Structural Typing

Structural Typing

```
type Point = {
  x: number;
  y: number;
};

// Imaginary predicate
{ p |
  p is an object and
  p is not null and
  p.x is a number and
  p.y is a number }
```

Structural Typing

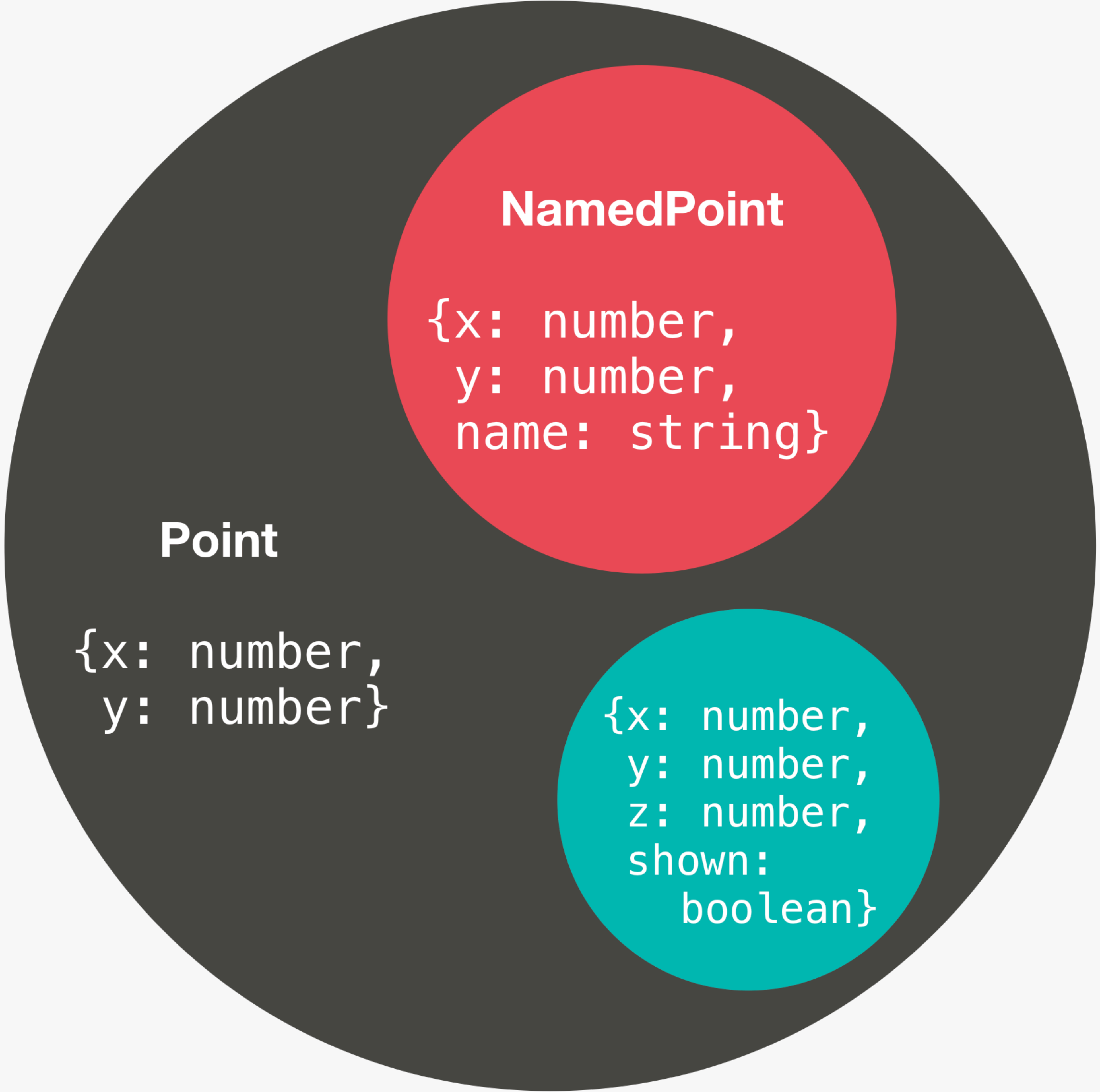
```
type Point = {
  x: number;
  y: number;
};

// Imaginary predicate
{ p |
  p is an object and
  p is not null and
  p.x is a number and
  p.y is a number }
```

Structural Compatibility

```
type Point = {
  x: number;
  y: number;
};

type NamedPoint = {
  name: string;
  x: number;
  y: number;
};
```



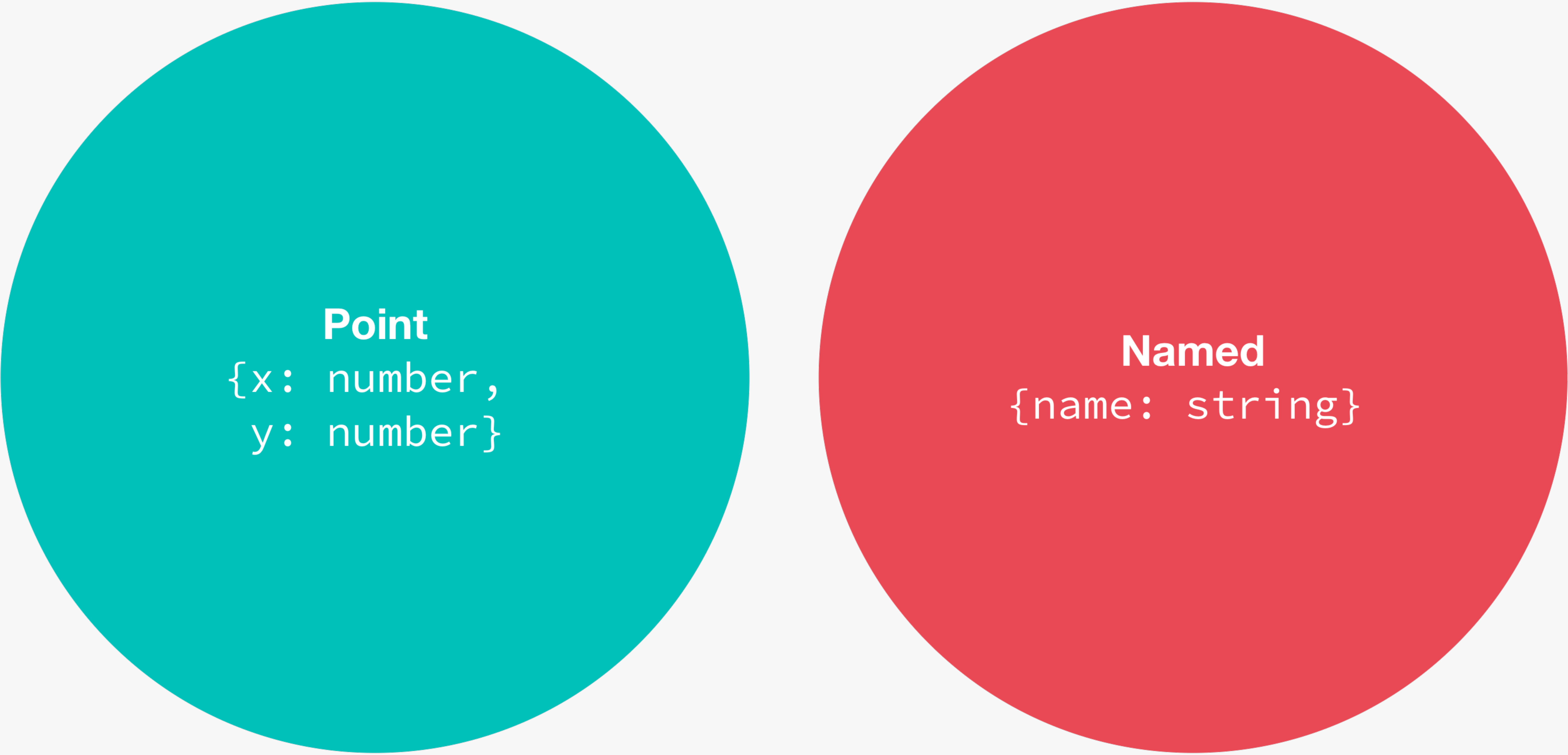
NamedPoint

```
{x: number,
 y: number,
 name: string}
```

Point

```
{x: number,
 y: number}
```

```
{x: number,
 y: number,
 z: number,
 shown:
   boolean}
```



