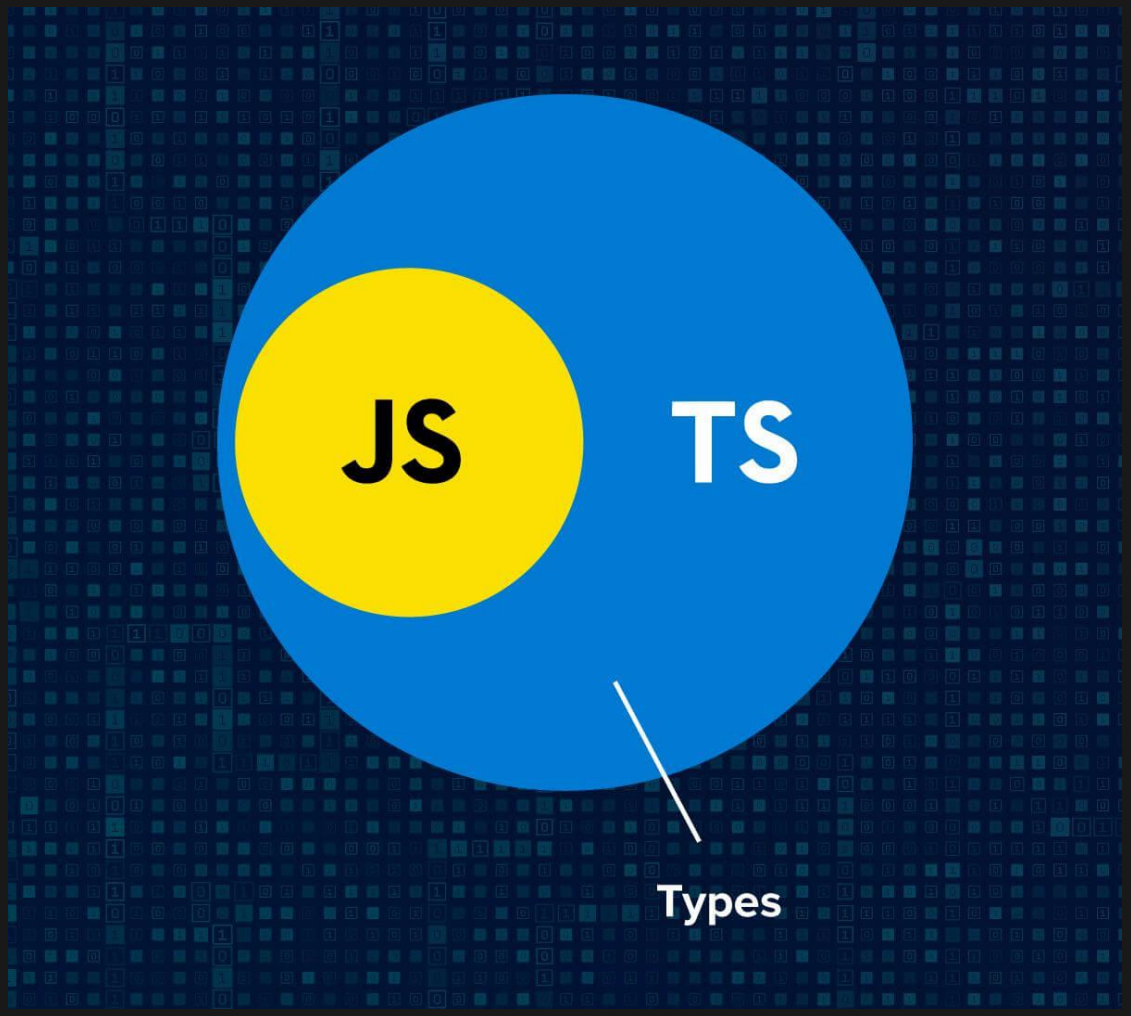
