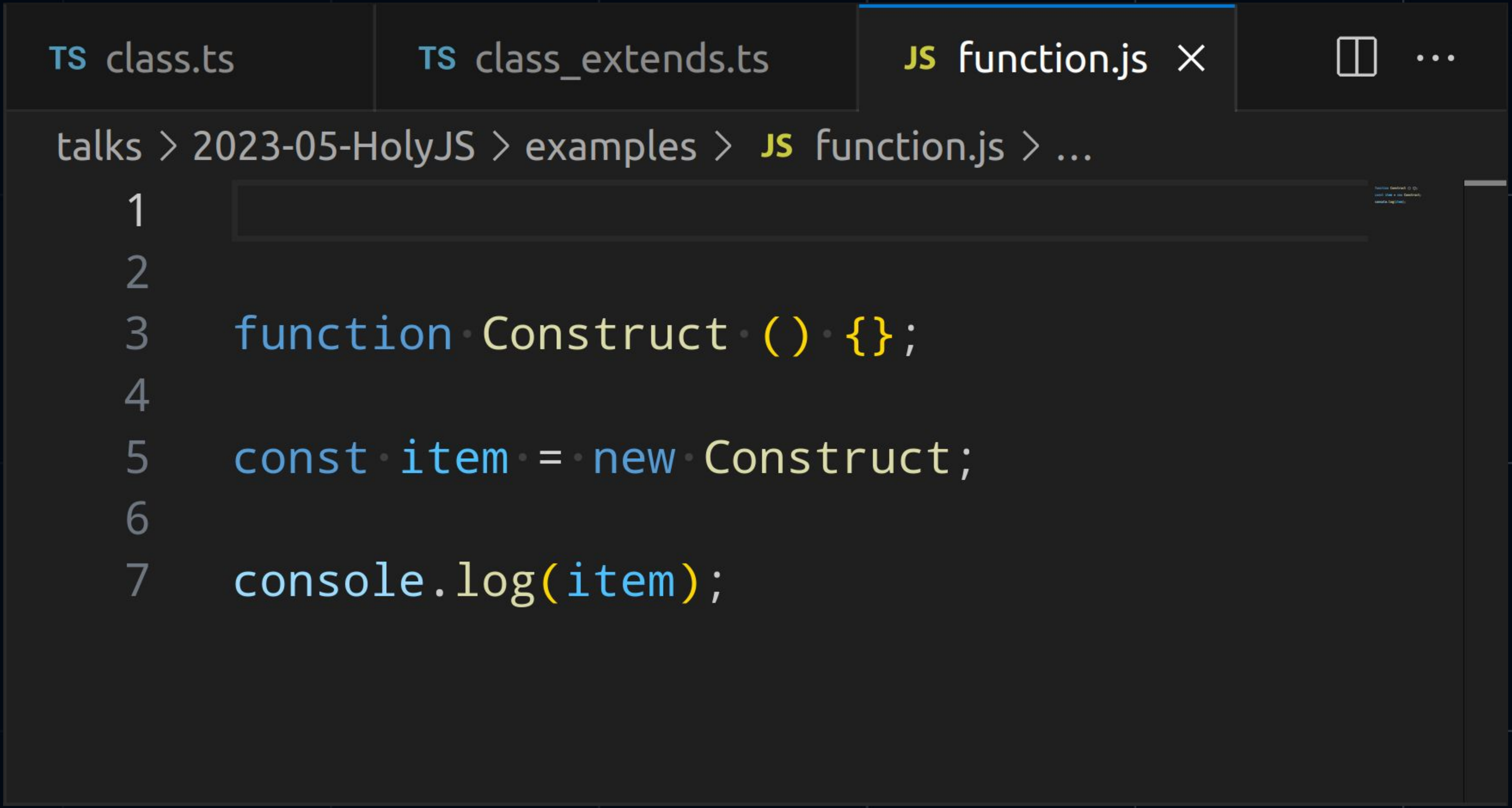
```ts

class TheClass {
    constructor () {}
};
```

Class vs Function


TS class.ts
talks > 2023-05-HolyJS > examples > TS class.ts > ExtendedClass

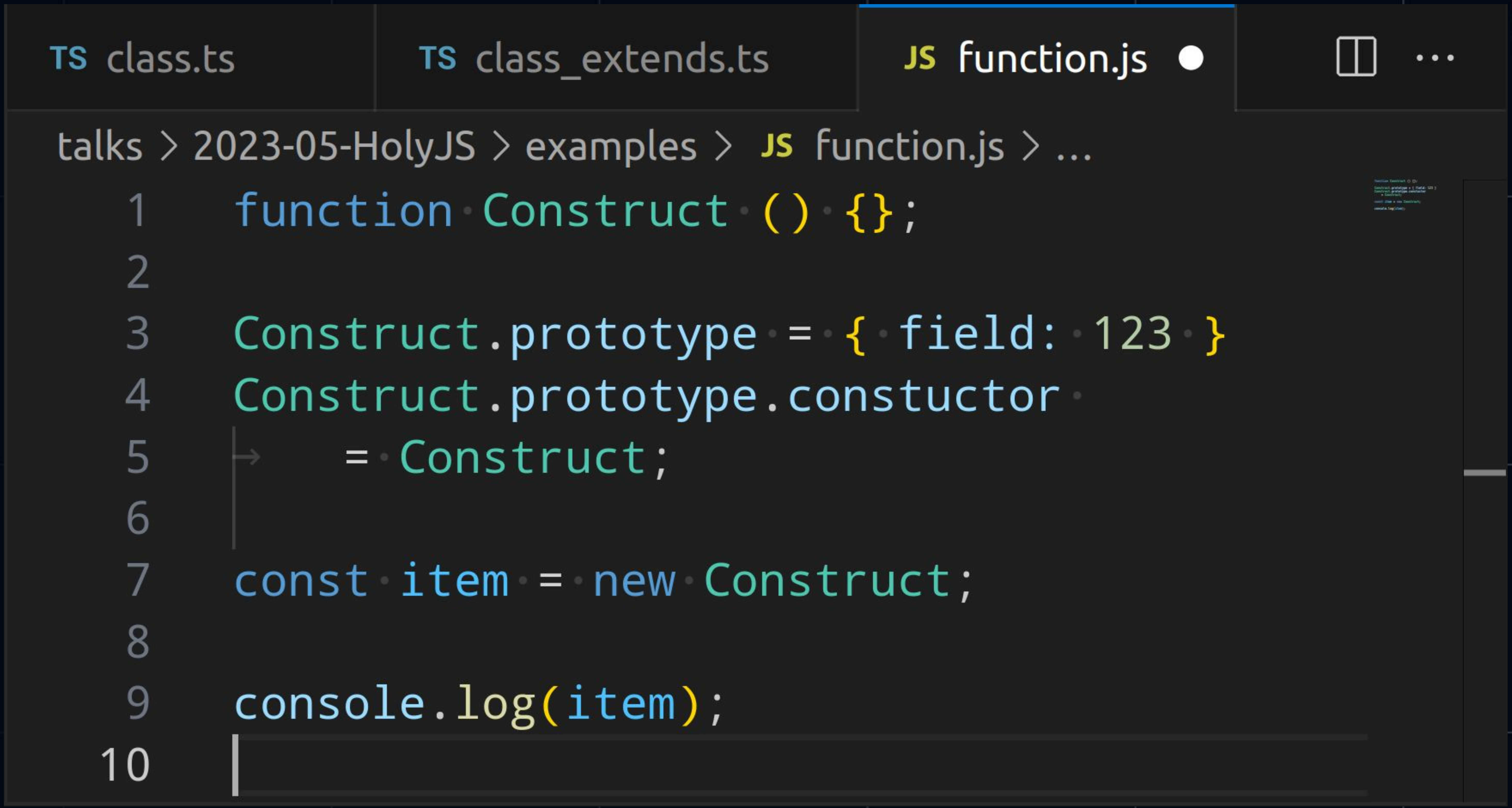
```ts


class BaseClass {
    constructor () {}
};
class ExtendedClass extends BaseClass {
};
```

Class vs Function



```js

function Construct () {};

const item = new Construct;

console.log(item);
```

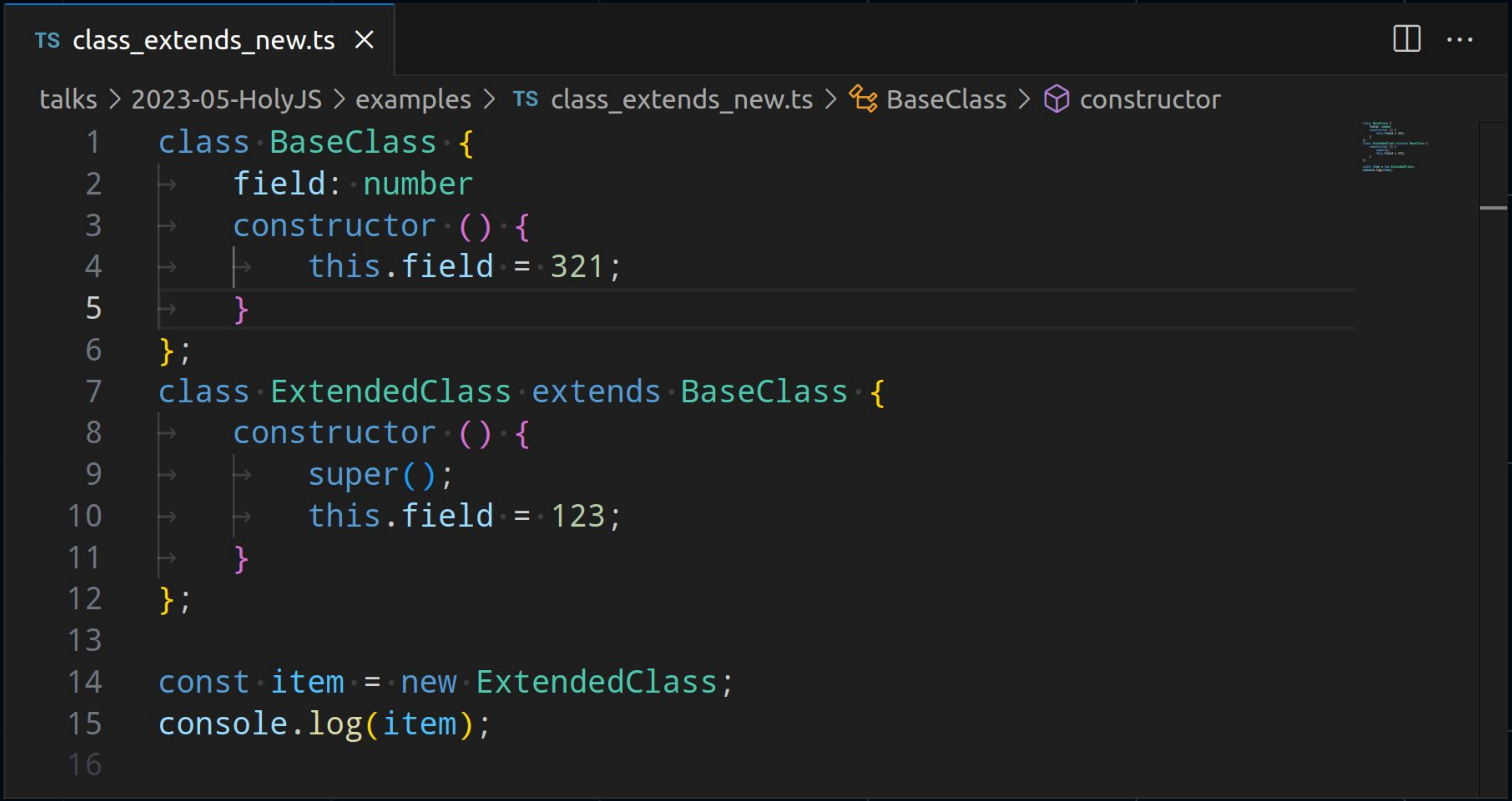
Class vs Function



```
function Construct () {};

Construct.prototype = { field: 123 }
Construct.prototype.constuctor
    = Construct;

const item = new Construct;

console.log(item);
```

time matters the difference

Time Matters

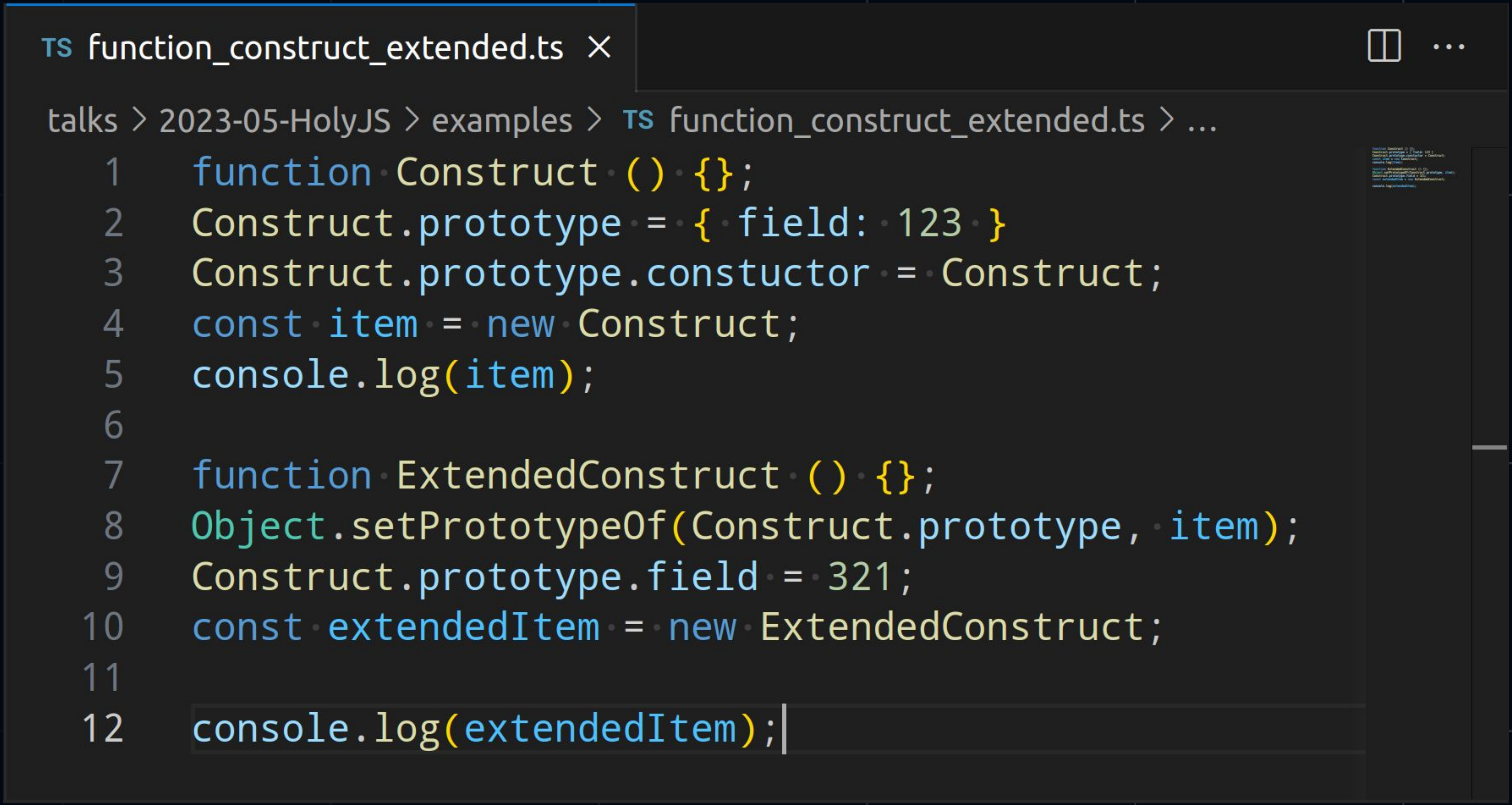


class_extends_new.ts

talks > 2023-05-HolyJS > examples > class_extends_new.ts > BaseClass > constructor

```ts
class BaseClass {
    field: number
    constructor () {
        this.field = 321;
    }
};
class ExtendedClass extends BaseClass {
    constructor () {
        super();
        this.field = 123;
    }
};

const item = new ExtendedClass;
console.log(item);
```

Time Matters


function_construct_extended.ts

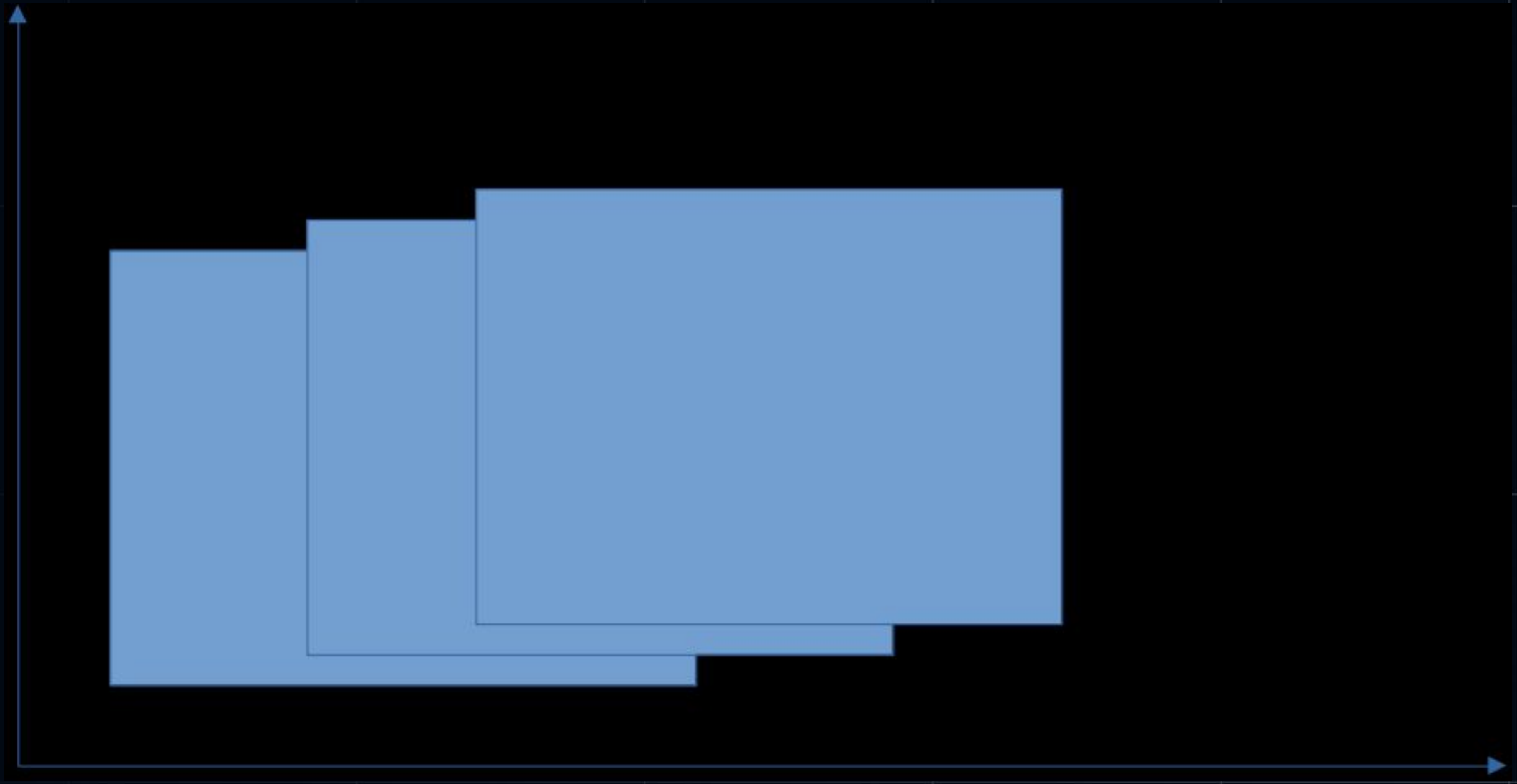
talks > 2023-05-HolyJS > examples > function_construct_extended.ts > ...

```
function Construct () {};
Construct.prototype = { field: 123 }
Construct.prototype.constuctor = Construct;
const item = new Construct;
console.log(item);

function ExtendedConstruct () {};
Object.setPrototypeOf(Construct.prototype, item);
Construct.prototype.field = 321;
const extendedItem = new ExtendedConstruct;

console.log(extendedItem);
```

