A Wild TypeScript Safari

Daria Caraway

CascadiaJS 2021

I'm Your Safari Guide

DARIA CARAWAY

Senior UI Engineer @ Netflix

Lover of TypeScript since 2014

@dariacaraway

I'm Your Safari Guide

DARIA CARAWAY

Senior UI Engineer @ Netflix

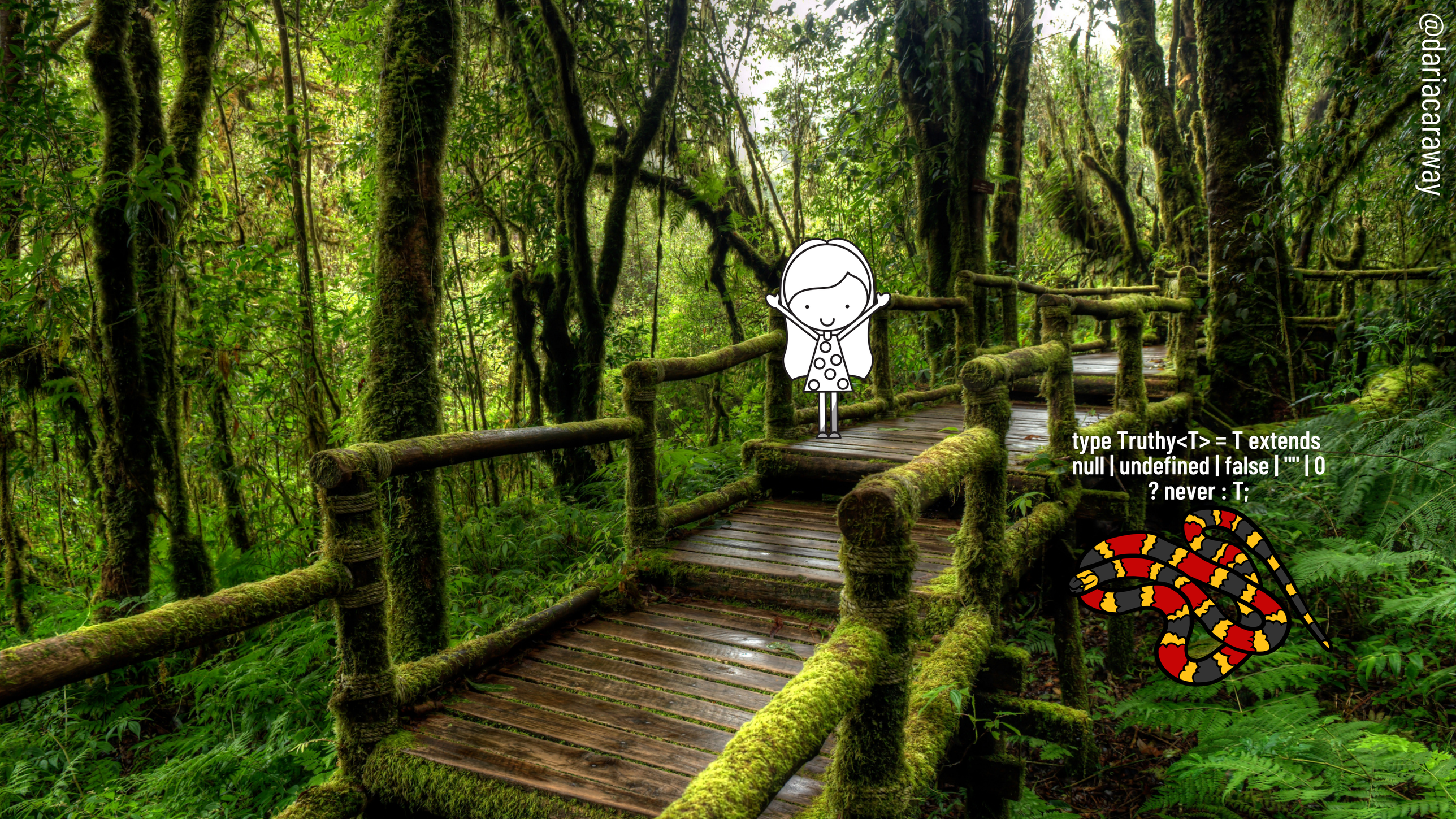
Lover of TypeScript since 2014

I do not know anything about jungle animals

@dariacaraway


@dariacaraway


boolean
string
number
@dariacaraway


type Truthy<T> = T extends
null | undefined | false | "" | 0
? never : T;
@dariacaraway

```
type Omit<T, K
extends keyof T> =
Pick<T, Exclude<keyof
T, K>>
```


type Foo<T> =
T extends { a: infer U;
b: infer U }
? U : never;
@dariacaraway

```
type ValidKeys<V> = {
 [K in keyof V]-?: (Exclude<V[K],
 null>) extends
 { a: U; b: U } ? IsTruthy<U>
 extends true ? K : never : never
 }[keyof V];
```


