DECODING PERPLEXING TYPESCRIPT

jsconf budapest 2024

PLAYER 1

Senior Software Engineer
Neftlix

DARIA CARARWAY

lover of TypeScript since 2014

Colorado, USA

THE TYPESCRIPT LABYRINTH

GAME OVERVIEW

LEVEL 1	type IsTruthy
LEVEL 2	type Omit
LEVEL 3	type ValidDoubleObject
LEVEL 4	The Final Boss...

THE FINAL BOSS

```
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];
```

PLAYER INVENTORY

PRIMITIVE TYPES	*string, number, boolean ...*

LEVELING UP

TypeScript gets easier to understand and breakdown the more tools we unlock in our inventory.

ENTERING THE LABYRINTH...

LEVEL 1

START >

LEVEL 1

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

LEVEL 1

generics

unions

conditionals

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

LEVEL 1

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

LOADING

UNIONS

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

```
type UnionType = null | undefined | false | 0 | ""
```

UnionType represents a value that can be either null, undefined, false, or 0

UNIONS

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

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

GENERICS

A stand-in representation for a type that is provided as an input

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

Type T represents a type that
the consumer will provide.

GENERICS

ISOLATED EXAMPLE

A stand-in representation for a type that is provided as an input

```
type GeneicType<T> = T | Array<T>
```

GENERICS

ISOLATED EXAMPLE

A stand-in representation for a type that is provided as an input

```
type GeneicType<T> = T | Array<T>
```

```
type StringOrStringArray = GeneicType<string>
```

GENERICS

ISOLATED EXAMPLE

A stand-in representation for a type that is provided as an input

```
type GeneicType<T> = T | Array<T>
```

```
type StringOrStringArray = string | string[]
type StringOrStringArray = GeneicType<string>
```

GENERICS

A stand-in representation for a type that is provided as an input

```
type IsTruthy<TypePassedIn> = TypePassedIn extends null or undefined or false or 0 or ""
  ? false
  : true;
```

CONDITIONALS

Much like JavaScript conditional expression, conditional types allow for an if-then assignment based on a generic

```
type IsTruthy<TypePassedIn> =
  if: TypePassedIn extends null or undefined or false or 0 or ""
    then: the value is false
    else: the value is true;
```

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

LEVEL 1

```
type IsTruthy<TypePassedIn> =
  if: TypePassedIn extends null or undefined or false or 0 or ""
    then: the value is false
    else: the value is true;
```

LEVEL 1

```
type IsTruthy<TypePassedIn> =
  if: TypePassedIn extends null or undefined or false or 0 or ""
    then: the value is false
    else: the value is true;
```

```
type NotTruthy = IsTruthy<null>
```

LEVEL 1

null

```
type IsTruthy<TypePassedIn> =
```

null

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

    then: the value is false

    else: the value is true;
```

```
type NotTruthy = IsTruthy<null>
```

LEVEL 1

null

```
type IsTruthy<TypePassedIn> =
```

null

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

    then: the value is false

    else: the value is true;
```

```
type NotTruthy = false
```

```
type NotTruthy = IsTruthy<null>
```

LEVEL 1

```
type IsTruthy<TypePassedIn> =
  if: TypePassedIn extends null or undefined or false or 0 or ""
    then: the value is false
    else: the value is true;
```

```
type NotTruthy = false
type NotTruthy = IsTruthy<null>
```

```
type PassesTruthy = true
type PassesTruthy = IsTruthy<string>
```

PLAYER INVENTORY

EVERYDAY TYPES	*string, number, boolean, union*
TYPE MANIPULATION	generics, conditionals

GAME OVERVIEW

LEVEL 1	type IsTruthy ★★★
LEVEL 2	type Omit
LEVEL 3	type ValidDoubleObject
LEVEL 4	The Final Boss...

