

# Types in Prototypes

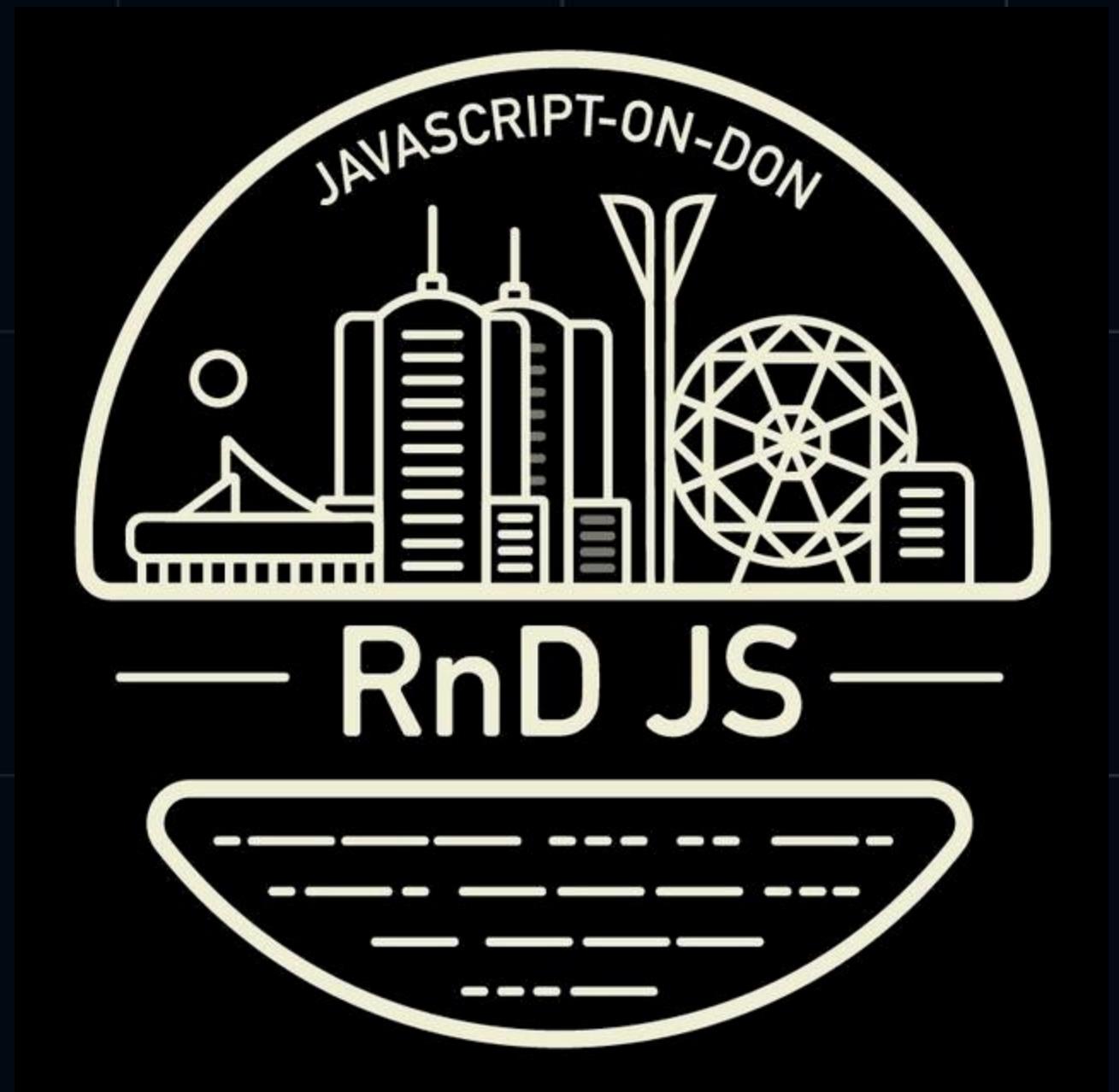


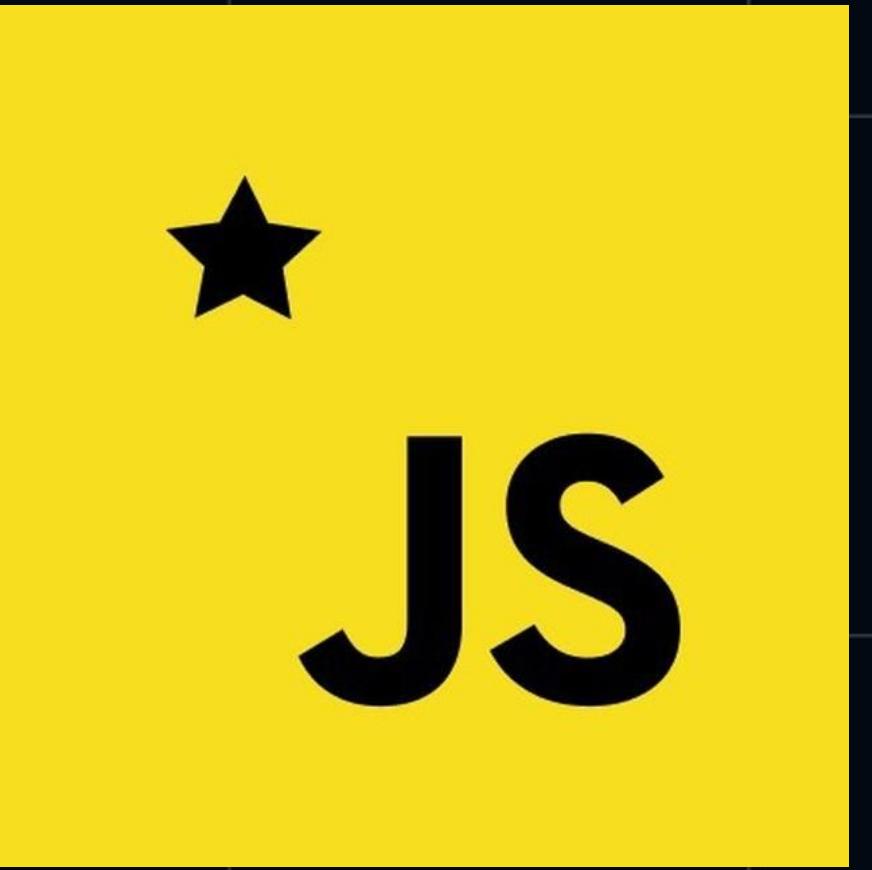
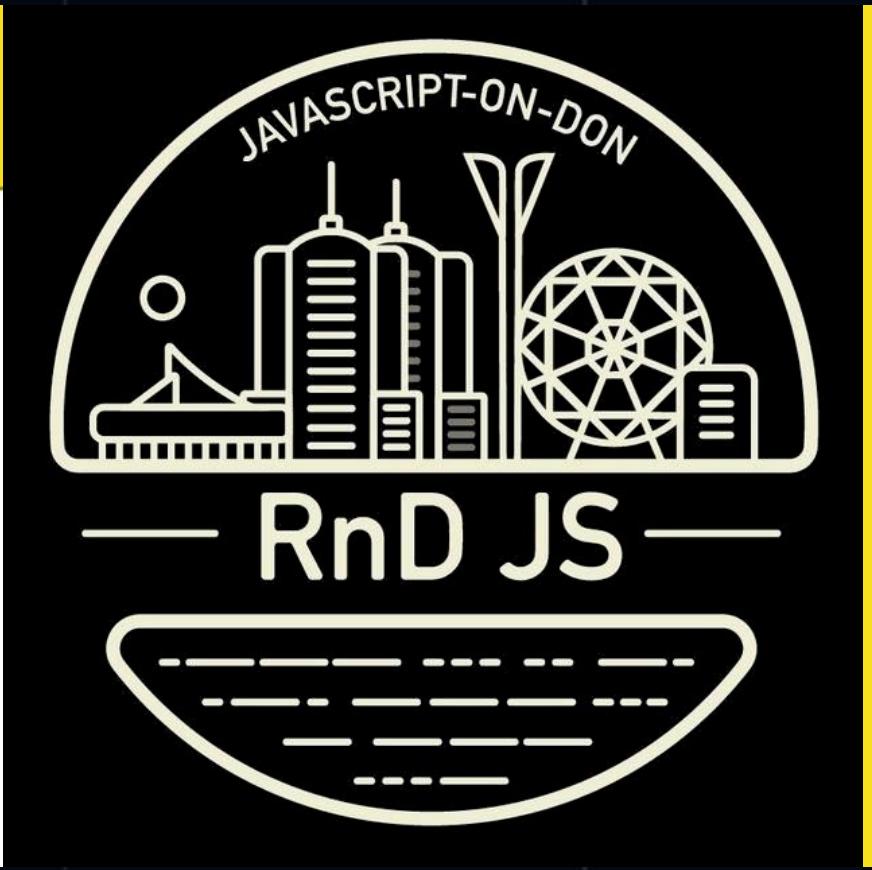
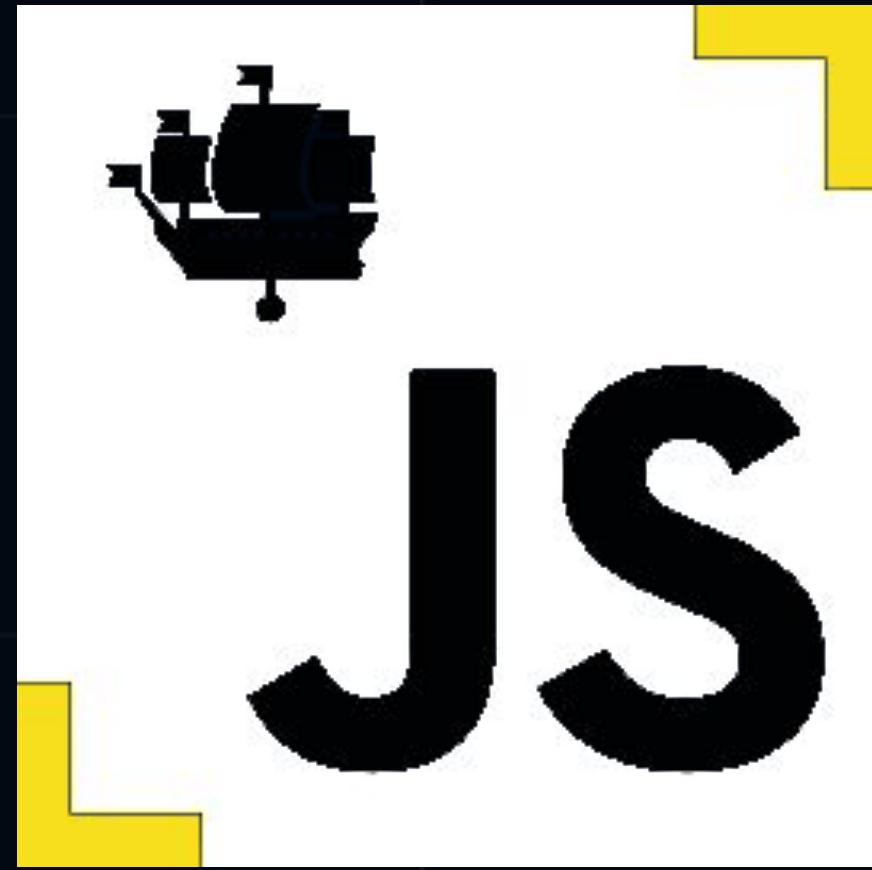
Viktor  
Vershanskiy

↗ wentout

HOLY  
JS!...









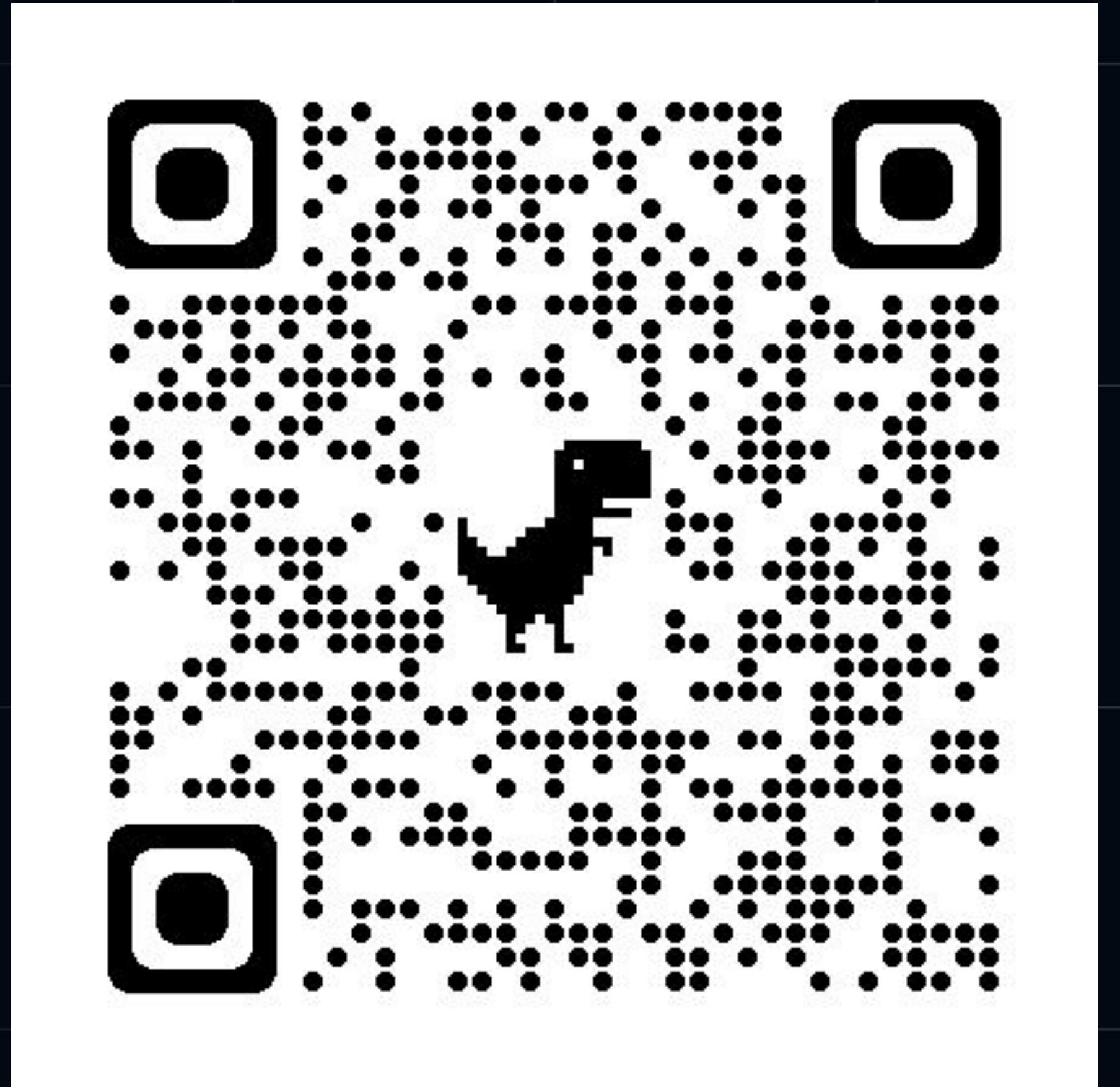
Виктор



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

The video player interface displays a presentation slide. In the top left corner, there is a logo for "HOLY JS" with a speech bubble icon, followed by the text "2019 Piter". Below this, the speaker's name is listed as "Дмитрий Харитонов" and their affiliation as "Isovalent". The main title of the presentation is "Подход к типобезопасной разработке на TypeScript". The video progress bar at the bottom shows a timestamp of "0:18 / 53:55". To the right of the video player, there is a portrait of a man with short brown hair and a slight beard, looking directly at the camera. The background of the slide features a dark grid pattern.