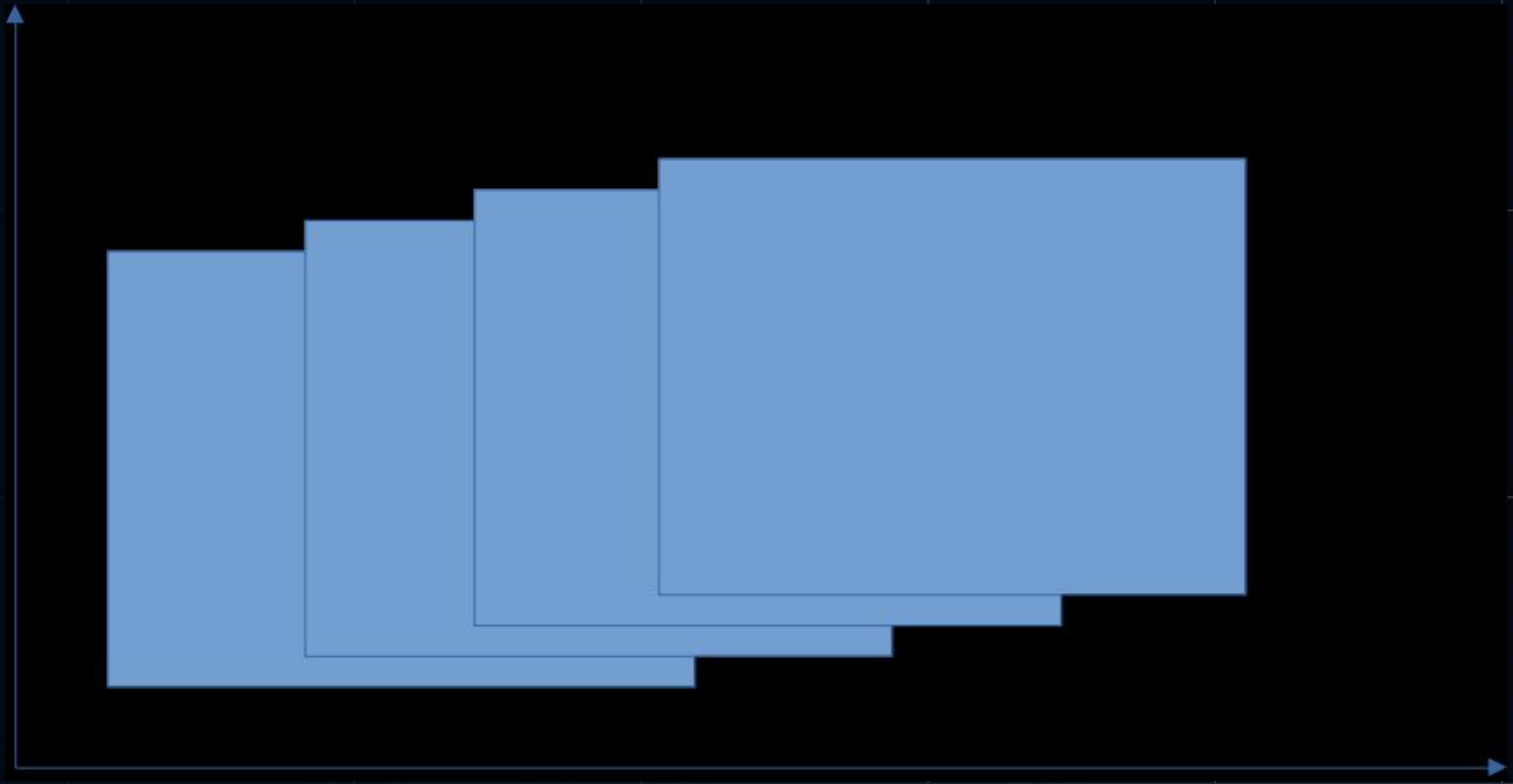
Time Matters

Time Matters

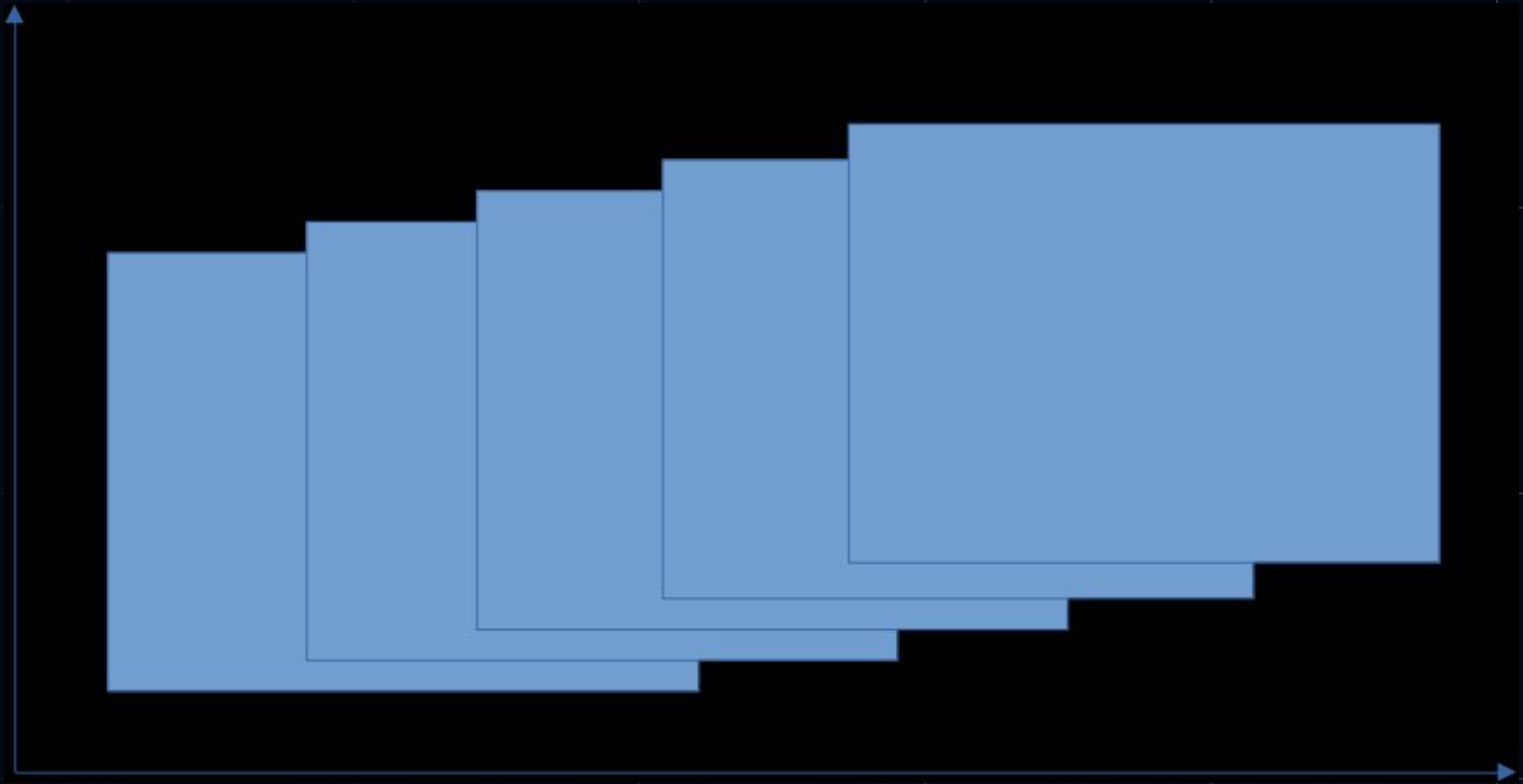
Time Matters



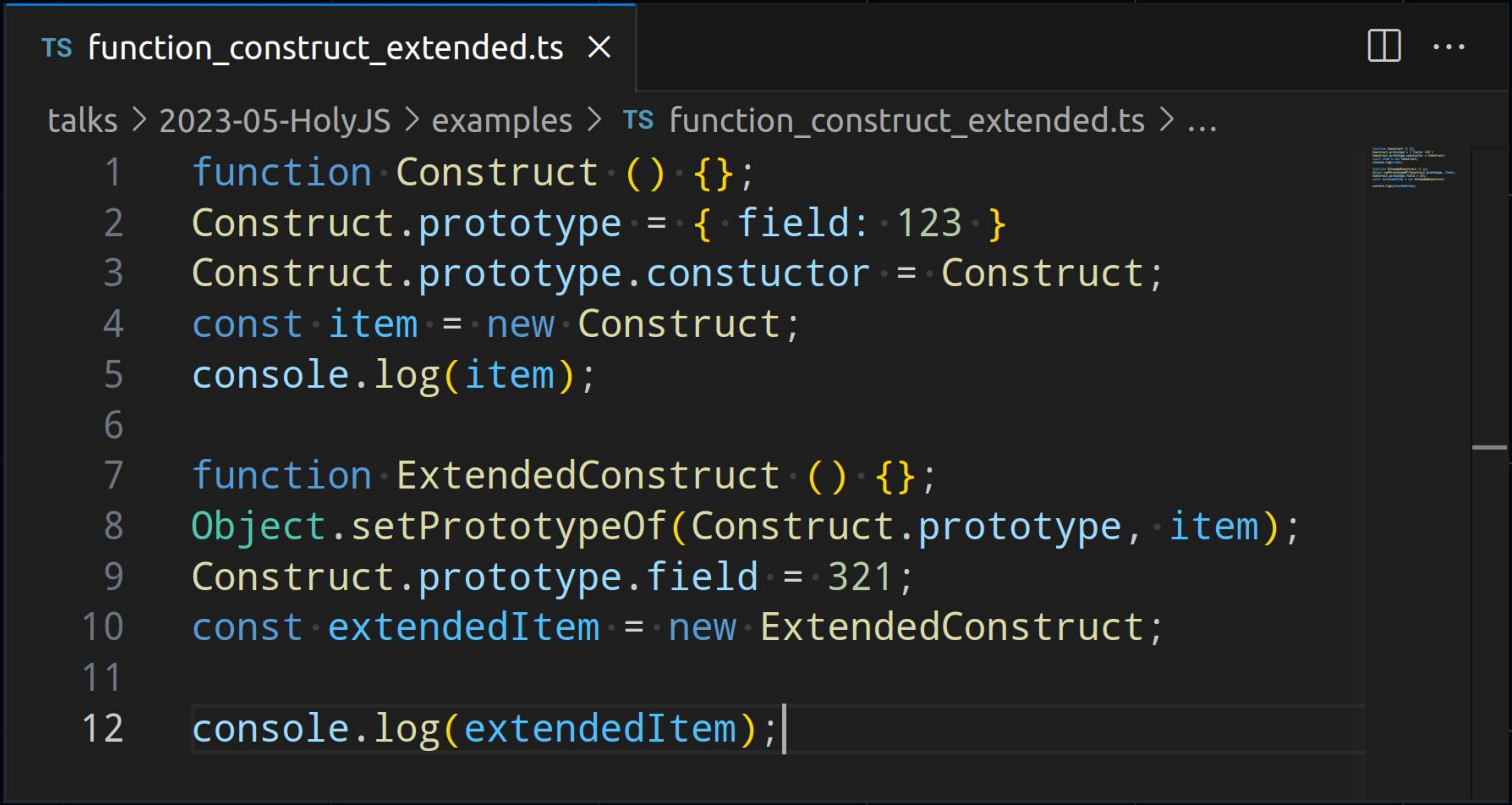
Time Matters



Time Matters



Time Matters

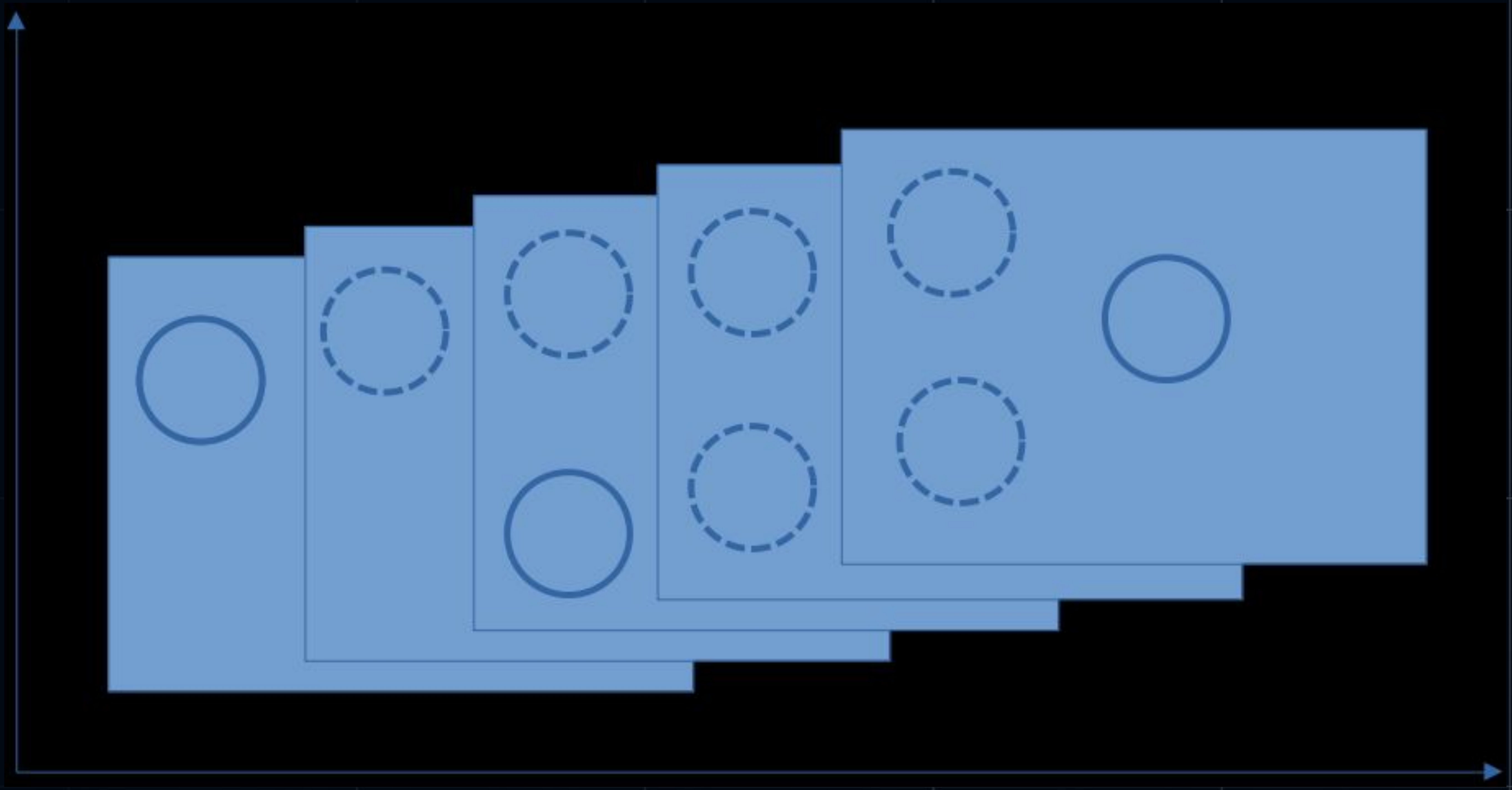


function_construct_extended.ts

talks > 2023-05-HolyJS > examples > function_construct_extended.ts > ...

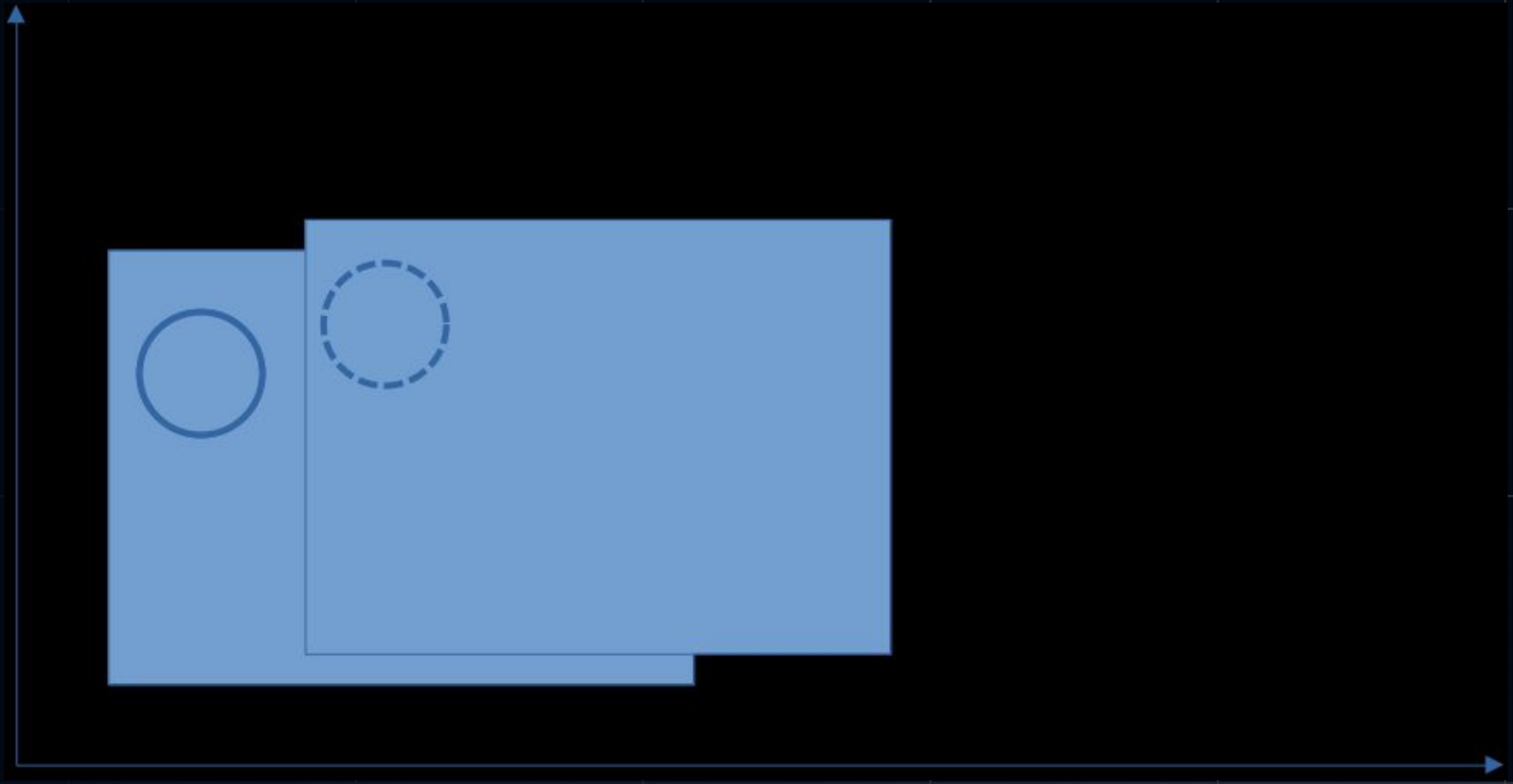
```ts
function Construct () {};
Construct.prototype = { field: 123 }
Construct.prototype.constuctor = Construct;
const item = new Construct;
console.log(item);

function ExtendedConstruct () {};
Object.setPrototypeOf(Construct.prototype, item);
Construct.prototype.field = 321;
const extendedItem = new ExtendedConstruct;

console.log(extendedItem);
```

