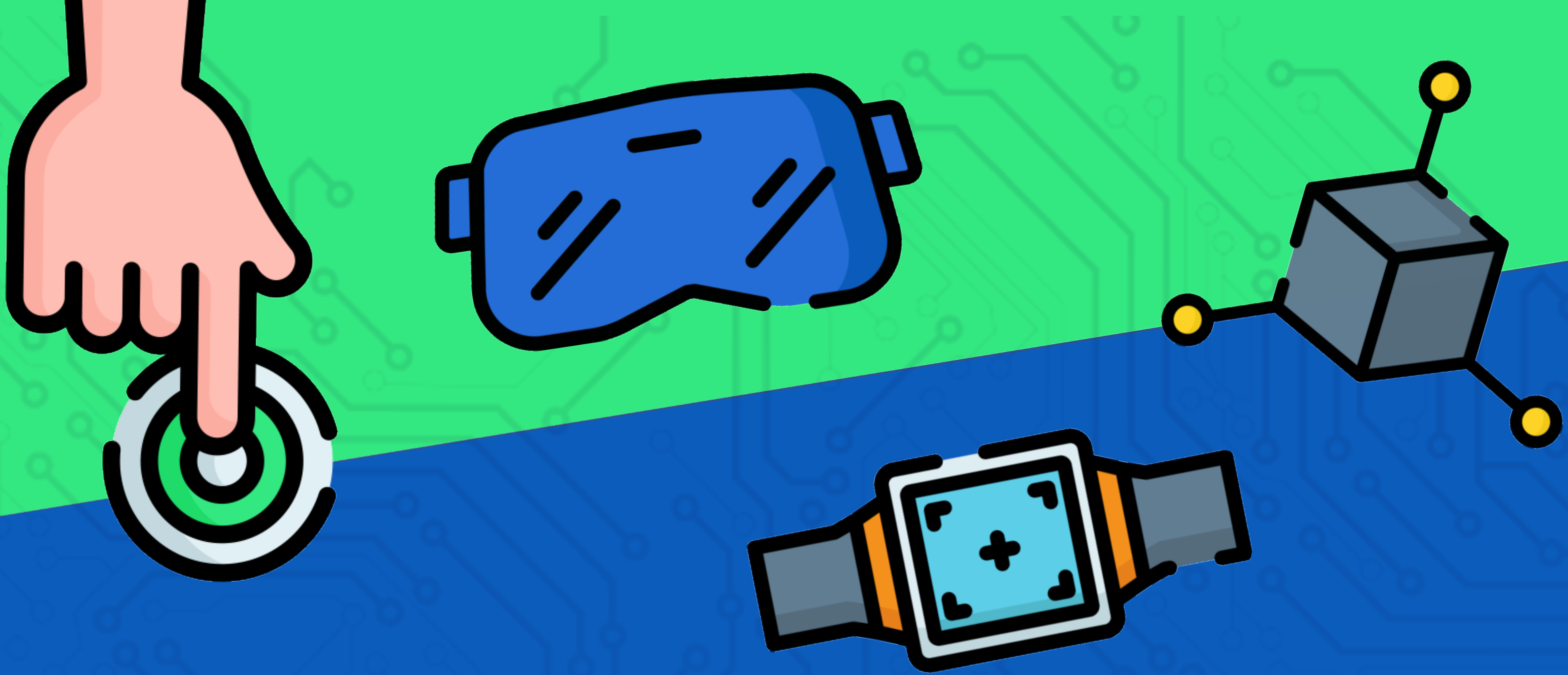
CLIENT
TECHNOLOGY
DAYS



It’s a Type of Magic

Yacine Mekesser

#CTD2019

TypeScript

Just JavaScript with type annotations?

TypeScript

- Generics
- Literal Types
- Intersection types
- Union Types
- Mapped Types
- Conditional Types

Typescript Config

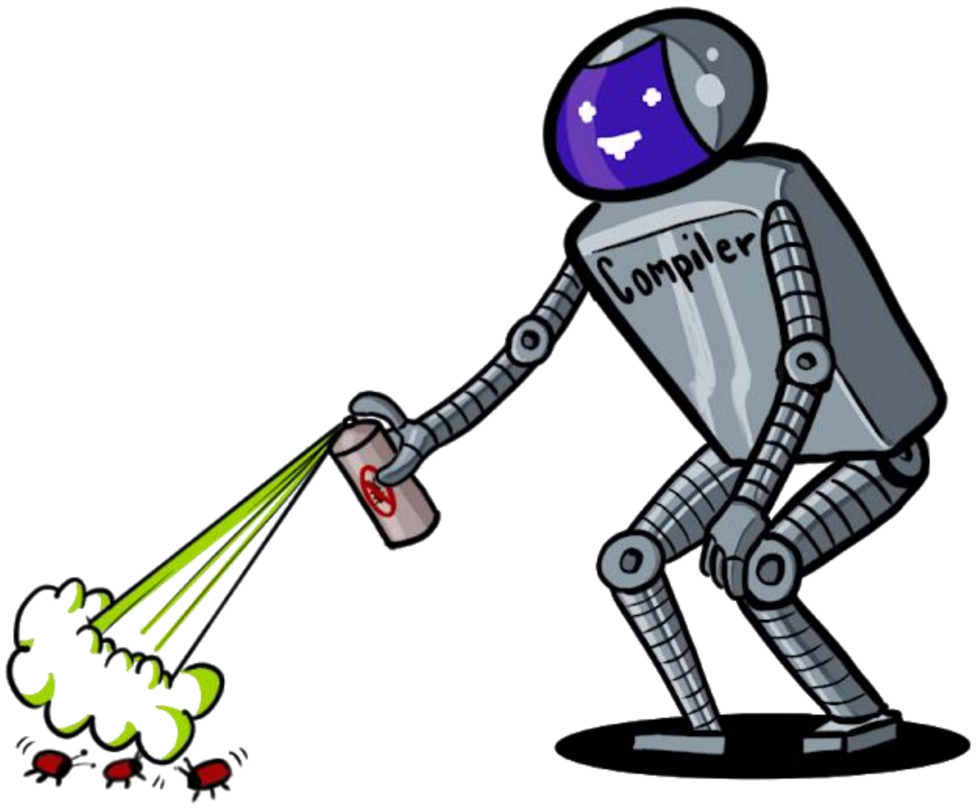
- Minimum config for type safety:
  - Variables are non-nullable unless explicitly stated
  - Expressions and declarations cannot be any unless explicitly stated

```
{
  "compilerOptions": {
    "strictNullChecks": true,
    "noImplicitAny": true,
    // ...
  }
}
```

- Enable **all** strict type checking options with **“strict”: true**

Why Type-Safety?