Point

```
{x: number,
 y: number}
```

Named

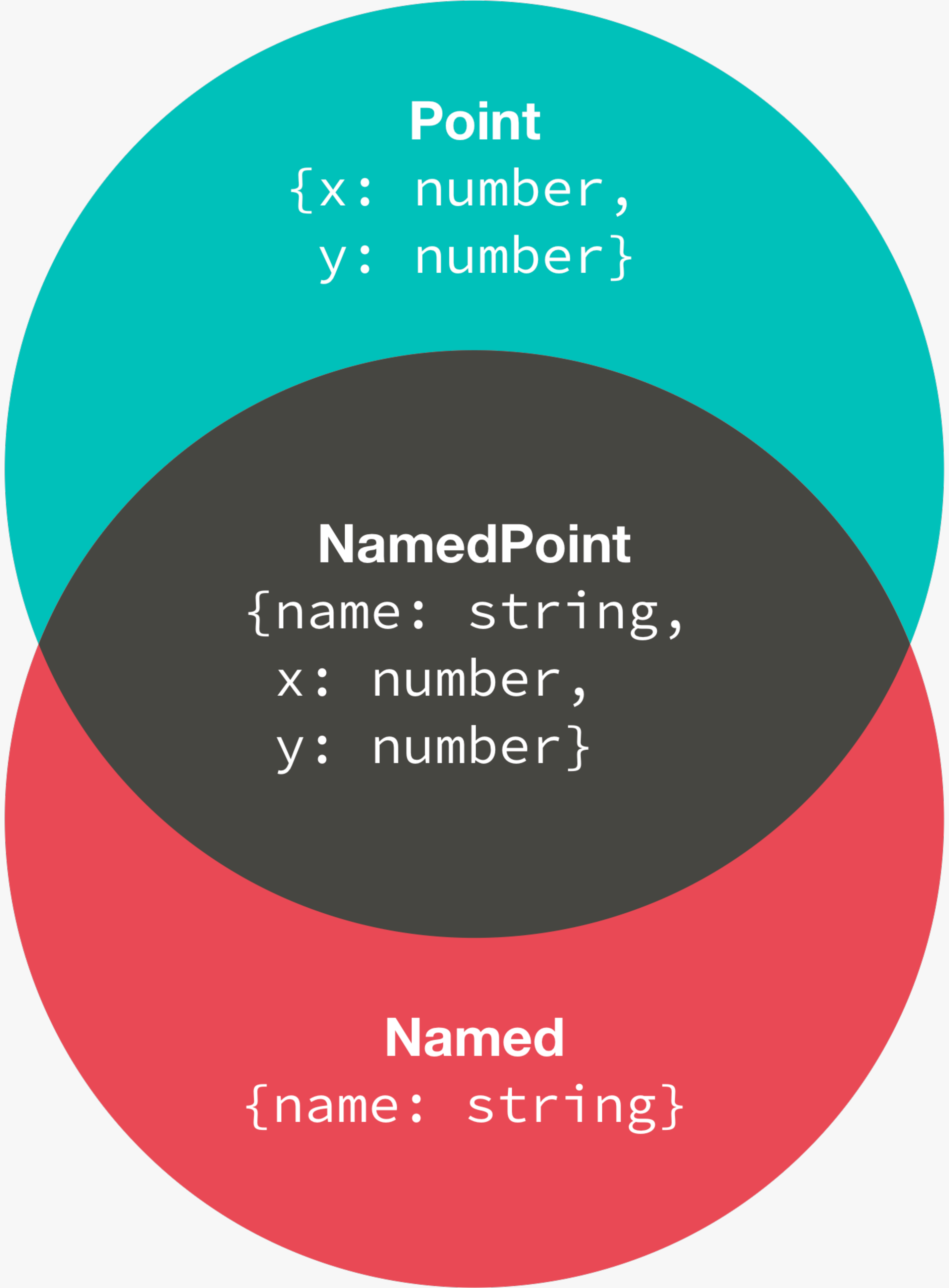
```
{name: string}
```

Types have multiple supertypes

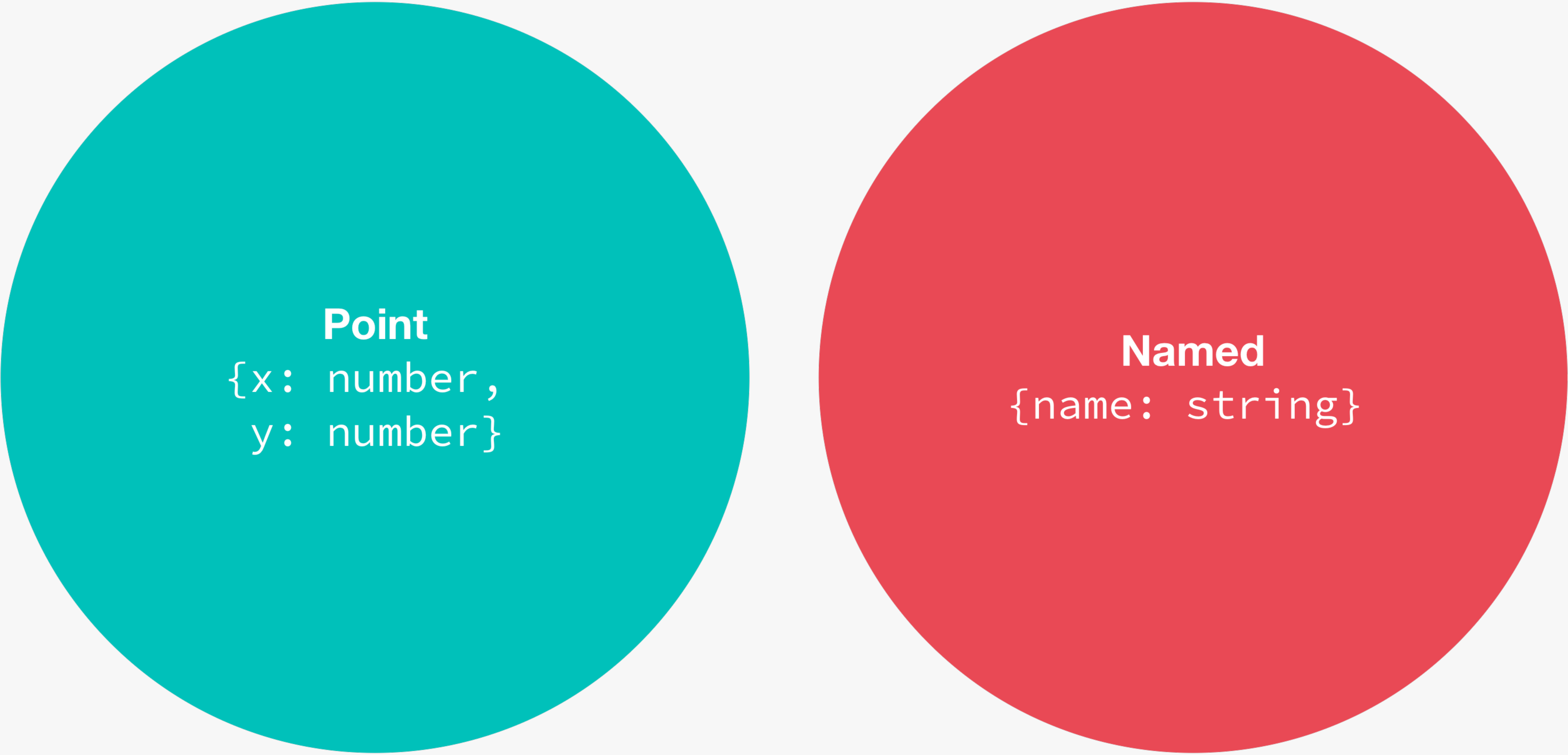
```
const aNamedPoint = {
  name: "there",
  x: 32,
  y: 14
};

// printName(name: Named): void
printName(aNamedPoint);

// plot(point: Point): void
plot(aNamedPoint);
```



