

HOW TO MAKE TYPESCRIPT WORK FOR YOU

THE BASICS

THE BASICS

- TS compiles to JS

THE BASICS

- TS compiles to JS
- Types have *no effect* at runtime!

THE BASICS

- TS compiles to JS
- Types have *no effect* at runtime!
- Type annotations are just assumptions

***TS'S HELP IS ONLY AS GOOD AS
YOUR ASSUMPTIONS!***

HOW TO AVOID ASSUMPTIONS

- NEVER use any
- Avoid type casting
- Verify I/O data at runtime

ONCE ASSUMPTIONS ARE GONE...

ONCE ASSUMPTIONS ARE GONE...

- Make types as restrictive as possible

ONCE ASSUMPTIONS ARE GONE...

- Make types as restrictive as possible
- TS says something could go wrong → Do you agree?

ONCE ASSUMPTIONS ARE GONE...

- Make types as restrictive as possible
- TS says something could go wrong → Do you agree?
 - Yes → add runtime checks!

ONCE ASSUMPTIONS ARE GONE...

- Make types as restrictive as possible
- TS says something could go wrong → Do you agree?
 - Yes → add runtime checks!
 - No → Improve types

NEVER USE any

NEVER USE any

```
1 type User = {  
2   name: string  
3   email: string  
4 }  
5  
6 const getUser = async () => {  
7   const response = await fetch('api/user-data')  
8   return await response.json()  
9 }
```

NEVER USE any

```
1 type User = {  
2   name: string  
3   email: string  
4 }  
5  
6 const getUser = async (): Promise<User> => {  
7   const response = await fetch('api/user-data')  
8   return await response.json()  
9 }
```

NEVER USE any

```
1 type User = {  
2   name: string  
3   email: string  
4 }  
5  
6 const getUser = async (): Promise<User> => {  
7   const response = await fetch('api/user-data')  
8   return await response.json()  
9 }
```


NEVER USE any

```
1 type User = {  
2   name: string  
3   email: string  
4 }  
5  
6 const getUser = async (): Promise<User> => {  
7   const response = await fetch('api/user-data')  
8   return (await response.json()) as User  
9 }
```

AVOID TYPE CASTING

AVOID TYPE CASTING

```
1 // command.ts
2 export type Command = {
3   name: string
4   alias?: string
5 }
6
7 // other stuff
```

```
1 // addCommand.ts
2
3 export default {
4   mane: 'add',
5 }
```

AVOID TYPE CASTING

```
1 // command.ts
2 export type Command = {
3   name: string
4   alias?: string
5 }
6
7 // other stuff
```

```
1 // addCommand.ts
2
3 export default {
4   name: 'add',
5 } as Command
```

AVOID TYPE CASTING

```
1 // command.ts
2 export type Command = {
3   name: string
4   alias?: string
5 }
6
7 // other stuff
```

```
1 // addCommand.ts
2
3 export default {
4   name: 'add',
5   altName: 'a',
6 } as Command
```

AVOID TYPE CASTING

```
1 type Something = {  
2   onlyProperty: number  
3 }  
4  
5 const foo: Something = {  
6   onlyProperty: 42,  
7   additional: 'nope',  
8 }  
9 // Object literal may only  
10 // specify known properties
```

```
1 // addCommand.ts  
2  
3 export default {  
4   name: 'add',  
5   altName: 'a',  
6 } as Command
```

AVOID TYPE CASTING

```
1 type Something = {  
2   onlyProperty: number  
3 }  
4  
5 const foo: Something = {  
6   onlyProperty: 42,  
7   additional: 'nope',  
8 }  
9 // Object literal may only  
10 // specify known properties
```

```
1 // addCommand.ts  
2  
3 const add: Command = {  
4   name: 'add',  
5   alias: 'a',  
6 }  
7  
8 export default add
```