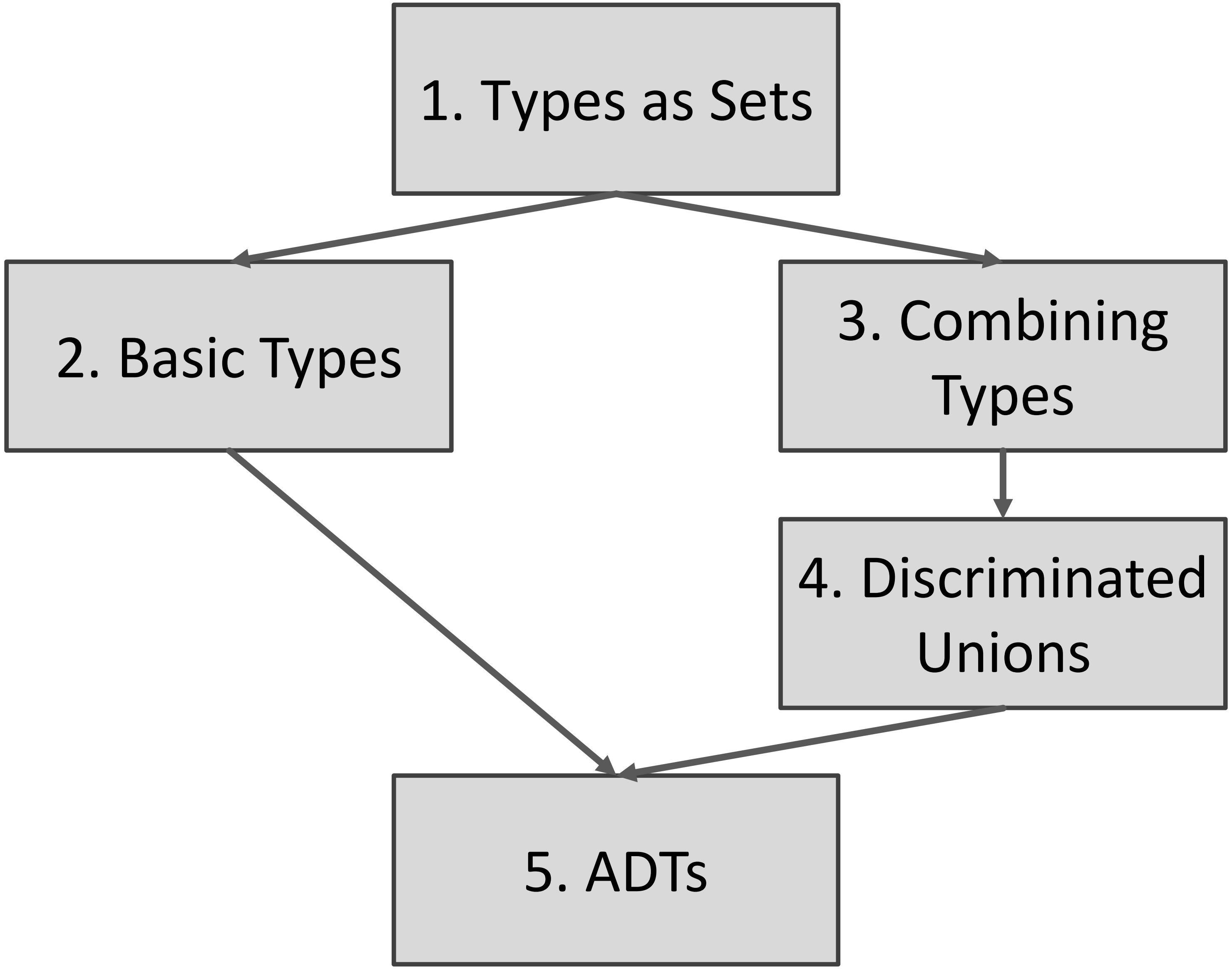
- **Let the compiler help you!**
- Prevents several classes of bugs
- Less testing effort
- Self-documentation
- Can be fun too!



Goals

- Get an intuition for types as a concept
- See some examples how to use type features of TS effectively
- Learn about ADTs