@dariacaraway


@dariacaraway

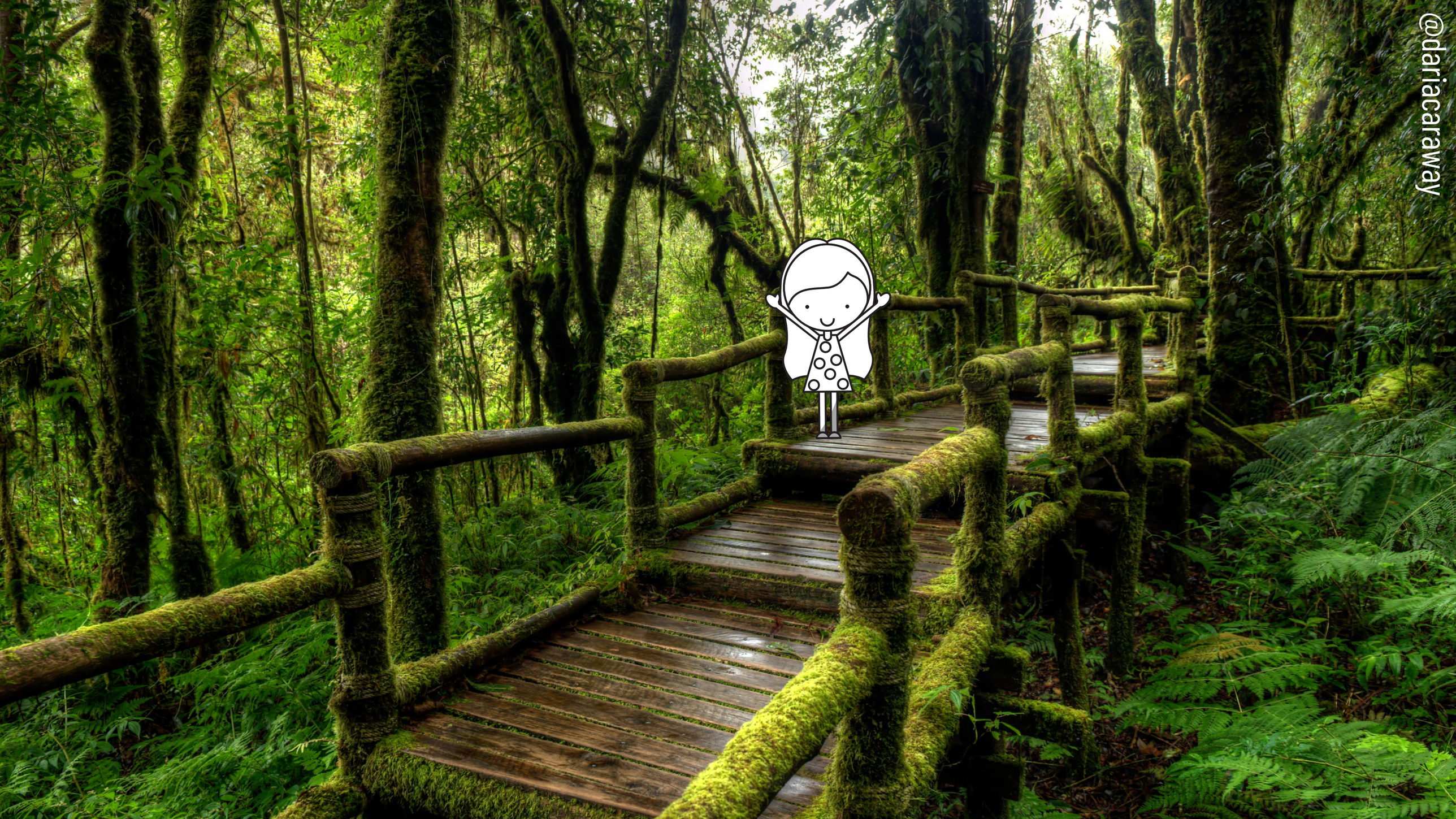

@dariacaraway

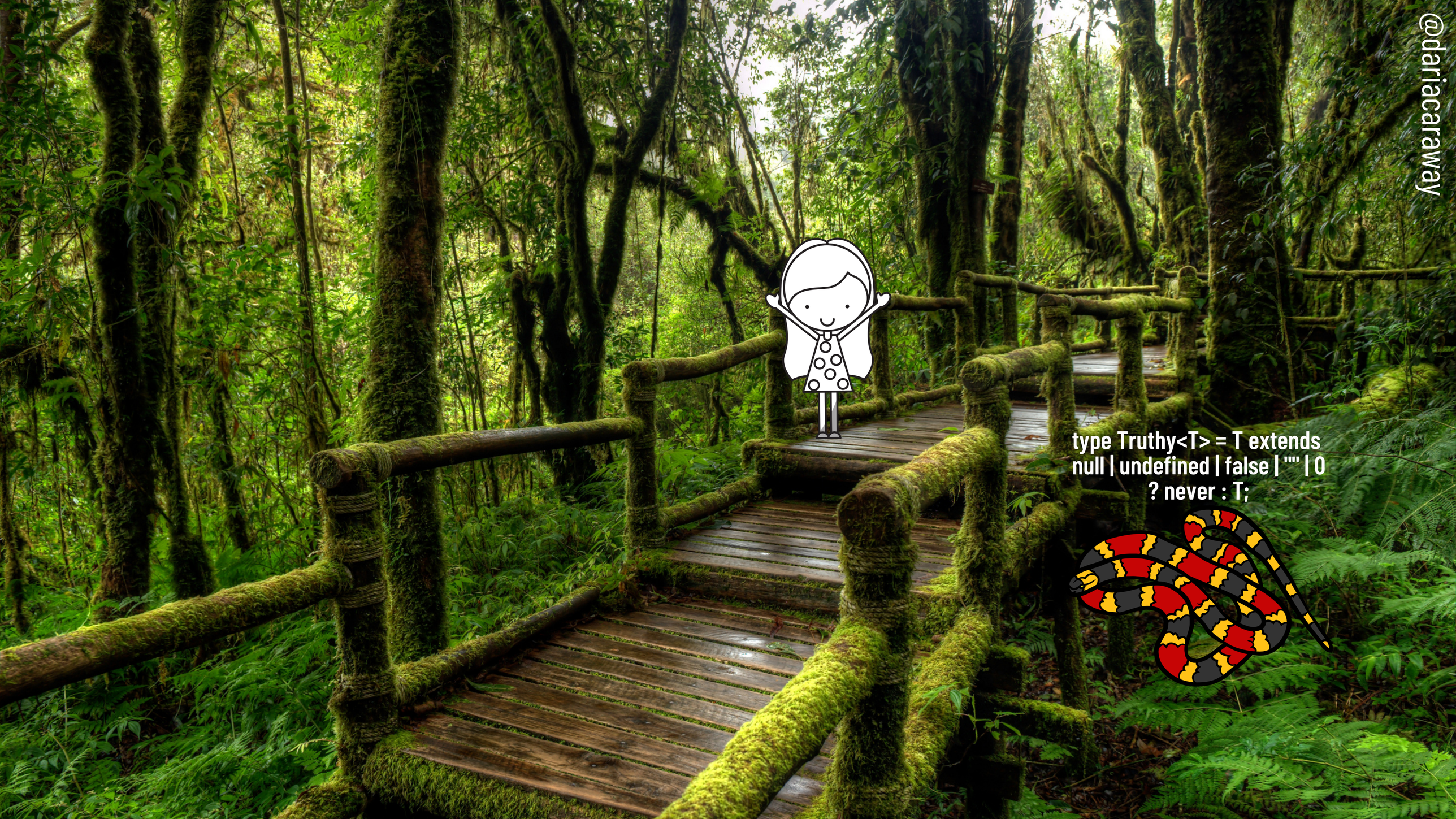
Good News Jungle Explorers!

TypeScript types get easier to identify and demystify with practice....

