TypeScript Fundamentals

October 27, 2017
Mike North

What's TypeScript?

- A **typed superset** of JavaScript
- Developed by **Microsoft**
- Compiles to JavaScript for either **Browsers or Node**
- Three parts: **Language**, **Language Service** and **Compiler**
- Seems to be **leading the way** for JavaScript language features

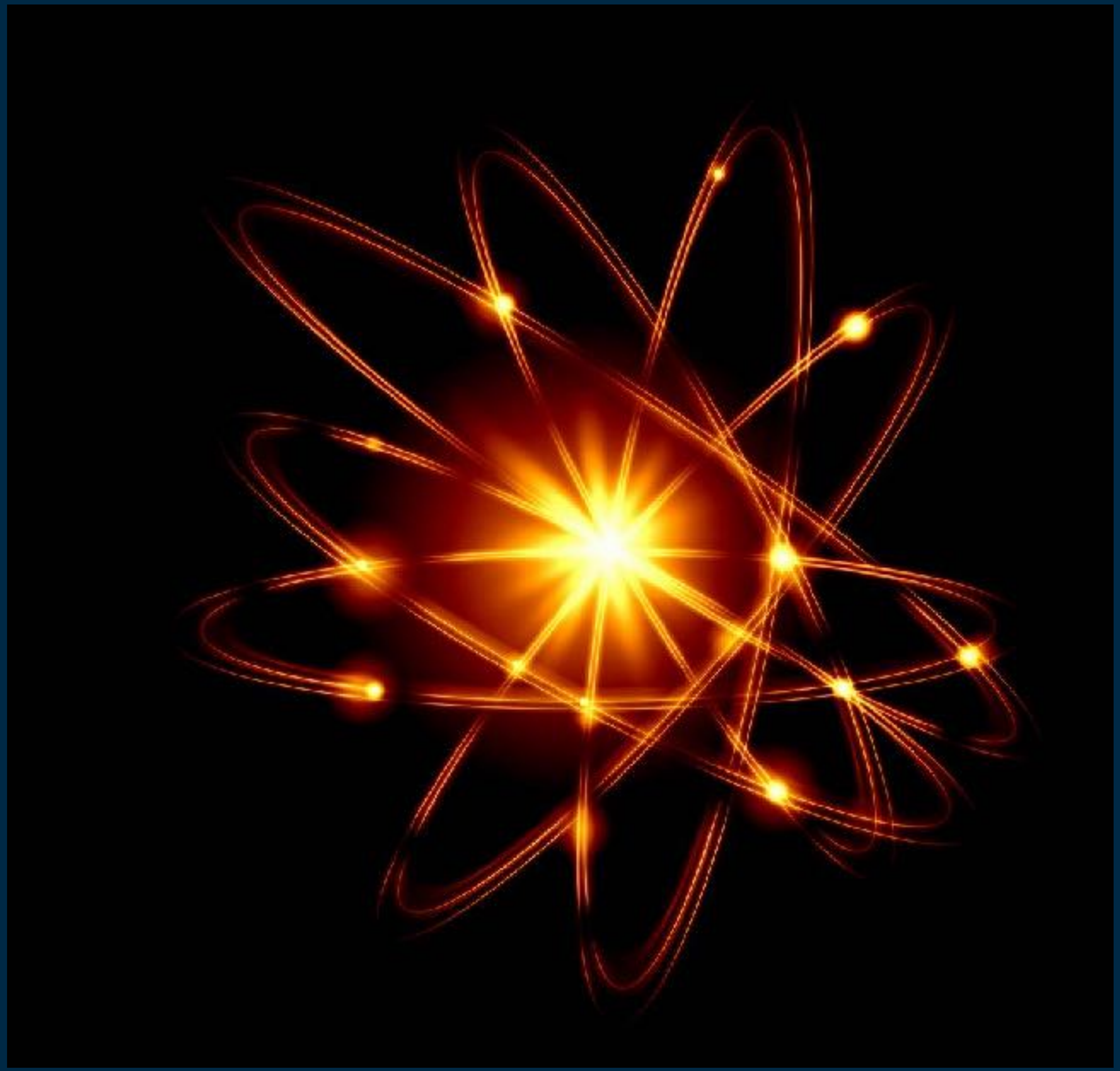
JS

Types & Operators

JavaScript - Primitive Types

JS

- Primitive values aren't objects, and have no methods*
- Six primitive types in JS:
  - `null`
  - `undefined`
  - `boolean`
  - `number`
  - `string`
  - `symbol` ES2015
- Everything else extends from: `Object`.

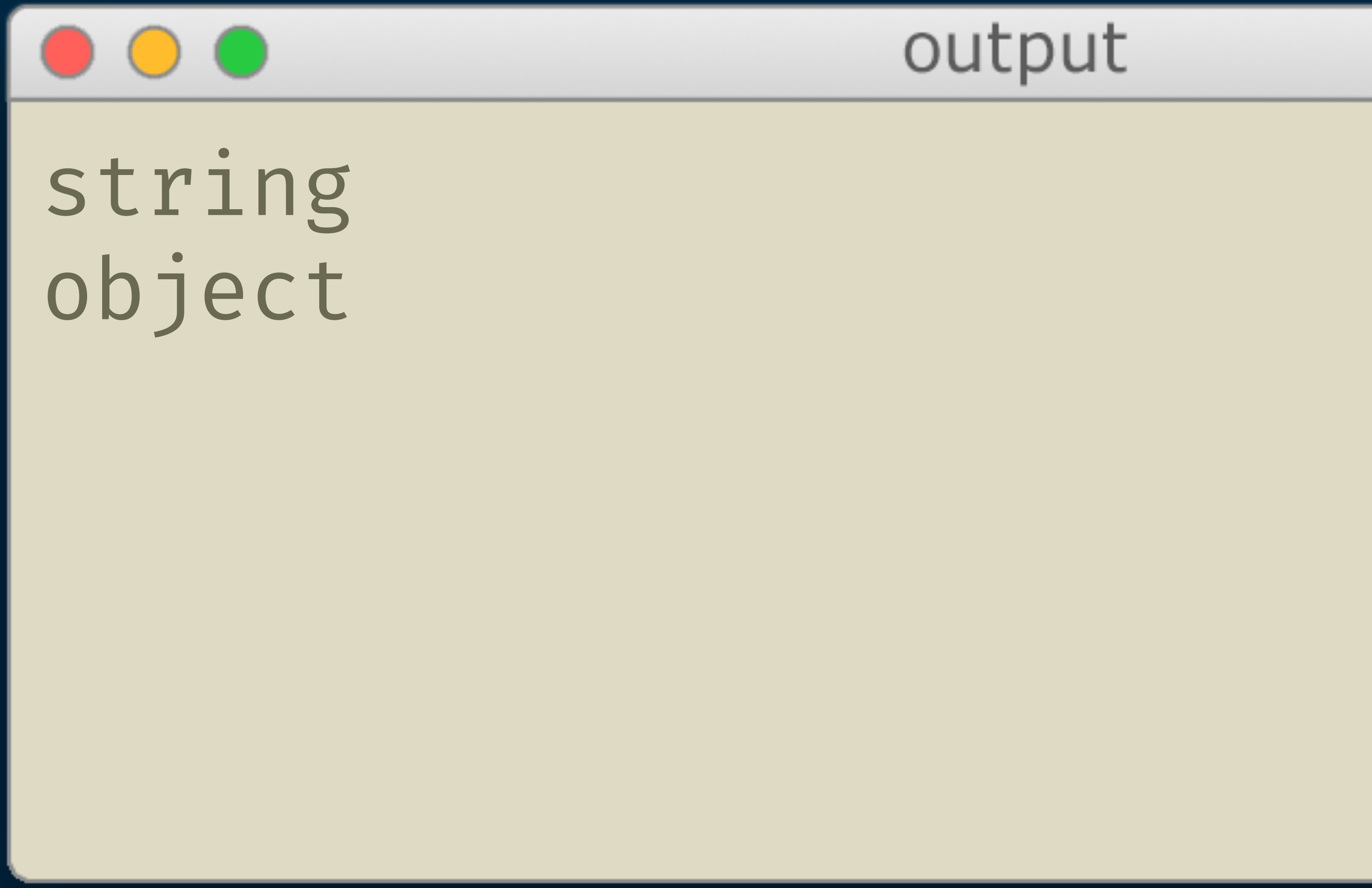


JavaScript - Auto-Boxing

JS

- When necessary, primitive types are "wrapped" by identically-named Objects, and then back to their primitive types again.

```
var x = "JavaScript";
console.log(typeof x);

var y = new String("ECMAScript");
console.log(typeof y);
```



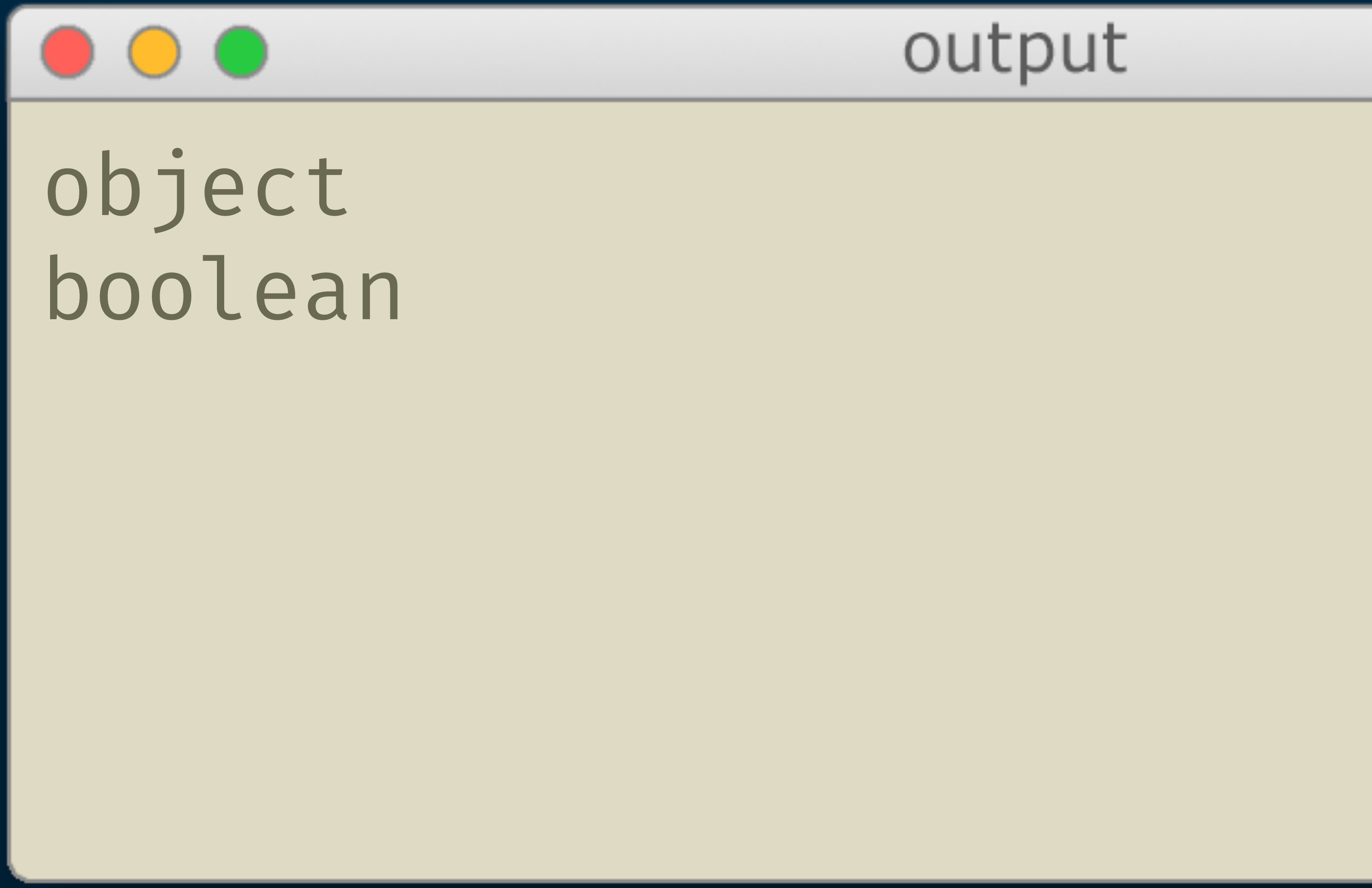
```
string
object
```

JavaScript - Auto-Boxing

JS

- When necessary, primitive types are "wrapped" by identically-named Objects, and then back to their primitive types again.

```
var b = new Boolean(false);
console.log(typeof b);
console.log(typeof true);
```



```
object
boolean
```

JavaScript - Auto-Boxing

JS

- When necessary, primitive types are "wrapped" by identically-named Objects, and then back to their primitive types again.
- Most of the time we don't care about this, except that...
- Primitive types are `immutable`.
- Direct use of the **boxed** types (i.e. `new String('wrong');`) is almost always a mistake

