Microsoft

TypeScript: Rethinking Type Systems with JavaScript

Daniel Rosenwasser
@drosenwasser

Retrofitting a type system into a language not designed for typechecking in mind can be tricky; ideally, language design should go hand-in-hand with type system design.

1.3. Type Systems and Language Design
Types and Programing Languages
Benjamin C. Pierce

What is TypeScript?

What was missing?

1. Modern constructs

2. Type-checking

3. Tooling

TypeScript = JavaScript

TypeScript = Modern JavaScript

TypeScript = Modern JavaScript + Types

- Open source *and* open development
- Closely track ECMAScript standard
- Innovate in type system
- Best of breed tooling
- Continuously lower barrier to entry
- Community, community, community

The assertion that types should be an integral part of a programming language is separate from the question of where the programmer must physically write down type annotations and where the can instead be inferred by the compiler.

1.3. Type Systems and Language Design
Types and Programing Languages
Benjamin C. Pierce

A well-designed statically typed language will never require huge amounts of type information to be explicitly and tediously maintained by the programmer.

1.3. Type Systems and Language Design
Types and Programing Languages
Benjamin C. Pierce

```java
ArrayList<Dog> dogs = new ArrayList<Dog>();
```

Goals

1. Must work well with unannotated/untyped code
2. Must deliver *some* value even in presence of unannotated code
3. Can’t examine the whole world to figure out the types

Gradual Types

```
function cost(items) {
  let total = 0;

  for (let item of items) {
    total += item.price;
  }

  return total;
}
```

```
function cost(items) {
  let total = 0;

  for (let item of items) {
    total += item.price;
  }

  return total;
}
```

any

```
function cost(items: any) {
  let total = 0;

  for (let item of items) {
    total += item.price;
  }

  return total;
}
```

```
declare let foo: any;

// All of these are allowed!
foo.bar;
foo.baz;
foo += foo;
foo *= foo / foo;
foo();
new foo();
```

```
function cost(items: any) {
  let total = 0;

  for (let item of items) {
    total += item.price;
  }

  return total;
}
```

```
function cost(items: any) {
  let total = 0;

  for (let item of items) {
    total += item.price;
  }

  return total;
}
```

number

```
function cost(items: any): number {
  let total: number = 0;

  for (let item of items) {
    total += item.price;
  }

  return total;
}
```

But why not look at the whole world?

1. Slow – an editor needs to provide completions *quickly*

2. Non-local inference is hard to reason about

Structural Types

```
let n: number = 0;

let s: string = "";

let b: boolean = false;
```

```
function originDistance(point) {
  return Math.sqrt(point.x ** 2 + point.y ** 2);
}
```

```
function originDistance(point) {
  return Math.sqrt(point.x ** 2 + point.y ** 2);
}
```

```
function originDistance(point) {
  return Math.sqrt(point.x ** 2 + point.y ** 2);
}

originDistance({ x: 100, y: 100 });

class Coordinate {
  x = 0;
  y = 0;
}

originDistance(new Coordinate());
```

```typescript
interface HasXY {
  x: number; y: number;
}

function originDistance(point: HasXY) {
  return Math.sqrt(point.x ** 2 + y ** 2);
}

class Coordinate {
  x = 0; y = 0;
}

originDistance(new Coordinate());
```

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

function originDistance(point: HasXY) {
  return Math.sqrt(point.x ** 2 + y ** 2);
}

class Coordinate {
  x = 0; y = 0;
}

originDistance({ x: 0, y: 0 });
```

```
class CoordinateC {
  x = 0; y = 0;
}

let p: CoordinateC;
```

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

let p: CoordinateI;
```