Types as Sets

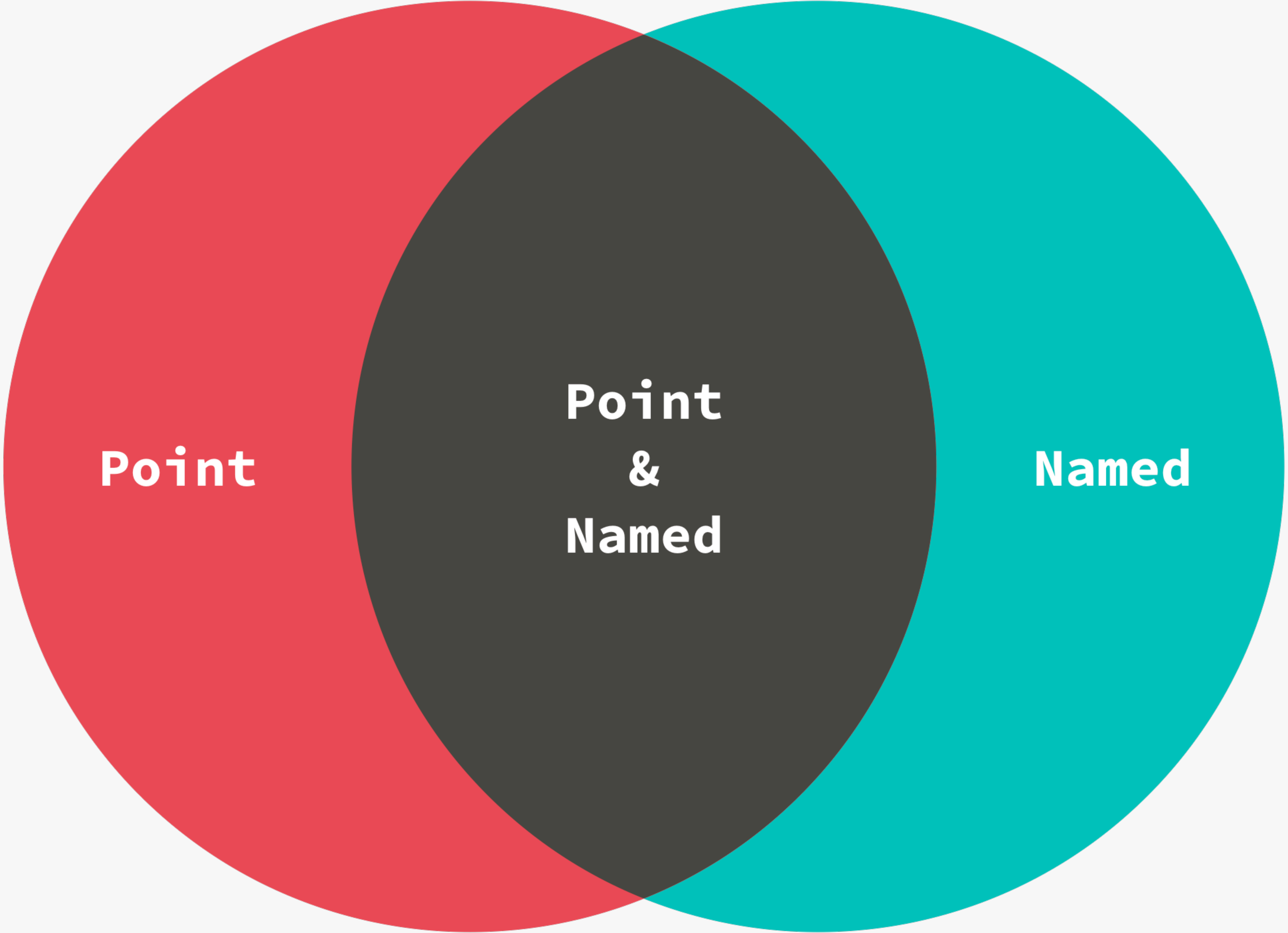


Point

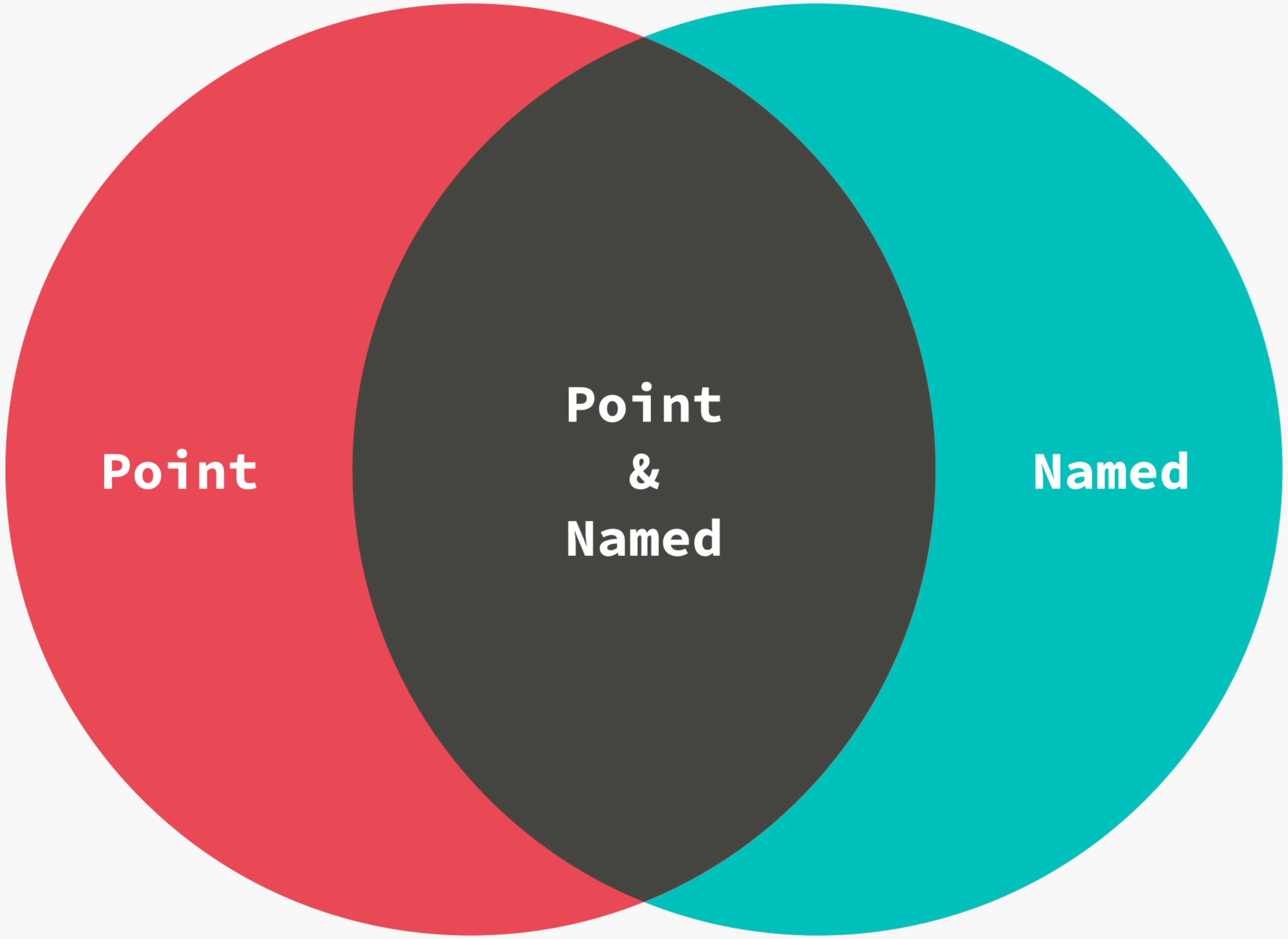
```
{x: number,
 y: number}
```

Named

```
{name: string}
```



Point
Point
&
Named
Named

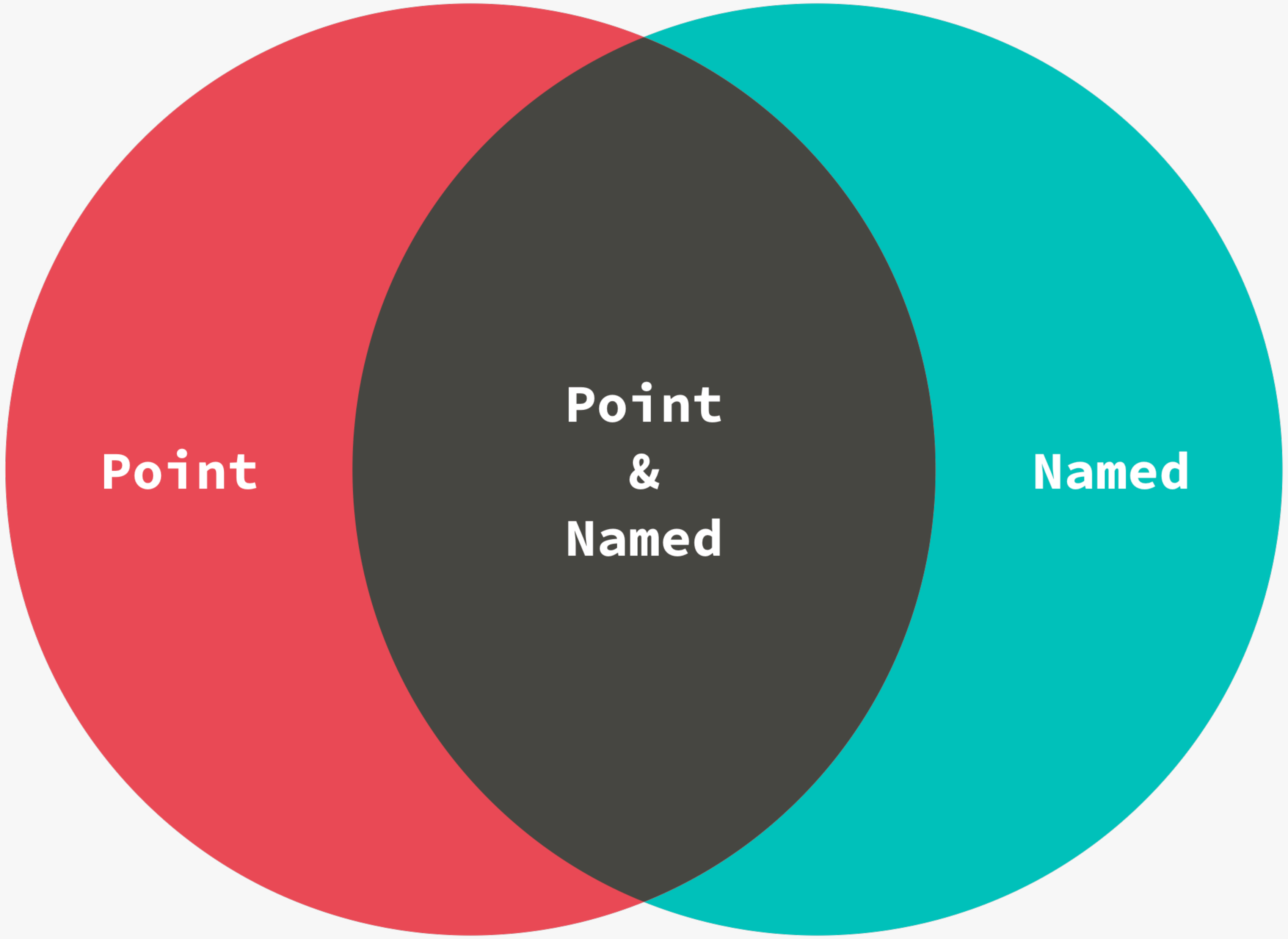


Type Intersections

```
type NamedPoint = Point & Named;

// Imaginary predicate
{ x |
  x compatibleWith Point
  && x compatibleWith Named }

// We can access properties present in ANY constituent
myNamedPoint.x // valid
myNamedPoint.name // valid
```

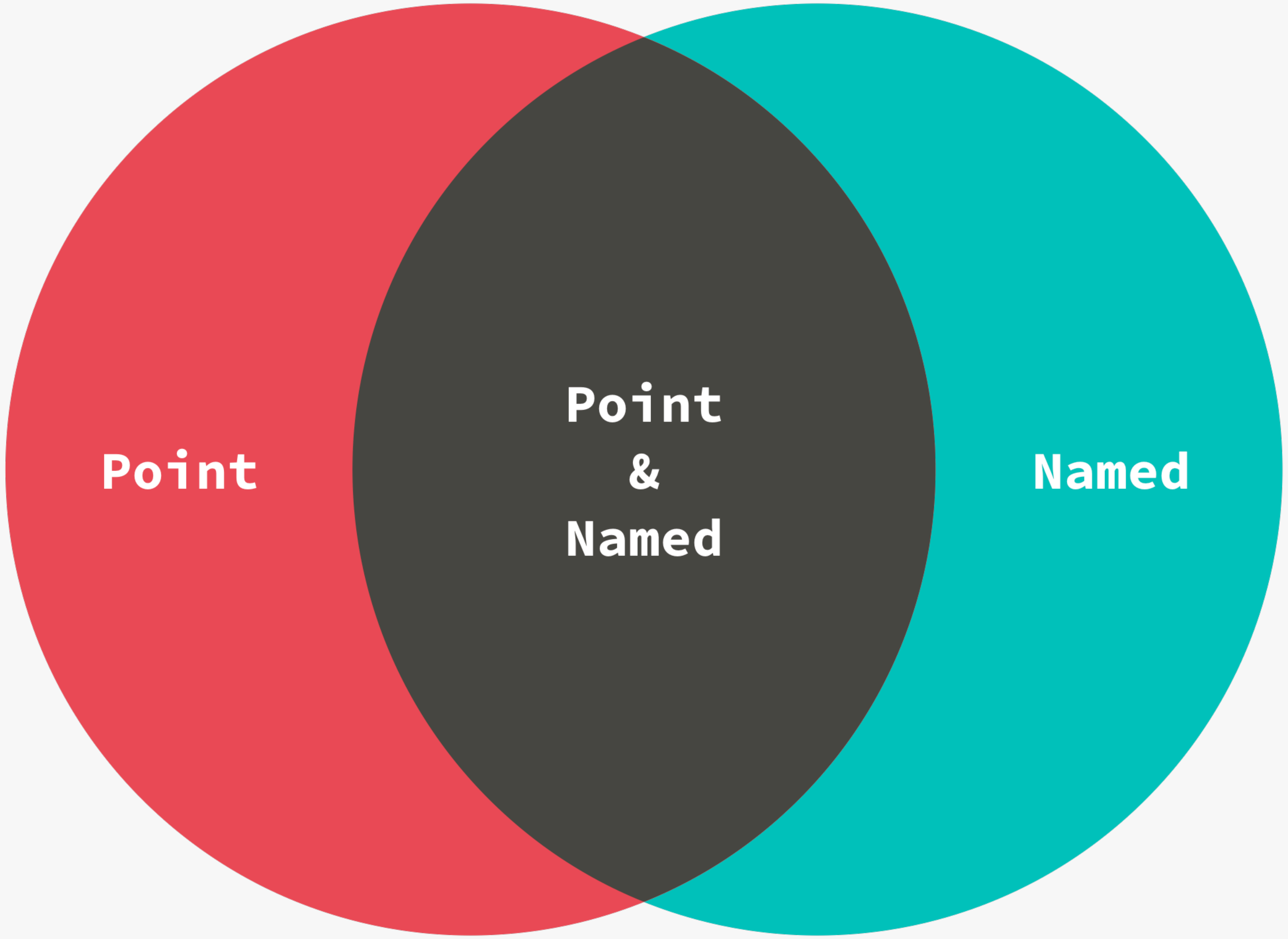


Type Intersections

```
type NamedPoint = Point & Named;

// Imaginary predicate
{ x |
  x compatibleWith Point
  && x compatibleWith Named }

// We can access properties present in ANY constituent