JavaScript - Variable declarations & scope

JS

- We can use three kinds of variable declarations

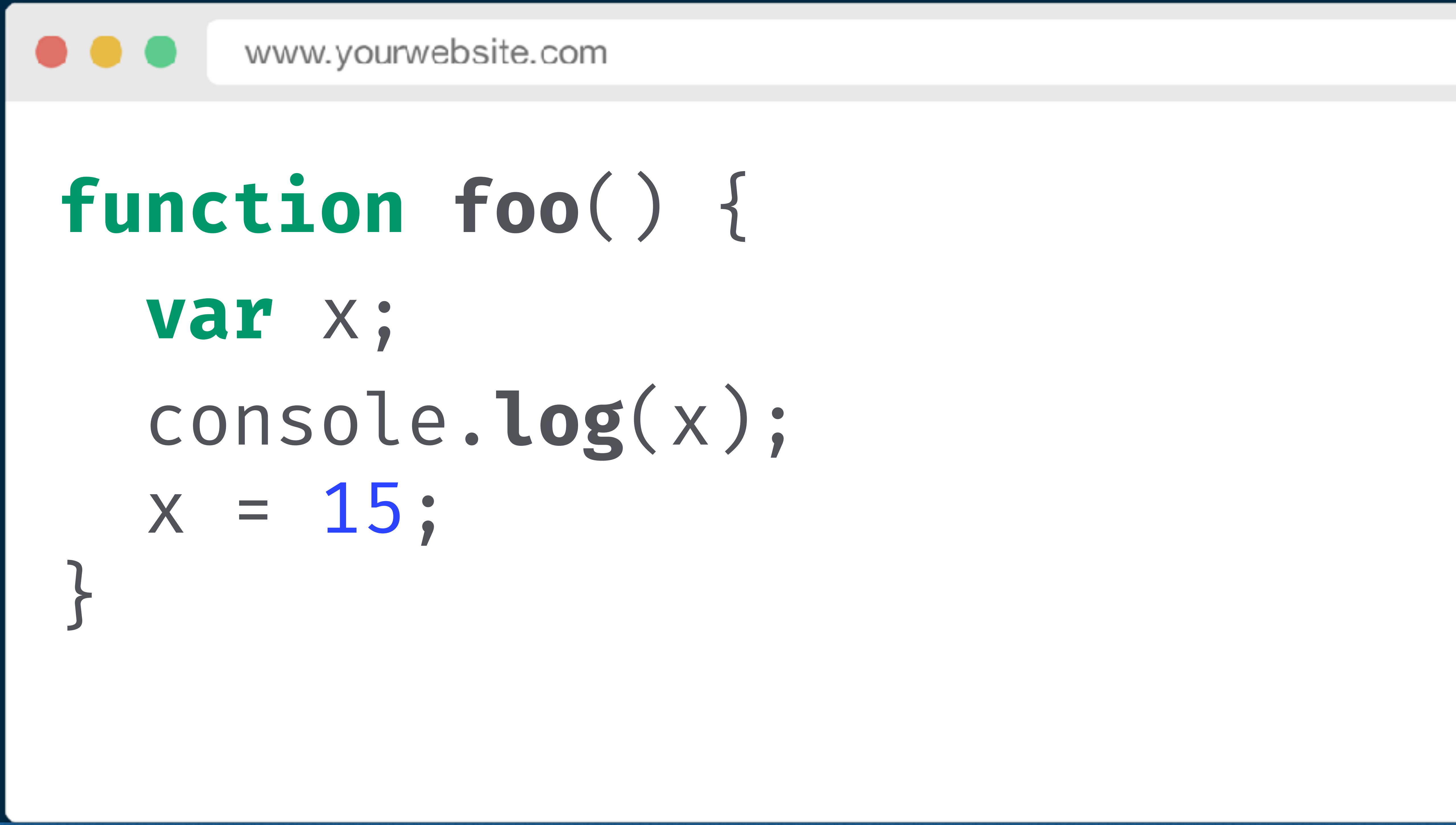
```
var x = 14;

let y = 'abc';

const z = 'JavaScript';
```

JavaScript - var

JS

- var declarations **ARE** hoisted - it's as if they're all declared at the top of the global or function scope in which they're defined

```
function foo() {

  console.log(x);
  var x = 15;
}
```



```
function foo() {
  var x;
  console.log(x);
  x = 15;
}
```

JavaScript - var

JS

- `var` declarations **ARE NOT** block-scoped
- "belong" to the function or global scope they're defined in

```
var x = 15;

if (x > 10) {
  var y = 21;
}

console.log(x + y); // 36
```

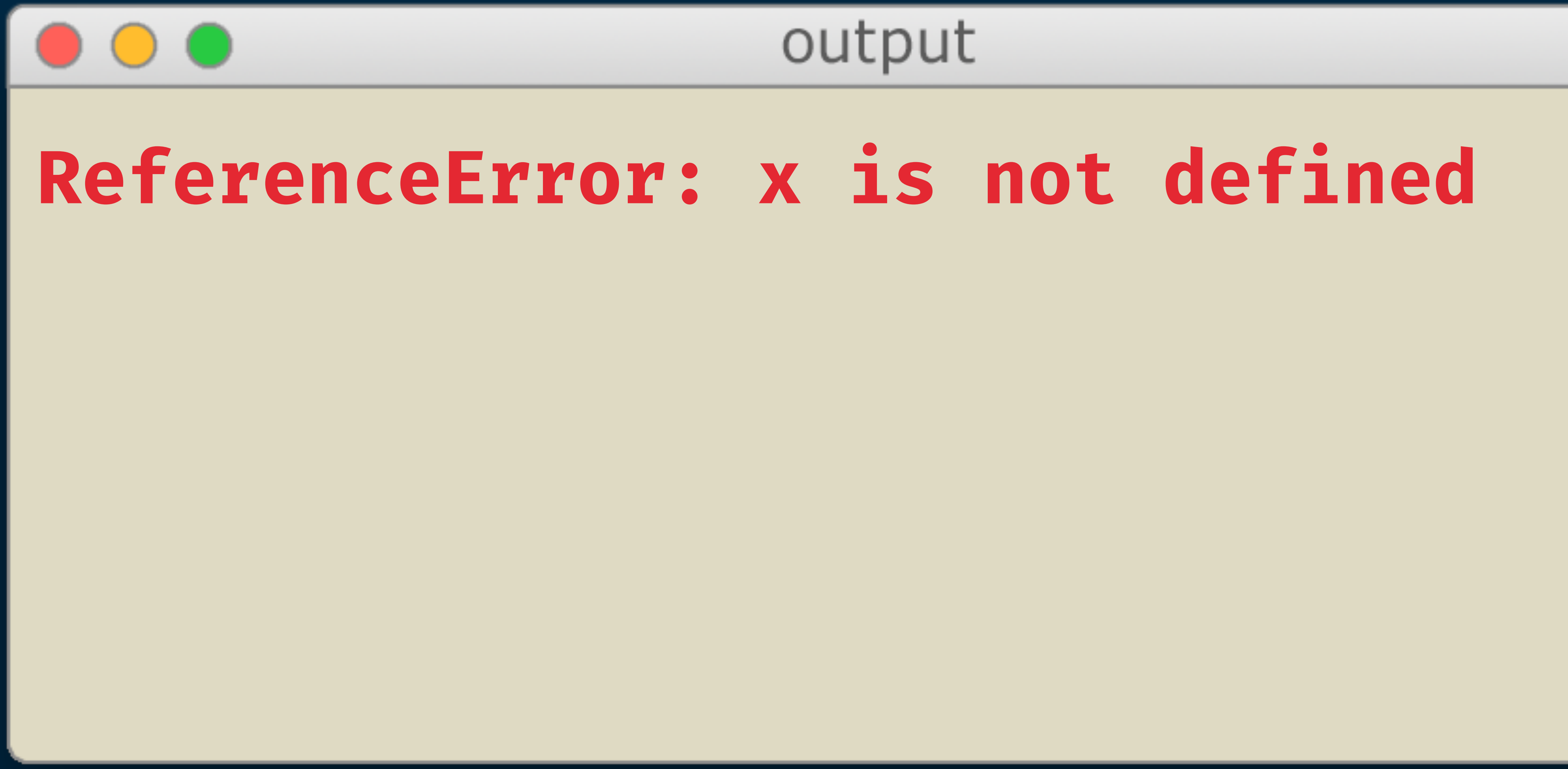
JavaScript - let

JS

- let declarations **ARE NOT** hoisted
- Polyfills & transpilers check for, and enforce this at build time

```
function foo() {

  console.log(x);
  let x = 15;
}
foo();
```

"Temporal Dead Zone"



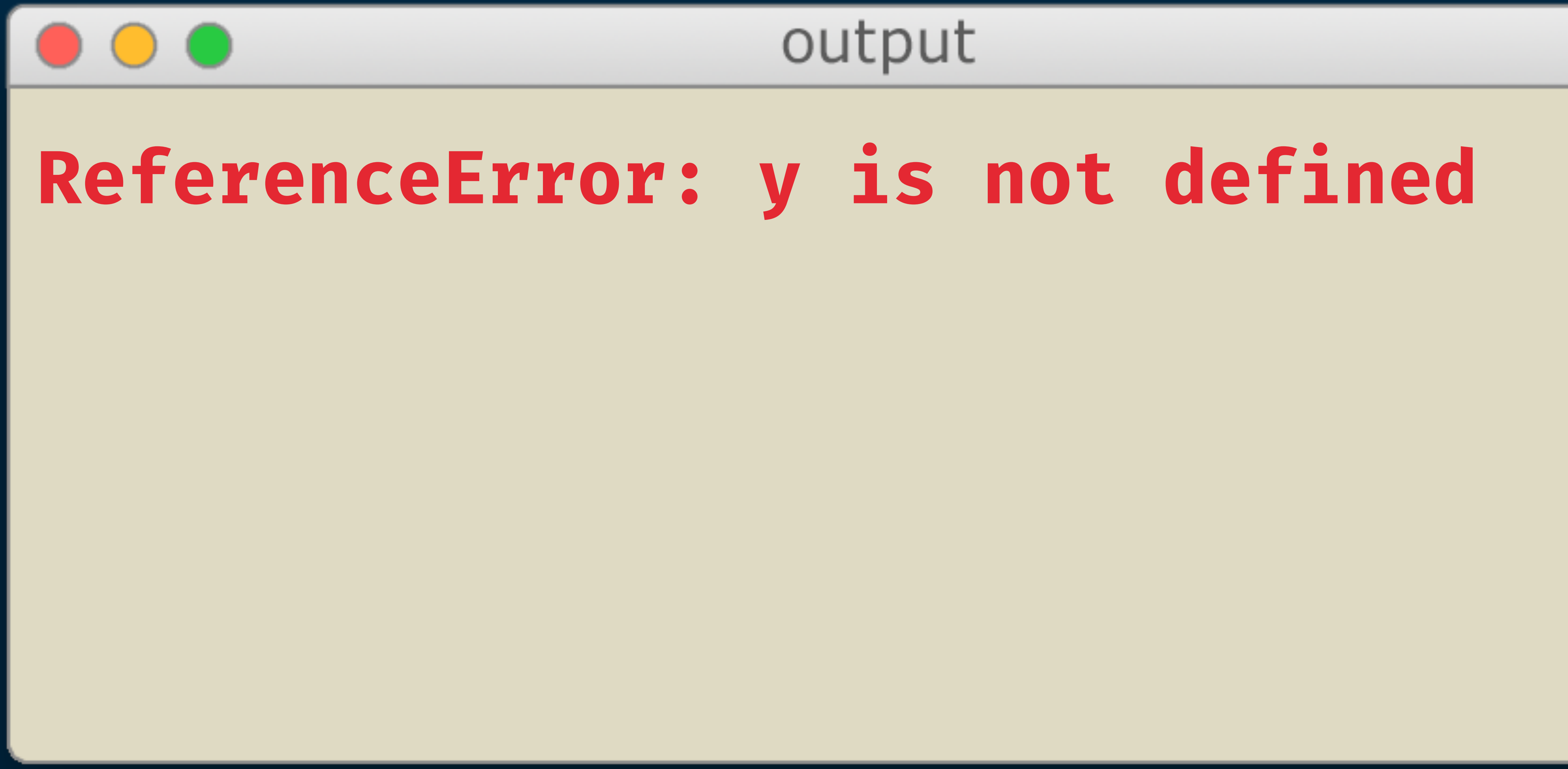
```
ReferenceError: x is not defined
```

JavaScript - let

JS

- let declarations **ARE** block-scoped
- "belong" to the block scope they're defined in

```
let x = 15;
if (x > 10) {
  let y = 21;
}
console.log(x + y);
```



output

```
ReferenceError: y is not defined
```

JavaScript - const

JS

- `const` declarations **ARE NOT** hoisted
- Must be initialized at the time of declaration
- Re-assignment is not allowed
- Constant variable does NOT mean "immutable value"
- Mutable values should be used with `Object.freeze` to get immutability

Declarations, in summary

JS

	var	let	const
Reassign	✅	✅	
Scope	function	block	block
hoisted	✅😩		

Type Conversion

JS

- As with many dynamically-typed languages, things are converted "as needed"

```
30 + 7;       // 37
'37' + 7;     // '377'
'37' - 7;     // 30
(+ '37');     // 37
(+ false);    // 0
```

Type Conversion

JS

- As with many dynamically-typed languages, things are converted "as needed"
- The + operator, when used with strings, converts all other operands to strings

```
30 + 7;      // 37
'37' + 7;    // '377'
'37' - 7;    // 30
```

- the unary + operator converts the operand to a Number

```
(+ '37');    // 37
(+ false);   // 0
```

😕

Seems Confusing

TS

In this class, we'll learn …

- Where the line between Modern JS and TypeScript is TS JS
- How TypeScript provides React components with much-needed structure for
- To use the power of types to make our code more expressive of our intent
- Strategies for applying constraints with a light touch
- A practical strategy for incremental adoption

Why Add Types?

- Sometimes JavaScript does unintuitive things to convert primitive types
- Move some common errors from **runtime** to **compile time**
- Great **documentation** for fellow developers
- Those clever abstractions you're so excited about are now safer to use
- Modern JavaScript runtimes are written in typed languages

Implicit Typing

TS

- The TypeScript compiler can make good guesses at types, just through assignment

```
let teacherAge = 34;
teacherAge = '35';
  // ↪ 🛑 Type 'string' is not assignable to type 'number'.
```

- After assigning a value to a variable, you're not allowed to change the type
  - JavaScript lets you do this, but it's a common cause of de-optimization in modern runtimes

Explicit Typing: Annotations

- Rather than let TypeScript make a guess, we can provide a type at variable declaration

```
let teacherAge: number = 34;
```

- This type information is known as a **type annotation**, and can be used anywhere a variable is declared (and other places)

Explicit Typing: Casting

TS

- Sometimes we need to "cast" a value to a particular type with the **as** keyword

```
let input =
  document.querySelector('input#name_field') as HTMLInputElement;
```

- There's another way to do this, but it doesn't mix well with JSX

```
let input =
  <HTMLInputElement>document.querySelector('input#name_field');
```

Explicit Typing: Function Parameters & Return

- Functions can provide type annotations for argument and return types

```
function login(username: string, password: string): User {
  /* do something */
}
```

Argument types

Return type

- Or, if you prefer arrow functions

```
const login = (username: string, password: string): User => {
  /* do something */
}
```

TS

Type Systems & Type Equivalence

```
function validateInputField(input: HTMLInputElement) {
  /*  ...  */
}

validateInputField(x);
```

Can we regard x as an HTMLInputElement?