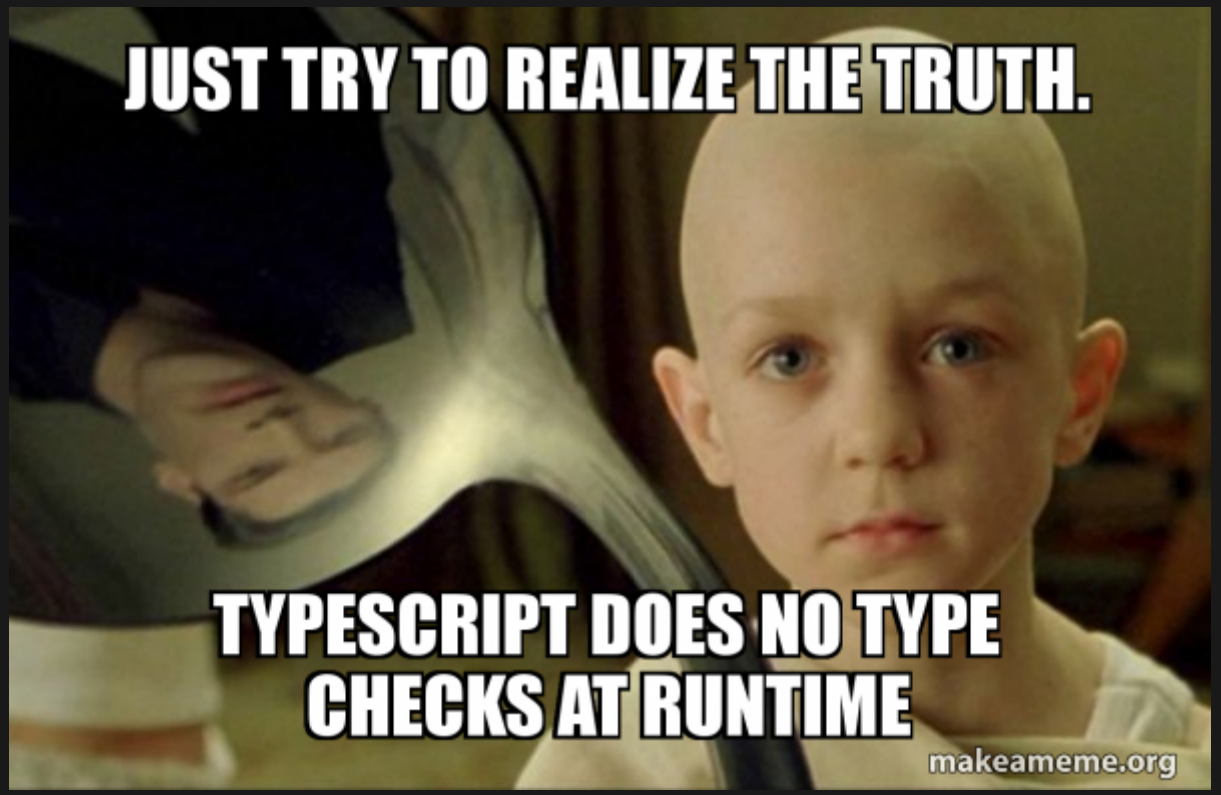
