Decoding Perplexing TypeScript

DARIA CARAWAY
@dariacaraway

TSConf 2021

TS

DARIA CARAWAY

Senior UI Engineer
@ NETFLIX

Lover of TypeScript
since 2014

Why I <3 TypeScript

Faster Bug Catching

Compile-time type-checking can catch bugs before your application even runs

Context Sharing

TS is a fantastic communication tool to add another level of context directly in the code

External Library DX

Auto-complete and strict type checking makes uptaking unfamiliar code more straighforward

@dariacaraway

Why I <3 TypeScript

... and many more

But...

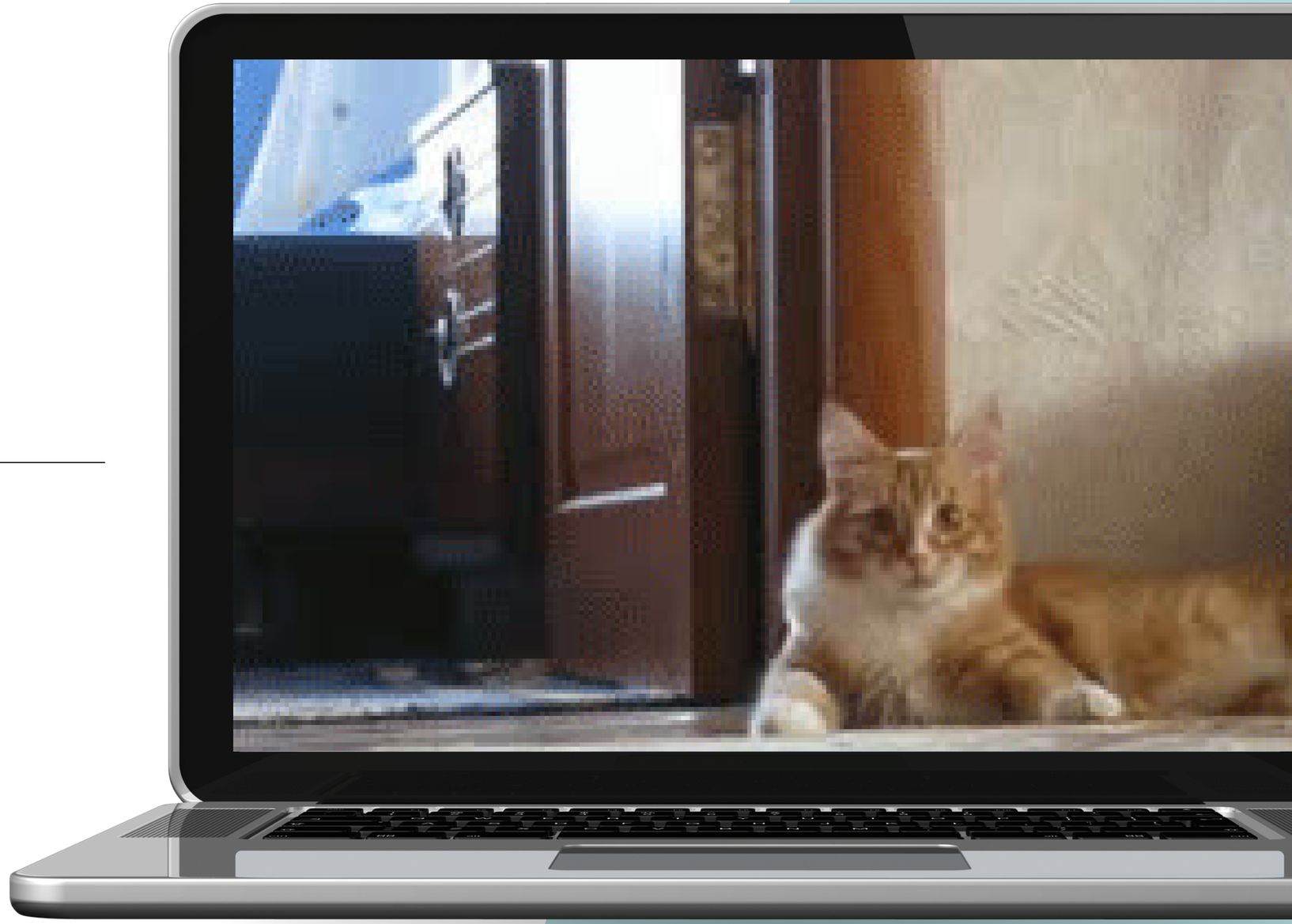
Complex code and use cases can turn into complex types.

Wonderful, helpful, generic, and protective types can sometimes get complicated...

But...

Complex code and use cases can turn into complex types.

Wonderful, helpful, generic, and protective types can sometimes get complicated...