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

Union Types

```
function padLeft(str: string, padding: any) {
  let padChar;
  let padCount;
  if (typeof padding === "number") {
    padChar = " ";
    padCount = padding;
  }
  else {
    padCount = padding.count;
    padChar = padding.char;
  }
  return Array(padCount + 1).join(padChar) + str;
}
```

```
function padLeft(str: string, padding: any) {
  let padChar;
  let padCount;
  if (typeof padding === "number") {
    padChar = " ";
    padCount = padding;
  }
  else {
    padCount = padding.count;
    padChar = padding.char;
  }
  return Array(padCount + 1).join(padChar) + str;
}
```

```
function padLeft(str: string, padding: any) {
  let padChar;
  let padCount;
  if (typeof padding === "number") {
    padChar = " ";
    padCount = padding;
  }
  else {
    padCount = padding.count;
    padChar = padding.char;
  }
  return Array(padCount + 1).join(padChar) + str;
}
```

```
function padLeft(str: string,
                 padding: number | Options) {
  let padChar;
  let padCount;
  if (typeof padding === "number") {
    padChar = " ";
    padCount = padding;
  }
  else {
    padCount = padding.count;
    padChar = padding.char;
  }
  return Array(padCount + 1).join(padChar) + str;
}
```

Singleton types

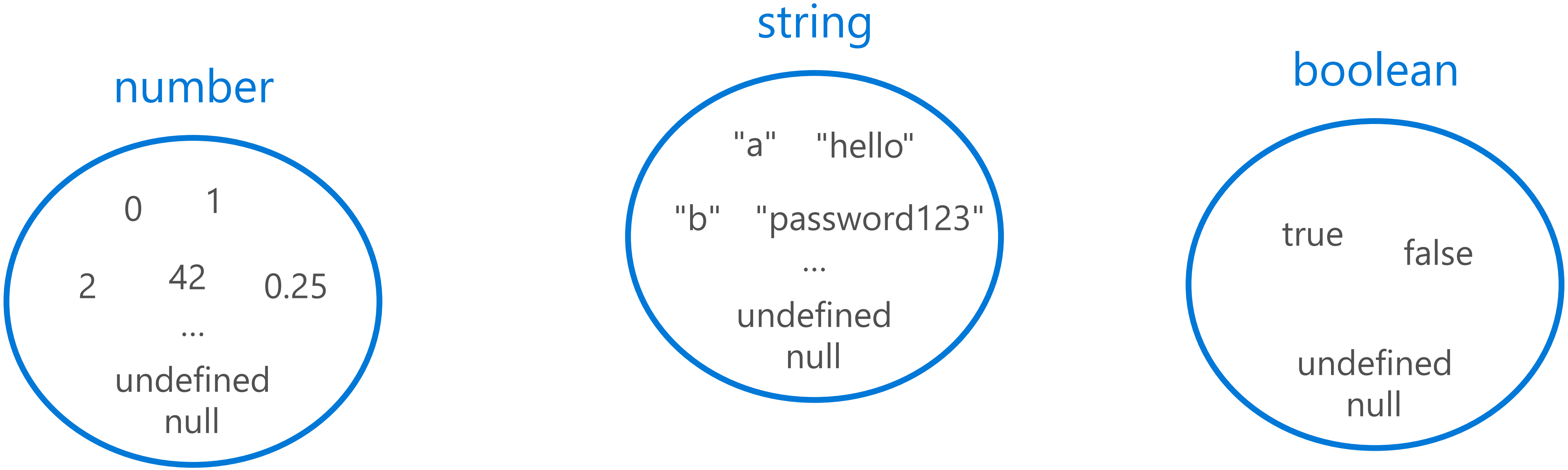
```
/**
 * @param component A component
 * @param value Must be either "left", "right" or "center"
 */
function align(component: any,
               value: string) {
    // ...
}
```

```
/**
 * @param component A component
 * @param value Must be either "left", "right" or "center"
 */
function align(component: any,
               value: "left" | "right" | "center") {
    // ...
}
```

```
/**
 * @param component A component
 * @param value Must be either "left", "right" or "center"
 */
function align(component: any,
               value: "left" | "right" | "center") {
    // ...
}

align(Foo, "centre");
```

Nullable types



number

0 1

2 42 0.25

...

undefined

null

string

"a" "hello"

"b" "password123"

...