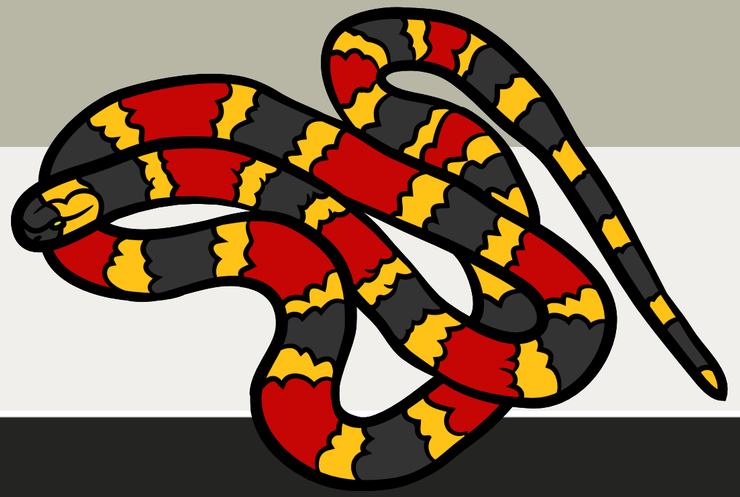
... let's
break down
some types

... let's
break down
some types


@dariacaraway

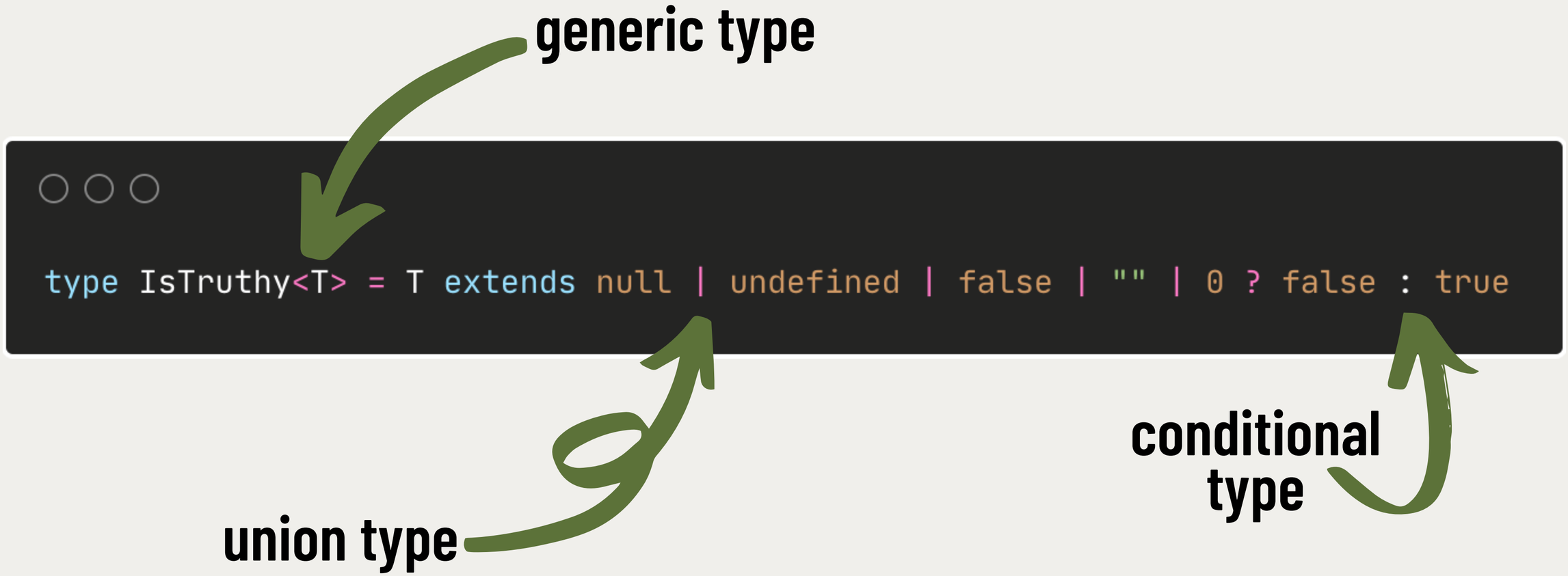

type Truthy<T> = T extends
null | undefined | false | "" | 0
? never : T;
@dariacaraway

The Coral Snake



```
type IsTruthy<T> = T extends null | undefined | false | "" | 0 ? false : true
```

The Coral Snake



The Coral Snake

```
type IsTruthy<T> =

  T extends null | undefined | false | "" | 0

    ? false

    : true
```

Generic Type

A stand-in representation for a type that is provided by the input or consumer.


```
type IsTruthy<T> =

  T extends null | undefined | false | "" | 0

    ? false

    : true
```

Type T represents a type that the consumer will provide. All Ts are enforced to be the same type. Generics are often notated as a single letter, but do not have to be.

Generic Type

A stand-in representation for a type that is provided by the input or consumer.

```
type IsTruthy<TypePassedIn> =

  TypePassedIn extends null | undefined | false | "" | 0

    ? false

    : true
```

Union Type

A value that can be one of many types. A or B or C

```
null | undefined | false | "" | 0
```

This union represents a value that can be either null, undefined, false, "", or 0

Union Type

A value that can be one of many types. A or B or C

```
type IsTruthy<TypePassedIn> =

  TypePassedIn extends null or undefined or false or "" or 0

    ? false

    : true
```

Conditional Type

Much like JavaScript conditional expression, conditional types allow for an if-then assignment

```
type IsTruthy<TypePassedIn> =

  TypePassedIn extends null or undefined or false or "" or 0

    ? false

    : true
```

The conditional type checks if the extension is possible or not and assigns values accordingly.

The Coral Snake


```
type IsTruthy<TypePassedIn> =

  if: TypePassedIn extends null or undefined or false or "" or 0

    then: the value is false

    else: the value is true
```

The Coral Snake


```
type IsTruthy<TypePassedIn> =

  if: TypePassedIn extends null or undefined or false or "" or 0

    then: the value is false

    else: the value is true
```

The Coral Snake


```
type IsTruthy<TypePassedIn> =

  if: TypePassedIn extends null or undefined or false or "" or 0

    then: the value is false

    else: the value is true
```

The Coral Snake


```
type IsTruthy<TypePassedIn> =

  if: TypePassedIn extends null or undefined or false or "" or 0

    then: the value is false

    else: the value is true
```

The Coral Snake


```
type IsTruthy<TypePassedIn> =

  if: TypePassedIn extends null or undefined or false or "" or 0

    then: the value is false

    else: the value is true
```

type IsTruthy<'hi'> = true
type IsTruthy<[ ]> = true
type IsTruthy<null> = false

just a cute harmless milk snake....