HOLY JS  
2019 Piter

Дмитрий Харитонов  
Isovalent

Подход к типобезопасной разработке на TypeScript

0:18 / 53:55



# Known TypeScript on HolyJS



Рефлексия  
в TypeScript



Александр Богачёв  
Arrival Ltd



# Known TypeScript on HolyJS



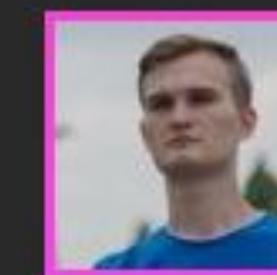
## Advanced types in TypeScript

### 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

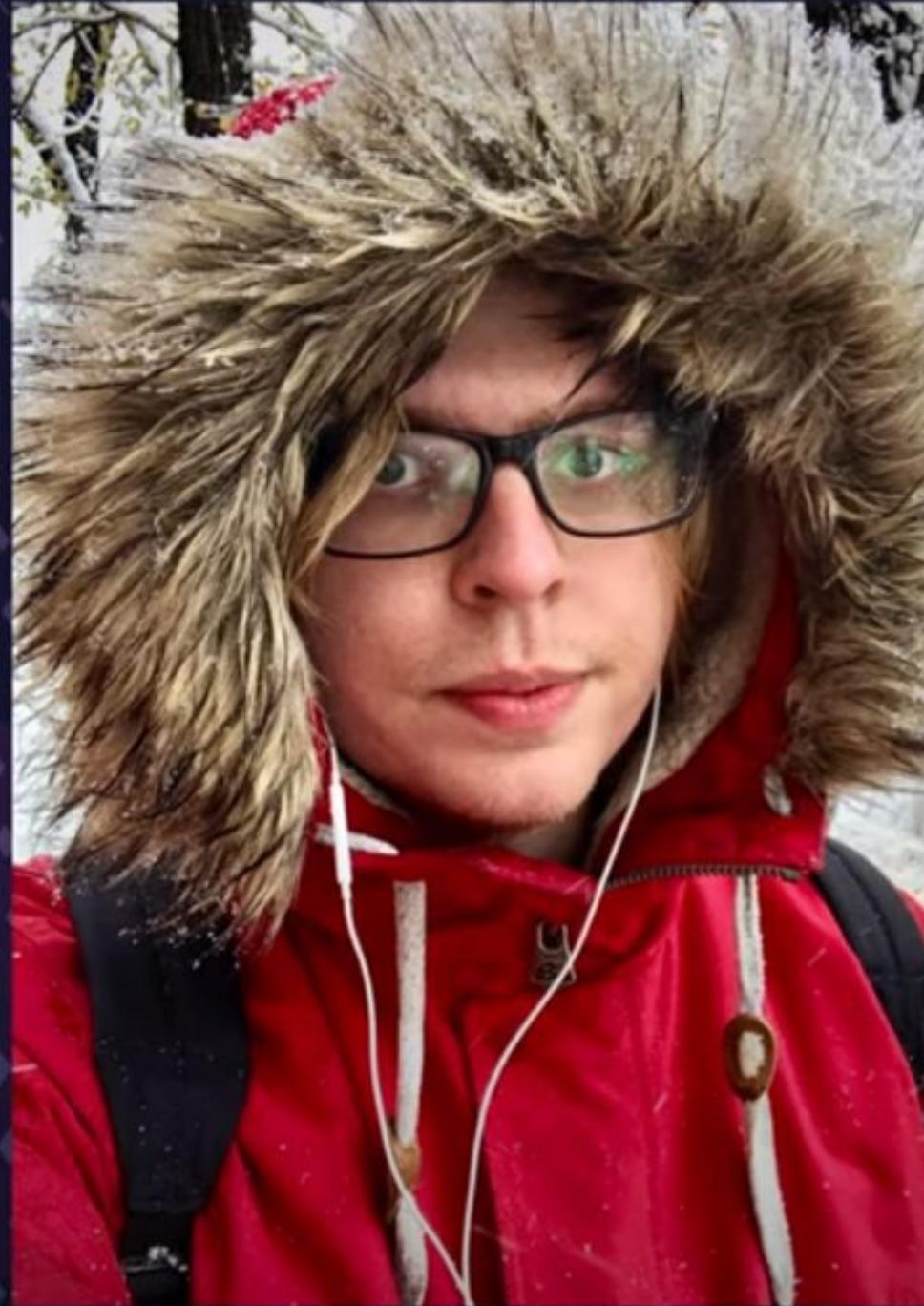
# Known TypeScript on HolyJS



2019  
MOSCOW

Дмитрий Пацура  
LOWL

Разработка компилятора для  
TypeScript на TypeScript на базе LLVM

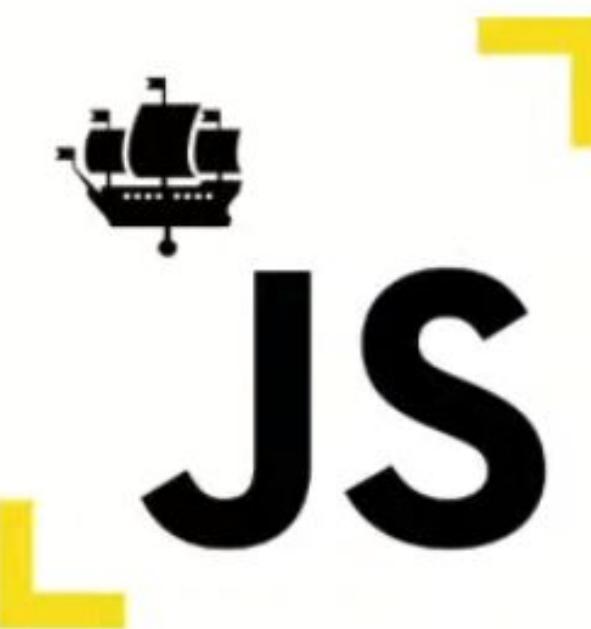




Moonshine  
Spiritual



# Moonshine Spiritual talks



**PiterJS #54**

**NodeJS SPb**



Онлайн митап  
10 декабря 19:00 - 20:30

Pro .prototype'ы



# Moonshine Spiritual talks



СОБЫТИЯ ВИДЕО ДОКЛАДЧИКИ ПОДАТЬ ДОКЛАД



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



## Strict Types in JavaScript

# Moonshine Spiritual talks



Multiple inheritance in JavaScript



# Class Based

vs

# Functional Based



# Class vs Function



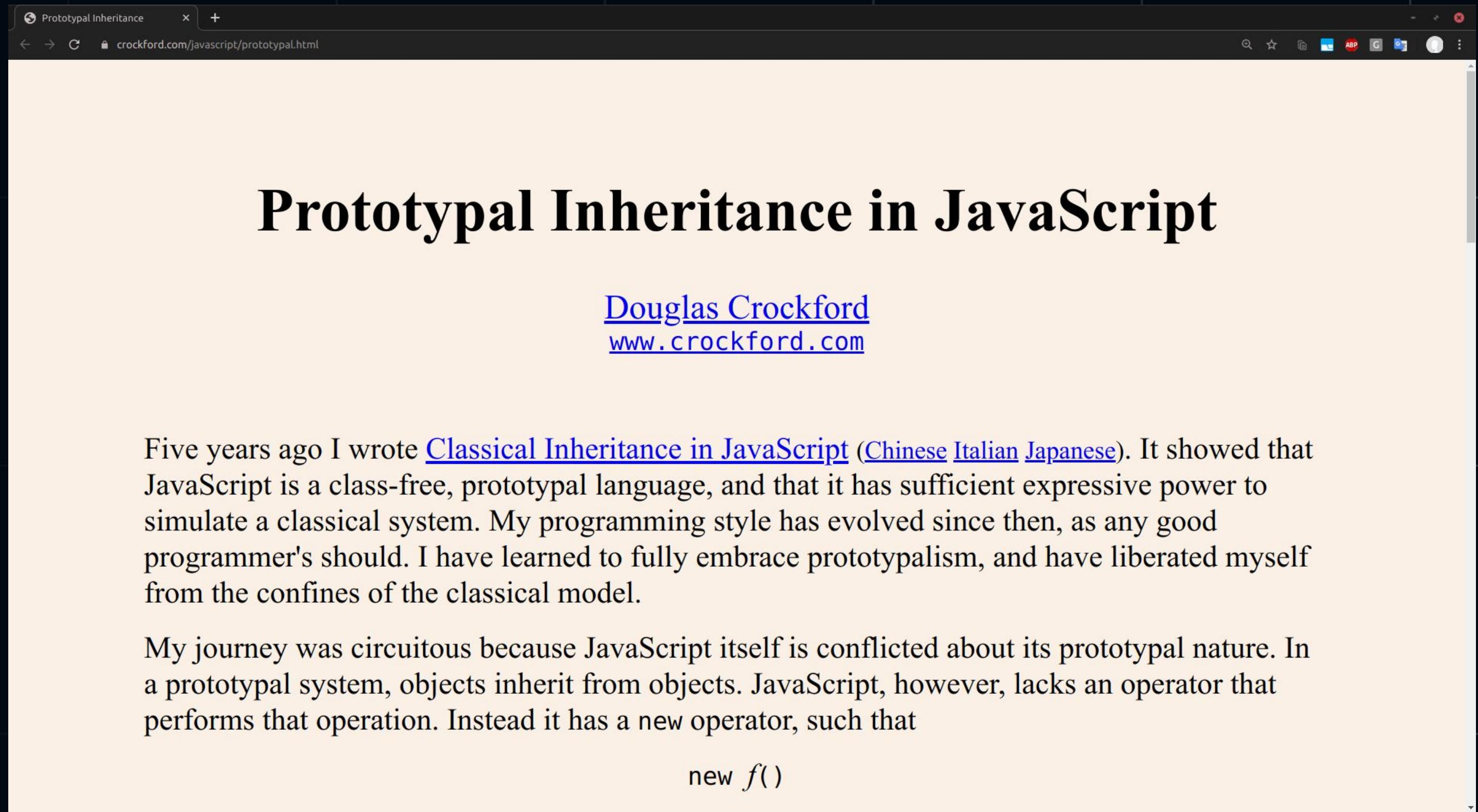
The screenshot shows a web browser window with the following details:

- Title bar: JavaScript: The World's Most Misunderstood Programming Language - crockford.com/javascript/javascript.html
- Address bar: Not secure | crockford.com/javascript/javascript.html
- Content area:
  - JavaScript:**  
**The World's Most Misunderstood  
Programming Language**
  - Douglas Crockford  
[www.crockford.com](http://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



The image shows a screenshot of a web browser window. The title bar reads "Prototypal Inheritance". The address bar shows the URL "crockford.com/javascript/prototypal.html". The main content area of the browser displays the text from the original article. The text is centered and reads:

## Prototypal Inheritance in JavaScript

Douglas Crockford  
[www.crockford.com](http://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



MDN web docs  Technologies ▾ References & Guides ▾ Feedback ▾ Search MDN 

## Inheritance and the prototype chain

[Edit in wiki](#)

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

**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](#)

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.

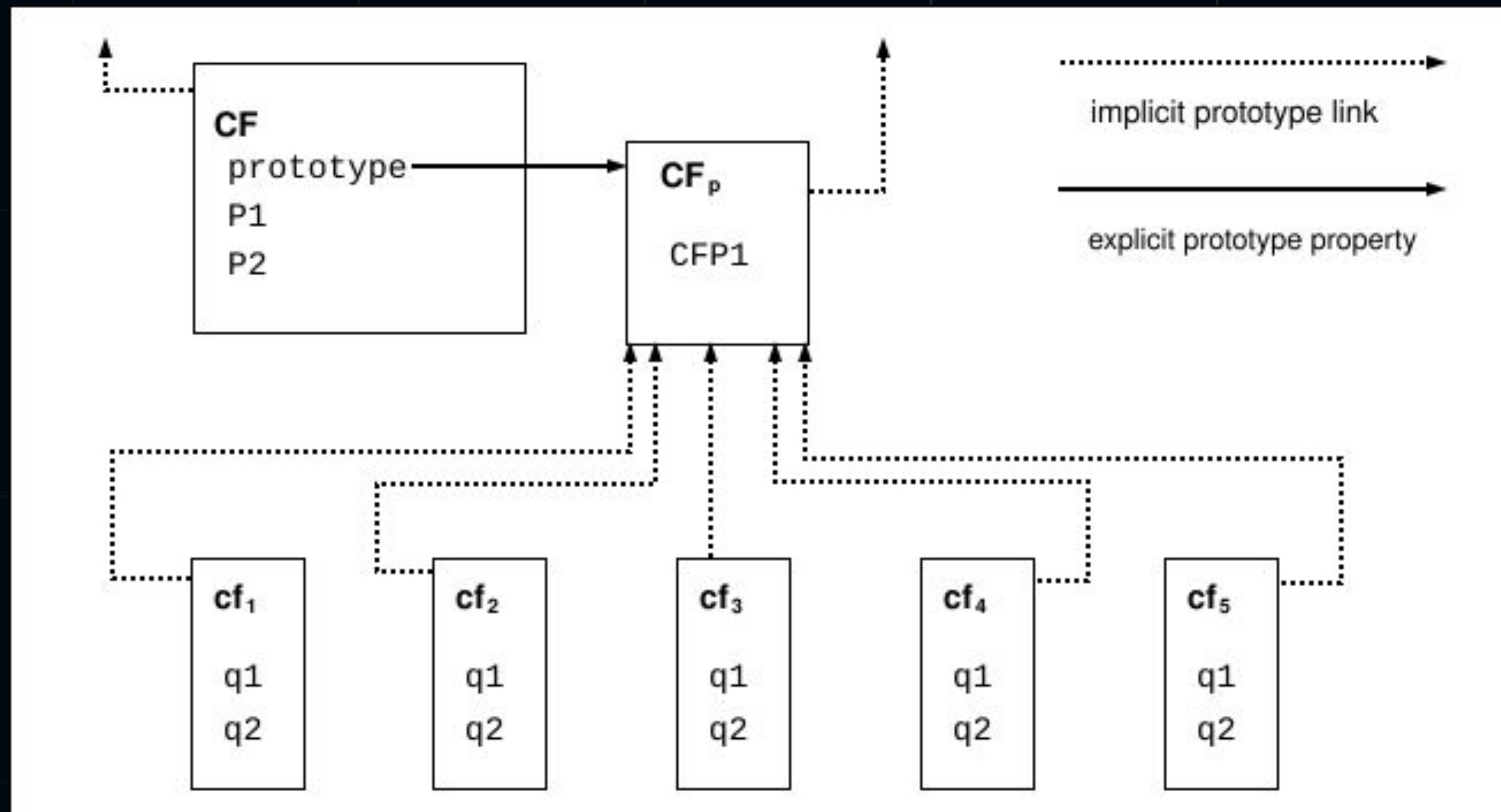
---

**References:**

- ▶ Built-in objects
- ▶ Expressions & operators

### Inheritance with the prototype chain

# Class vs Function



# Class vs Function



BrendanEich ✅

@BrendanEich

Replies 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

Replies to @BrendanEich, @Andre\_487 and @jsunderhood

It is absolutely Outstanding point!