Time Matters

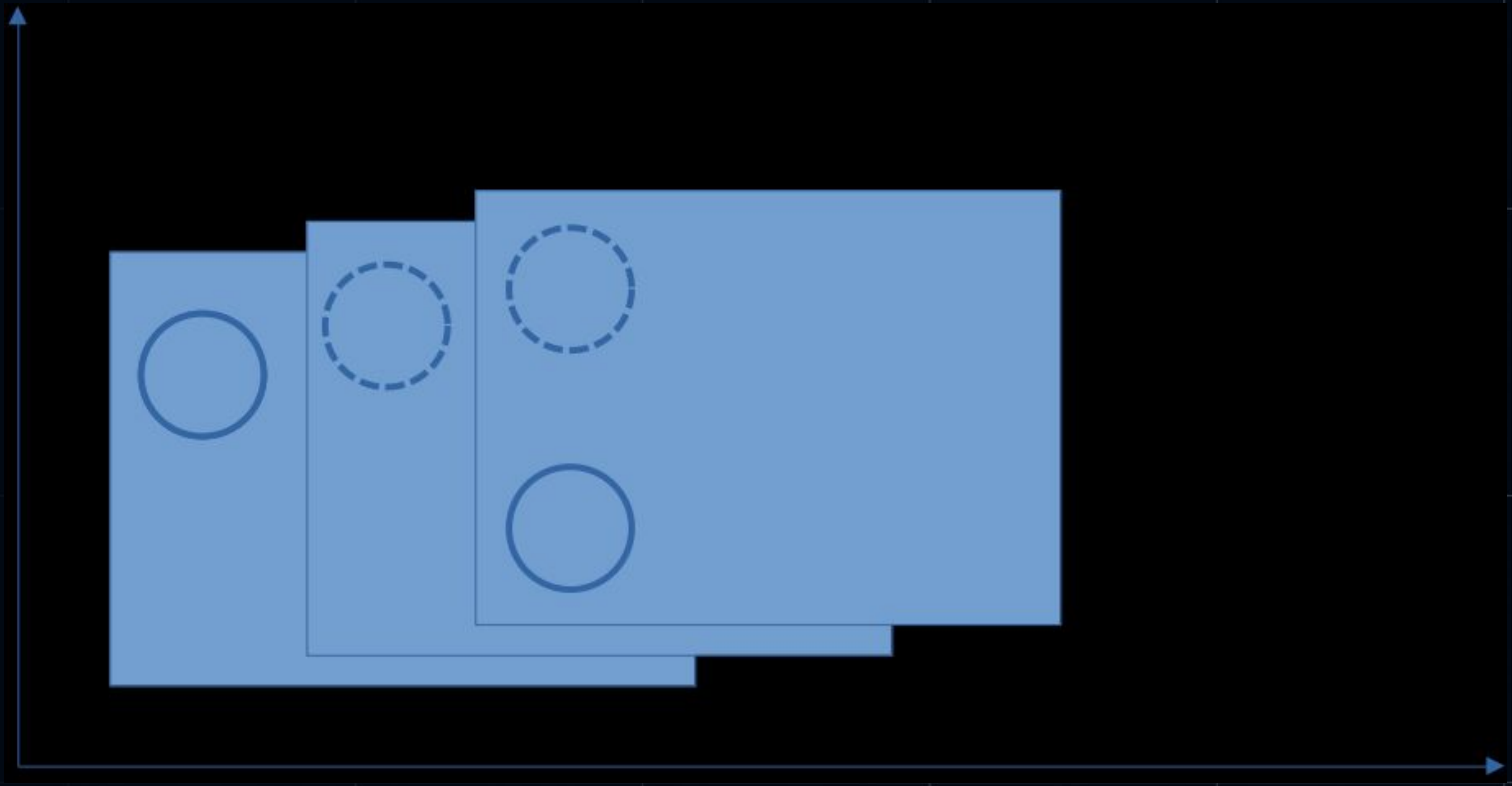


Time Matters

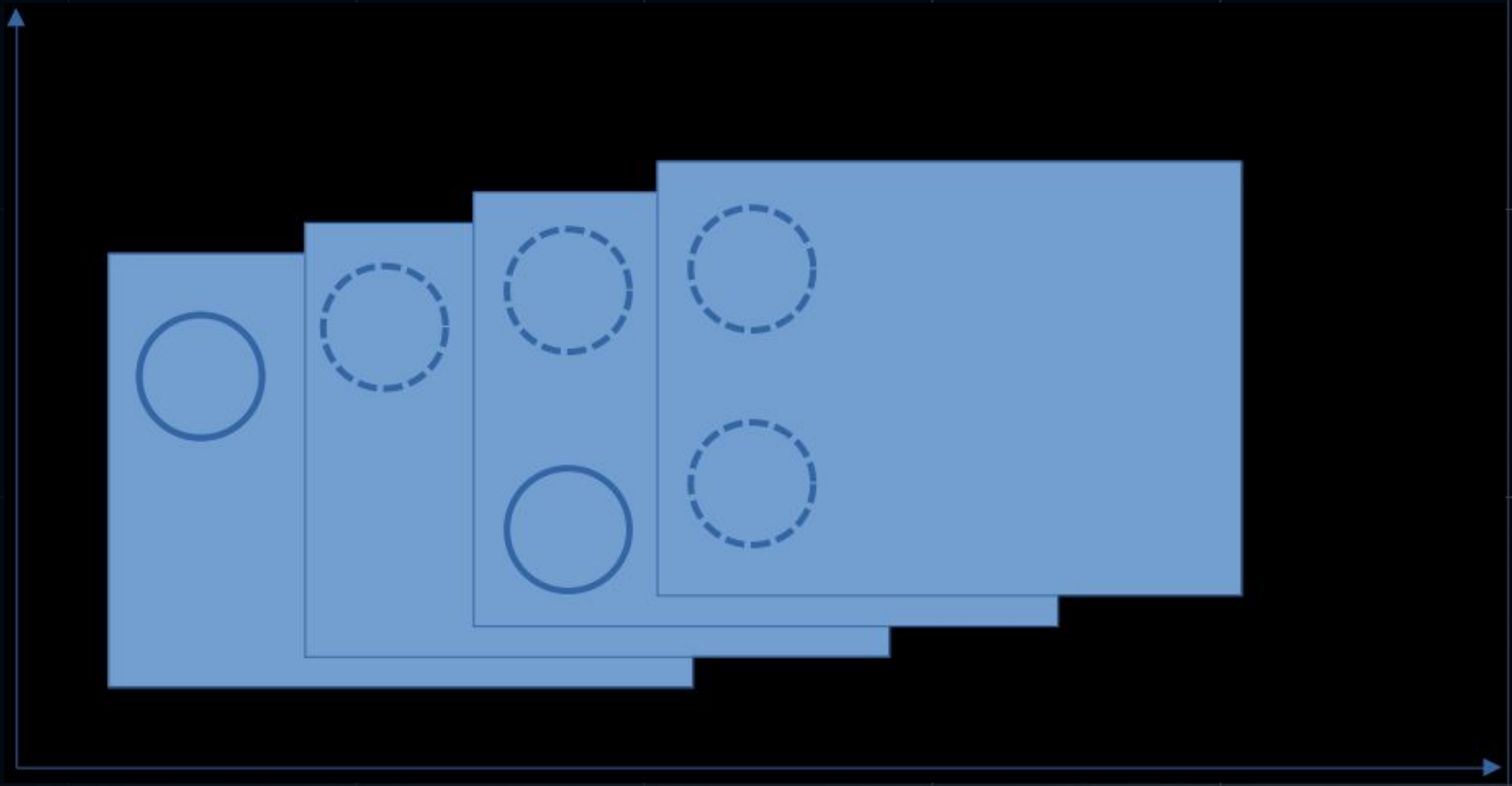
Time Matters



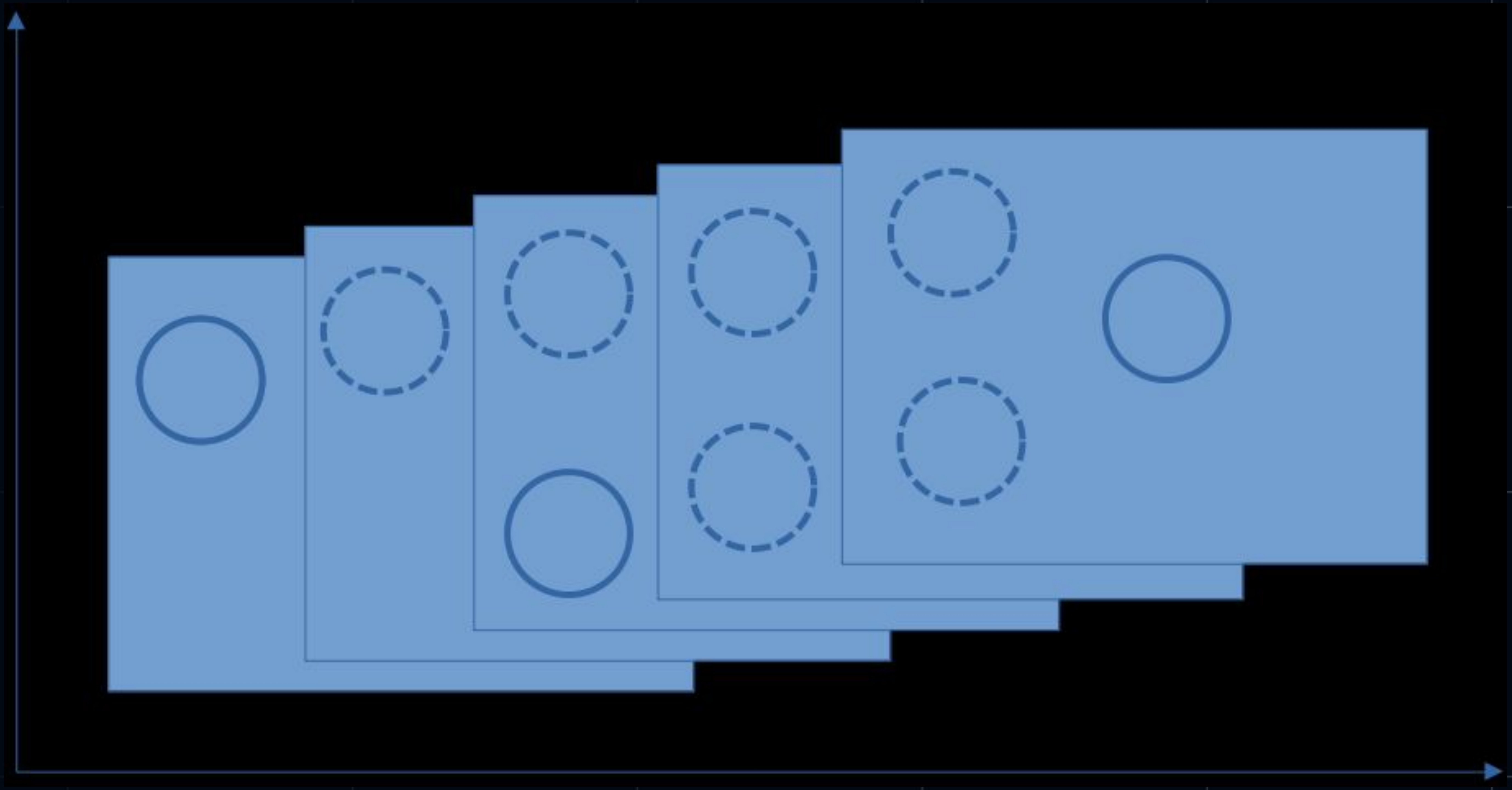
Time Matters



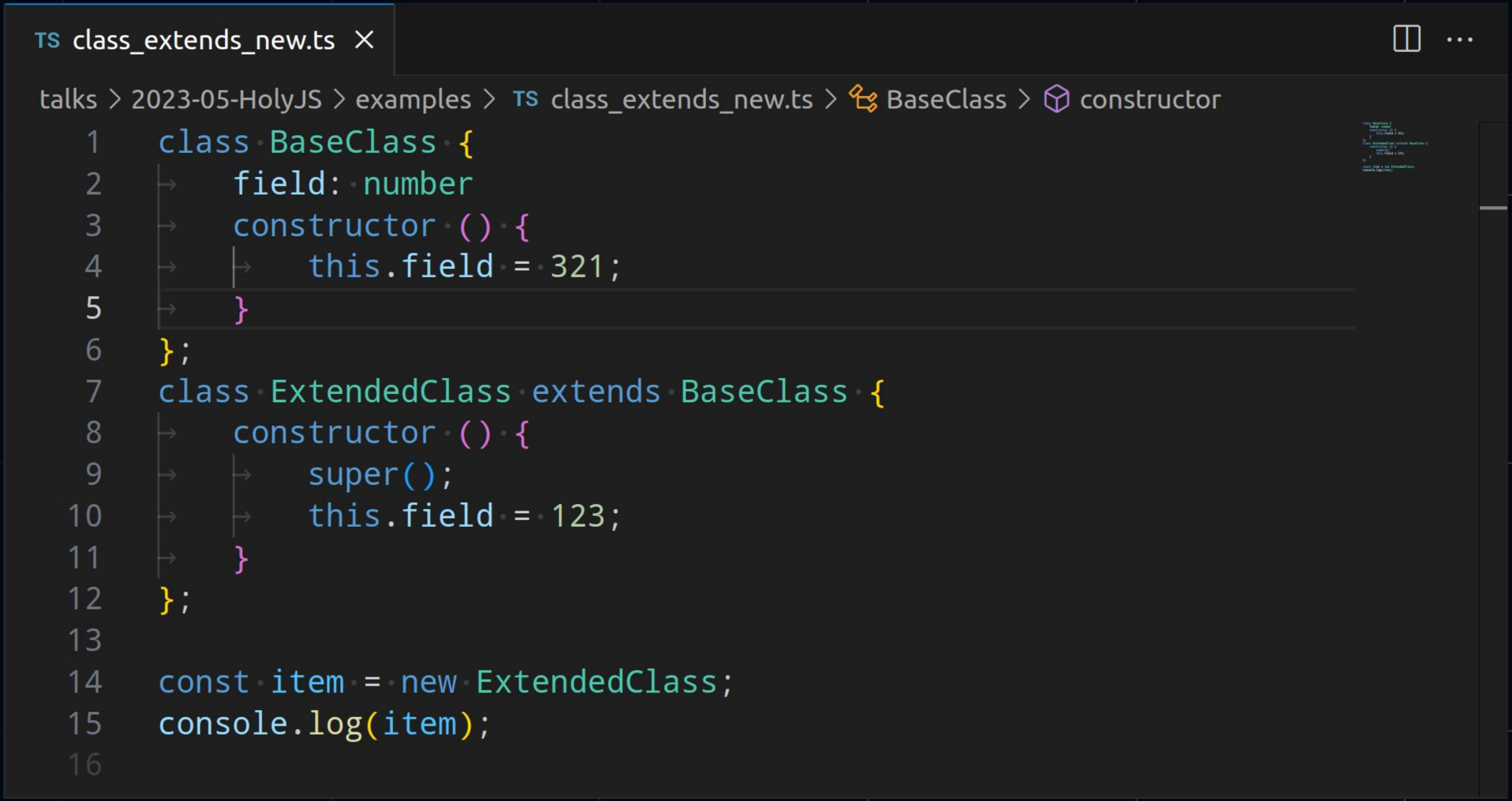
Time Matters



Time Matters



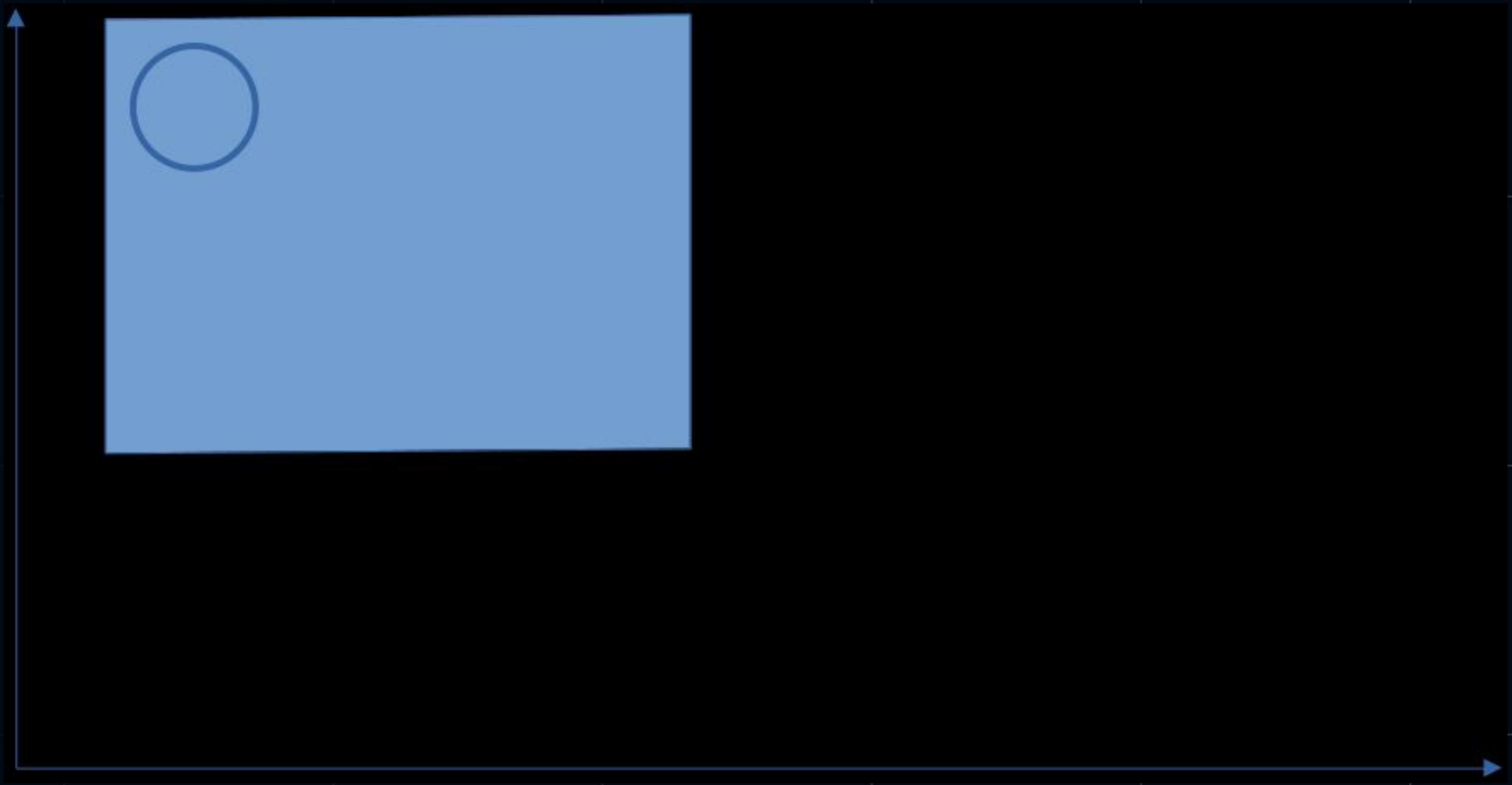
Class vs Function



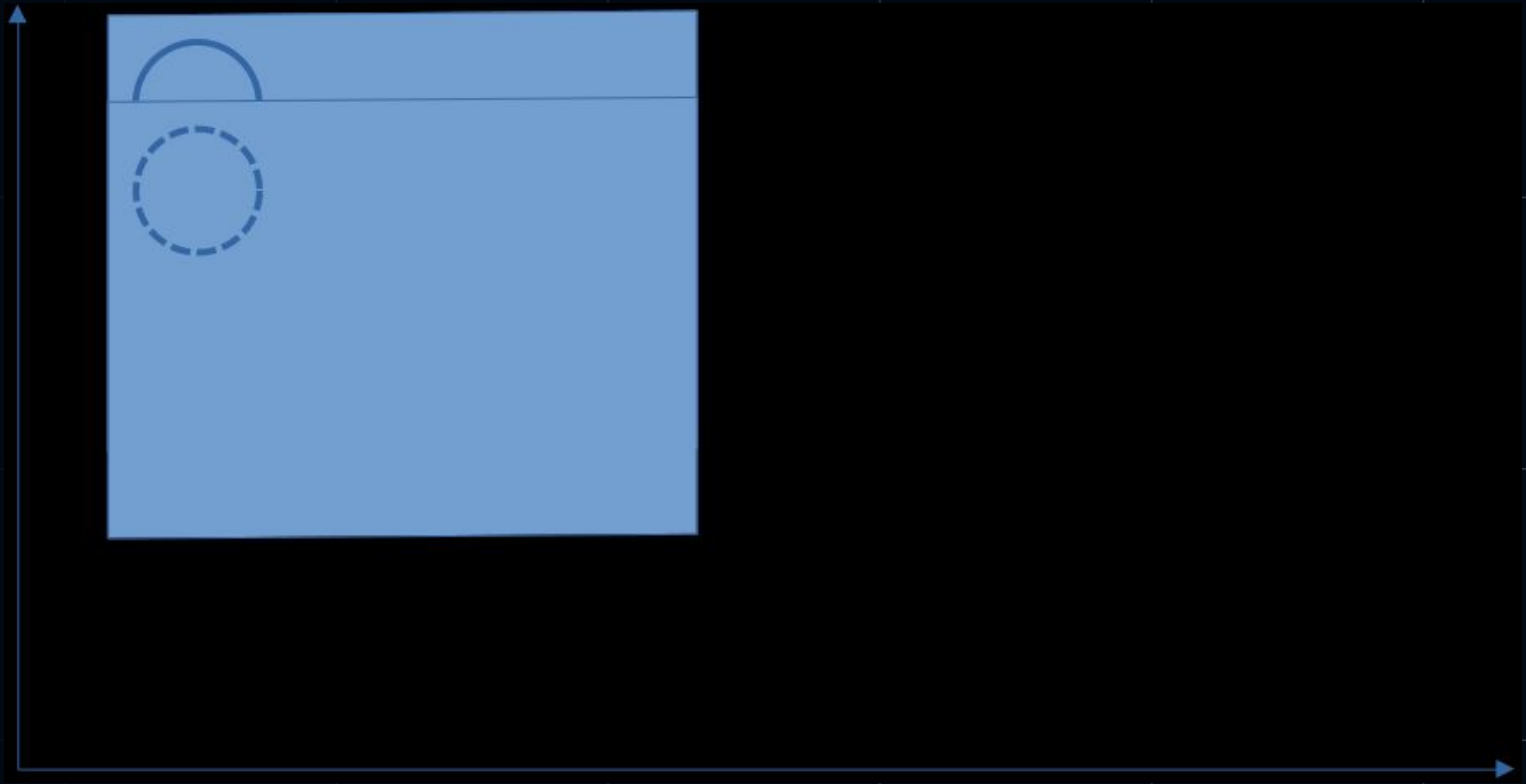
TS class_extends_new.ts
talks > 2023-05-HolyJS > examples > TS class_extends_new.ts > BaseClass > constructor

```ts
class BaseClass {
    field: number
    constructor () {
        this.field = 321;
    }
};
class ExtendedClass extends BaseClass {
    constructor () {
        super();
        this.field = 123;
    }
};