undefined

null

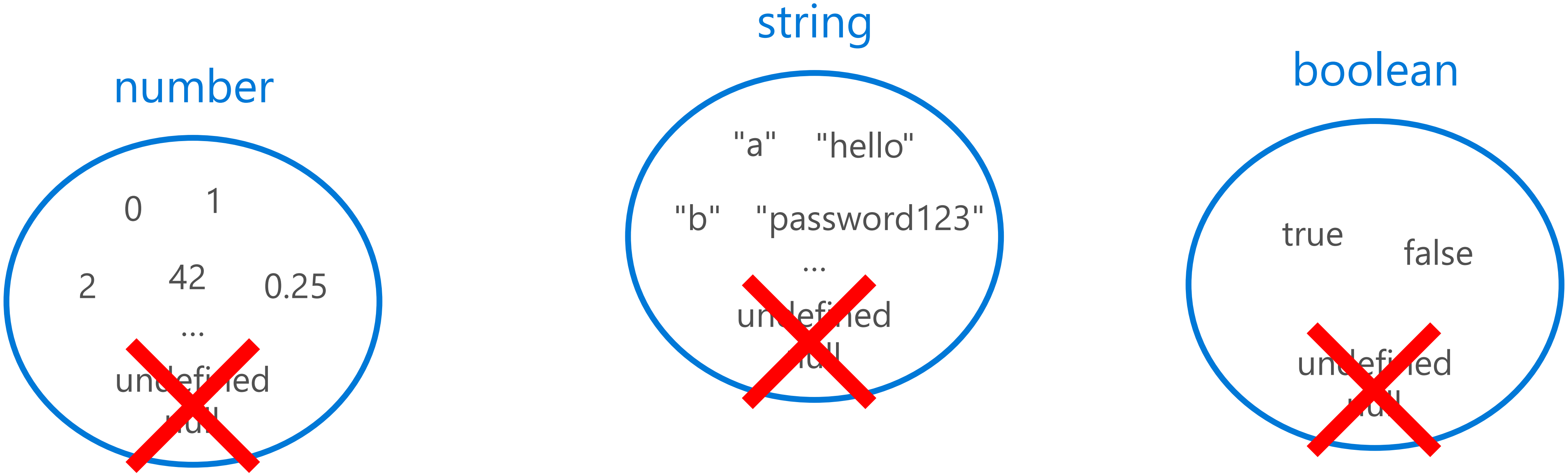
boolean

true false

undefined

null

Non-nullable types



number

0 1

2 42 0.25

…

undefined

null

string

"a" "hello"

"b" "password123"