- **Nominal Type Systems** answer this question based on whether x is an instance of a class/type **named** HTMLInputElement
- **Structural Type Systems** only care about the **shape** of an object. This is how typescript works!

Object Shapes

JS

- When we talk about the **shape** of an object, we're referring to the names of properties and types of their values

Car

Make	String
Model	String
Year	Number

Object Shapes

- We can use this **shape** the same way we've been using primitive types like `string`, in a type annotation

```
let myCar: { make: string, model: string, year: number };

myCar = {
  make: 'Honda',
  model: 'Accord',
  year: 1992
};
```

Object Shapes

TS

- The **shape** can be thought of as a requirement of structure.
- If properties are missing, or of the wrong type, the compiler will tell you

```
function washCar(car: { make: string, model: string, year: number })

let myCar = {
  make: 'Honda',
  model: 'Accord'
};

washCar(myCar);
```

Object Shapes

- Excess properties are fine, as long as the the type is **at least** the right **shape**

```
function washCar(car: { make: string, model: string, year: number })

let myCar = {
  make: 'Honda',
  model: 'Accord',
  year: 1992,
  color: {r: 255, g: 0, b: 0}
};

washCar(myCar);
```

Warm Up: Color Functions

- In your `./exercises/color-functions/color-utils.ts` file, create two functions, `rgbToHex` - and `hexToRgb`.
- **rgbToHex** should take three 8-bit decimal (0-255) color channels, corresponding to red, green and blue, and return the corresponding hex string
- **hexToRgb** should take a 3 or 6-digit hex string, and return an object with properties r, g, and b having the equivalent 8-bit decimal color values.
- Both of these functions should be named exports from the color-utils.js module

```
rgbToHex(255, 0, 0); // "ff0000"
hexToRgb('00ff00'); // {r: 0, g: 255, b: 0}
```

```
npm test color-functions
```

Warm Up: Color Functions

1

- Coersing (converting types, keeping content similar) string to an integer

```
parseInt("124", 10); === 124;
```

- Converting an integer into its hexidecimal representation (string)

```
parseInt(124, 10).toString(16) === '7c'
```

- Converting a hexadecimal number (string) into an integer

```
parseInt('7c', 16) === 124;
```

```
npm test color-functions
```

Object Shape: Excess Property Checking

TS

- Although, when working with object literals, **shape** is checked more strictly. Excess properties in this situation are regarded as a possible bug

```
let myCar: { make: string, model: string, year: number };

myCar = {
  make: 'Honda',
  model: 'Accord',
  year: 1992,
  color: {r: 255, g: 0, b: 0}
}
```

```
Type '{ make: string; model: string;
year: number; color: { r: number; g:
number; b: number; }; }' is not
assignable to type '{ make: string;
model: string; year: number; }'.
```

```
Object literal may only specify known properties, and 'color' does
not exist in type '{ make: string; model: string; year: number; }'.
```

Object Shape: Excess Property Checking

TS

- Easy way to deal with this: explicitly cast the type of the object to the appropriate type

```
let myCar: { make: string, model: string, year: number };

myCar = {
  make: 'Honda',
  model: 'Accord',
  year: 1992,
  color: {r: 255, g: 0, b: 0}
} as { make: string, model: string, year: number };
```

Object Shapes

TS

- This is going to get repetitive very quickly
- What if we want to alter the shape of this type?

```
let myCar: { make: string, model: string, year: number } = {
  make: 'Honda',
  model: 'Accord',
  year: 1992
};

let lisasCar: { make: string, model: string, year: number } = {
  make: 'Ford',
  model: 'Monster Truck',
  year: 2016
};

function carCageMatch(
  a: { make: string, model: string, year: number },
  b: { make: string, model: string, year: number }
) { ... }
```

Object Shapes: Interfaces

TS

- Interfaces allow us to define a structure and **refer to it by name**

```
interface Car {
  make: string;
  model: string;
  year: number;
};

let myCar: Car =     { make: 'Honda', model: 'Accord',        year: 1992};
let lisasCar: Car = { make: 'Ford',  model: 'Monster Truck', year: 2016};

function carCageMatch(car1: Car, car2: Car) {
   ...
}
```

Object Shapes: Interfaces

- Interfaces only describe structure, they have no implementation
- They don't compile to any JavaScript code.
- DRY type definition allows for easy refactoring later
- Interfaces are "open" and can be extended later on!

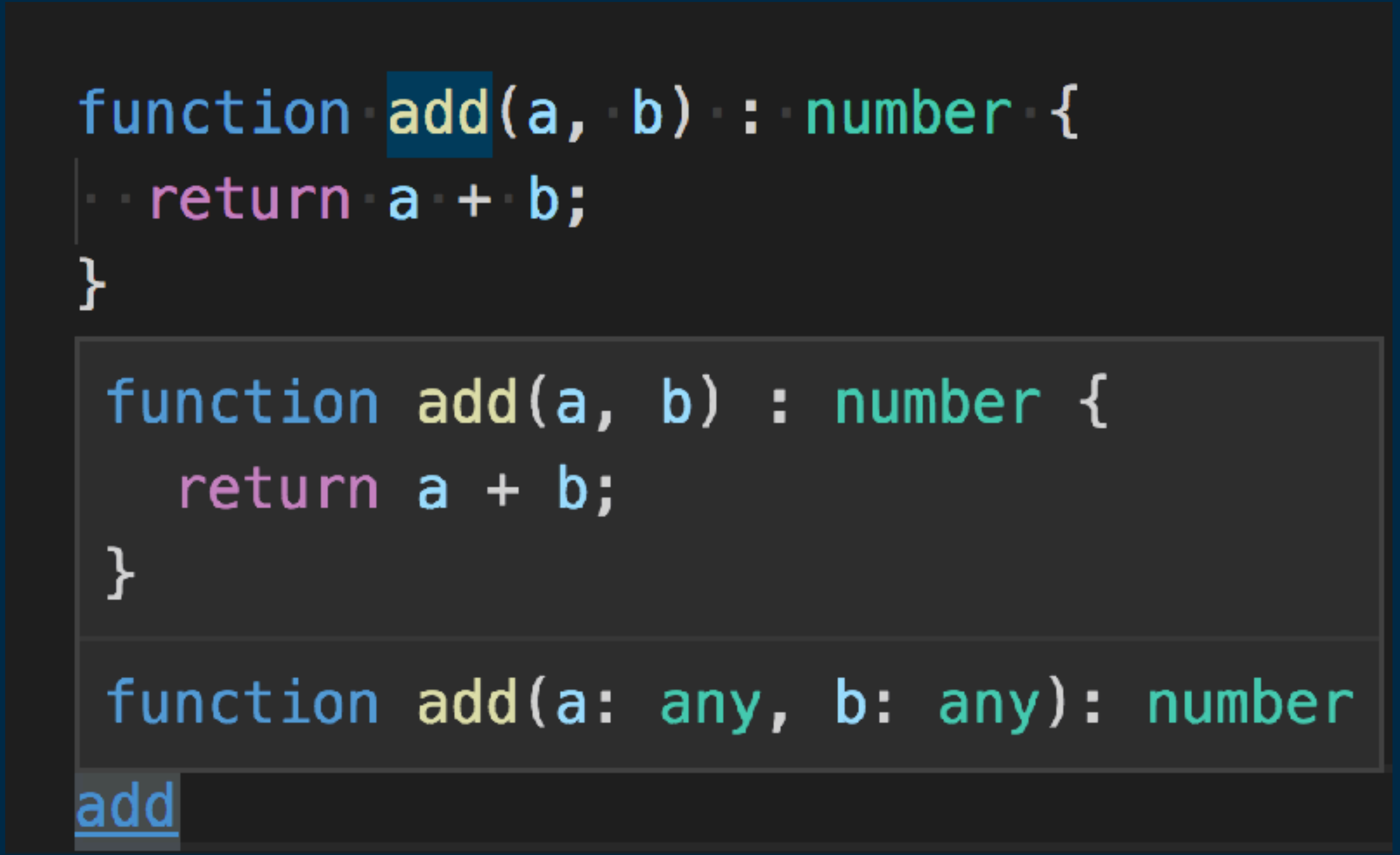
```
interface Car {
  make: string;
  model: string;
  year: number;
};

interface Car {
  color: string
}

let lisasCar: Car = {
  make: 'Ford',
  model: 'Monster Truck',
  year: 2016,
  color: "#fff" // ✅
};
```

TS

The any type

- Allows for a value of any kind
- How every mutable JS value is treated
- Useful as you migrate code from JS to TS
- Start with making all anys explicit, and then squash as many as possible.
- There's also a `never` type, which is compatible with NOTHING.

```
let age = 34;
let myAge = age as any;
myAge = '35';
```



```
function add(a, b) : number {
  return a + b;
}
```

```
function add(a, b) : number {
  return a + b;
}
```

```
function add(a: any, b: any): number
```

add

Account Manager

- In this exercise, we have two types of accounts: **`user`** and **`admin`**
- Design an interface for each, given that users have **`email`**, **`password`** and **`isActive`** properties, and admins additionally have an **`adminSince`** property, which is of type **`Date`**.
  - Export these interfaces from the **`account-manager.ts`** module as **`IUser`** and **`IAdmin`**
- Update the **`AccountManager`** class you've been given, such that any type mismatching is caught by the TypeScript compiler at build time

```
npm test accounts
```

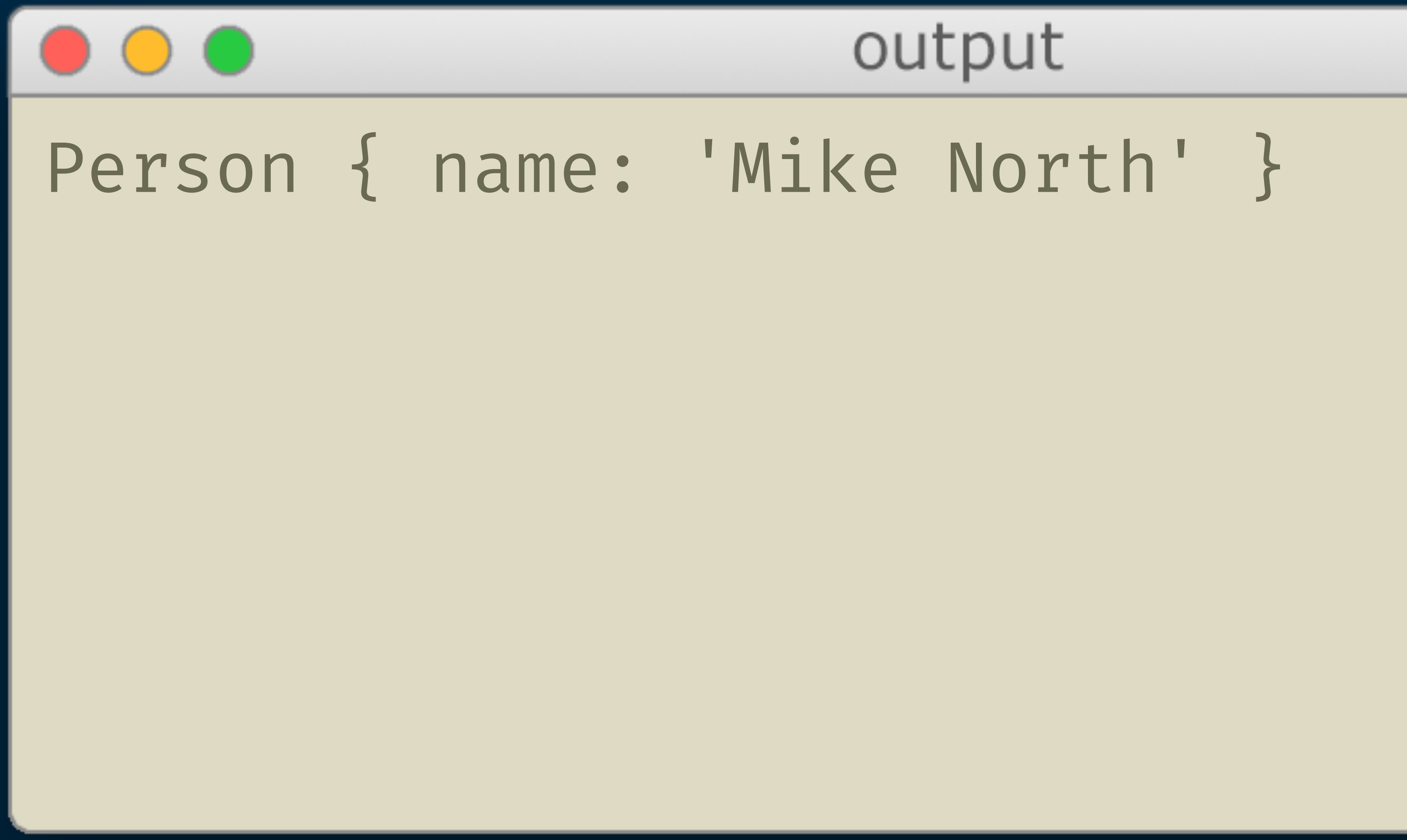
📦 Classes

Classes - Defining & Creating Instances

JS

- STILL prototypal inheritance, just a better syntax
- special `constructor` function to initialize instances.

```
class Person {
  constructor(name) {
    this.name = name;
  }
}

let mike = new Person('Mike North');
console.log(mike);
```



```
Person { name: 'Mike North' }
```