myNamedPoint.x // valid
myNamedPoint.name // valid
```



Type Intersections

```
type NamedPoint = Point & Named;

// Imaginary predicate
{ x |
  x compatibleWith Point
  && x compatibleWith Named }

// We can access properties present in ANY constituent
myNamedPoint.x // valid
myNamedPoint.name // valid
```

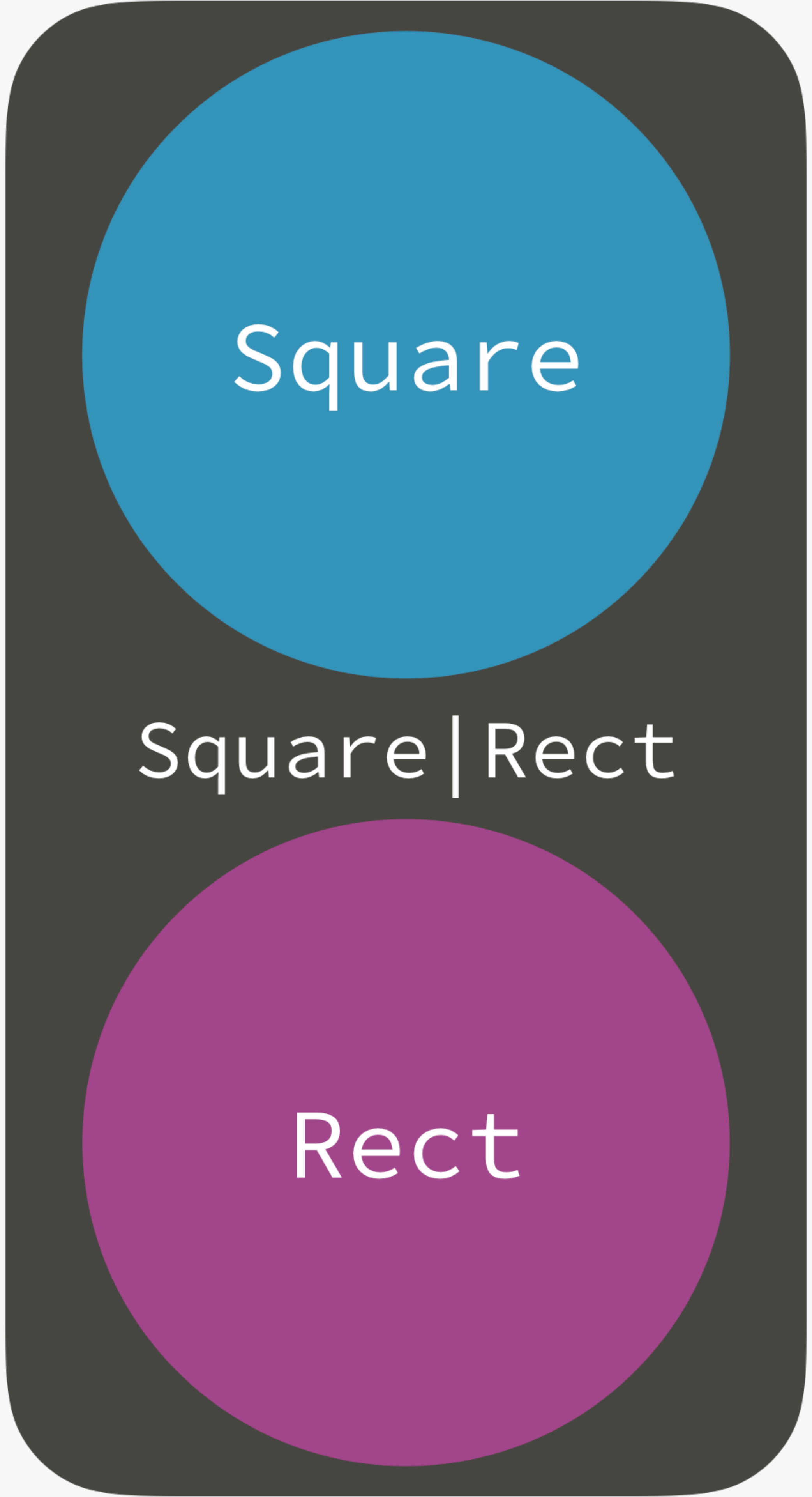
Unions

```
type Square = {
  color: string,
  size: number
};
type Rect = {
  color: string,
  width: number,
  height: number
};

type Shape = Square | Rect;

{ x | x compatibleWith Square OR
      x compatibleWith Rect }

// We can access properties common to ALL cases
someShape.color // valid! common to both
someShape.width // error! Not common to all cases
```