…

undefined

null

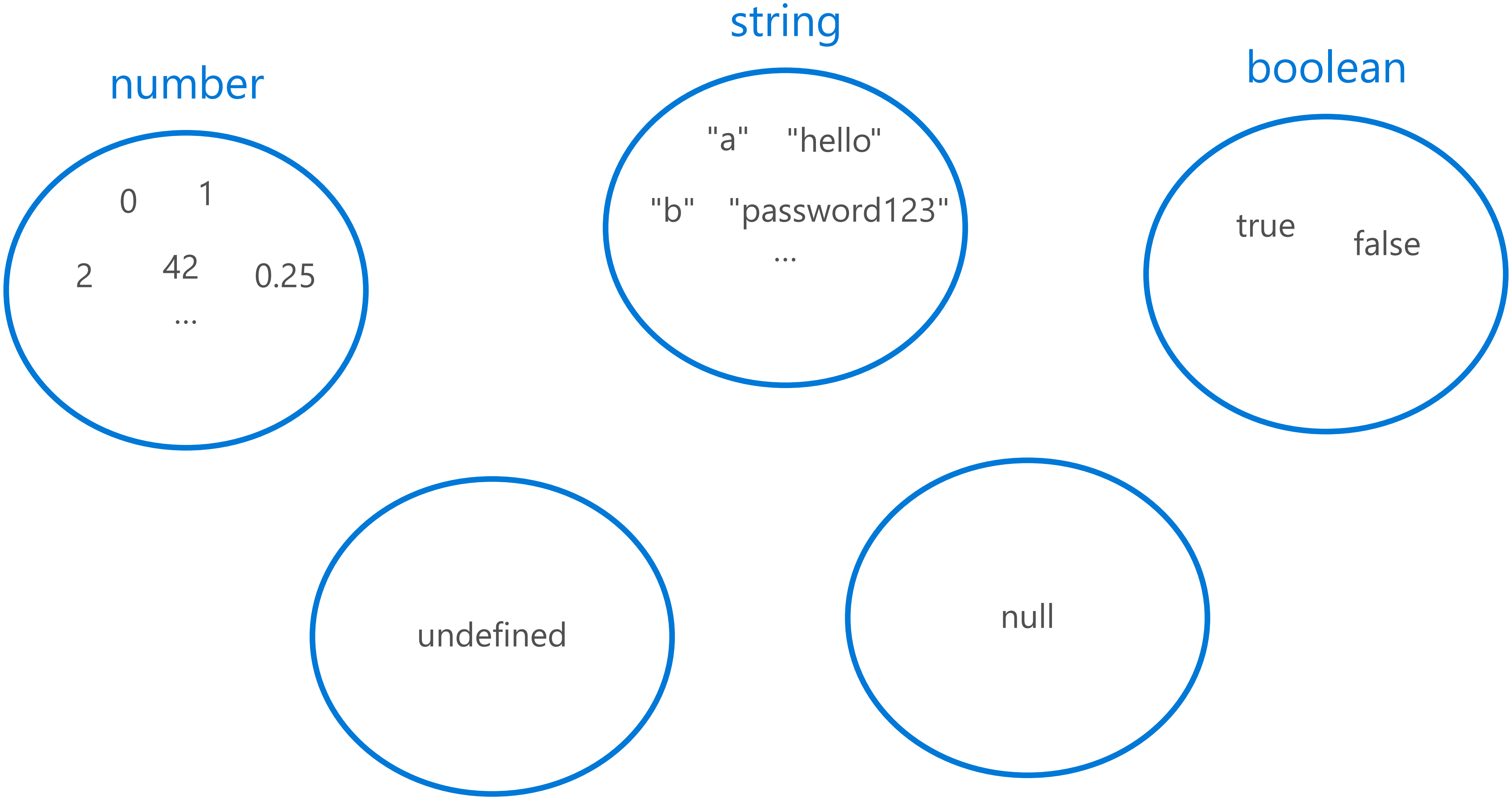
boolean

true false

undefined

null

Non-nullable types



number

0 1

2 42 0.25

...

string

"a" "hello"

"b" "password123"

...