LEVEL 2

START >

LEVEL 2

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

- ✓ generics
- utility types
- keyof operator

LEVEL 2

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

LEVEL 2

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

LOADING

KEYOF OPERATOR

Creates a new union type from the keys of the provided type

```
keyof TypePassedIn
```

KEYOF OPERATOR

ISOLATED EXAMPLE

Creates a new union type from the keys of the provided type

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

KEYOF OPERATOR

ISOLATED EXAMPLE

Creates a new union type from the keys of the provided type

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

```
keyof ExampleObjectType
```

=

```
"a" | "b" | "c"
```

*string literals

KEYOF OPERATOR

Creates a new union type from the keys of the provided type

```
K extends keyof TypePassedIn
```

Paired with extends, K must be a subset of keys from TypePassedIn, otherwise the extend is invalid

KEYOF OPERATOR

Creates a new union type from the keys of the provided type

```
type Omit<TypePassedIn, KeysOfTypePassedIn> =
  Pick<TypePassedIn, Exclude<TypePassedIn key union, KeysOfTypePassedIn>>
```

UTILITY TYPES

Types that represent common type transformations provided by TypeScript

EXCLUDE

Returns the provided union, minus any union items to exclude

```
Exclude<keyof TypePassedIn, KeysOfTypePassedIn>
```

EXCLUDE

ISOLATED EXAMPLE

Returns the provided union, minus any union items to exclude

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

```
type ExcludedKey = "b"
```

```
type KeysToInclude = Exclude<"a" | "b" | "c", "b">
```

keyof ExampleObjectType

EXCLUDE

ISOLATED EXAMPLE

Returns the provided union, minus any union items to exclude

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

```
type ExcludedKey = "b"
```

```
type KeysToInclude = "a" | "c"
```

```
type KeysToInclude = Exclude<"a" | "b" | "c", "b">
```

keyof ExampleObjectType

PICK

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

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

PICK

ISOLATED EXAMPLE

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

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

```
Exclude<"a" | "b" | "c", "b">

type PickedObject = Pick<ExampleObjectType, "a" | "c">
```

PICK

ISOLATED EXAMPLE

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

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

```
type PickedObject = {
    a: string;
    c: string[];
}
```

```
type PickedObject = Pick<ExampleObjectType, "a" | "c">
```

LEVEL 2

```
type Omit<TypePassedIn, KeysOfTypePassedIn> =

  Pick everthing in (Exclude KeysOfTypePassedIn from TypePassedIn key union) from TypePassedIn
```

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

```
type ObjectWithoutB = Omit<ExampleObjectType, "b">
```

LEVEL 2

```
type Omit<TypePassedIn, KeysOfTypePassedIn> =

  Pick everthing in (Exclude KeysOfTypePassedIn from TypePassedIn key union) from TypePassedIn
```

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

```
type ObjectWithoutB = {
    a: string;
    c: string[];
}
```

```
type ObjectWithoutB = Omit<ExampleObjectType, "b">
```

PLAYER INVENTORY

EVERYDAY TYPES	*string, number, boolean, union*
TYPE MANIPULATION	generics, conditionals, keyof
UTILITY TYPES	*exclude, pick, omit*

GAME OVERVIEW

LEVEL 1 type IsTruthy ★★★

LEVEL 2 type Omit ★★★

LEVEL 3 type ValidDoubleObject

LEVEL 4 The Final Boss...

LEVEL 3

START >

LEVEL 3

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

- ✓ generics
- ✓ conditionals

LEVEL 3

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

- ✓ generics
- ✓ conditionals

LEVEL 3

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

?

U?

LOADING

TYPE INFERENCE

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

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

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

TYPE INFERENCE

ISOLATED EXAMPLE

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

```typescript
type ExampleType = {
  a: string,
  b: string
}
```

```typescript
{
  a: infer U;
  b: infer U
}
```

TYPE INFERENCE

ISOLATED EXAMPLE

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

```typescript
type ExampleType = {
  a: string,
  b: string
}
```

```typescript
{
  a: infer U;
  b: infer U
}
```

U = string

LEVEL 3

```
type ValidDoubleObject<TypePassedIn> =

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

    then: the value is the InferedType

    else: the value is never
```

