Better TYPESCRIPT Types

getting typescript to help me more because i make a lot of mistakes

Rob Howard
@damncabbage
http://robhoward.id.au

Rob Howard
@damncabbage
http://robhoward.id.au

Audience Interaction Time

0.

Why

Write down my assumptions in a way the computer understands, then have it tell me when I'm wrong.

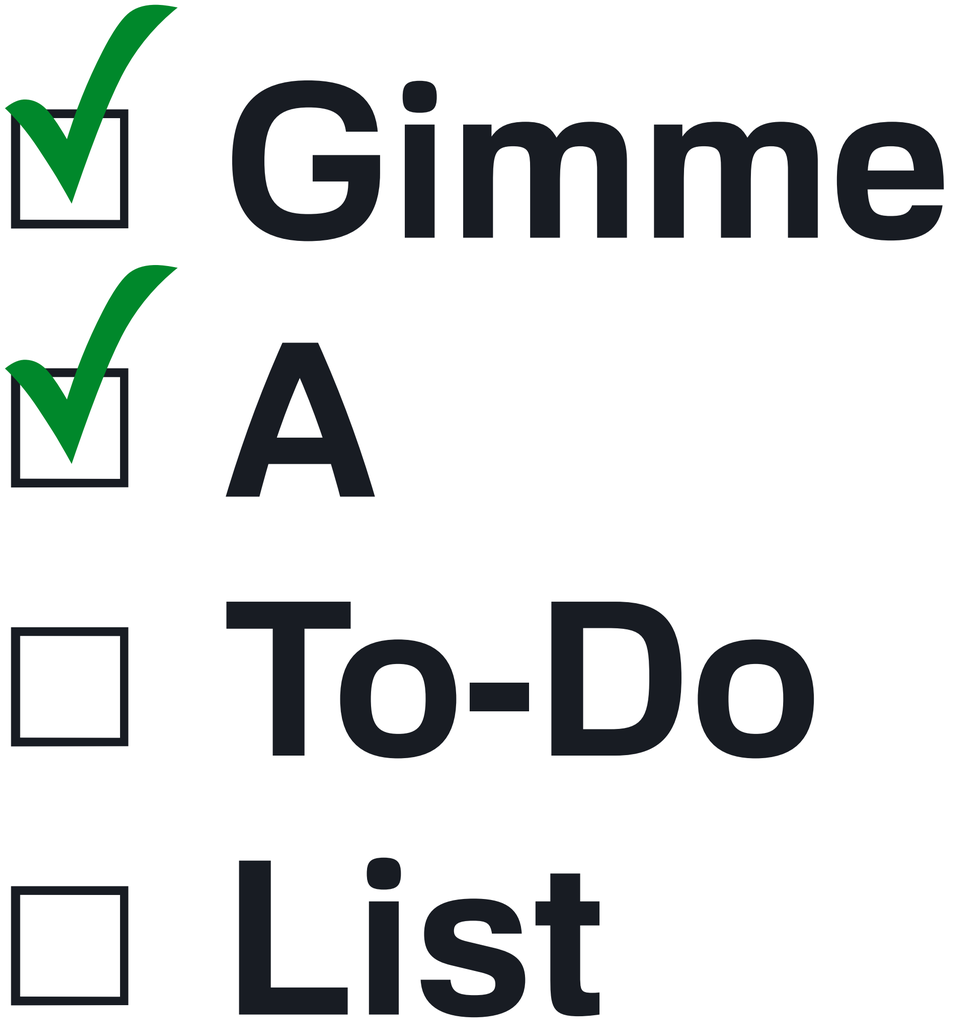

Dear Kate,
Here's to the crazy ones. The misfits.
The troublemakers. The round pegs in the
holes. The ones who see things differently.
not fond of rules. And they have no respect for the
status quo. You can quote them, disagree with them,
glorify or vilify them. About the only thing you can't
do is ignore them. Because they change things.
They push the human race forward. And while some
may see them as the crazy ones, we see genius.
Because the people who are crazy enough to think
they can change the world, are the ones who do.
Take care.
John Appleseed

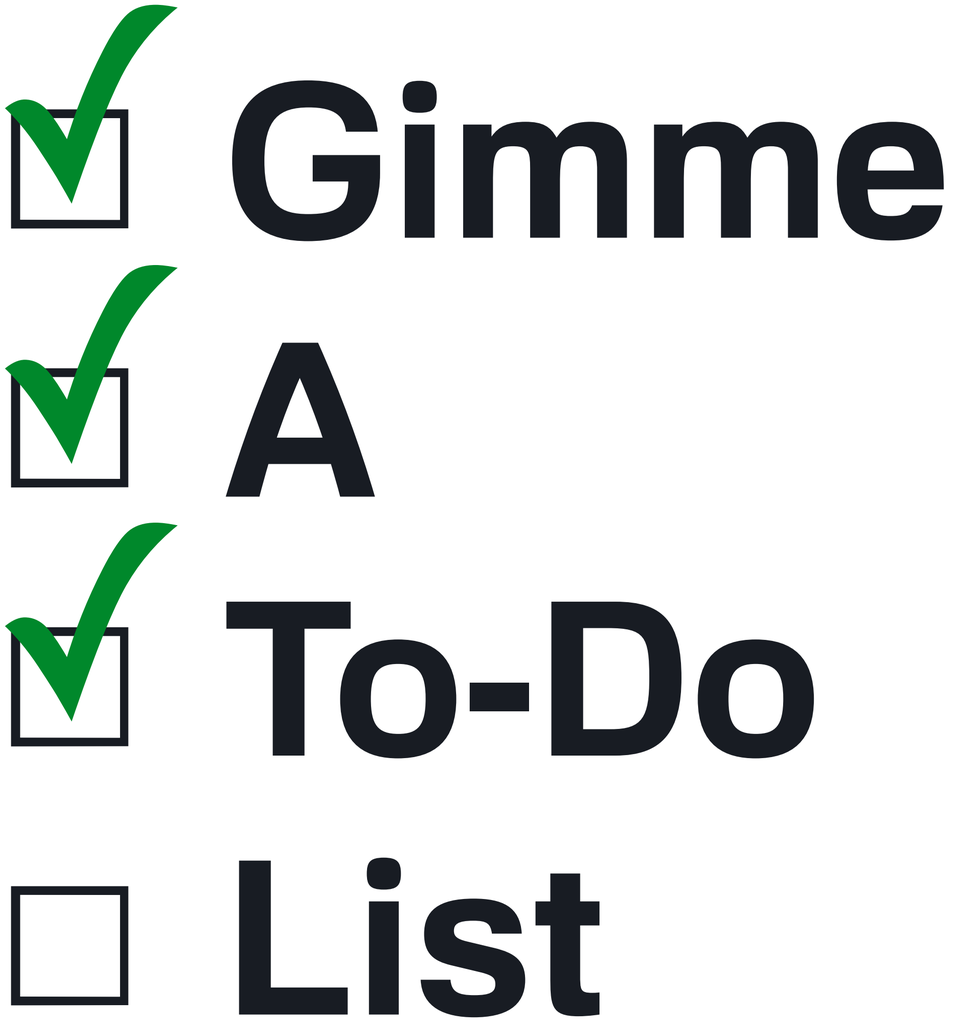

Dear Kate,
Here's to the crazy ones. The misfits.
The troublemakers. The round pegs in the
holes. The ones who see things differently.
not fond of rules. And they have no respect for the
status quo. You can quote them, disagree with them,
glorify or vilify them. About the only thing you can't
do is ignore them. Because they change things.
They push the human race forward. And while some
may see them as the crazy ones, we see genius.
Because the people who are crazy enough to think
they can change the world, are the ones who do.
Take care.
John Appleseed

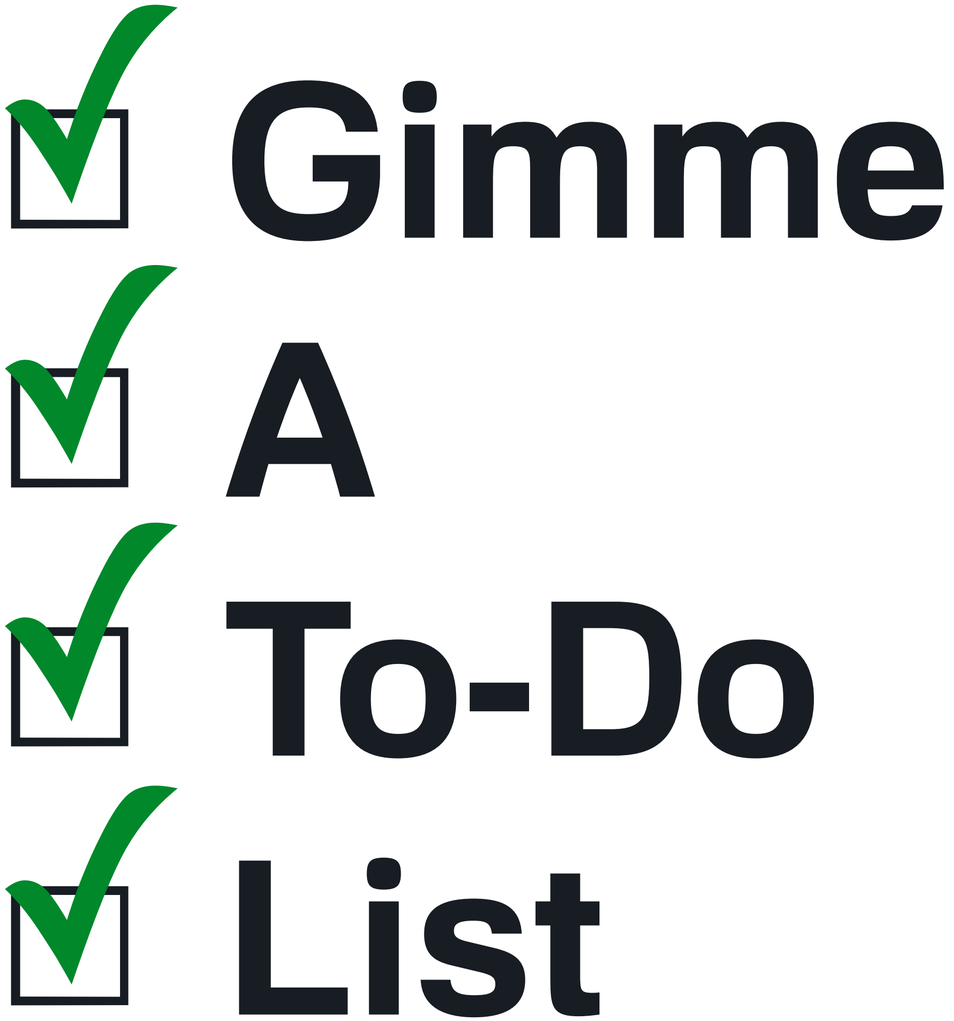

Dear Kate,
Here's to the crazy ones. The
The troublemakers. The round
holes. The ones who see things
not fond of rules. And they have
status quo. You can quote them, disagree with them,
glorify or vilify them. About the only thing you can't
do is ignore them. Because they change things.
They push the human race forward. And while some
may see them as the crazy ones, we see genius.
Because the people who are crazy enough to think
they can change the world, are the ones who do.
Take care.
John Appleseed

- [ ] Gimme
- [ ] A
- [ ] To-Do
- [ ] List

- [x] Gimme
- [ ] A
- [ ] To-Do
- [ ] List


Gimme
A
To-Do
List


Gimme
A
To-Do
List


Gimme
A
To-Do
List

1.

Types
as describing
Groups of Stuff

1.1

The Boring Ones

```
type Group = ...
```

```
type Group = boolean
```

```
type Group = boolean

const a: Group = true
```

```
type Group = boolean

const a: Group = true
const b: Group = false
```

```
type Group = boolean

const a: Group = true
const b: Group = false
const c: Group = null
```

```
type Group = boolean

const a: Group = true
const b: Group = false
const c: Group = null
```

```
type Group = boolean

const a: Group = true
const b: Group = false
const c: Group = null
const d: Group = undefined
```

```
type Group = boolean

const a: Group = true
const b: Group = false
const c: Group = null
const d: Group = undefined
```

```
type Group = boolean

const a: Group = true
const b: Group = false
const c: Group = null
const d: Group = undefined
const d: Group = 123
```

```
type Group = boolean

const a: Group = true
const b: Group = false
const c: Group = null
const d: Group = undefined
const d: Group = 123
```

```
type Group = number
```

```
type Group = number

const a: Group = 1
```

```
type Group = number

const a: Group = 1
const b: Group = 2.3
```

```
type Group = number

const a: Group = 1
const b: Group = 2.3
const c: Group = undefined
```

```
type Group = number

const a: Group = 1
const b: Group = 2.3
const c: Group = undefined
```

```
type Group = number

const a: Group = 1
const b: Group = 2.3
const c: Group = undefined
const d: Group = NaN
```

```
type Group = number

const a: Group = 1
const b: Group = 2.3
const c: Group = undefined
const d: Group = NaN
```

```
type Group = number

const a: Group = 1
const b: Group = 2.3
const c: Group = undefined
const d: Group = NaN
```

```
type Group = null
```

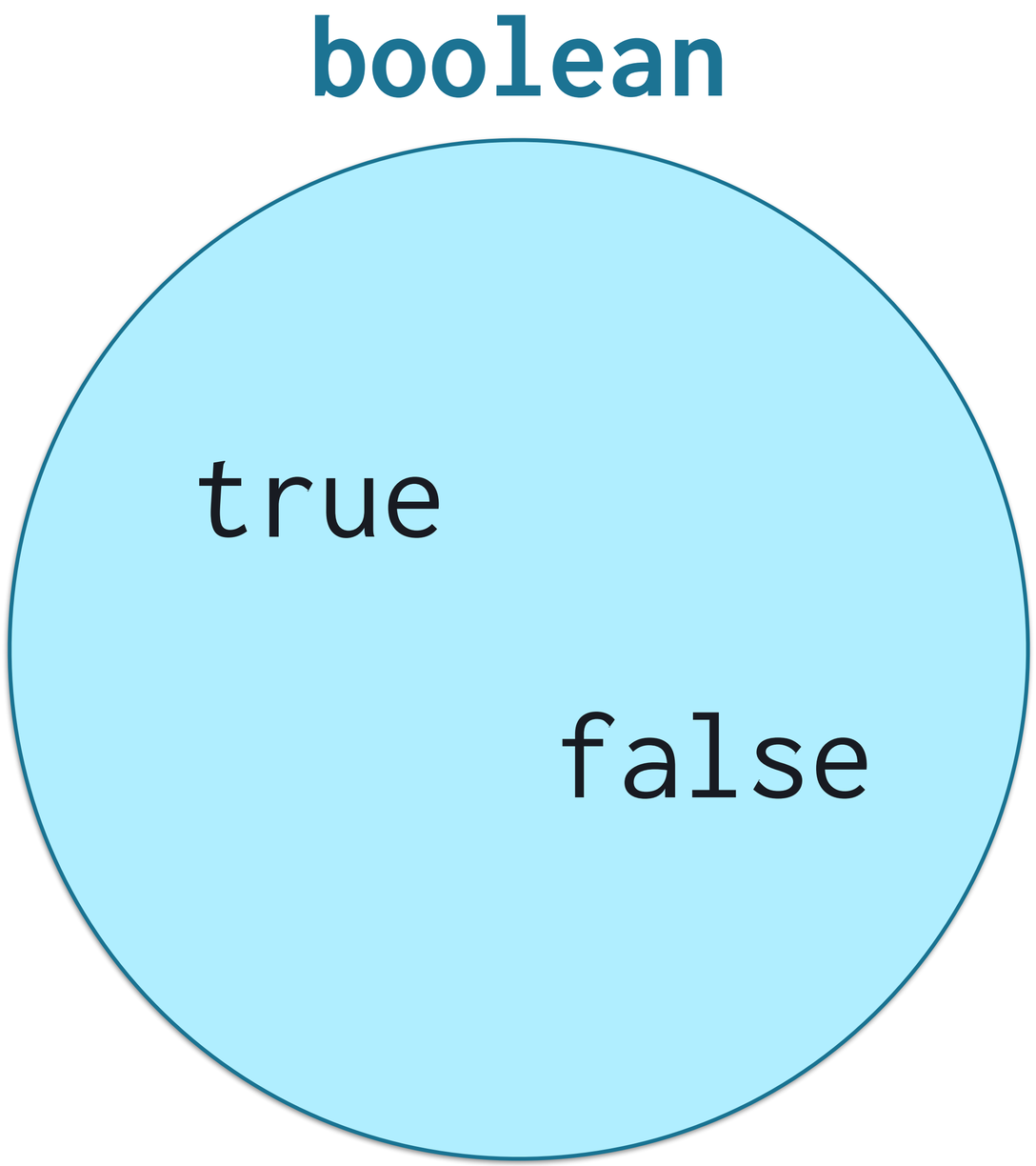
```
type Group = null

const a: Group = null
```