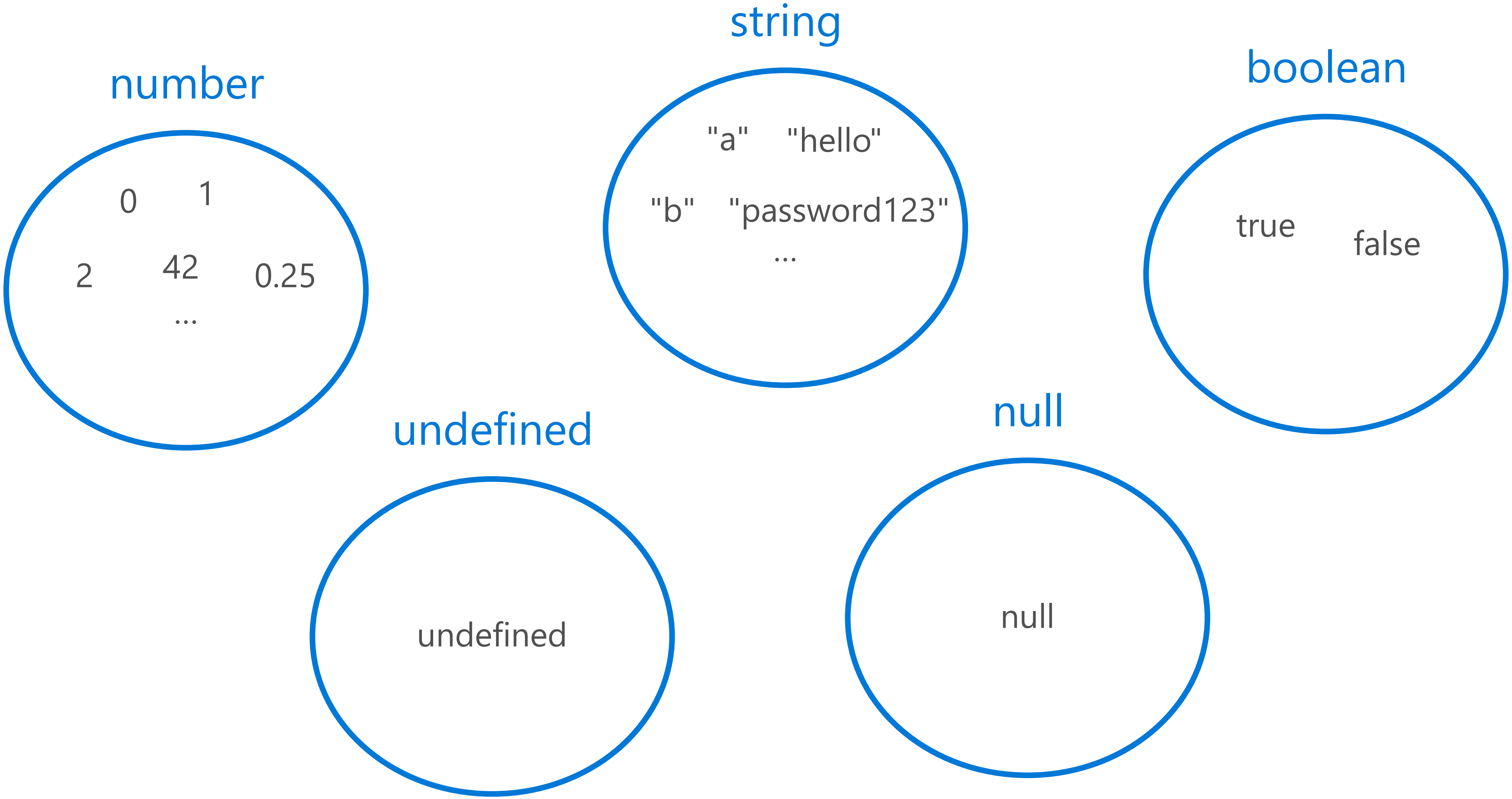
boolean

true false

undefined

null

Non-nullable types



string

"a" "hello"

"b" "password123"

...

number

0 1

2 42 0.25

...