"TypeScript Nope"

When you go to look at a type and it is pretty long and scary so you get nervous and close the file .

types of nopes...

types of nopes...

lots of unfamiliar characters

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

types of nopes...

letters all over

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

```
false: true
```

types of nopes...

what are these words

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

types of nopes...

long scary ternary

```
export type ExtractRouteOptionalParam<T extends string, U =
string | number | boolean> = T extends `${infer Param}?`
    ? { [k in Param]?: U }
    : T extends `${infer Param}*`
    ? { [k in Param]?: U }
    : T extends `${infer Param}+`
    ? { [k in Param]: U }
    : { [k in T]: U };
```

types of nopes...

just nope

```
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]
```

```
? { [k in Param
: { [k in T]: U };
```

types of nopes...

omg they disabled the max line length

```
 */
// tslint:disable:max-line-length
export function pipe<T1>(fn0: () => T1): () => T1;
export function pipe<V0, T1>(fn0: (x0: V0) => T1): (x0: V0) => T1;
export function pipe<V0, V1, T1>(fn0: (x0: V0, x1: V1) => T1): (x0: V0, x1: V1) => T1;
export function pipe<V0, V1, V2, T1>(fn0: (x0: V0, x1: V1, x2: V2) => T1): (x0: V0, x1: V1, x2: V2) => T1;

export function pipe<T1, T2>(fn0: () => T1, fn1: (x: T1) => T2): () => T2;
export function pipe<V0, T1, T2>(fn0: (x0: V0) => T1, fn1: (x: T1) => T2): (x0: V0) => T2;
export function pipe<V0, V1, T1, T2>(fn0: (x0: V0, x1: V1) => T1, fn1: (x: T1) => T2): (x0: V0, x1: V1) => T2;
export function pipe<V0, V1, V2, T1, T2>(fn0: (x0: V0, x1: V1, x2: V2) => T1, fn1: (x: T1) => T2): (x0: V0, x1: V1, x2: V2) => T2;

export function pipe<T1, T2, T3>(fn0: () => T1, fn1: (x: T1) => T2, fn2: (x: T2) => T3): () => T3;
export function pipe<V0, T1, T2, T3>(fn0: (x: V0) => T1, fn1: (x: T1) => T2, fn2: (x: T2) => T3): (x: V0) => T3;
export function pipe<V0, V1, T1, T2, T3>(fn0: (x0: V0, x1: V1) => T1, fn1: (x: T1) => T2, fn2: (x: T2) => T3): (x0: V0, x1: V1) => T3;
export function pipe<V0, V1, V2, T1, T2, T3>(fn0: (x0: V0, x1: V1, x2: V2) => T1, fn1: (x: T1) => T2, fn2: (x: T2) => T3): (x0: V0, x1: V1, x2: V2) => T3;

export function pipe<T1, T2, T3, T4>(fn0: () => T1, fn1: (x: T1) => T2, fn2: (x: T2) => T3, fn3: (x: T3) => T4): () => T4;
export function pipe<V0, T1, T2, T3, T4>(fn0: (x: V0) => T1, fn1: (x: T1) => T2, fn2: (x: T2) => T3, fn3: (x: T3) => T4): (x: V0) => T4;
export function pipe<V0, V1, T1, T2, T3, T4>(fn0: (x0: V0, x1: V1) => T1, fn1: (x: T1) => T2, fn2: (x: T2) => T3, fn3: (x: T3) => T4): (x0: V0, x1: V1) => T4;
export function pipe<V0, V1, V2, T1, T2, T3, T4>(fn0: (x0: V0, x1: V1, x2: V2) => T1, fn1: (x: T1) => T2, fn2: (x: T2) => T3, fn3: (x: T3) => T4): (x0: V0, x1:

export function pipe<T1, T2, T3, T4, T5>(fn0: () => T1, fn1: (x: T1) => T2, fn2: (x: T2) => T3, fn3: (x: T3) => T4, fn4: (x: T4) => T5): () => T5;
export function pipe<V0, T1, T2, T3, T4, T5>(fn0: (x: V0) => T1, fn1: (x: T1) => T2, fn2: (x: T2) => T3, fn3: (x: T3) => T4, fn4: (x: T4) => T5): (x: V0) => T5;
export function pipe<V0, V1, T1, T2, T3, T4, T5>(fn0: (x0: V0, x1: V1) => T1, fn1: (x: T1) => T2, fn2: (x: T2) => T3, fn3: (x: T3) => T4, fn4: (x: T4) => T5): (
export function pipe<V0, V1, V2, T1, T2, T3, T4, T5>(fn0: (x0: V0, x1: V1, x2: V2) => T1, fn1: (x: T1) => T2, fn2: (x: T2) => T3, fn3: (x: T3) => T4, fn4: (x: T

export function pipe<T1, T2, T3, T4, T5, T6>(fn0: () => T1, fn1: (x: T1) => T2, fn2: (x: T2) => T3, fn3: (x: T3) => T4, fn4: (x: T4) => T5, fn5: (x: T5) => T6):
export function pipe<V0, T1, T2, T3, T4, T5, T6>(fn0: (x: V0) => T1, fn1: (x: T1) => T2, fn2: (x: T2) => T3, fn3: (x: T3) => T4, fn4: (x: T4) => T5, fn5: (x: T5
export function pipe<V0, V1, T1, T2, T3, T4, T5, T6>(fn0: (x0: V0, x1: V1) => T1, fn1: (x: T1) => T2, fn2: (x: T2) => T3, fn3: (x: T3) => T4, fn4: (x: T4) => T5
export function pipe<V0, V1, V2, T1, T2, T3, T4, T5, T6>(fn0: (x0: V0, x1: V1, x2: V2) => T1, fn1: (x: T1) => T2, fn2: (x: T2) => T3, fn3: (x: T3) => T4, fn4: (
```