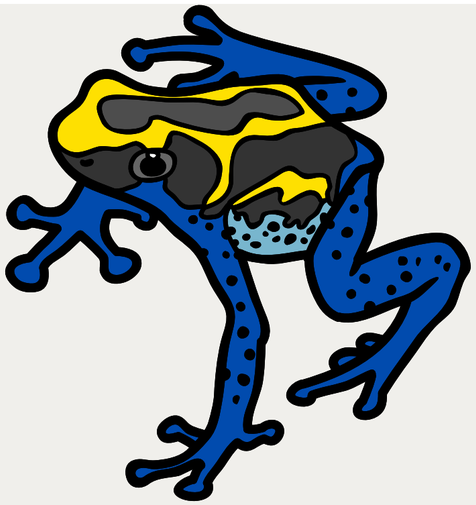
a coral snake copy-cat


@dariacaraway

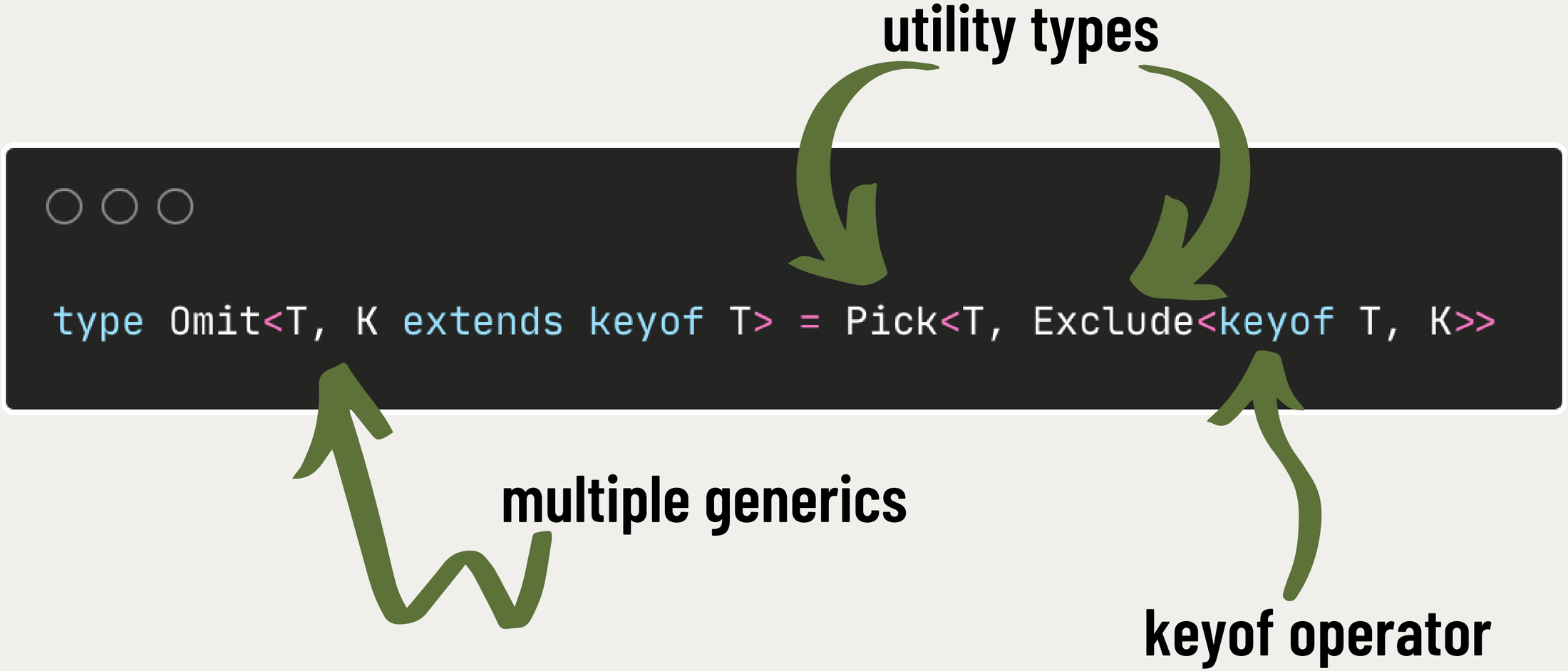
type Omit<T, K extends keyof T> = Pick<T, Exclude<keyof T, K>>

The Poisonous Dart Frog

```
type Omit<T, K extends keyof T> = Pick<T, Exclude<keyof T, K>>
```



The Poisonous Dart Frog

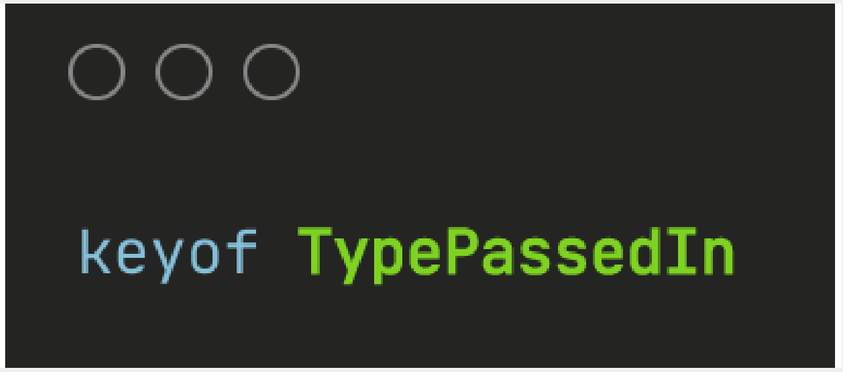


The Poisonous Dart Frog


```
type Omit<TypePassedIn, KPassedIn extends keyof TypePassedIn> =

  Pick<TypePassedIn, Exclude<keyof TypePassedIn, KPassedIn>>
```

keyof

A type operator that crates a union type from the keys of the provided type



Returns a union of keys
from the TypePassedIn

keyof

A type operator that crates a union type from the keys of the provided type

```
keyof TypePassedIn
```

Returns a union of keys
from the TypePassedIn

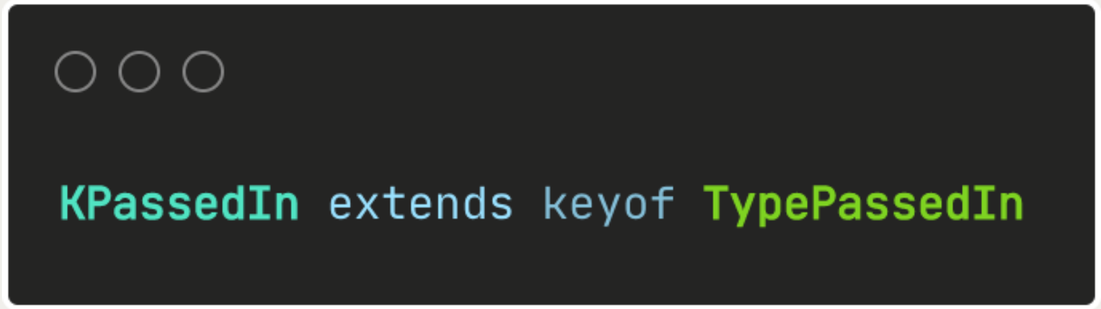
```
type ExampleType = {
    a: string
    b: number
    c: string[ ]
}
```

keyof ExampleType

"a" | "b" | "c"

keyof

A type operator that crates a union type from the keys of the provided type



Paired with extension, KPassedIn in must be a subset of keys from TypePassedIn, otherwise the extend is false

keyof

A type operator that crates a union type from the keys of the provided type

```
type Omit<TypePassedIn, KeysOfTypePassedInToOmit> =

  Pick<TypePassedIn, Exclude<union of TypePassedIn keys, KeysOfTypePassedInToOmit>>
```

Utility Types

Globally available types that represent common type transformations provided by TypeScript


```
Pick<TypePassedIn, Exclude<union of TypePassedIn keys, KeysOfTypePassedInToOmit>>
```

Pick returns a new type from the provided type with only the keys you give it

Exclude returns the provided type minus any union members that match the union to exclude

Exclude

Returns the provided type minus any union members that match the union to exclude

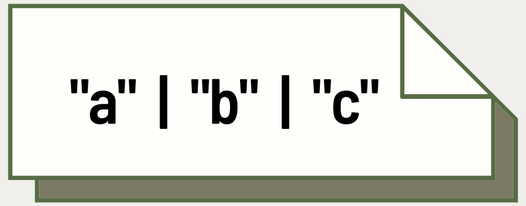
```
Exclude<union of TypePassedIn keys, KeysOfTypePassedInToOmit>
```

This represents a union type of keys from TypePassedIn in minus the keys to omit

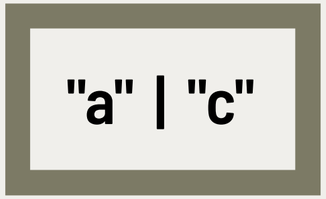
Exclude

Returns the provided type minus any union members that match the union to exclude

```
Exclude KeysOfTypePassedInToOmit from union of TypePassedIn keys
```