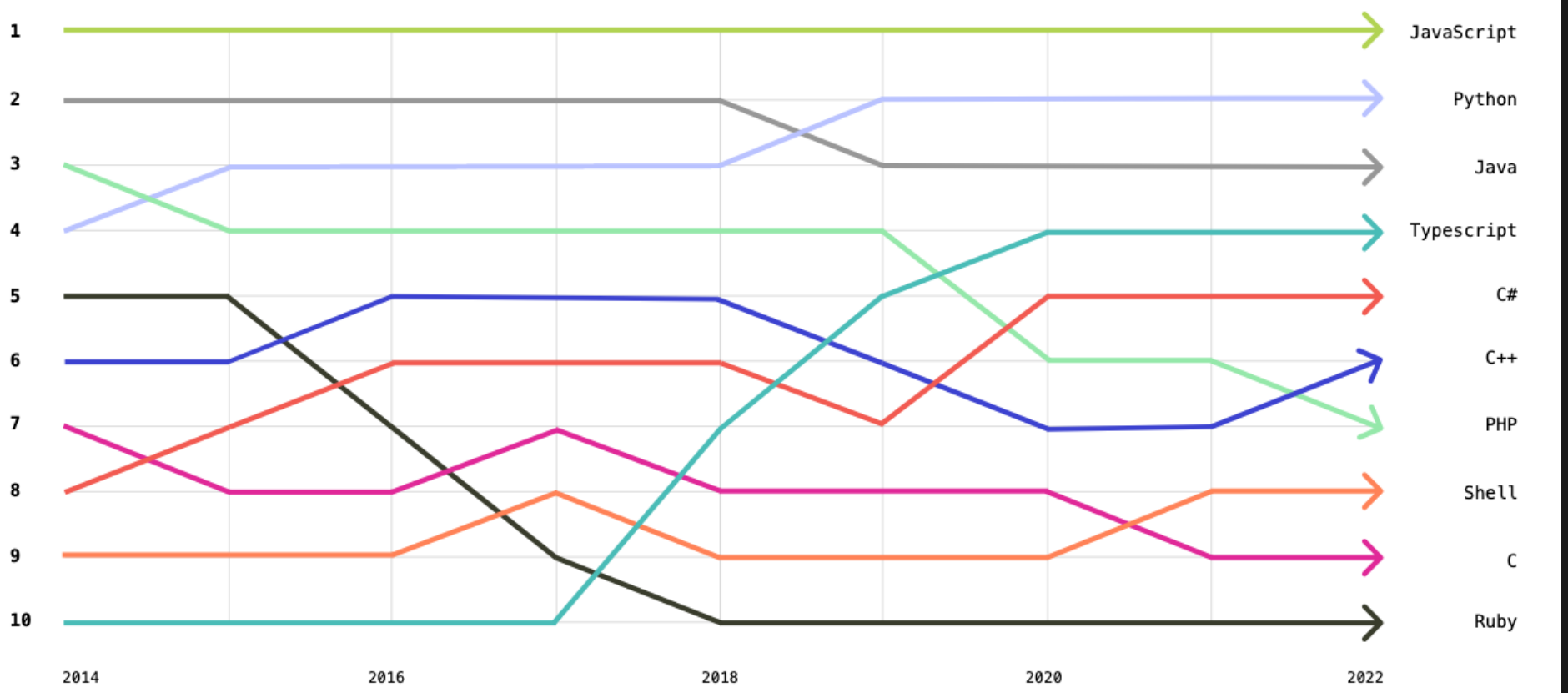

