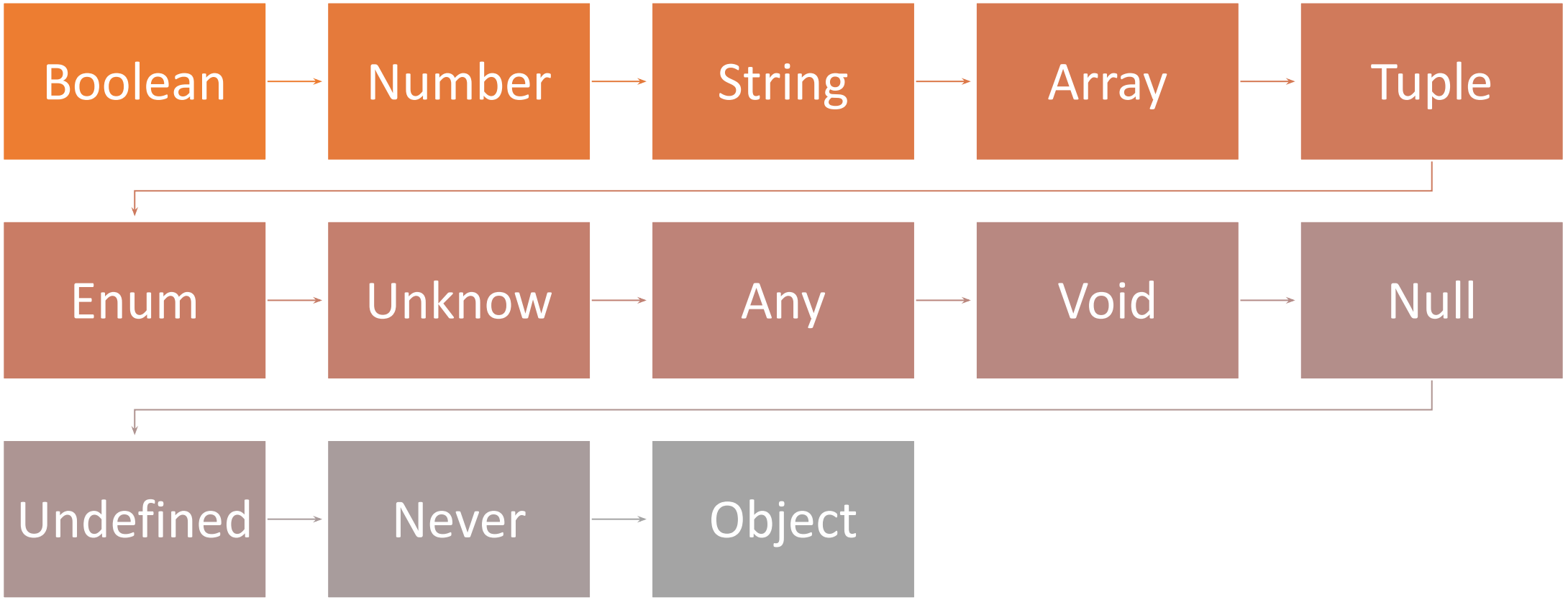
Typescript

JavaScript with syntax for types

Basic types


Boolean
Number
String
Array
Tuple
Enum
Unknow
Any
Void
Null
Undefined
Never
Object

Types

Types

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

// Exactly the same as the earlier example
function printCoord(pt: Point) {
  console.log("The coordinate's x value is " + pt.x);
  console.log("The coordinate's y value is " + pt.y);
}

printCoord({ x: 100, y: 100 });
```

- Basic type:

  Boolean, string and etc…
- You can create you own type

Types

Union type

```
const a = string | number;
```

Type Assertions

For example, if you're using `document.getElementById`, TypeScript only knows that this will return *some* kind of `HTMLElement`, but you might know that your page will always have an `HTMLCanvasElement` with a given ID.

In this situation, you can use a *type assertion* to specify a more specific type:

```
const myCanvas = document.getElementById("main_canvas") as HTMLCanvasElement;
```

```
interface Point {
  x: number,
  y: number,
  z?: number
}

const a = {
  x: 5,
  y: 2
} as Point;
```

Narrowing or type Guard

```
function padLeft(padding: number | string, input: string) {
  return " ".repeat(padding) + input;
Argument of type 'string | number' is not assignable to parameter of type 'number'.
  Type 'string' is not assignable to type 'number'.
}
```

Try

Fix:

```
function padLeft(padding: number | string, input: string) {
  if (typeof padding === "number") {
    return " ".repeat(padding) + input;
  }
  return padding + input;
}
```

Try

Interfaces

Interfaces

Interfaces declare type of object

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

function printCoord(pt: Point) {
  console.log("The coordinate's x value is " + pt.x);
  console.log("The coordinate's y value is " + pt.y);
}

printCoord({ x: 100, y: 100 });
```

Interface or type ? Different

Type aliases and interfaces are very similar, and in many cases you can choose between them freely. Almost all features of an `interface` are available in `type`, the key distinction is that a type cannot be re-opened to add new properties vs an interface which is always extendable.

Extending an interface

```
interface Animal {
  name: string
}

interface Bear extends Animal {
  honey: boolean
}

const bear = getBear()
bear.name
bear.honey
```

Extending a type via intersections

```
type Animal = {
  name: string
}

type Bear = Animal & {
  honey: boolean
}

const bear = getBear();
bear.name;
bear.honey;
```

Adding new fields to an existing interface

```
interface Window {
  title: string
}

interface Window {
  ts: TypeScriptAPI
}

const src = 'const a = "Hello World"';
window.ts.transpileModule(src, {});
```

A type cannot be changed after being created

```
type Window = {
  title: string
}

type Window = {
  ts: TypeScriptAPI
}

 // Error: Duplicate identifier 'Window'.
```

Optional = Properties?

```ts
interface Point {
  x: number,
  y: number,
  z?: number
}

const a: Point = {
  x: 5,
  y: 2
}
```

```ts
function printName(obj: { first: string; last?: string }) {
  // ...
}
// Both OK
printName({ first: "Bob" });
printName({ first: "Alice", last: "Alisson" });
```

Function

- Function Type Expressions
- Generic Functions
- Function Overloads

Function Type Expressions

```
function greeter(fn: (a: string) => void) {
  fn("Hello, World");
}

function printToConsole(s: string) {
  console.log(s);
}

greeter(printToConsole);
```

Function Generic

```ts
function firstElement<Type>(arr: Type[]): Type | undefined {
    return arr[0];
}

// s is of type 'string'
const s = firstElement(["a", "b", "c"]);
// n is of type 'number'
const n = firstElement([1, 2, 3]);
// u is of type undefined
const u = firstElement([]);
```

Function Overload

```
function len(s: string): number;
function len(arr: any[]): number;
function len(x: any) {
  return x.length;
}
```