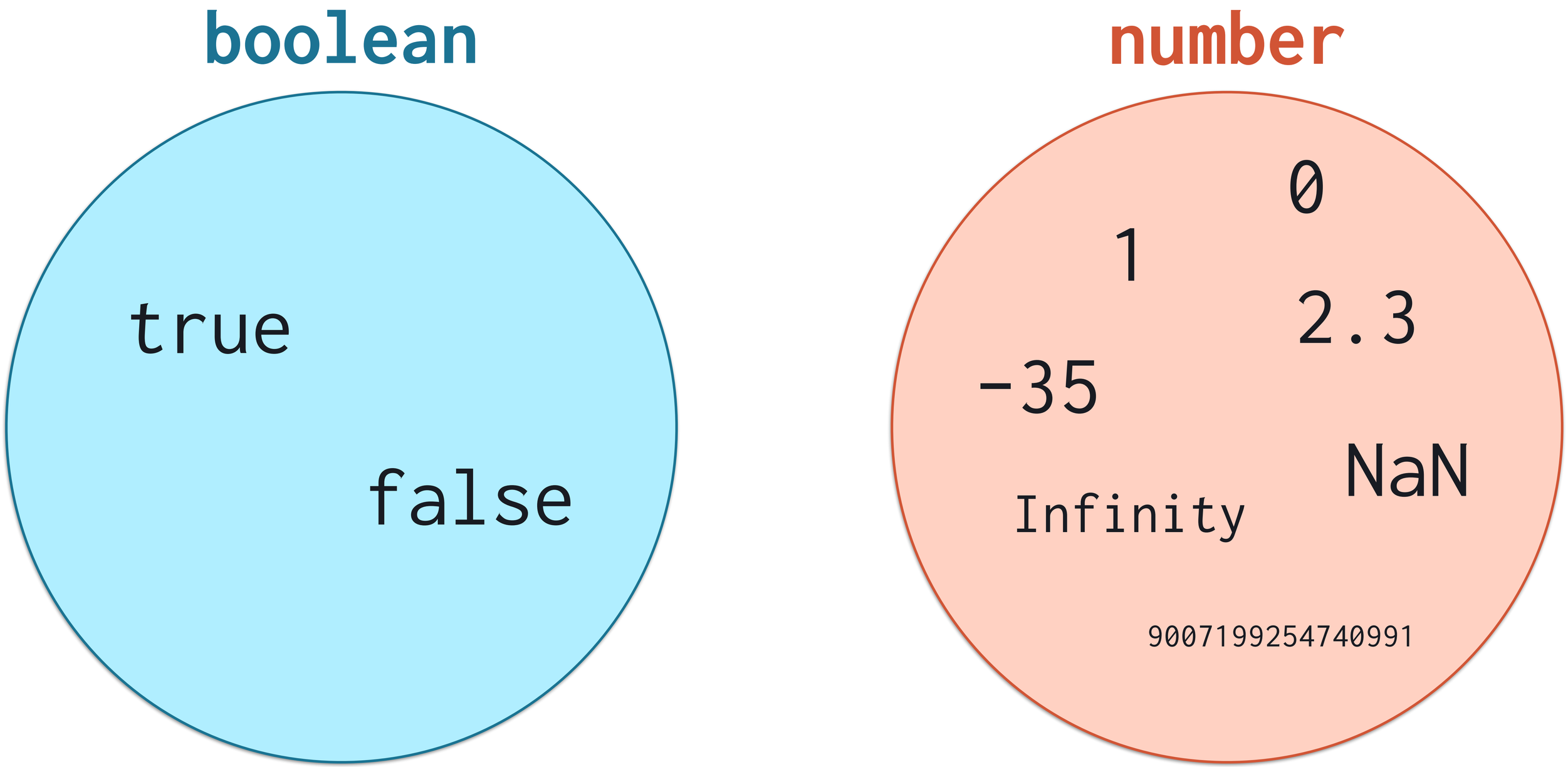
```
type Group = null

const a: Group = null
const b: Group = undefined
```

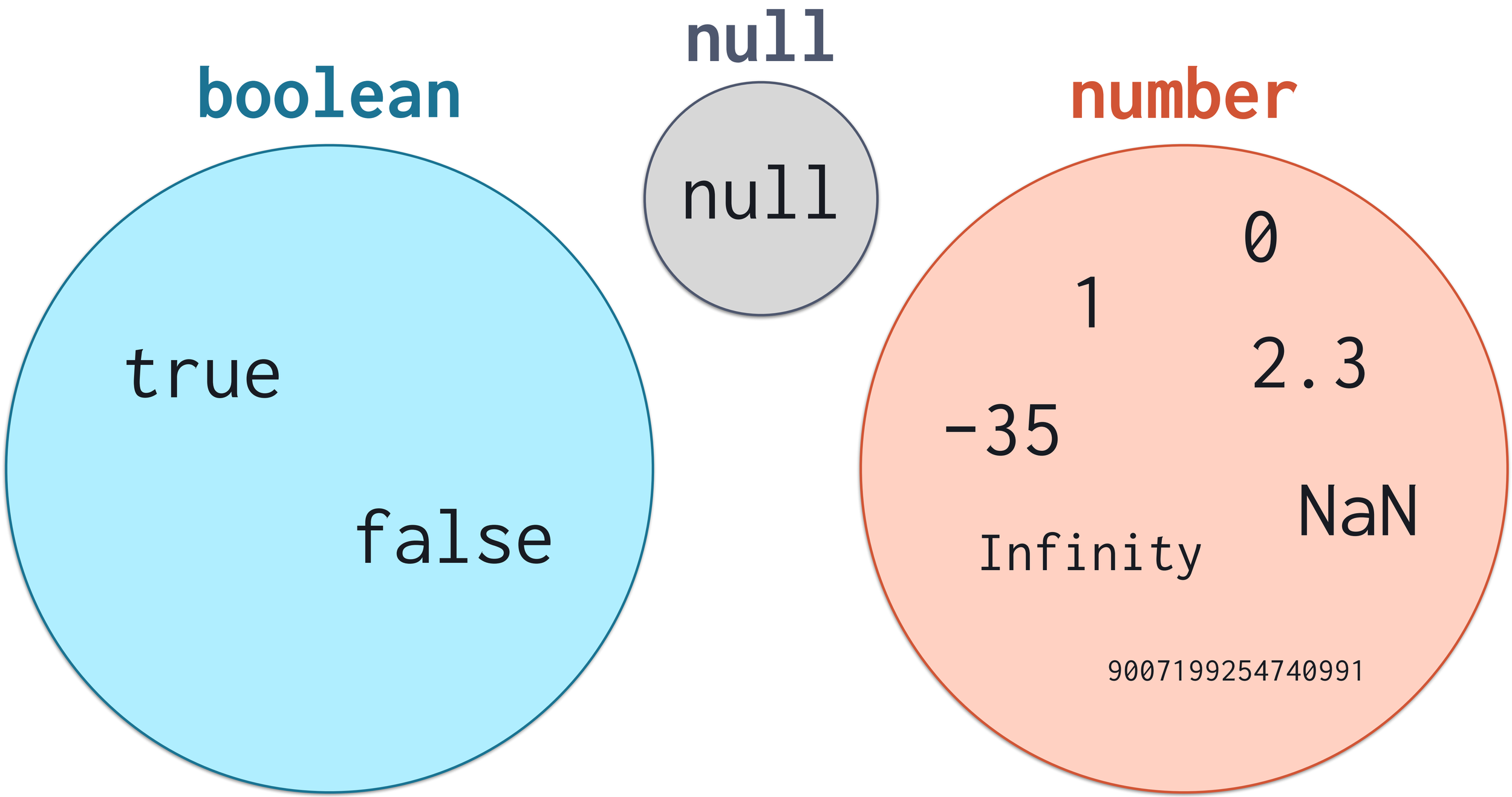
```
type Group = null

const a: Group = null
const b: Group = undefined
```

```
type Group = null

const a: Group = null
const b: Group = undefined
const c: Group = 123
```

```
type Group = null

const a: Group = null
const b: Group = undefined
const c: Group = 123
```


boolean
true
false


boolean
true
false
number
0
1
2.3
-35
NaN
Infinity
9007199254740991


boolean
true
false
null
null
number
0
1
2.3
-35
NaN
Infinity
9007199254740991

```
type Group = undefined

const a: Group = undefined
const b: Group = 123
```

```
type Group = void

const a: Group = undefined
const b: Group = 123
```

```
type Group = string
```

```
type Group = string

const a: Group = ""
```

```
type Group = string

const a: Group = ""
const b: Group = "Hello!"
```

```
type Group = string

const a: Group = ""
const b: Group = "Hello!"
const c: Group = "冷蔵庫"
```

```
type Group = string

const a: Group = ""
const b: Group = "Hello!"
const c: Group = "冷蔵庫"
const d: Group = "ACT I SCENE I.
Athens. The palace of THESEUS.
Enter THESEUS, HIPPOLYTA, …
```

```
type Group = {}
```

```
type Group = {}

const a: Group = {}
```

```
type Group = {}

const a: Group = {}
const b: Group = false
```

```
type Group = {}

const a: Group = {}
const b: Group = false
```

```
type Group = { abc: number }
```

```
type Group = { abc: number }

const a: Group = { abc: 123 }
```

```
type Group = { abc: number }

const a: Group = { abc: 123 }
const b: Group = { abc: true }
```

```
type Group = { abc: number }

const a: Group = { abc: 123 }
const b: Group = { abc: true }
```

```
type Group = { abc: number }

const a: Group = { abc: 123 }
const b: Group = { abc: true }
const c: Group = {}
```

```
type Group = { abc: number }

const a: Group = { abc: 123 }
const b: Group = { abc: true }
const c: Group = {}
```

```
type Group = { abc: number }

const a: Group = { abc: 123 }
const b: Group = { abc: true }
const c: Group = {}
const d: Group = { abc: 123,
                   def: 456,
                 }
```

```
type Group = { abc: number }

const a: Group = { abc: 123 }
const b: Group = { abc: true }
const c: Group = {}
const d: Group = { abc: 123,
    def: 456,
}
```

```
type Group = { abc: number }

const a: Group = { abc: 123 }
const b: Group = { abc: true }
const c: Group = {}
const d: Group = { abc: 123,
    def: 456,
}
```

```
type Group = { abc: number }

const a: Group = { abc: 123 }
const b: Group = { abc: true }
const c: Group = {}
const d: Group = { abc: 123,
                   def: 456,
                 }
```

```ts
interface Group {
  abc: number;
}

const a: Group = { abc: 123 }
const b: Group = { abc: true }
const c: Group = {}
const d: Group = { abc: 123,
```

1.2

"Literal" Types

```
type Group = 123
```

```
type Group = 123

const a: Group = 123
```

```
type Group = 123

const a: Group = 123
const b: Group = 124
```

```
type Group = 123

const a: Group = 123
const b: Group = 124
```

123

```
type Group = "red"

const a: Group = "red"
const b: Group = "blue"
```

```
type Group = null

const a: Group = null
const b: Group = undefined
const c: Group = 123
```

```
type Group = undefined

const a: Group = undefined
const b: Group = 123
```

```
type Group = true

const a: Group = true
const b: Group = false
```

```
type Group = false

const a: Group = false
const b: Group = true
```

```
type Group = { abc: 123 }

const a: Group = { abc: 123 }
const b: Group = { abc: 456 }
```

1.3

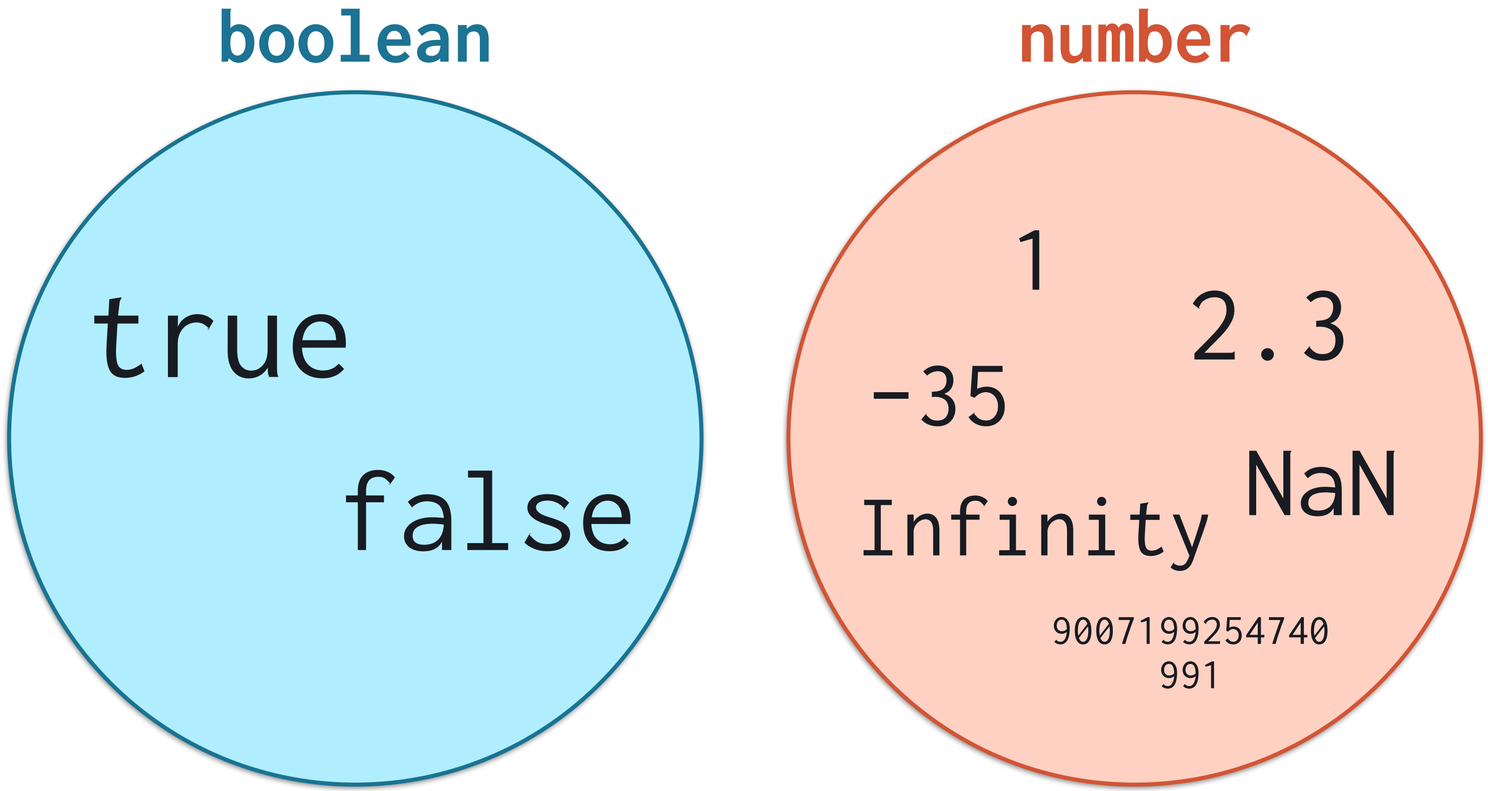
Unholy Unions

```
type Group = boolean | number
```