# Class vs Function



BrendanEich ✅

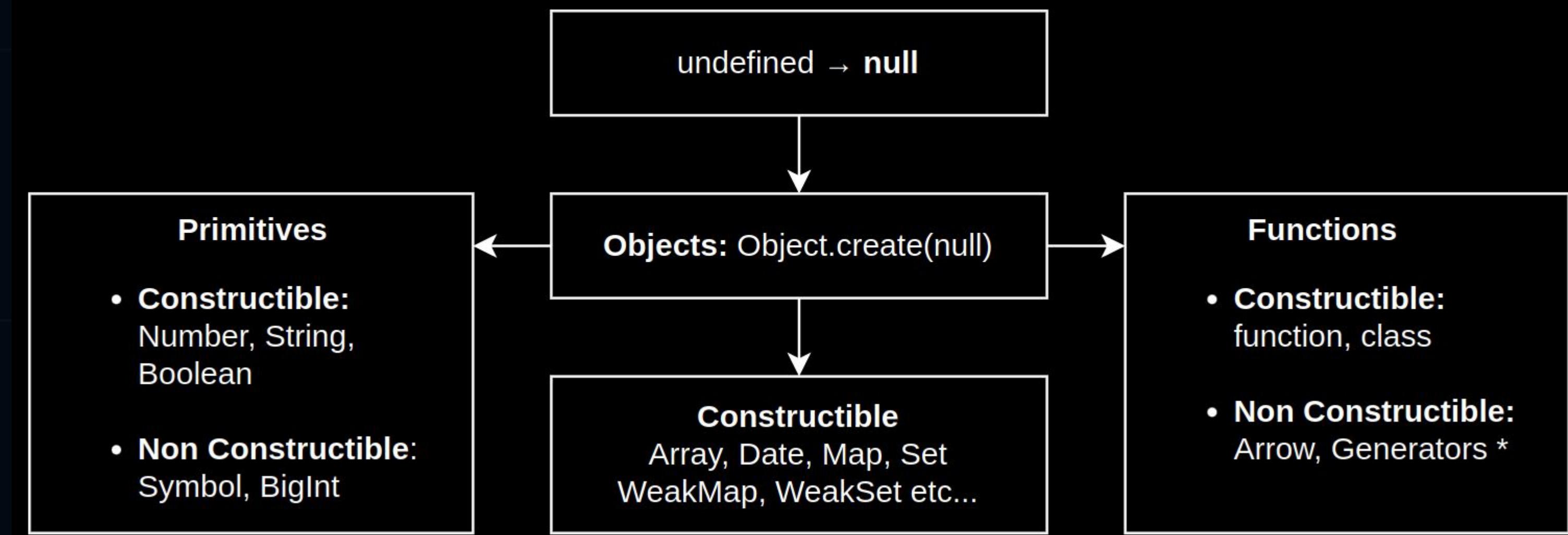
@BrendanEich

Replies 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

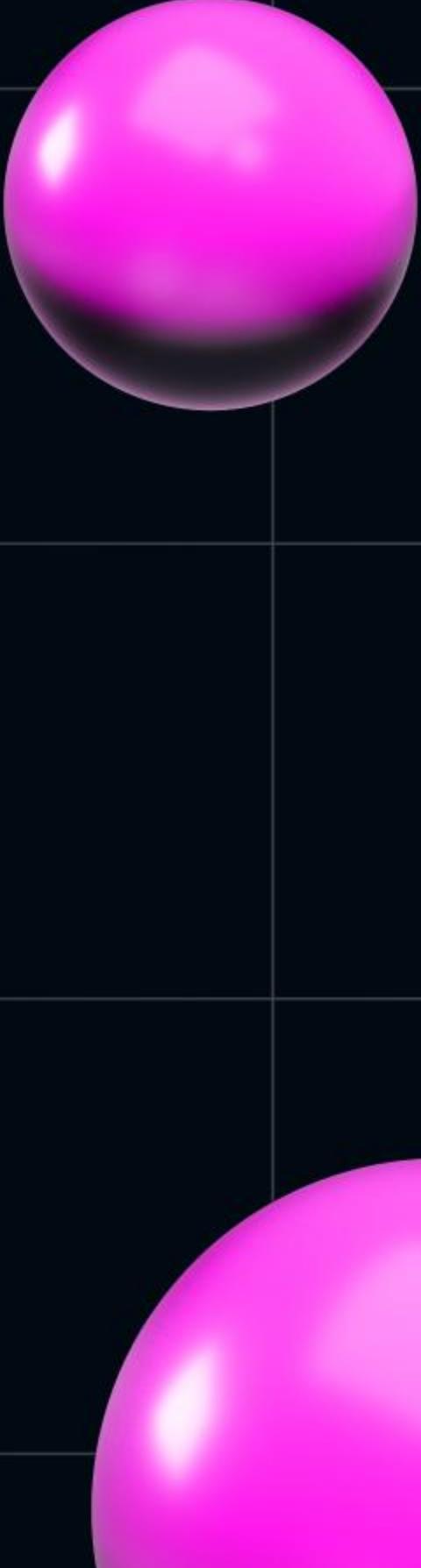
## JavaScript Objects Topology



# Class vs Function



# Class vs Function

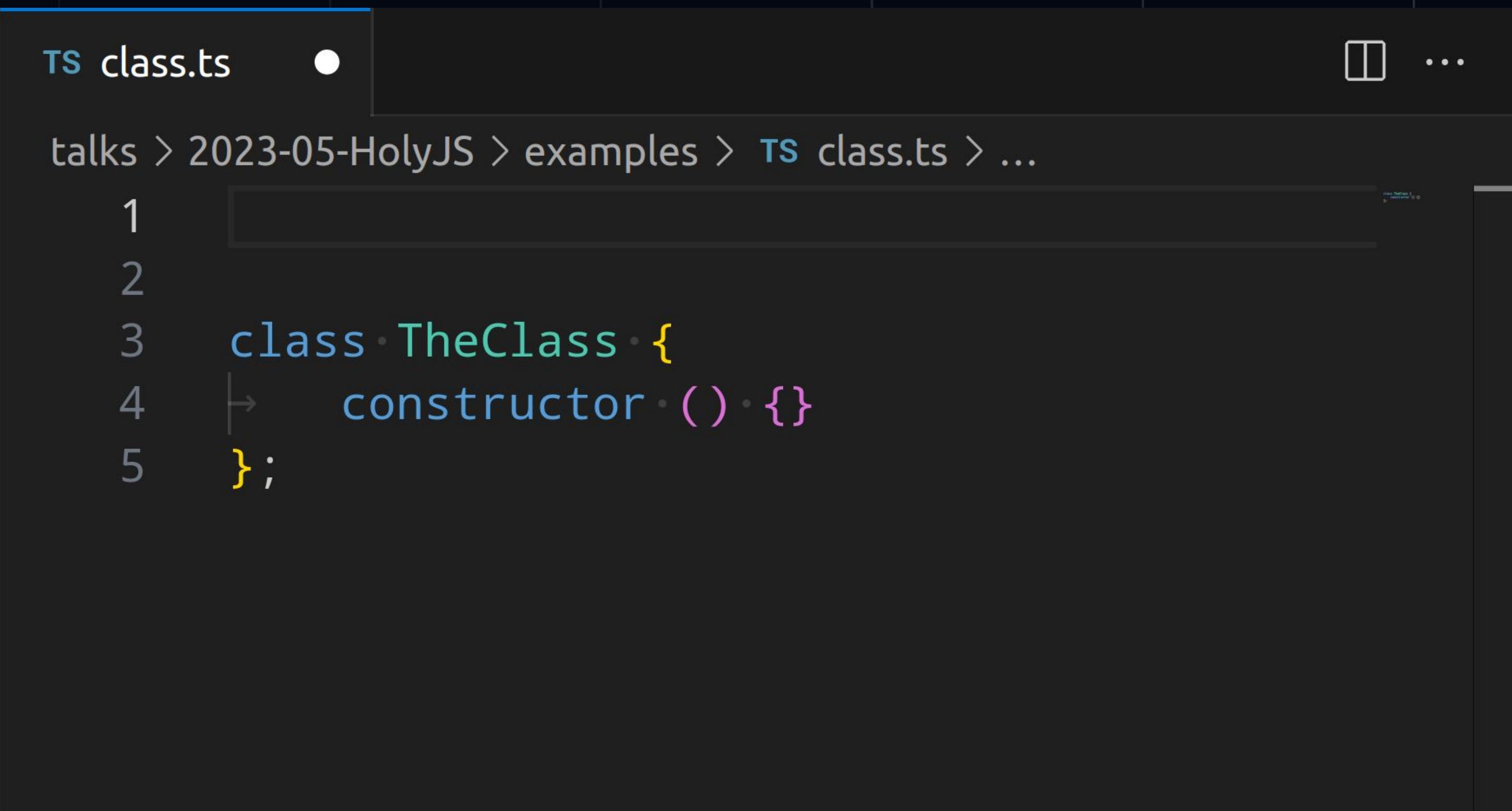


TS class.ts X ⏹ ...

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

```
1
2
3 class ·TheClass ·{} ;|
```

# Class vs Function



TS class.ts

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

```
1
2
3 class ·TheClass ·{
4   ↗ constructor ·() ·{}
5 }
```

# Class vs Function

TS class.ts



...

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

```
1
2
3  class ·BaseClass ·{
4    → constructor ·() ·{}
5  };
6  class ·ExtendedClass ·extends ·BaseClass ·{
7    };
```



# Class vs Function

TS class.ts

TS class\_extends.ts

JS function.js ×

...

talks > 2023-05-HolyJS > examples > JS function.js > ...

```
1
2
3 function·Construct·()·{};
4
5 const·item·=·new·Construct;
6
7 console.log(item);
```



# Class vs Function

TS class.ts

TS class\_extends.ts

JS function.js

...

talks > 2023-05-HolyJS > examples > JS function.js > ...

```
1 function · Construct · () · {};  
2  
3 Construct.prototype · = · { · field: · 123 · }  
4 Construct.prototype.constructor ·  
5 →     = · Construct;  
6  
7 const · item · = · new · Construct;  
8  
9 console.log(item);  
10
```





time matters

the difference



# Time Matters



ts class\_extends\_new.ts X

talks > 2023-05-HolyJS > examples > ts class\_extends\_new.ts > BaseClass > constructor

```
1  class·BaseClass·{  
2    field:·number  
3    constructor·()·{  
4      this.field·=·321;  
5    }  
6  };  
7  class·ExtendedClass·extends·BaseClass·{  
8    constructor·()·{  
9      super();  
10     this.field·=·123;  
11   }  
12 };  
13  
14  const·item·=·new·ExtendedClass;  
15  console.log(item);  
16
```

# Time Matters

```
ts function_construct_extended.ts ×  
talks > 2023-05-HolyJS > examples > ts function_construct_extended.ts > ...  
1  function Construct() {};  
2  Construct.prototype = { field: 123 }  
3  Construct.prototype.constructor = Construct;  
4  const item = new Construct;  
5  console.log(item);  
6  
7  function ExtendedConstruct() {};  
8  Object.setPrototypeOf(Construct.prototype, item);  
9  Construct.prototype.field = 321;  
10 const extendedItem = new ExtendedConstruct;  
11  
12 console.log(extendedItem);
```