boolean

true false

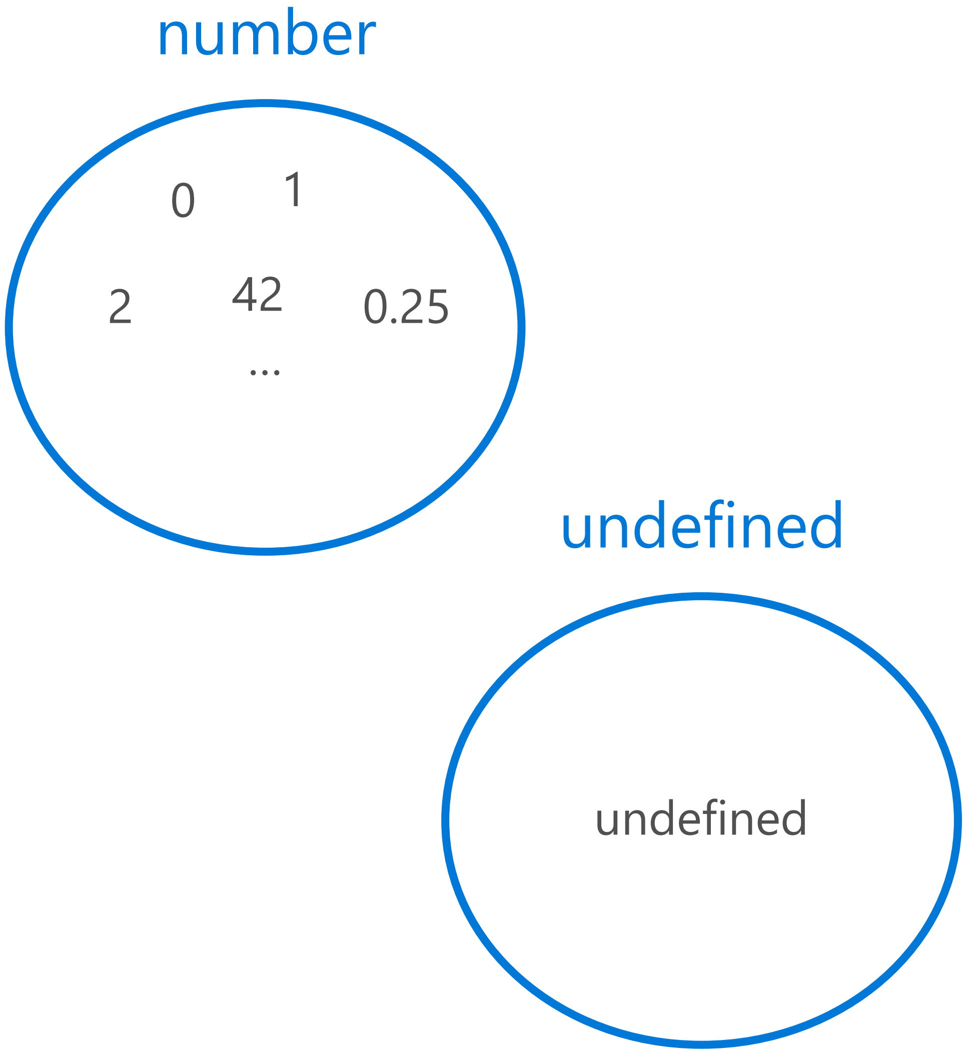
undefined

undefined

null

null

Non-nullable types



number

0 1

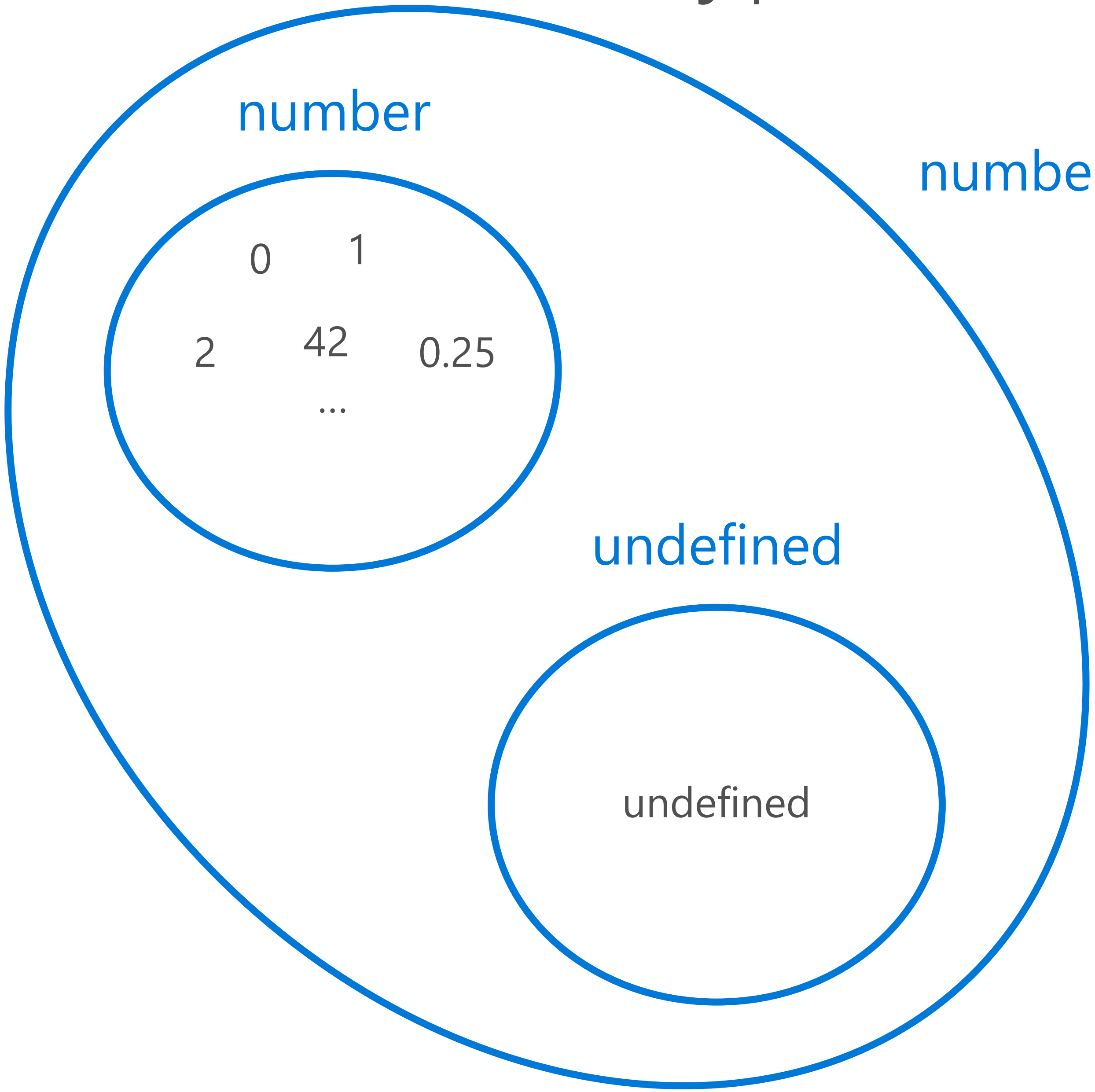
2 42 0.25

…

undefined

undefined

Non-nullable types



number | undefined

`keyof` and lookup types

```
let foo = { bar: 0 };

foo.bar = 100;

foo["bar"] = 100;
```

```
function get(obj, propName) {
 // do some stuff...
  return obj[propName];
}

function set(obj, propName, value) {
  // do some stuff...
  obj[propName] = value;
}

get(someObj, "some-property");
set(someObj, "other-property", 100);
```

Type system

Gradual, Structural, Generic

Extensive type inference

Control flow based type analysis

Novel type constructors

Object-oriented *and* functional

```
{ x: T, y: U }

T | U

T & U

keyof T

T[K]

{ [P in K]: X }

T extends U ? X : Y
```

Conditional types

```
T extends U ? X : Y
```

Higher order type equivalences

$$T \mid \mathrm{never} \;\Leftrightarrow\; T$$

$$T \;\&\; \mathrm{never} \;\Leftrightarrow\; \mathrm{never}$$

$$(A \mid B) \;\&\; (C \mid D) \;\Leftrightarrow\; A \;\&\; C \mid A \;\&\; D \mid B \;\&\; C \mid B \;\&\; D$$

$$\mathrm{keyof}\ (A \;\&\; B) \;\Leftrightarrow\; \mathrm{keyof}\ A \mid \mathrm{keyof}\ B$$

$$S[X] <: T[Y] \;\Leftrightarrow\; S <: T \;\wedge\; X :> Y$$

$$\mathrm{keyof}\ A <: \mathrm{keyof}\ B \;\Leftrightarrow\; B :> A$$

Retrofitting a type system into a language not designed for typechecking in mind can be tricky; ideally, language design should go hand-in-hand with type system design.

1.3. Type Systems and Language Design
Types and Programing Languages
Benjamin C. Pierce

Microsoft

Thank you!

Daniel Rosenwasser

@drosenwasser

http://typescriptlang.org

Starts and ends with JavaScript

Strong tools for large apps

State of the art JavaScript