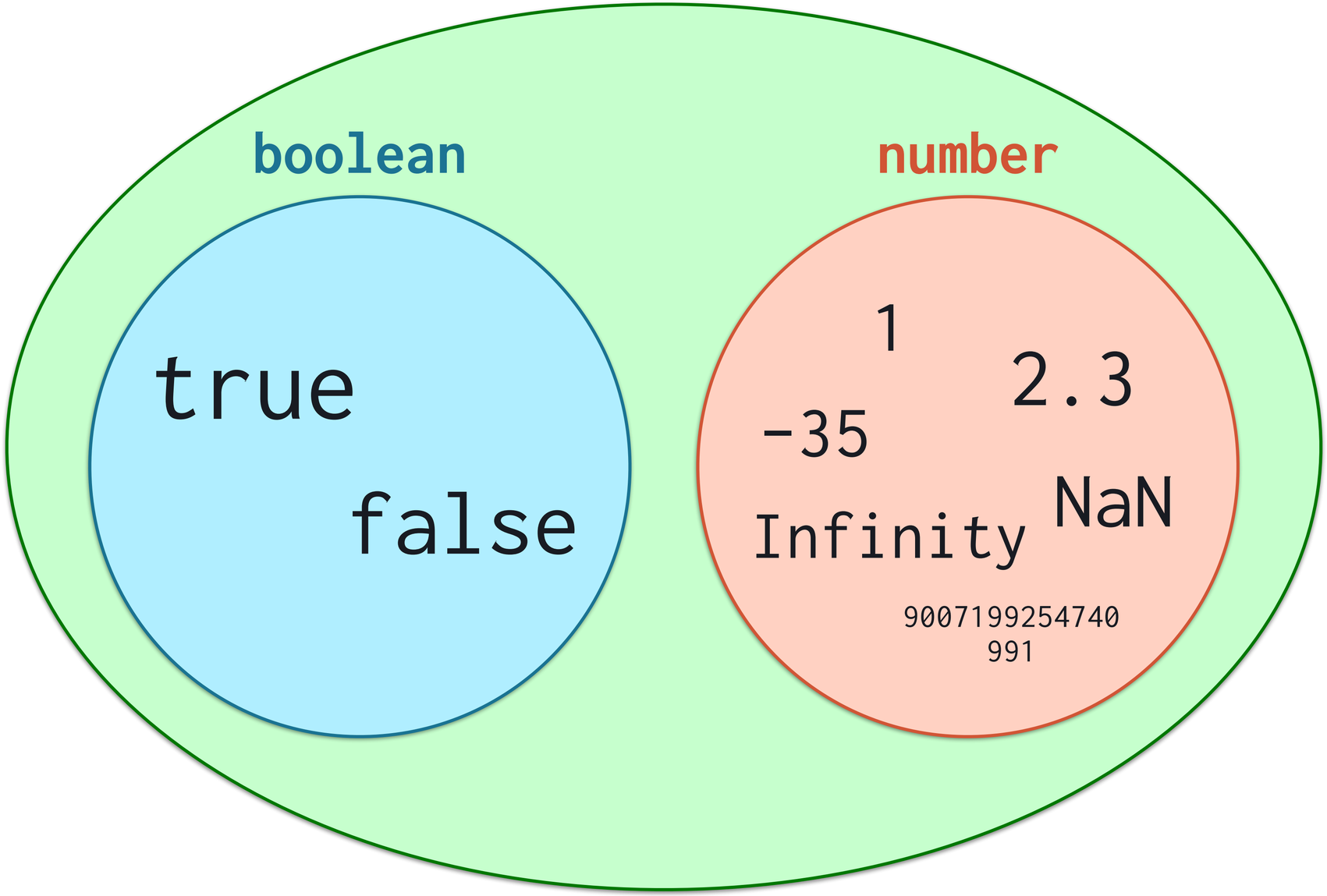
```
type Group = boolean | number

const a: Group = true
```

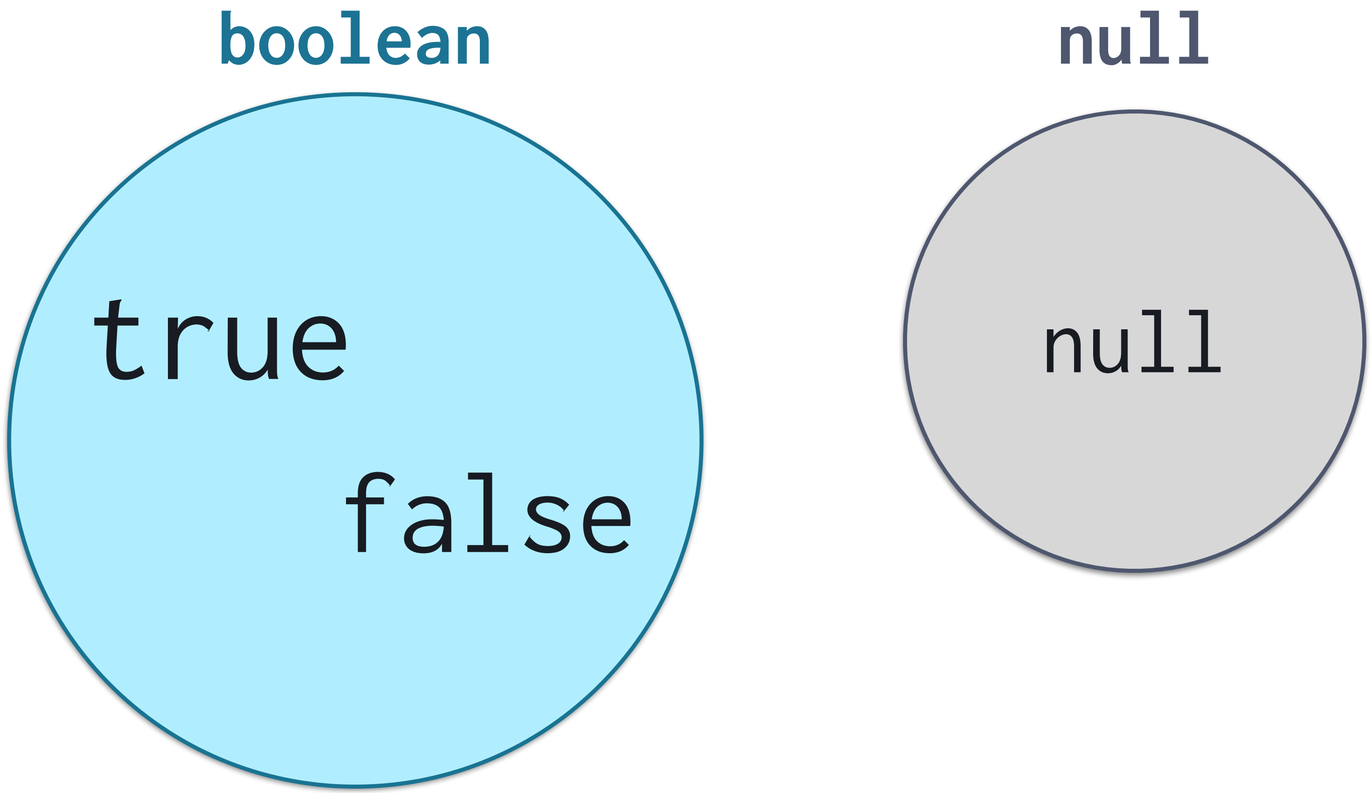
```
type Group = boolean | number

const a: Group = true
const b: Group = false
```

```
type Group = boolean | number

const a: Group = true
const b: Group = false
const c: Group = 5
```

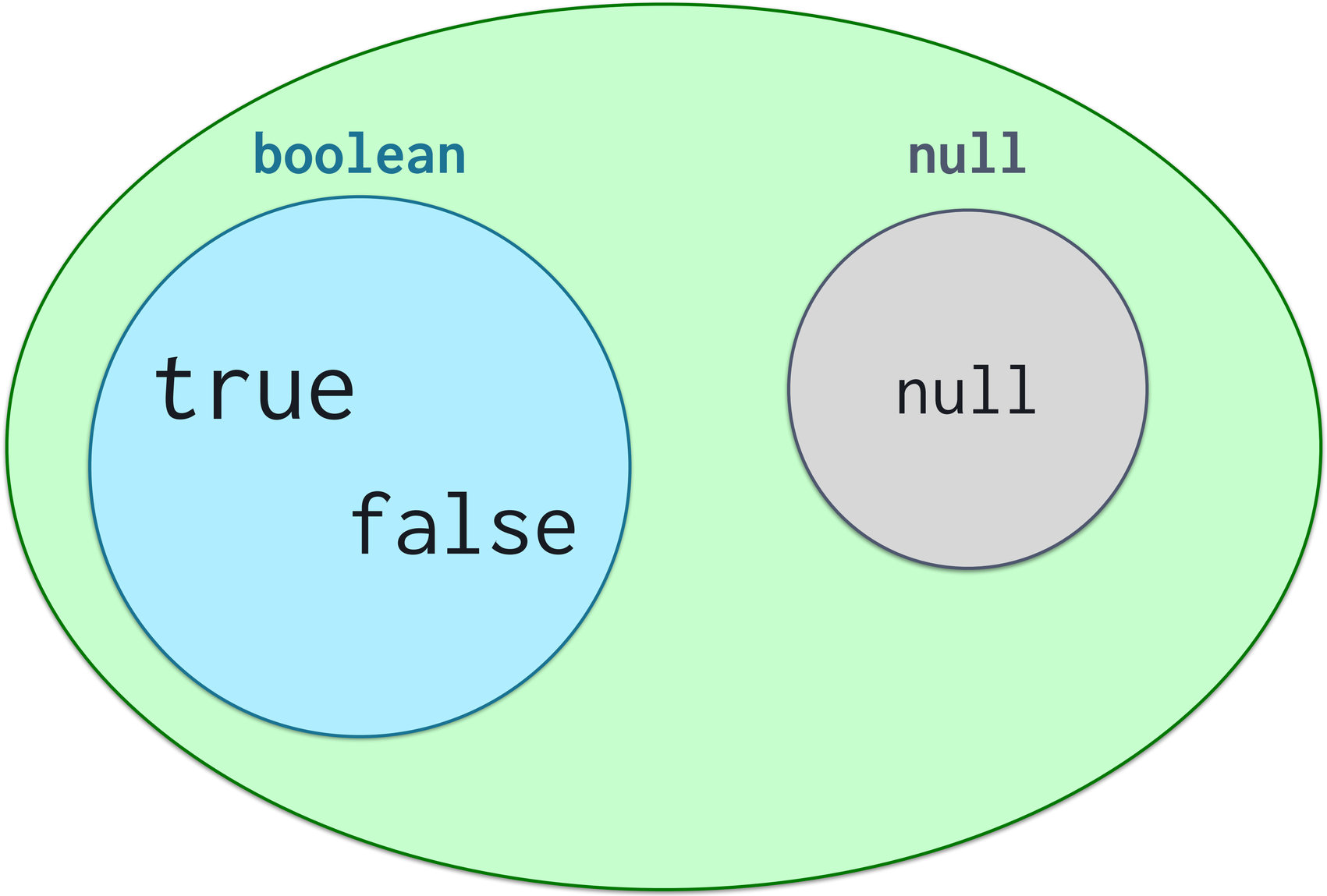

boolean
true
false
number
1
2.3
-35
NaN
Infinity
9007199254740
991

boolean | number




boolean
true
false
null
null

boolean | null



```
type Group =
  | { type: "loading" }
  | { type: "done", data: number }
```

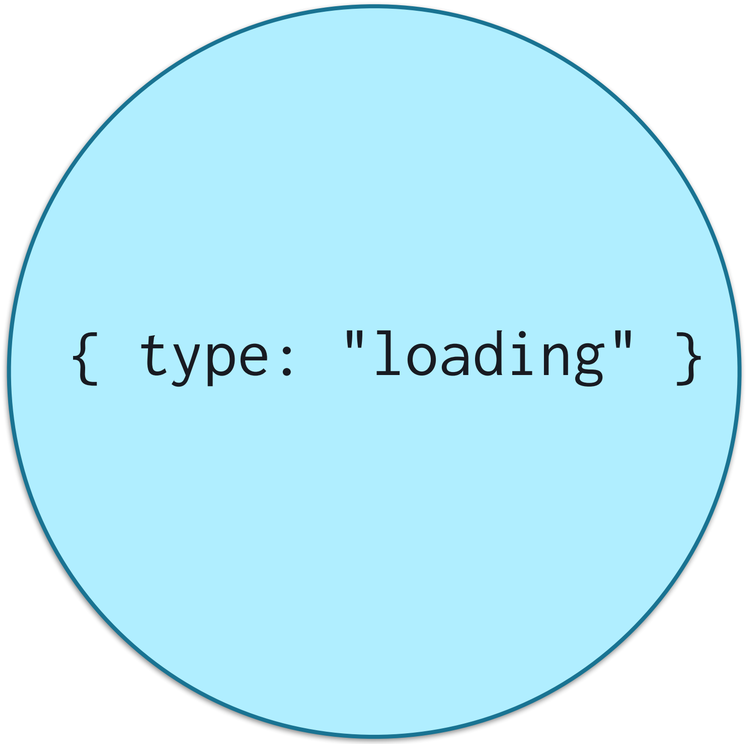
```
type Group =
  | { type: "loading" }
  | { type: "done", data: number }

const a: Group = { type: "loading" }
```

```
type Group =
  | { type: "loading" }
  | { type: "done", data: number }

const a: Group = { type: "loading" }
const b: Group = { type: "done",
                   data: 123 }
```

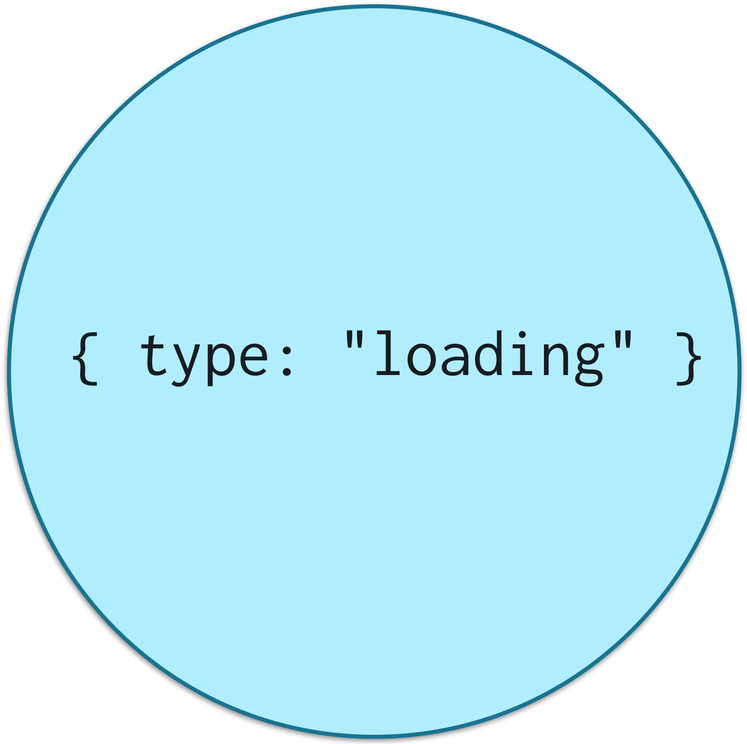
```
type Group =
  | { type: "loading" }
  | { type: "done", data: number }

const a: Group = { type: "loading" }
const b: Group = { type: "done",
                   data: 123 }
const c: Group = { type: "loading",
                   data: 123 }
```

```
type Group =
  | { type: "loading" }
  | { type: "done", data: number }

const a: Group = { type: "loading" }
const b: Group = { type: "done",
                   data: 123 }
const c: Group = { type: "loading",
                   data: 123 }
```

