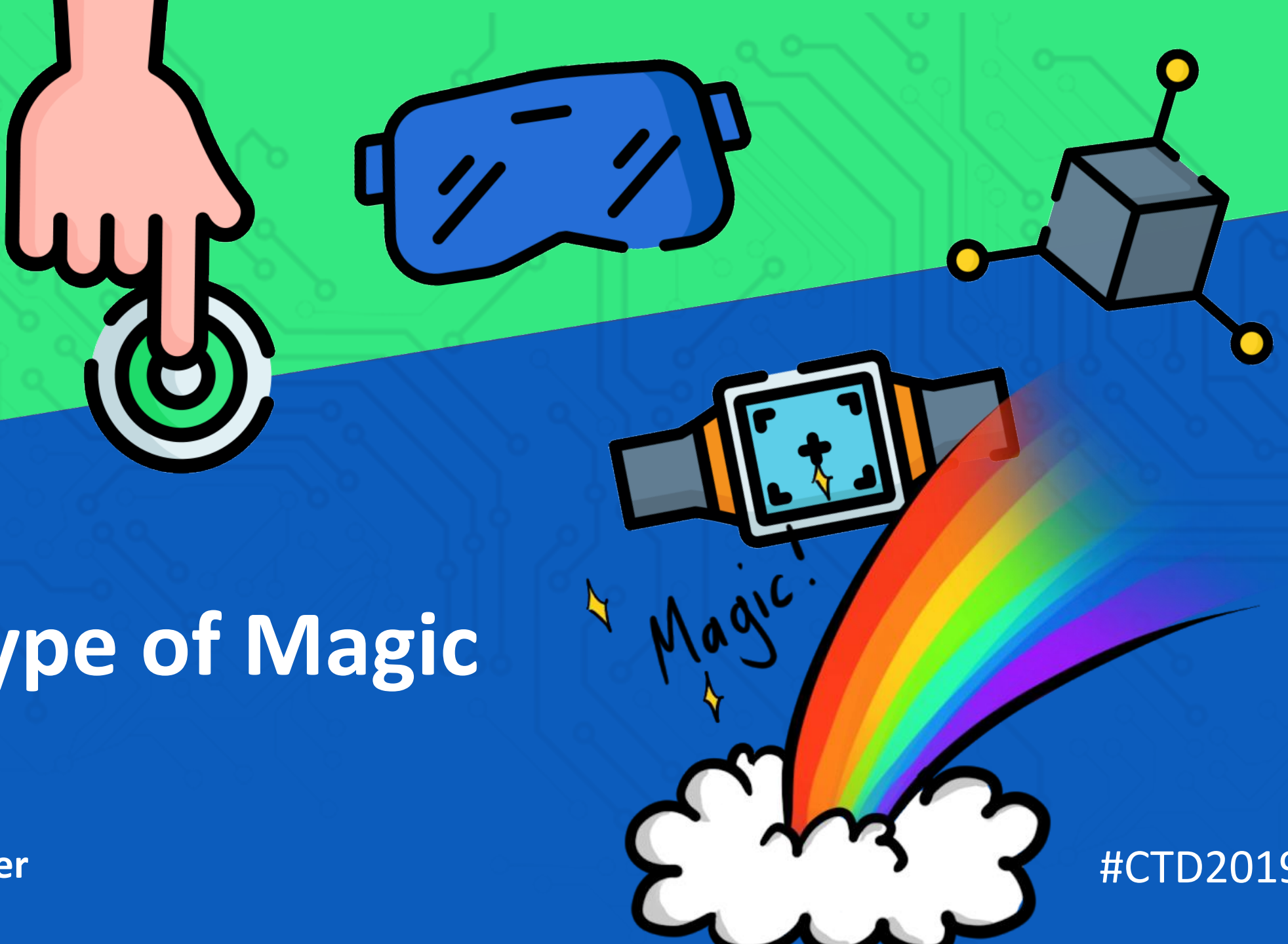


CLIENT  
TECHNOLOGY  
DAYS



It's a Type of Magic

Yacine Mekesser

#CTD2019

# TypeScript

Just JavaScript with type annotations?

# TypeScript

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

# TypeScript Config

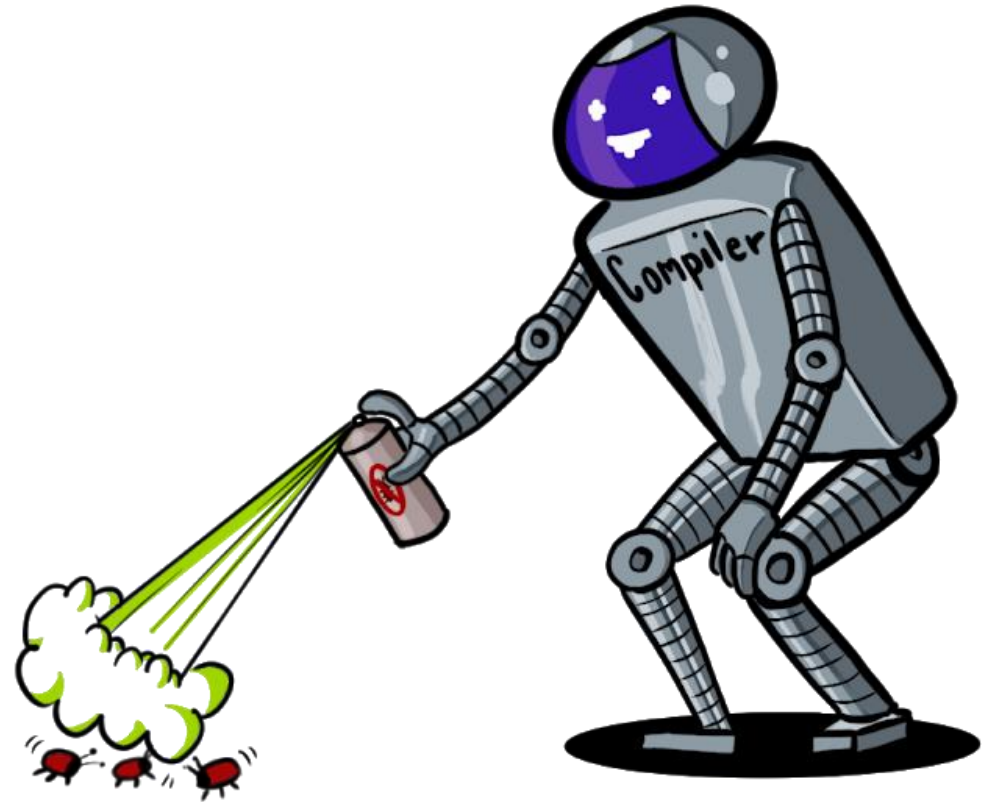
- Minimum config for type safety:
  - Variables are non-nullable unless explicitly stated
  - Expressions and declarations cannot be any unless explicitly stated