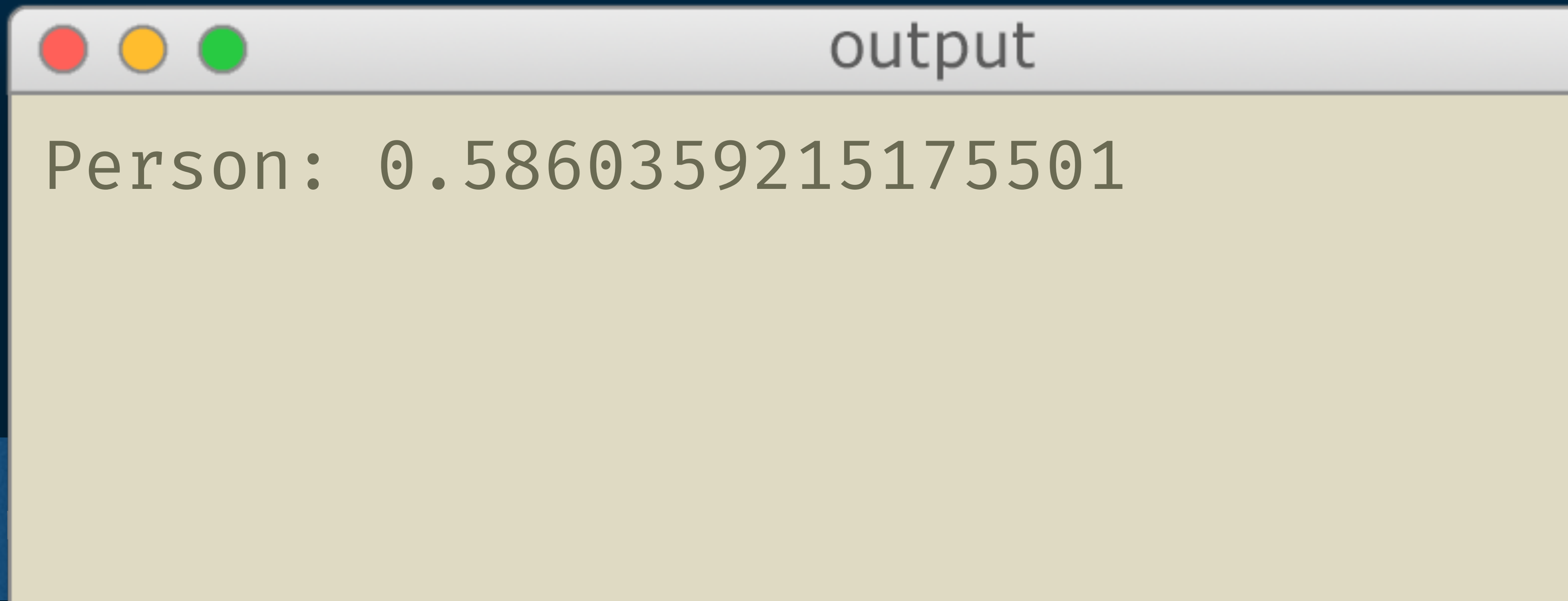
Classes - Methods

JS

- Methods can be defined in a similar way as on objects
- Static methods can be defined using the `static` keyword.

```
class Person {
  constructor(name) {
    this.name = name;
  }
  toString() {
    return `Person: ${this.name}`;
  }
  static createRandom() {
    return new Person(`${Math.random()}`);
  }
}
```

```
let stranger = Person.createRandom()
console.log(stranger.toString());
```



```
Person: 0.5860359215175501
```

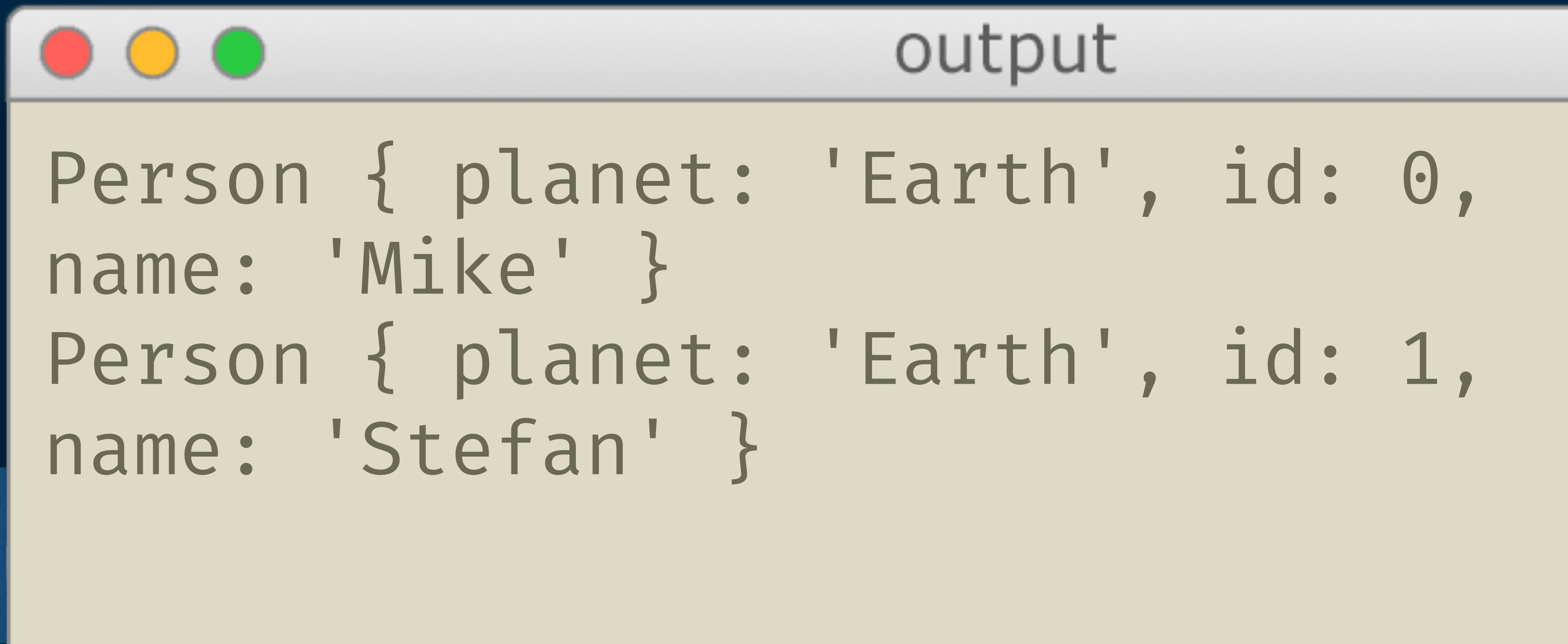
Classes - Public & Instance Fields

ES2018: STAGE 2 JS

- Instance fields - equivalent to putting a property on an instance (in a constructor)
- Public (static) fields do not require an instance - equivalent to putting a property on a constructor

```
class Person {
  static _counter = 0
  planet = 'Earth'
  constructor(name) {
    this.id = Person._counter++;
    this.name = name;
  }
}
```

```
let mike = new Person('Mike');
let stef = new Person('Stefan');

console.log(mike, stef);
```



output

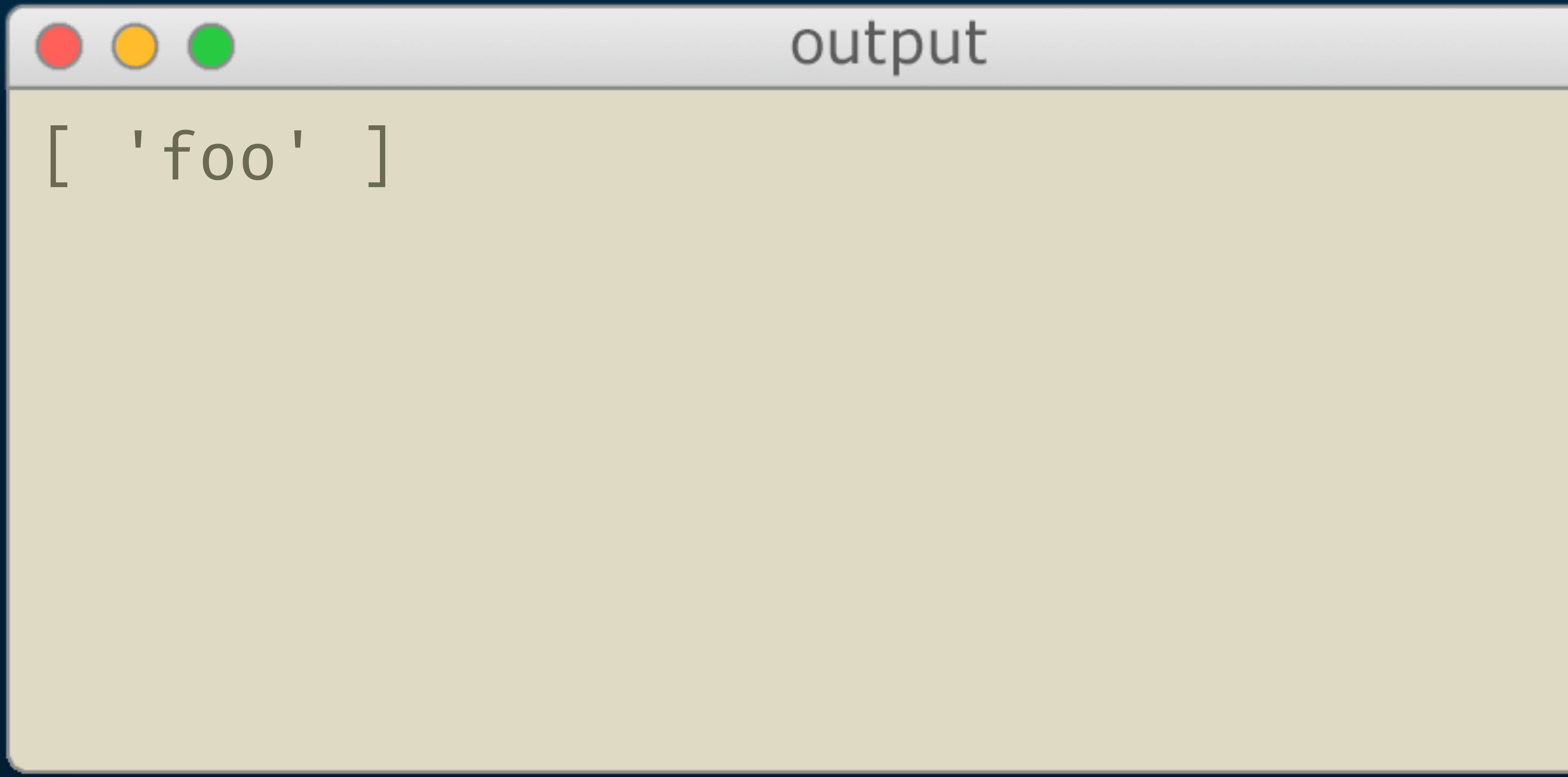
```
Person { planet: 'Earth', id: 0,
name: 'Mike' }
Person { planet: 'Earth', id: 1,
name: 'Stefan' }
```

Classes - Instance vs Prototype Fields

ES2018: STAGE 2

JS

```
function Person() {};
Person.prototype = {
  tags: []
}
var p1 = new Person();
var p2 = new Person();
p1.tags.push('foo');
console.log(p2.tags);
```



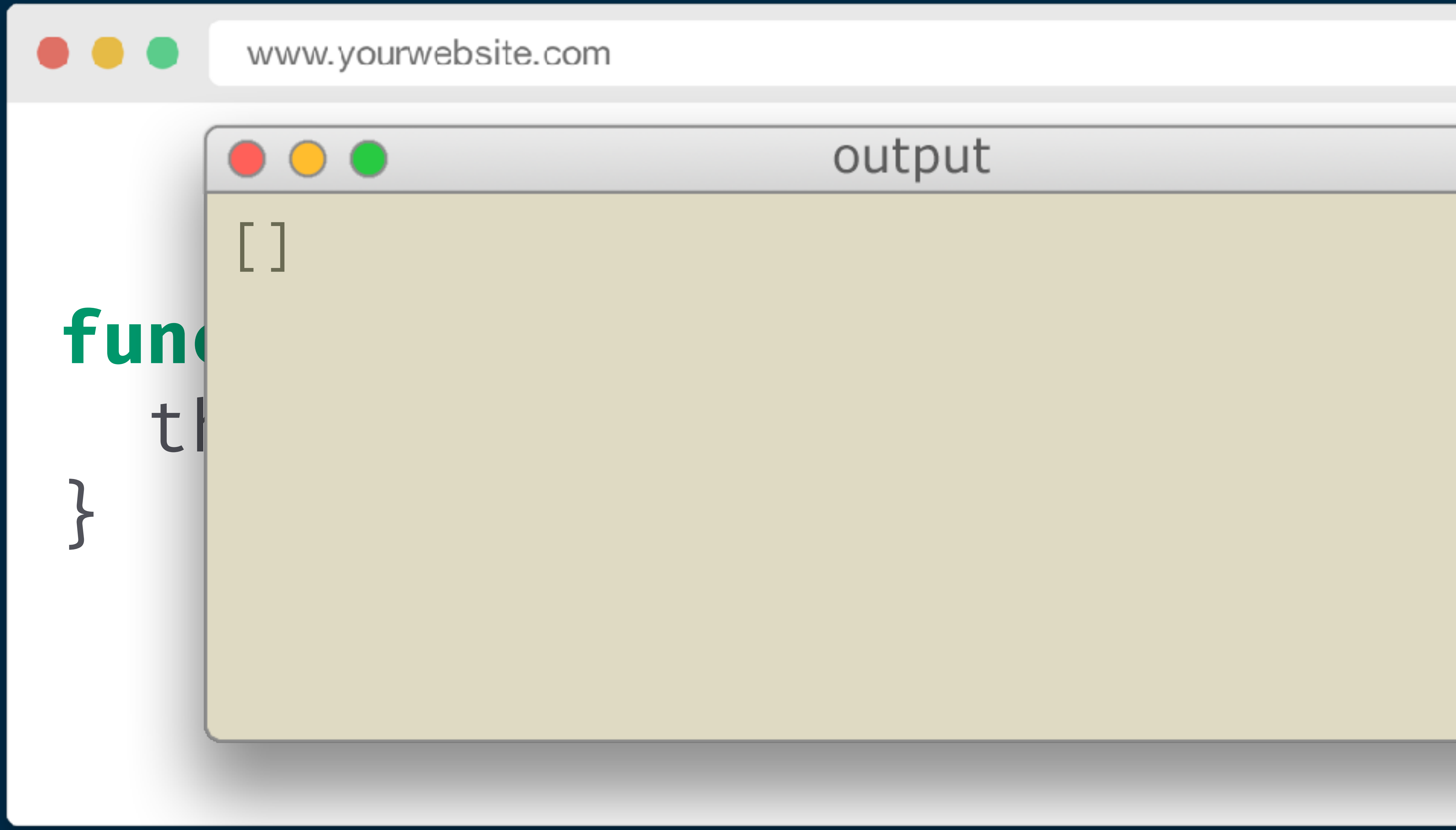
```
[ 'foo' ]
```

Classes - Instance vs Prototype Fields

JS

ES2018: STAGE 2

```
class Person {
  tags = []
}

var p1 = new Person();
var p2 = new Person();
p1.tags.push('foo');
console.log(p2.tags);
```



```
[]
```

```
func
  th
}
```