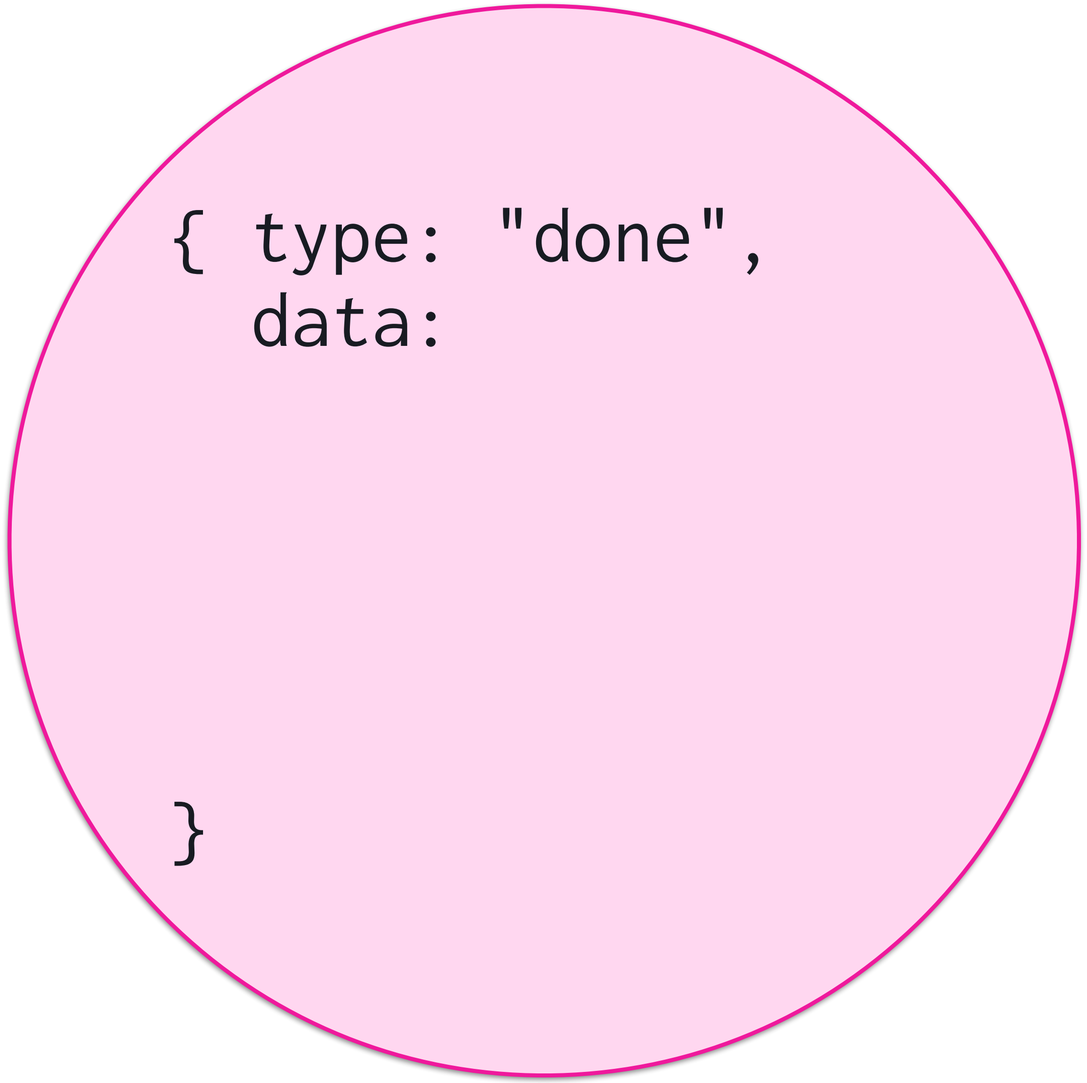
{ type: "loading" }



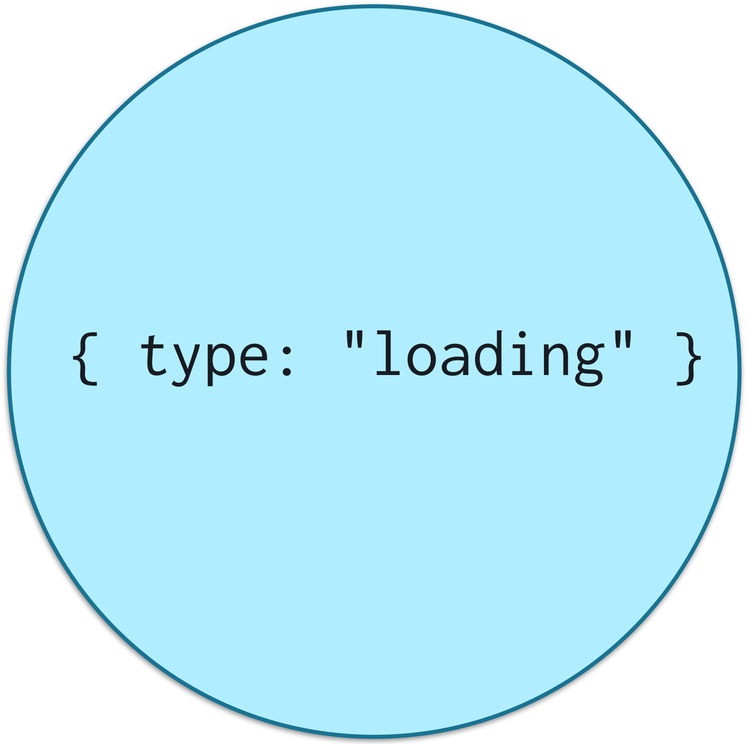
{ type: "loading" }



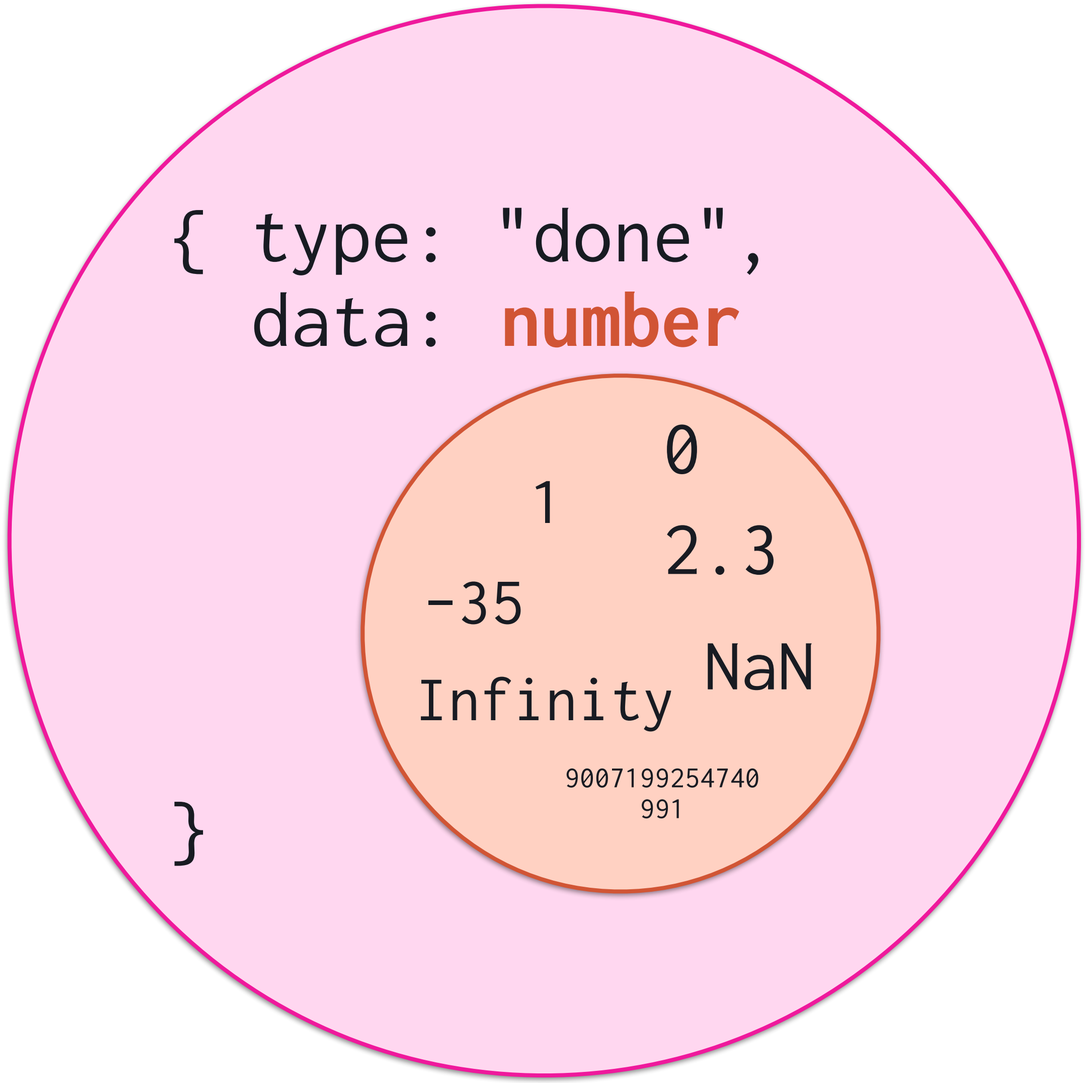
{ type: "done", data: number }

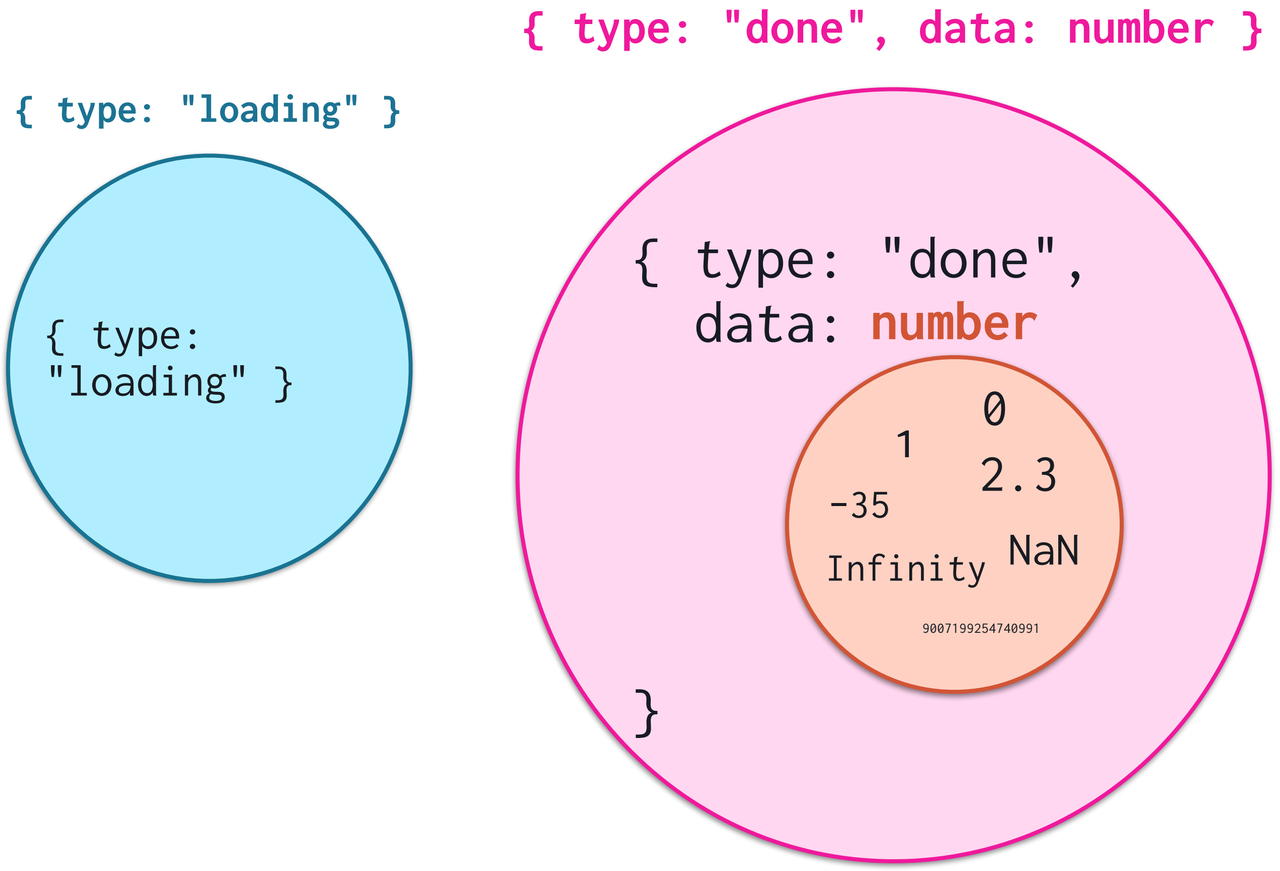


{ type: "loading" }

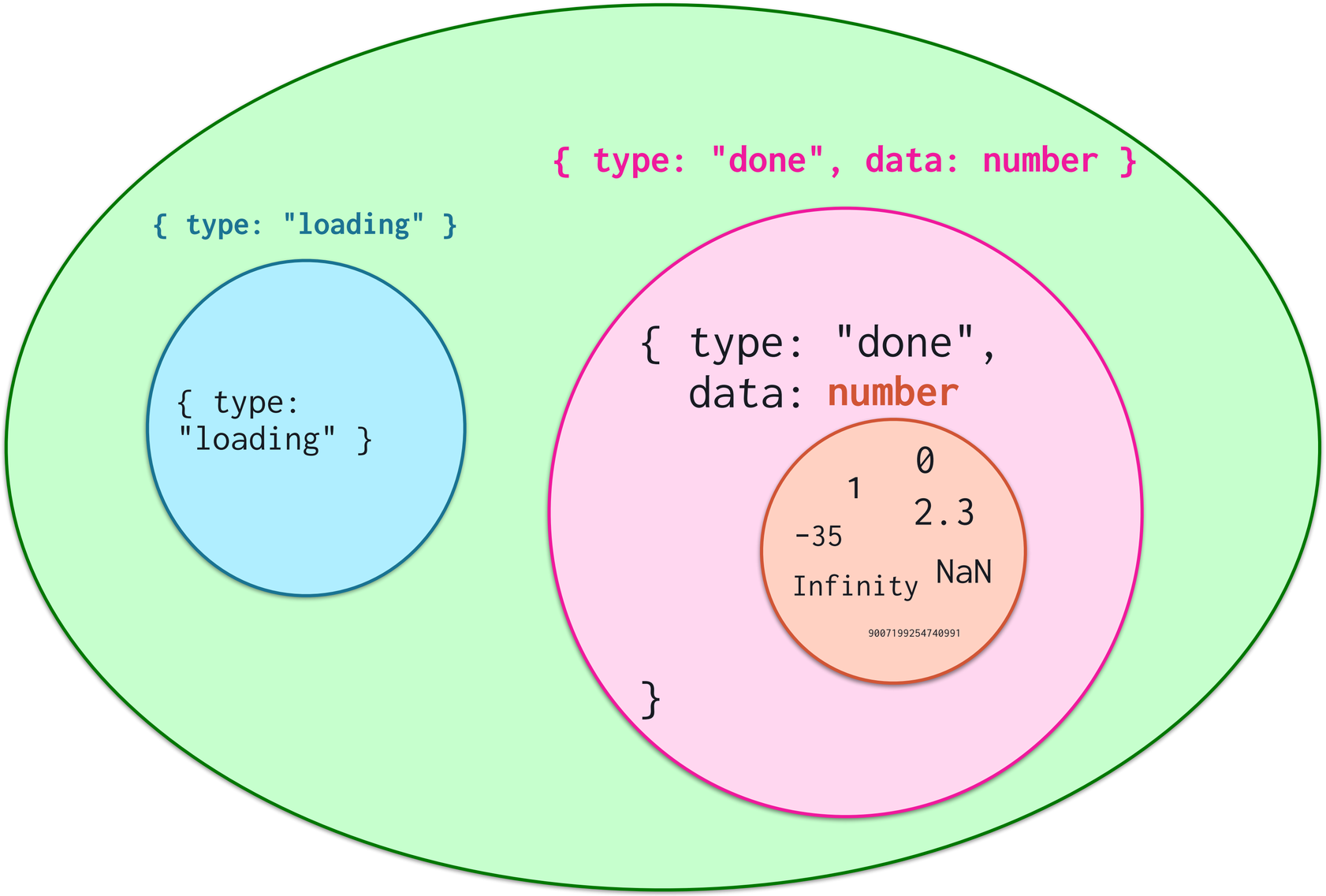


{ type: "done", data: number }




{ type: "loading" }
{ type: "loading" }
{ type: "done", data: number }
{ type: "done",
data: number
0
1
2.3
-35
Infinity
NaN
9007199254740991
}