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

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

# Why Type-Safety?

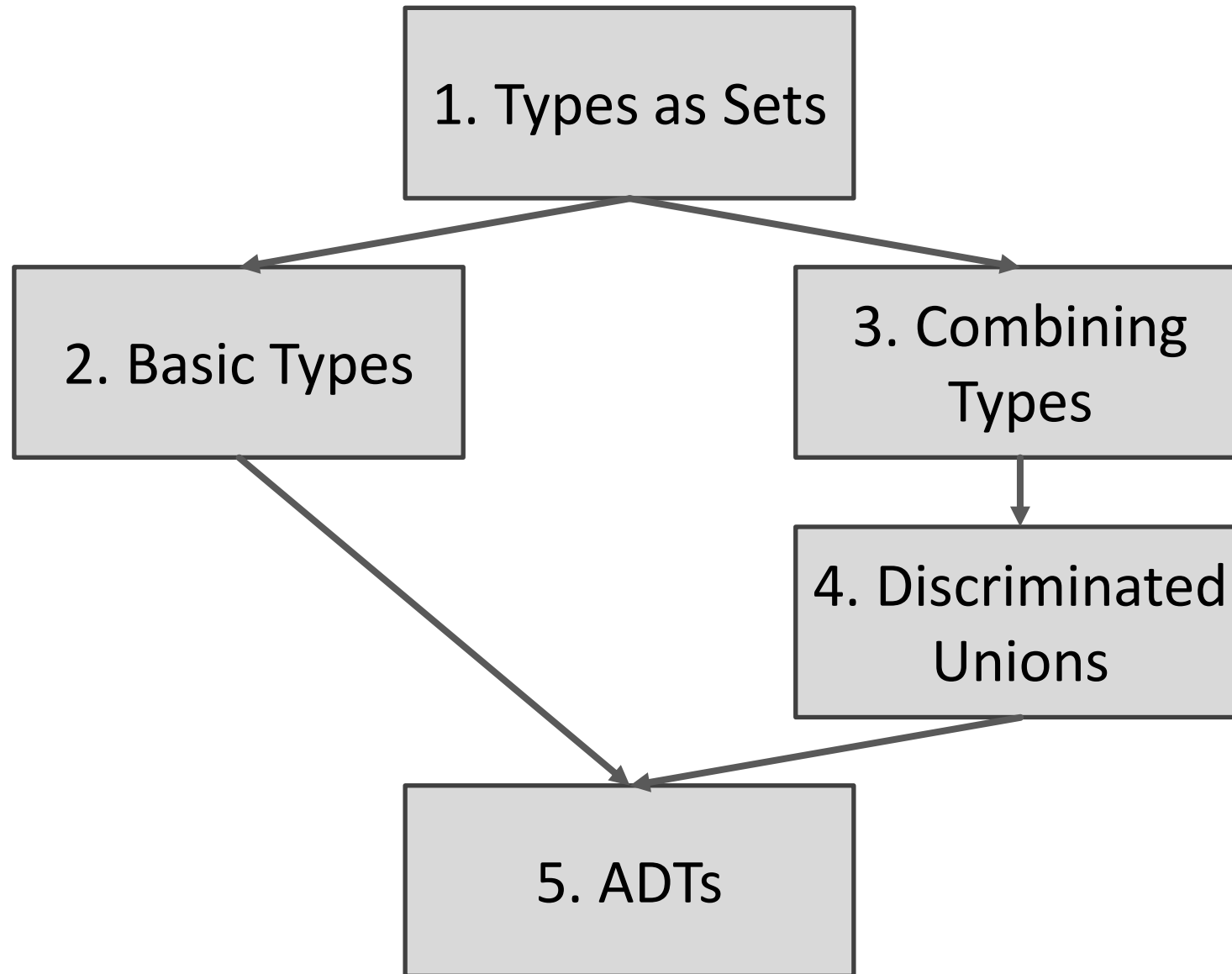
- **Let the compiler help you!**
- Prevents several classes of bugs
- Less testing effort
- Self-documentation
- Can be fun too!