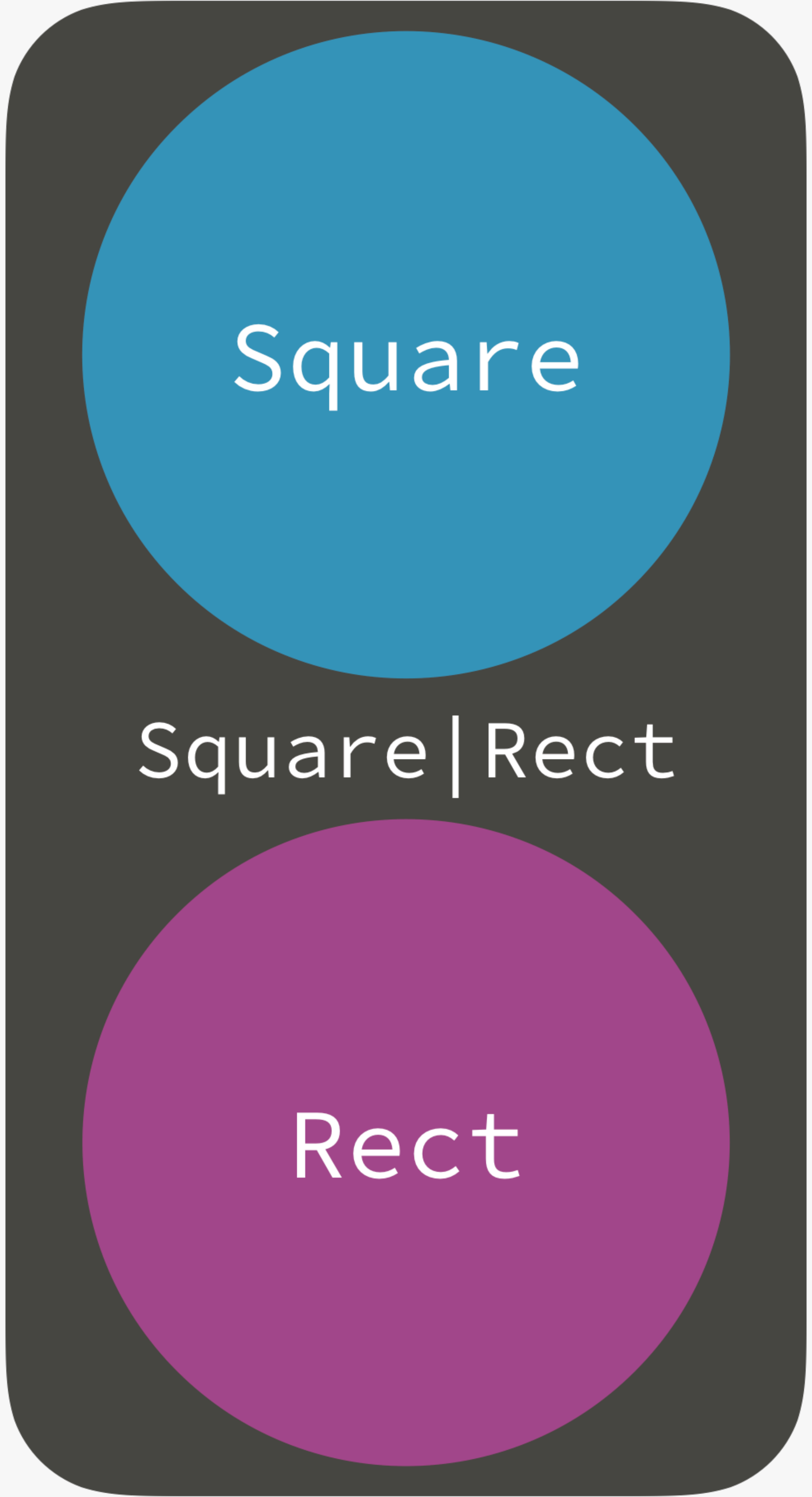
Unions

```
type Square = {
  color: string,
  size: number
};
type Rect = {
  color: string,
  width: number,
  height: number
};

type Shape = Square | Rect;

{ x | x compatibleWith Square OR
      x compatibleWith Rect }

// We can access properties common to ALL cases
someShape.color // valid! common to both
someShape.width // error! Not common to all cases
```



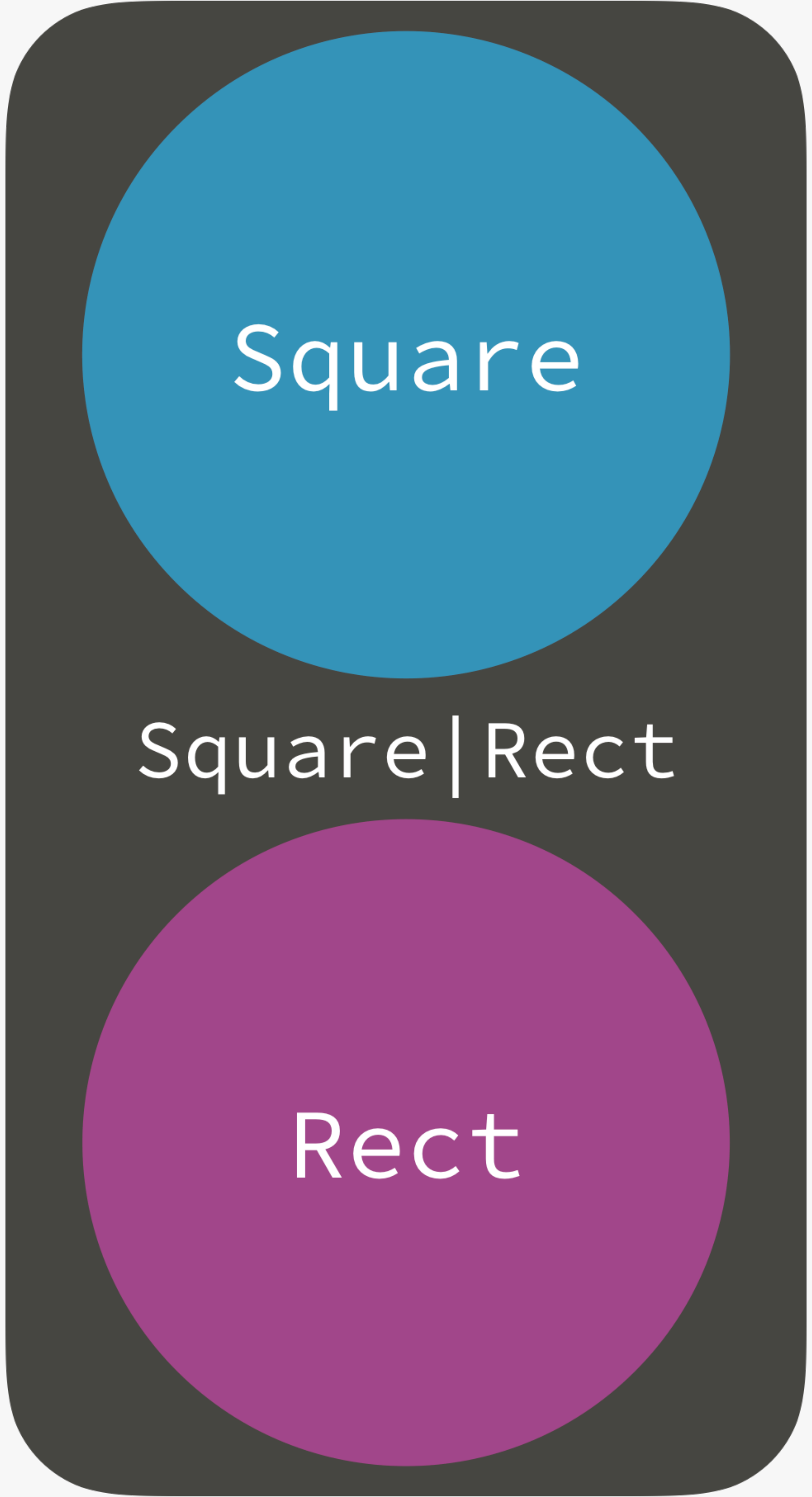
Unions

```
type Square = {
  color: string,
  size: number
};
type Rect = {
  color: string,
  width: number,
  height: number
};

type Shape = Square | Rect;

{ x | x compatibleWith Square OR
      x compatibleWith Rect }

// We can access properties common to ALL cases
someShape.color // valid! common to both
someShape.width // error! Not common to all cases
```