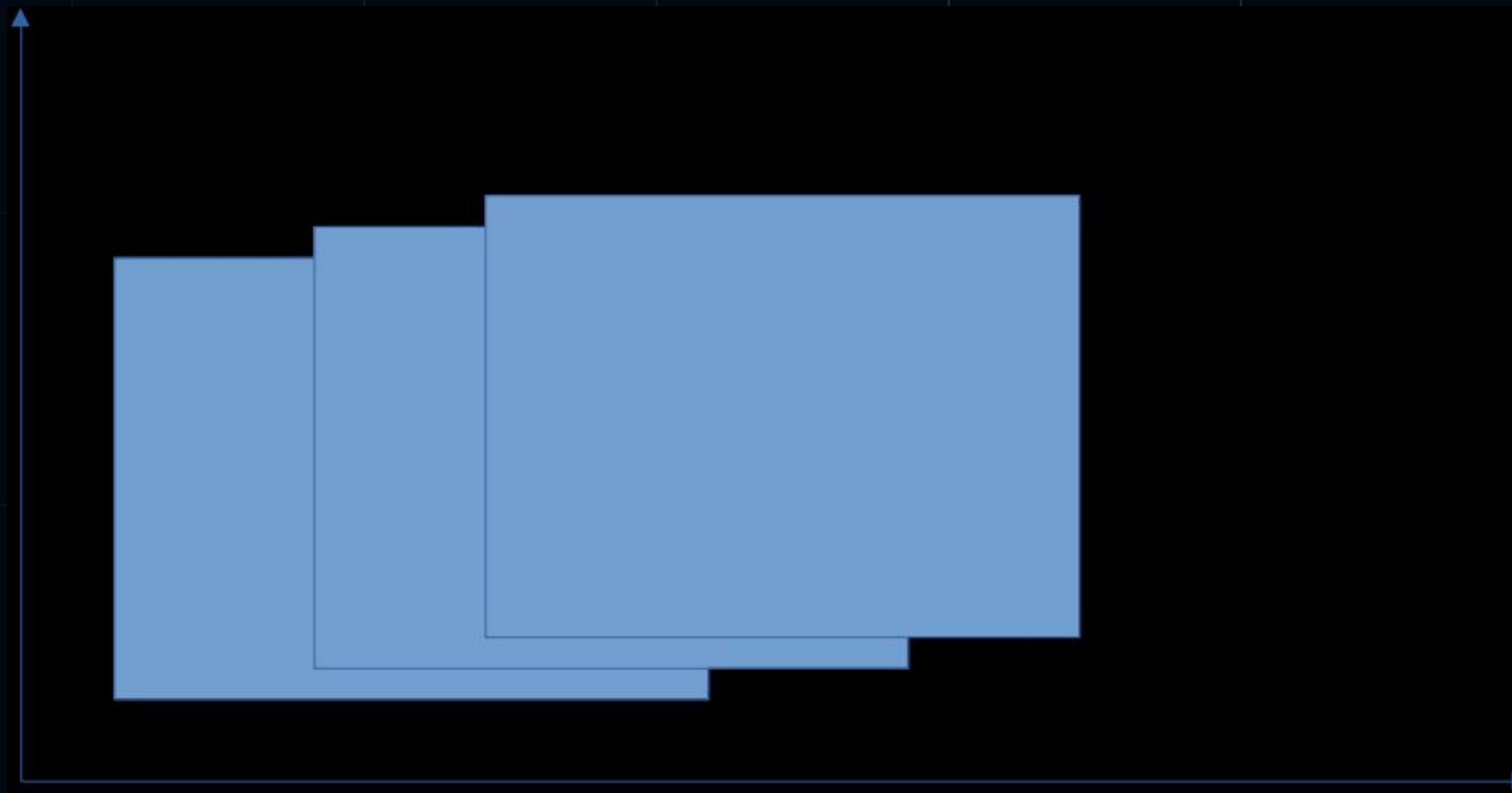
# Time Matters



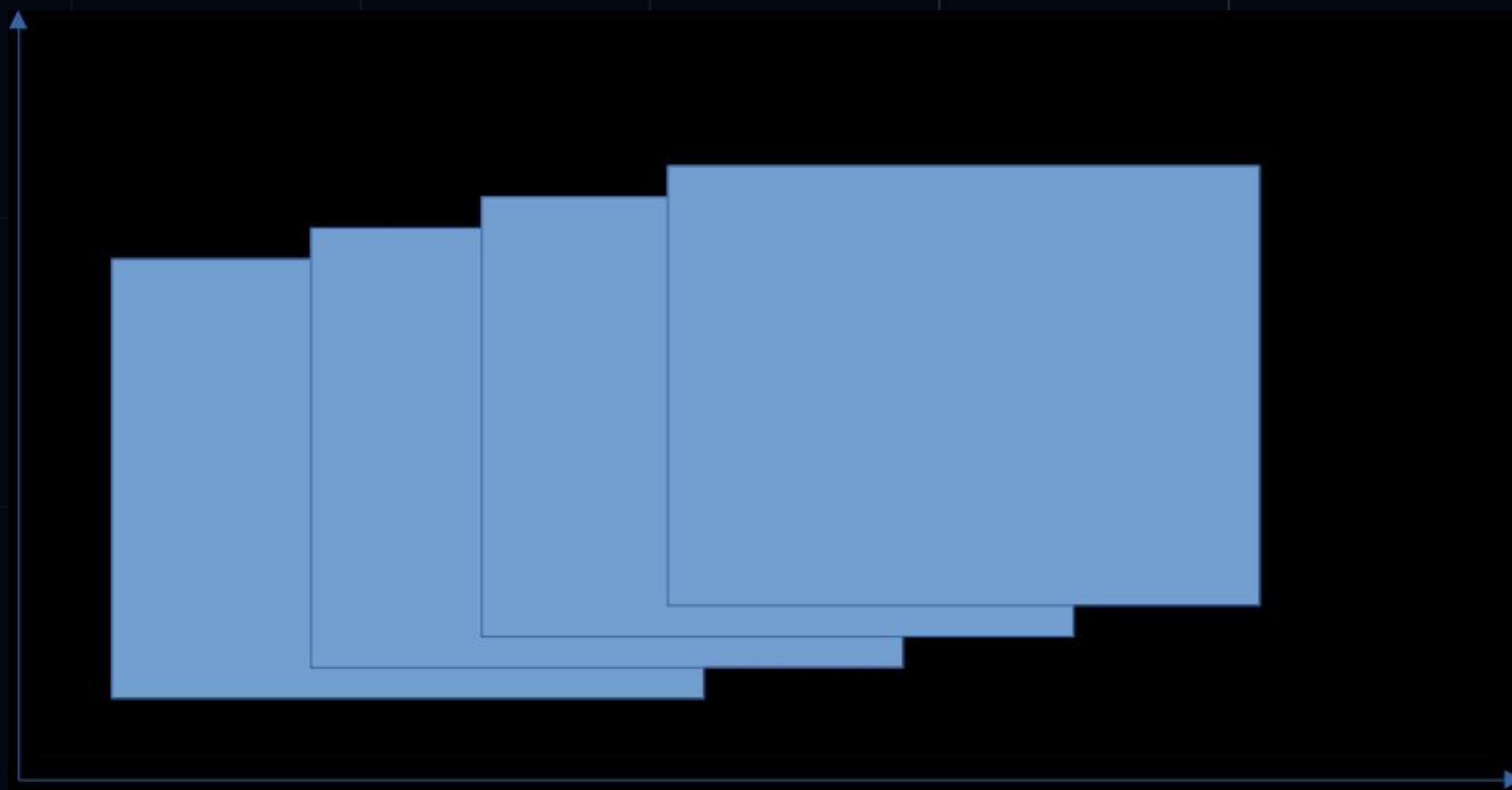
# Time Matters



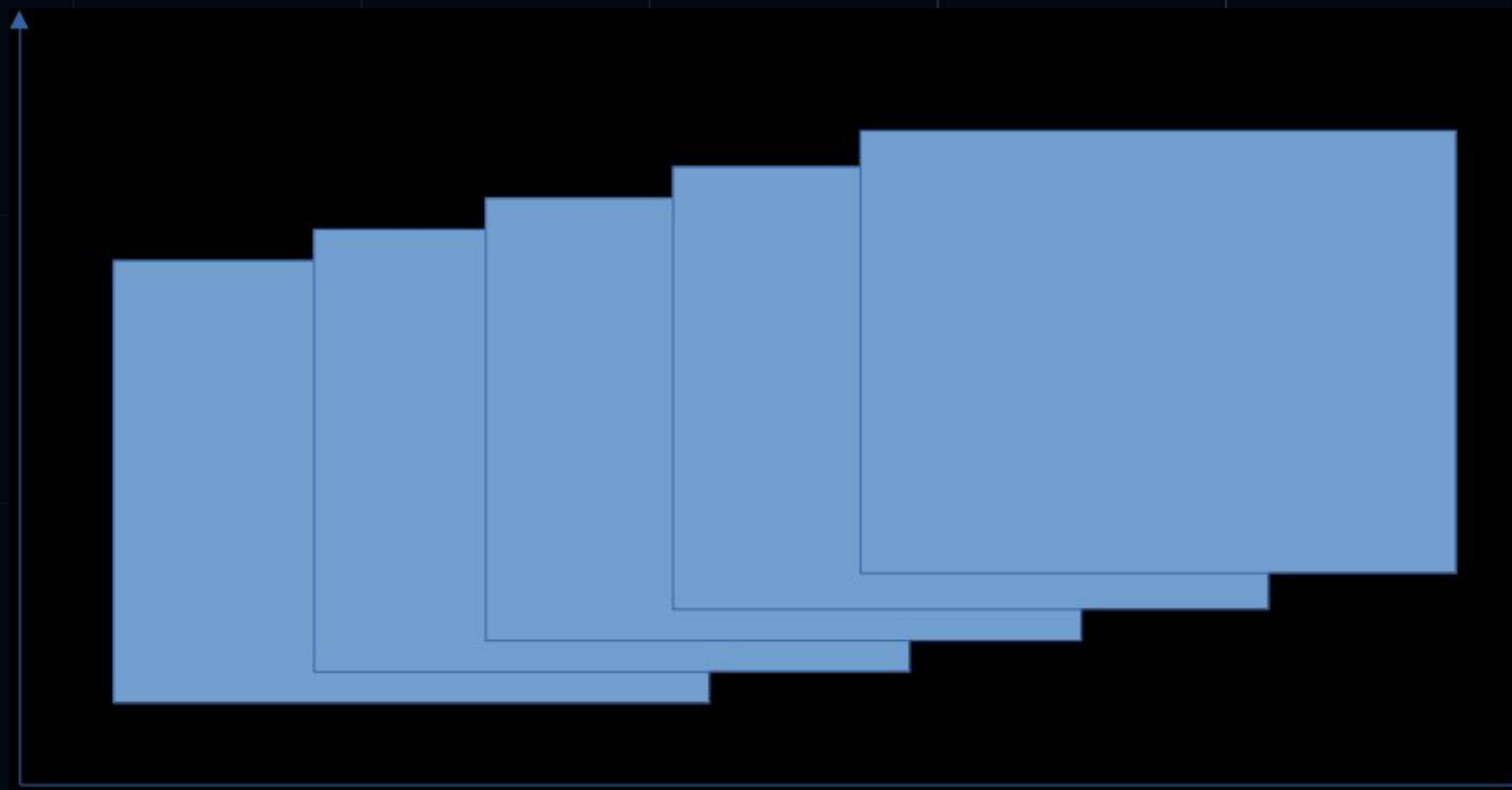
# Time Matters



# Time Matters



# Time Matters

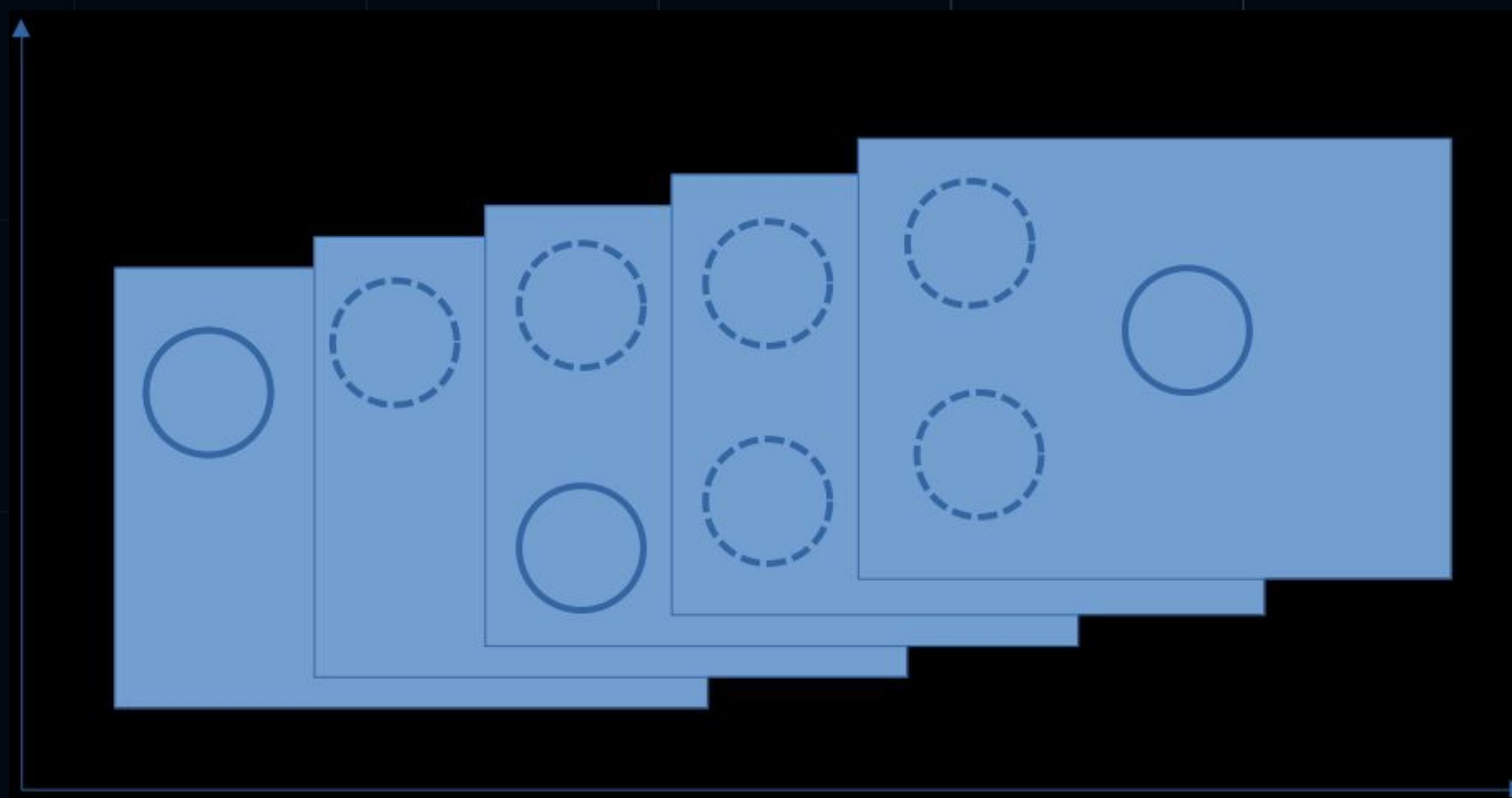


# Time Matters

```
ts function_construct_extended.ts ×  
talks > 2023-05-HolyJS > examples > ts function_construct_extended.ts > ...  
1  function Construct() {};  
2  Construct.prototype = { field: 123 }  
3  Construct.prototype.constructor = Construct;  
4  const item = new Construct;  
5  console.log(item);  
6  
7  function ExtendedConstruct() {};  
8  Object.setPrototypeOf(Construct.prototype, item);  
9  Construct.prototype.field = 321;  
10 const extendedItem = new ExtendedConstruct;  
11  
12 console.log(extendedItem);
```