# Goals

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

# Structure

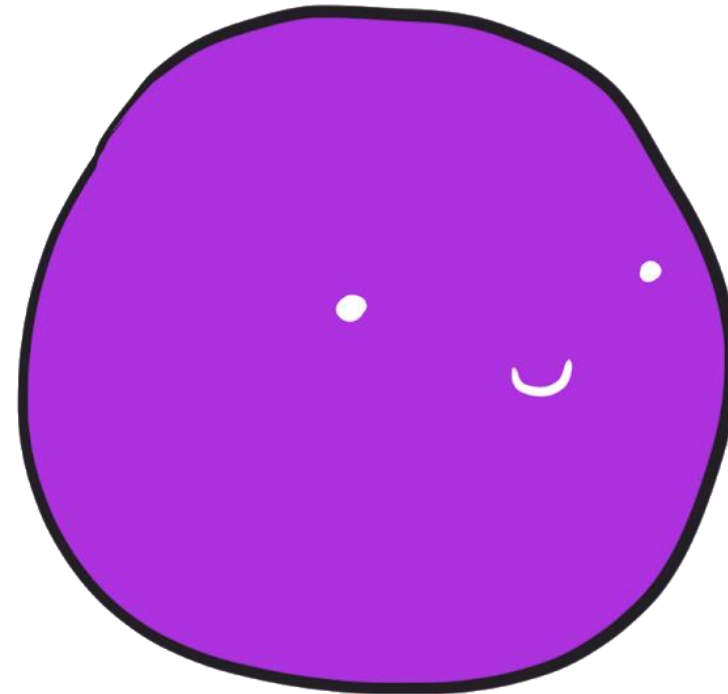


# Types as Sets



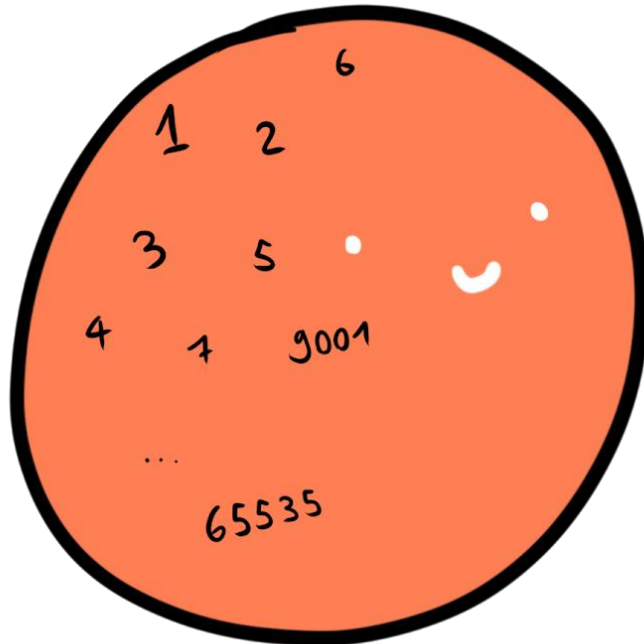
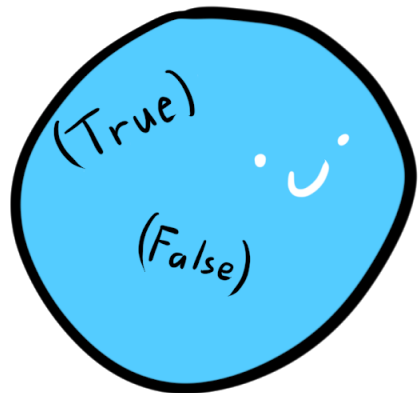
# Types as Sets

- 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?