AVOID TYPE CASTING

```
1 // addCommand.ts
2
3 export default {
4   name: 'add',
5   alias: 'a',
6 } satisfies Command
```

```
1 // addCommand.ts
2
3 const add: Command = {
4   name: 'add',
5   alias: 'a',
6 }
7
8 export default add
```


AVOID TYPE CASTING

```
1 // addCommand.ts
2
3 export default {
4   name: 'add',
5   alias: 'a',
6 } satisfies Command
7 /*
8  * => {
9  *   name: string
10  *   alias: string
11  * }
12 */
```

```
1 // addCommand.ts
2
3 const add: Command = {
4   name: 'add',
5   alias: 'a',
6 }
7
8 export default add
9 /*
10  * => {
11  *   name: string
12  *   alias?: string
13  * }
14 */
```

AVOID TYPE CASTING

```
1 // addCommand.ts
2
3 export default {
4   name: 'add',
5   alias: 'a',
6 } satisfies Command
7 /*
8  * => {
9  *   name: string
10  *   alias: string
11  * }
12 */
```

```
1 // addCommand.ts
2
3 const add: Command = {
4   name: 'add',
5   alias: 'a',
6 }
7
8 export default add
9 /*
10  * => {
11  *   name: string
12  *   alias?: string
13  * }
14 */
```

AVOID TYPE CASTING

```
// addCommand.ts

export default {
  name: 'add',
  alias: 'a',
} as const satisfies Command
/*
 * => {
 *   name: "add"
 *   alias: "a"
 * }
 */
```

```
// addCommand.ts

const add: Command = {
  name: 'add',
  alias: 'a',
} as const

export default add
/*
 * => {
 *   name: string
 *   alias?: string
 * }
 */
```

VERIFY I/O DATA AT RUNTIME

VERIFY I/O DATA AT RUNTIME

```
1  const verifyUser = (input: unknown): User => {
2    if (typeof input !== 'object' || input === null) {
3      throw new Error('user needs to be an object')
4    }
5
6    const result: User = {
7      name: '',
8      email: '',
9    }
10
11    if (
12      !('name' in input) ||
13      typeof input.name !== 'string'
14    ) {
15      // ...

```

VERIFY I/O DATA AT RUNTIME

```
1  const verifyUser = (input: unknown): User => {
2    if (typeof input !== 'object' || input === null) {
3      throw new Error('user needs to be an object')
4    }
5
6    const result: User = {
7      name: '',
8      email: '',
9    }
10
11    if (
12      !('name' in input) ||
13      typeof input.name !== 'string'
14    ) {
15      // ...
16    }
17  }
```

VERIFY I/O DATA AT RUNTIME

```
5
6  const result: User = {
7      name: '',
8      email: '',
9  }
10
11  if (
12      !('name' in input) ||
13      typeof input.name !== 'string'
14  ) {
15      throw new Error('missing name of type string')
16  } else {
17      result.name = input.name
18  }
19
```

VERIFY I/O DATA AT RUNTIME

```
17     result.name = input.name
18 }
19
20 if (
21     !('email' in input) ||
22     typeof input.email !== 'string'
23 ) {
24     throw new Error('missing name of type string')
25 } else {
26     result.email = input.email
27 }
28
29 return result
30 }
```


VERIFY I/O DATA AT RUNTIME

```
1 import { z } from 'zod'
2
3 const userSchema = z.object({
4   name: z.string(),
5   email: z.string().email(),
6 })
7 type User = z.infer<typeof userSchema>
8
9 const getUser = async () => {
10   const response = await fetch('api/user-data')
11   const fromJson = await response.json()
12   return userSchema.parse(fromJson)
13 }
14
15
```

VERIFY I/O DATA AT RUNTIME

```
1 import { z } from 'zod'
2
3 const userSchema = z.object({
4   name: z.string(),
5   email: z.string().email(),
6 })
7 type User = z.infer<typeof userSchema>
8
9 const getUser = async () => {
10   const response = await fetch('api/user-data')
11   const fromJson = await response.json()
12   return userSchema.parse(fromJson)
13 }
14
15
```