# Time Matters



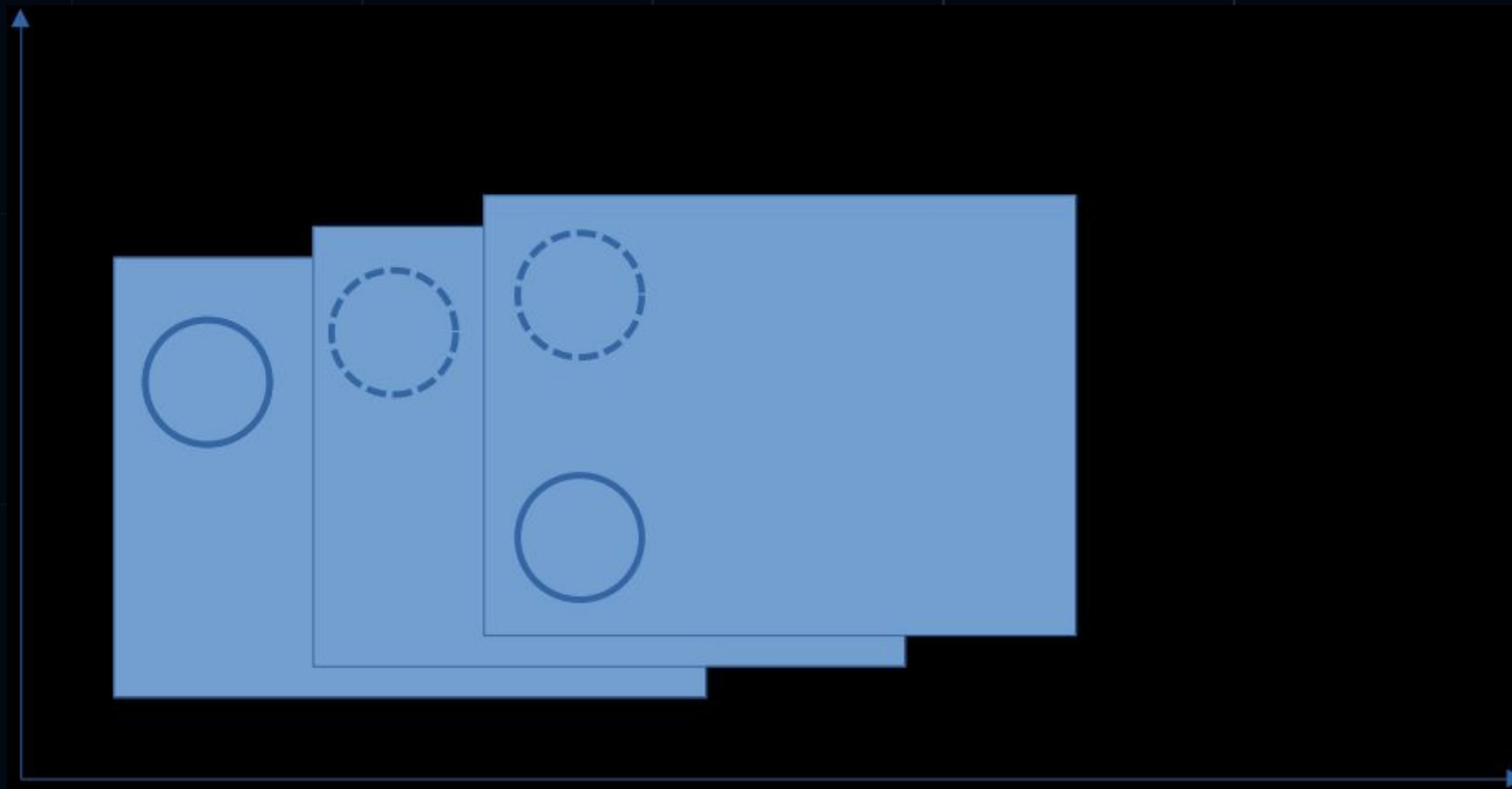
# Time Matters



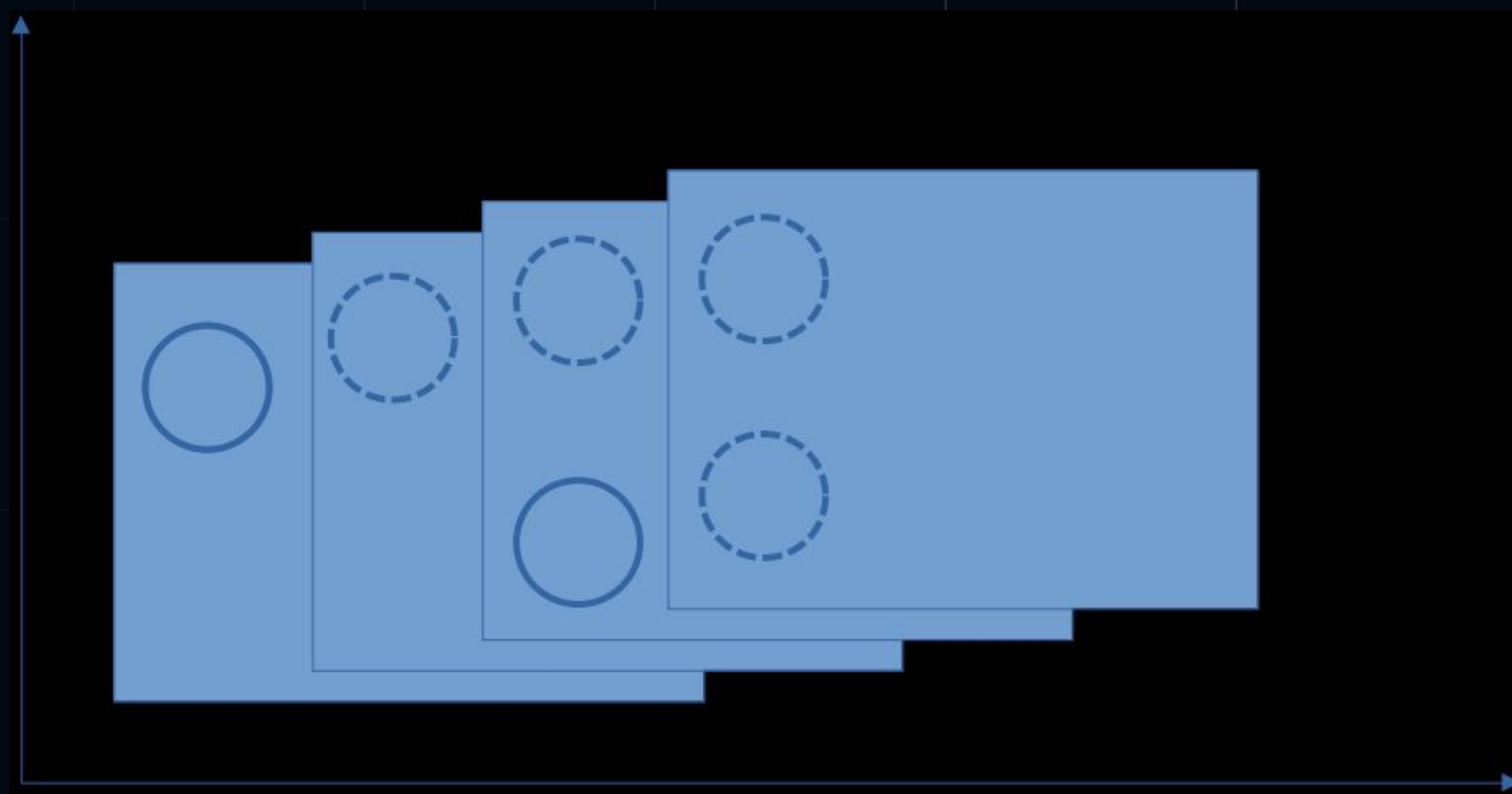
# Time Matters



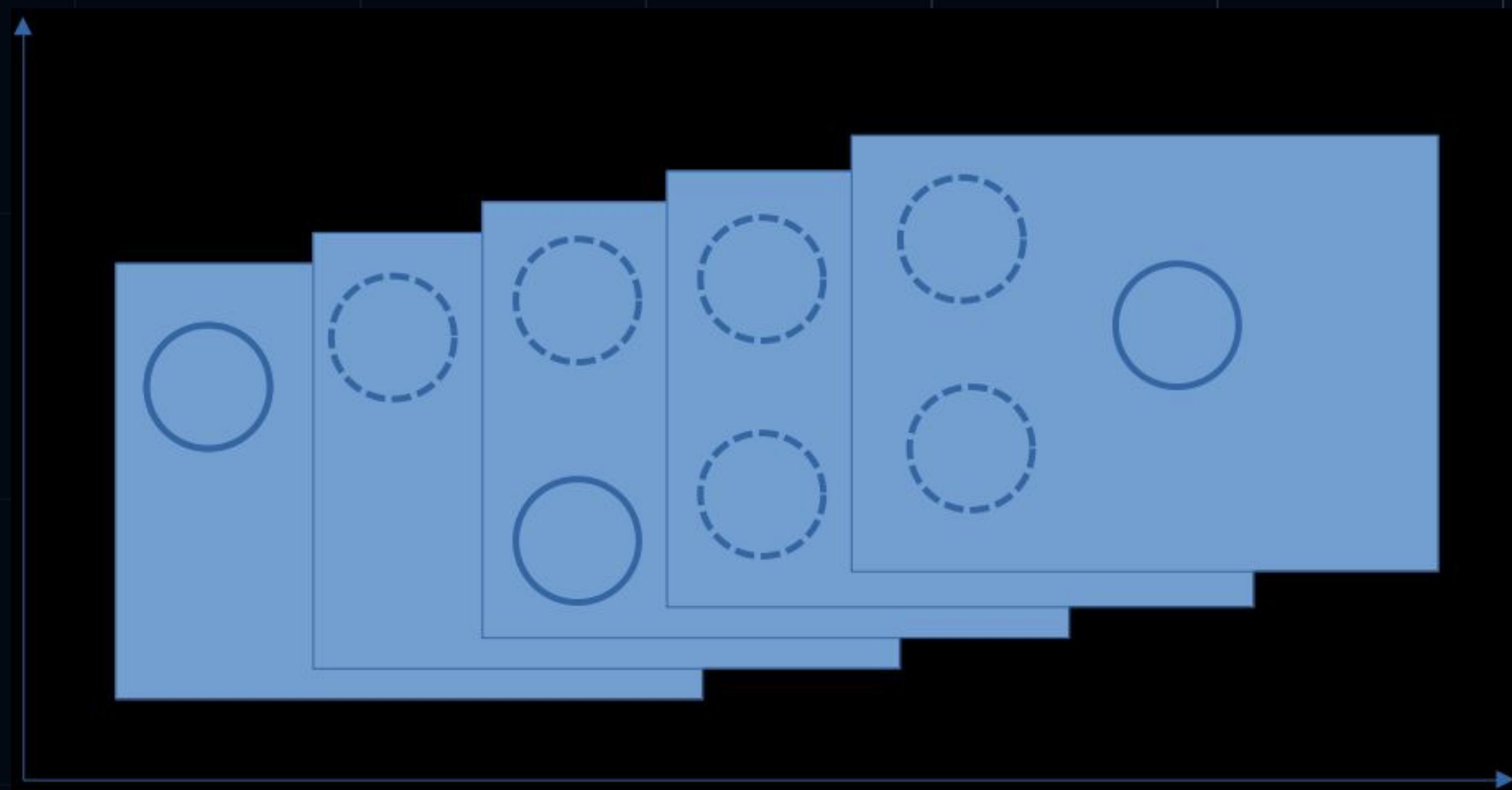
# Time Matters



# Time Matters



# Time Matters



# Class vs Function

ts class\_extends\_new.ts X

talks > 2023-05-HolyJS > examples > ts class\_extends\_new.ts > BaseClass > constructor

```
1  class ·BaseClass ·{  
2    →   field: ·number  
3    →   constructor ·() ·{  
4      →     this.field ·= ·321;  
5    }  
6  };  
7  class ·ExtendedClass ·extends ·BaseClass ·{  
8    →   constructor ·() ·{  
9      →     super();  
10     →     this.field ·= ·123;  
11    }  
12  };  
13  
14  const ·item ·= ·new ·ExtendedClass;  
15  console.log(item);  
16
```