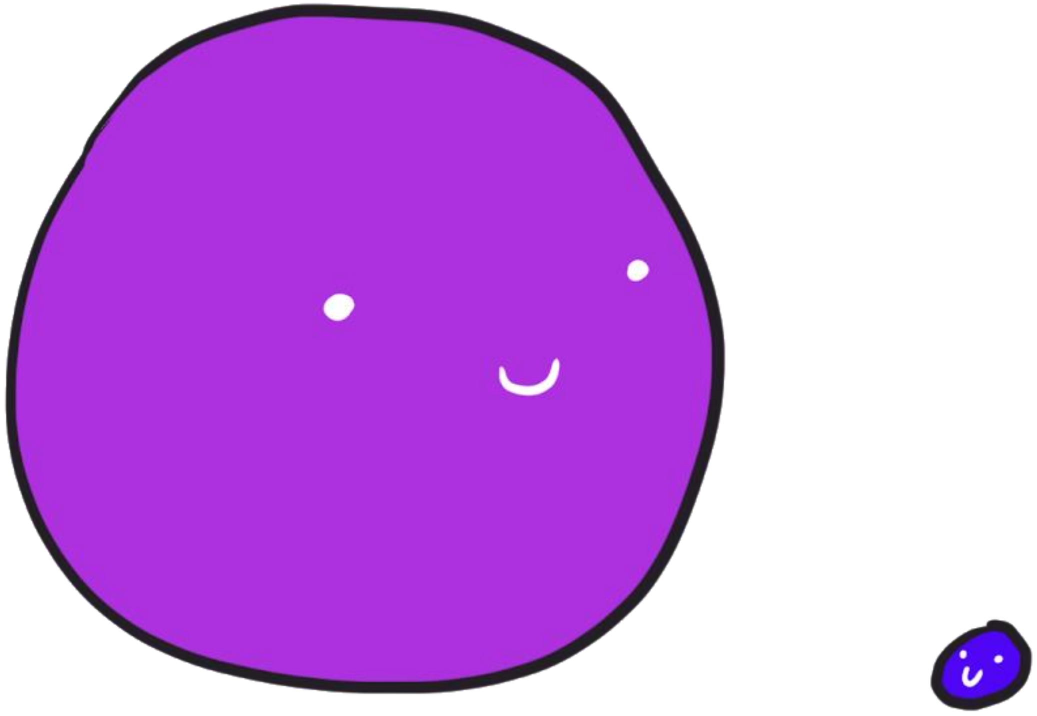
Structure


1. Types as Sets
2. Basic Types
3. Combining Types
4. Discriminated Unions
5. ADTs

Types as Sets

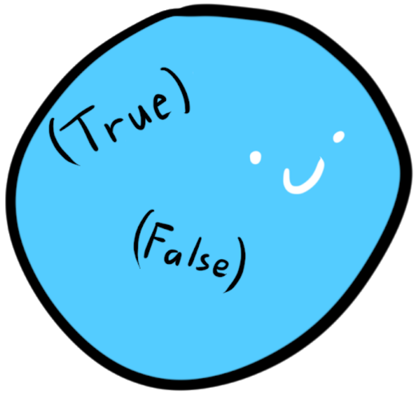
Types as Sets

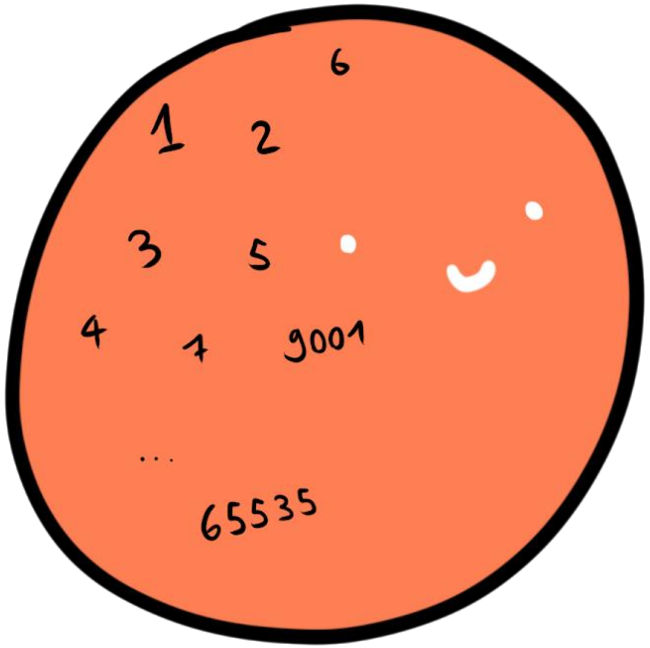
- Looking at types as sets helps to understand the underlying concepts
- What is a Set? A collection of Objects
- What is a Type? A class of Objects confining to a set of constraints
- Types can be described with simple algebra



Intuition: Counting inhabitants

- Boolean?
- Integer?
- String?





Structural Type Equality

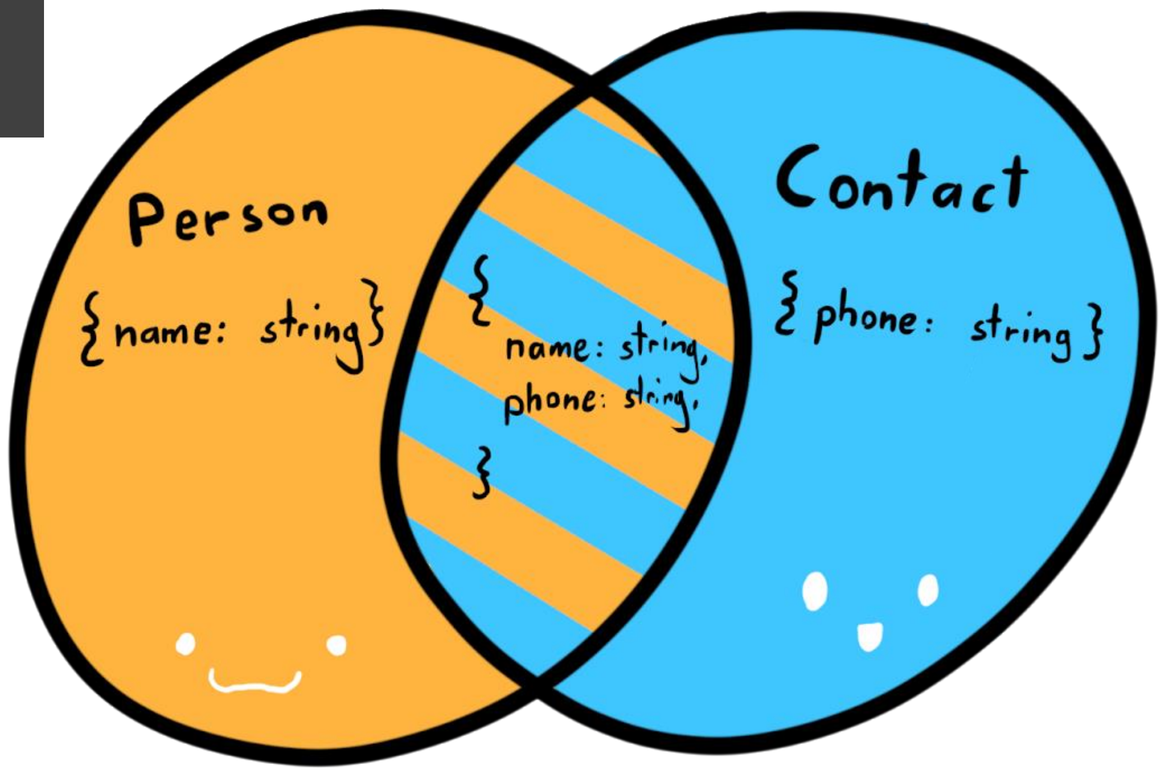
- C#, Java, etc: *Nominal* type systems
- TypeScript, Haskell, Elm: types are defined by their shape, *not* their names

```
type A = { foo: string };
type B = { foo: string };

const a: A = { foo: "bar" }
const b: B = a; // no compile error, type A is equal to B
```

Structural Type Equality

```
declare function call(c: Contact): boolean;

var personWithContact = {
name: "JoJo",
phone: "079 123 45 67"
};

call(personWithContact);
```

- Types can also overlap
- personWithContact is a subtype of both Person and Contact



Basic Types

Singleton Types

- Types with only one inhabitant

- In Typescript:

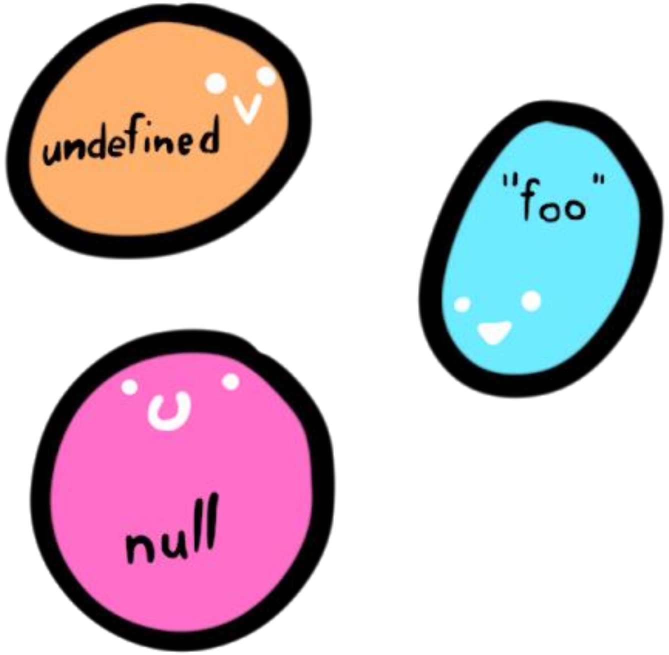
```
let u; // undefined
```

```
let a = null; // null
```

```
const x = "foo"; // "foo"
```

```
const y = 5; // 5
```