const item = new ExtendedClass;
console.log(item);
```

Time Matters



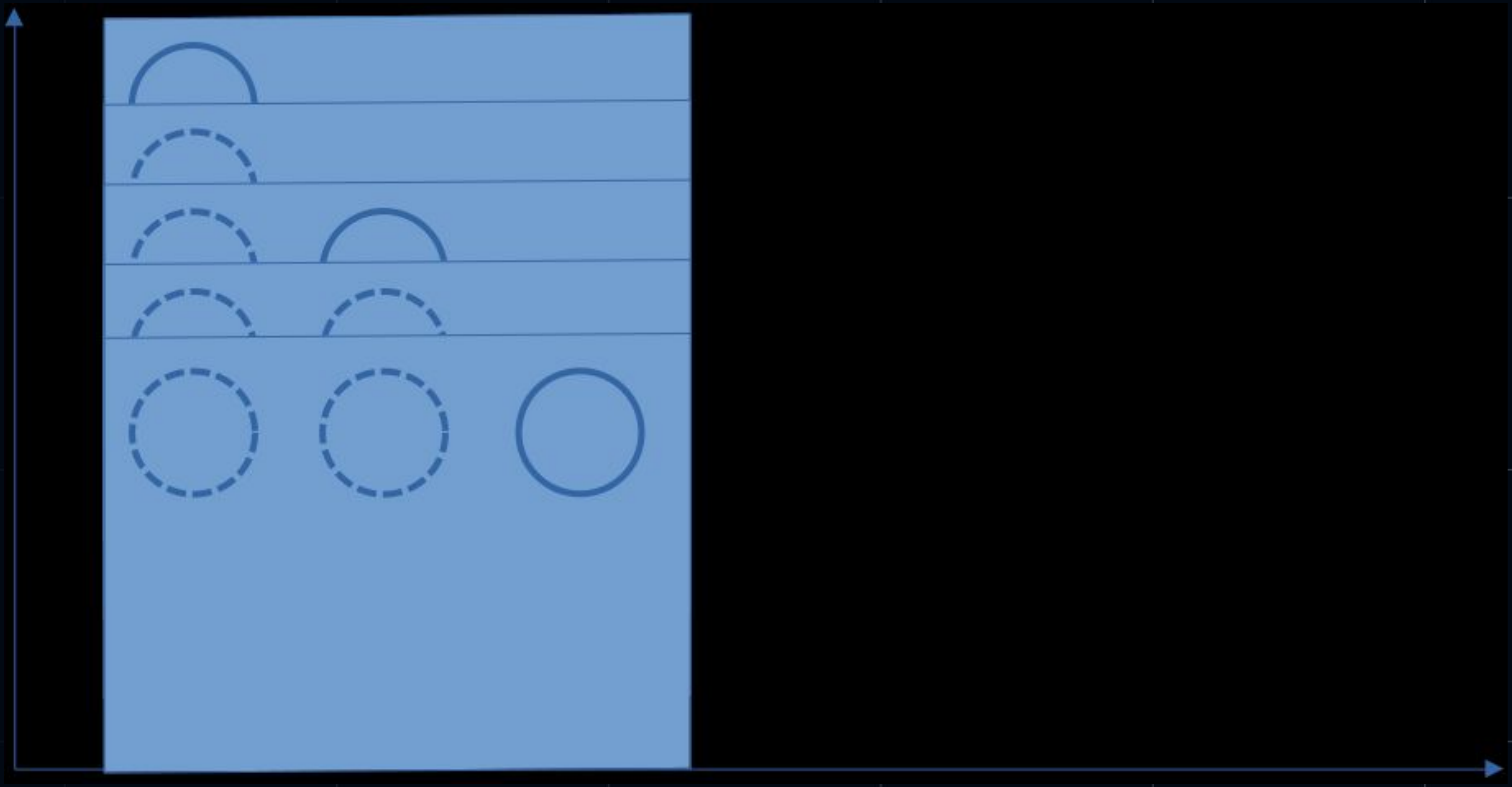
Time Matters



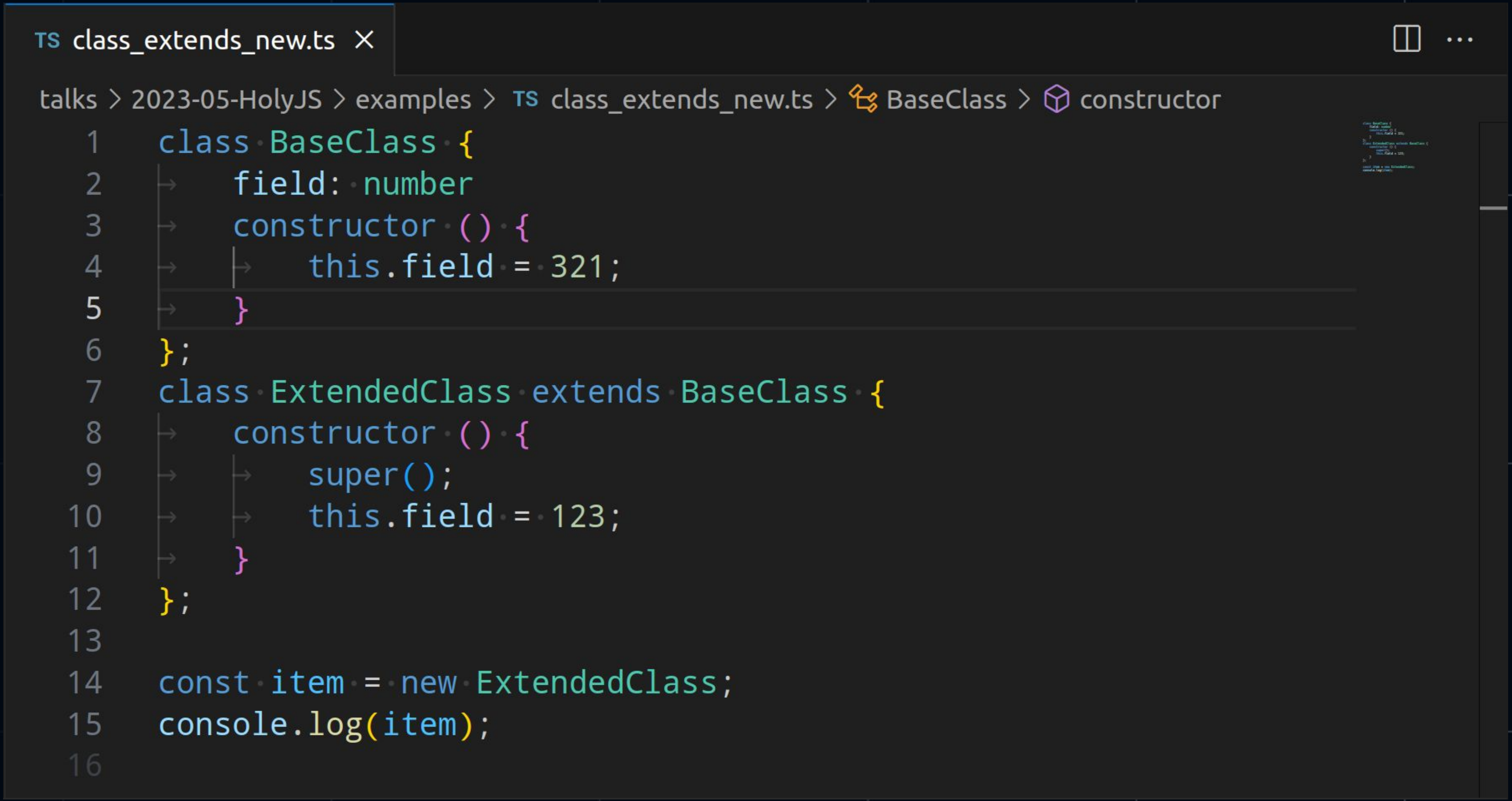
Time Matters

Time Matters

Time Matters

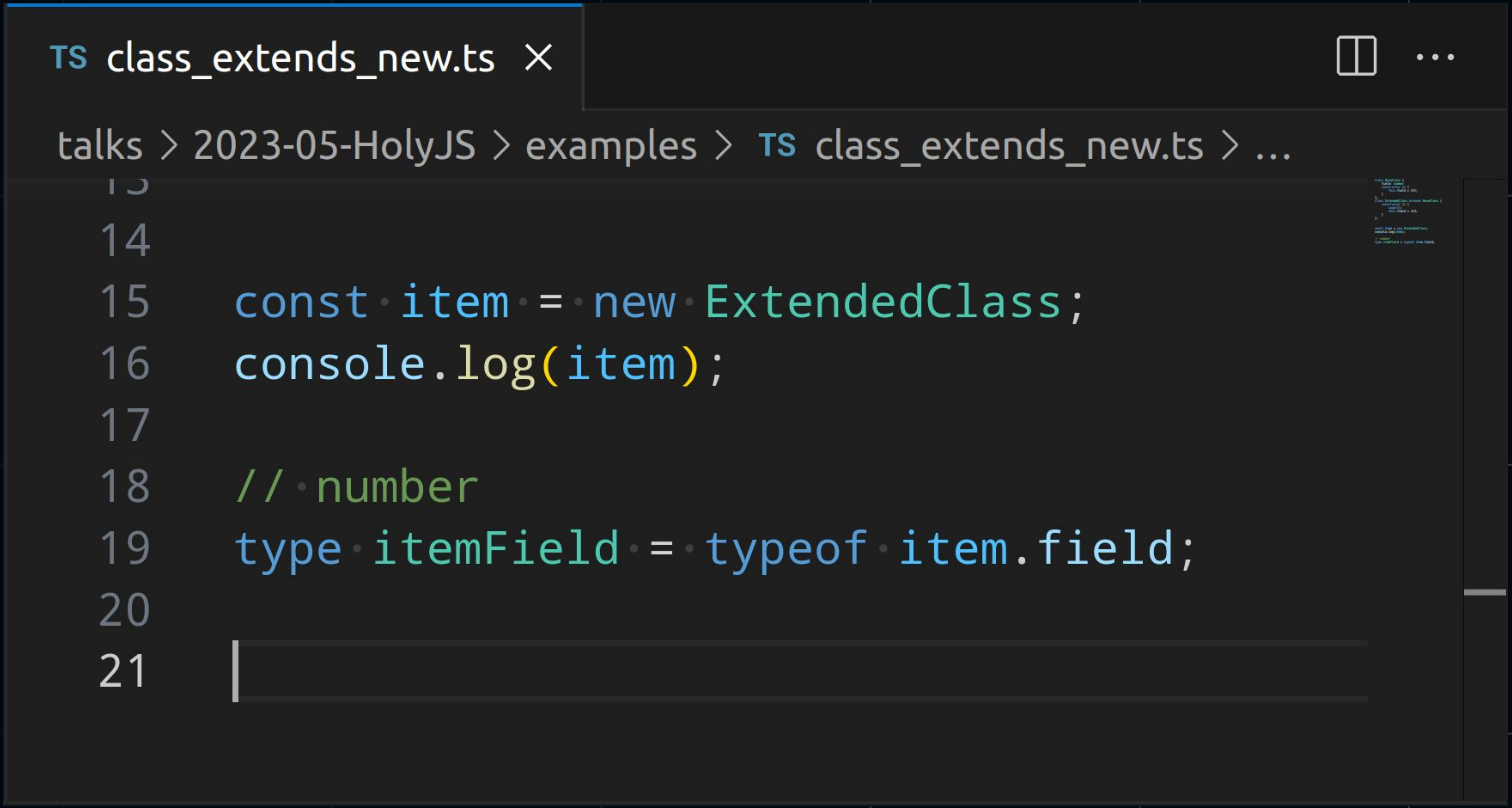


Time Matters


TS class_extends_new.ts
talks > 2023-05-HolyJS > examples > TS class_extends_new.ts > BaseClass > constructor

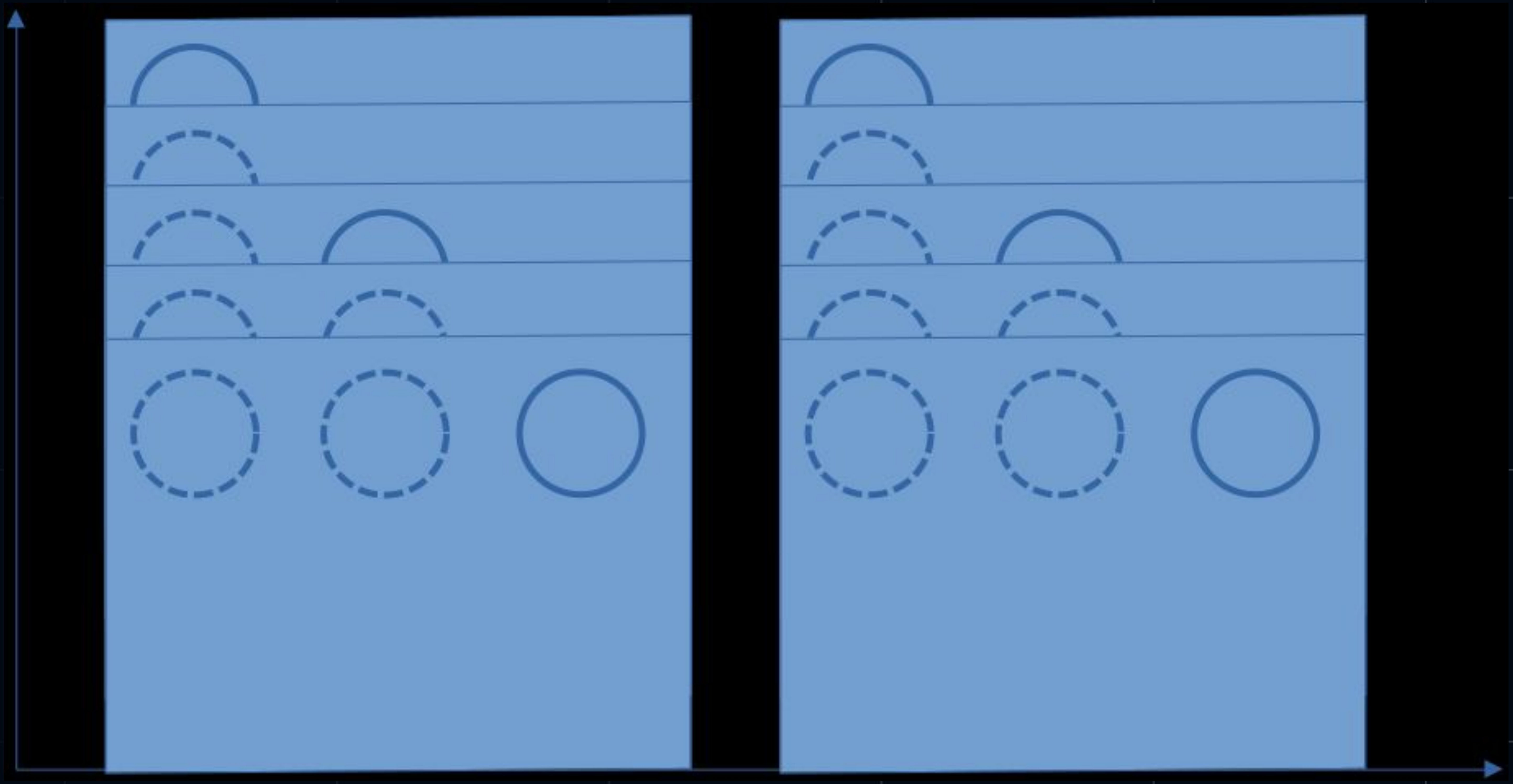
```ts
class BaseClass {
    field: number
    constructor () {
        this.field = 321;
    }
};
class ExtendedClass extends BaseClass {
    constructor () {
        super();
        this.field = 123;
    }
};

const item = new ExtendedClass;
console.log(item);
```