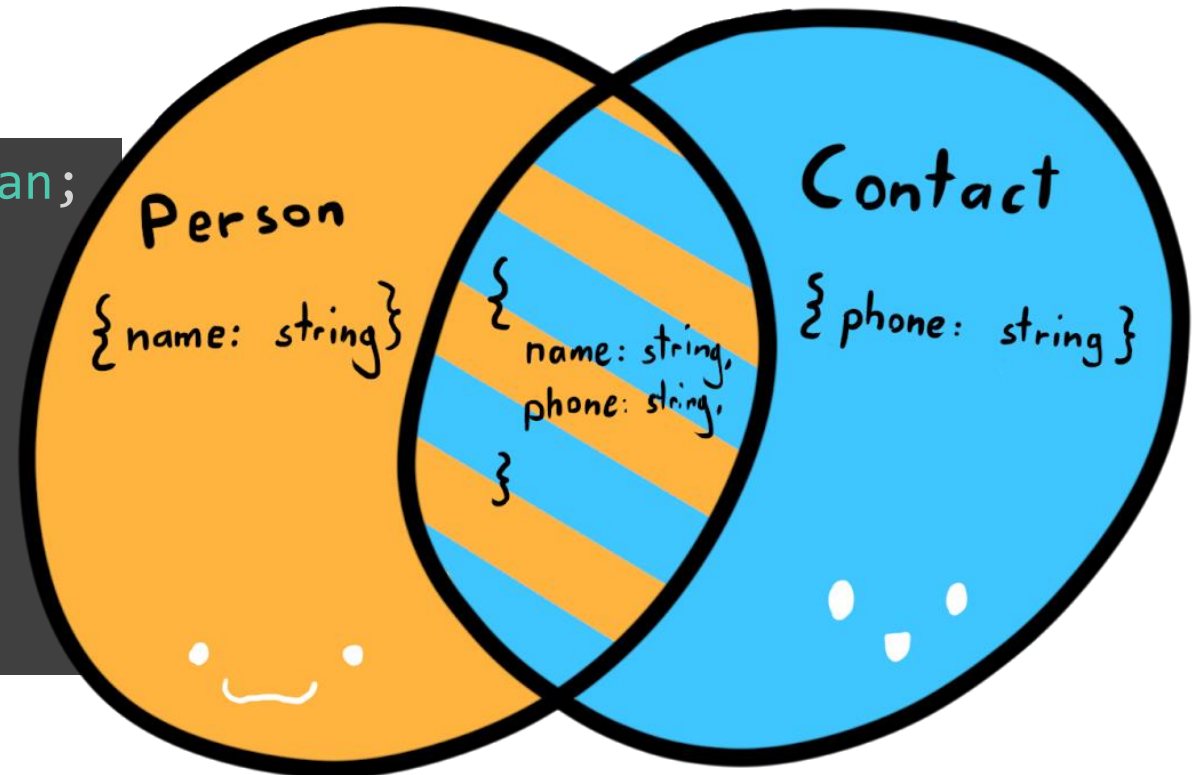
# Structural Type Equality

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

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

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

```
call(personWithContact);
```



# 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 **unknown**

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



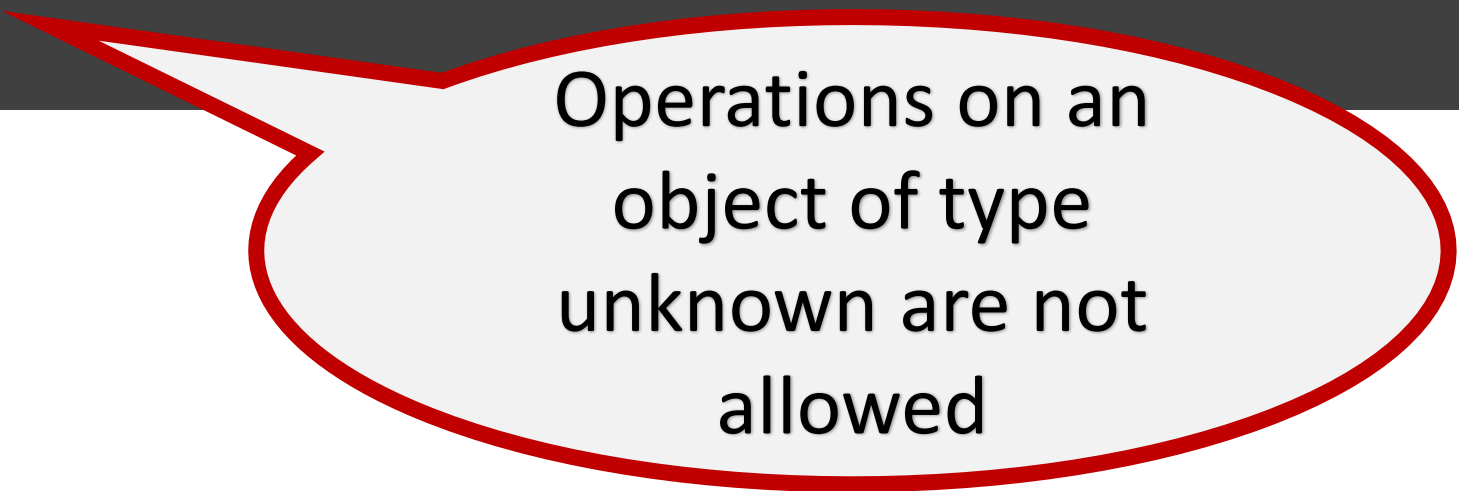
```
let e: unknown = 42;  
let f: string = e; // type error
```



# 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 guard

```
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.");  
}
```

The compiler  
knows that x  
must be of type  
Duck



# Bottom Type

- A type that **contains no values**, equals the empty set  $\emptyset$

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

- Has several interesting applications, similar to 0 in algebra
- If the typescript ever reaches a never, it will not compile



# Basic Types as Sets: Summary

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

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

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

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

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

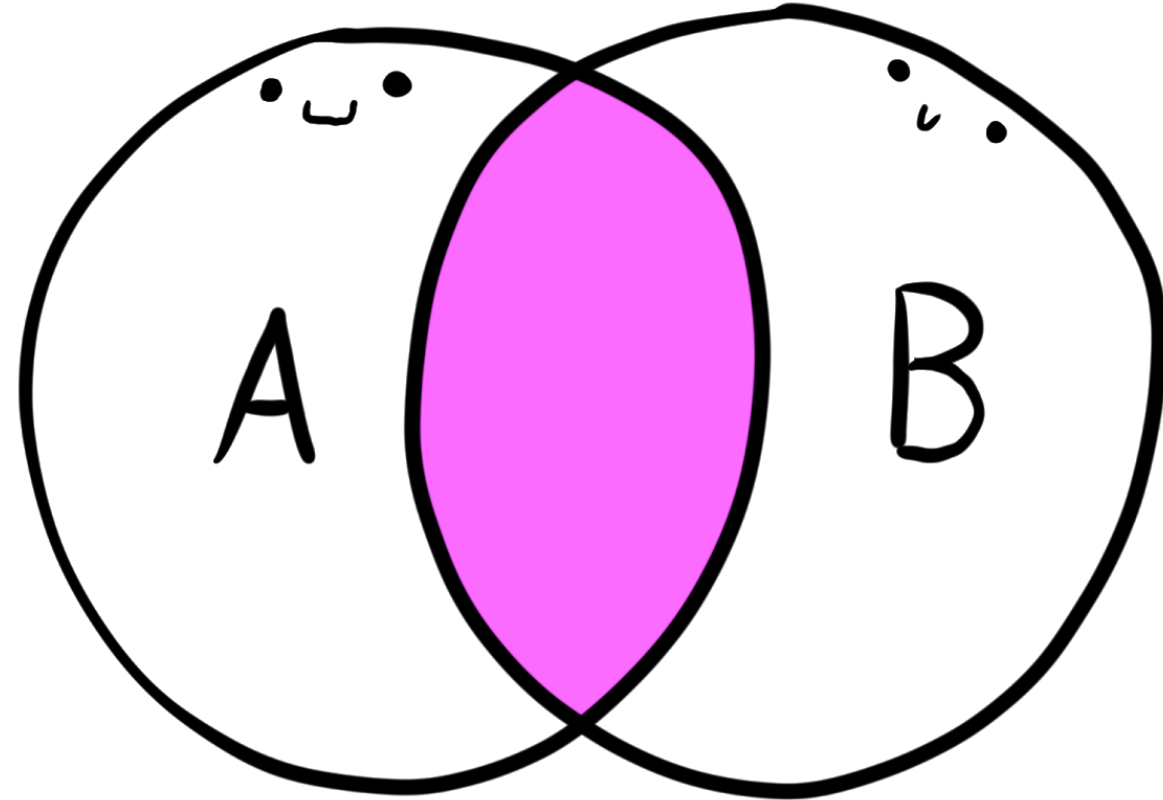
E.g.  $U \supseteq \mathbb{R} \supseteq \emptyset$