Exclude "b"



This represents a union type of keys from TypePassedIn in minus the keys to omit

Pick

Pick returns a new type from the provided type with only the keys you give it

```
Pick<TypePassedIn, Exclude KeysOfTypePassedInToOmit from union of TypePassedIn keys>
```

This represents a union type of keys from TypePassedIn in minus the keys to omit

Pick

Pick returns a new type from the provided type with only the keys you give it

```
Pick everthing in (Exclude KeysOfTypePassedInToOmit from union of TypePassedIn keys) from TypePassedIn
```

```
type ExampleType = {
    a: string
    b: number
    c: string[ ]
}
```

Pick "a" | "c"

```
type ExampleType = {
    a: string
    c: string[ ]
}
```

The Poisonous Dart Frog


```
type Omit<TypePassedIn, KeysOfTypePassedInToOmit> =

  Pick everthing in (Exclude KeysOfTypePassedInToOmit from union of TypePassedIn keys) from TypePassedIn
```

```
type ExampleType = {
    a: string
    b: number
    c: string[ ]
}
```

Omit "b"

```
type ExampleType = {
    a: string
    c: string[ ]
}
```

just a colorful mimic poison frog

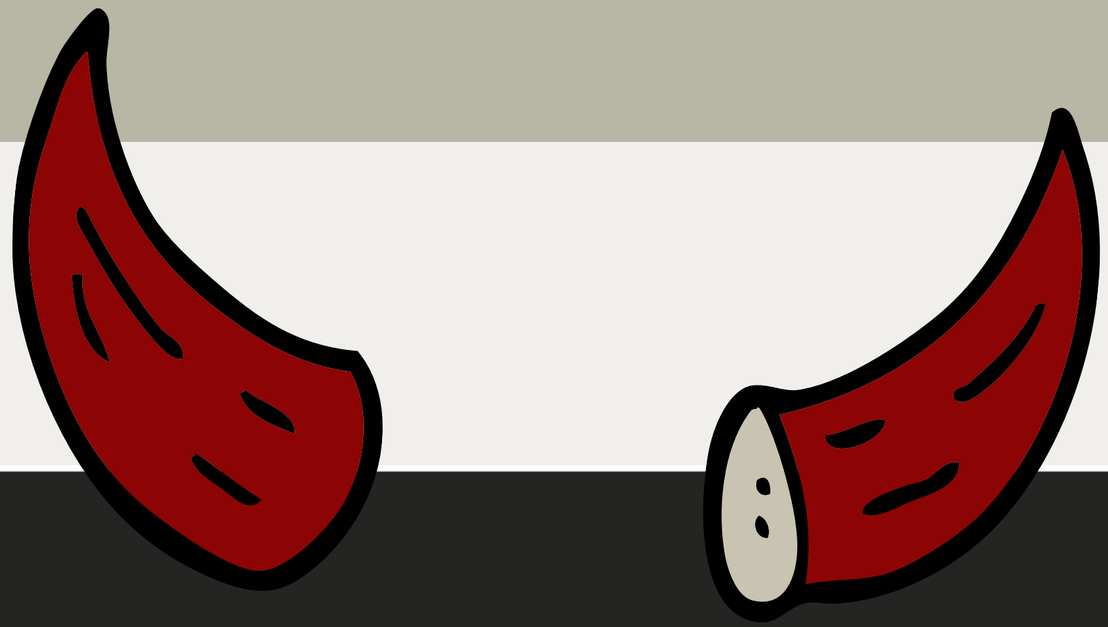
just a colorful mimic poison frog

will eat all your poisonous bugs

@dariacaraway

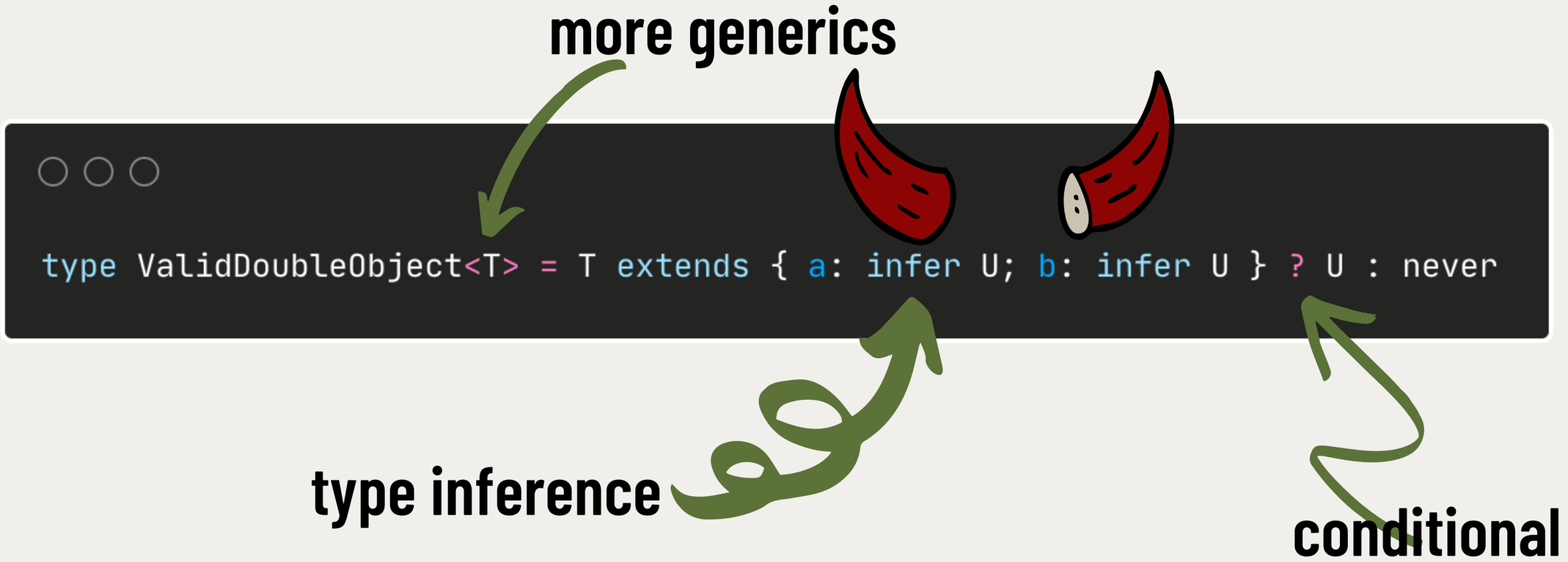

type Foo<T> =
T extends { a: infer U;
b: infer U }
? U : never;
@dariacaraway

The Horns



```
type ValidDoubleObject<T> = T extends { a: infer U; b: infer U } ? U : never
```

The Horns



The Horns

```
type ValidDoubleObject<TypePassedIn> =

  if: TypePassedIn extends { a: infer U; b: infer U }

    then: the value is U

    else: the value is never
```

The Horns

```
type ValidDoubleObject<TypePassedIn> =

  if: TypePassedIn extends { a: infer U; b: infer U }

    then: the value is U

    else: the value is never
```