Time Matters

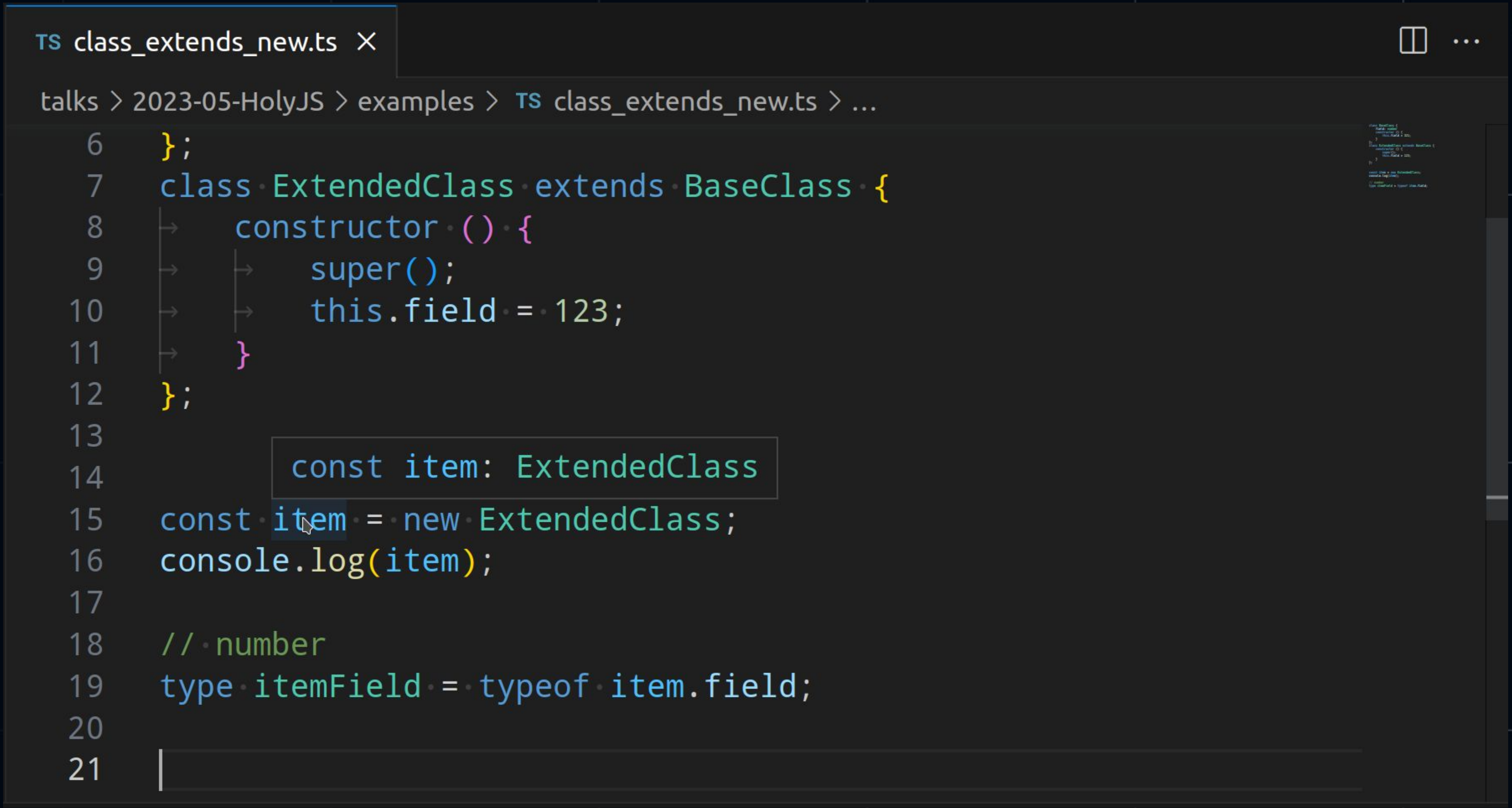

class_extends_new.ts
talks > 2023-05-HolyJS > examples > class_extends_new.ts > ...

```
const item = new ExtendedClass;
console.log(item);

// number
type itemField = typeof item.field;

```

Time Matters

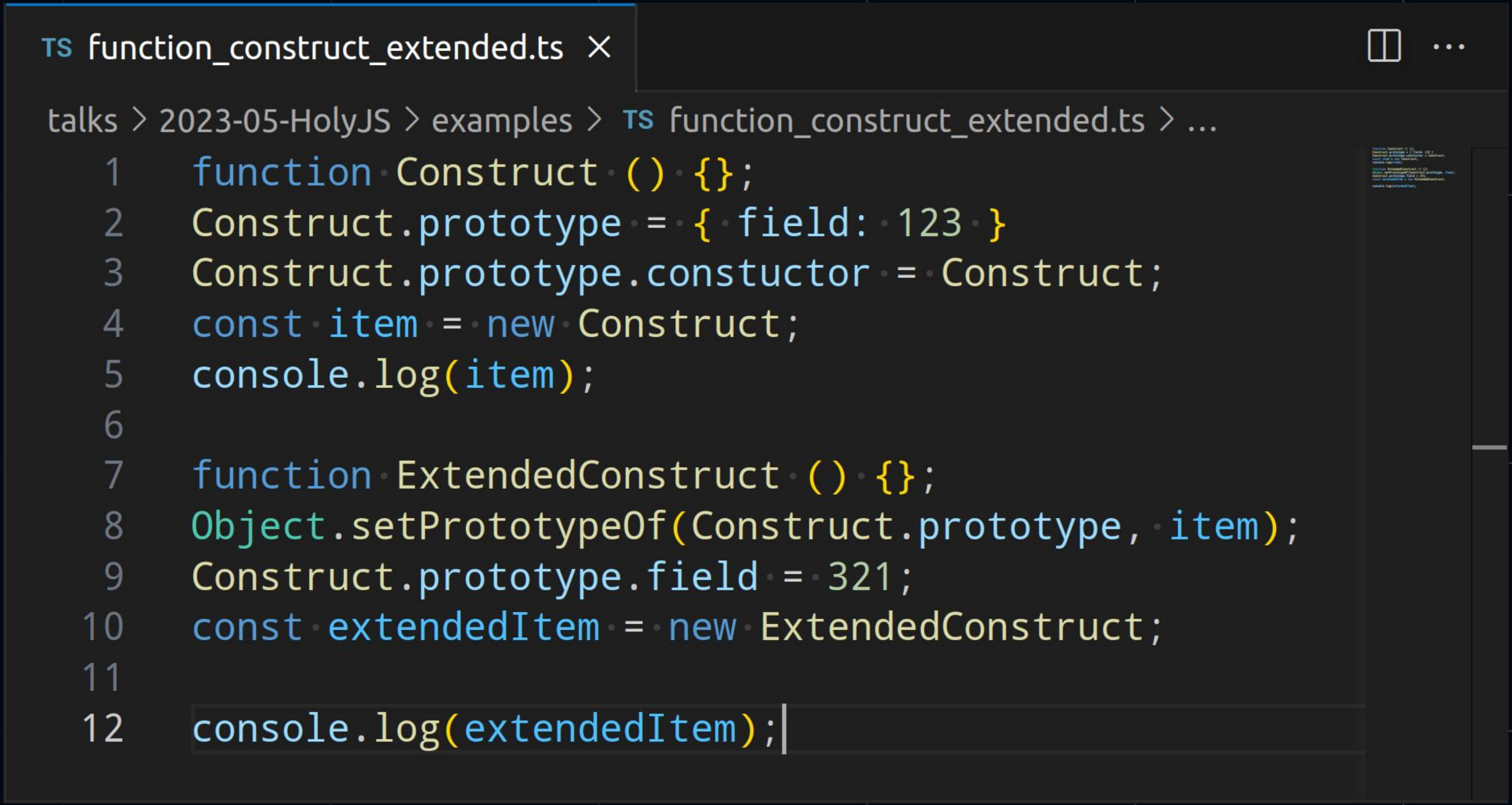


Time Matters


TS class_extends_new.ts
talks > 2023-05-HolyJS > examples > TS class_extends_new.ts > ...
const item: ExtendedClass

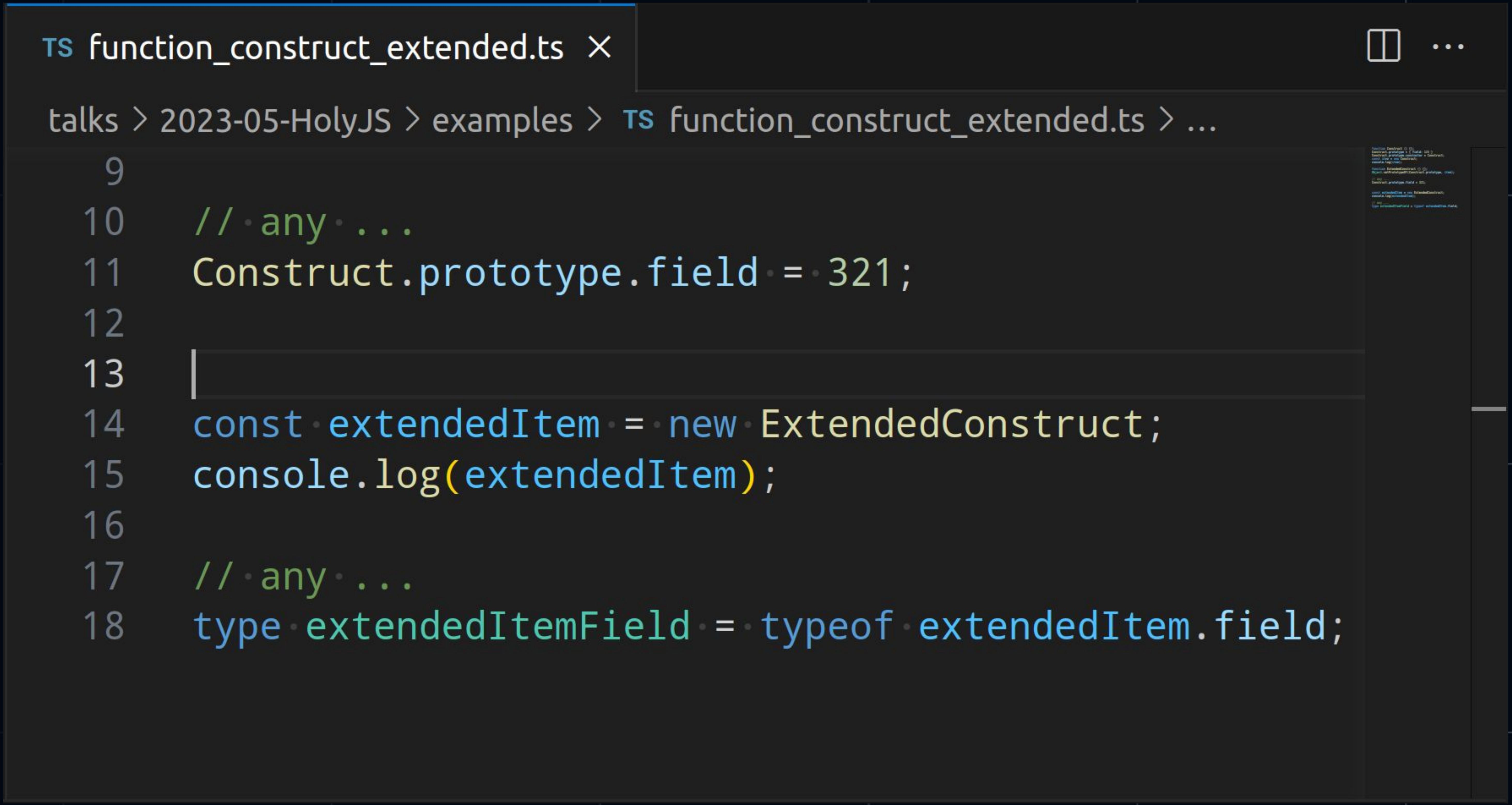
```typescript
};
class ExtendedClass extends BaseClass {
    constructor () {
        super();
        this.field = 123;
    }
};


const item = new ExtendedClass;
console.log(item);

// number
type itemField = typeof item.field;

```

Time Matters


TS function_construct_extended.ts
talks > 2023-05-HolyJS > examples > TS function_construct_extended.ts > ...

```
function Construct () {};
Construct.prototype = { field: 123 }
Construct.prototype.constuctor = Construct;
const item = new Construct;
console.log(item);

function ExtendedConstruct () {};
Object.setPrototypeOf(Construct.prototype, item);
Construct.prototype.field = 321;
const extendedItem = new ExtendedConstruct;

console.log(extendedItem);
```

Time Matters

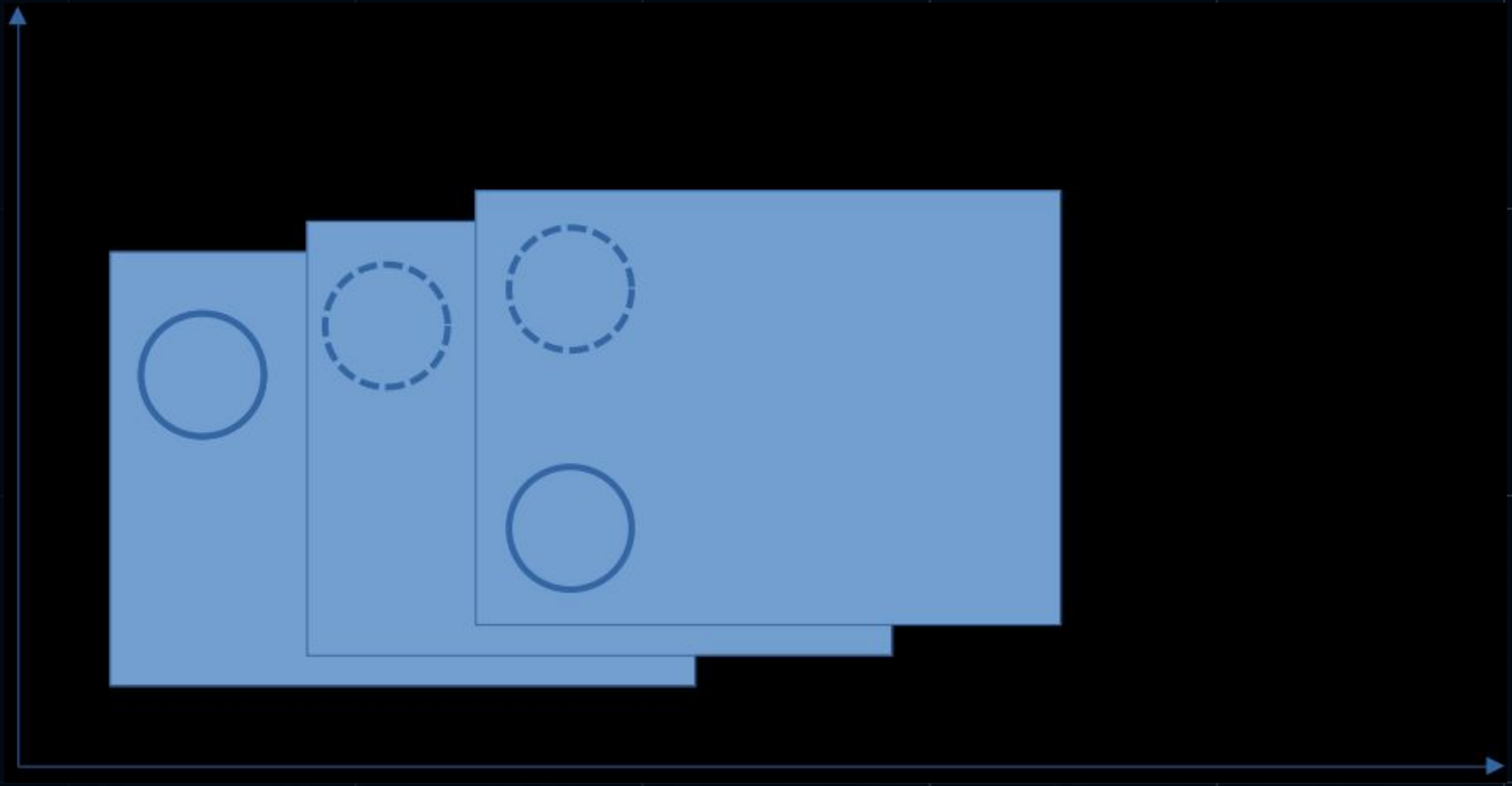


function_construct_extended.ts

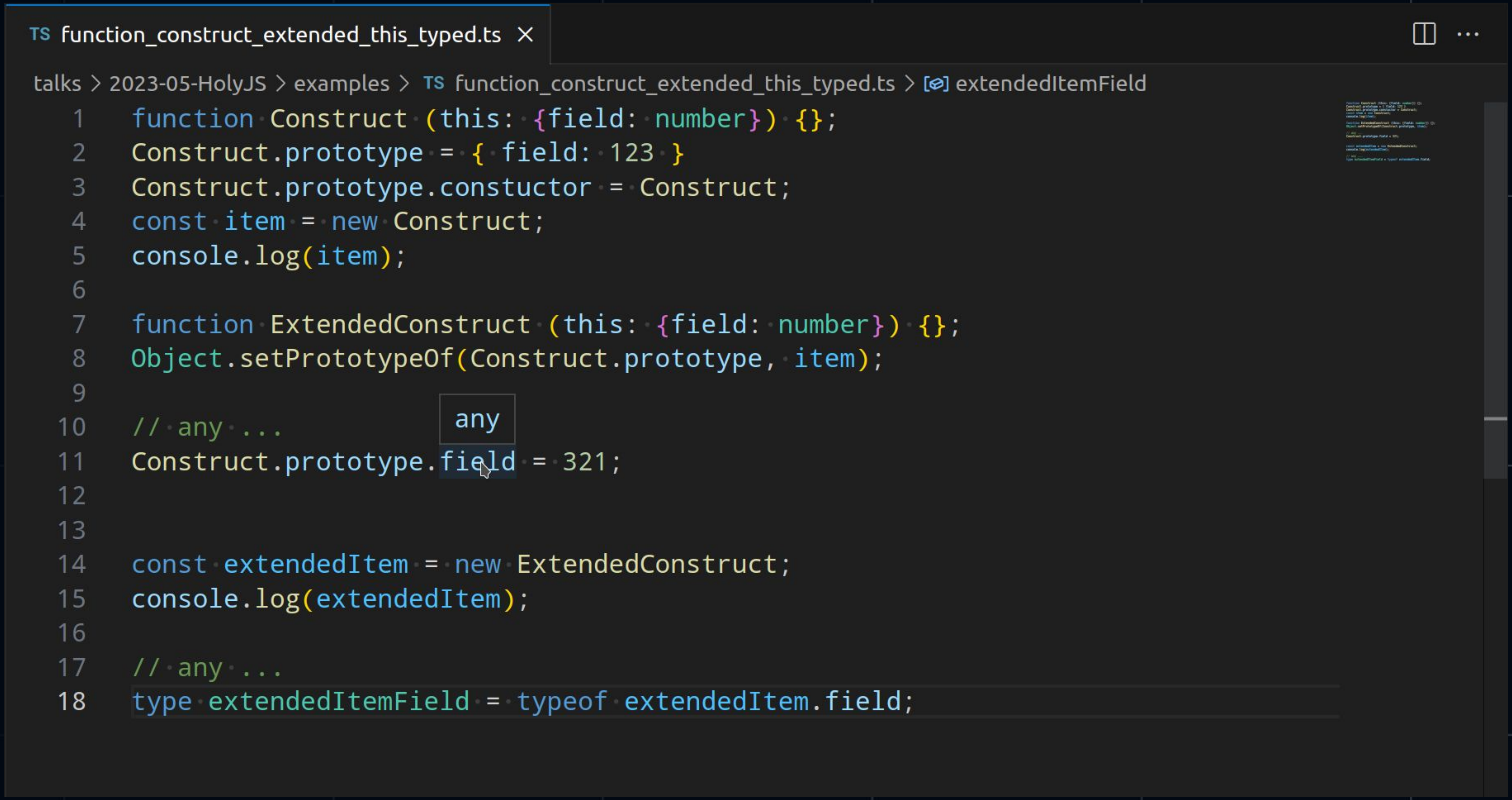
talks > 2023-05-HolyJS > examples > function_construct_extended.ts > ...

```ts

// any ...
Construct.prototype.field = 321;



const extendedItem = new ExtendedConstruct;
console.log(extendedItem);

// any ...
type extendedItemField = typeof extendedItem.field;
```