# Combining Types

# Intersection Type

- As sets:  $A \cap B$
- Objects of an intersection type must contain properties of both types

```
type Person = { name: string }  
type Contact = { phone: string }  
  
let p: Person & Contact = {  
  name = "JoJo",  
  phone = "079 123 45 67",  
};
```



# Union Type

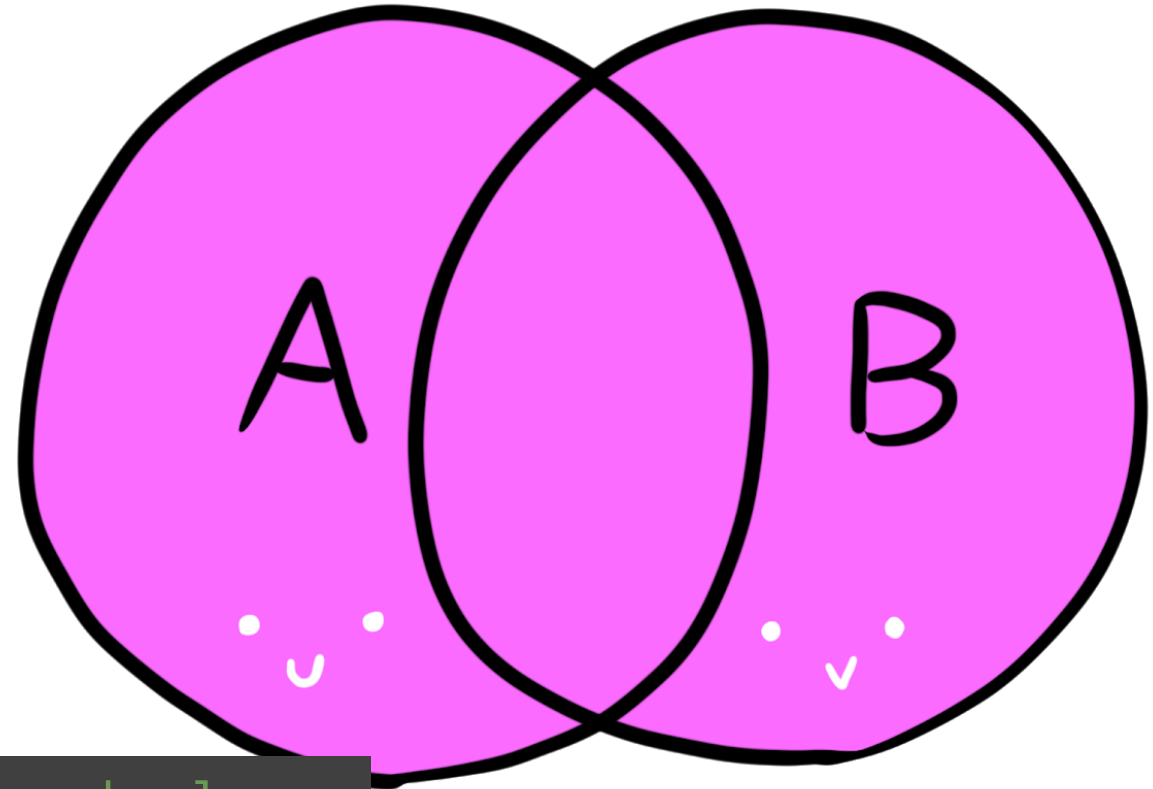
- As Sets:  $A \cup B$

- The object can be either of the two types

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

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

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



# 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:

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

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

Associativity:

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

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

And others apply as well 😊

# Example of using Union Types

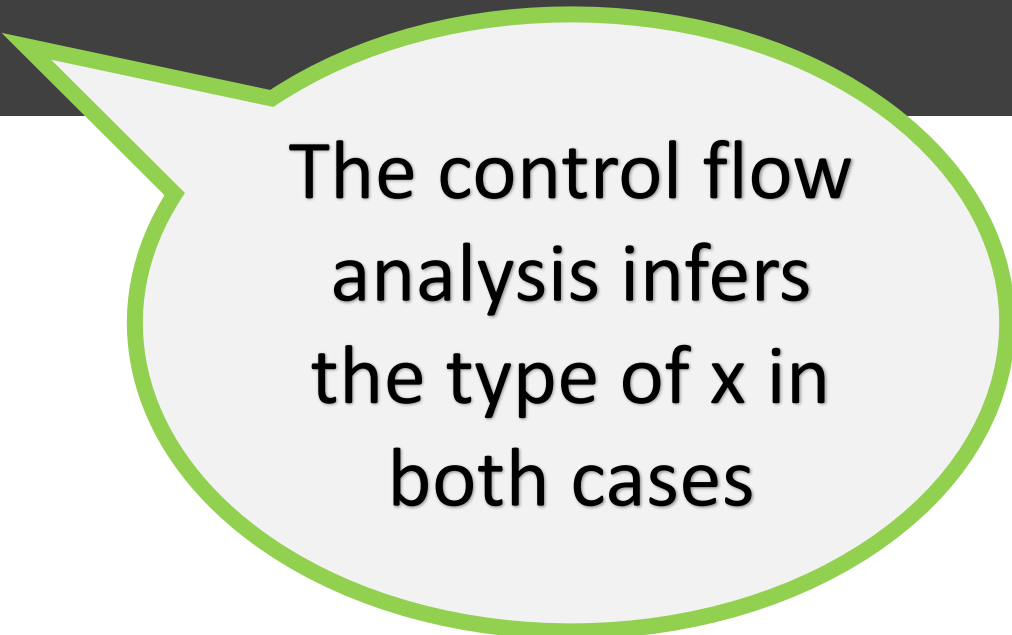
```
function double(x: number | string) {  
  return x * 2;  
}
```



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 TypeScript's 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 handleAction2(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

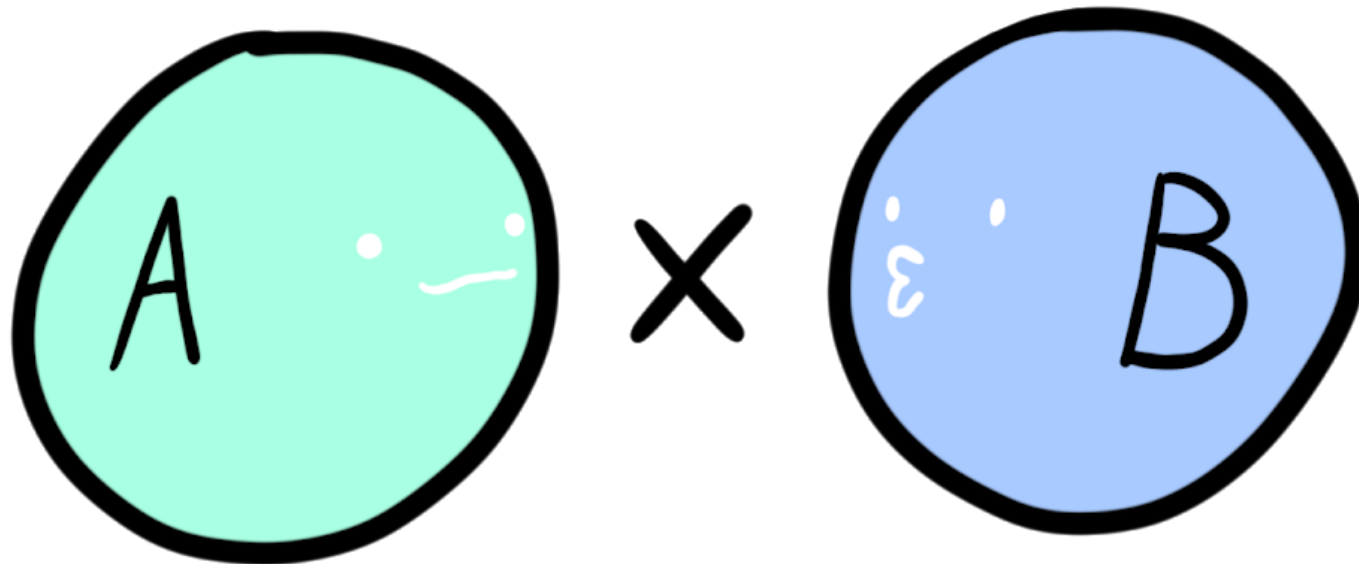
# Algebraic data types

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

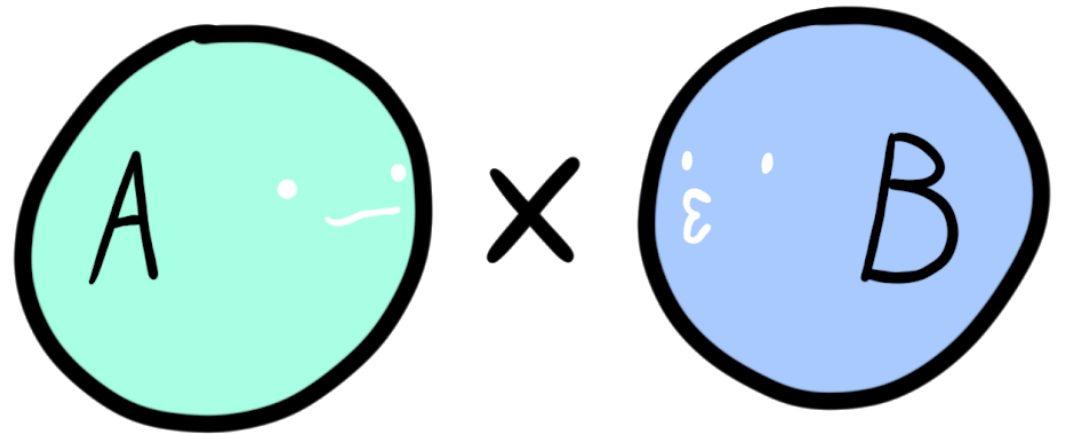
```
type Product<A, B> = { fst: A, snd: B };  
let prod: Product<number, boolean> = { fst: 5, snd: true };
```