Type Literals

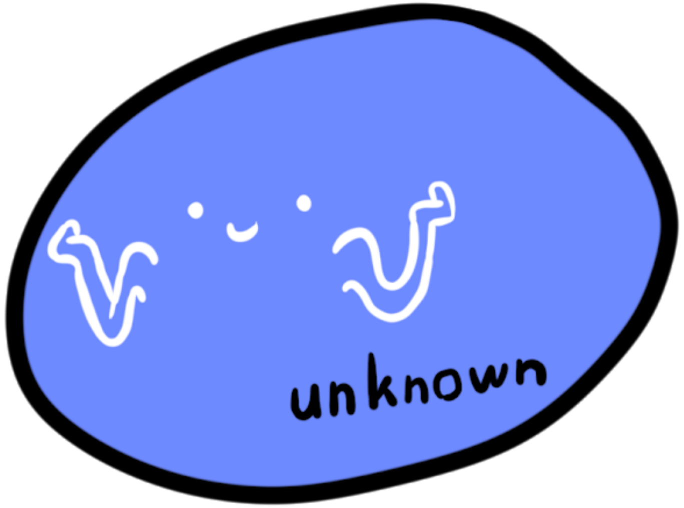


Top Types

- Also *universal supertype*, contains every possible value, equals the universal set *U*
- In typescript: **any** and **unkown**

```
let c: any = "foo";

c.someFunction();
let d: number = c;
```

```
let e: unknown = 42;

e.toString() // type error
let f: string = e; // type error
```



Example of using unknown

```
type Duck = { walkSpeed: number, quack: () => string };






function tryQuack(x: unknown) {
  return x.quack();
}
```

Operations on an object of type unknown are not allowed

Example of using unknown

```
type Duck = { walkSpeed: number, quack: () => string };

function isDuck(thing: unknown): thing is Duck {
  return typeof (thing as Duck).walkSpeed === "number"
      && typeof (thing as Duck).quack === "function";
}

function tryQuack(x: unknown) {
  if (isDuck(x)) {
    return x.quack();
  }

  console.log("Wasn't a duck, sorry.")
}
```

Type guard

The compiler knows that x must be of type Duck

Bottom Type

- A type that **contains no values**, equals the empty set Ø

```
let g: never = ({} as any); // Error
```

- Has several interesting applications, similar to 0 in algebra

- If the typescript compiler ever reaches a never, it will not compile

Basic Types as Sets: Summary

$\text{never} = \{\,\} = \emptyset$

$\text{any} = \text{unknown} = U$

$\text{null} = \{null\}$

$\text{"foo"} = \{"foo"\}$

$\text{number} = \{1, 2, \ldots\} = \mathbb{R}$

E.g. $\mathrm{U} \supseteq \mathbb{R} \supseteq \emptyset$

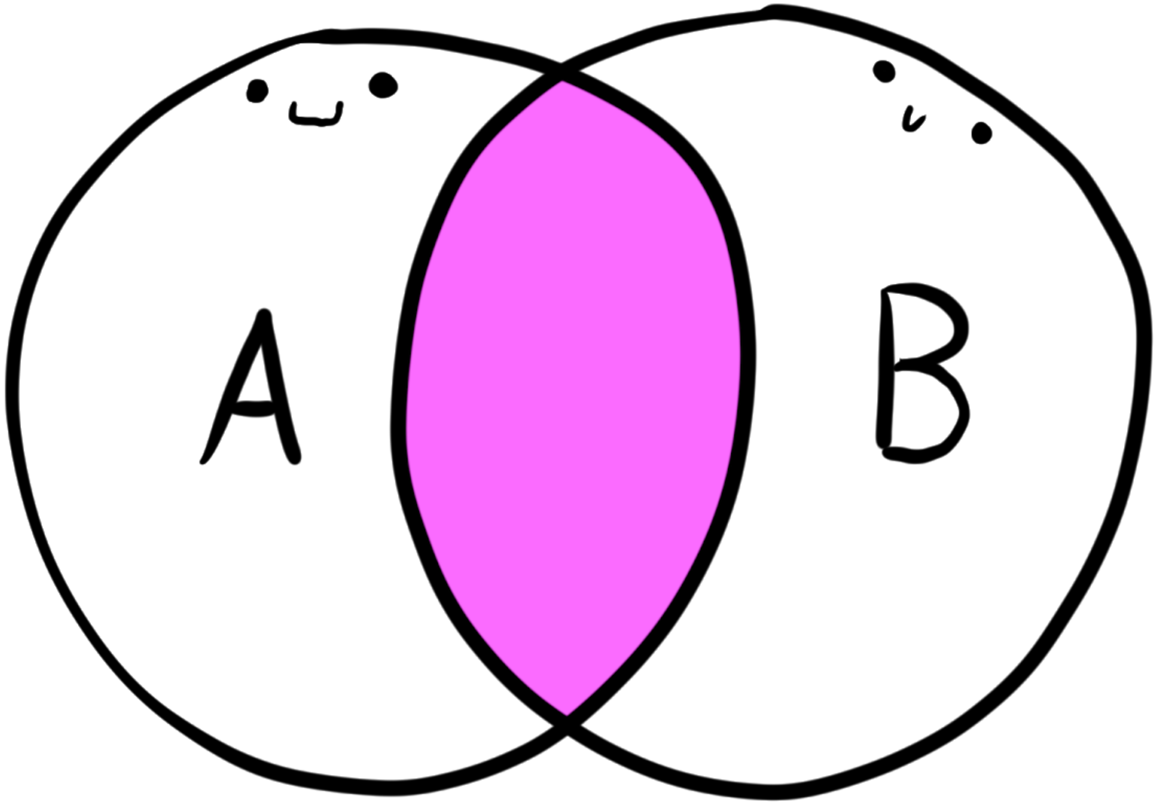
Combining Types

Intersection Type

- As sets: $A \cap B$

- Objects of an intersection type must contain properties of both types

```
type Person = { name: string, age: number }
type Contact = { phone: string }

let i: Person & Contact = {
    name: "JoJo",
    age: 17,
    phone: "079 123 45 67",
};
```