{ type: "loading" } | { type: "done", data: number }



1.4

When Circles Collide

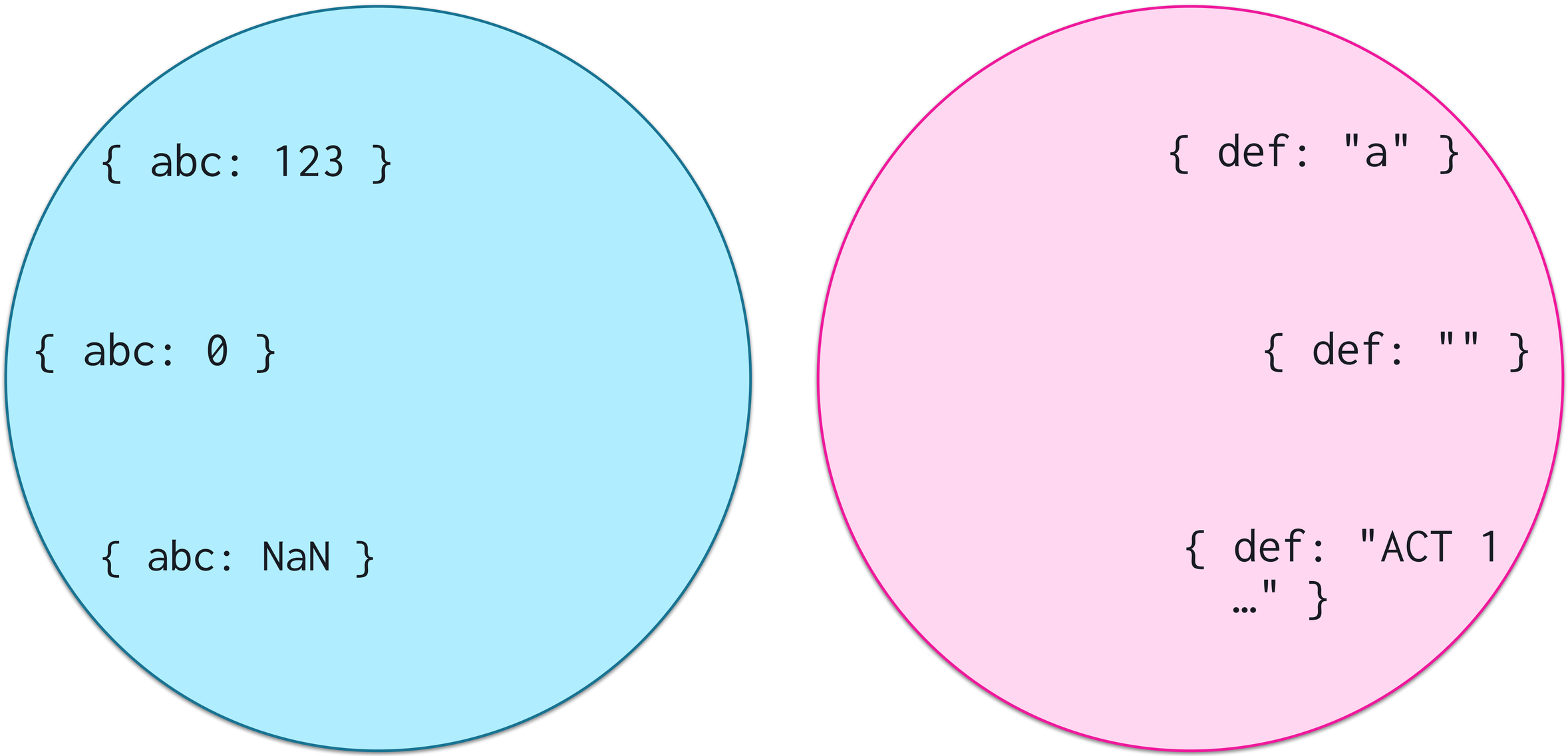
```
type Group =
      { abc: number }
    & { def: string }
```

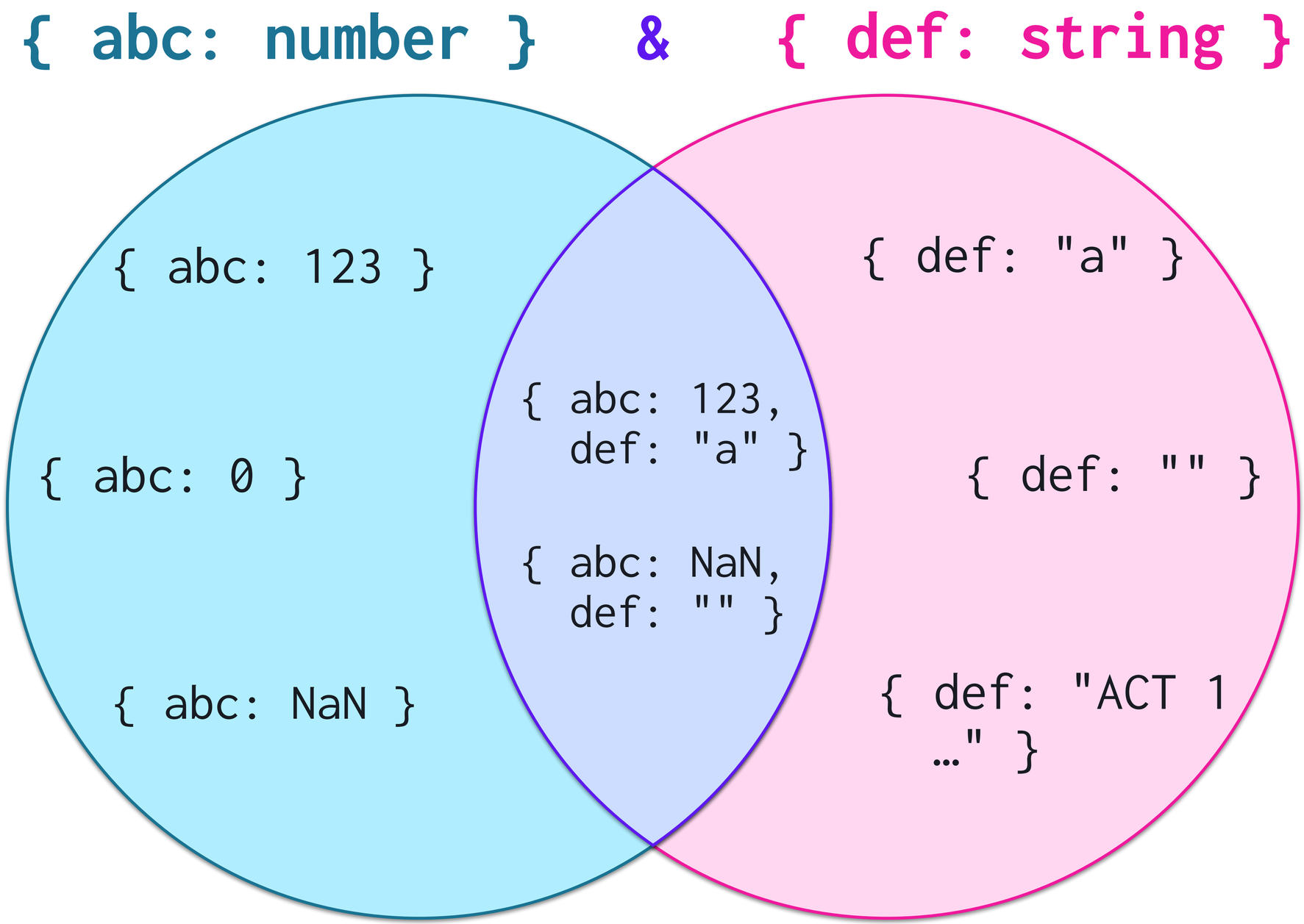
```
type Group =
      { abc: number }
    & { def: string }

const a: Group = { abc: 1,
                   def: "hi" }
```

```
type Group =
      { abc: number }
    & { def: string }

const a: Group = { abc: 1,
                   def: "hi" }
const b: Group = { abc: 1 }
```

```
type Group =
    { abc: number }
  & { def: string }

const a: Group = { abc: 1,
                   def: "hi" }
const b: Group = { abc: 1 }
const c: Group = { def: "hi" }
```

```
type Group =
      { abc: number }
    & { def: string }

const a: Group = { abc: 1,
                   def: "hi" }
const b: Group = { abc: 1 }
const c: Group = { def: "hi" }
```


{ abc: number }
{ abc: 123 }
{ abc: 0 }
{ abc: NaN }
{ def: string }
{ def: "a" }
{ def: "" }
{ def: "ACT 1 …" }


{ abc: number } & { def: string }
{ abc: 123 }
{ abc: 0 }
{ abc: NaN }
{ abc: 123,
def: "a" }
{ abc: NaN,
def: "" }
{ def: "a" }
{ def: "" }
{ def: "ACT 1
…" }

1.5

WTF

```
type Group = never
```

```
type Group = never

const a: Group = 1
```

```
type Group = never

const a: Group = 1
```

```
type Group = never

const a: Group = 1
const b: Group = "hi…?"
```

```
type Group = never

const a: Group = 1
const b: Group = "hi…?"
```

```
type Group = never

const a: Group = 1
const b: Group = "hi…?"
const c: Group = undefined
```

```
type Group = never

const a: Group = 1
const b: Group = "hi…?"
const c: Group = undefined
```



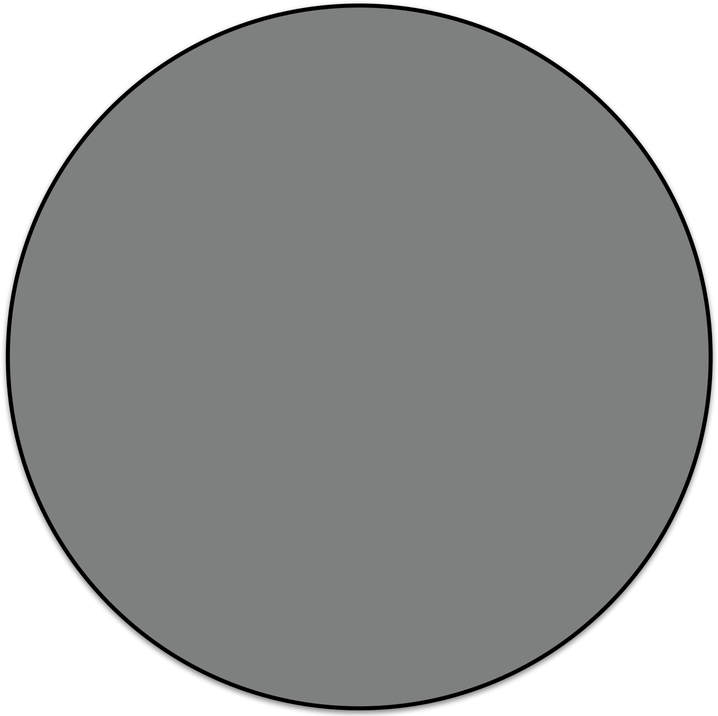
```
type Group = never

const a: Group = 1
const b: Group = "hi…?"
const c: Group = undefined
const d: Group = { abc: 123 }
```



```
type Group = never

const a: Group = 1
const b: Group = "hi…?"
const c: Group = undefined
const d: Group = { abc: 123 }
```




never



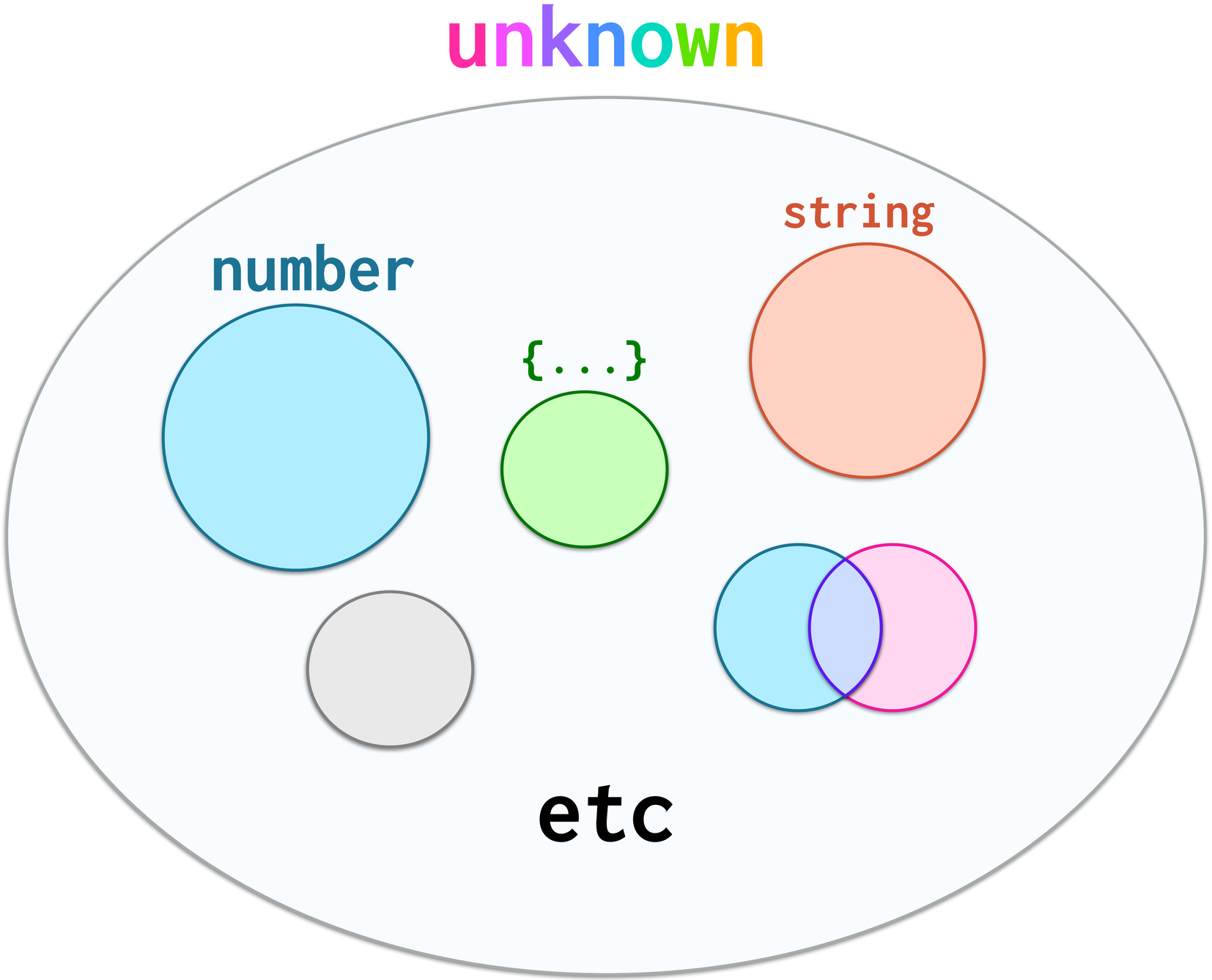
```
type Group = unknown
```

```
type Group = unknown

const a: Group = "hi…?"
```

```
type Group = unknown

const a: Group = "hi…?"
const b: Group = 1
```

```
type Group = unknown

const a: Group = "hi…?"
const b: Group = 1
const c: Group = undefined
```

```
type Group = unknown

const a: Group = "hi…?"
const b: Group = 1
const c: Group = undefined
const d: Group = { abc: 123 }
```


unknown
string
number
{...}
etc

```
type Group = unknown

const a: Group = "hi…?"
const b: Group = 1
const c: Group = undefined
const d: Group = { abc: 123 }
```

```
type Group = unknown

const a: Group = "hi…?"
const b: Group = 1
const c: Group = undefined
const d: Group = { abc: 123 }

console.log(a.length)
```

```
type Group = unknown

const a: Group = "hi…?"
const b: Group = 1
const c: Group = undefined
const d: Group = { abc: 123 }

console.log(a.length)
```


```
const a: unknown = "hi…?"
```

```
const a: unknown = "hi…?"

if (typeof a === 'string') {
  console.log(a.length);
}
```

```
const a: unknown = "hi…?"

if (typeof a === 'string') {
  console.log(a.length);
}
```

```
type Group = any

const a: Group = "hi…?"
const b: Group = 1
const c: Group = undefined
const d: Group = { abc: 123 }
```

any

any

```
type Group = any

const a: Group = 1
const b: Group = "hi…?"
const c: Group = undefined
const d: Group = { abc: 123 }

console.log(d.iWannaPony())
```

2.