VERIFY I/O DATA AT RUNTIME

```
4   name: z.string(),
5   email: z.string().email(),
6 })
7 type User = z.infer<typeof userSchema>
8
9 const getUser = async () => {
10   const response = await fetch('api/user-data')
11   const fromJson = await response.json()
12   return userSchema.parse(fromJson)
13 }
14
15 const parseResult = userSchema.safeParse({
16   name: 1,
17   email: 'not an email!',
18 })
```

VERIFY I/O DATA AT RUNTIME

```
10  const response = await fetch('api/user-data')
11  const fromJson = await response.json()
12  return userSchema.parse(fromJson)
13 }
14
15 const parseResult = userSchema.safeParse({
16   name: 1,
17   email: 'not an email!',
18 })
19
20 parseResult.data // <- User | undefined
21 parseResult.error // <- ZodError | undefined
22
23 if (parseResult.success) {
```

VERIFY I/O DATA AT RUNTIME

```
14
15 const parseResult = userSchema.safeParse({
16   name: 1,
17   email: 'not an email!',
18 })
19
20 parseResult.data // <- User | undefined
21 parseResult.error // <- ZodError | undefined
22
23 if (parseResult.success) {
24   parseResult.data // <- User
25   parseResult.error // <- undefined
26 } else {
27   parseResult.data // <- undefined
28 }
```

VERIFY I/O DATA AT RUNTIME

```
19
20 parseResult.data // <- User | undefined
21 parseResult.error // <- ZodError | undefined
22
23 if (parseResult.success) {
24   parseResult.data // <- User
25   parseResult.error // <- undefined
26 } else {
27   parseResult.data // <- undefined
28   parseResult.error // <- ZodError
29
30   parseResult.error.issues
31   /*
32     [
33
```

VERIFY I/O DATA AT RUNTIME

```
28  parseResult.error // <- ZodError
29
30  parseResult.error.issues
31  /*
32      [
33          {
34              code: "invalid_type",
35              expected: "string",
36              received: "number",
37              path: [ "name" ],
38              message: "Expected string, received number"
39          },
40          {
41              validation: "email",
```

VERIFY I/O DATA AT RUNTIME

```
37         path: [ "name" ],
38         message: "Expected string, received number"
39     },
40     {
41         validation: "email",
42         code: "invalid_string",
43         message: "Invalid email",
44         path: [ "email" ]
45     }
46 ]
47 */
48
49 const errorsByProperty = parseResult.error.format()
50 errorsByProperty.name?._errors
51 // ["Expected string, received number", "Invalid email"]
```


VERIFY I/O DATA AT RUNTIME

```
41         validation: "email",
42         code: "invalid_string",
43         message: "Invalid email",
44         path: [ "email" ]
45     }
46 ]
47 */
48
49 const errorsByProperty = parseResult.error.format()
50 errorsByProperty.name?._errors
51 // -> ["Expected string, received number"]
52 errorsByProperty.email?._errors
53 // -> ["Invalid email"]
54 }
```

MAKING ILLEGAL VALUES
UNREPRESENTABLE

MAKING ILLEGAL VALUES *UNREPRESENTABLE*

```
1 // Attention: this may only ever be either CSV,  
2 // one of the standard options or the config form data  
3 type UserInput = {  
4   csv?: string  
5   providedStandardOptionId?: string  
6  
7   // the rest are input values from the config form  
8   start?: Date  
9   end?: Date  
10  values?: number[]  
11 }
```

MAKING ILLEGAL VALUES *UNREPRESENTABLE*

```
11 }  
12  
13 let userInput: UserInput = {}  
14  
15 const setCsv = (csv: string) => {  
16   userInput = { csv }  
17 }  
18  
19 const setConfigData = (start: Date, end: Date) => {  
20   userInput = { start, end }  
21 }
```