The Horns

```
type ValidDoubleObject<TypePassedIn> =

  if: TypePassedIn extends { a: infer U; b: infer U }

    then: the value is U

    else: the value is never
```

The Horns

```
type ValidDoubleObject<TypePassedIn> =

  if: TypePassedIn extends { a: infer U; b: infer U }

    then: the value is U

    else: the value is never
```

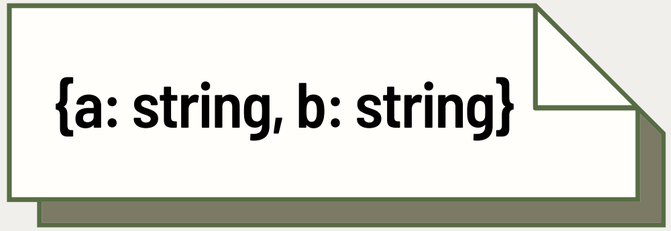
The Horns

```
type ValidDoubleObject<TypePassedIn> =

  if: TypePassedIn extends { a: infer U; b: infer U }

    then: the value is U

    else: the value is never
```

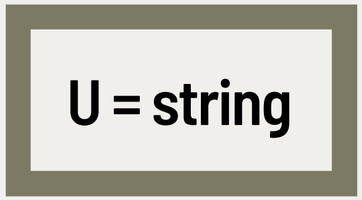
Type Inference

Based on surrounding context, TypeScript will decide what the type is - only if it can


```
if: TypePassedIn extends { a: infer U; b: infer U }
```



{ a: infer U; b: infer U }



Generic type U will be inferred based on the type that is passed in if possible.

The Horns


```
type ValidDoubleObject<TypePassedIn> =

  if: TypePassedIn extends { a: InferredType; b: InferredType }

    then: the value is the InferedType

    else: the value is never
```

The Horns

```
type ValidDoubleObject<TypePassedIn> =

  if: TypePassedIn extends { a: InferredType; b: InferredType }

    then: the value is the InferedType

    else: the value is never
```

The Horns


```
type ValidDoubleObject<TypePassedIn> =

  if: TypePassedIn extends { a: InferredType; b: InferredType }

    then: the value is the InferedType

    else: the value is never
```

The Horns

```
type ValidDoubleObject<TypePassedIn> =

  if: TypePassedIn extends { a: InferredType; b: InferredType }

    then: the value is the InferedType

    else: the value is never
```

The Horns

```
type ValidDoubleObject<TypePassedIn> =

  if: TypePassedIn extends { a: InferredType; b: InferredType }

    then: the value is the InferedType

    else: the value is never
```

ValidDoubleObject<string> = never

ValidDoubleObject <{a: string, b: string}> = string

just a beautiful rhinoceros hornbill

@dariacaraway

```
type ValidKeys<V> = {
 [K in keyof V]-?: (Exclude<V[K],
 null>) extends
 { a: U; b: U } ? IsTruthy<U>
 extends true ? K : never : never
 }[keyof V];
```

The Jaguar


```ts
type ValidKeys<V> = {
  [K in keyof V]-?: (Exclude<V[K], null>) extends
  { a: infer U; b: infer U } ? IsTruthy<U> extends true ? K : never : never
}[keyof V]
```

The Jaguar

The Jaguar

mapped type

utility type

mapped type modifier

generics all over

```
type Val  Keys<V    {
  [K in keyof V]-?: (Exclude<V[K], null>) extends
  { a: infer U; b: infer U } ? IsTruthy<U> extends true ? K : never : never
}[keyof V]
```

type inference

conditional type

The Jaguar


```
type ValidKeys<TypePassedIn> = {
  [K in keyof TypePassedIn]-?:

  if: (Exclude null from TypePassedIn[K]) extends { a: InferredType; b: InferredType }
    then if: IsTruthy<InferredType> extends true
      then: the value is K
      else: the value is never
    else: the value is never

}[keyof TypePassedIn]
```

Exclude utility

IsTruthy type

DoubleObject inferred type

The Jaguar


```
type ValidKeys<TypePassedIn> = {
  [K in keyof TypePassedIn]-?:

  if: (Exclude null from TypePassedIn[K]) extends { a: InferredType; b: InferredType }
    then if: IsTruthy<InferredType> extends true
      then: the value is K
      else: the value is never
    else: the value is never

}[keyof TypePassedIn]
```

The Jaguar


```
type ValidKeys<TypePassedIn> = {
  [K in keyof TypePassedIn]-?:

  if: (Exclude null from TypePassedIn[K]) extends { a: InferredType; b: InferredType }
    then if: IsTruthy<InferredType> extends true
      then: the value is K
      else: the value is never
    else: the value is never

}[keyof TypePassedIn]
```

Mapped Type

Creates a new type with all of the keys in the key iterator

```
{ [K in keyof TypePassedIn]-?: value }
```

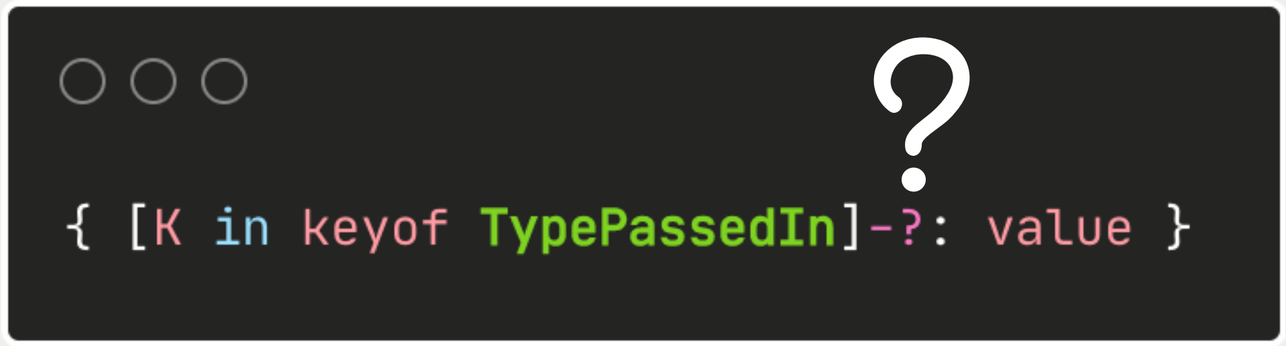
The output of this type will have a mapped value for each key of the type passed in

{[K in keyof ExampleType] : value }

```
{
  a: value
  b: value
  c: value
}
```

Mapped Type

Creates a new type with all of the keys in the key iterator



Mapped Type Modifiers

A variety of operators that can be applied over a mapped type

```
{ [K in keyof TypePassedIn]-?: value }
```

Each key of the passed in type will be made required if it was previously optional. The ? and undefined values will be removed from the type.

```
ExampleType = {
  a?: string
  b: string
}
```

{[K in keyof ExampleType]-? : value }

{ a: value, b: value }

Mapped Type Modifiers

A variety of operators that can be applied over a mapped type


```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassed as required]:

  if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
    then if: IsTruthy<InferredType> extends true
      then: the value is KeyOfTypePassed
      else: the value is never
    else: the value is never
}[keyof TypePassedIn]
```