LEVEL 3

```
type ValidDoubleObject<TypePassedIn> =

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

    then: the value is the InferedType

    else: the value is never
```

LEVEL 3

```
type ValidDoubleObject<TypePassedIn> =

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

    then: the value is the InferedType

    else: the value is never
```

LEVEL 3

```
type ValidDoubleObject<TypePassedIn> =

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

    then: the value is the InferedType

    else: the value is never
```

LEVEL 3

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

```
type NotValid = ValidDoubleObject<string>
```

LEVEL 3

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

```
type NotValid = never
```

```
type NotValid = ValidDoubleObject<string>
```

LEVEL 3

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

```
type ValidObject = string
type ValidObject = ValidDoubleObject<{a: string, b: string}>
```

PLAYER INVENTORY

EVERYDAY TYPES	*string, number, boolean, union*
TYPE MANIPULATION	generics, conditionals, keyof, infer
UTILITY TYPES	*exclude, pick, omit*

GAME OVERVIEW

LEVEL 1 type IsTruthy ★★★

LEVEL 2 type Omit ★★★

LEVEL 3 type ValidDoubleObject ★

LEVEL 4 The Final Boss...

LEVEL 4

START >

LEVEL 4

```typescript
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];
```

LEVEL 4

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

LEVEL 4

```
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];
```

- ✓ generics
- ✓ conditionals
- ✓ keyof
- ✓ infer
- ✓ utility types

LEVEL 4

```
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];
```

- ✓ generics
- ✓ conditionals
- ✓ keyof
- ✓ infer
- ✓ utility types

LEVEL 4

```
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 types

mapped type modifiers

LOADING

MAPPED TYPES

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

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

K is an iterator of each key in TypePassedIn

MAPPED TYPES

ISOLATED EXAMPLE

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

```
type ExampleMappedType<TypePassedIn> = { [K in keyof TypePassedIn]-?: true}
```

MAPPED TYPES

ISOLATED EXAMPLE

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

```
type ExampleMappedType<TypePassedIn> = { [K in keyof TypePassedIn]-?: true}
```

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

```
type ConvertedValues = ExampleMappedType<ExampleObjectType>
```

MAPPED TYPES

ISOLATED EXAMPLE

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

```
type ExampleMappedType<TypePassedIn> = { [K in keyof TypePassedIn]-?: true}
```

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

```
type ConvertedValues = {
    a: true;
    b: true;
    c: true;
}
```

```
type ConvertedValues = ExampleMappedType<ExampleObjectType>
```

INDEX ACCESS TYPE

Key-ing into a specific propetry of a type

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

K is an iterator of each key in TypePassedIn - K can be used as an index access type

INDEX ACCESS TYPE

ISOLATED EXAMPLE

Key-ing into a specific propetry of a type

```
TypePassedIn[K]
```

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

```
K = "a"
```

```
type IndexedValue = ExampleObjectType["a"]
```

INDEX ACCESS TYPE

ISOLATED EXAMPLE

Key-ing into a specific propetry of a type

```
TypePassedIn[K]
```

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

```
K = "a"
```

```
type IndexedValue = string
```

```
type IndexedValue = ExampleObjectType["a"]
```

MAPPED TYPE MODIFIERS

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

MAPPED TYPE MODIFIERS

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.

MAPPED TYPE MODIFIERS

ISOLATED EXAMPLE

Operators that can be applied over a mapped type

```
type ExampleMappedType<TypePassedIn> = { [K in keyof TypePassedIn]-?: true}
```

```
type ExampleObjectType = {
  a?: string,
  b: number,
}
```

```
type ConvertedValues = ExampleMappedType<ExampleObjectType>
```

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

MAPPED TYPE MODIFIERS

ISOLATED EXAMPLE

Operators that can be applied over a mapped type

```
type ExampleMappedType<TypePassedIn> = { [K in keyof TypePassedIn]-?: true}
```