Union Type

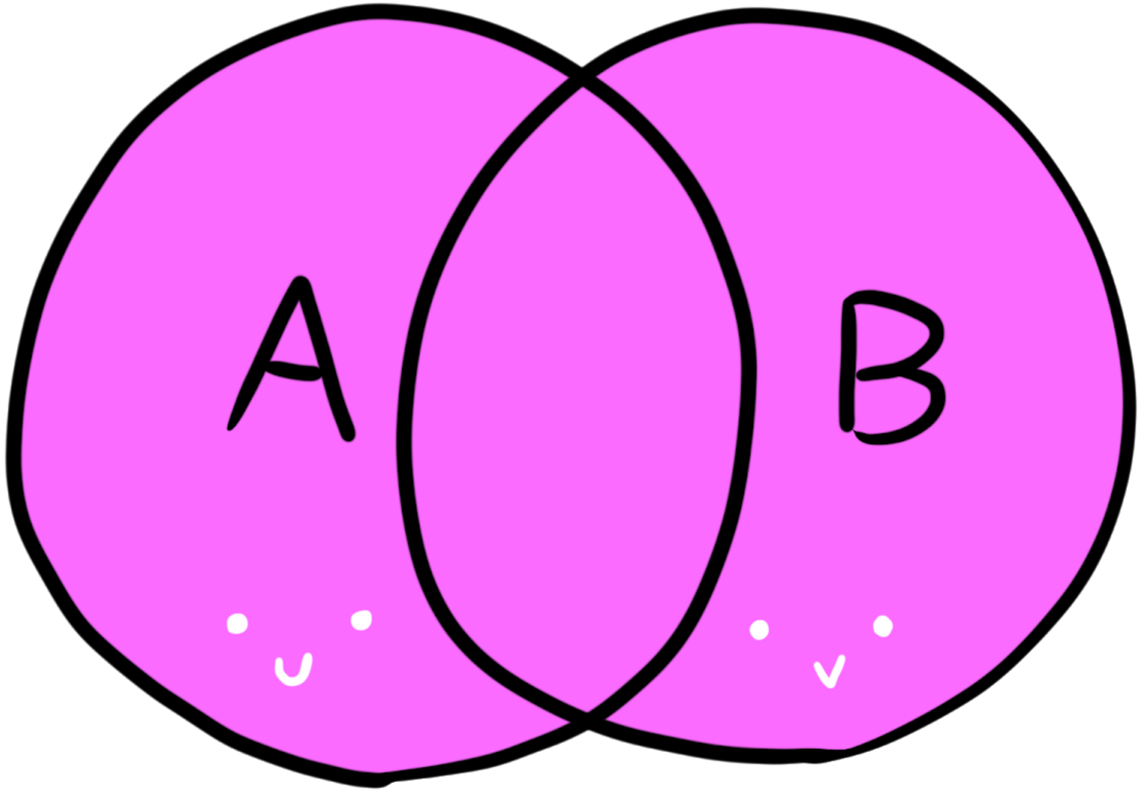
- As Sets: $A \cup B$

- The object can be either of the two types

```
let h: string | number;
h = "hello";
h = 42;
```

```
type Workdays = "Mon" | "Tue" | "Wed" | "Thu" | "Fri";
type Weekdays = Workdays | "Sat" | "Sun";
```

```
type MyBoolean = true | false; // MyBoolean = boolean
```



Some set algebra

Identity Laws:

$$A \cup \emptyset = A$$

$$A \cap U = A$$

```
type X = number | never; // X = number

type Y = number & unknown; // Y = number
```

Commutativity:

$$A \cup B = B \cup A$$

$$A \cap B = B \cap A$$

```
type A = "foo" | "bar";
type B = "bar" | "foo";
let a: A = "foo";
let b: B = "bar";
a = b;
```

Associativity:

$$(A \cup B) \cup C = A \cup (B \cup C)$$

$$(A \cap B) \cap C = A \cap (B \cap C)$$

```
type E = ( A | B) | C;
type F = A | (B | C);
let e: E = "foo";
let f: F = "bar";
e = f;
```

Example of using Union Types

```
function double(x: number | string) {
  return x * 2;
}
```

A string cannot be multiplied

Example of using Union Types

```
function double(x: number | string) {
  if(typeof x == "number") {
    return x * 2;
  }

  return (2 * Number(x)).toString();
}
```

The control flow analysis infers the type of x in both cases

Discriminated Unions

Opaque Types

- The structural type equality can stand in our way

- Sometimes we want to differentiate values with identical structure

```
type Username = string;
type Password = string;

const displayUserName = (u: Username) => `User: ${u}`;

const pw: Password = "hunter2";

displayUserName(pw); // valid, but not what we want
```

Opaque Types

- How to make your types “Opaque”?
  - Type Literals
  - Intersection Types

These types are now (structurally) different!

```
type Username = string & { type: "Username" };
type Password = string & { type: "Password" };

const displayUserName = (u: Username) => `User: ${u}`;

const pw: Password = "hunter2" as Password;

displayUserName(pw); // compiler prevents us from misusing sensitive data
```

Opaque Types

- Can be made even cleaner with mapped types

- The compiler prevents us to use a password value where we shouldn't

```
type Opaque<K, T> = T & {_type: K};

type Username = Opaque<"Username", string>;
type Password = Opaque<"Password", string>;

const displayUserName = (u: Username) => `User: ${u}`;

const pw: Password = "hunter2" as Password;

displayUserName(pw); // A-OK!
```

Discriminated Unions

- The concept of «Opaque» Types can be extended to Discriminated Unions
  - In Set Theory called **disjoint union**: $A \sqcup B$
  - Also called **tagged union**

- Sets are explicitly split with a «Tag» or «Discriminator»

- Especially useful if combined with TypeScripts control flow analysis

Example of using Discriminated Unions

```
type InitAction = { type: "INIT" }
type LoginAction = { type: "LOGIN", name: string, password: string }

type Actions = InitAction | LoginAction;

function handleAction(action: Actions) {
  switch (action.type) {
    case "INIT":
      return "Initialized";
    case "LOGIN":
      console.log(action.name + " logged in");
      return "Logged In"
    default:
      console.log("WTF");
  }
}
```

Property “name” can be safely accessed here

Exhaustiveness Checking

- We often have types that cover several cases

- What if you can guarantee that you cover them all?

- Benefits:
  - Cut down bugs
  - Make extending functionality easier

Example of using Discriminated Unions

```
type InitAction = { type: "INIT" }
type LoginAction = { type: "LOGIN", name: string, password: string }
type LogOutAction = { type: "LOGOUT" }

type Actions = InitAction | LoginAction | LogOutAction;

function handleAction(action: Actions) {
  switch (action.type) {
    case "INIT":
      return "Initialized";
    case "LOGIN":
      console.log(action.name + " logged in");
      return "Logged In"
    default:
      console.log("WTF");
  }
}
```

We don’t handle all cases anymore!