Mapped Type Modifiers

A variety of operators that can be applied over a mapped type


```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassed as required]:

  if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
    then if: IsTruthy<InferredType> extends true
      then: the value is KeyOfTypePassed
      else: the value is never
    else: the value is never
}[keyof TypePassedIn]
```

TypePassedIn[KeyOfTypePassed] = ValueOfTypePassedIn

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassed as required]:

  if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
    then if: IsTruthy<InferredType> extends true
      then: the value is KeyOfTypePassed
      else: the value is never
    else: the value is never
}[keyof TypePassedIn]
```

```
type ExampleType = {
  valid1: {a: string, b: string} | null
}
```

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassed as required]:

  if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
    then if: IsTruthy<InferredType> extends true
      then: the value is KeyOfTypePassed
      else: the value is never
    else: the value is never
}[keyof TypePassedIn]
```

```
type ExampleType = {
  valid1: {a: string, b: string} | null
}
```

mapped type...

```
{
  valid1: to be determined
}
```

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassed as required]:

  if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
    then if: IsTruthy<InferredType> extends true
      then: the value is KeyOfTypePassed
      else: the value is never
    else: the value is never
}[keyof TypePassedIn]
```

ValueOfTypePassedIn

{a: string, b: string} | null

Exclude null

{a: string, b: string}

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassed as required]:

  if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
    then if: IsTruthy<InferredType> extends true
      then: the value is KeyOfTypePassed
      else: the value is never
    else: the value is never
}[keyof TypePassedIn]
```

Exclude null from ValueOfTypePassedIn

{a: string, b: string}

extends { a: InferredType; b: InferredType }

true

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassed as required]:

  if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
    then if: IsTruthy<InferredType> extends true
      then: the value is KeyOfTypePassed
      else: the value is never
    else: the value is never
}[keyof TypePassedIn]
```

InferredType

isTruthy?

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassed as required]:

  if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
    then if: IsTruthy<InferredType> extends true
      then: the value is KeyOfTypePassed
      else: the value is never
    else: the value is never
}[keyof TypePassedIn]
```

```
{
  valid1: to be determined
}
```

maps to...

```
{
  valid1: "valid1"
}
```

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassed as required]:

  if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
    then if: IsTruthy<InferredType> extends true
      then: the value is KeyOfTypePassed
      else: the value is never
    else: the value is never
}[keyof TypePassedIn]
```

```
{
  valid1: "valid1"
}
```

keyof TypePassedIn

"valid1"

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassed as required]:

  if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
    then if: IsTruthy<InferredType> extends true
      then: the value is KeyOfTypePassed
      else: the value is never
    else: the value is never
}[keyof TypePassedIn]
```

```
type ExampleType = {
  valid1: {a: string, b: string} | null
}
```

ValidKeys<ExampleType>

"valid1"

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassed as required]:

  if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
    then if: IsTruthy<InferredType> extends true
      then: the value is KeyOfTypePassed
      else: the value is never
    else: the value is never
}[keyof TypePassedIn]
```

```
type ExampleType = {
    valid1: {a: string, b: string} | null
    valid2: {a: number, b: number}
    invalid1?: {a: string, b: string}
    invalid2: {notValid: string}
}
```

ValidKeys<ExampleType>

"valid1" | "valid2"

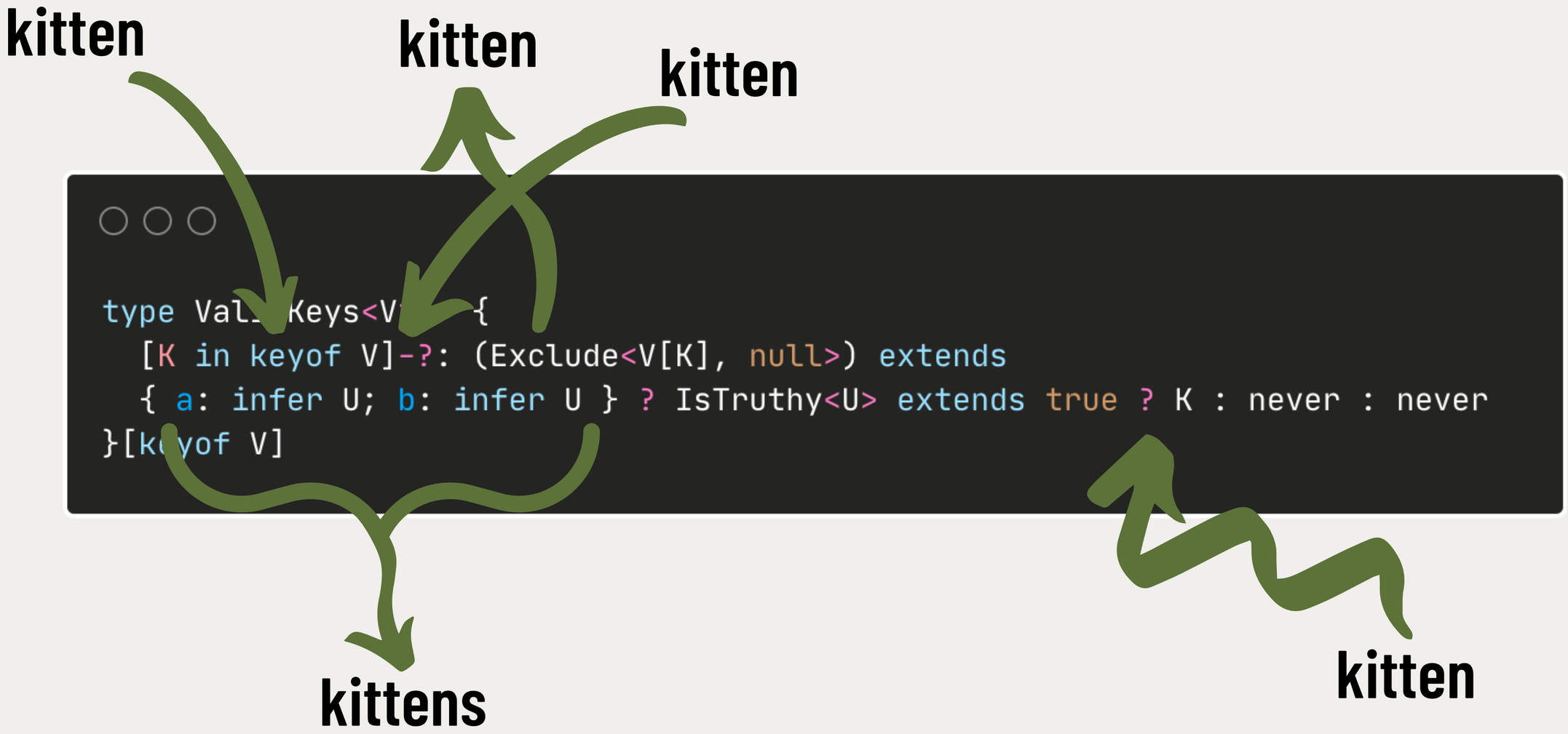
@dariacaraway

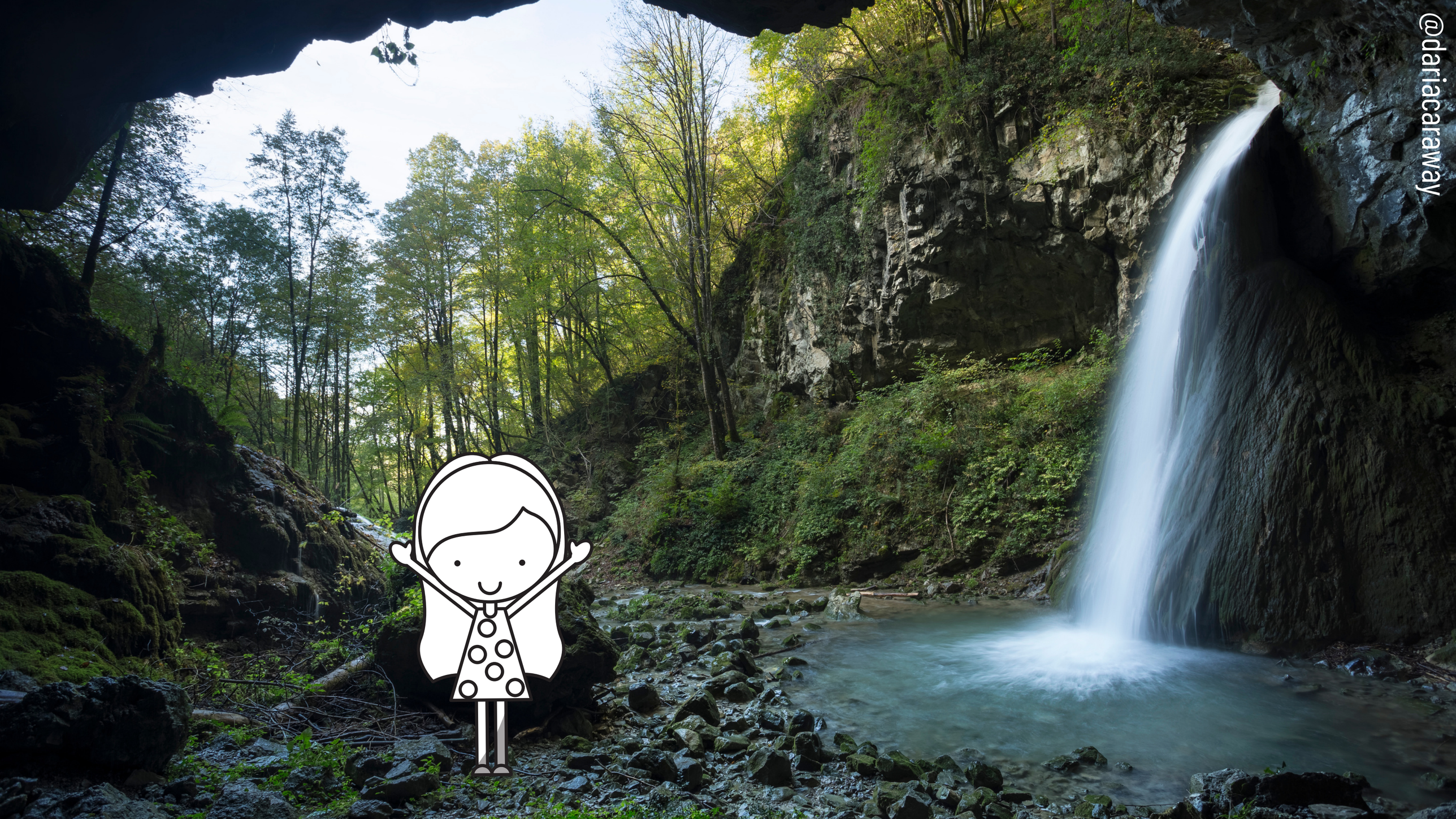
just an adorable group of kittens

The Kittens

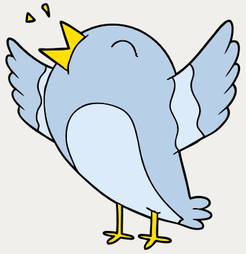