Time Matters

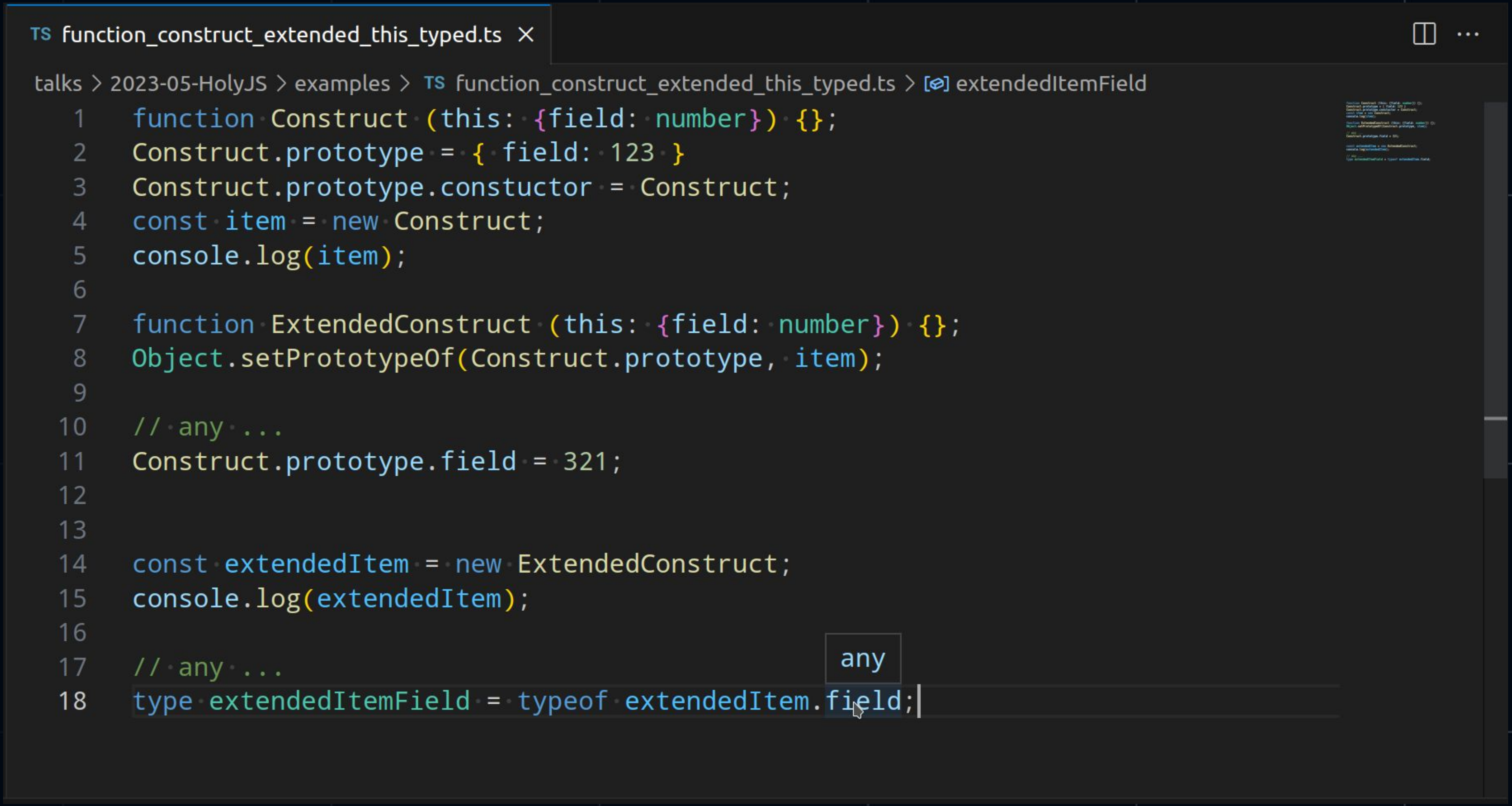


Time Matters


function_construct_extended_this_typed.ts
talks > 2023-05-HolyJS > examples > function_construct_extended_this_typed.ts > extendedItemField
any

```ts
function Construct (this: {field: number}) {};
Construct.prototype = { field: 123 }
Construct.prototype.constuctor = Construct;
const item = new Construct;
console.log(item);

function ExtendedConstruct (this: {field: number}) {};
Object.setPrototypeOf(Construct.prototype, item);

// any ...
Construct.prototype.field = 321;


const extendedItem = new ExtendedConstruct;
console.log(extendedItem);

// any ...
type extendedItemField = typeof extendedItem.field;
```

Time Matters

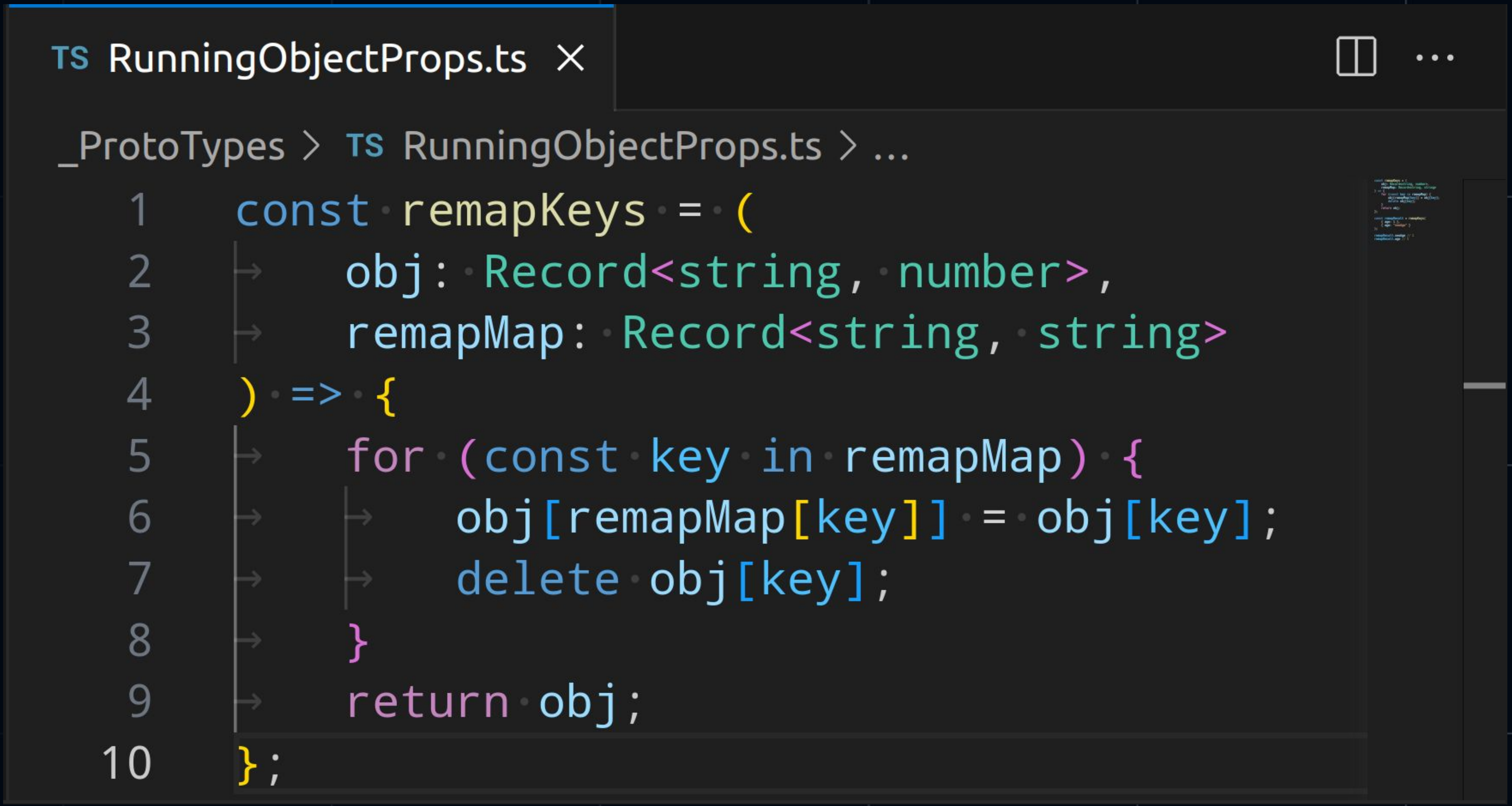


TS function_construct_extended_this_typed.ts
talks > 2023-05-HolyJS > examples > TS function_construct_extended_this_typed.ts > extendedItemField
any

```typescript
function Construct (this: {field: number}) {};
Construct.prototype = { field: 123 }
Construct.prototype.constuctor = Construct;
const item = new Construct;
console.log(item);

function ExtendedConstruct (this: {field: number}) {};
Object.setPrototypeOf(Construct.prototype, item);

// any ...
Construct.prototype.field = 321;


const extendedItem = new ExtendedConstruct;
console.log(extendedItem);

// any ...
type extendedItemField = typeof extendedItem.field;
```

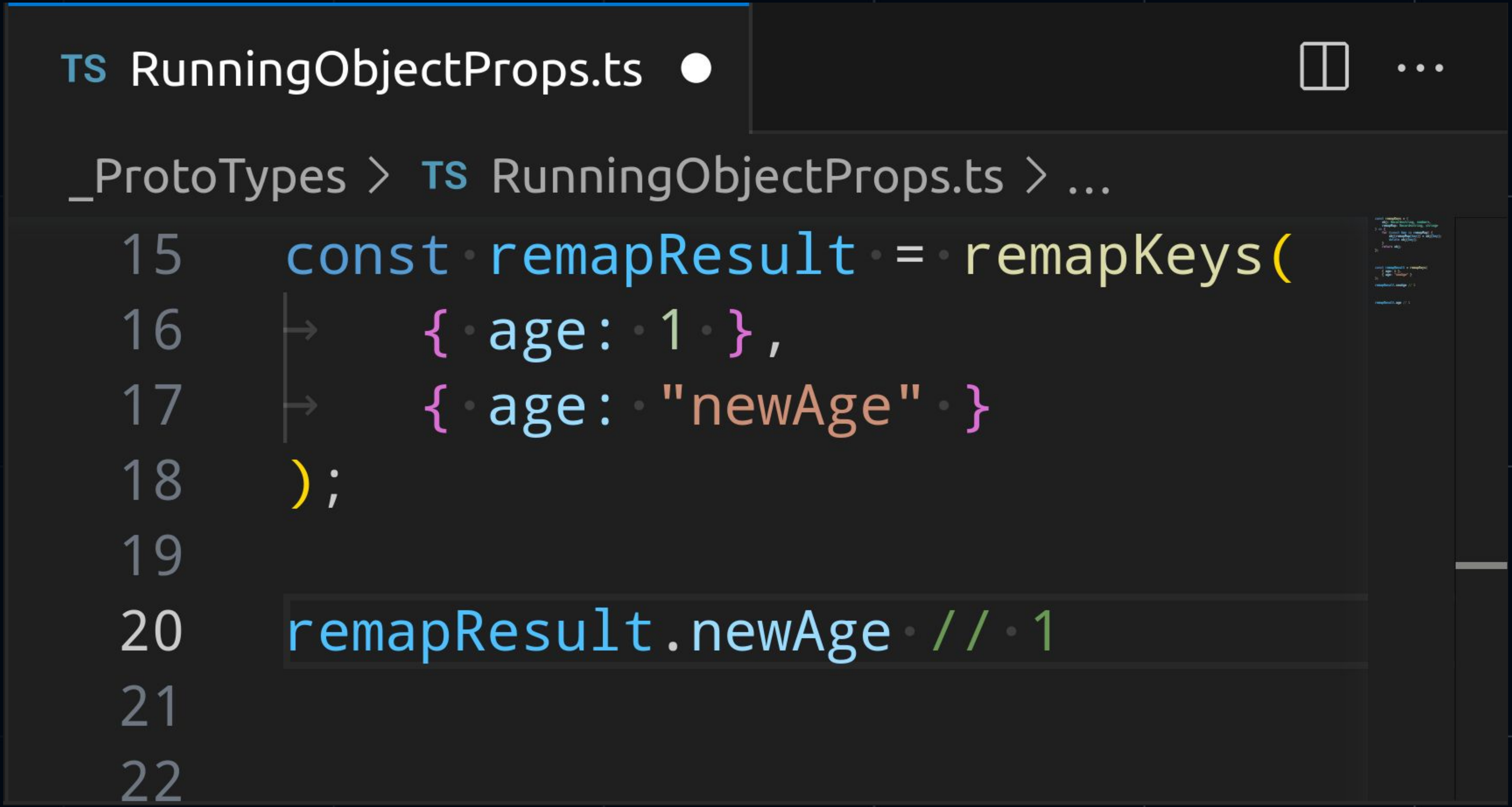
types of Inheritance in JS~TS

types of Inheritance



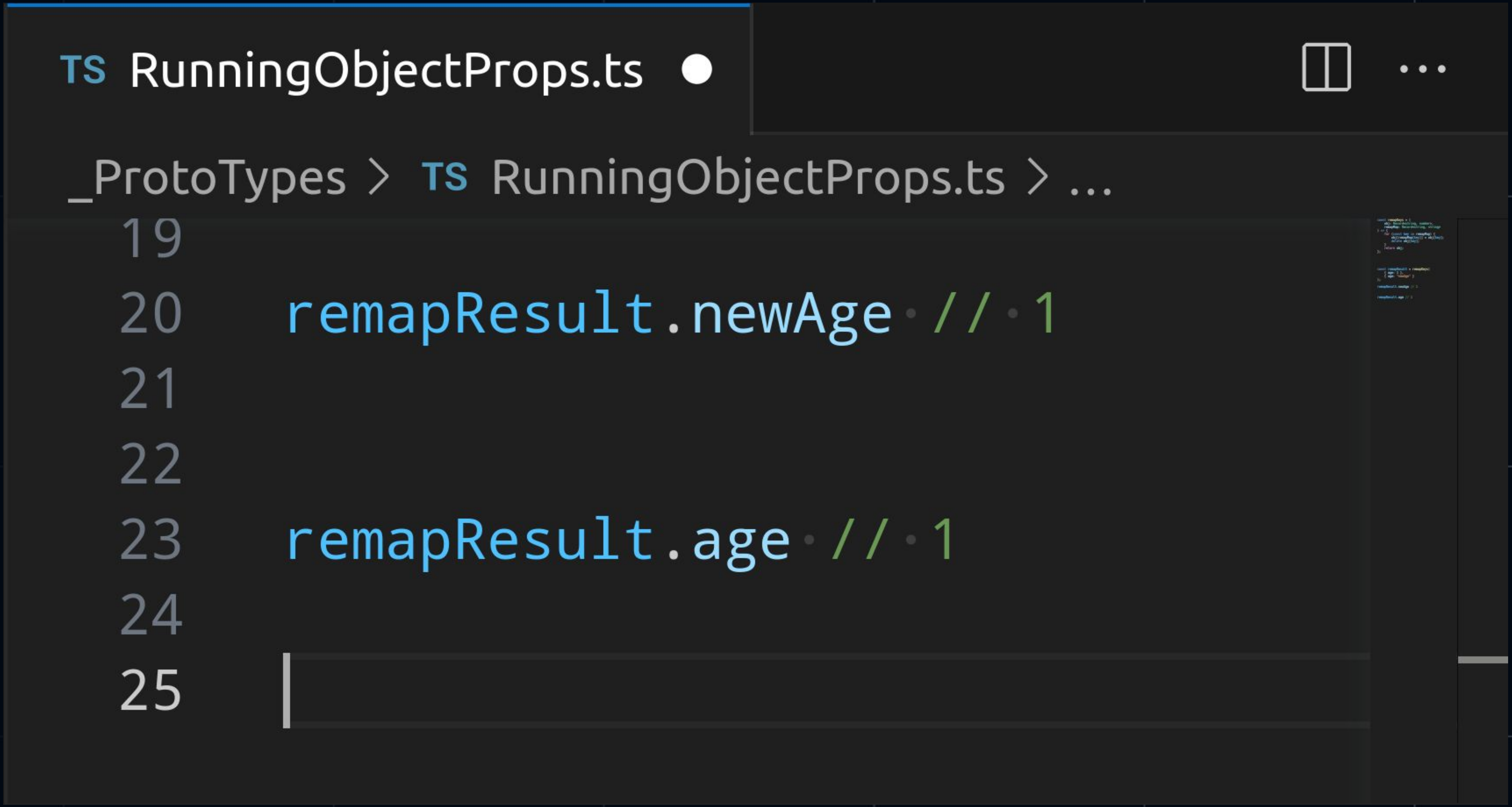
```typescript
const remapKeys = (
    obj: Record<string, number>,
    remapMap: Record<string, string>
) => {
    for (const key in remapMap) {
        obj[remapMap[key]] = obj[key];
        delete obj[key];
    }
    return obj;
};
```

types of Inheritance



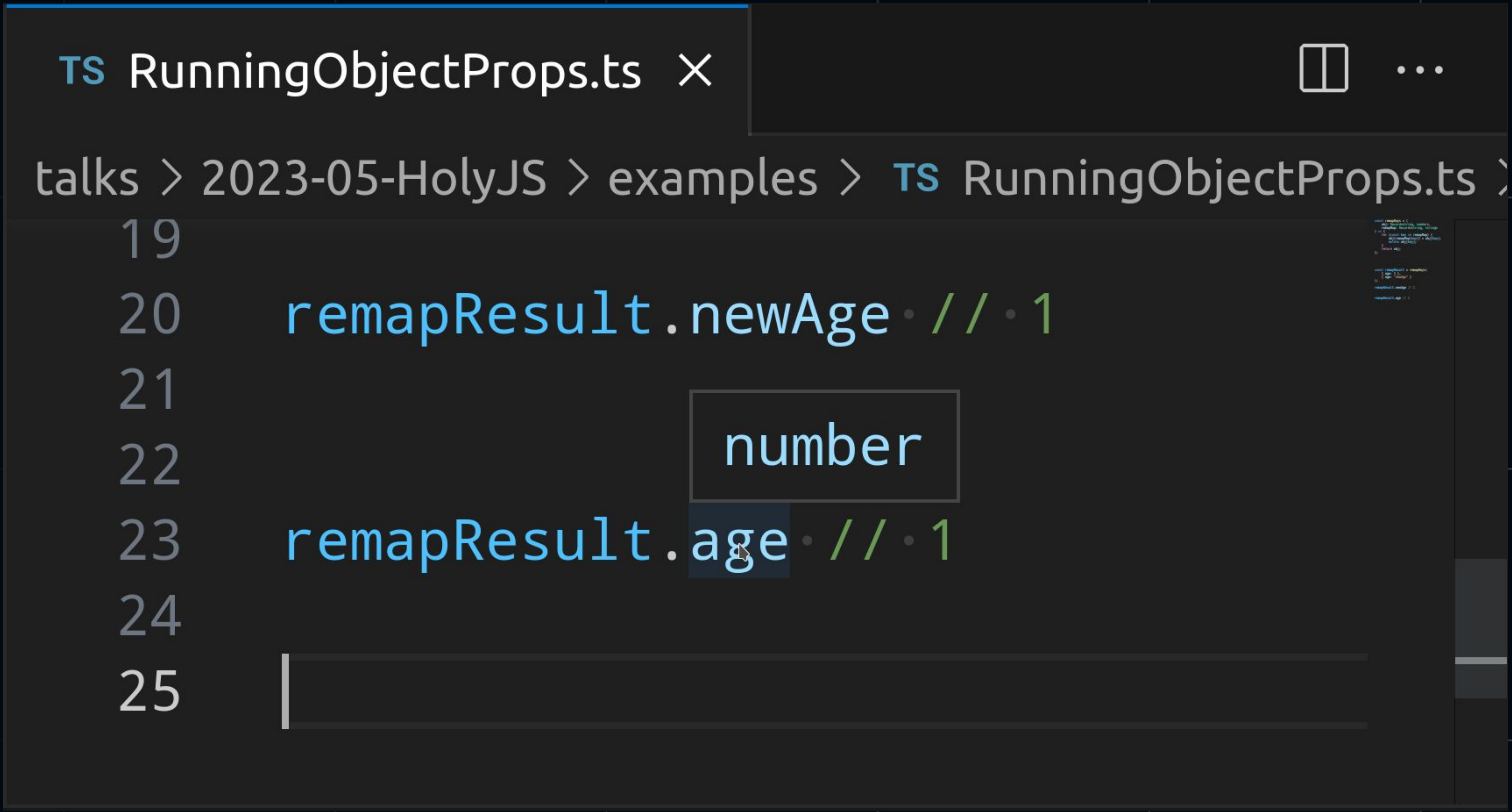
```ts
const remapResult = remapKeys(
    { age: 1 },
    { age: "newAge" }
);

remapResult.newAge // 1
```

types of Inheritance



```
remapResult.newAge // 1


remapResult.age // 1

```

types of Inheritance



TS RunningObjectProps.ts
talks > 2023-05-HolyJS > examples > TS RunningObjectProps.ts
number

```
remapResult.newAge // 1


remapResult.age // 1