MAKING ILLEGAL VALUES *UNREPRESENTABLE*

```
24     userInput = { providedStandardOptionId: optionId }
25   }
26
27   const getInputType = (userInput: UserInput): string => {
28     if (userInput.csv !== undefined) {
29       return 'csv'
30     } else if (
31       userInput.providedStandardOptionId !== undefined
32     ) {
33       return 'standardOption'
34     } else {
```

MAKING ILLEGAL VALUES *UNREPRESENTABLE*

```
12  
13 let userInput: UserInput = {}  
14  
15 const setCsv = (csv: string) => {  
16   userInput = { csv }  
17 }  
18  
19 const setConfigData = (start: Date, end: Date) => {  
20   userInput = { start, end }  
21 }  
22
```

MAKING ILLEGAL VALUES *UNREPRESENTABLE*

```
29     return csv
30   } else if (
31     userInput.providedStandardOptionId !== undefined
32   ) {
33     return 'standardOption'
34   } else {
35     return 'configForm'
36   }
37 }
38
39 // Problem: this type-checks
40 const everything: IUserInput = {
```

MAKING ILLEGAL VALUES *UNREPRESENTABLE*

```
1 type UserInput =  
2   | {  
3     csv: string  
4   }  
5   | {  
6     providedStandardOptionId: string  
7   }  
8   | {  
9     start: Date  
10    end: Date  
11    values: number[]
```


MAKING ILLEGAL VALUES *UNREPRESENTABLE*

```
14 let userInput: UserInput | undefined
15
16 const setCsv = (csv: string) => {
17   userInput = { csv }
18 }
19
20 const setConfigData = (
21   start: Date,
22   end: Date,
23   values: number[],
24 ) => {
```

MAKING ILLEGAL VALUES *UNREPRESENTABLE*

```
28 const setStandardOption = (optionId: string) => {  
29   userInput = { providedStandardOptionId: optionId }  
30 }  
31  
32 const getInputType = (userInput: UserInput): string => {  
33   if ('csv' in userInput) {  
34     return 'csv'  
35   } else if ('providedStandardOptionId' in userInput) {  
36     return 'standardOption'  
37   } else {  
38     return 'configForm'
```

MAKING ILLEGAL VALUES *UNREPRESENTABLE*

```
32 const getInputType = (userInput: userInput): string => {  
33   if ('csv' in userInput) {  
34     return 'csv'  
35   } else if ('providedStandardOptionId' in userInput) {  
36     return 'standardOption'  
37   } else {  
38     return 'configForm'  
39   }  
40 }  
41  
42 // Problem: this type-checks  
43 const everything: IUserInput = {
```

MAKING ILLEGAL VALUES *UNREPRESENTABLE*

```
1  type UserInput =  
2    | {  
3        type: 'csv'  
4        csv: string  
5    }  
6    | {  
7        type: 'standardOption'  
8        providedStandardOptionId: string  
9    }  
10   | {  
11       type: 'configForm'
```

MAKING ILLEGAL VALUES *UNREPRESENTABLE*

```
17 let userInput: UserInput | undefined
18
19 const setCsv = (csv: string) => {
20   userInput = { type: 'csv', csv }
21 }
22
23 const setConfigData = (
24   start: Date,
25   end: Date,
26   values: number[],
27 ) => {
```

MAKING ILLEGAL VALUES *UNREPRESENTABLE*

```
31 const setStandardOption = (optionId: string) => {
32   userInput = {
33     type: 'standardOption',
34     providedStandardOptionId: optionId,
35   }
36 }
37
38 const getInputType = (userInput: UserInput): string => {
39   return userInput.type
40 }
41
```

MAKING ILLEGAL VALUES *UNREPRESENTABLE*

```
33     type: StandardOption ,  
34     providedStandardOptionId: optionId,  
35   }  
36 }  
37  
38 const getInputType = (userInput: UserInput): string => {  
39   return userInput.type  
40 }  
41  
42 // This doesn't type-check -> problem solved  
43 // @ts-expect-error  
44 const everything: UserInput = {
```