Possibilities

2.1

Shrink types to only allow what you expect

```
function iconClass(type: string): string {
   ...
}
```

```
// CSS class name for icon
function iconClass(type: string): string {
  return `icon icon-${type}`;
}
```

```
// CSS class name for icon
function iconClass(type: string): string {
  return `icon icon-${type}`;
}
```

```
type Icon = 'spinner' | 'save';

function iconClass(type: Icon): string {
   ...
}
```

```
type Icon = 'spinner' | 'save';

function iconClass(type: Icon): string {
   ...
}
```

string

""

"spinner"

"Hello, world!"

"ACT I
SCENE I. Athens. The palace of
THESEUS.

Enter THESEUS, HIPPOLYTA,
PHILOSTRATE, and Attendants

..."

"冷蔵庫"

```
"spinner" | "save"
```

```
"spinner" | "save"
```

```
"spinner"
```

```
"spinner" | "save"
```

```
"spinner"
```

```
"save"
```

Possibilities:
lim n -> ∞...?

Possibilities:
2

2.2

Don't spread a choice across multiple values

```
type Props = {
  success?: boolean,
  warning?: boolean,
  error?:   boolean,
  ...
};

function Notification(props: Props): {
  ...
}
```

```
<Notification
  error={true}
>
  This is bad.
</Notification>
```

```
Notification({
   error: true,
   children: [
      "This is bad.",
   ]
})
```

```
<Notification
  error={true}
>
  This is bad.
</Notification>
```

```
<Notification
  error={true}
>
  This is bad.
</Notification>
```

```
<Notification
   success={true}
>
   This is good.
</Notification>
```

```
<Notification
   success={true}
   error={true}
>
   This is ... ???
</Notification>
```

```
<Notification
   success={true}
   error={true}
>
   This is ... ???
</Notification>
```

```
type Props = {
  success?: boolean,
  warning?: boolean,
  error?:   boolean,
  ...
};

function Notification(props: Props): {
  ...
}
```

```
type Props = {
  success?: boolean,
  warning?: boolean,
  error?:   boolean,
  ...
};

function Notification(props: Props): {
  ...
}
```

```
type Props = {
  type: 'success' | 'warning' | 'error',
  ...
};



function Notification(props: Props): {
  ...
}
```

```
<Notification
  type="success"
>
  This is good.
</Notification>
```

```
<Notification
  type="success"
>
  This is good.
</Notification>
```

Possibilities:
(true,true,true + true,true,false + true,false,false + ...)
8

Possibilities:
3

2.3

Things that change separately are totally fine

```
type Props = {
  type: 'success' | 'warning' | 'error',
  dismissable: boolean,
};



function Notification(props: Props): {
  ...
}
```

```
type Props = {
  type: 'success' | 'warning' | 'error',
  dismissable: boolean,
};



function Notification(props: Props): {
  ...
}
```

2.4

Convenient types are tempting but can leave gaps.

```
type Input = {
  items: Array<Item>,
};



function Carousel(props: Input): {
  const items = useState<Input>(props);
  ...
}
```

```
<Carousel
  items=[...]
/>
```

```
<Carousel
  items=[]
/>
```

```
type Input = {
  items: Array<Item>,
};



function Carousel(props: Input): {
  const items = useState<Input>(props);
  ...
}
```

```
type Input = {
  itemsBefore: Array<Item>
  currentItem: Item,
  itemsAfter: Array<Item>,
};

function Carousel(props: Input): {
  const items = useState<Input>(props);
  ...
}
```

```
type Input = {
  itemsBefore: Array<Item>
  currentItem: Item,
  itemsAfter: Array<Item>,
};

function Carousel(props: Input): {
  const items = useState<Input>(props);
  ...
}
```

```
<Carousel
   itemsBefore=[]
   item=...
   itemsAfter=[]
/>
```

```
<Carousel
   itemsBefore=[]
   item=...
   itemsAfter=[]
/>
```

2.5

Combining Unions, and values that change separately

```
function Example(props: Props): {
  const [isLoading, setLoading] =
    useState(false);
  const [isError, setError] =
    useState(false);
  const [data, setData] =
    useState<string | null>(null);


  ...
}
```

```
type State = {
  isLoading: boolean,
  isError: boolean,
  data: null | string,
}
```

```
type State = {
  isLoading: boolean,
  isError: boolean,
  data: null | string,
}
```

```
isLoading: true,
isError:   false,
data:      null,
```

```
type State = {
  isLoading: boolean,
  isError: boolean,
  data: null | string,
}
```

```
isLoading: true,
isError:   false,
data:      null,
```

```
type State = {
  isLoading: boolean,
  isError: boolean,
  data: null | string,
}
```

```
  isLoading: false,
  isError:   false,
  data: "yay data",
```

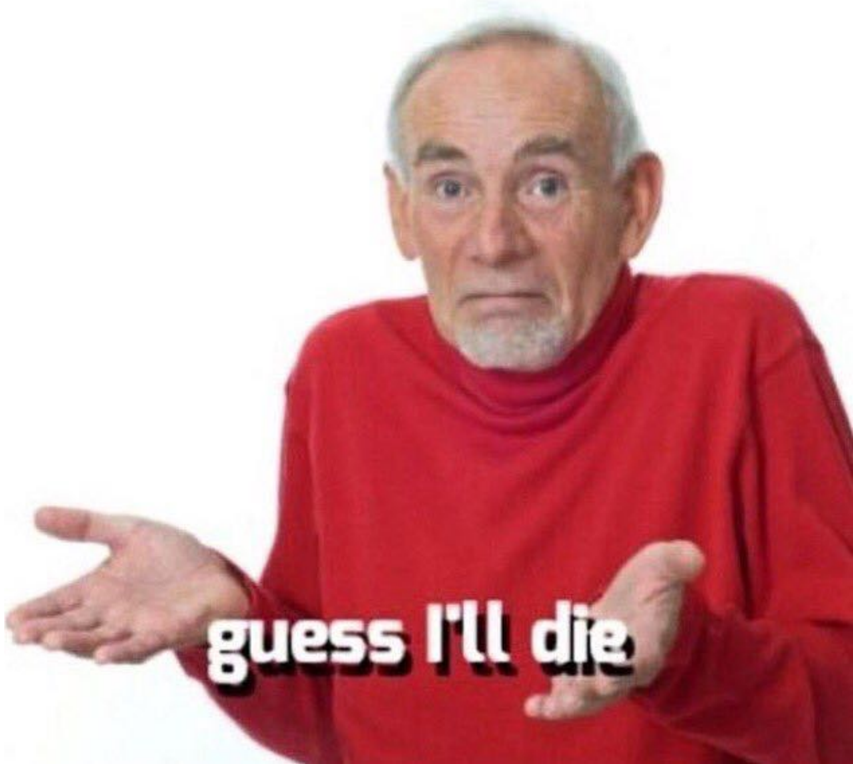
```
type State = {
  isLoading: boolean,
  isError: boolean,
  data: null | string,
}
```

```
isLoading: false,
isError:   true,
data:      null,
```

```
type State = {
  isLoading: boolean,  isLoading: true,
  isError: boolean,    isError:   true,
  data: null | string, data: "yay data",
}
```

```
type State = {
   isLoading: boolean,
   isError: boolean,
   data: null | string,
}
```

```
isLoading: true,
isError:   true,
data: "yay data",
```



```
type State = {
  isLoading: boolean,
  isError: boolean,
  data: null | string,
}
```

```
type State =
  | { type: 'loading' }
  | { type: 'error' }
  | { type: 'success',
      data: string
    }
```

Make Invalid States Impossible

https://kentcdodds.com/blog/make-impossible-states-impossible

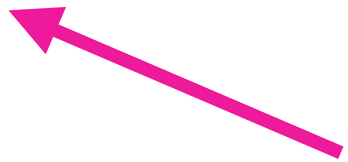
3.

Exhaustivity

```
type Icon = 'spinner' | 'save';

function icon(type: Icon): string {
  ...
}
```

```
type Icon = 'spinner' | 'save';

function icon(type: Icon): string {
  switch (type) {
    case 'spinner': ...
    case 'save': ...
  }
}
```

```
type Icon = 'spinner' | 'save';

function icon(type: Icon): string {
  switch (type) {
    case 'spinner': ...
    case 'save': ...
  }
}
```



Important: No `default:` here.

```
type Icon = 'spinner' | 'save';

function icon(type: Icon): string {
  switch (type) {
    case 'spinner': ...
    case 'save': ...
  }
}
```

```
type Icon = 'spinner' | 'save' | 'ok';

function icon(type: Icon): string {
  switch (type) {
    case 'spinner': ...
    case 'save': ...
  }
}
```

```
type Icon = 'spinner' | 'save' | 'ok';

function icon(type: Icon): string {
  switch (type) {
    case 'spinner': ...
    case 'save': ...
  }
}
```

```
type Icon = 'spinner' | 'save' | 'ok';

function icon(type: Icon): string {
  switch (type) {
    case 'spinner': ...
    case 'save': ...
  }
}
```

```
Function lacks ending
return statement and
return type does not
include 'undefined'.
```

```
type Icon = 'spinner' | 'save' | 'ok';

function icon(type: Icon): string {
  switch (type) {
    case 'spinner': ...
    case 'save': ...
    case 'ok': ...
  }
}
```

```
type Icon = 'spinner' | 'save' | 'ok';

function icon(type: Icon): string {
  switch (type) {
    case 'spinner': ...
    case 'save': ...
    case 'ok': ...
  }
}
```

```
type Icon = 'spinner' | 'save' | 'ok';

function icon(type: Icon): string {
  switch (type) {
    case 'spinner': ...
    case 'save': ...
    case 'ok': ...
  }
}
```

If we'd had a `default:` here, TypeScript would <u>not</u> have complained.

The new `'ok'` Icon would have been swallowed up by the `default:` case, depriving us of a useful to-do item.


Dear Kate,
Here's to the crazy ones. The misfits.
The troublemakers. The round pegs in the
holes. The ones who see things differently.
not fond of rules. And they have no respect for the
status quo. You can quote them, disagree with them,
glorify or vilify them. About the only thing you can't
do is ignore them. Because they change things.
They push the human race forward. And while some
may see them as the crazy ones, we see genius.
Because the people who are crazy enough to think
they can change the world, are the ones who do.
Take care.
John Appleseed

```
type Example = boolean | string;


function example(x: Example): number {
  switch (typeof x) {
    case 'boolean':
      ...
    case 'string':
      ...
  }
}
```

Exhaustivity

https://www.typescriptlang.org/docs/handbook/advanced-types.html#exhaustiveness-checking

(PS: `never` shows up here.)

4.

Intentionally–Different Types

If you mean something different, use a different type.

number

Row ID

Kilogram

Count

number

Length (cm)

Currency (AUD)

Currency (JP Yen)

Row ID

Works of Shakespeare

Name

`string`

Gender

Phone Number

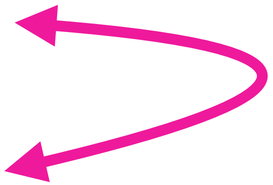
Email

```
async function saveUser(
  id:    string,
  email: string
): Promise<User> {
  ...
}
// ...
await saveUser(id, "a@b.c");
```

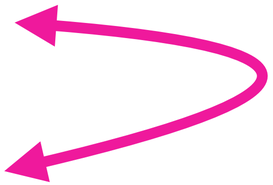
```
async function saveUser(
  id:    string,
  email: string
): Promise<User> {
  ...
}
// ...
await saveUser(id, "a@b.c");
```

```
async function saveUser(
  id:    string,
  email: string
): Promise<User> {
  ...
}
// ...
await saveUser(id, "a@b.c");
```

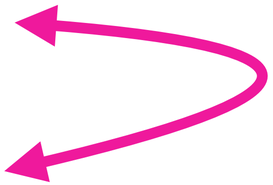
```
async function saveUser(
  id:    string,
  email: string
): Promise<User> {
  ...
}
// ...
await saveUser(id, "a@b.c");
```



```
async function saveUser(
  id:    string,
  email: string
): Promise<User> {
  ...
}
// ...
await saveUser("a@b.c", id);
```



```
async function saveUser(
  id:    string,
  email: string
): Promise<User> {
  ...
}
// ...
await saveUser(id, "a@b.c");
```



Method #1: Context

```
async function saveUser(
  id:    string,
  email: string
): Promise<User> {
  ...
}
// ...
await saveUser(id, "a@b.c");
```

```
async function saveUser({ id, email }:{
   id:    string,
   email: string
}): Promise<User> {
   ...
}
// ...
await saveUser({ id, email: "a@b.c"});
```

```
async function saveUser({ id, email }:{
   id:    string,
   email: string
}): Promise<User> {
   ...
}
// ...
await saveUser({ id, email: "a@b.c"});
```

```ts
type User = {
  id:    string,
  email: string,
};

async function saveUser(
  { id, email }: User
): Promise<User> {
  ...
}
```

```ts
interface User {
  id:    string;
  email: string;
}

async function saveUser(
  { id, email }: User
): Promise<User> {
  ...
}
```

```ts
type User = {
  id:    string,
  email: string,
};

async function saveUser(
  { id, email }: User
): Promise<User> {
  ...
}
```

```
type User = {
  id:    string,
  email: string,
};

async function saveUser(
  { id, email }: User
): Promise<User> {
  ...
}
```

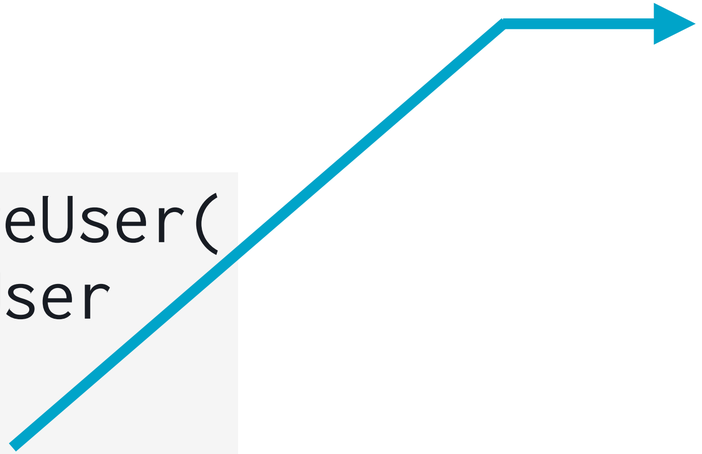
```
async function saveUser(
  { id, email }: User
): Promise<User> {
  ...
}
```

```
async function saveUser(
  { id, email }: User
): Promise<User> {
  ...
}
```

```
function checkId(string) {
  ...
}
```

```
function validEmail(string) {
  ...
}
```