Example of using Discriminated Unions

```
type InitAction = { type: "INIT" }
type LoginAction = { type: "LOGIN", name: string, password: string }
type LogOutAction = { type: "LOGOUT" }

type Actions = InitAction | LoginAction | LogOutAction;

function handleAction(action: Actions): string {
  switch (action.type) {
    case "INIT":
      return "Initialized";
    case "LOGIN":
      console.log(action.name + " logged in");
      return "Logged In"
    default:
      console.log("WTF");
  }
}
```

Compiler complains:
Function returns
string | undefined

Another Example of using Union Types

```
type InitAction = { type: "INIT" } ...

type Actions = InitAction | LoginAction | LogOutAction;

function assertNever(x: never): never {
  throw new Error("Unexpected object: " + x);
}

function exhaustiveHandleAction(action: Actions) {
  switch (action.type) {
    case "INIT":
      return "Initialized";
    case "LOGIN":
      console.log(action.name + " logged in");
      return "Logged In"
    default:
      return assertNever(action);
  }
}
```

The compiler complains if it ever reaches this point ✔

Algebraic Data Types

Algebraic data types

- Algebraic data types (ADTs) are composite types with algebraic properties

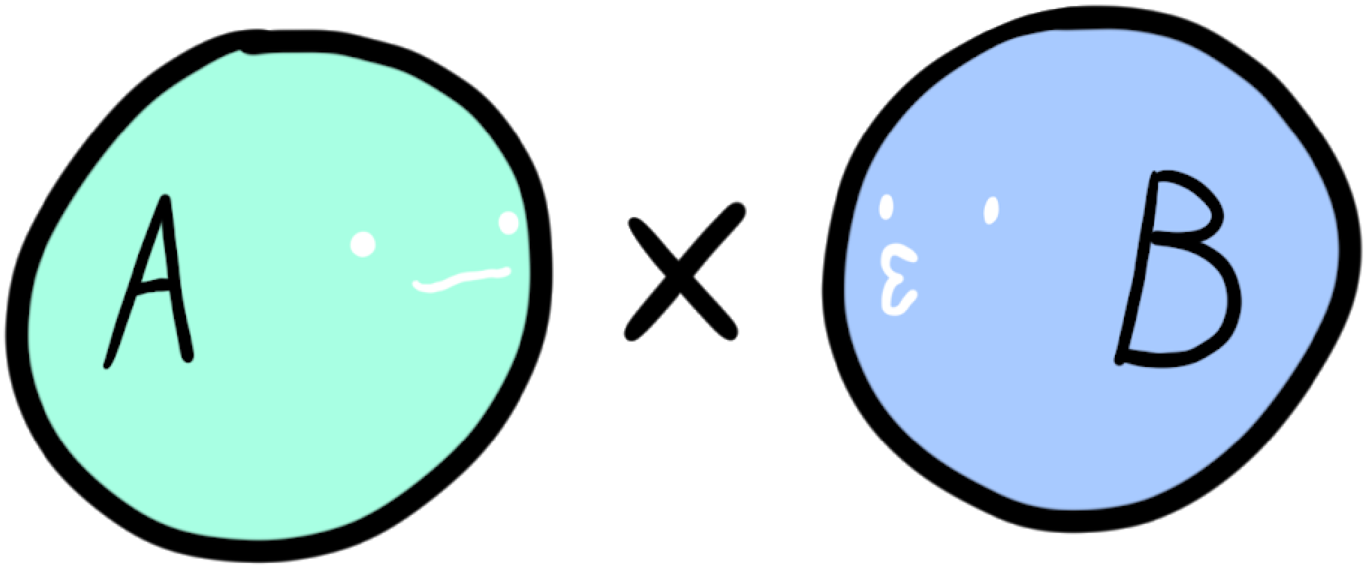
- Most common classes: **Products** and **Sums**

- We want:
  - Type Checking
  - Exhaustivity

Product Types

- Is equal to the carthesian product of sets
- Can be modelled as a Tuple: `let p: [number, boolean] = [5, true];`
- Or, more explicit, as an object with two properties:

```
type Product<A, B> = { fst: A, snd: B };
let prod: Product<number, boolean> = { fst: 5, snd: true };
```

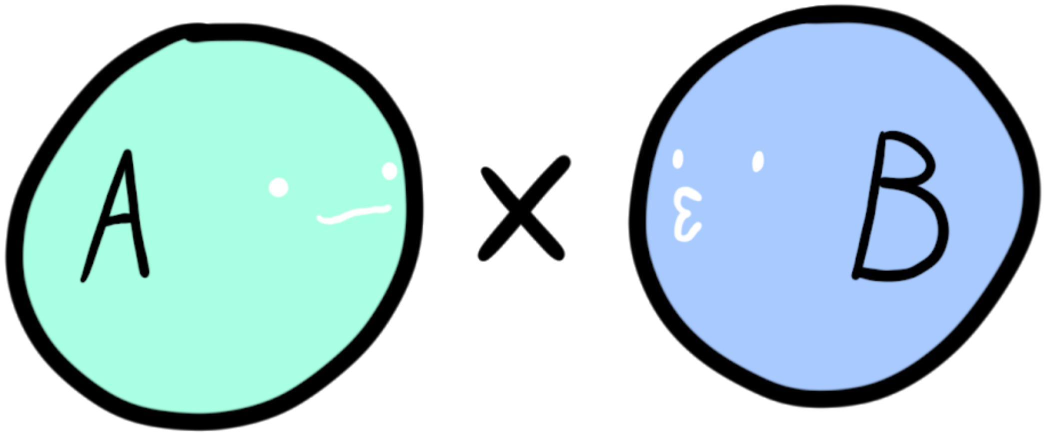


Product Types

- Why «Product»?

- Try counting the inhabitants

$$|[A, B]| = |A| \times |B|$$



Product Types

- Does other algebra apply?

Commutativity: $A \times B = B \times A$

```
let p1: [A, B] = ["A", "B"];
let p2: [B, A] = ["B", "A"];
const swap = ([x, y]) => [y, x];
```

Associativity: $(A \times B) \times C = A \times (B \times C)$

```
let p2: [A, [B, C]] = ["A", ["B", "C"]];
let p3: [[A, B], C] = [["A", "B"], "C"];
```

```
const f = ([a, [b, c]]: [A, [B, C]]) => [[a, b], c];
const f_inv = ([[a, b], c]: [[A, B], C]) => [a, [b, c]];
```