# WHAT IS TYPESCRIPT?

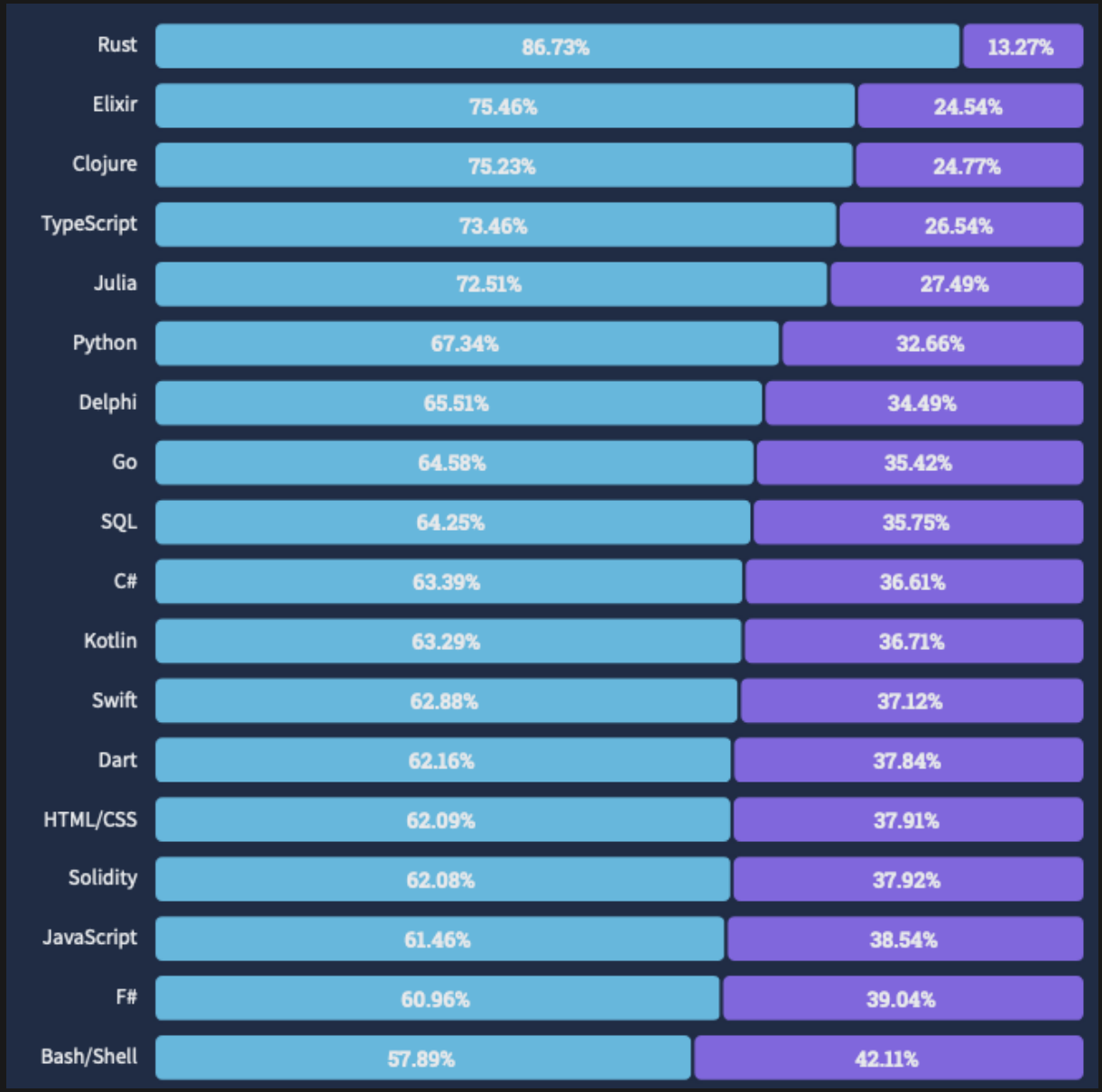

JS
TS
Types


JUST TRY TO REALIZE THE TRUTH.
TYPESCRIPT DOES NO TYPE CHECKS AT RUNTIME
makeameme.org

# TOP LANGUAGES ON GITHUB



# SO DEVELOPER SURVEY (2022): LOVED VS DREADED




[adult swim]
SHOW ME
WHAT
YOU GOT!

# TYPING VALUES

## explicitly typed

```
const myString: string = "Hello World";

const myNumber: number = 42;

const myBool: boolean = false;
```

# TYPING VALUES

## implicitly typed

```
const myString = "Hello World"; // type is string

const myNumber = 42; // type is number

const myBool = false; // type is boolean
```

# TYPING VALUES

`as const`

# TYPING VALUES

`as const`

Defines a value with an exact type

# TYPING VALUES

## as const

```
const myString = "Hello World" as const;
// ^^ type is 'Hello World'

const myNumber = 42 as const;
// ^^ type is 42

const myBool = false as const;
// ^^ type is false
```

# TYPING VALUES

## `satisfies`

# TYPING VALUES

## `satisfies`

Asserts that a value extends a type

# TYPING VALUES

## satisfies

```ts
interface AppBase {
  name: string;
  url: string;
}

const app: AppBase = {
  name: 'My App',
  url: 'https://myapp.com',

  version: '1.0.0',
  // ^^ TS complains about an unknown key...
};
```

# TYPING VALUES

## satisfies

```
1  interface AppBase {
2    name: string;
3    url: string;
4  }
5
6  const app = {
7    name: 'My App',
8    url: 'https://myapp.com',
9    version: '1.0.0',
10 } satisfies AppBase;
11
12 // we can say that app extends AppBase
```

# TYPING VALUES

## satisfies

```
interface AppBase {
  name: string;
  url: string;
}

const app = {
  name: 'My App',
  url: 'https://myapp.com',
  version: '1.0.0',
} satisfies AppBase;

// we can say that app extends AppBase
```

# interface

```
interface Shoe {
  size: number;
  brand: string;
}
```

# interface

## ... extends

```
interface Brand {
  brand: string;
}

interface Shoe extends Brand {
  size: number;
}
```

# type

```
type MyType = string | boolean;
```

**type**

... extends?