DEALING WITH DISCRIMINATED UNIONS

```
1  const sumValues = (userInput: UserInput): number => {
2    let values: number[] = []
3    if (userInput.type === 'configForm') {
4      values = userInput.values
5    } else if (userInput.type === 'csv') {
6      values = userInput.csv
7        .split(',')
8        .map((n) => parseInt(n))
9    } else {
10     switch (userInput.providedStandardOptionId) {
11       case 'default':
```


DEALING WITH DISCRIMINATED UNIONS

```
6     values = userInput.csv
7         .split(',')
8         .map((n) => parseInt(n))
9     } else {
10         switch (userInput.providedStandardOptionId) {
11             case 'default':
12                 values = [1, 2, 3]
13                 break
14             case 'special':
15                 values = [42, 42, 42, 42]
16                 break
17             default:
```

DEALING WITH DISCRIMINATED UNIONS

```
1  const sumValues = (userInput: UserInput): number => {  
2    let values: number[] = []  
3    if (userInput.type === 'configForm') {  
4      values = userInput.values  
5    } else if (userInput.type === 'csv') {  
6      values = userInput.csv  
7        .split(',')  
8        .map((n) => parseInt(n))  
9    } else {  
10     switch (userInput.providedStandardOptionId) {  
11       case 'default':
```

DEALING WITH DISCRIMINATED UNIONS

```
1  const sumValues = (userInput: UserInput): number => {
2    let values: number[] = []
3    if (userInput.type === 'configForm') {
4      values = userInput.values
5    } else if (userInput.type === 'csv') {
6      values = userInput.csv
7        .split(',')
8        .map((n) => parseInt(n))
9    } else {
10     switch (userInput.providedStandardOptionId) {
11       case 'default':
```

DEALING WITH DISCRIMINATED UNIONS

```
1  const sumValues = (userInput: UserInput): number => {
2    let values: number[] = []
3    if (userInput.type === 'configForm') {
4      values = userInput.values
5    } else if (userInput.type === 'csv') {
6      values = userInput.csv
7        .split(',')
8        .map((n) => parseInt(n))
9    } else {
10     switch (userInput.providedStandardOptionId) {
11       case 'default':
```

DEALING WITH DISCRIMINATED UNIONS

```
6     values = userInput.csv
7         .split(',')
8         .map((n) => parseInt(n))
9 } else {
10     switch (userInput.providedStandardOptionId) {
11         case 'default':
12             values = [1, 2, 3]
13             break
14         case 'special':
15             values = [42, 42, 42, 42]
16             break
```

DEALING WITH DISCRIMINATED UNIONS

```
1  const sumValues = (userInput: UserInput): number => {
2    let values: number[] = []
3    if (userInput.type === 'configForm') {
4      values = userInput.values
5    } else if (userInput.type === 'csv') {
6      values = userInput.csv
7        .split(',')
8        .map((n) => parseInt(n))
9    } else {
10     switch (userInput.providedStandardOptionId) {
11       case 'default':
```

PATTERN MATCHING

```
1 import { UserInput } from './userInput'
2 import { P, match } from 'ts-pattern'
3
4 const sumValues = (userInput: UserInput): number => {
5   const values = match(userInput)
6     .with(
7     { type: 'configForm' },
8     (userInput) => userInput.values,
9   )
10  .with({ type: 'csv' }, ({ csv }) =>
11    csv.split(',').map((n) => parseInt(n)),
```

PATTERN MATCHING

```
1 import { UserInput } from './userInput'
2 import { P, match } from 'ts-pattern'
3
4 const sumValues = (userInput: UserInput): number => {
5   const values = match(userInput)
6     .with(
7     { type: 'configForm' },
8     (userInput) => userInput.values,
9   )
10   .with({ type: 'csv' }, ({ csv }) =>
11     csv.split(',').map((n) => parseInt(n)),
```


PATTERN MATCHING

```
import scala.util.matching.Regex

3
4 const sumValues = (userInput: UserInput): number => {
5   const values = match(userInput)
6     .with(
7       { type: 'configForm' },
8       (userInput) => userInput.values,
9     )
10    .with({ type: 'csv' }, ({ csv }) =>
11      csv.split(',').map((n) => parseInt(n)),
12    )
13    .with(
```