JS

Inheritance

- Subclasses can be created by using the `extends` keyword.
- The `super` keyword can be used to call methods on the parent class

```
class Person {
  constructor(name) {
    this.name = name;
  }
  toJSON() {
    return {
      name: this.name
    };
  }
}
```

```
class Employee extends Person {
  constructor(id, name) {
    super(name);
    this._employeeId = id
  }
  toJSON() {
    return {
      ...super.toJSON(),
      id: this._employeeId
    };
  }
}
```

parent constructor

parent prototype method

```
let me = new Employee(123, 'Mike');
console.log(me.toJSON());
```

output

```
{ name: 'Mike', id: 123 }
```

Classes - Species

JS

- There's a special property on classes called `Symbol.species` that's used when building "derived objects"
- EXAMPLE: An array that doesn't print any private info via `console.log`, but `map` and `filter` return regular arrays that don't have this restriction.

```
class MyArray extends Array {
  toString() {
    return '[PRIVATE]';
  }
}
```

```
let a = new MyArray(1, 2, 3);
  console.log(`${a}`);
let filtered = a.filter((y) => y <= 2);
  console.log(`${filtered}`);
```

output

```
[PRIVATE]
[PRIVATE]
```

Classes - Species

JS

- There's a special property on classes called Symbol.species that's used when building "derived objects"
- EXAMPLE: An array that doesn't print any private info via `console.log`, but `map` and `filter` return regular arrays that don't have this restriction.

```
class MyArray extends Array {
  toString() {
    return '[PRIVATE]';
  }
  static get [Symbol.species]() {
    return Array;
  }
}
```

```
let a = new MyArray(1, 2, 3);
  console.log(`${a}`);
let filtered = a.filter((y) => y <= 2);
  console.log(`${filtered}`);
```

output

```
[PRIVATE]
1,2
```

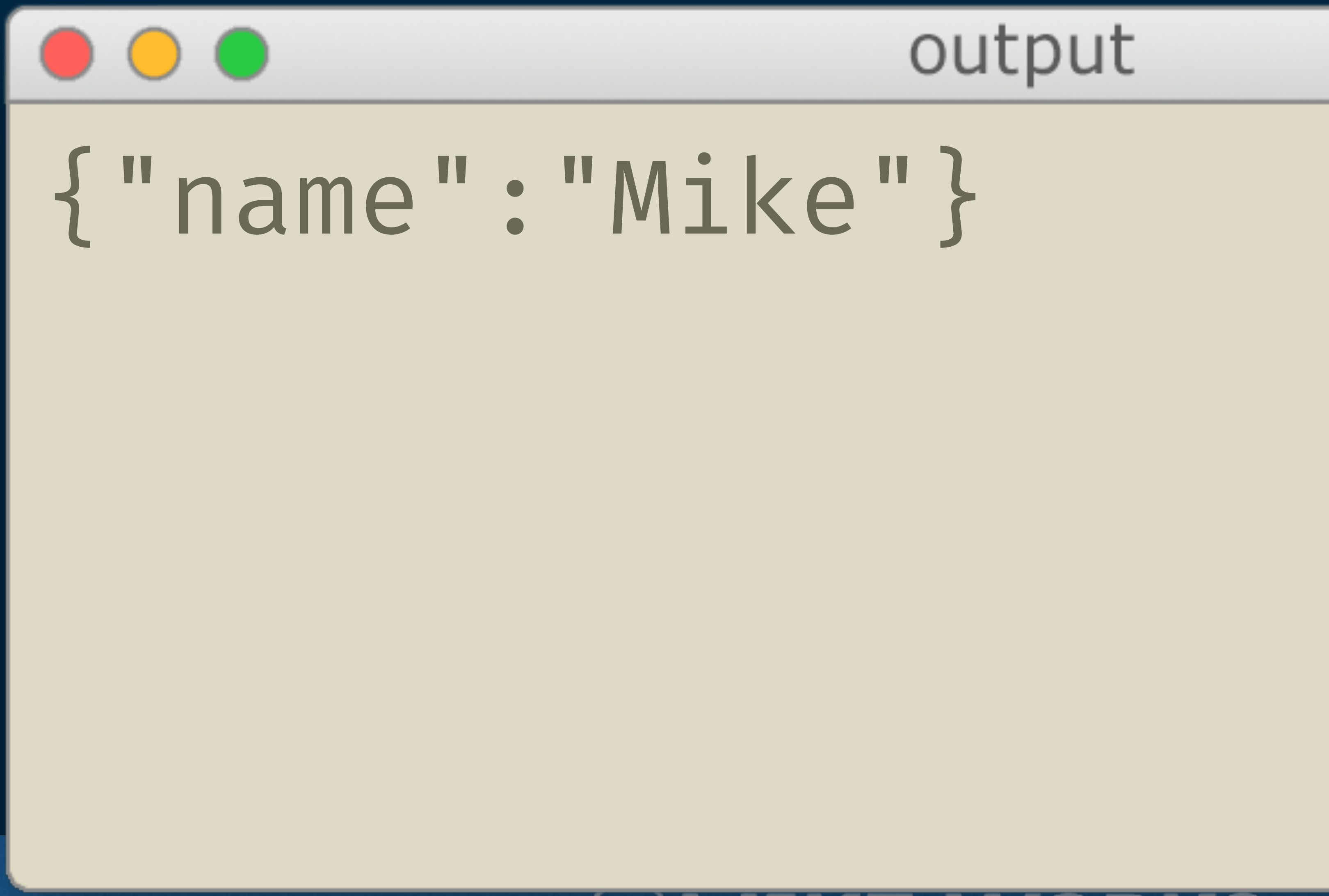
Classes - Mixins

JS

- Mixins are abstract classes or "templates for classes"

```
const AsJSON = x => class extends x {
  asJSON() {
    return JSON.stringify(this);
  }
};

class Person extends AsJSON(Object) {
  constructor(name) {
    super();
    this.name = name;
  }
}
```

```
let me = new Person('Mike');
console.log(me.asJSON());
```



```
{"name":"Mike"}
```

Classes

- Shape defined up front, like Interfaces
- Constructor for creating new instances
- Make sure to add type annotations properties AND function arguments

```
class Car {
  make: string
  model: string
  year: number
  constructor(make: string,
              model: string,
              year: number) {
    this.make = make;
    this.model = model;
    this.year = year;
  }
  startEngine() {
    return 'VROOOOOM!';
  }
}

let myCar =
  new Car('Honda', 'Accord', 2017);
```

Enums

TS

- Used to define a type consisting of ordered members
- Each has a name and a value
- Often we don't care about the value
  - ...beyond an equality check
- Get number of members via:

```
enum AcctType {
  Checking,
  Savings,
  MoneyMarket
};

type Acct =
  [number, AcctType];

let account: Acct = [
  9142.14, AcctType.Checking
];
```

Enums: JS Representation

TS

```ts
enum Suit {
  Club, Diamond, Heart, Spade
}
```

JS

```js
var Suit;
(function (Suit) {
    Suit[Suit["Club"] = 0] = "Club";
    Suit[Suit["Diamond"] = 1] = "Diamond";
    Suit[Suit["Heart"] = 2] = "Heart";
    Suit[Suit["Spade"] = 3] = "Spade";
})(Suit || (Suit = {}));
```

Enums: JS Representation

TS

```
enum Suit {
  Club, Diamond, Heart, Spade
}
```

JS

```
var Suit;
(function (Suit) {
    Suit["Club"] = 0
    Suit[0] = "Club";
    Suit["Diamond"] = 1
    Suit[1] = "Diamond";
    Suit["Heart"] = 2
    Suit[2] = "Heart";
    Suit["Spade"] = 3
    Suit[3] = "Spade";
})(Suit || (Suit = {}));
```

Number of members

```
Object.keys(Suit).length / 2; // 4
```

Arrays

- By default, arrays work same as in JavaScript
- Adding a type constraint helps us keep contents consistent
- When initializing class properties with empty arrays, provide a type
  - I'll explain more later

```
let a = [];
a.push(5);
a.push("not a number");

let nums: number[] = [1, 2, 3];


class ShoppingCart {
  items = [];
  constructor() {
    this.items.push(5);
  }
}
```

```
🛑Argument of type '5' is not
assignable to parameter of type
'never
```

Arrays

- By default, arrays work same as in JavaScript
- Adding a type constraint helps us keep contents consistent
- When initializing class properties with empty arrays, provide a type
  - I'll explain more later

```
let a = [];
a.push(5);
a.push("not a number");

let nums: number[] = [1, 2, 3];
```

```
class ShoppingCart {
  items: number[] = [];
  constructor() {
    this.items.push(5);
  }
}
```

✅

Tuples

- Arrays of fixed length
- Typically represent values that are related in some way
- Consumers need to know about order
- Shines with destructured assignment

```
let dependency: [string, number];
dependency = ['react', 16];

let dependencies: [string, number][] = [];

dependencies.push(dependency); // ✅
dependencies.push([
  'webpack', 3
]); // ✅

dependencies.push([
  'typescript', '2.5'
]);
/* 🛑 Argument of type '[string, string]' is
not assignable to parameter of type '[string,
number]'.
Type 'string' is not assignable to type
'number'. */
```

Type Aliases

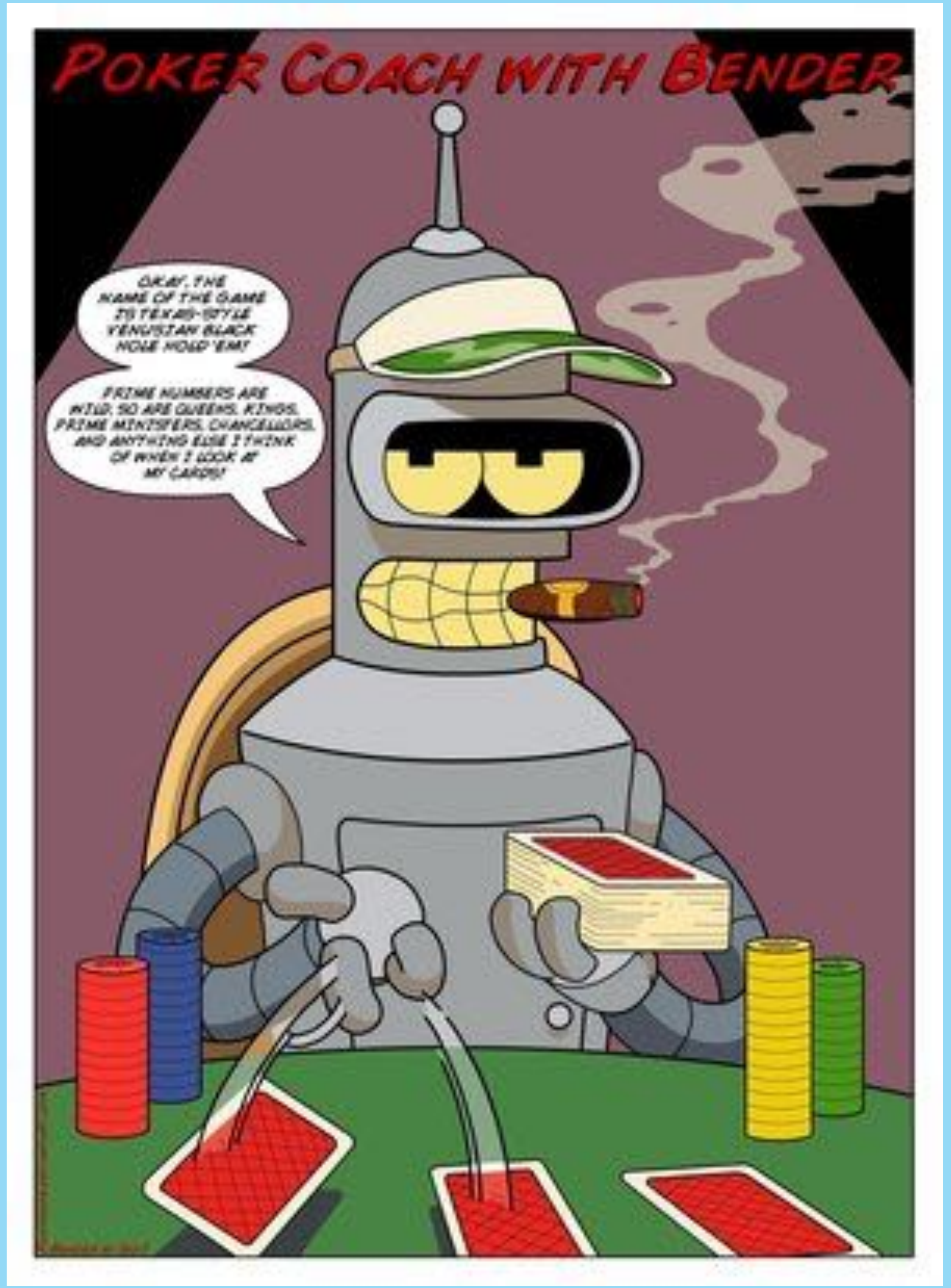
- Sometimes an interface isn't the best way to describe a structure
- We can use the `type` keyword to define a type alias

```
type Color = [number, number, number];

let red: Color = [255, 0, 0];
```

- You can export types and interfaces, so you can consume them in other modules!