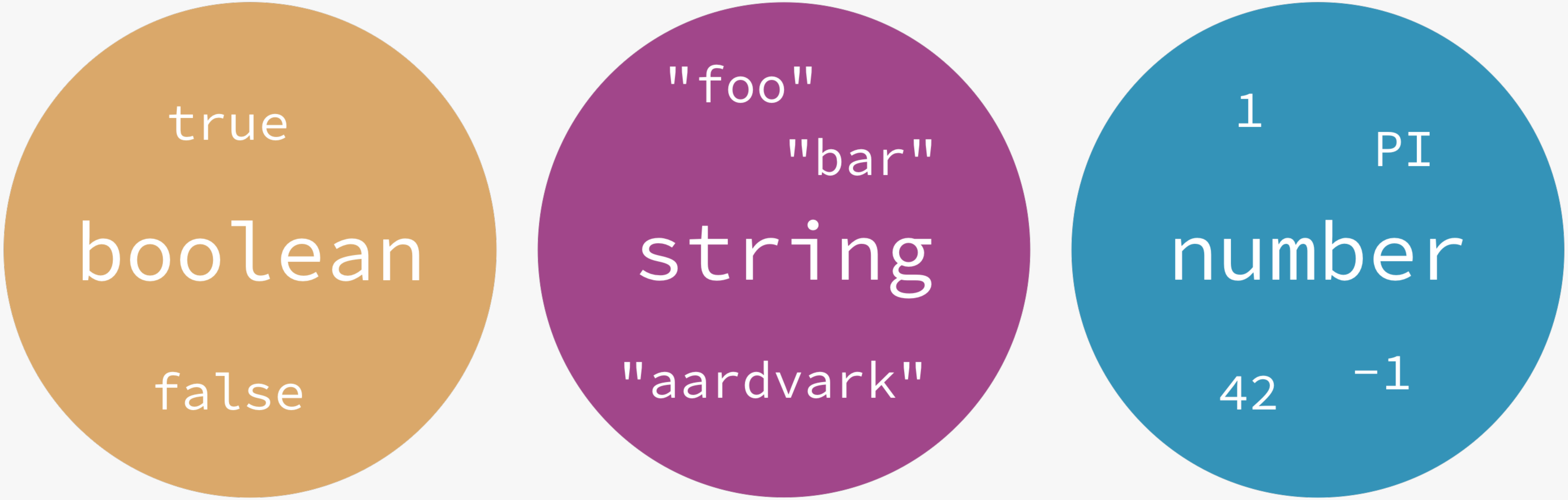


So what?

boolean

string

number



true

boolean

false

"foo"

"bar"

string

"aardvark"

1

PI

number

42

-1

Literal Types

```
let aFoo: 'foo';
let aTrue: true;
let a42: 42;
let manyFoos: 'foo'[];
```

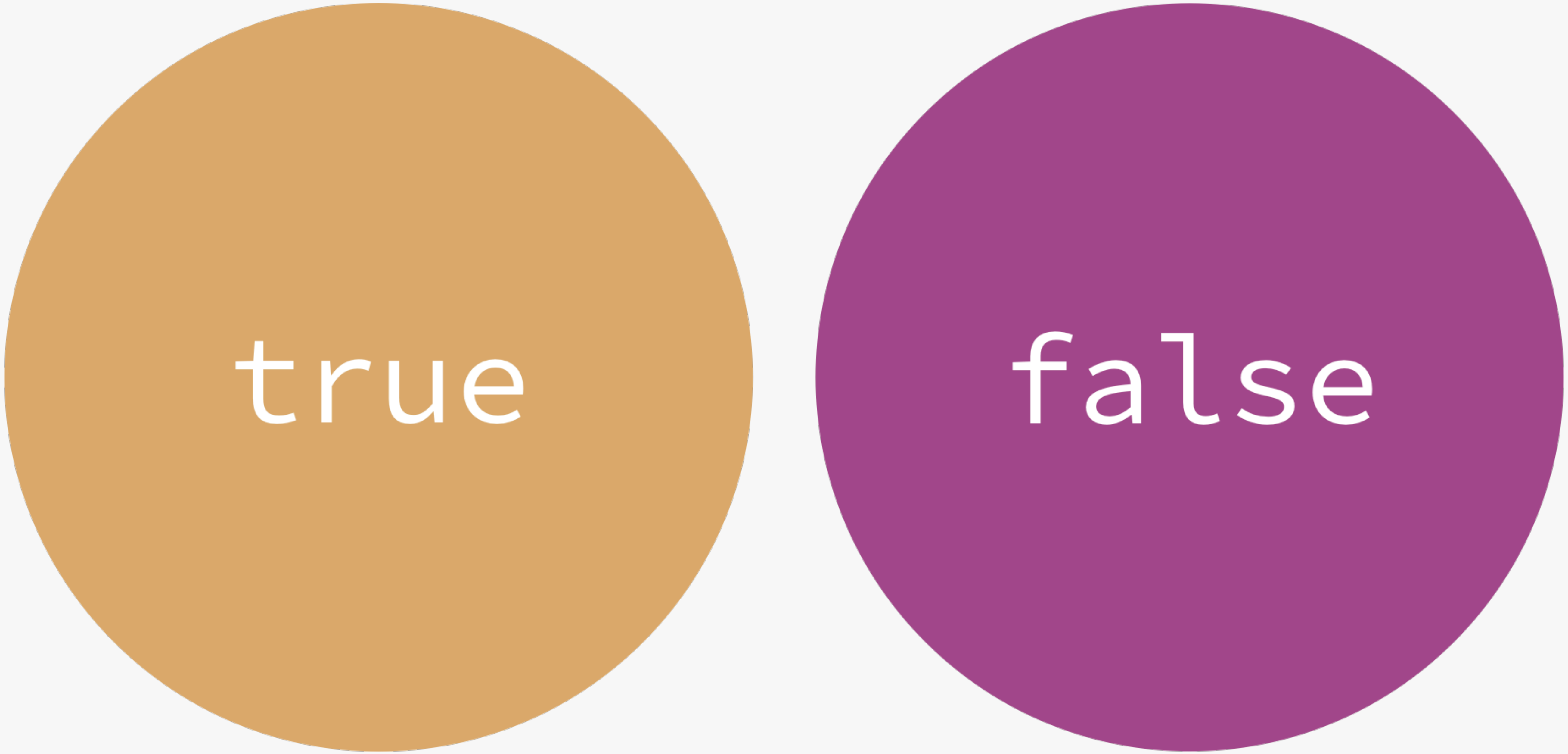
Literal Types

```
let aFoo: 'foo';
let aTrue: true;
let a42: 42;
let manyFoos: 'foo'[];

// All of these are invalid, uninitialized accesses!
aFoo; aTrue; a42; manyFoos;

aFoo = 'foo'; // Great!
aTrue = false; // Error, false is not assignable to true
manyFoos = ['foo','foo','foo', 'bar'] // 'bar' not assignable to 'foo'
```