**`type`**

`... extends?`

*kind of*

# type

## & and |

```ts
interface Foo { foo: true }
interface Bar { bar: true }
interface Baz { baz: true }

type MyType = (Foo | Bar) & Baz

// equivalent
type MyType =
  | { foo: true; baz: true }
  | { bar: true; baz: true }
```

# type

## & and |

```ts
interface Foo { foo: true }
interface Bar { bar: true }
interface Baz { baz: true }

type MyType = (Foo | Bar) & Baz

// equivalent
type MyType =
  | { foo: true; baz: true }
  | { bar: true; baz: true }
```

# type

## & and |

```ts
interface Foo { foo: true }
interface Bar { bar: true }
interface Baz { baz: true }

type MyType = (Foo | Bar) & Baz

// equivalent
type MyType =
  | { foo: true; baz: true }
  | { bar: true; baz: true }
```

# GENERIC TYPES

```
1  interface MyInterface <TFoo extends string> {
2    foo: TFoo;
3    bar: boolean;
4  }
5
6  type FooType = MyInterface<'foo'>;
7
8  // equivalent
9  type FooType = {
10   foo: 'foo';
11   bar: boolean;
12 };
```

# GENERIC TYPES

```
1  interface MyInterface <TFoo extends string> {
2    foo: TFoo;
3    bar: boolean;
4  }
5
6  type FooType = MyInterface<'foo'>;
7
8  // equivalent
9  type FooType = {
10   foo: 'foo';
11   bar: boolean;
12 };
```

# GENERIC TYPES

```ts
interface MyInterface <TFoo extends string> {
  foo: TFoo;
  bar: boolean;
}

type FooType = MyInterface<'foo'>;

// equivalent
type FooType = {
  foo: 'foo';
  bar: boolean;
};
```

# Record<K extends string, V>

```ts
type MyRecord = Record<'foo' | 'bar', boolean>;

// equivalent
interface MyRecord {
  foo: boolean;
  bar: boolean;
};
```

# Record<K extends string, V>

```ts
type MyRecord = Record<'foo' | 'bar', boolean>;

// equivalent
interface MyRecord {
  foo: boolean;
  bar: boolean;
};
```

# keyof

## `keyof`

Gives the keys of an object type

# keyof

```ts
type MyRecord = Record<'foo' | 'bar', boolean>;

type MyRecordKey = keyof MyRecord;

// equivalent
type MyRecordKey = 'foo' | 'bar';
```

# keyof

```ts
type MyRecord = Record<'foo' | 'bar', boolean>;

type MyRecordKey = keyof MyRecord;

// equivalent
type MyRecordKey = 'foo' | 'bar';
```

# keyof

```ts
type MyRecord = Record<'foo' | 'bar', boolean>;

type MyRecordKey = keyof MyRecord;

// equivalent
type MyRecordKey = 'foo' | 'bar';
```

# Pick<T, K extends keyof T>

```
interface MyInterface {
  foo: 'fooValue';
  bar: 'barValue';
  baz: 'bazValue';
}

type MyPicked = Pick<MyInterface, 'foo' | 'bar'>;

// equivalent
interface MyPicked {
  foo: 'fooValue';
  bar: 'barValue';
};
```

# Pick<T, K extends keyof T>

```ts
interface MyInterface {
  foo: 'fooValue';
  bar: 'barValue';
  baz: 'bazValue';
}

type MyPicked = Pick<MyInterface, 'foo' | 'bar'>;

// equivalent
interface MyPicked {
  foo: 'fooValue';
  bar: 'barValue';
};
```

# Pick<T, K extends keyof T>

```ts
interface MyInterface {
  foo: 'fooValue';
  bar: 'barValue';
  baz: 'bazValue';
}

type MyPicked = Pick<MyInterface, 'foo' | 'bar'>;

// equivalent
interface MyPicked {
  foo: 'fooValue';
  bar: 'barValue';
};
```