Card Dealing

- Implement a card dealer, using enum types to represent Suit and CardNumber
    - 0 = clubs, 1 = diamonds, 2 = hearts, 3 = spades
- Cards should be represented as [suit, cardnumber] (i.e., [0, 6] is "Seven of Clubs")
- Each dealer should have its own deck of cards
- Pass all of the currently failing tests


POKER COACH WITH BENDER
OKAY, THE NAME OF THE GAME IS TEXAS-STYLE VENUSIAN BLACK HOLE HOLD 'EM!
PRIME NUMBERS ARE WILD; SO ARE QUEENS, KINGS, PRIME MINISTERS, CHANCELLORS, AND ANYTHING ELSE I THINK OF WHEN I LOOK AT MY CARDS!

```
npm test dealer
```

Card Dealing

- Dealer
  - `dealHand(5) → deals 5 cards [Suit(0-3), Number(0-12)]`
  - `getLength() → number of cards left in the deck`
  - `readCard(card) → "Seven of Spades"`
  - `Make sure to shuffle your cards!`

```
npm test dealer
```

🚂 Object Literals

JavaScript Objects

JS

{ }

JavaScript Objects

JS

```
{
  name: 'Mike',
  age: 34
}
```

JavaScript Objects

JS

```
{
  name: 'Mike',
  age: 34,
  toString: function() {
    return `${this.name} - ${this.age}`;
  }
}
```

Enhanced Object Literal

JS

```
let company = 'linkedin';
let mike = {
  __proto__: MyObject.prototype,
  name: 'Mike',
  age: 34,
  company,
  [`${company}Title`]: 'Staff Engineer',
  toString() {
    return `${super.toString()} + ${this.name} - ${this.age}`;
  }
}
```

specify prototype at construction

shorthand for `company: company`

dynamic property name

super calls

methods

Destructured Assignment

JS

- A cleaner way to pluck one or more deep properties off of an object

```
let person = {
  name: {
    first: 'Mike',
    last: 'North'},
  languages: {
    backend: {
      elixir: {
        experience: '3 years'}}}};
let {
  name: { first },
  languages: { backend: serverSkills } } = person;

console.log(`${first} - ${Object.keys(serverSkills)}`);
```

optional renaming

output

```
Mike - elixir
```

Object - Rest and Spread Properties

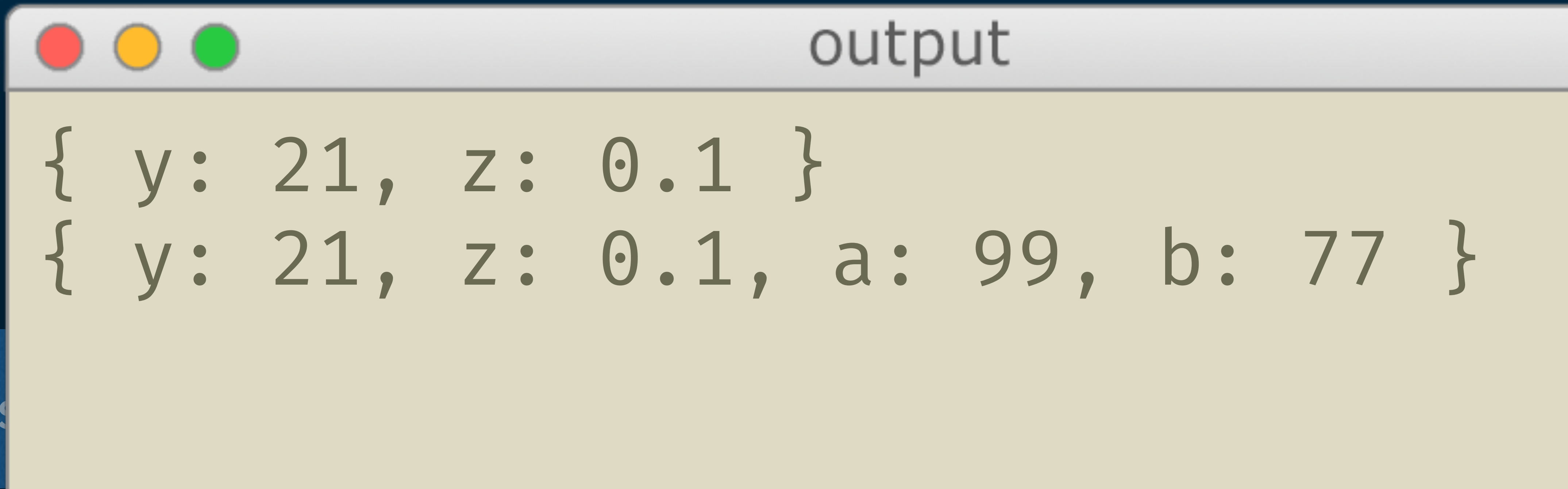
ES2018: STAGE 3

- Rest - sugar for "and the rest go here" when destructuring

```
let data = { x: 34, y: 21, z: 0.1 };
let { x, ...others } = data;
console.log(others);
```

- Spread - sugar for, "and all the properties on this object" when defining a {}

```
let values = {...others, a: 99, b: 77 };
console.log(values);
```



```
{ y: 21, z: 0.1 }
{ y: 21, z: 0.1, a: 99, b: 77 }
```

Getters & Setters

JS

- Getters are methods that return the value of a property
- Setters are methods that handle the setting of a property
- From the outside world, we treat it like any other "value based" property

```
let name = {
  first: 'Michael',
  last: 'North',
  get full() {
    return `${this.first} ${this.last}`;
  },
  set full(newVal) {
    let [a, b] = newVal.split(/\s+/g);
    this.first = a;
    this.last = b;
  }
}

console.log(name.first, name.last);
name.full = 'Mike North';
console.log(name.first, name.last);
```

Functions: Types

TS

- Functions have a type just like any other value

```
let login: (username: string, password: string) => User;
```

A function type

```
login = (username, password) => { return new User(); };
```

A function value

Functions: Types

TS

- Interfaces aren't just for describing object structures
- Here's one describing a function type
- Note the `this` property in the interface...

```
interface ClickListener {
  (this: Window, e: MouseEvent): void
}

const myListener: ClickListener =
  e ⇒ {
    console.log('mouse clicked!', e);
  }

addEventListener('click', myListener);
```

Functions: Types

- Interfaces aren't just for describing object structures
- Here's one describing a function type
- Note the `this` property in the interface…

```
interface ClickListener {
  (this: Window, e: MouseEvent): void
}

const myListener: ClickListener =
  e => {
    console.log('mouse clicked!', e);
  }

addEventListener('click', myListener); ✅

myListener(new MouseEvent('click'));
```

```
The 'this' context of type 'void'
     is not assignable to method's
          'this' of type 'Window'.
```

Functions: Required Parameters

TS

- Unless you say otherwise, TypeScript assumes every argument in the function is required

```
function createTwitterPost(body: string,
                           username: string,
                           imageUrl: URL) {
  // ...
}

createTwitterPost('I ate a ham sandwich today.', 'MichaelLNorth');
🛑 Expected 3 arguments, but got 2.
```

Functions: Optional Parameters

- We can fix this with an **optional parameter**.

```
function createTwitterPost(body: string,
                           username: string,
                           imageUrl?: URL) {
  // ...
}

createTwitterPost('I ate a ham sandwich today.', 'MichaelLNorth');
✅
```

Functions: Default Parameter Values

JS

- We can also provide a **default value** to use, in the event an argument isn't passed

```
function createTwitterPost(body: string,
                           username: string = 'MichaelLNorth',
                           imageUrl?: URL) {
  // ...
}

createTwitterPost('I ate a ham sandwich today.');
```

✅

Functions: Rest Parameters

JS

- A (boundless) group of optional parameters

```
function orderSandwich(bread:string,
                       name:string,
                     ... toppings: string[]) {
  /*  ...  */
}

orderSandwich('Bagel', 'Ham & Cheese');
orderSandwich('Wheat', 'Turkey Club', 'Mustard', 'Sprouts');
```

Functional Cashier

- We're building a shopping cart that can be used as follows

```
let cart = cashier();

cart
  .add('Apple', 0.99) // Add one Apple
  .add('Pear', 1.99, 2) // Add two Pears
  .addItem({ // Add three Banannas
    name: 'Bananna',
    price: 2.99,
    qty: 3});

console.log(`Your total for ${cart.length} items is $${cart.total}`);
```

```
npm test cashier
```

Functional Cashier

- Items that can be added to the cart as an object should look like this

Cart Item

name	String
price	Number
qty	Number

- And the object returned by the `cashier()` function (and `add`, `addItem`) should look like

CartAPI

length	Number
total	Number
addItem	Takes a cart item, returns a CartAPI
add	Takes (name, price, qty), returns a CartAPI

```
npm test cashier
```

Generics

TS

- Generics allow us to reuse code across many types, interfaces and functions
- We still get compile-time type safety!

Type Parameter

Determined by argument type

```
function gimmieFive<T>(x: T): T[] {
  return [x, x, x, x, x];
}

let threes: number[] = gimmieFive(3);
let eggs: string[] = gimmieFive('egg');
```

Generics

- Arrays can be expressed this way too

```
let cards = Array<[Suit, CardNumber]>(52);
```

- So can Promises

```
let data: Promise<Response> = fetch('http://example.com');
```

- And React components, which we'll look at later!

```
interface MyProps {title: string}
interface MyState {isClicked: boolean}
class MyComponent extends Component<MyProps, MyState> { }
```

Generics

TS

- We can specify constraints on generic types

```
function midpoint<T extends Point2D>(p1: T, p2: T): T {

}
```

- Generics can be used with interfaces as well

```
interface IFileReader<T extends File> {
  readFile(file: T): Blob
}
```

Access Modifier Keywords

- **public** - anyone can access
- **protected** - self and subclasses can access
- **private** - self can access

```
class Account {
  protected email: string;
  private password: string;
  public accountId: number;
}
```

```
class SharedAccount extends Account {
  setEmail(newEmail: string) {
    this.email = newEmail;
  }
}
```

Function Overloading

- TypeScript allows us to have more than one function "head", although we're still limited to a single implementation

```
function add(x: number, y: number): number;
function add(x: string, y: string, radix: number): number;

function add(x: number|string,
             y: number|string,
             radix: number=10): number {
  return parseInt(`${x}`, radix) + parseInt(`${y}`, radix);
}
```

Function Overloading

TS

```
add('3', 4);
```

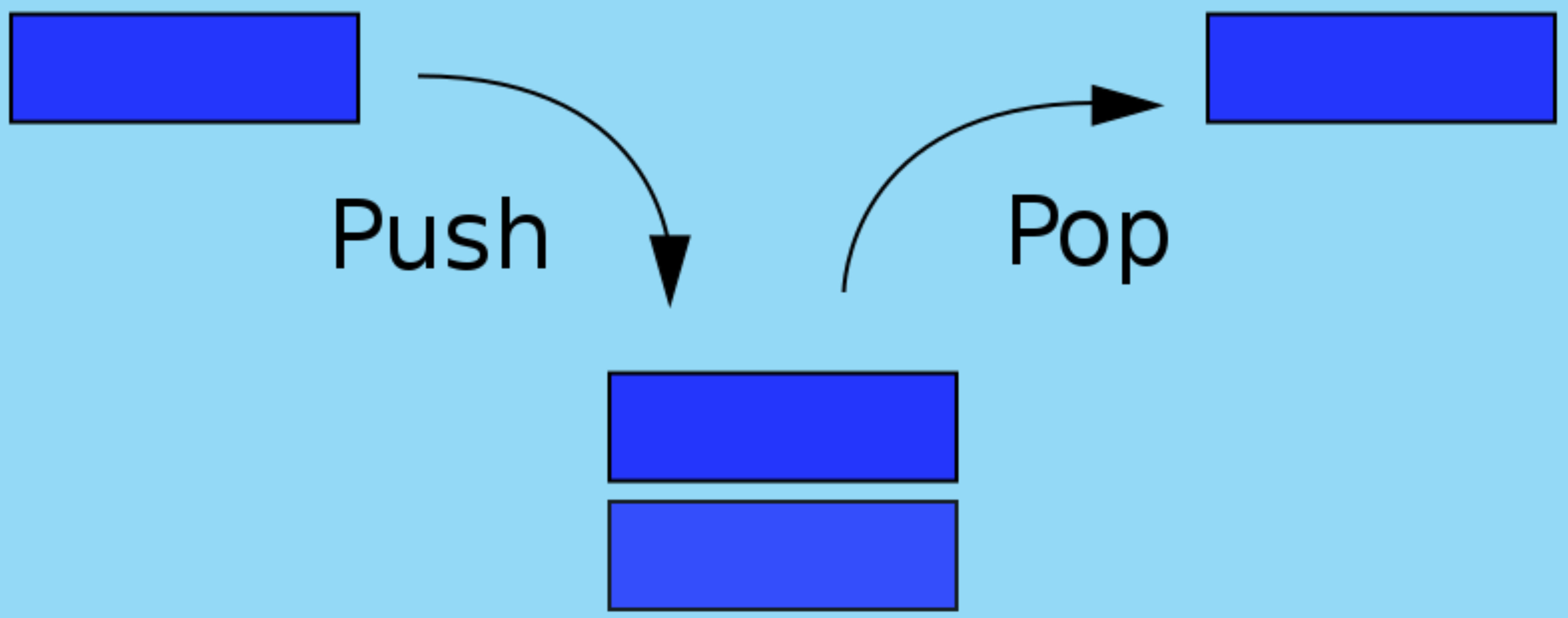
```
Argument of type '"3"' is
not assignable to parameter
of type 'number'.
```

- Must specify return type of each function
- More specific function signatures come first
- The signature of your implementation is not directly available for use!

```
function add(x: number, y: number): number;
function add(x: string, y: string, radix: number): number;

function add(x: number|string,
             y: number|string,
             radix: number=10): number {
  return parseInt(`${x}`, radix) + parseInt(`${y}`, radix);
}
```