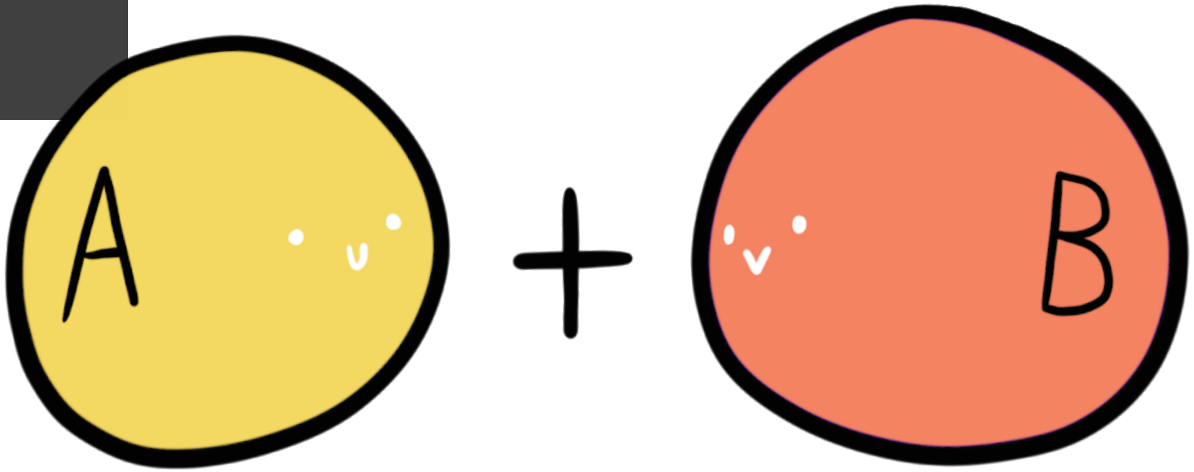
Neutral-Element: $A \times 1 = A$

```
let p4 = ["A", null];

const rho = ([a, _]: [any, null]) => a;
const rho_inv = (a: any) => [a, null];
```

Sum Types

- Sum types aren't a native language feature of TS
- But they can be built with discriminated unions

```
type Left<T> = { type: "Left", val: T };
type Right<T> = { type: "Right", val: T };

type Sum<A, B> = Left<A> | Right<B>;
```



- The sum type is also known as «Either»

```
type Either<A, B> = Sum<A, B>;
```

Sum Types

- Does other algebra apply?

Commutativity: $A + B = B + A$

```
type S1 = Sum<"A", "B">;
type S2 = Sum<"B", "A">;

function swapSum(s: Sum<"A", "B">): Sum<"B", "A"> {
        return s.type == "Left"
            ? { type: "Right", val: s.val }
            : { type: "Left", val: s.val };
    }
```

Associativity: $(A + B) + C = A \times (B \times C)$

Neutral-Element: $A + \emptyset = A$

```
const s: Sum<string, never> = { type: "Left", val: "foo"};

const roh = <T>(x: Sum<T, never>): T => x.val;
const roh_inv = <T>(x: T): Sum<T, never> => ({ type: "Left", val: x });
```

Type Algebra: Distributivity

(A x (B + C))
= (A x B) + (A x C)

```
type X<A, B, C> = [A, Either<B, C>];
type Y<A, B, C> = Either<[A, B], [A, C]>

function prodToSum<A, B, C>([x, y]: [A, Either<B, C>]): Either<[A, B], [A, C]> {
  if (e.type === "Left") {
    return { type: "Left", val: [x, y.val] }
  } else {
    return { type: "Right", val: [x, y.val] }
  }
}

function sumToProd<A, B, C>(x: Either<[A, B], [A, C]>): [A, Either<B, C>] {
  if (x.type === "Left") {
    return [x.val[0], { type: "Left", val: y.val[1] }]
  } else {
    return [x.val[0], { type: "Right", val: y.val[1] }]
  }
}
```

Type Algebra

Numbers	Types
0	Never
1	undefined, null, «foo»
a + b	Either<A, B> = Left<A> | Right<B>
a x b	[A, B], Product<A, B> = [A, B]
2 = 1 + 1	type bool = True | False
1 + a	Optional<A> = «nothing» | Some<A>

Logic	Types
False	never
True	undefined, null, «foo»
a || b	Either<A, B> = Left<A> | Right<B>
a && b	[A, B], Product<A, B> = [A, B]

Applications

- Many common data structures can be modelled as ADTs

```
type Success = { value: string, correlationId: number };
type Failure = { error: string };
type Response
  = { type: "success", value: Success } | { type: "failure", value: Failure };
```

$$= \text{Success} + \text{Failure}$$

```
// Empty + (Head x Tail)
type List<T> = Either<"Empty", { head: T, tail: List<T> }>;
```

$$= \text{Empty} + (\text{Head} \times \text{Tail})$$

Example: Query Service

```
type Found<T> = { status: "Found", value: T };
type NotFound = { status: "Not Found" };
type Error = { status: "Error", message: string };
type QueryResult<T> = Found<T> | NotFound | Error;

function query<T>(sql: string): QueryResult<T> {
  // ...
}
```

Example: Query Service

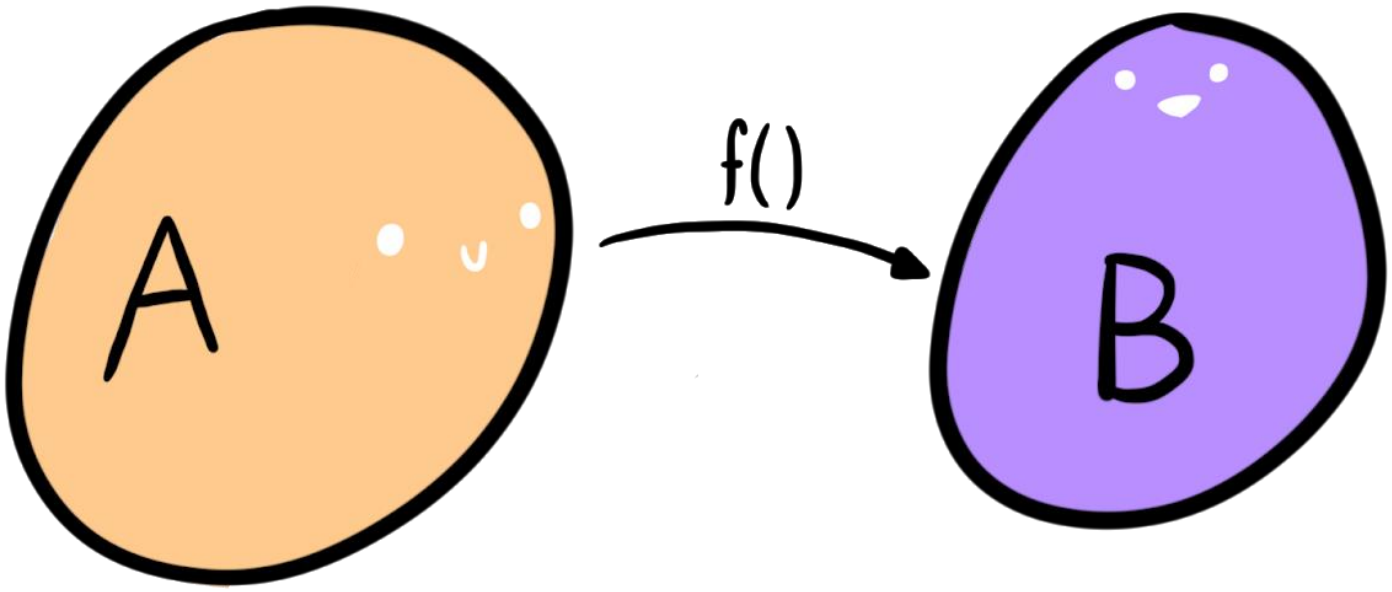
```
function handleResult<T, U>(
  result: QueryResult<T>,
  found: (r: Found<T>) => U,
  notFound: (r: NotFound) => U,
) {
  switch (result.status) {
    case "Found":
      return found(result);
    case "Not Found":
      return notFound(result);
    case "Error":
      return printError(result.message);
    default:
      assertNever(result);
  }
}
```