PATTERN MATCHING

```
12     )
13     .with(
14         {
15             type: 'standardOption',
16             providedStandardOptionId: P.select(),
17         },
18         (optionId) => {
19             switch (optionId) {
20                 case 'default':
21                     return [1, 2, 3]
22                 case 'special':
```

PATTERN MATCHING

```
15         type: StandardOption ,
16         providedStandardOptionId: P.select(),
17     },
18     (optionId) => {
19         switch (optionId) {
20             case 'default':
21                 return [1, 2, 3]
22             case 'special':
23                 return [42, 42, 42, 42]
24             default:
25                 return []
26         }
```

PATTERN MATCHING

```
1 import { UserInput } from './userInput'
2 import { P, match } from 'ts-pattern'
3
4 const sumValues = (userInput: UserInput): number => {
5   const values = match(userInput)
6     .with(
7     { type: 'configForm' },
8     (userInput) => userInput.values,
9   )
10  .with({ type: 'csv' }, ({ csv }) =>
11    csv.split(',').map((n) => parseInt(n)),
```

PATTERN MATCHING

```
12     )
13     .with(
14         {
15             type: 'standardOption',
16             providedStandardOptionId: P.select(),
17         },
18         (optionId) => {
19             switch (optionId) {
20                 case 'default':
21                     return [1, 2, 3]
22                 case 'special':
```

PATTERN MATCHING

```
1 import { UserInput } from './userInput'
2 import { match } from 'ts-pattern'
3
4 const sumValues = (userInput: UserInput): number => {
5   const values = match(userInput)
6     .with(
7     { type: 'configForm' },
8     (userInput) => userInput.values,
9   )
10  .with({ type: 'csv' }, ({ csv }) =>
11    csv.split(',').map((n) => parseInt(n)),
```

PATTERN MATCHING

```
13     .with(  
14         {  
15             type: 'standardOption',  
16             providedStandardOptionId: 'default',  
17         },  
18         () => [1, 2, 3],  
19     )  
20     .with(  
21         {  
22             type: 'standardOption',  
23             providedStandardOptionId: 'special',  
24         },  
25     )
```

PATTERN MATCHING

```
13     .with(  
14         {  
15             type: 'standardOption',  
16             providedStandardOptionId: 'default',  
17         },  
18         () => [1, 2, 3],  
19     )  
20     .with(  
21         {  
22             type: 'standardOption',  
23             providedStandardOptionId: 'special',  
24         },  
25     )
```


PATTERN MATCHING

```
1 import { UserInput } from './userInput'
2 import { P, match } from 'ts-pattern'
3
4 const sumValues = (userInput: UserInput): number => {
5   const values = match(userInput)
6     .with(
7     { type: 'configForm' },
8     (userInput) => userInput.values,
9   )
10  .with({ type: 'csv' }, ({ csv }) =>
11    csv.split(',').map((n) => parseInt(n)),
```

UTILITY TYPES

UTILITY TYPES

```
1 type User = {  
2   name: string  
3   age: number  
4   foodPreference: 'none' | 'vegetarian' | 'vegan'  
5 }  
6  
7 type UserUpdate = {  
8   name?: User['name']  
9   age?: User['age']  
10  foodPreference?: User['foodPreference']  
11 }
```

UTILITY TYPES

```
1 type User = {  
2   name: string  
3   age: number  
4   foodPreference: 'none' | 'vegetarian' | 'vegan'  
5 }  
6  
7 type UserUpdate = Partial<User>
```

UTILITY TYPES

```
1 type User = {  
2   name: string  
3   age: number  
4   foodPreference: 'none' | 'vegetarian' | 'vegan'  
5 }  
6  
7 type UserUpdate = Partial<User>  
8 type ComplicatedUser = Required<UserUpdate>  
9  
10 type WithoutFood = Omit<User, 'foodPreference'>  
11 type WithoutFood2 = Pick<User, 'name' | 'age'>  
12  
13 type InterestingKeys = Exclude<keyof User, 'age'>
```