Typed Stack

5



- Build a Stack data structure that uses generics to constrain the types it accepts
- Overload the push function, so that it can either take a single item or an array
- Pop should return an object of the appropriate type
- Keep the internal data structure private

```
let l = new Stack<string>();
l.push(['cherry', 'apple', 'grape']);
l.push('lemon');

l.pop(); // 'lemon'
l.pop(); // 'grape'
l.pop(); // 'apple'
l.pop(); // 'cherry'
l.pop(); // undefined
```

```
npm test stack
```

Typed Stack

```
interface IStack<T> {
  push(item: T): IStack<T>;
  push(items: T[]): IStack<T>;
  pop(): T | undefined;
  length(): number;
  print(): void;
}
```

```
npm test stack
```

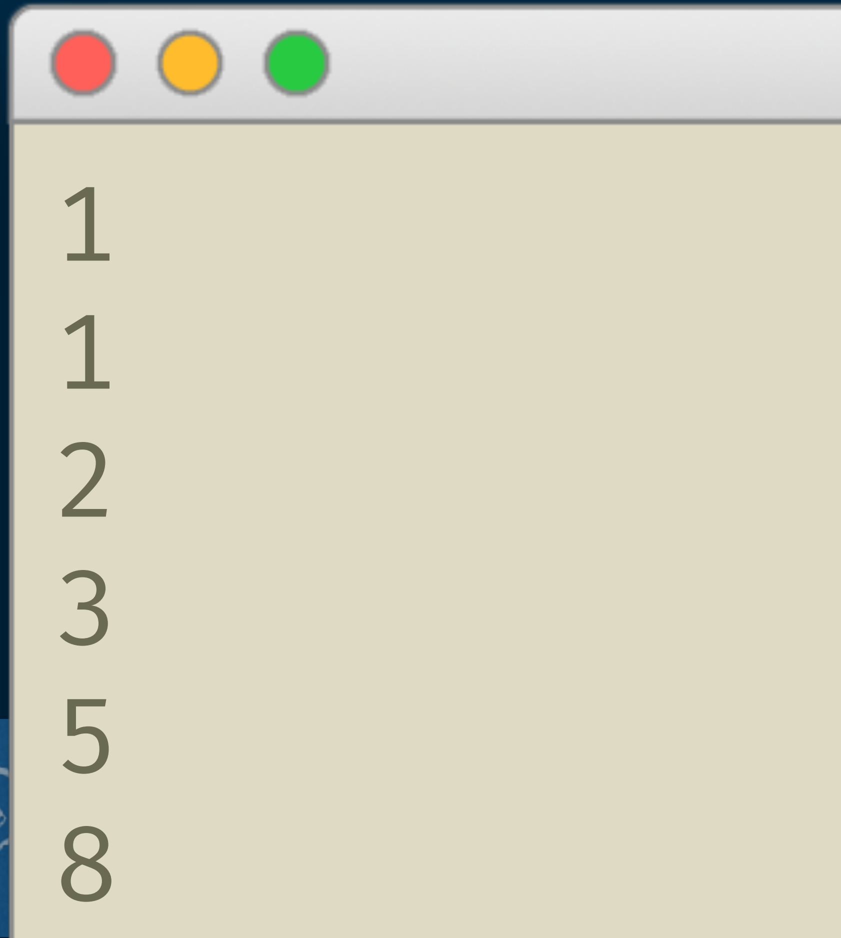
🤹‍♀️ Iterators & Generators

Iterators

```
function fibonacci() {
  let lastLast = 1;
  let last = 0;
  return {
    next() {
      let val = last + lastLast;
      if (val > 10) { // Termination
        return { done: true };
      }
      lastLast = last;
      last = val;
      return { value: val, done: false };
    }
  };
}
```

- Iterators allow access one item from a collection at a time, keeping track of current position
- the `next()` method is what's used to get the next item in the sequence.

```
let it = fibonacci();
for (let p = it.next(); !p.done; p = it.next())
{
  console.log(p.value);
}
```



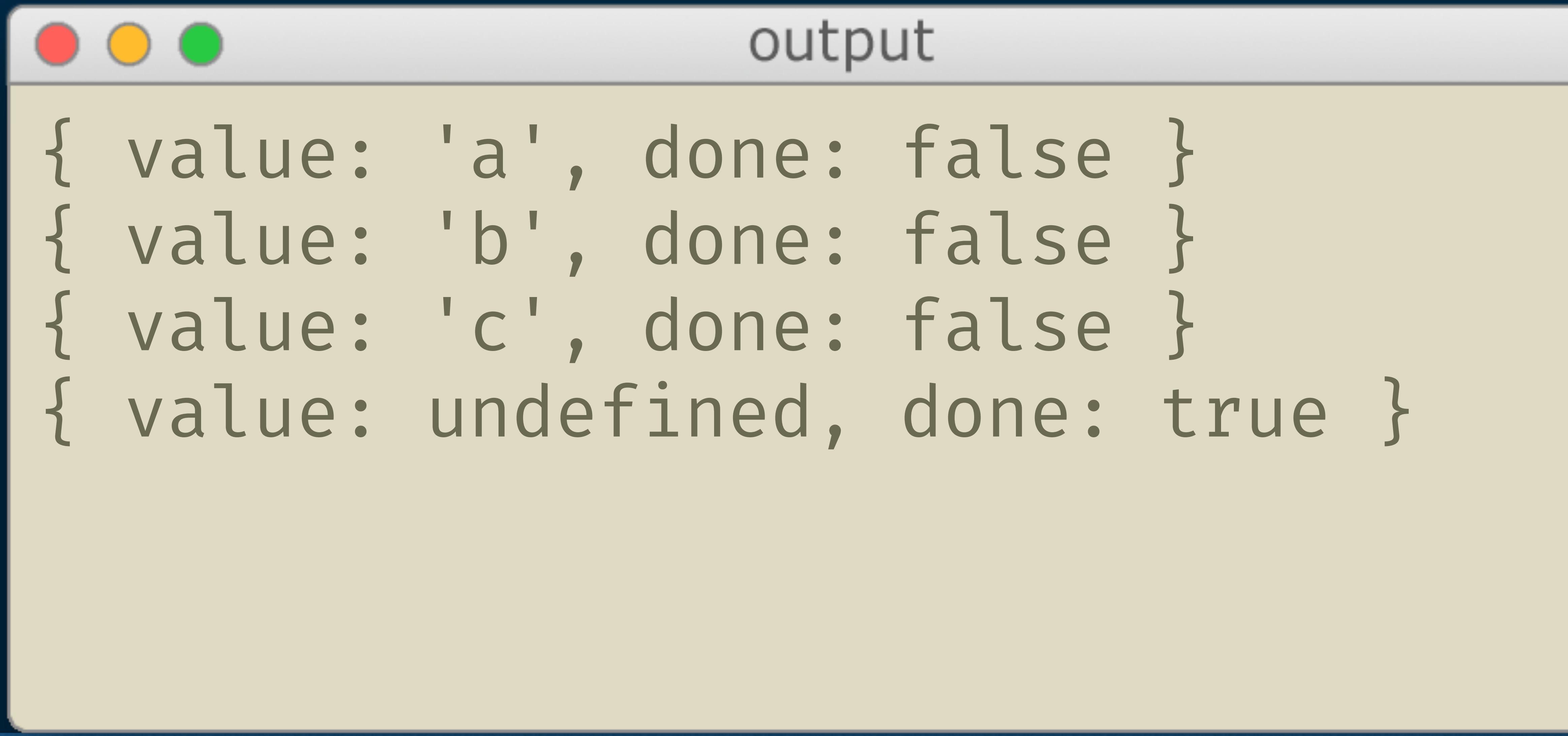
```
1
1
2
3
5
8
```

Iterables

JS

- Support iteration within a `for .. of` loop
- Requires implementation of the `Symbol.iterator` method
- `Array` and `Map` already support this!

```
let arr = ['a', 'b', 'c'];
let it = arr[Symbol.iterator]();
console.log(it.next());
console.log(it.next());
console.log(it.next());
console.log(it.next());
```


output

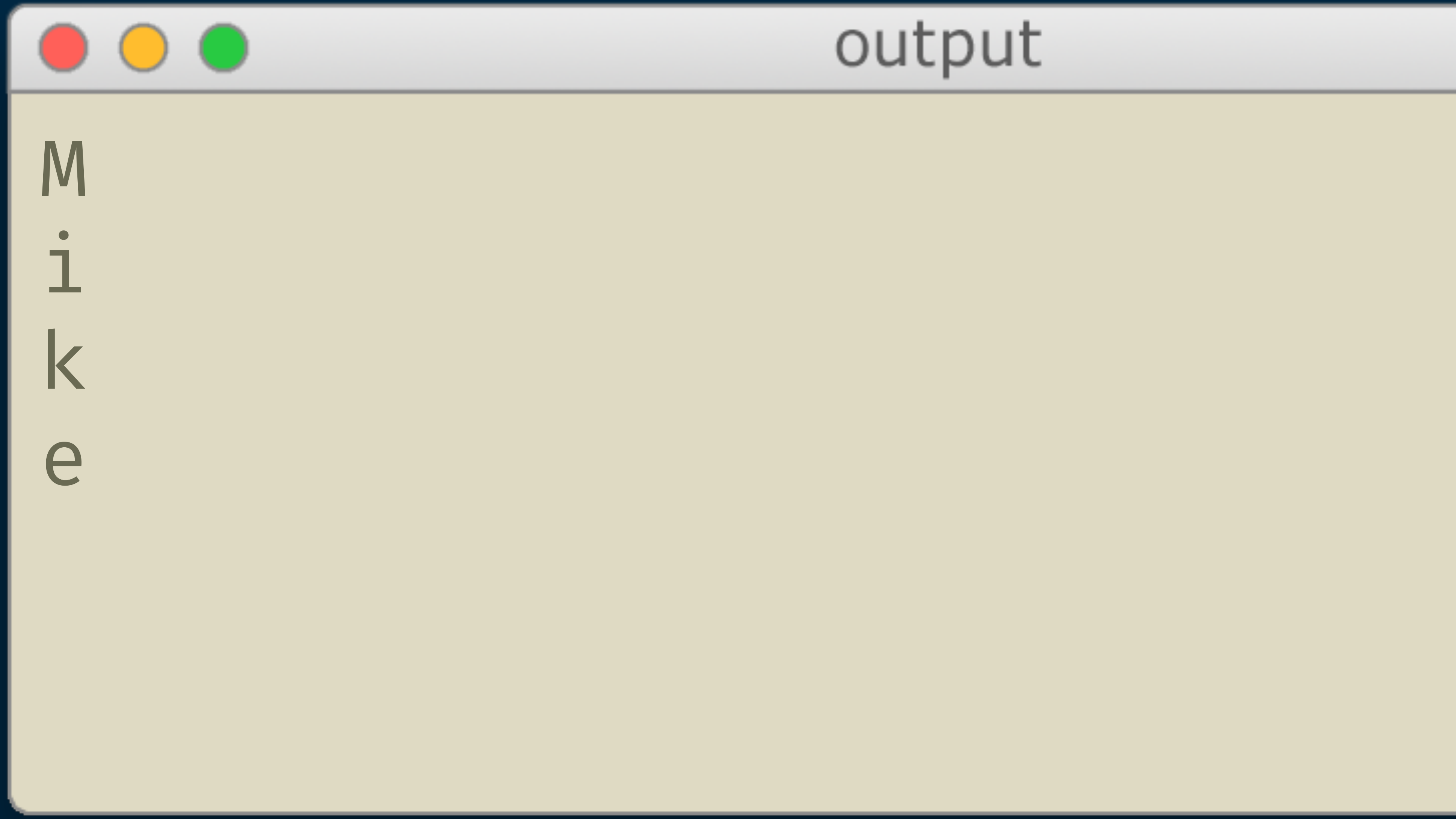
```
{ value: 'a', done: false }
{ value: 'b', done: false }
{ value: 'c', done: false }
{ value: undefined, done: true }
```

Iterables - Defining our own iterable

JS

```
let mike = {
  _name: 'Mike',
  [Symbol.iterator]() {
    let i = 0;
    let str = this._name;
    return {
      next() {
        if (i < str.length) {
          return {done: false, value: str[i++] };
        }
        return { done: true };
      }
    };
  }
};
```

```
for (let m of mike) {
  console.log(m);
}
```