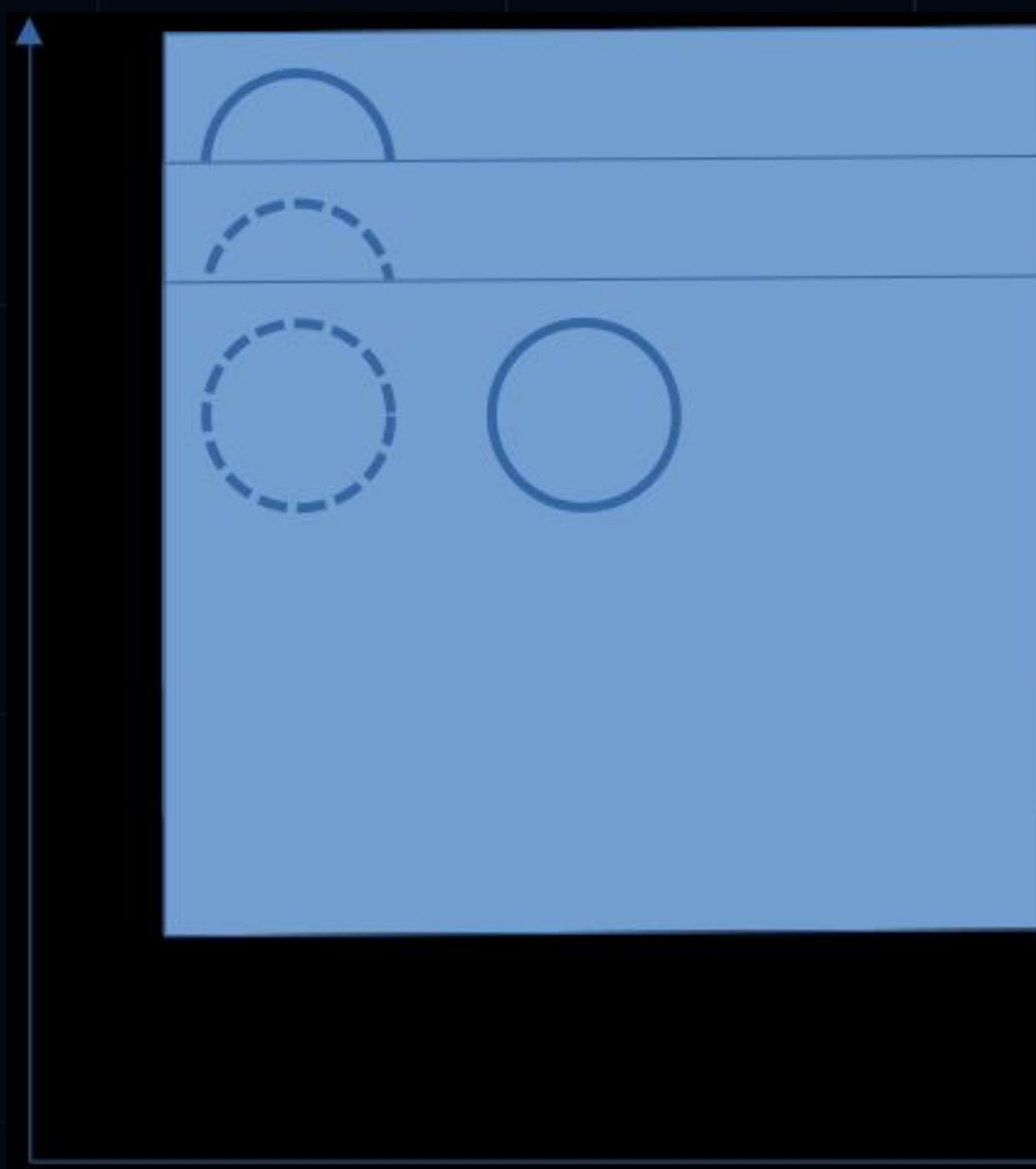
# Time Matters



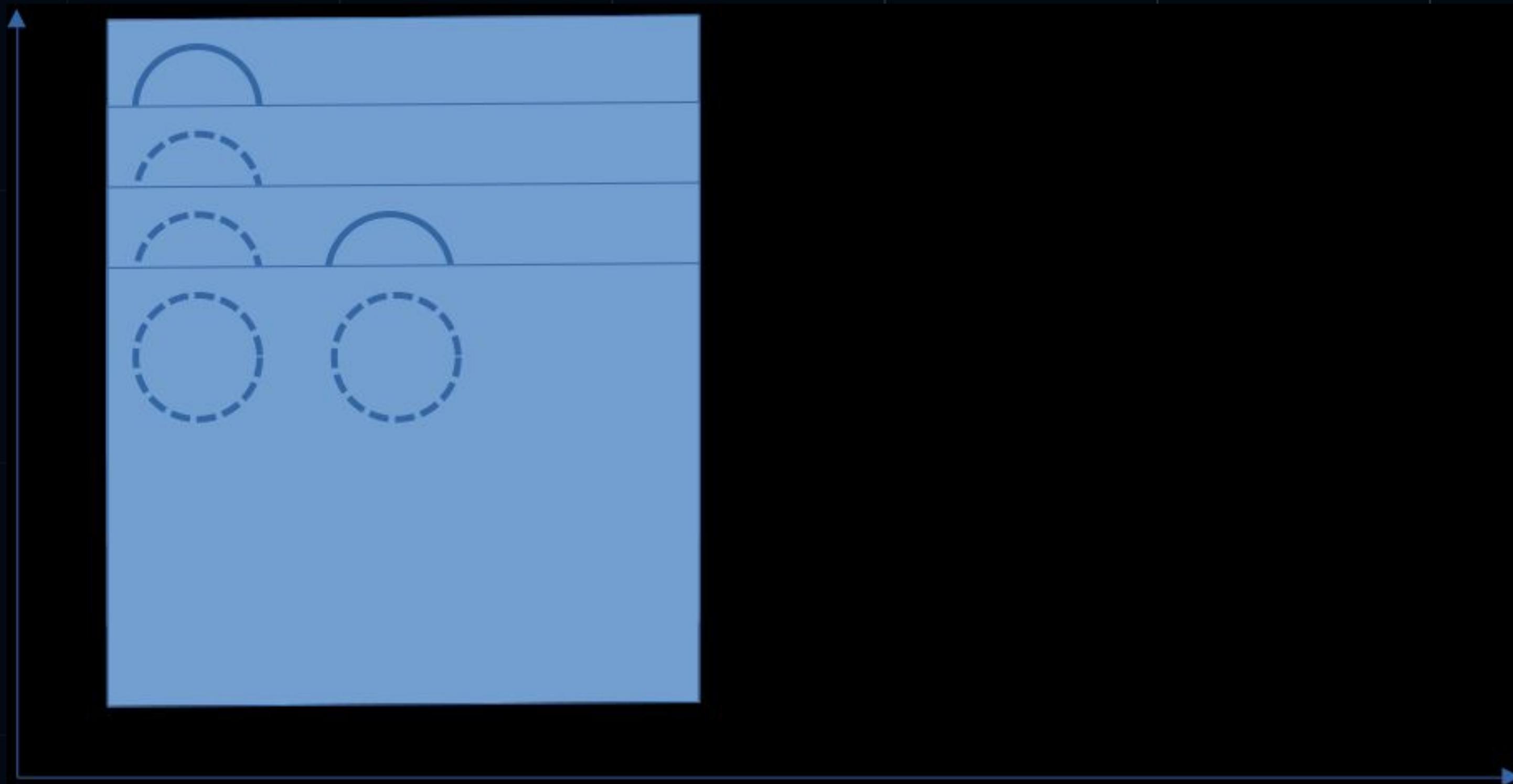
# Time Matters



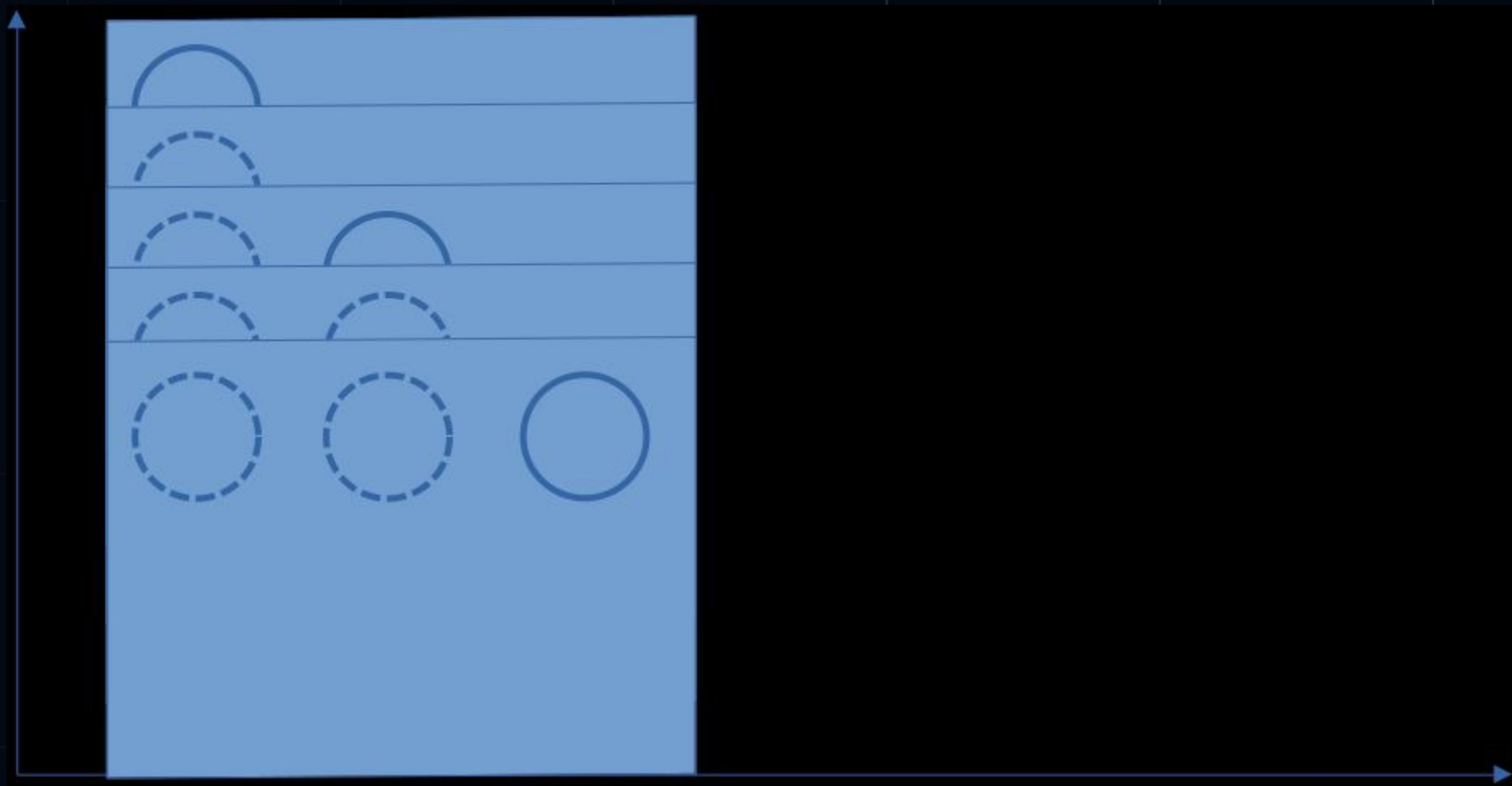
# Time Matters



# Time Matters



# Time Matters



# Time Matters



ts class\_extends\_new.ts X

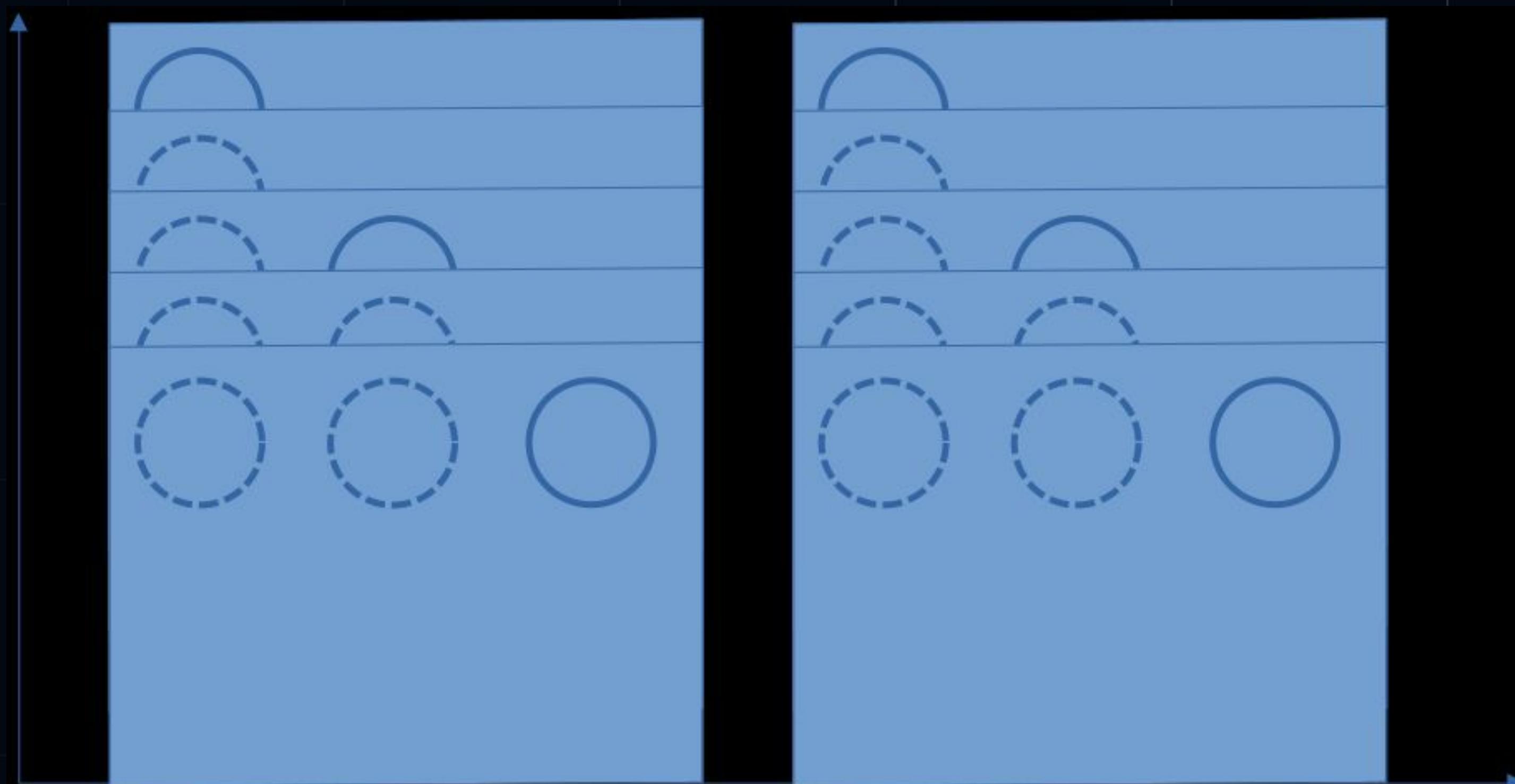
talks > 2023-05-HolyJS > examples > ts class\_extends\_new.ts > BaseClass > constructor

```
1  class·BaseClass·{  
2    field:·number  
3    constructor·()·{  
4      this.field·=·321;  
5    }  
6  };  
7  class·ExtendedClass·extends·BaseClass·{  
8    constructor·()·{  
9      super();  
10     this.field·=·123;  
11   }  
12 };  
13  
14  const·item·=·new·ExtendedClass;  
15  console.log(item);  
16
```

# Time Matters

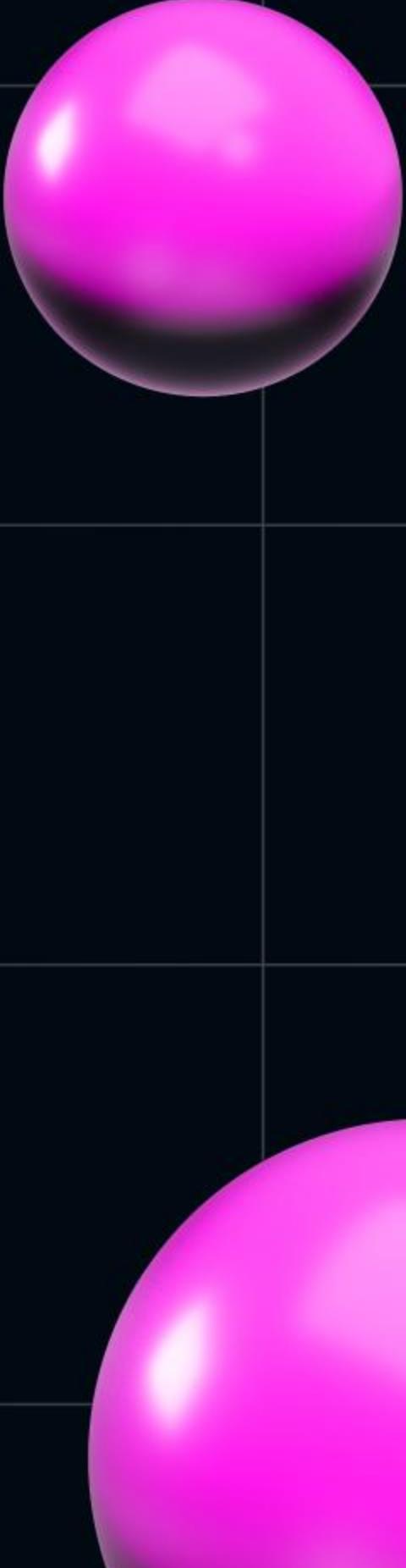
```
ts class_extends_new.ts ×  
talks > 2023-05-HolyJS > examples > TS class_extends_new.ts > ...  
13  
14  
15 const item = new ExtendedClass;  
16 console.log(item);  
17  
18 // number  
19 type itemField = typeof item.field;  
20  
21
```