GOOD NEWS

TypeScript gets easier to breakdown and understand with practice.

... let's break down some types

unfamiliar characters **breakdown**

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

unfamiliar characters **breakdown**

generic type

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

union type

conditional type

unfamiliar characters **breakdown**

```
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


```
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.

unfamiliar characters **breakdown**

```
type IsTruthy<TypePassedIn> =

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

    then: the value is false

    else: the value is true
```

unfamiliar characters **breakdown**

```
type IsTruthy<TypePassedIn> =

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

    then: the value is false

    else: the value is true
```

unfamiliar characters **breakdown**

```
type IsTruthy<TypePassedIn> =

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

    then: the value is false

    else: the value is true
```

unfamiliar characters **breakdown**

```
type IsTruthy<TypePassedIn> =

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

    then: the value is false

    else: the value is true
```

unfamiliar characters **breakdown**

```
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

~~unfamiliar characters **breakdown**~~

letters all over **breakdown**

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

letters all over **breakdown**

utility types

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

multiple generics

keyof operator

letters all over **breakdown**

```
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

```
keyof TypePassedIn
```

Returns a union of keys from the TypePassedIn

keyof

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

```
KPassedIn extends keyof TypePassedIn
```

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

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

"a" | "b" | "c"

Exclude "b"

"a" | "c"

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 new type with only the properties of TypePassedIn that are also in the return value of the exclude clause

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
```

Pick "a " | "c"

from:

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

=

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

letters all over **breakdown**

```
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
    b: string[ ]
}
```

~~letters all over **breakdown**~~

what are these words **breakdown**

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

what are these words **breakdown**

more generics

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

type inference

conditional type

what are these words **breakdown**

```
type ValidDoubleObject<TypePassedIn> =

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

    then: the value is U

    else: the value is never
```

what are these words **breakdown**

```
type ValidDoubleObject<TypePassedIn> =

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

    then: the value is U

    else: the value is never
```

what are these words **breakdown**

```
type ValidDoubleObject<TypePassedIn> =

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

    then: the value is U

    else: the value is never
```

?

what are these words **breakdown**

```
type ValidDoubleObject<TypePassedIn> =

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

    then: the value is U

    else: the value is never
```

what are these words **breakdown**

```
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 }
```

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

{a: string, b: string}

{ a: infer U; b: infer U }

U = string

what are these words **breakdown**

```
type ValidDoubleObject<TypePassedIn> =

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

    then: the value is the InferredType

    else: the value is never
```

what are these words **breakdown**

```
type ValidDoubleObject<TypePassedIn> =

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

    then: the value is the InferredType

    else: the value is never
```

what are these words **breakdown**

```
type ValidDoubleObject<TypePassedIn> =

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

    then: the value is the InferredType

    else: the value is never
```

what are these words **breakdown**

```
type ValidDoubleObject<TypePassedIn> =

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

    then: the value is the InferredType

    else: the value is never
```

what are these words **breakdown**

```
type ValidDoubleObject<TypePassedIn> =
    if: TypePassedIn extends { a: InferredType; b: InferredType }
    then: the value is the InferredType
    else: the value is never
```

ValidDoubleObject<string> = never

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

~~what are these words **breakdown**~~

just nope **breakdown**

```
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]
```

just nope **breakdown**

mapped type

utility type

mapped type modifier

generics all over

```
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]
```

type inference

conditional types

just nope **breakdown**

```
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 type

IsTruthy type

Double Object
Inferred type

just nope **breakdown**

```
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

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

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 Modifier

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 Modifier

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

just nope breakdown

```
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
}
```

just nope breakdown

```
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
}
```

just nope breakdown

```
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}

just nope breakdown

```
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

just nope breakdown

```
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

string

isTruthy?

true

just nope breakdown

```
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"
}
```

just nope breakdown

```
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]
```

keyof TypePassedIn

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

"valid1"

just nope breakdown

```
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"

just nope breakdown

```
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 nope **breakdown**~~

@dariacaraway

TypeScript can be Perplexing

But, with practice, all types can be broken down and decoded.

THANK YOU

DARIA CARAWAY
@dariacaraway

TS CONF