# TYPE GUARDS

```ts
type Fruit = '🍎' | '🍊' | '🍌';

const isApple = (x: Fruit): x is '🍎' => x === '🍎';

declare const myFruit: Fruit;

if (isApple(myFruit)) {
  // myFruit is '🍎' in this block
}
```

# TYPE GUARDS

```ts
type Fruit = '🍎' | '🍊' | '🍌';

const isApple = (x: Fruit): x is '🍎' => x === '🍎';

declare const myFruit: Fruit;

if (isApple(myFruit)) {
  // myFruit is '🍎' in this block
}
```

# TYPE GUARDS

```ts
type Fruit = '🍎' | '🍊' | '🍌';

const isApple = (x: Fruit): x is '🍎' => x === '🍎';

declare const myFruit: Fruit;

if (isApple(myFruit)) {
  // myFruit is '🍎' in this block
}
```

# TYPE GUARDS

```ts
type Fruit = '🍎' | '🍊' | '🍌';

const isApple = (x: Fruit): x is '🍎' => x === '🍎';

declare const myFruit: Fruit;

if (isApple(myFruit)) {
  // myFruit is '🍎' in this block
}
```

# GENERIC FUNCTIONS

```
1 const identity = <T>(value: T): T => {
2   return value;
3 }
4
5 const myNumber = identity<number>(42);  // type is number
6
7 const myString = identity("Hello");     // type is string
```

# GENERIC FUNCTIONS

```
1 const identity = <T>(value: T): T => {
2   return value;
3 }
4
5 const myNumber = identity<number>(42);  // type is number
6
7 const myString = identity("Hello");     // type is string
```

# GENERIC FUNCTIONS

```
1 const identity = <T>(value: T): T => {
2   return value;
3 }
4
5 const myNumber = identity<number>(42);  // type is number
6
7 const myString = identity("Hello");     // type is string
```

# GENERIC FUNCTIONS

## with const

```
1 const identity = <const T>(value: T): T => {
2   return value;
3 }
4
5 const myNumber = identity(42);      // type is 42
6
7 const myString = identity("Hello"); // type is 'Hello'
```

# GENERIC FUNCTIONS

## with const

```
1 const identity = <const T>(value: T): T => {
2   return value;
3 }
4
5 const myNumber = identity(42);      // type is 42
6
7 const myString = identity("Hello"); // type is 'Hello'
```

# GENERIC FUNCTIONS

## with const

```
1 const identity = <const T>(value: T): T => {
2   return value;
3 }
4
5 const myNumber = identity(42);       // type is 42
6
7 const myString = identity("Hello"); // type is 'Hello'
```

# GENERIC FUNCTIONS

## and extends

```
1 const identity = <const T extends string>(value: T): T => {
2   return value;
3 }
4
5 const myNumber = identity(42);
6 // ^^ TS complains number is not assignable to string
7
8 const myString = identity("Hello"); // type is 'Hello'
```

# GENERIC FUNCTIONS

## and extends

```
1 const identity = <const T extends string>(value: T): T => {
2   return value;
3 }
4
5 const myNumber = identity(42);
6 // ^^ TS complains number is not assignable to string
7
8 const myString = identity("Hello"); // type is 'Hello'
```

# GENERIC FUNCTIONS

## and extends

```
1 const identity = <const T extends string>(value: T): T => {
2   return value;
3 }
4
5 const myNumber = identity(42);
6 // ^^ TS complains number is not assignable to string
7
8 const myString = identity("Hello"); // type is 'Hello'
```

# ONE MORE THING...

# ONE MORE THING...

# TYPESCRIPT CODE GOLF

```ts
// Construct the following type

type Output =
  | [0, 0, 0]
  | [0, 0, 1]
  | [0, 1, 0]
  | [0, 1, 1]
  | [1, 0, 0]
  | [1, 0, 1]
  | [1, 1, 0]
  | [1, 1, 1]
```

# TYPESCRIPT CODE GOLF

```ts
// Construct the following type

type Output =
  | [0, 0, 0]
  | [0, 0, 1]
  | [0, 1, 0]
  | [0, 1, 1]
  | [1, 0, 0]
  | [1, 0, 1]
  | [1, 1, 0]
  | [1, 1, 1]
```

Shortest answer wins