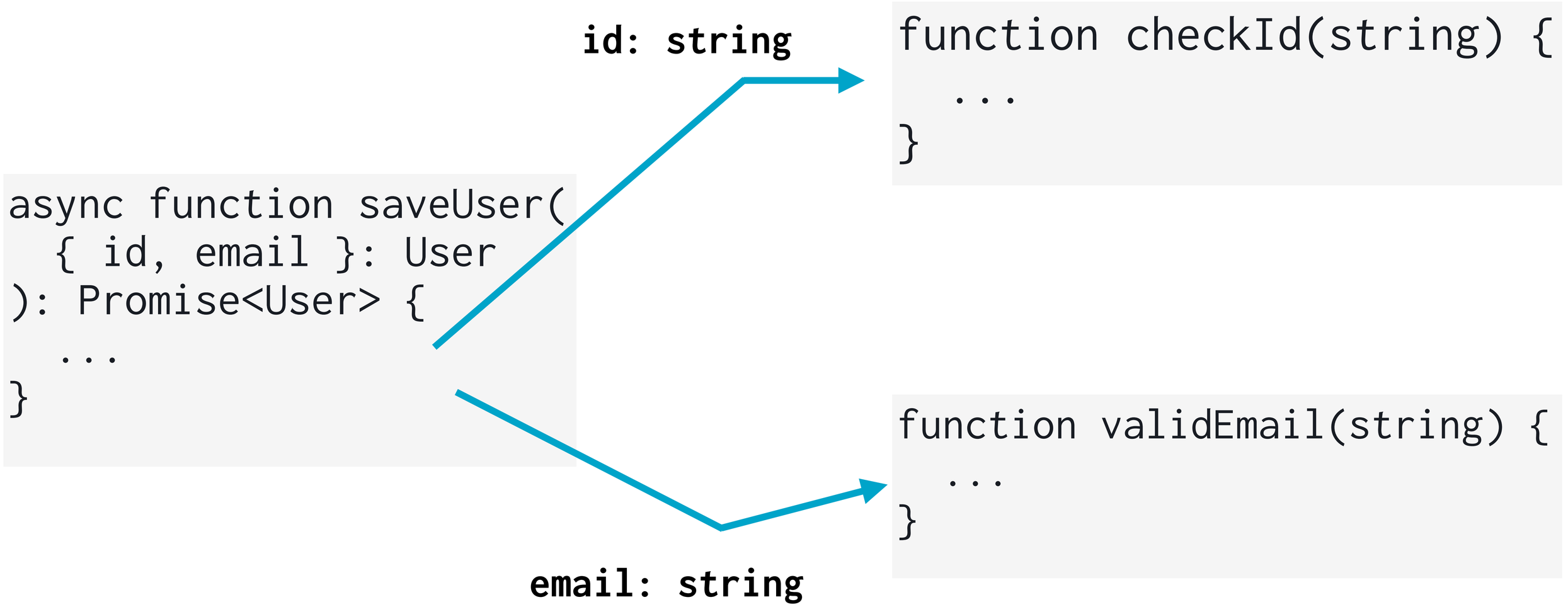
id: string

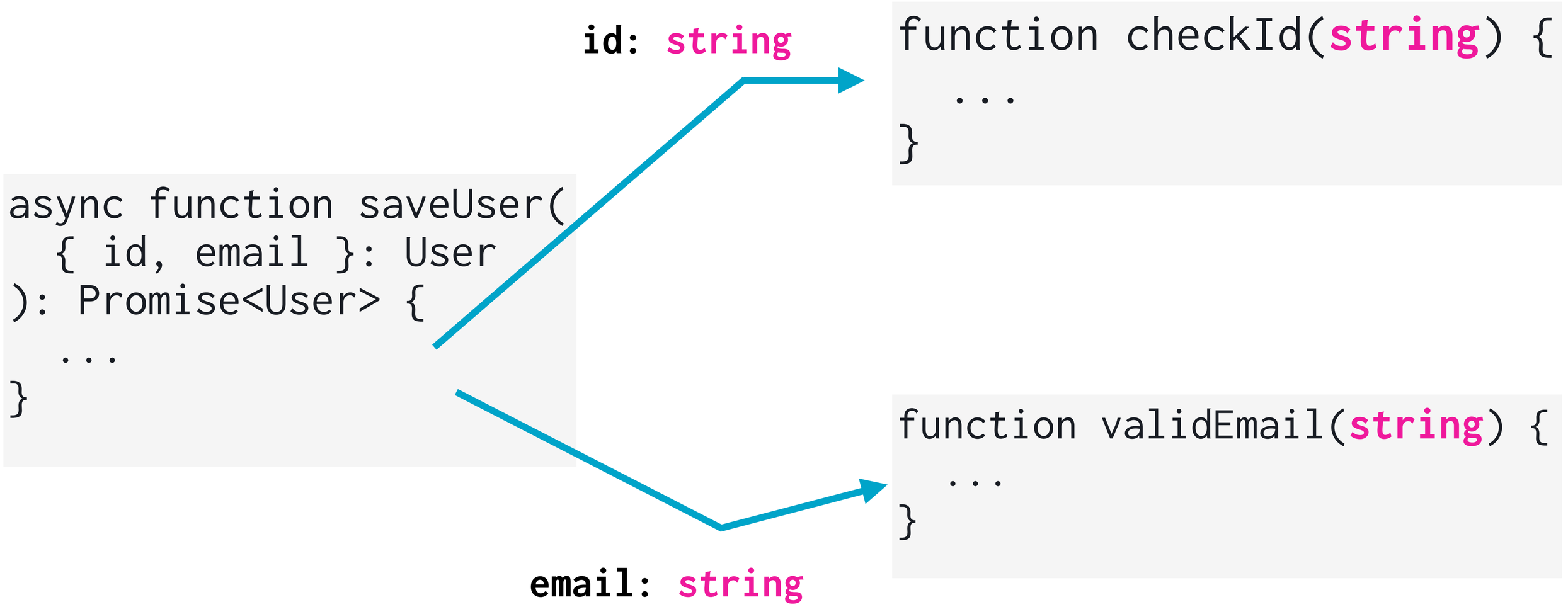


```
async function saveUser(
  { id, email }: User
): Promise<User> {
  ...
}
```

```
function checkId(string) {
  ...
}
```

```
function validEmail(string) {
  ...
}
```


id: string
function checkId(string) {
...
}
async function saveUser(
{ id, email }: User
): Promise<User> {
...
}
function validEmail(string) {
...
}
email: string


id: string
email: string
async function saveUser(
{ id, email }: User
): Promise<User> {
...
}
function checkId(string) {
...
}
function validEmail(string) {
...
}

Method #2: Brands

(Opaque/nominal types)

```
type Brand<WrappedType, TypeName> =
  WrappedType & { __brand: TypeName };
```

```
type Brand<WrappedType, TypeName> =
  WrappedType & { __brand: TypeName };
```

```
type Brand<WrappedType, TypeName> =
  WrappedType & { __brand: TypeName };
```

```
type Brand<WrappedType, TypeName> =
  WrappedType & { __brand: TypeName };
```

```
type Brand<WrappedType, TypeName> =
  WrappedType & { __brand: TypeName };

type UserId = Brand<string, 'UserId'>;
```

```
type Brand<WrappedType, TypeName> =
  WrappedType & { __brand: TypeName };

type UserId = Brand<string, 'UserId'>;

const id = '1234-abcd' as UserId;
```

```
type Brand<WrappedType, TypeName> =
  WrappedType & { __brand: TypeName };

type UserId = Brand<string, 'UserId'>;

const id = '1234-abcd' as UserId;
const email = 'a@b.c';
```

```ts
type Brand<WrappedType, TypeName> =
  WrappedType & { __brand: TypeName };

type UserId = Brand<string, 'UserId'>;

const id = '1234-abcd' as UserId;
const email = 'a@b.c';

expectsUserId(id);
expectsUserId(email);
```

```ts
type Brand<WrappedType, TypeName> =
  WrappedType & { __brand: TypeName };

type UserId = Brand<string, 'UserId'>;

const id = '1234-abcd' as UserId;
const email = 'a@b.c';

expectsUserId(id);    // ✅
expectsUserId(email);
```

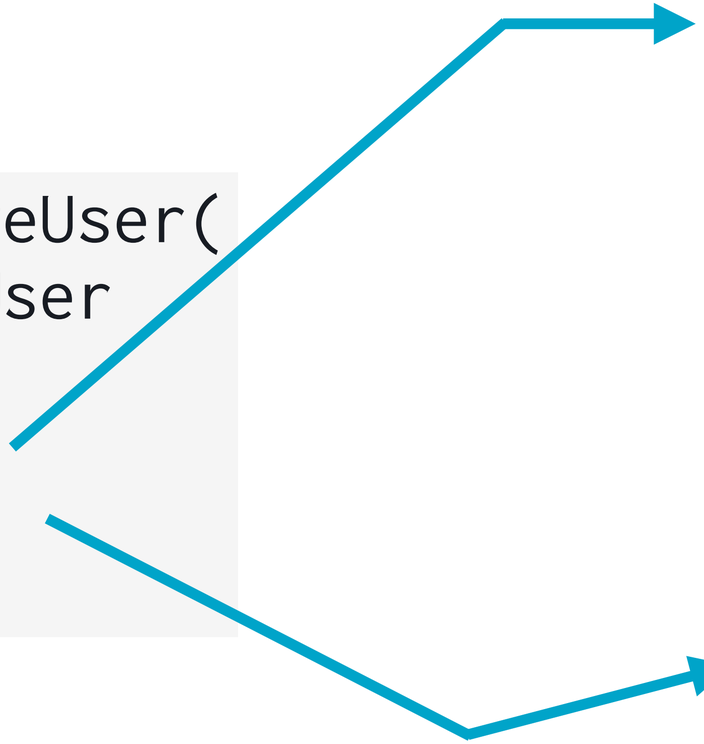
```ts
type Brand<WrappedType, TypeName> =
  WrappedType & { __brand: TypeName };

type UserId = Brand<string, 'UserId'>;

const id = '1234-abcd' as UserId;
const email = 'a@b.c';

expectsUserId(id);    // ✅
expectsUserId(email); // ❌
```

```
stringToUserId(id: string): UserId | null;

userIdToString(id: UserId): string
```

```
type User = {
  id:    UserId,
  email: string,
};

async function saveUser(
  { id, email }: User
): Promise<User> {
  ...
}
```

id: UserId

```
async function saveUser(
  { id, email }: User
): Promise<User> {
  ...
}
```



```
function checkId(UserId) {
  ...
}
```

```
function validEmail(string) {
  ...
}
```

email: string

Brands

https://spin.atomicobject.com/2018/01/15/typescript-flexible-nominal-typing/

5.

Proving things to the type-checker

```
const a: unknown = "hi…?"

if (typeof a === 'string') {
  console.log(d.length);
}
```

unknown → known

```
function messageFromApi(
  raw: unknown
): { message: string } {
  if (
    typeof raw == 'object'
    && raw != ... && ...
  ) {
    return ...;
  }
  throw new Error('Not valid: ...').
}
```

```
const a: unknown = "hi…?"

if (typeof a === 'string') {
  console.log(a.length);
}
```

```
const a: unknown = "hi…?"

if (isString(a)) {
  console.log(a.length);
}
```

```
const a: unknown = "hi…?"

if (isString(a)) {
  console.log(a.length);
}

function isString(v: unknown): v is string {
  return (typeof value === 'string');
}
```

```
const a: unknown = "hi…?"

if (isString(a)) {
  console.log(a.length);
}

function isString(v: unknown): v is string {
  return (typeof value === 'string');
}
```

```
function userFromApi(
  raw: unknown
): { id: string, email: string } {
  if (typeof raw === 'object' && raw != null) {
    const { id, email } =
      (raw as { id: unknown, email: unknown });
    if (
      typeof id === 'string'
      && typeof email === 'string'
    ) {
      return { id, email };
    }
  }
  throw new Error('Not valid: ...');
}
```

```
function userFromApi(
  raw: unknown
): { id: string, email: string } {
  if (isObject(raw)) {
    const { id, email } =
      hasKeys(raw, ['id', 'email']);
    if (isString(id) && isString(email)) {
      return { id, email };
    }
  }
  throw new Error('Not valid: ...').
}
```

```ts
import * as t from 'io-ts';

const User = t.type({
  id: t.string,
  email: t.string,
});
type User = t.TypeOf<typeof User>; // { id: string, … }

// Valid input:
User.decode(JSON.parse('{"id":1,"email":"g@z.com"}'));
  // => Right({ id: 1, name: "g@z.com" })

// Invalid input:
User.decode(JSON.parse('{"email":"g@z.com"}'));
  // => Left([...])
```

```
type NonEmptyArray<Thing> =
  Brand<string, 'UserId'>;
```

```
type NonEmptyArray<Thing> =
  Brand<string, 'UserId'>;

makeNonEmptyArray<Thing>(
  mayBeEmpty: Array<Thing>
): NonEmptyArray<Thing> | null;
```

Proving to TypeScript you've validated input

https://mariusschulz.com/blog/typescript-3-0-the-unknown-type

https://gcanti.github.io/io-ts/

Type Guards

https://medium.com/@wittydeveloper/typescript-make-types-real-the-type-guard-functions-814364e8dbe3

6.

Lightning-Round of Odd Tips

--strict

Avoid numeric
enum

```
enum MyEnum {
   a = 1,
   b = 2,
}
function fun(en: MyEnum) {
   // ...
}

fun(666); // no error
```

https://twitter.com/GiulioCanti/status/1105873882882412545

(But *string* enums are
fortunately unaffected by this.
They're weird in other ways,
but are safe to use.)

readonly & ReadOnly<...>

https://www.typescriptlang.org/docs/handbook/utility-types.html

```ts
type User = {
  id: string, name: string, age: number,
};

// NewUser has everything except
// the 'id' field from User:
type NewUser =
  Exclude<User, 'id'>;
```

https://www.typescriptlang.org/docs/handbook/utility-types.html

`type` **vs** `interface`

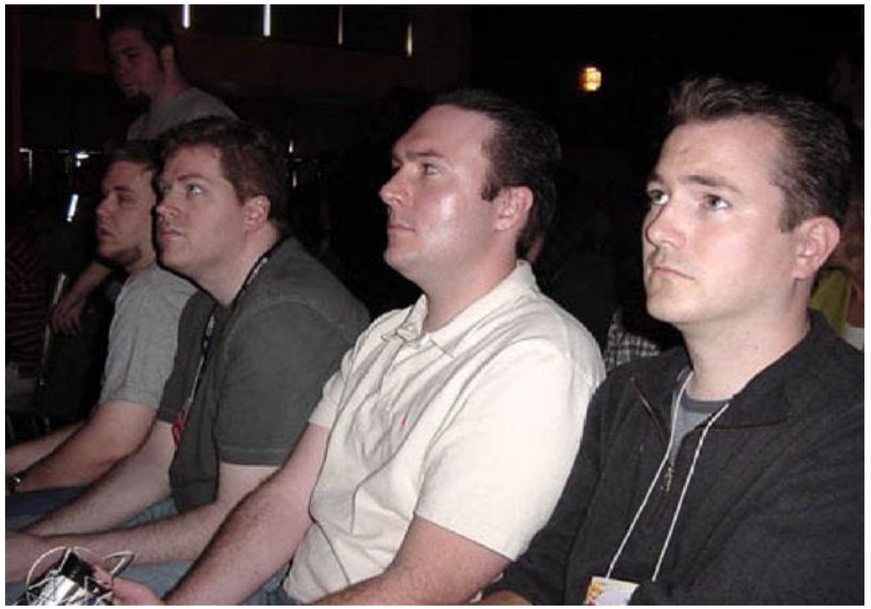
https://medium.com/@martin_hotell/interface-vs-type-alias-in-typescript-2-7-2a8f1777af4c