```

types of Inheritance

it is not the thing you think about …

- Primitive to Primitive
- Primitive to Object
- Object to Primitive
- Object to Object

types of Inheritance

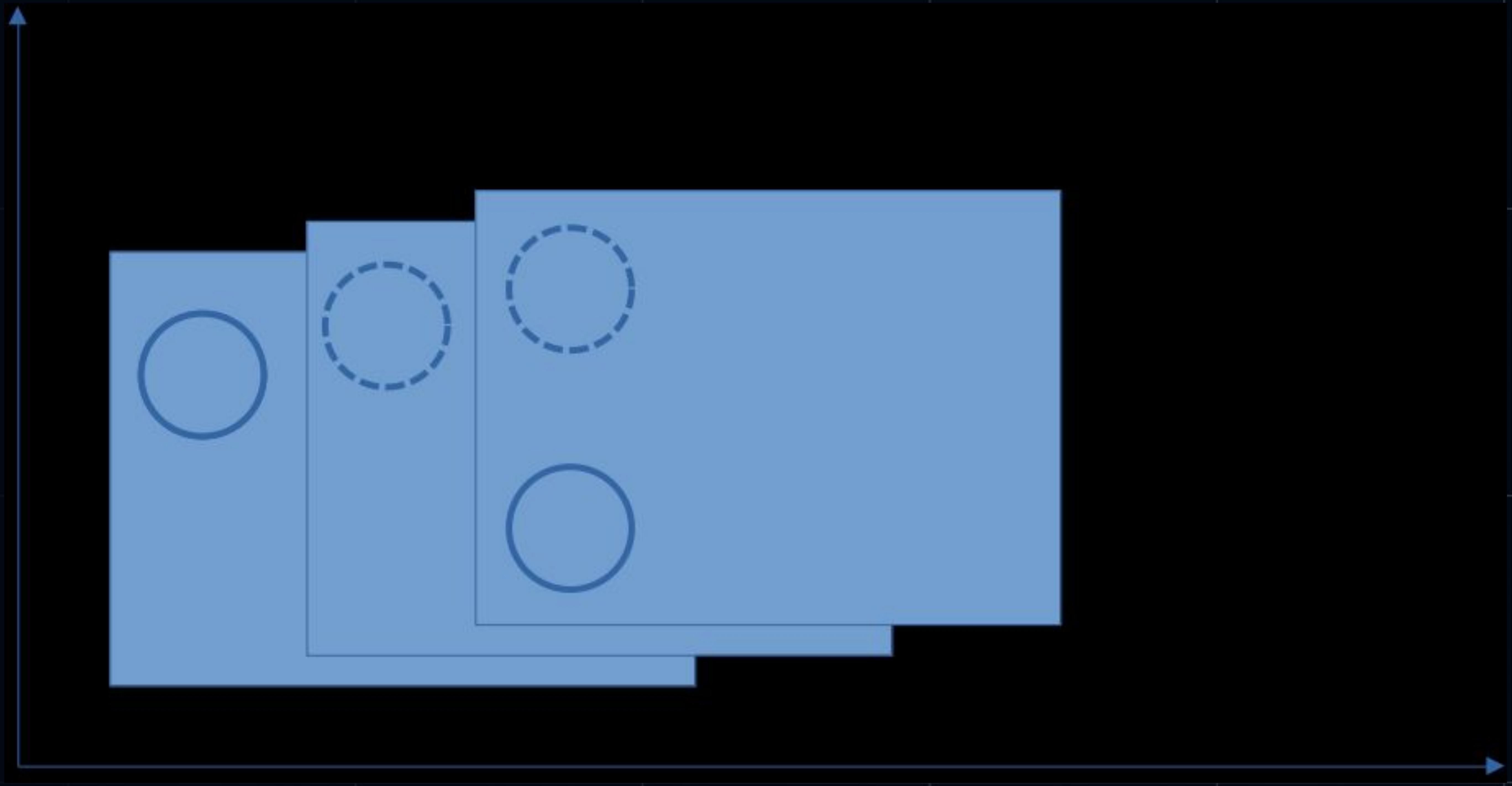
it is not the thing you think about …

- Primitive to Primitive
- Primitive to Object
- Object to Primitive
- Object to Object

DEMO

Optional Fields Definitions

optional fields …



optional fields …

again is not the usual thing …

- get ~ set only fields
- and this might be deep …
- and mixed with Primitive | Object

optional fields …

again is not the usual thing …

- get ~ set only fields
- and this might be deep …
- and mixed with Primitive | Object

DEMO

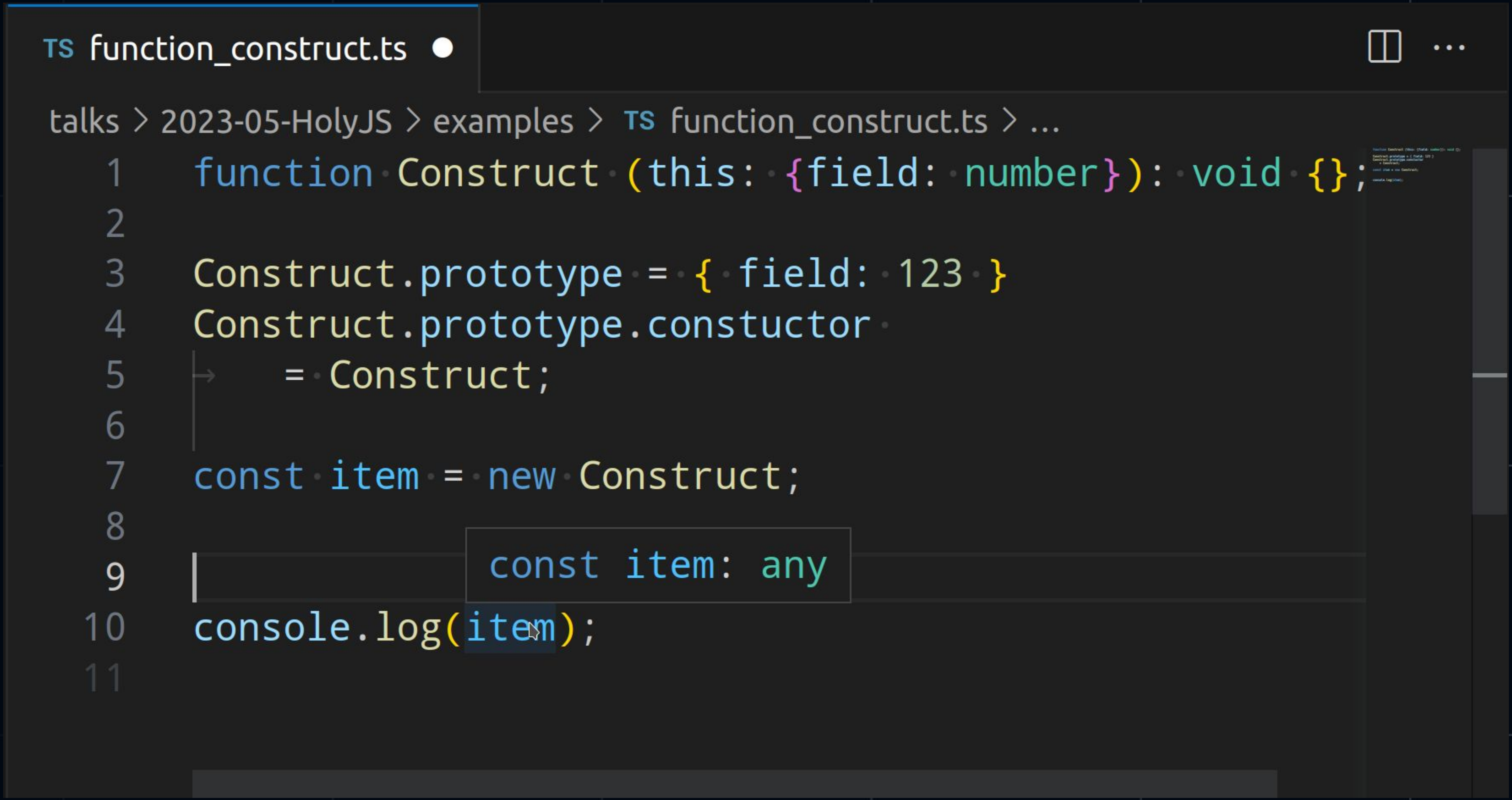
Identity of Chaining

identity of chaining

to define constructible something we need

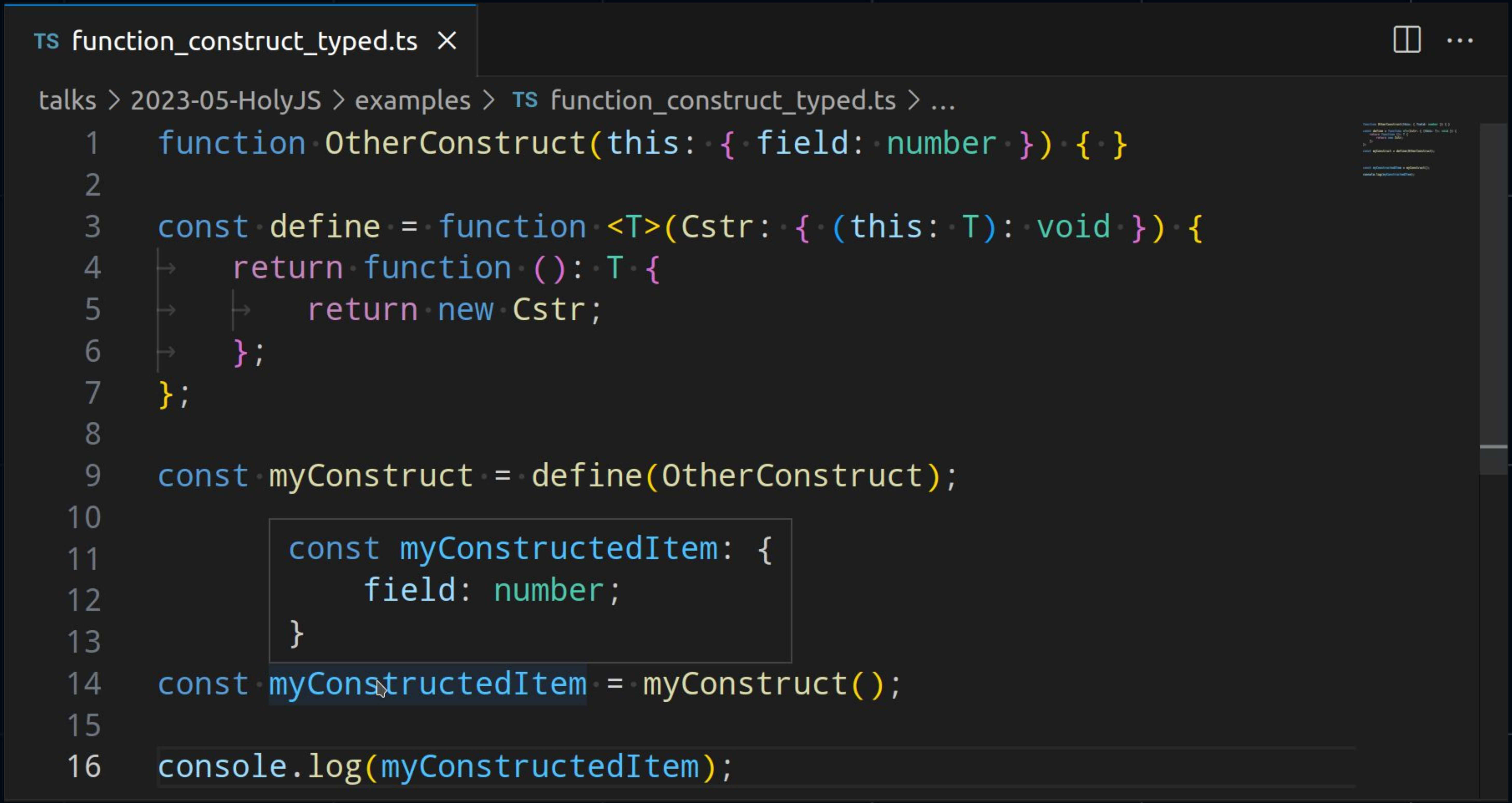
1. to get existing instance type
2. be familiar with future type
3. mix existent and future types

identity of chaining


TS function_construct.ts
talks > 2023-05-HolyJS > examples > TS function_construct.ts > ...

```ts
function Construct (this: {field: number}): void {};

Construct.prototype = { field: 123 }
Construct.prototype.constuctor
    = Construct;

const item = new Construct;

                 const item: any
console.log(item);
```

identity of chaining



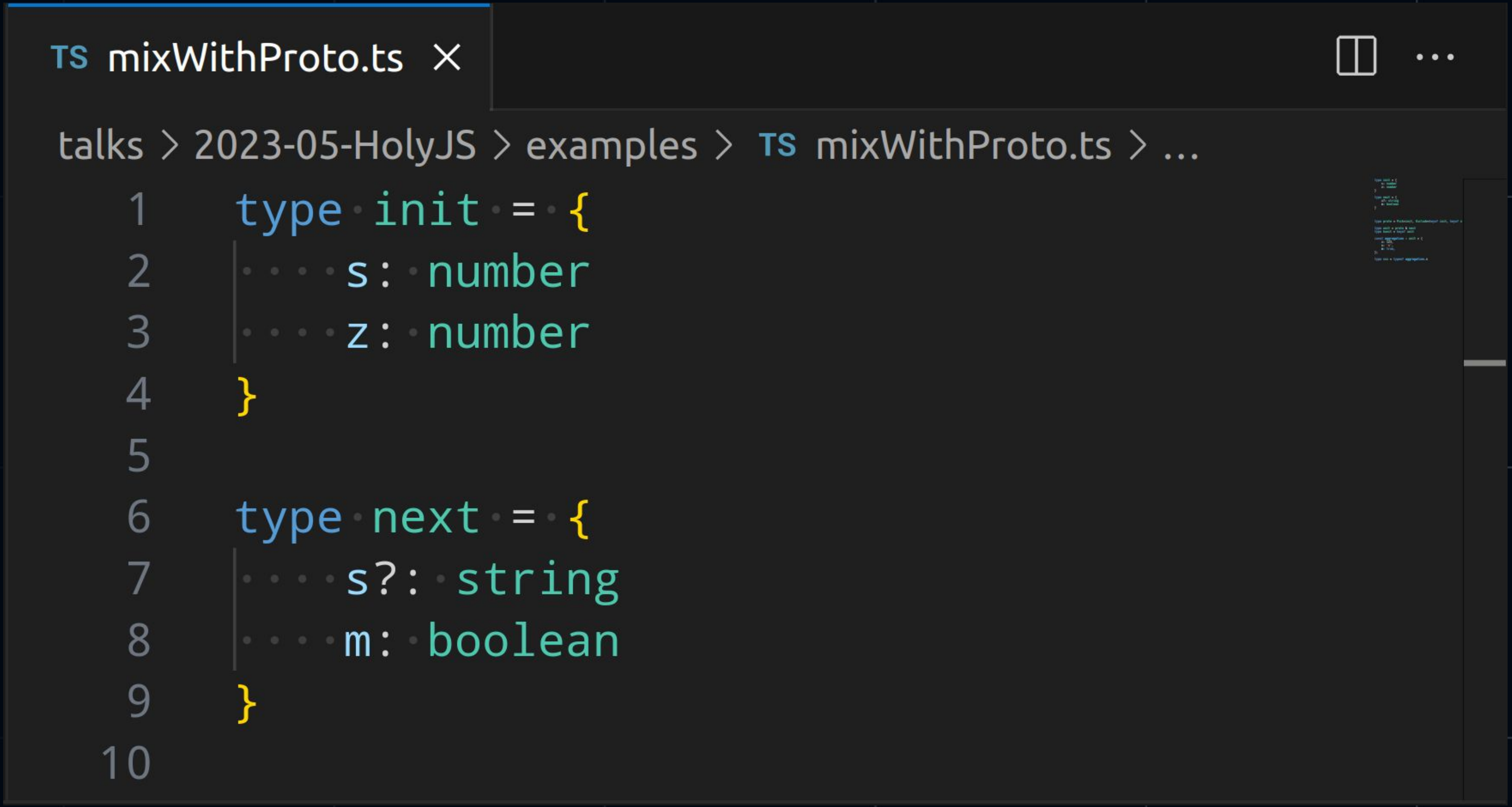
TS function_construct_typed.ts ×

talks > 2023-05-HolyJS > examples > TS function_construct_typed.ts > ...

```ts
function OtherConstruct(this: { field: number }) { }

const define = function <T>(Cstr: { (this: T): void }) {
    return function (): T {
        return new Cstr;
    };
};

const myConstruct = define(OtherConstruct);

const myConstructedItem = myConstruct();

console.log(myConstructedItem);
```