# Time Matters





# Time Matters



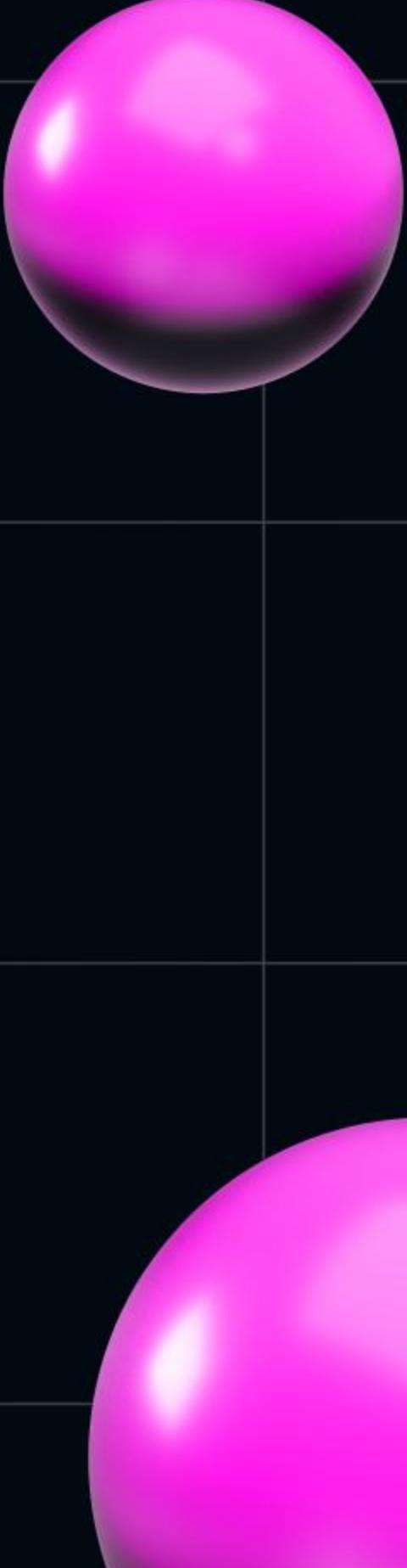
```
ts class_extends_new.ts X
talks > 2023-05-HolyJS > examples > ts class_extends_new.ts > ...
6  };
7  class ExtendedClass extends BaseClass {
8    constructor () {
9      super();
10     this.field = 123;
11   }
12 }
13
14  const item: ExtendedClass
15 const item = new ExtendedClass;
16 console.log(item);
17
18 // number
19 type itemField = typeof item.field;
20
21 |
```

# Time Matters

```
ts function_construct_extended.ts ×  
talks > 2023-05-HolyJS > examples > ts function_construct_extended.ts > ...  
1  function Construct() {};  
2  Construct.prototype = { field: 123 }  
3  Construct.prototype.constructor = Construct;  
4  const item = new Construct;  
5  console.log(item);  
6  
7  function ExtendedConstruct() {};  
8  Object.setPrototypeOf(Construct.prototype, item);  
9  Construct.prototype.field = 321;  
10 const extendedItem = new ExtendedConstruct;  
11  
12 console.log(extendedItem);
```



# Time Matters

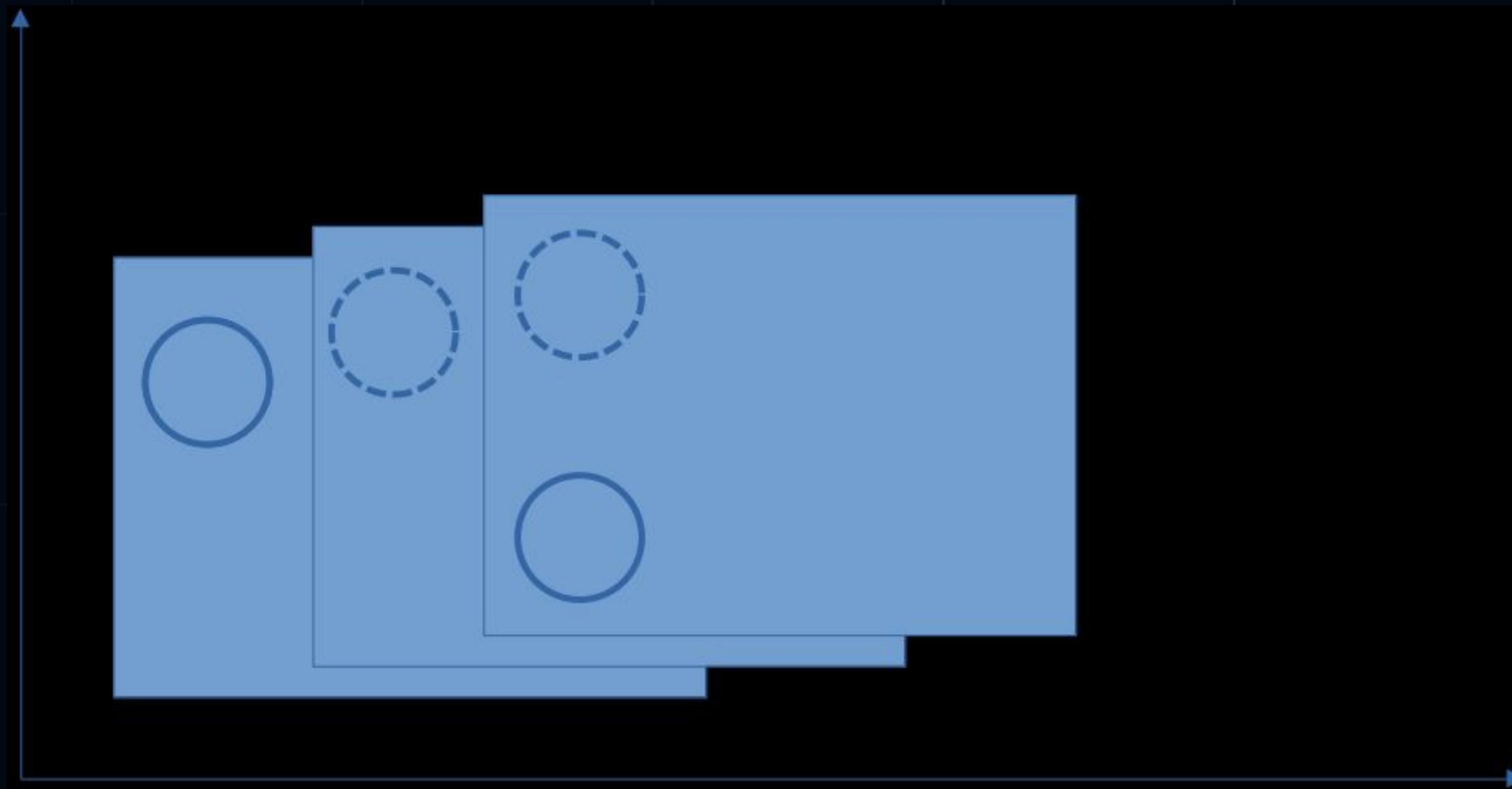


ts function\_construct\_extended.ts X

talks > 2023-05-HolyJS > examples > ts function\_construct\_extended.ts > ...

```
9
10 // ·any ·...
11 Construct.prototype.field ·= ·321;
12
13
14 const ·extendedItem ·= ·new ·ExtendedConstruct;
15 console.log(extendedItem);
16
17 // ·any ·...
18 type ·extendedItemField ·= ·typeof ·extendedItem.field;
```

# Time Matters





# Time Matters



```
TS function_construct_extended_this_typed.ts ×

talks > 2023-05-HolyJS > examples > TS function_construct_extended_this_typed.ts > [e] extendedItemField

1  function Construct (this: {field: number}) {}
2  Construct.prototype = { field: 123 }
3  Construct.prototype.constructor = Construct;
4  const item = new Construct;
5  console.log(item);
6
7  function ExtendedConstruct (this: {field: number}) {}
8  Object.setPrototypeOf(Construct.prototype, item);
9
10 // any ...
11 Construct.prototype.field = 321;
12
13
14 const extendedItem = new ExtendedConstruct;
15 console.log(extendedItem);
16
17 // any ...
18 type extendedItemField = typeof extendedItem.field;
```

# Time Matters

```
ts function_construct_extended_this_typed.ts X
talks > 2023-05-HolyJS > examples > ts function_construct_extended_this_typed.ts > [e] extendedItemField
1  function Construct (this: {field: number}) {}
2  Construct.prototype = { field: 123 }
3  Construct.prototype.constructor = Construct;
4  const item = new Construct;
5  console.log(item);
6
7  function ExtendedConstruct (this: {field: number}) {}
8  Object.setPrototypeOf(Construct.prototype, item);
9
10 // any ...
11 Construct.prototype.field = 321;
12
13
14 const extendedItem = new ExtendedConstruct;
15 console.log(extendedItem);
16
17 // any ...
18 type extendedItemField = typeof extendedItem.field;
```





types of  
Inheritance  
in JS~TS



# types of Inheritance

TS RunningObjectProps.ts X

\_ProtoTypes > TS RunningObjectProps.ts > ...

```
1 const remapKeys = (
2   obj: Record<string, number>,
3   remapMap: Record<string, string>
4 ) => {
5   for (const key in remapMap) {
6     obj[remapMap[key]] = obj[key];
7     delete obj[key];
8   }
9   return obj;
10 };
```



# types of Inheritance

TS RunningObjectProps.ts ●



...

\_Prototypes > TS RunningObjectProps.ts > ...

```
15 const remapResult = remapKeys(  
16   { age: 1 },  
17   { age: "newAge" }  
18 );  
19  
20 remapResult.newAge // 1  
21  
22
```



# types of Inheritance

TS RunningObjectProps.ts •



...

\_ProtoTypes > TS RunningObjectProps.ts > ...

19

20 remapResult.newAge • // • 1

21

22

23 remapResult.age • // • 1

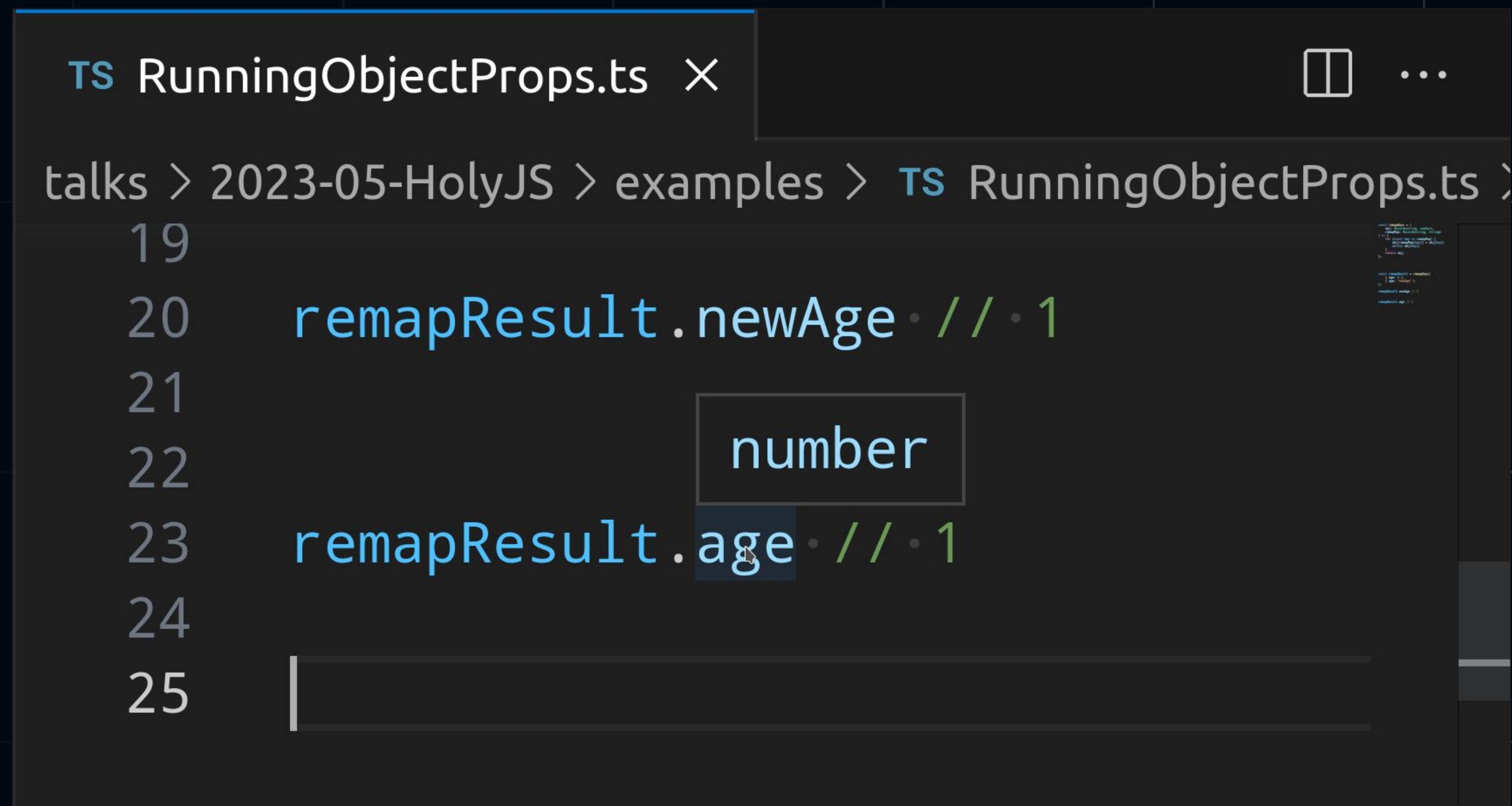
24

25 |

A screenshot of a code editor showing several lines of TypeScript code. The code includes variable declarations like 'const remapResult: { newAge: number; age: number; } = {}' and method definitions like 'function remap(result: { age: number; })'. The editor has a dark theme with syntax highlighting.