UTILITY TYPES

```
1 type User = {  
2   name: string  
3   age: number  
4   foodPreference: 'none' | 'vegetarian' | 'vegan'  
5 }  
6  
7 type UserUpdate = Partial<User>  
8 type ComplicatedUser = Required<UserUpdate>  
9  
10 type WithoutFood = Omit<User, 'foodPreference'>  
11 type WithoutFood2 = Pick<User, 'name' | 'age'>  
12  
13 type InterestingKeys = Exclude<keyof User, 'age'>
```

UTILITY TYPES

```
1 type User = {  
2   name: string  
3   age: number  
4   foodPreference: 'none' | 'vegetarian' | 'vegan'  
5 }  
6  
7 type UserUpdate = Partial<User>  
8 type ComplicatedUser = Required<UserUpdate>  
9  
10 type WithoutFood = Omit<User, 'foodPreference'>  
11 type WithoutFood2 = Pick<User, 'name' | 'age'>  
12  
13 type InterestingKeys = Exclude<keyof User, 'age'>
```

UTILITY TYPES

```
1 type User = {  
2   name: string  
3   age: number  
4   foodPreference: 'none' | 'vegetarian' | 'vegan'  
5 }  
6  
7 type UserUpdate = Partial<User>  
8 type ComplicatedUser = Required<UserUpdate>  
9  
10 type WithoutFood = Omit<User, 'foodPreference'>  
11 type WithoutFood2 = Pick<User, 'name' | 'age'>  
12  
13 type InterestingKeys = Exclude<keyof User, 'age'>
```


UTILITY TYPES

```
1 type User = {  
2   name: string  
3   age: number  
4   foodPreference: 'none' | 'vegetarian' | 'vegan'  
5 }  
6  
7 type UserUpdate = Partial<User>  
8 type ComplicatedUser = Required<UserUpdate>  
9  
10 type WithoutFood = Omit<User, 'foodPreference'>  
11 type WithoutFood2 = Pick<User, 'name' | 'age'>  
12  
13 type InterestingKeys = Exclude<keyof User, 'age'>
```

UTILITY TYPES

```
1  const magicFunction = (a: number, b: string) => {
2      return {
3          foo: 42,
4          bar: {
5              biz: 'baz',
6          },
7      }
8  }
9
10 type Result = ReturnType<typeof magicFunction>
11 type Params = Parameters<typeof magicFunction>
12 // => [a: number, B; string]
```

UTILITY TYPES

```
1  const magicFunction = (a: number, b: string) => {
2      return {
3          foo: 42,
4          bar: {
5              biz: 'baz',
6          },
7      }
8  }
9
10 type Result = ReturnType<typeof magicFunction>
11 type Params = Parameters<typeof magicFunction>
12 // => [a: number, B; string]
```

UTILITY TYPES

```
1  const magicFunction = (a: number, b: string) => {  
2    return {  
3      foo: 42,  
4      bar: {  
5        biz: 'baz',  
6      },  
7    }  
8  }  
9  
10 type Result = ReturnType<typeof magicFunction>  
11 type Params = Parameters<typeof magicFunction>  
12 // => [a: number, B; string]
```



LIVE DEMO 🙌

TAKE-AWAYS

- Never use any
- Types should always match reality
 - Make illegal values unrepresentable
 - Don't assume (= cast), parse / validate!
 - If a check happens at runtime, let TypeScript know about it

TAKE-AWAYS

- Never use any
- Types should always match reality
 - Make illegal values unrepresentable
 - Don't assume (= cast), parse / validate!
 - If a check happens at runtime, let TypeScript know about it
- Compiler errors are bugs you don't have to catch - appreciate them!

THANK YOU!



Michael Kuckuk



@LBBO@ruhr.social



inovex