`interface`

`type`

interface

type



`interface`

`type`

"Literal Type Widening"

https://mariusschulz.com/blog/literal-type-widening-in-typescript

```
interface Point {
  x: number;
  y: number;
};
```

```
interface Point {
  x: number;
  y: number;
};

const xyz1: Point = { x: 5, y: 5, z: 5 };
```

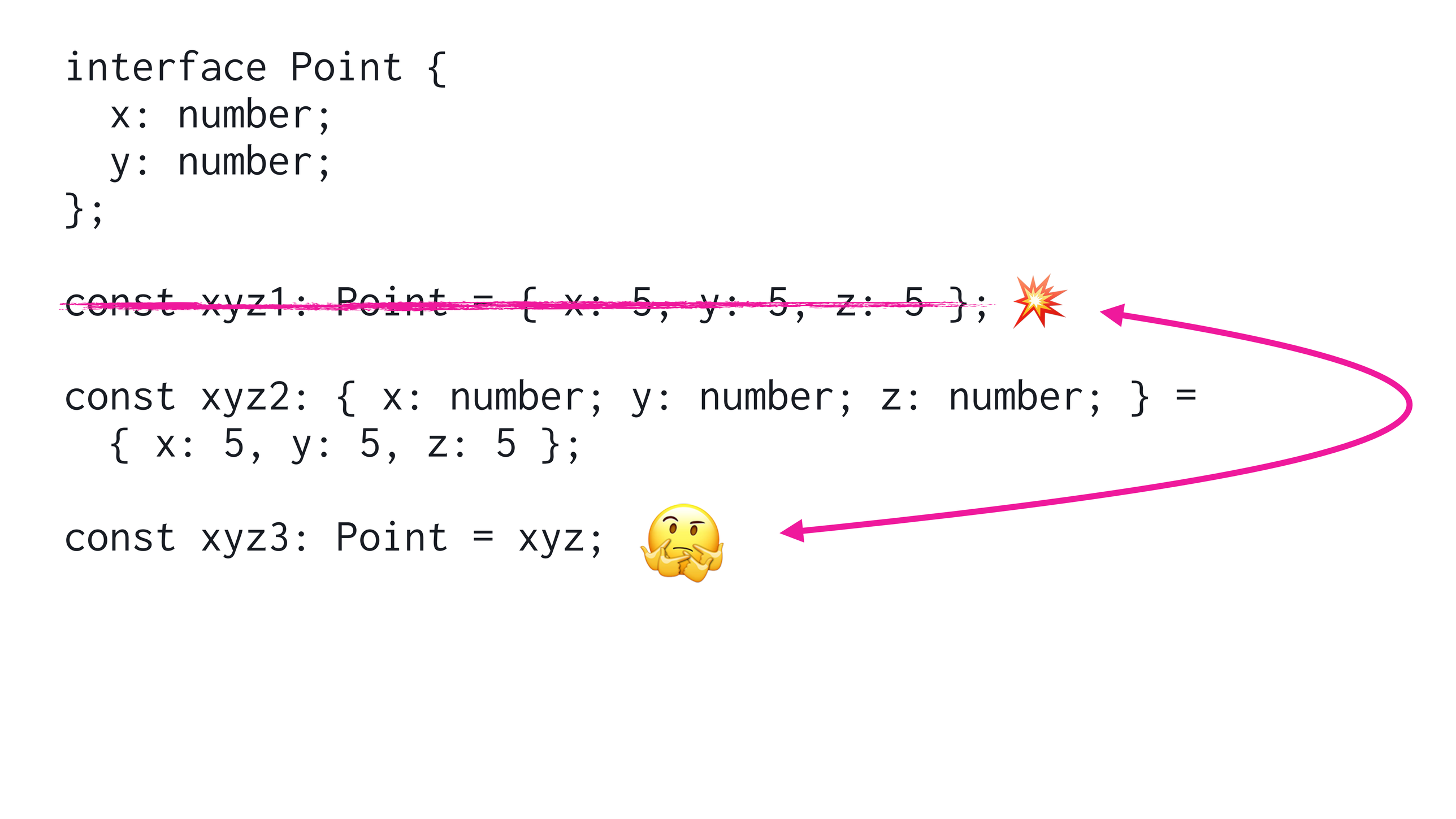
```ts
interface Point {
  x: number;
  y: number;
};

const xyz1: Point = { x: 5, y: 5, z: 5 };
```

```ts
interface Point {
  x: number;
  y: number;
};

const xyz1: Point = { x: 5, y: 5, z: 5 };

const xyz2: { x: number; y: number; z: number; } =
  { x: 5, y: 5, z: 5 };
```

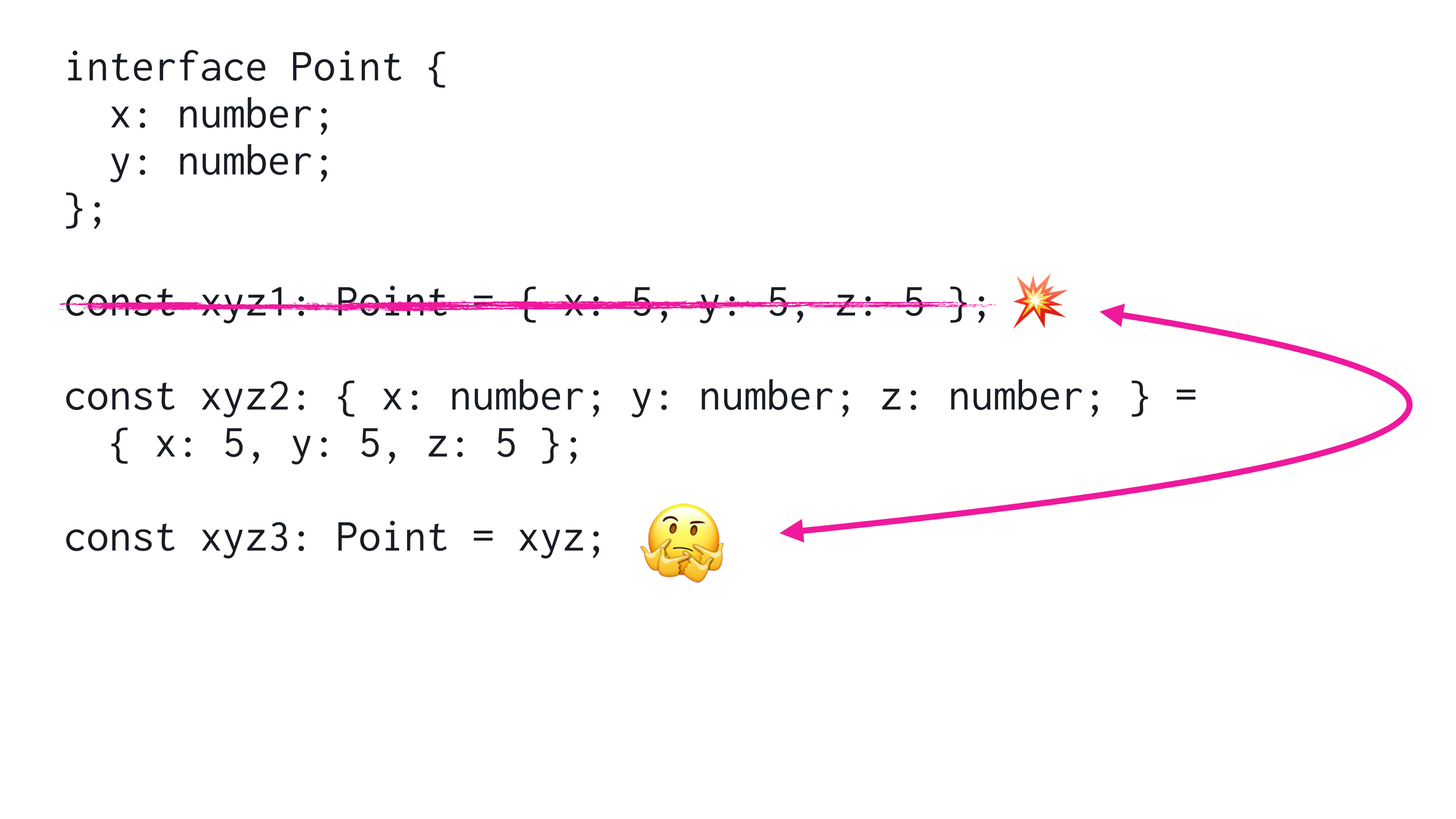
```ts
interface Point {
  x: number;
  y: number;
};

const xyz1: Point = { x: 5, y: 5, z: 5 };

const xyz2: { x: number; y: number; z: number; } =
  { x: 5, y: 5, z: 5 };

const xyz3: Point = xyz;
```

```ts
interface Point {
  x: number;
  y: number;
};

const xyz1: Point = { x: 5, y: 5, z: 5 };

const xyz2: { x: number; y: number; z: number; } =
  { x: 5, y: 5, z: 5 };

const xyz3: Point = xyz;
```

```ts
interface Point {
  x: number;
  y: number;
};

const xyz1: Point = { x: 5, y: 5, z: 5 };

const xyz2: { x: number; y: number; z: number; } =
  { x: 5, y: 5, z: 5 };

const xyz3: Point = xyz;
```

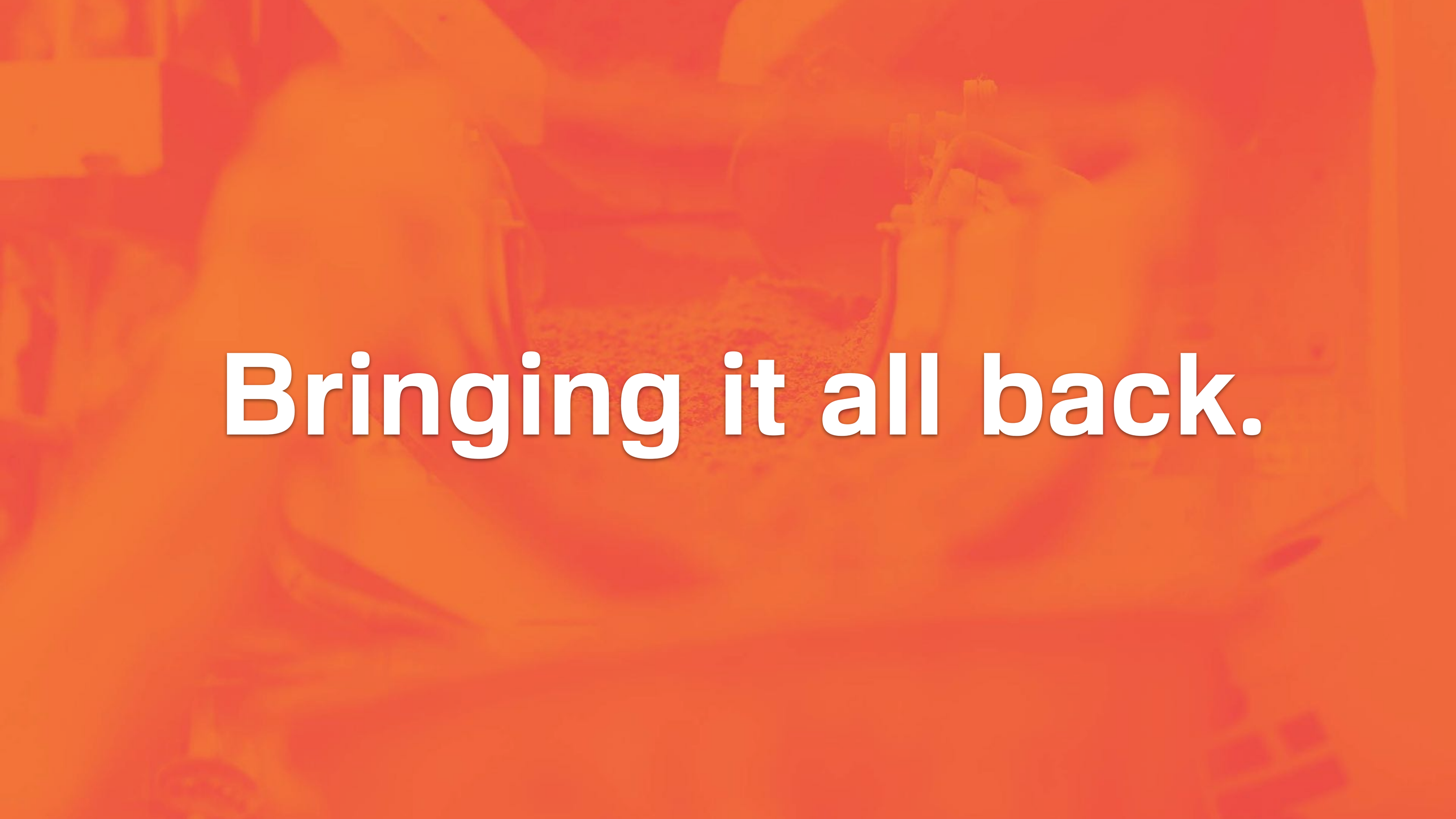


```ts
interface Point {
  x: number;
  y: number;
};

const xyz1: Point = { x: 5, y: 5, z: 5 };

const xyz2: { x: number; y: number; z: number; } =
  { x: 5, y: 5, z: 5 };

const xyz3: Point = xyz;
```



```
interface Point {
  x: number;
  y: number;
};

const xyz1: Point = { x: 5, y: 5, z: 5 }; 💥

const xyz2: { x: number; y: number; z: number; } =
  { x: 5, y: 5, z: 5 };

const xyz3: Point = xyz; 🤔

// and also
const a: Point = Object.assign({}, {x: 1, y: 2, z: 3});
```

```ts
interface Point {
  x: number;
  y: number;
};

const xyz1: Point = { x: 5, y: 5, z: 5 };

const xyz2: { x: number; y: number; z: number; } =
  { x: 5, y: 5, z: 5 };

const xyz3: Point = xyz;

// and also
const a: Point = Object.assign({}, {x: 1, y: 2, z: 3});
const b = <Point>{ x: 5, y: 5, z: 5 };
```


Bringing it all back.

- Shrink the possibilities for your input and output to as closely match what's going on in your head as you can muster.

- Shrink the possibilities for your input and output to as closely match what's going on in your head as you can muster.
- Use exhaustivity to give yourself a to-do list when you change code.

- Shrink the possibilities for your input and output to as closely match what's going on in your head as you can muster.
- Use exhaustivity to give yourself a to-do list when you change code.
- Use different types when you mean different things.

- Shrink the possibilities for your input and output to as closely match what's going on in your head as you can muster.
- Use exhaustivity to give yourself a to-do list when you change code.
- Use different types when you mean different things.
- 'Prove' things to the type checker so that it can do work for you.

- Shrink the possibilities for your input and output to as closely match what's going on in your head as you can muster.
- Use exhaustivity to give yourself a to-do list when you change code.
- Use different types when you mean different things.
- 'Prove' things to the type checker so that it can do work for you.
- don't use numeric enums, please

Better TypeScript Types

Rob Howard
@damncabbage
http://robhoward.id.au

- https://kentcdodds.com/blog/make-impossible-states-impossible
- https://www.typescriptlang.org/docs/handbook/advanced-types.html#exhaustiveness-checking
- https://michalzalecki.com/nominal-typing-in-typescript/
- https://spin.atomicobject.com/2018/01/15/typescript-flexible-nominal-typing/
- https://www.typescriptlang.org/docs/handbook/utility-types.html
- https://dev.to/busypeoples/notes-on-typescript-pick-exclude-and-higher-order-components-40cp
- https://medium.com/@martin_hotell/interface-vs-type-alias-in-typescript-2-7-2a8f1777af4c
- https://mariusschulz.com/blog/typescript-3-0-the-unknown-type
- https://gcanti.github.io/io-ts/
- https://basarat.gitbooks.io/typescript/