```
type ExampleObjectType = {
  a?: string,
  b: number,
}
```

```
type ConvertedValues = {
  a: true;
  b: true;
}
```

```
type ConvertedValues = ExampleMappedType<ExampleObjectType>
```

```
type ConvertedValues = ExampleMappedType<ExampleObjectType>
```

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

LEVEL 4

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassedIn as required]:
    if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
      then if: IsTruthy<InferredType> extends true
        then: the value is KeyOfTypePassedIn
        else: the value is never
      else: the value is never
}[keyof TypePassedIn];
```

- ✓ mapped types
- ✓ mapped type modifiers

LEVEL 4

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassedIn as required]:
    if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
      then if: IsTruthy<InferredType> extends true
        then: the value is KeyOfTypePassedIn
        else: the value is never
      else: the value is never
}[keyof TypePassedIn];
```

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

```
type ResultType = {
  valid1: ...
}
```

LEVEL 4

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassedIn as required]:
    if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
      then if: IsTruthy<InferredType> extends true
        then: the value is KeyOfTypePassedIn
        else: the value is never
      else: the value is never
}[keyof TypePassedIn];
```

```
{a: string, b: string} | null
```

```
{a: string, b: string}
```

LEVEL 4

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassedIn as required]:
    if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
      then if: IsTruthy<InferredType> extends true
        then: the value is KeyOfTypePassedIn
        else: the value is never
      else: the value is never
}[keyof TypePassedIn];
```

```
{a: string, b: string}
```

```
type InferedType = string
```

LEVEL 4

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassedIn as required]:
    if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
      then if: IsTruthy<InferredType> extends true
        then: the value is KeyOfTypePassedIn
        else: the value is never
      else: the value is never
}[keyof TypePassedIn];
```

```
type InferedType = string
```

LEVEL 4

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassedIn as required]:
    if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
      then if: IsTruthy<InferredType> extends true
        then: the value is KeyOfTypePassedIn
        else: the value is never
      else: the value is never
}[keyof TypePassedIn];
```

```
type ResultType = {
  valid1: 'valid1'
}
```

LEVEL 4

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassedIn as required]:
    if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
      then if: IsTruthy<InferredType> extends true
        then: the value is KeyOfTypePassedIn
        else: the value is never
      else: the value is never
}[keyof TypePassedIn];
```

```
type ResultType = {
  valid1: 'valid1'
}
```

⟫

"valid1"

LEVEL 4

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassedIn as required]:
    if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
      then if: IsTruthy<InferredType> extends true
        then: the value is KeyOfTypePassedIn
        else: the value is never
      else: the value is never
}[keyof TypePassedIn];
```

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

```
type ExampleValidKeys = ValidKeys<ExampleType>
```

LEVEL 4

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassedIn as required]:
    if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
      then if: IsTruthy<InferredType> extends true
        then: the value is KeyOfTypePassedIn
        else: the value is never
      else: the value is never
}[keyof TypePassedIn];
```

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

```
type ExampleValidKeys = "valid1"
```

```
type ExampleValidKeys = ValidKeys<ExampleType>
```

LEVEL 4

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassedIn as required]:
    if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
      then if: IsTruthy<InferredType> extends true
        then: the value is KeyOfTypePassedIn
        else: the value is never
      else: the value is never
}[keyof TypePassedIn];
```

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

LEVEL 4

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassedIn as required]:
    if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
      then if: IsTruthy<InferredType> extends true
        then: the value is KeyOfTypePassedIn
        else: the value is never
      else: the value is never
}[keyof TypePassedIn];
```

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

LEVEL 4

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassedIn as required]:
    if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
      then if: IsTruthy<InferredType> extends true
        then: the value is KeyOfTypePassedIn
        else: the value is never
      else: the value is never
}[keyof TypePassedIn];
```

```
type ResultType = {
  valid1: 'valid1'
  invalid1: never
}
```

LEVEL 4

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassedIn as required]:
    if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
      then if: IsTruthy<InferredType> extends true
        then: the value is KeyOfTypePassedIn
        else: the value is never
      else: the value is never
}[keyof TypePassedIn];
```

```
type ResultType = {
  valid1: 'valid1'
  invalid1: never
}
```

»» "valid1"