```ts
type ValidKeys<V> = {
  [K in keyof V]-?: (Exclude<V[K], null>) extends
  { a: infer U; b: infer U } ? IsTruthy<U> extends true ? K : never : never
}[keyof V]
```

The Kittens



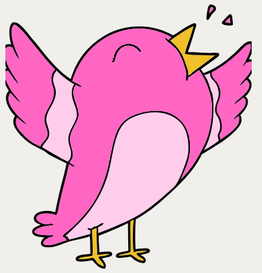

@dariacaraway


@dariacaraway

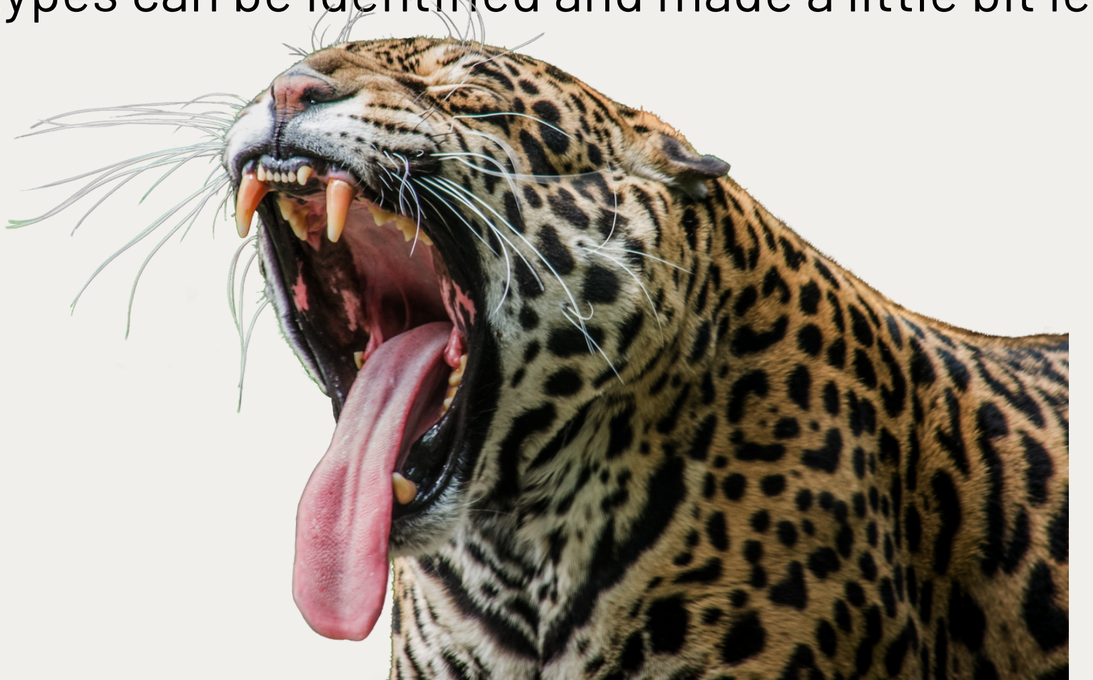


@dariacaraway

Working with typescript can sometimes feel like you're on a jungle safari....



But with practice, all types can be identified and made a little bit less wild



THANK YOU!

@DariaCaraway