# types of Inheritance



TS RunningObjectProps.ts X

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

```
19
20   remapResult.newAge • // • 1
21
22   number
23   remapResult.age • // • 1
24
25 |
```



# 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**

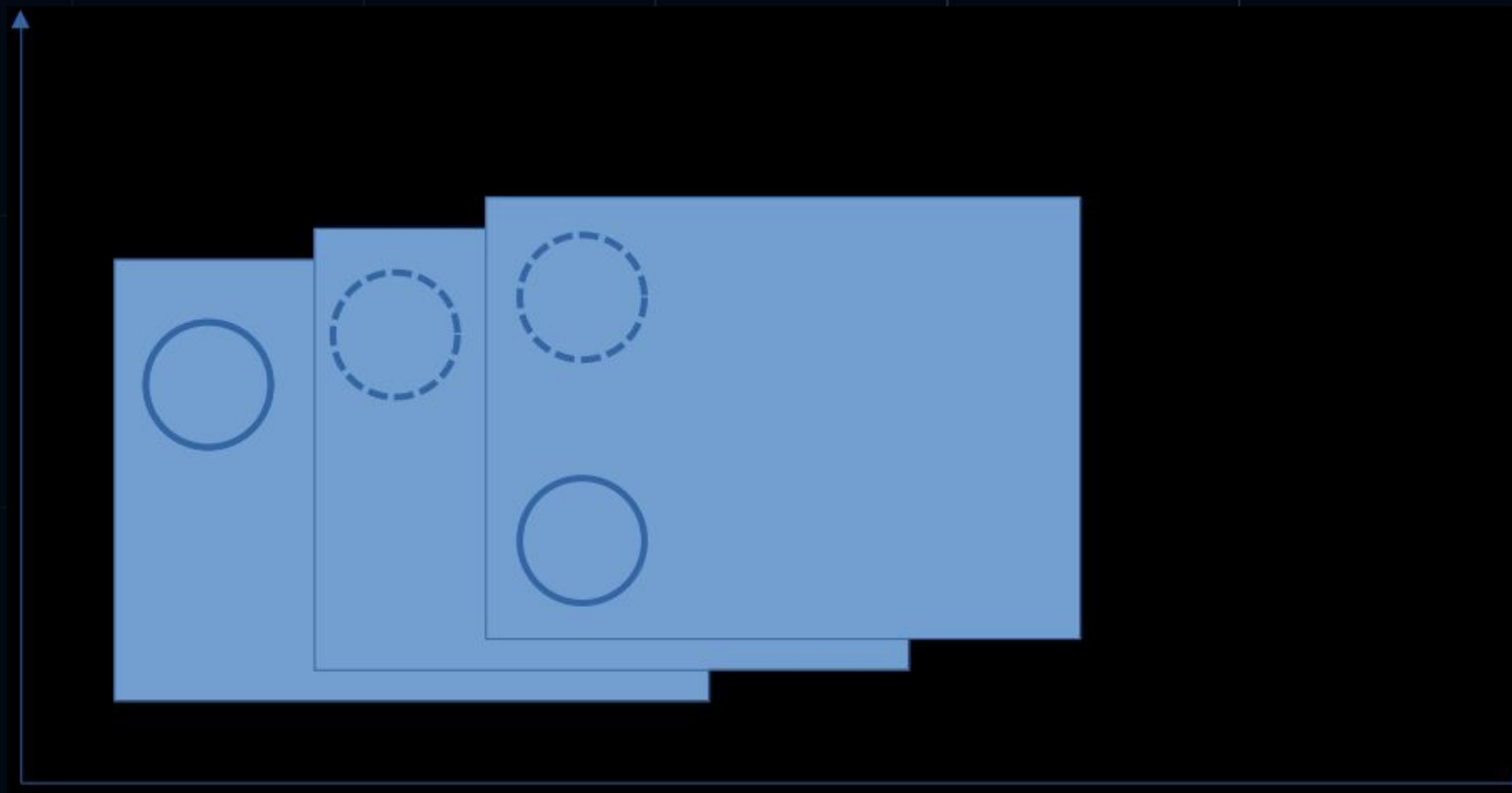


# Optional Fields

## 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 > ...

```
1 function·Construct·(this:·{field:·number}):·void·{};  
2  
3 Construct.prototype·=·{·field:·123·}  
4 Construct.prototype.constructor·  
5 →     =·Construct;  
6  
7 const·item·=·new·Construct;  
8  
9 |         const item: any  
10 console.log(item);  
11
```



# identity of chaining



```
ts function_construct_typed.ts ×

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

1  function OtherConstruct(this: { field: number }) {}
2
3  const define = function<T>(Cstr: { (this: T): void }) {
4    return function(): T {
5      return new Cstr;
6    };
7  };
8
9  const myConstruct = define(OtherConstruct);
10
11  const myConstructedItem: {
12    field: number;
13  }
14  const myConstructedItem = myConstruct();
15
16  console.log(myConstructedItem);
```

# identity of chaining

TS mixWithProto.ts ×

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

```
1 type init = {
2   s: number
3   z: number
4 }
5
6 type next = {
7   s?: string
8   m: boolean
9 }
10
```



# identity of chaining

TS mixWithProto.ts ×

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

```
10
11   type proto = {
12     z: number;
13   }
14   type proto = Pick<
15     init,
16     Exclude<
17       keyof init,
18       keyof next
19     >>
```



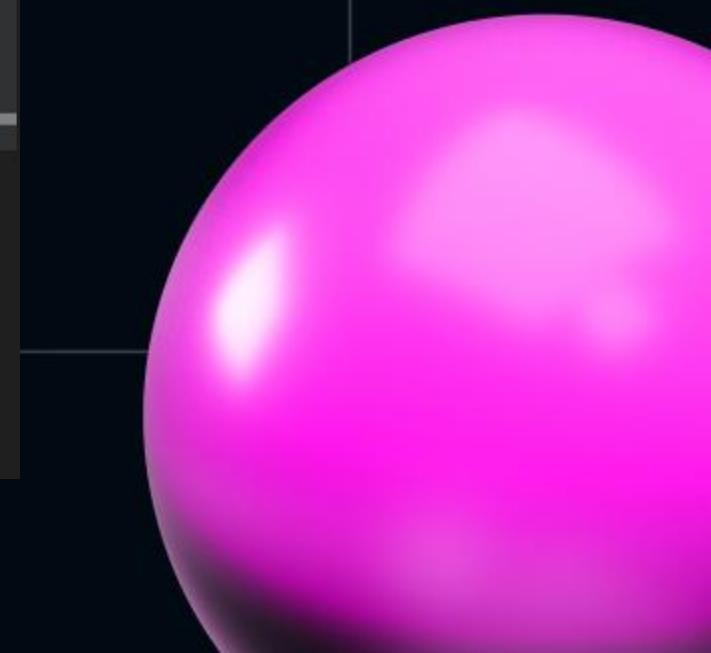
# identity of chaining

TS mixWithProto.ts X



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

```
22 type unit = proto & next
23
24 const aggregation: unit = {
25   z: 123,
26   s: 'x',
27   m: true,
28 };
29 type sss = string | undefined
30 type sss = typeof aggregation.s
31
```



# identity of chaining

```
ts function_construct_typed.ts X
talks > 2023-05-HolyJS > examples > ts function_construct_typed.ts > ...
1  function OtherConstruct(this: { field: number }) {}
2  OtherConstruct.prototype = {
3    otherField: true
4  }
5
6  type Proto<P, T> = Pick<P, Exclude<keyof P, keyof T>> & T;
7
8  const define = function <P extends object, T>(Cstr: { (this: T): void }, proto: {
9    const MyConstructor = function (): Proto<P, T> {
10      return new Cstr;
11    };
12    Object.setPrototypeOf(MyConstructor, proto);
13    return MyConstructor;
14  });
15
16  const myConstruct = define(OtherConstruct, { otherField: true });
17
18  const myConstructedItem = myConstruct();
19
20  console.log(myConstructedItem);
```

# identity of chaining

# DEMO





∅



# special thanks to



2018 Moscow

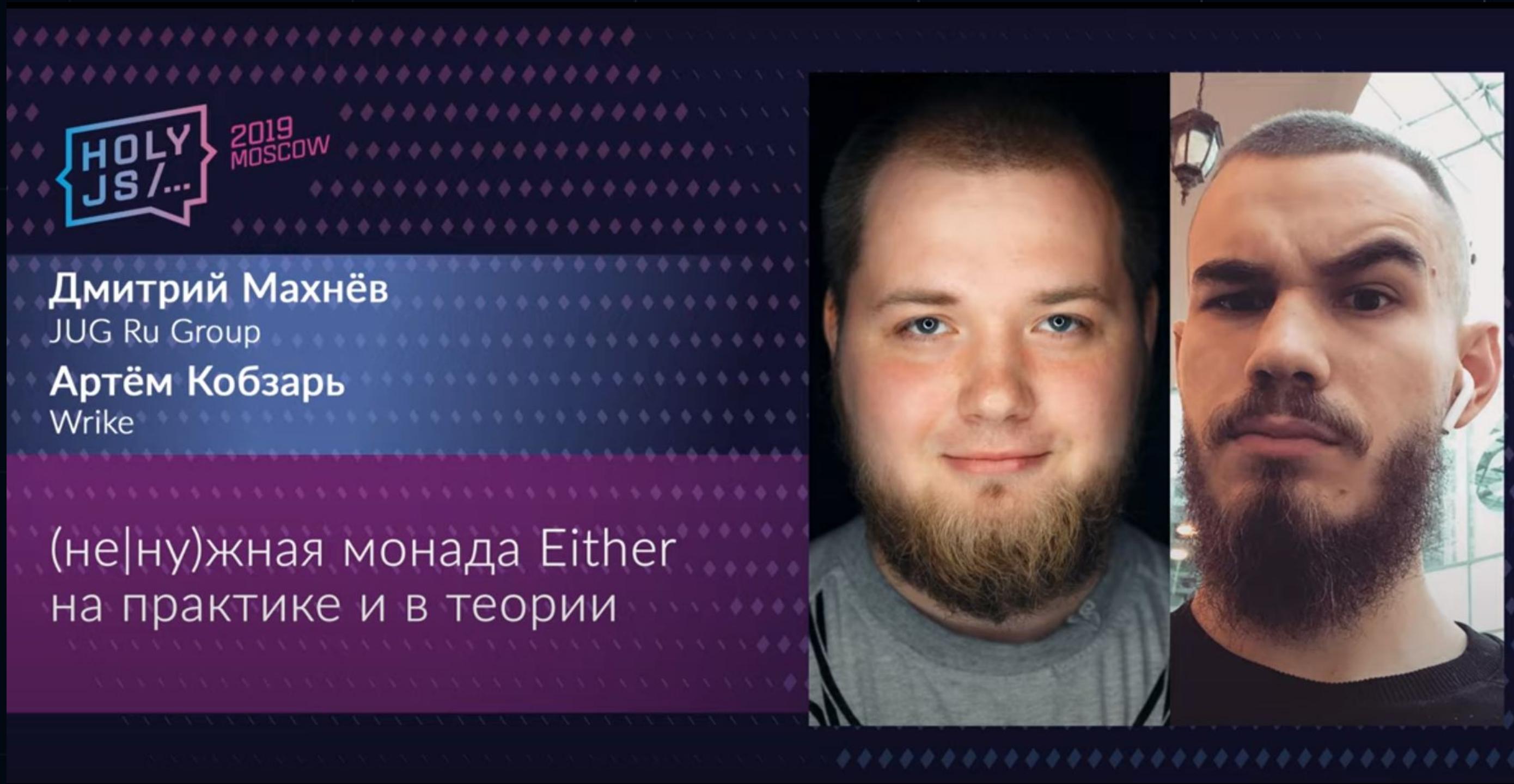
## Дмитрий Пацура

Fintier

Микросервисная архитектура



# special thanks to



The banner features a dark blue background with a light blue diamond pattern. In the top left corner is the HolyJS logo, which consists of the word "HOLY" above "JS" in a white, sans-serif font, all contained within a white speech bubble-like shape with blue outlines. To the right of the logo, the text "2019 MOSCOW" is written in a smaller, white, sans-serif font. Below the logo, there are two sections of text. The first section contains the name "Дмитрий Махнёв" in a large, bold, white sans-serif font, followed by "JUG Ru Group" in a smaller, white sans-serif font. The second section contains the name "Артём Кобзарь" in a large, bold, white sans-serif font, followed by "Wrike" in a smaller, white sans-serif font. At the bottom of the banner, the title "(не|ну)жная монада Either на практике и в теории" is displayed in a large, white, sans-serif font.

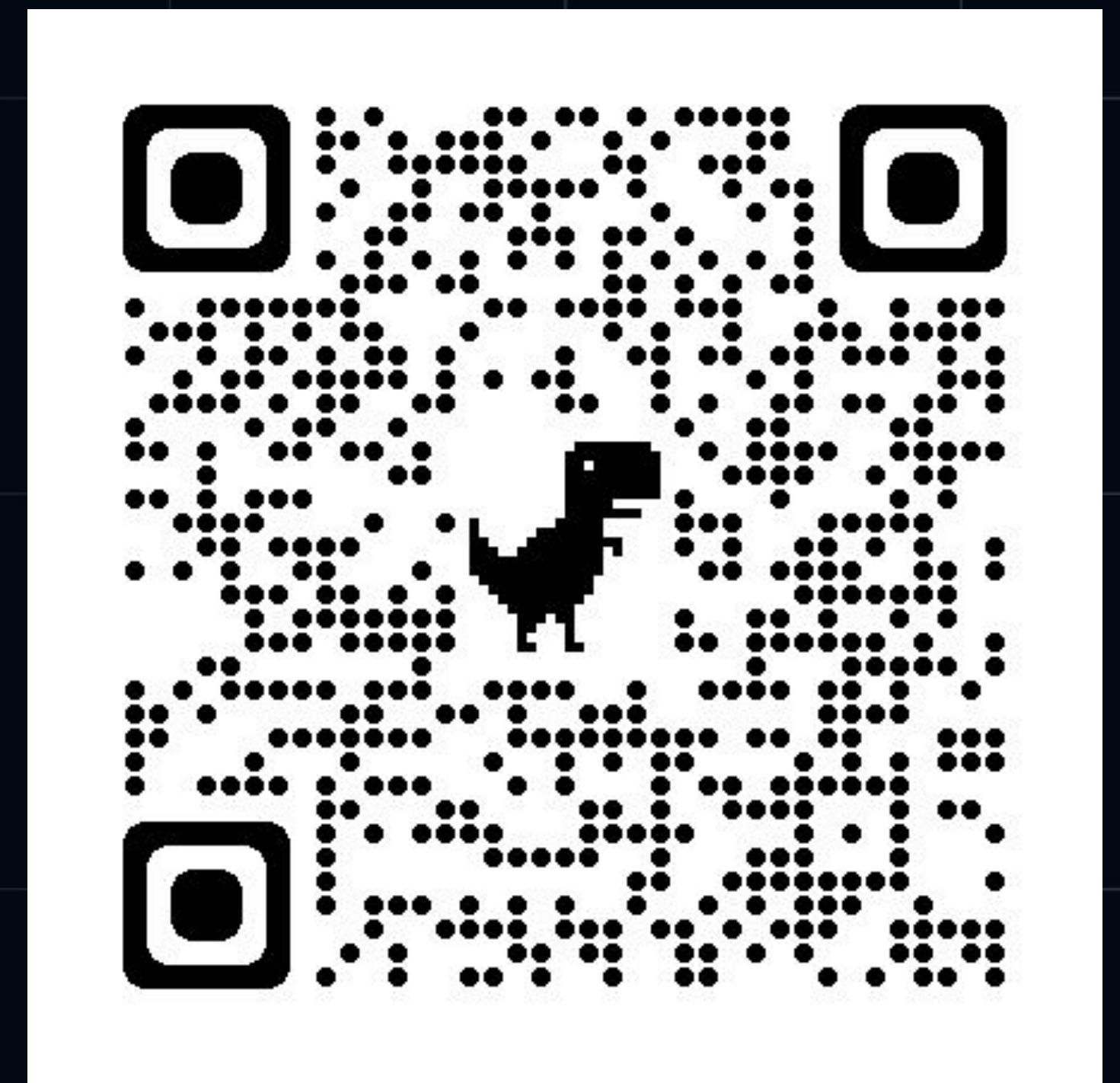


A composite image showing two side-by-side portraits of men. On the left is Dmitry Mahnev, a man with light brown hair and a full, reddish-brown beard, wearing a grey t-shirt. On the right is Artem Kobzарь, a man with short, light-colored hair and a well-groomed, dark brown beard, wearing a black t-shirt. Both men are looking directly at the camera.



thank you





to be continued ...

next talk announce

# Mnemonica Project



Viktor  
Vershanskiy

wentout

HOLY  
JS!...