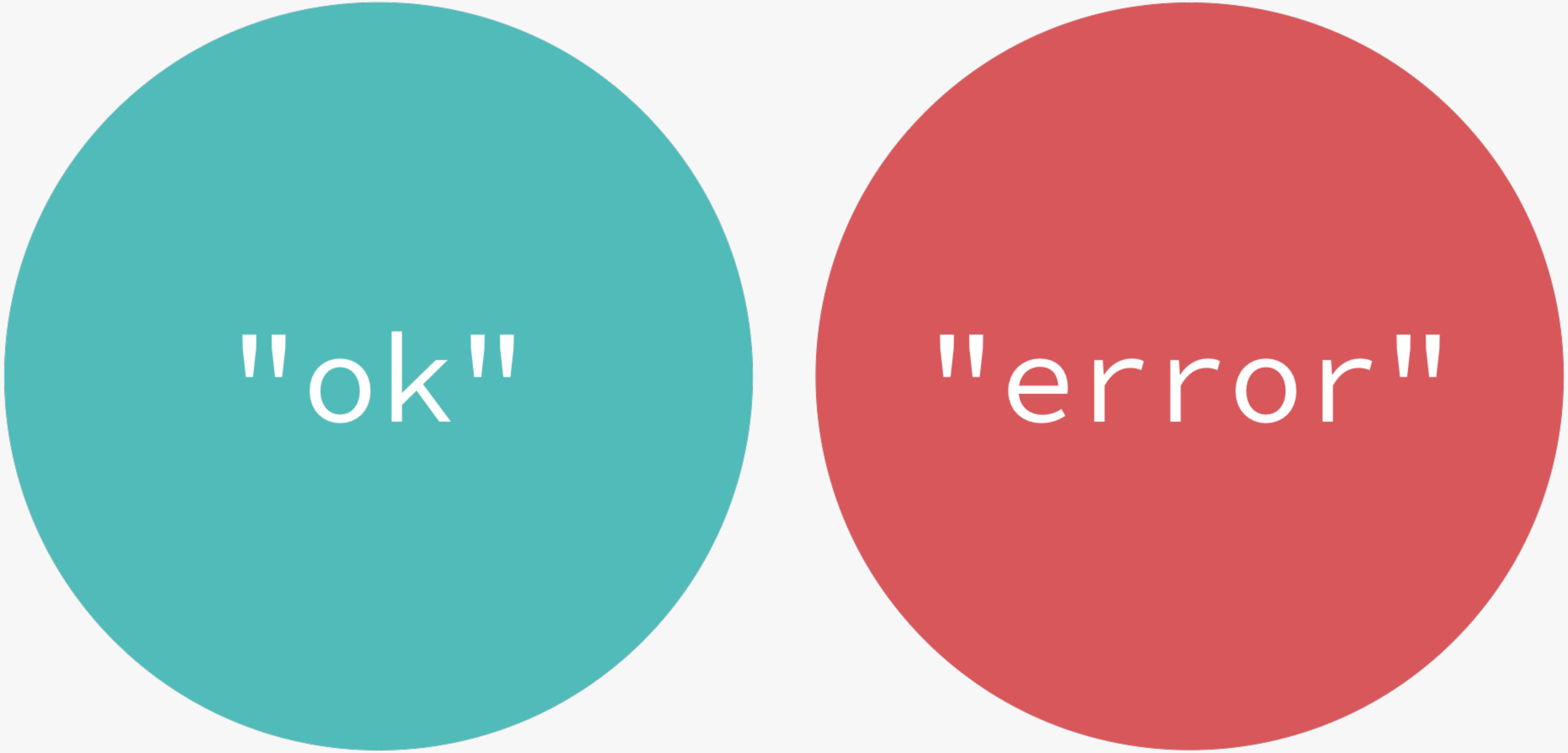
```
type MyBoolean = true | false
```



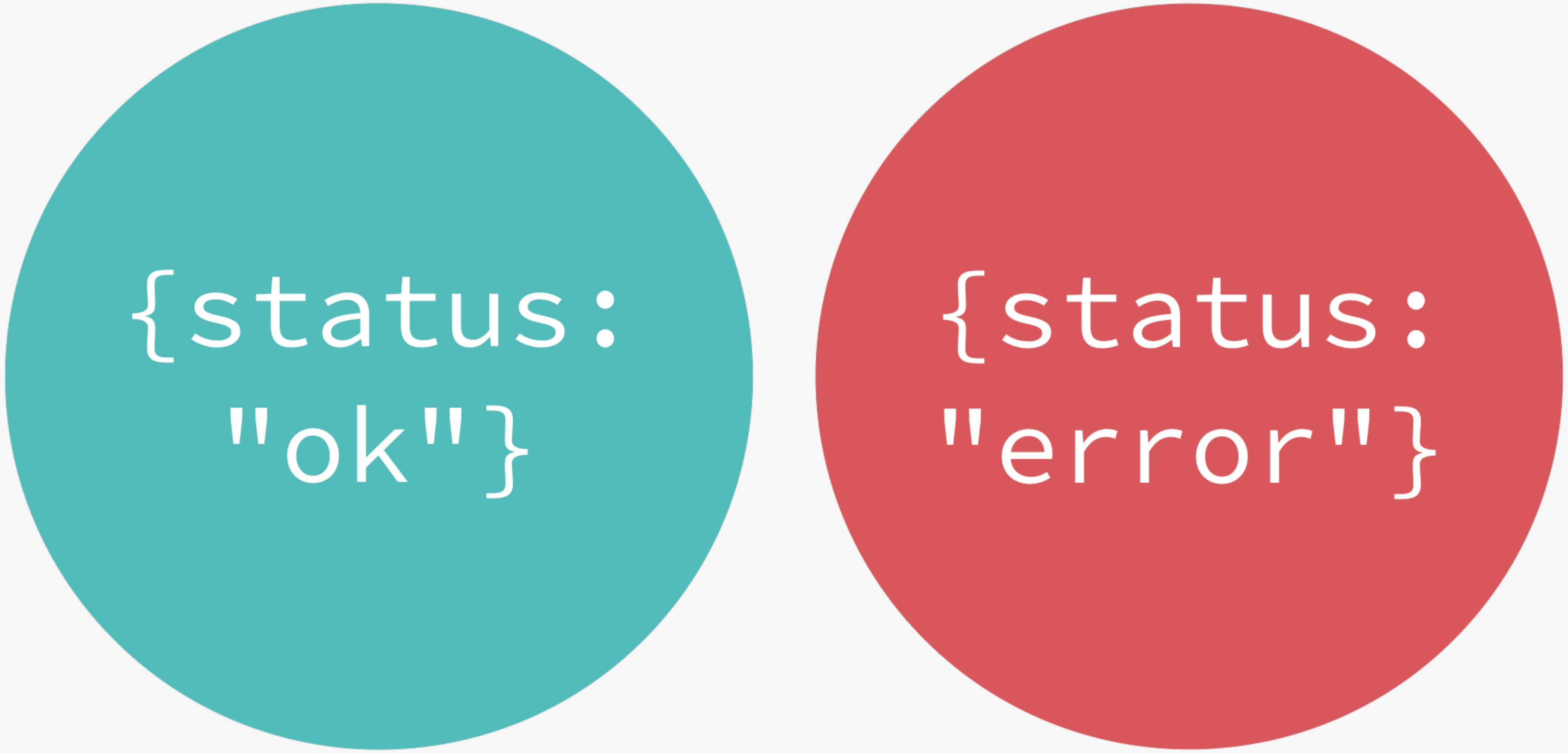
true

false

```
type Result = "ok" | "error"
```



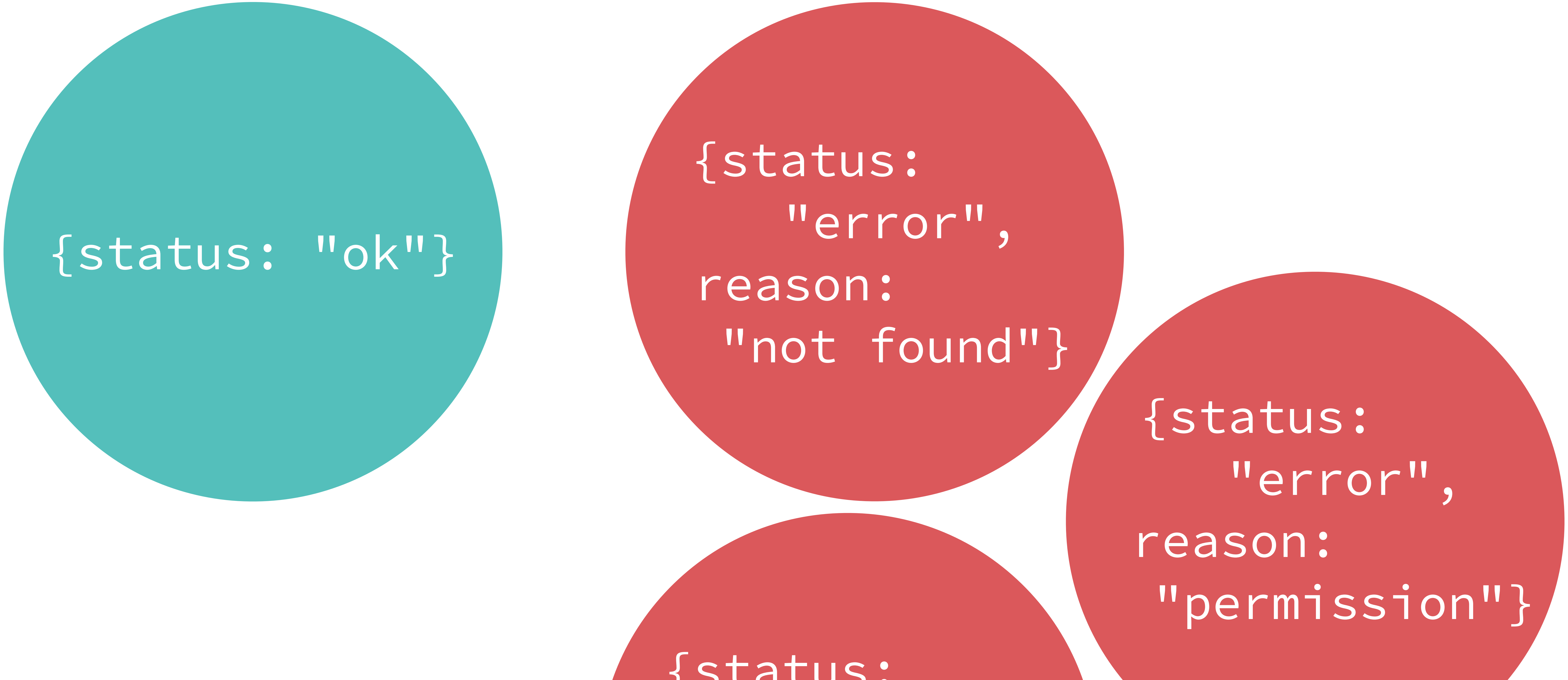
```
type Result =
    | { status: "ok"}
    | { status: "error"  };
```