LEVEL 4

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassedIn as required]:
    if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
      then if: IsTruthy<InferredType> extends true
        then: the value is KeyOfTypePassedIn
        else: the value is never
      else: the value is never
}[keyof TypePassedIn];
```

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

```
type ExampleValidKeys = "valid1"
```

```
type ExampleValidKeys = ValidKeys<ExampleType>
```

LEVEL 4

```
type ExampleType = {
    invalid1?: {a: string, b: string}
}
```

LEVEL 4

```
type ExampleType = {
    invalid1?: {a: string, b: string}
}
```

```
type TestingValidKeysExample = never
```

```
type TestingValidKeysExample = ValidKeys<ExampleType>
```

LEVEL 4

```
type ExampleType = {
    invalid1?: {a: string, b: string}
}
```

```
type TestingValidKeysExample = never
```

```
type TestingValidKeysExample = ValidKeys<ExampleType>
```

OPTIONAL PROPERTIES

Describes a property that may or may not be set and is allowed to be empty

```
type ExampleType = {
    invalid1?: {a: string, b: string}
}
```

OPTIONAL PROPERTIES

Describes a property that may or may not be set and is allowed to be empty

```
type ExampleType = {
    invalid1?: {
        a: string;
        b: string;
    } | undefined;
}
```

```
type ExampleType = {
    invalid1?: {a: string, b: string}
}
```

Optional properties resolve to an undefined union

LEVEL 4

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassedIn as required]:
    if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
      then if: IsTruthy<InferredType> extends true
        then: the value is KeyOfTypePassedIn
        else: the value is never
      else: the value is never
}[keyof TypePassedIn];
```

```
type ExampleTypeAlternative = {
    invalid1?: {a: string, b: string} | undefined
}
```

```
type ResultType = {
  invalid1: ...
}
```

LEVEL 4

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassedIn as required]:
    if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
      then if: IsTruthy<InferredType> extends true
        then: the value is KeyOfTypePassedIn
        else: the value is never
      else: the value is never
}[keyof TypePassedIn];
```

```
{a: string, b: string} | undefined
```

```
{a: string, b: string} | undefined
```

LEVEL 4

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassedIn as required]:
    if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
      then if: IsTruthy<InferredType> extends true
        then: the value is KeyOfTypePassedIn
        else: the value is never
      else: the value is never
}[keyof TypePassedIn];
```

```
{a: string, b: string} | undefined
```

LEVEL 4

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassedIn as required]:
    if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
      then if: IsTruthy<InferredType> extends true
        then: the value is KeyOfTypePassedIn
        else: the value is never
      else: the value is never
}[keyof TypePassedIn];
```

```
type ResultType = {
  invalid1: never
}
```

LEVEL 4

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassedIn as required]:
    if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
      then if: IsTruthy<InferredType> extends true
        then: the value is KeyOfTypePassedIn
        else: the value is never
      else: the value is never
}[keyof TypePassedIn];
```

```
type ResultType = {
  invalid1: never
}
```

⟫ never !

LEVEL 4

```
type ValidKeys<TypePassedIn> = {
  [each KeyOfTypePassedIn as required]:
    if: (Exclude null from ValueOfTypePassedIn) extends { a: InferredType; b: InferredType }
      then if: IsTruthy<InferredType> extends true
        then: the value is KeyOfTypePassedIn
        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}
}
```

```
type ExampleValidKeys = "valid1" | "valid2"
```

```
type ExampleValidKeys = ValidKeys<ExampleType>
```

GAME OVERVIEW

LEVEL 1 type IsTruthy ★★★

LEVEL 2 type Omit ★★★

LEVEL 3 type ValidDoubleObject ★

LEVEL 4 The Final Boss... ★★★★

EXIT

FINAL INVENTORY

EVERYDAY TYPES	*string, number, boolean, union*
TYPE MANIPULATION	generics, conditionals, keyof, infer, mapped types + modifiers, index access
UTILITY TYPES	*exclude, pick, omit*
OBJECT TYPES	*optional properties*

GAME
OVER