# 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"];
```

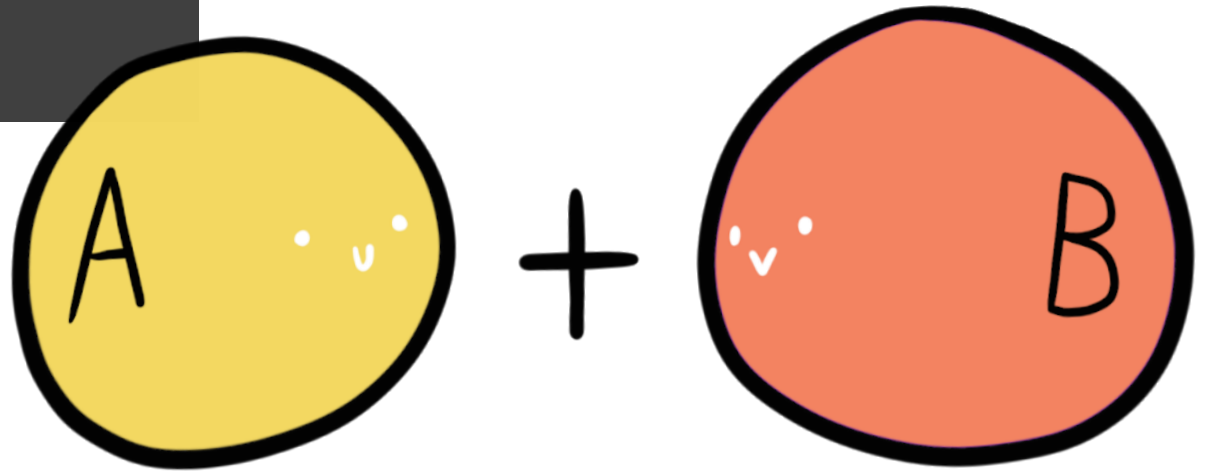
Neutral-Element:  $A \times 1 = A$

```
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>;
```



# Type Algebra: Commutativity

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

```
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»
$a + b$	$\text{Either}\langle A, B \rangle = \text{Left}\langle A \rangle \mid \text{Right}\langle B \rangle$
$a \times b$	$[A, B], \text{Product}\langle A, B \rangle = [A, B]$
$2 = 1 + 1$	$\text{type bool} = \text{True} \mid \text{False}$
$1 + a$	$\text{Optional}\langle A \rangle = \text{«nothing»} \mid \text{Some}\langle A \rangle$

Logic	Types
False	never
True	undefined, null, «foo»
$a \mid \mid b$	$\text{Either}\langle A, B \rangle = \text{Left}\langle A \rangle \mid \text{Right}\langle B \rangle$
$a \&\& b$	$[A, B], \text{Product}\langle A, B \rangle = [A, B]$

# Applications

- Many common data structures can be modelled as ADTs

```
type List<T> = { head: T, tail: List<T> };
```

```
type Leaf<T> = { kind: "leaf", value: T };  
type Node<T> = { kind: "node", element: T, left: Tree<T>, right: Tree<T> };  
type Tree<T> = Leaf<T> | Node<T>;
```

# 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);
  }
}
```

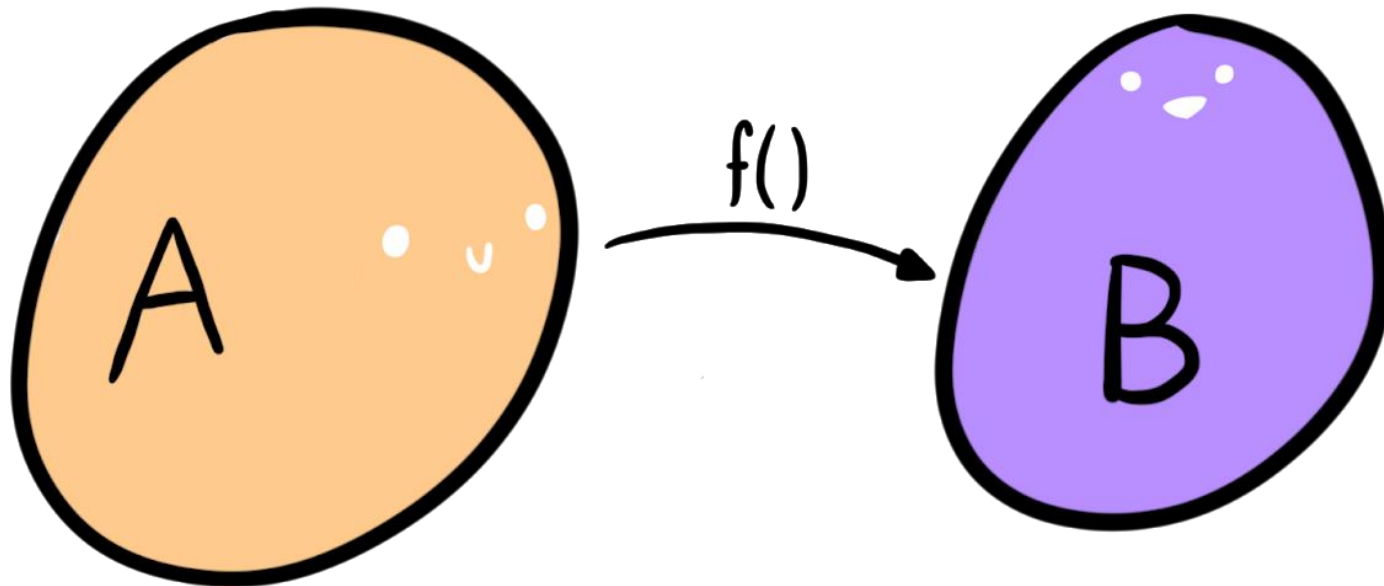
# Example: Query Service