Example: Query Service

```
function handleResult<T, U>(
  result: QueryResult<T>,
  found: (r: Found<T>) => U,
  notFound: (r: NotFound) => U,
) { ... }

function getPetNames(personName: number): string[] | undefined {
  return handleResult(
    query<Person>(`Select * FROM Person WHERE name = ${personName}`),
    foundPersonResult => handleResult(
      query<Pet[]>(`Select * FROM Pet WHERE ownerId = ${foundPersonResult.value.id}`),
      foundPetResult => foundPetResult.value.map(pet => pet.name),
      _ => [],
    ),
    _ => [],
  );
}
```

Bonus round: Set of Functions

- Function Types are also ADTs
- $f(A) \Rightarrow B = B^A$
- Functions are **Exponentials**



Bonus round: Set of Functions

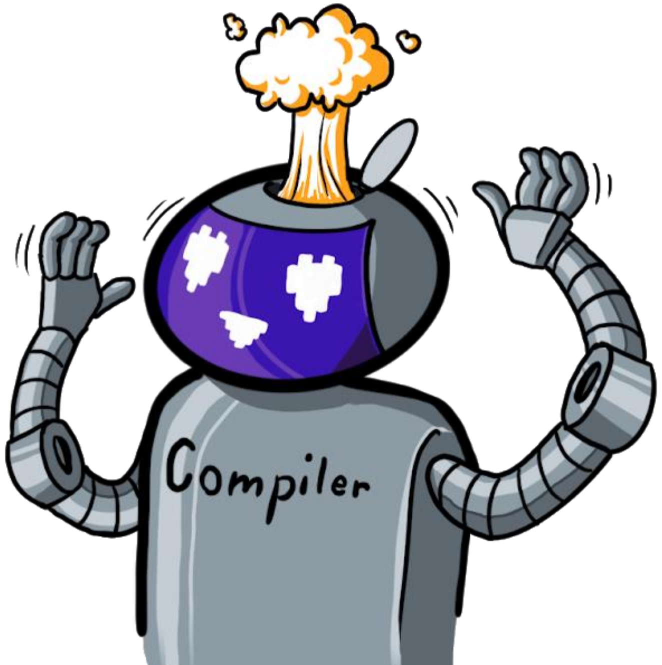
$$
\begin{aligned}
&\mathrm{Sum}{<}A, B{>} \Rightarrow C \\
&= C^{A+B} \\
&= C^{A} \times C^{B} \\
&= (A \Rightarrow C, B \Rightarrow C)
\end{aligned}
$$

$$
\begin{aligned}
&(A, B) \Rightarrow C \\
&= C^{A \times B} \\
&= (C^{A})^{B} \\
&= A \Rightarrow (B \Rightarrow C)
\end{aligned}
$$

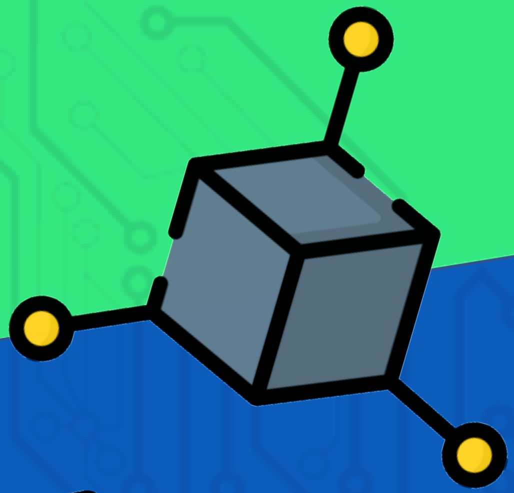
Oops we've just proven currying

Conclusion

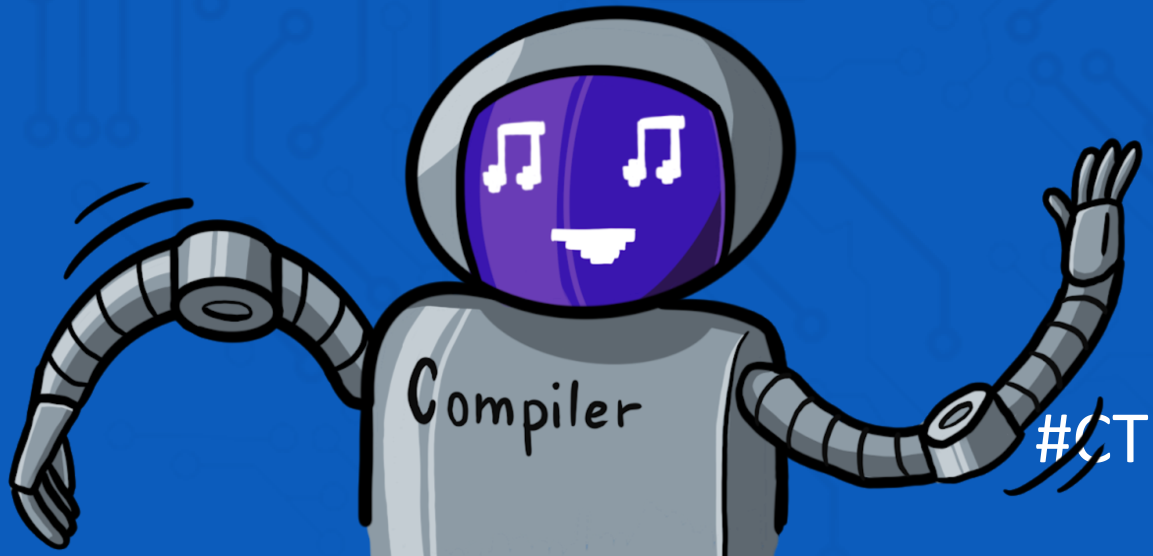
- We have a **powerful**, **functional-language-level** type system built into **JavaScript**, one of the most used, handiest languages of our time



CLIENT TECHNOLOGY DAYS



Thanks!



Yacine Mekesser

#CTD2019

Reading List

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

- https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/

- Git repository of this presentation incl. code samples: https://github.com/ymekesser/a-type-of-magic-presentation