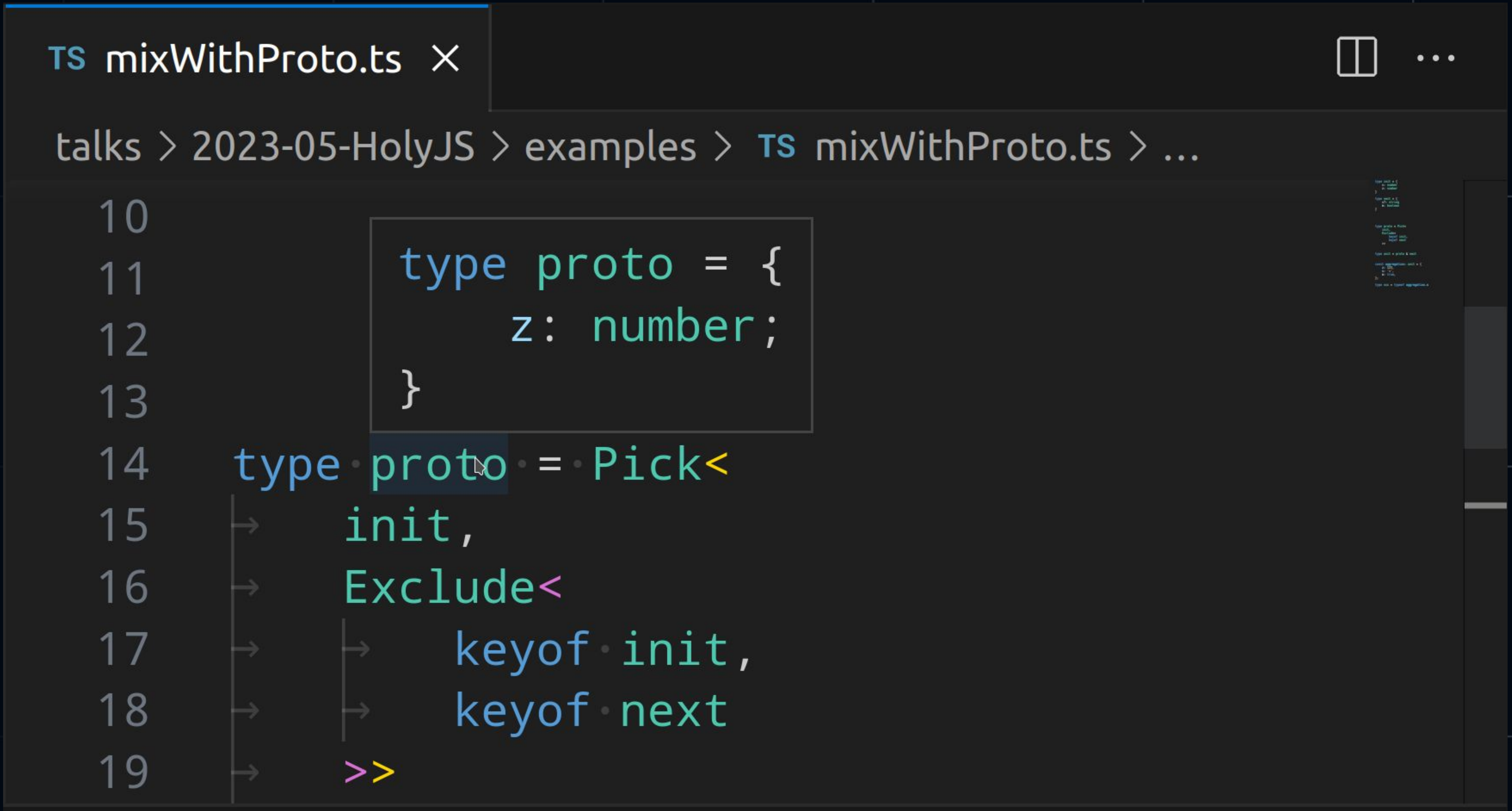
```ts
const myConstructedItem: {
    field: number;
}
```

identity of chaining


TS mixWithProto.ts
talks > 2023-05-HolyJS > examples > TS mixWithProto.ts > ...

```ts
type init = {
    s: number
    z: number
}

type next = {
    s?: string
    m: boolean
}

```

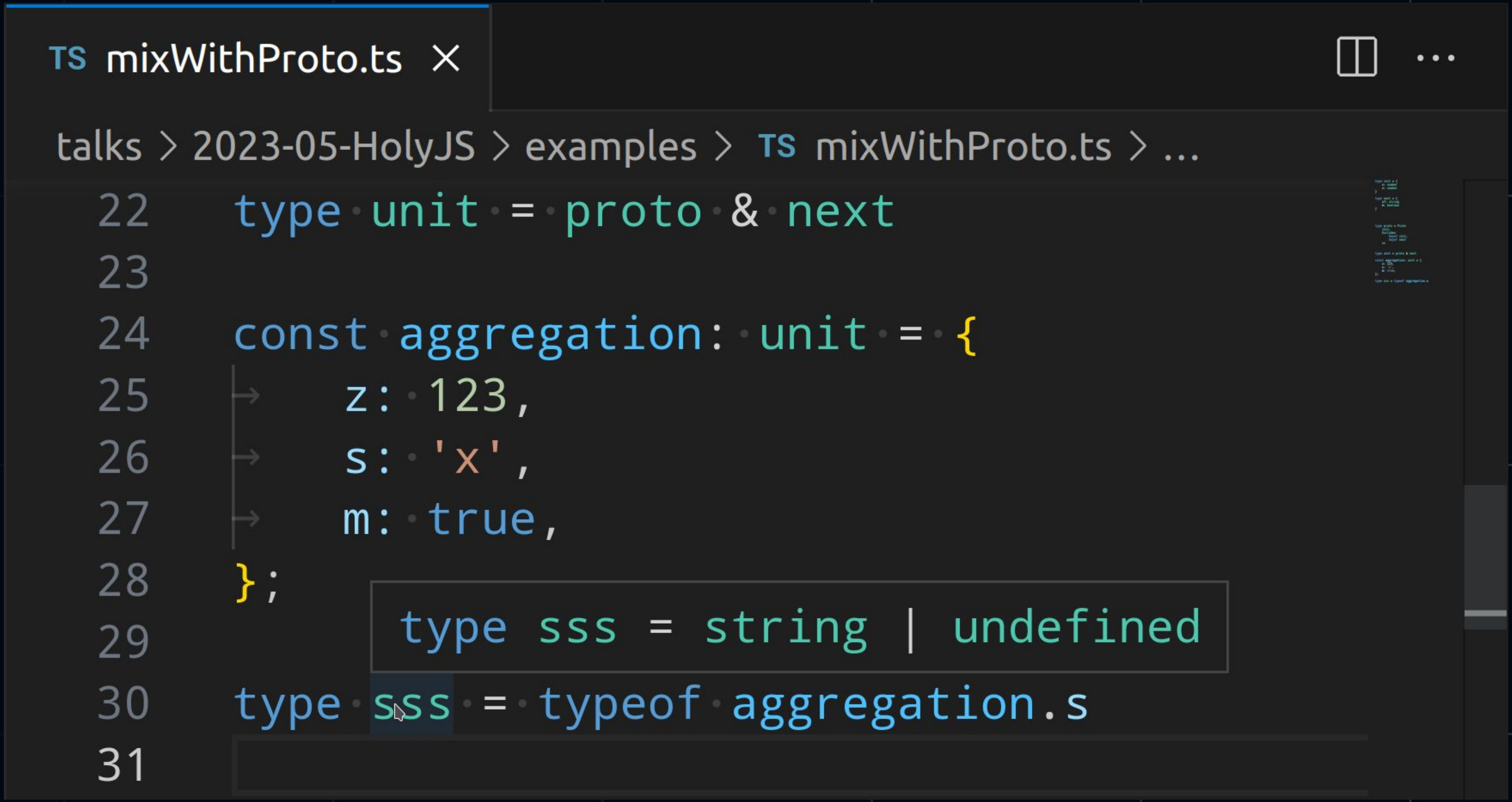
identity of chaining


TS mixWithProto.ts
talks > 2023-05-HolyJS > examples > TS mixWithProto.ts > ...

```
type proto = {
    z: number;
}
```

```
type proto = Pick<
    init,
    Exclude<
        keyof init,
        keyof next
    >>
```

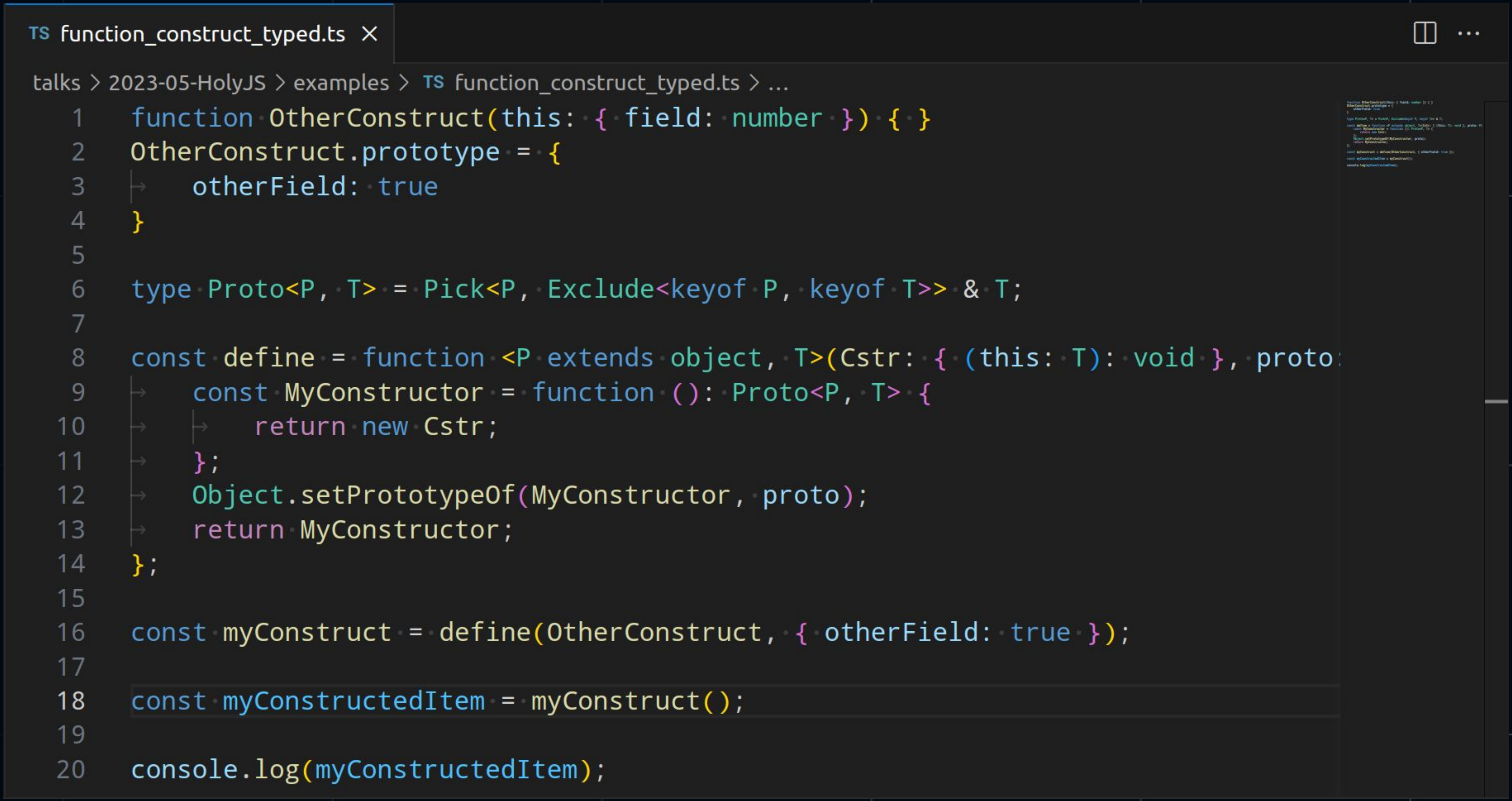
identity of chaining


TS mixWithProto.ts
talks > 2023-05-HolyJS > examples > TS mixWithProto.ts > ...

```ts
type unit = proto & next

const aggregation: unit = {
    z: 123,
    s: 'x',
    m: true,
};

type sss = typeof aggregation.s

```

```ts
type sss = string | undefined
```

identity of chaining


TS function_construct_typed.ts
talks > 2023-05-HolyJS > examples > TS function_construct_typed.ts > ...

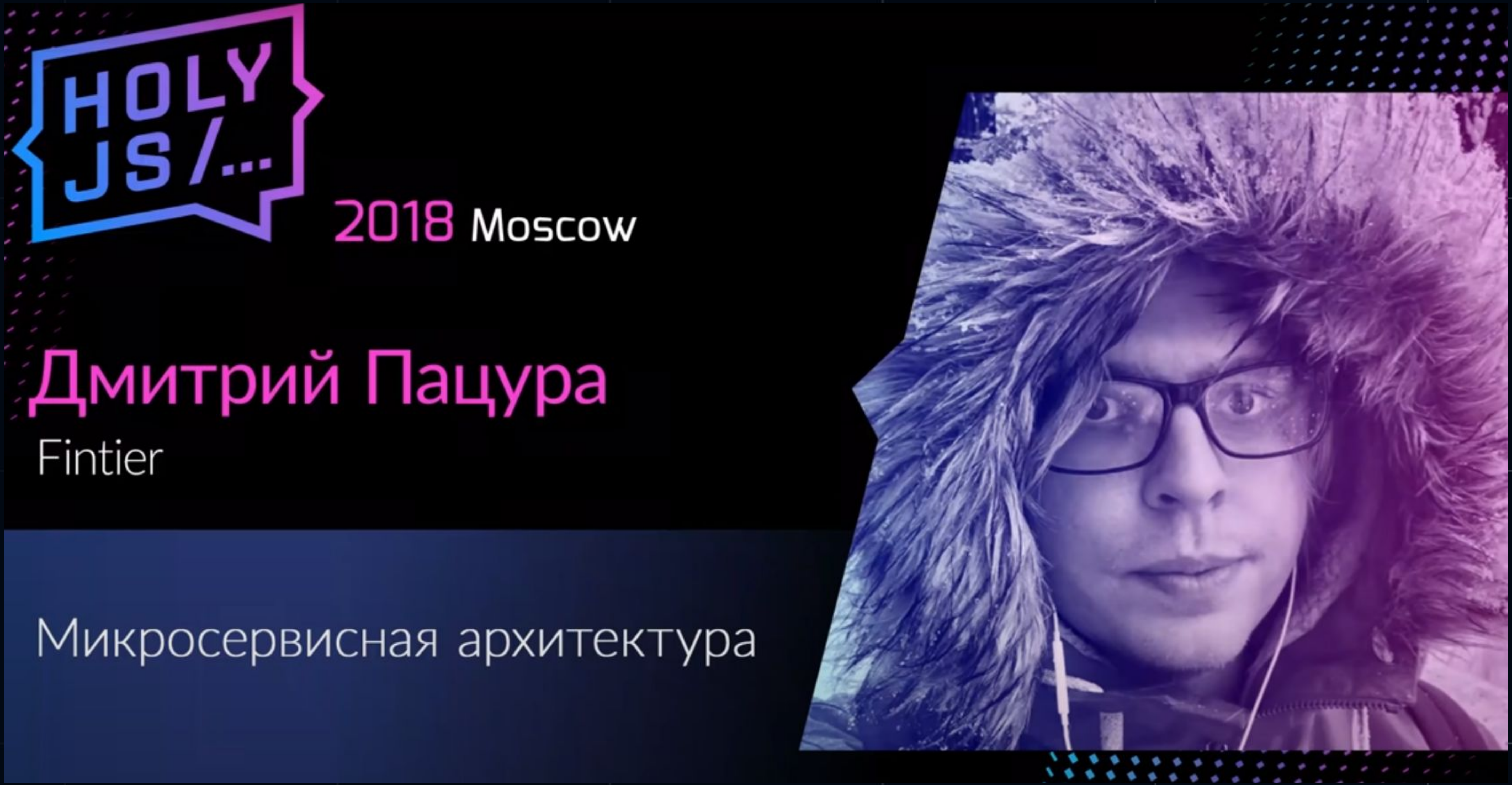
```ts
function OtherConstruct(this: { field: number }) { }
OtherConstruct.prototype = {
    otherField: true
}

type Proto<P, T> = Pick<P, Exclude<keyof P, keyof T>> & T;

const define = function <P extends object, T>(Cstr: { (this: T): void }, proto
    const MyConstructor = function (): Proto<P, T> {
        return new Cstr;
    };
    Object.setPrototypeOf(MyConstructor, proto);
    return MyConstructor;
};

const myConstruct = define(OtherConstruct, { otherField: true });

const myConstructedItem = myConstruct();

console.log(myConstructedItem);
```

identity of chaining

DEMO

ø

special thanks to



HOLY JS /...

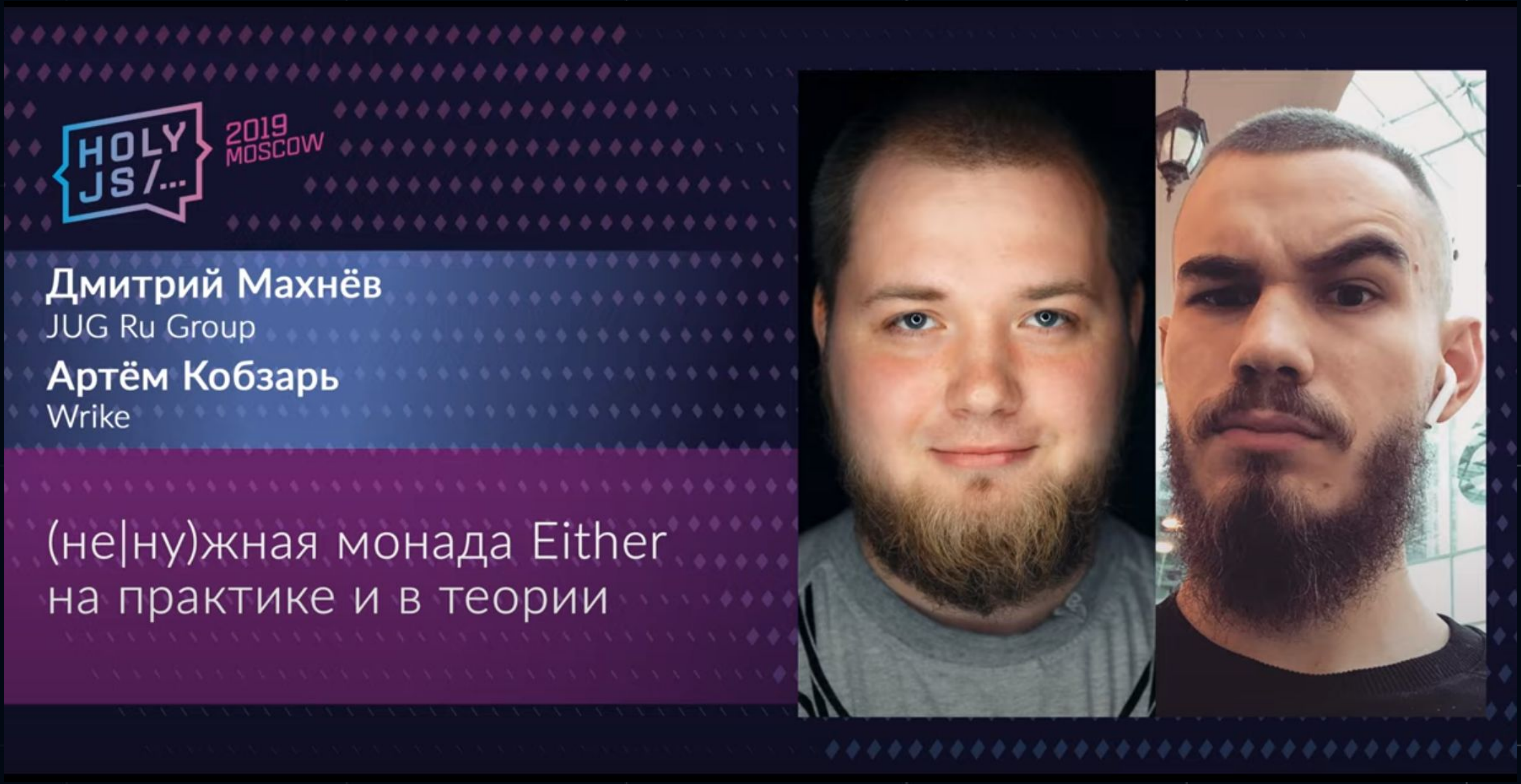
2018 Moscow

Дмитрий Пацура

Fintier

Микросервисная архитектура

special thanks to



HOLY JS /... 2019 MOSCOW

Дмитрий Махнёв
JUG Ru Group
Артём Кобзарь
Wrike

(не|ну)жная монада Either
на практике и в теории

thank you

to be continued ...

next talk announce

Mnemonica Project

Viktor
Vershanskiy

wentout