{status: "ok"}

{status: "error"}

```
type Result =
    | { status: "ok"}
    | { status: "error",
        reason:  string };
```



```
{status: "ok"}
```

```
{status:
    "error",
reason:
 "not found"}
```

```
{status:
    "error",
reason:
 "permission"}
```

```
{status:
```

Control-flow based
Type Analysis

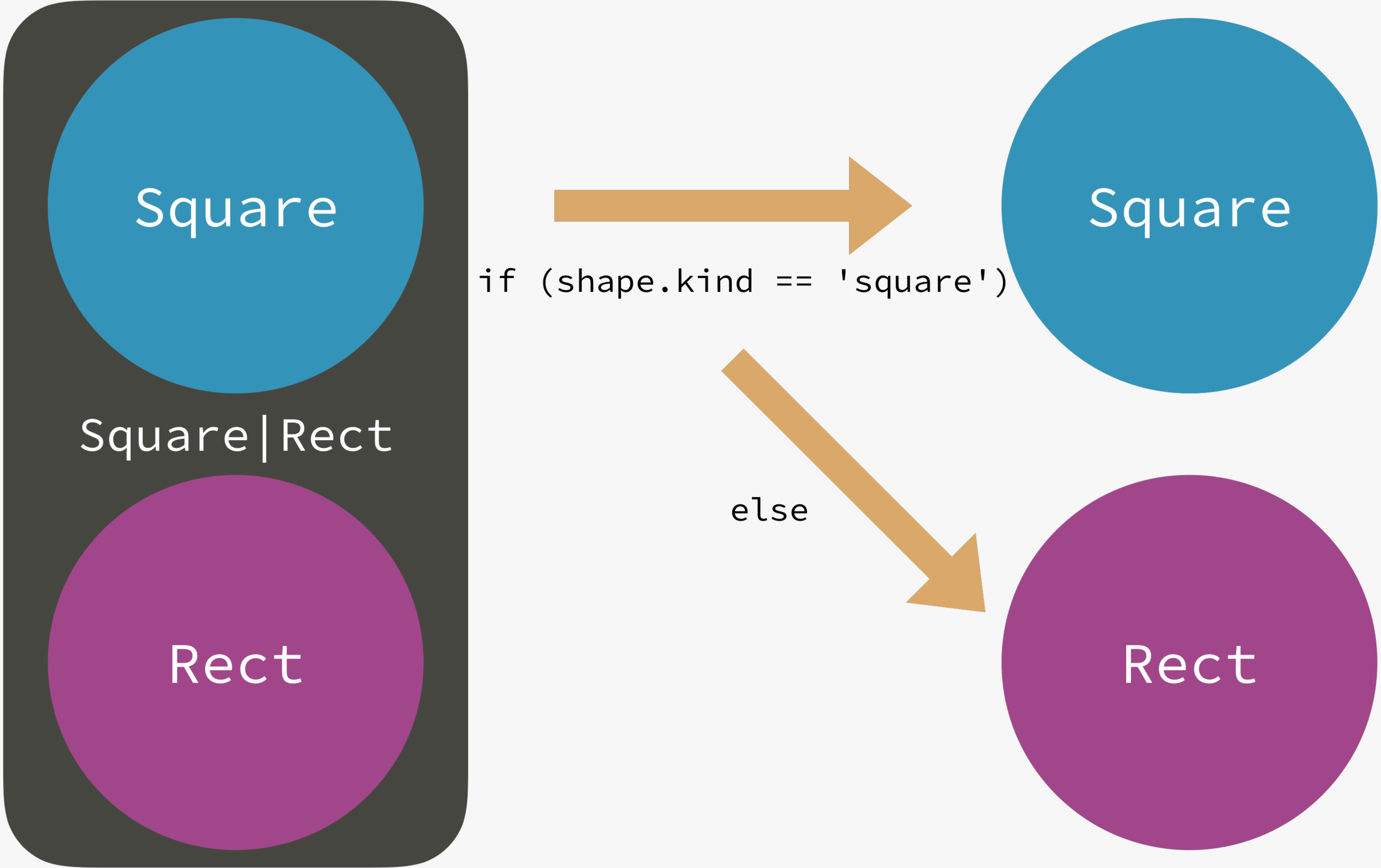
Discriminated Unions / Sum Types

```
type Square = {
  kind: "square";  // <- New addition
  size: number;
  color: string;
};

type Rectangle = {
  kind: "rectangle";  // <- New addition
  width: number;
  height: number;
  color: string;
};

type Shape = Square | Rectangle;

aShape.kind // 'square' | 'rectangle'
```



Square

Square|Rect

Rect

`if (shape.kind == 'square')`

Square

`else`

Rect

Discriminated Unions

```
function area(s: Shape): number {
  // Common to all cases, safe to access
  switch (s.kind) {
    case "square":
      // Now proven to be a Square
      return s.size * s.size;
    case "rectangle":
      // Now proven to be a rectangle
      return s.width * s.height;
  }
  // All possible cases have been handled!
  // (Otherwise, bad implicit return of undefined!)
}
```

Discriminated Unions

```
function area(s: Shape): number {
  // Common to all cases, safe to access
  switch (s.kind) {
    case "square":
      // Now proven to be a Square
      return s.size * s.size;
    case "rectangle":
      // Now proven to be a rectangle
      return s.width * s.height;
  }
  // All possible cases have been handled!
  // (Otherwise, bad implicit return of undefined!)
}
```

Discriminated Unions

```
function area(s: Shape): number {
  // Common to all cases, safe to access
  switch (s.kind) {
    case "square":
      // Now proven to be a Square
      return s.size * s.size;
    case "rectangle":
      // Now proven to be a rectangle
      return s.width * s.height;
  }
  // All possible cases have been handled!
  // (Otherwise, bad implicit return of undefined!)
}
```

Discriminated Unions

```
function area(s: Shape): number {
  // Common to all cases, safe to access
  switch (s.kind) {
    case "square":
      // Now proven to be a Square
      return s.size * s.size;
    case "rectangle":
      // Now proven to be a rectangle
      return s.width * s.height;
  }
  // All possible cases have been handled!
  // (Otherwise, bad implicit return of undefined!)
}
```

Discriminated Unions

```
function area(s: Shape): number {
  // Common to all cases, safe to access
  switch (s.kind) {
    case "square":
      // Now proven to be a Square
      return s.size * s.size;
    case "rectangle":
      // Now proven to be a rectangle
      return s.width * s.height;
  }
  // All possible cases have been handled!
  // (Otherwise, bad implicit return of undefined!)
}
```

All Together

```
// Define cases with discriminant and
// case-specific properties
type Square = {
  kind: "square";  // <- New addition
  size: number;
};

type Rectangle = {
  kind: "rectangle";  // <- New addition
  width: number;
  height: number;
};

// Common properties can be defined once
type WithColor = { color: string }

// Construct the complete type from components
type Shape = (Square | Rectangle) & WithColor;
```

All Together

```
// Define cases with discriminant and
// case-specific properties
type Square = {
  kind: "square";  // <- New addition
  size: number;
};

type Rectangle = {
  kind: "rectangle";  // <- New addition
  width: number;
  height: number;
};

// Common properties can be defined once
type WithColor = { color: string }

// Construct the complete type from components
type Shape = (Square | Rectangle) & WithColor;
```

All Together

```
// Define cases with discriminant and
// case-specific properties
type Square = {
  kind: "square";  // <- New addition
  size: number;
};

type Rectangle = {
  kind: "rectangle";  // <- New addition
  width: number;
  height: number;
};

// Common properties can be defined once
type WithColor = { color: string }

// Construct the complete type from components
type Shape = (Square | Rectangle) & WithColor;
```

Next

Practice with:

- Literal types
- Unions
- Intersections
- Control-flow based type analysis