```
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

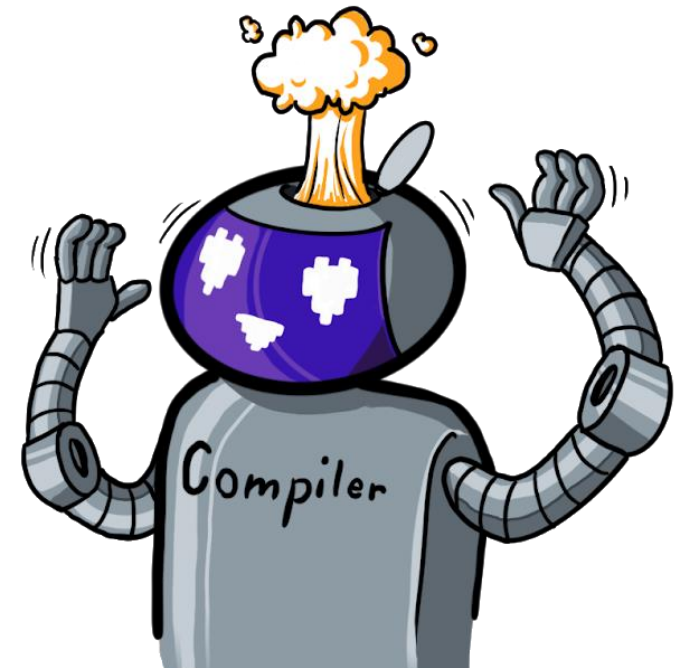
$$\begin{aligned}\text{Sum}\langle A, B \rangle &\Rightarrow C \\ &= C^{A+B} \\ &= C^A \times C^B \\ &= (A \Rightarrow C, B \Rightarrow C)\end{aligned}$$

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

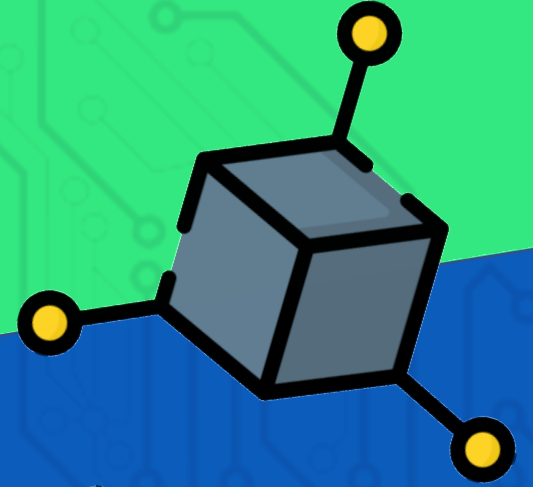


# Conclusion

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

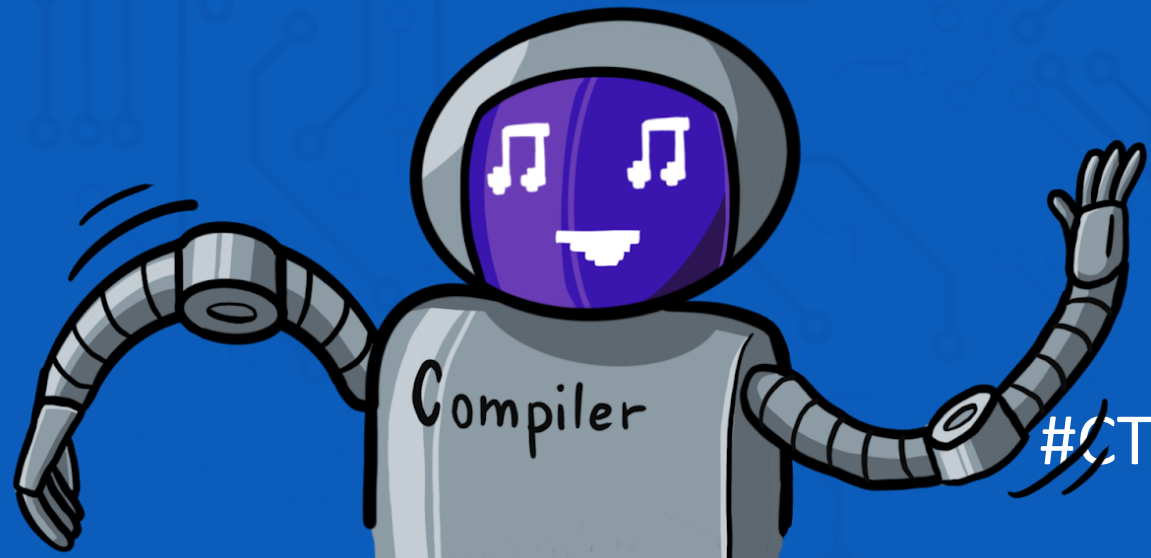


CLIENT  
TECHNOLOGY  
DAYS



# 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>