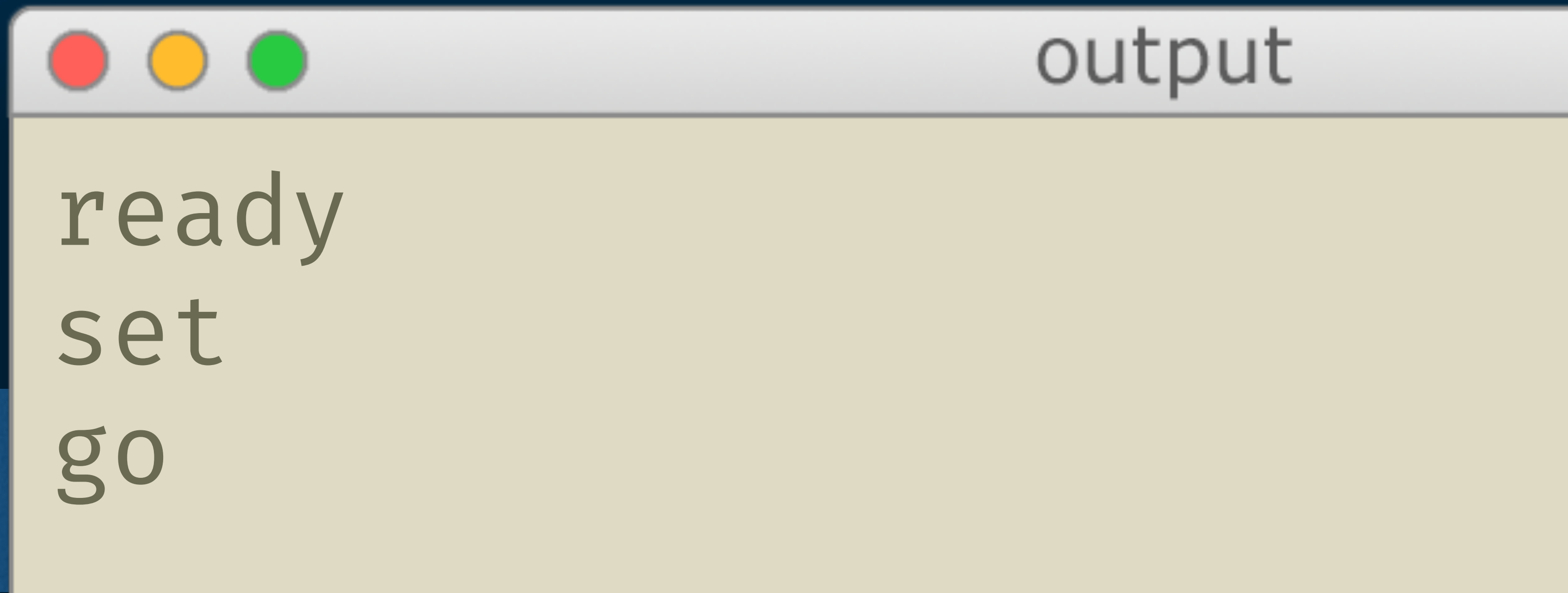


```
M
i
k
e
```

Generators

- Define their own iterative algorithm, yielding each item in the sequence
- Use the `function*()` syntax
- Returns an iterator
- State of the closure is preserved between `.next()` calls.
- EXECUTION IS PAUSED

```
function* fib() {
  let lastLast = 1;
  let last = 0;
  while (true) {
    let val = last + lastLast;
    yield val;
    lastLast = last;
    last = val;
  }
}

for (let x of fib()) {
  console.log(x);
}
```



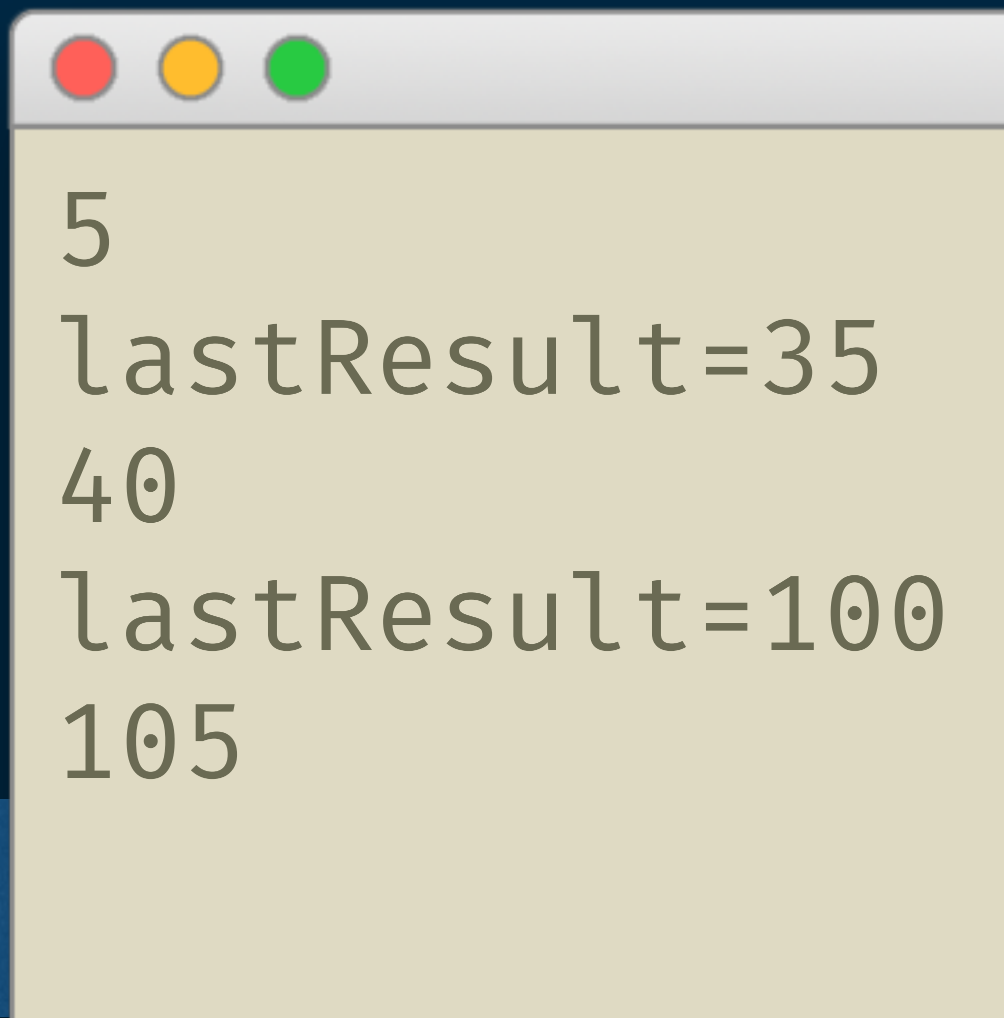
```
ready
set
go
```

Iterators

JS

- The ability to pass values IN while iterating is important, and serves as the foundation for many great JavaScript patterns

```
function* sequence() {
  let lastResult = 0;
  while(true) {
    lastResult = yield lastResult + 5;
    console.log(`lastResult=${lastResult}`);
  }
}

let it = sequence();
console.log(it.next().value);
console.log(it.next(35).value);
console.log(it.next(100).value);
```

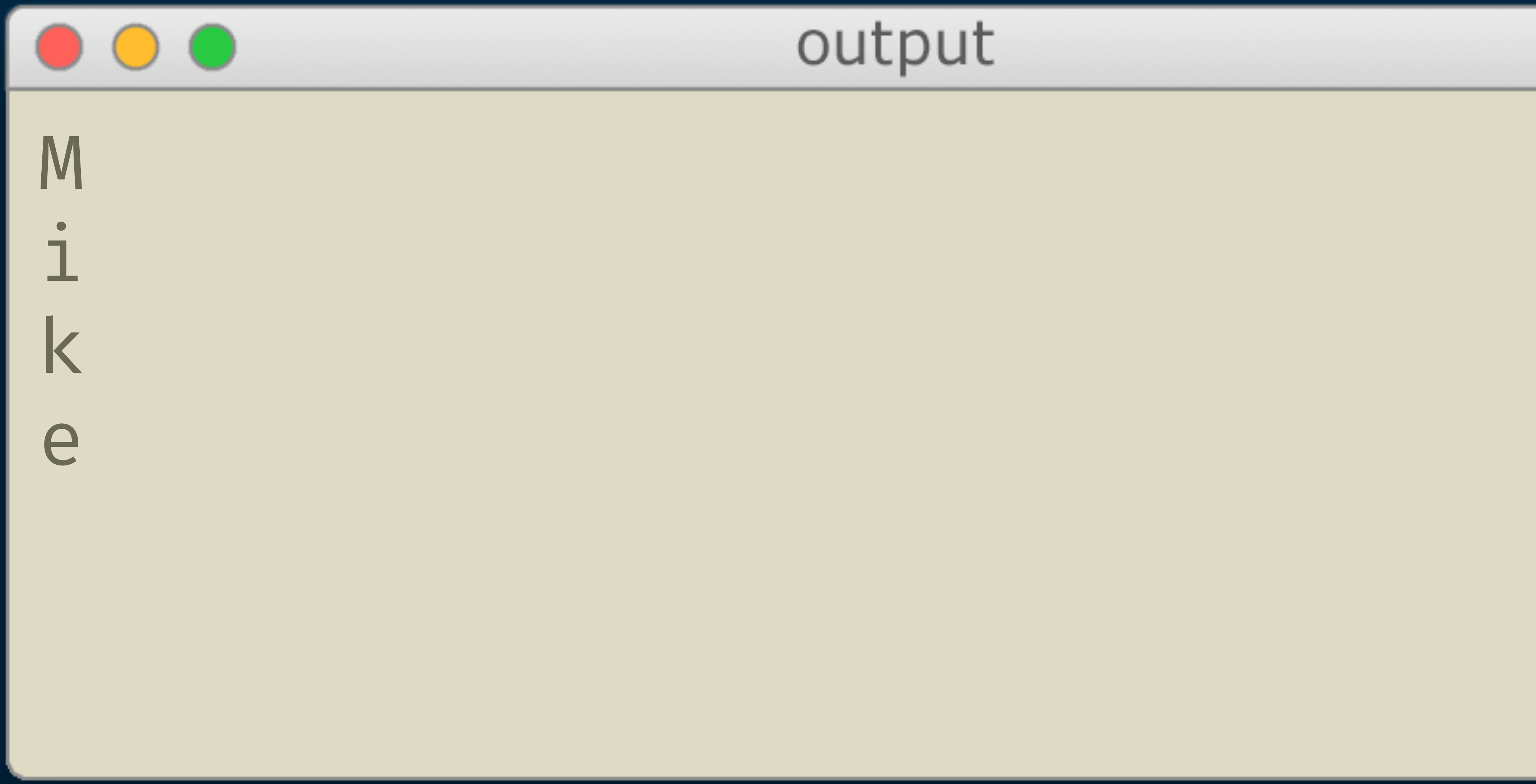


```
5
lastResult=35
40
lastResult=100
105
```

Iterables - Defining our own iterable

- Generator function makes this very simple

```
let mike = {
  [Symbol.iterator]: function*() {
    yield 'M';
    yield 'i';
    yield 'k';
    yield 'e';
  }
}
for (let m of mike) {
  console.log(m);
}
```

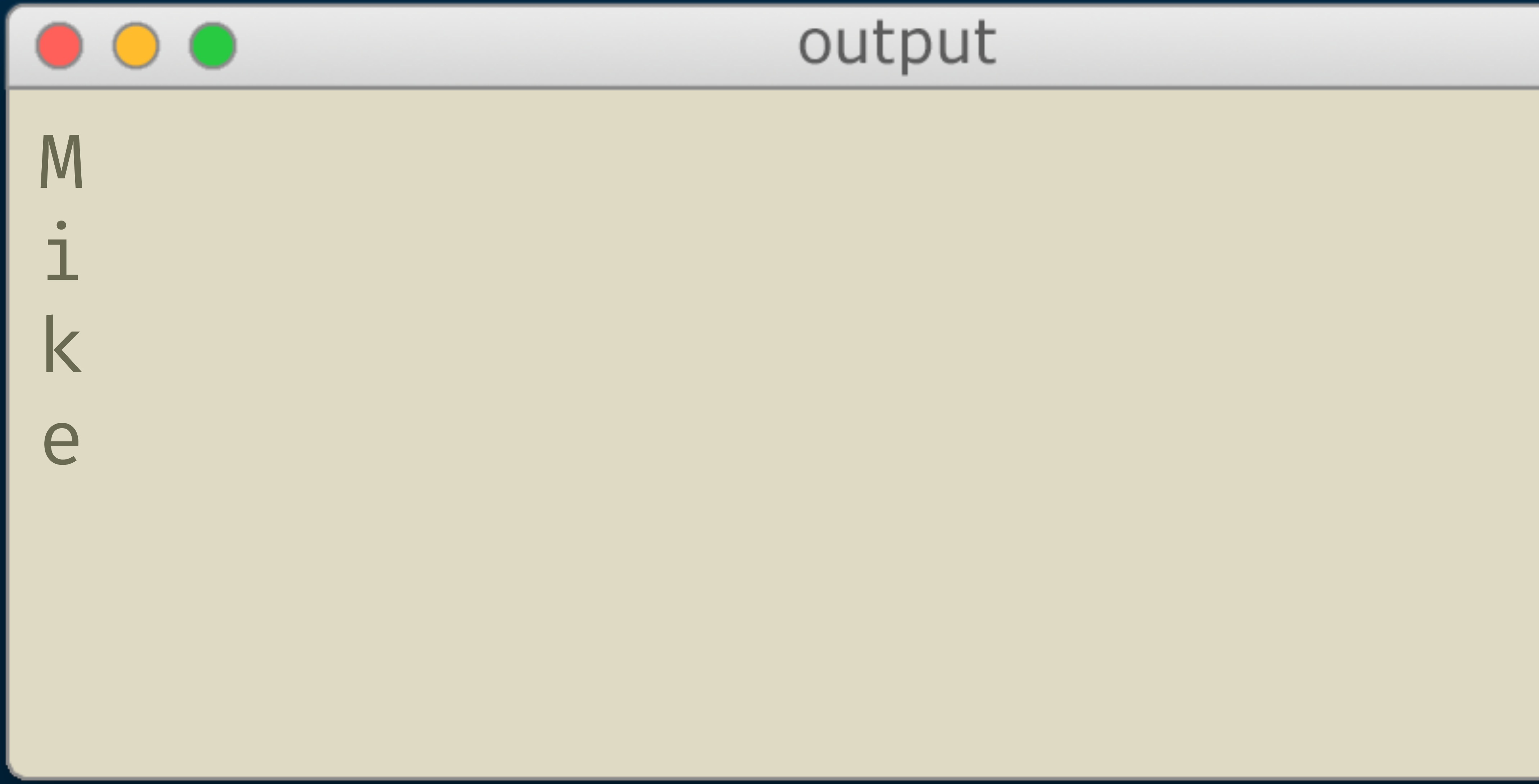


```
M
i
k
e
```

Using Iterables - yield*

- In generator functions, the `yield*` keyword will yield each value of an iterable, one by one.

```
let mike = {
  [Symbol.iterator]: function*() {
    yield* 'Mike';
  }
}
for (let m of mike) {
  console.log(m);
}
```



```
M
i
k
e
```

Using Iterables - Destructured Assignment

- Destructured assignment works with any iterable, not just arrays!

```
let nums = {
  [Symbol.iterator]: function*() {
    yield* [1, 2, 3];
    yield 4;
    yield* [5, 6];
  }
}

console.log([ ... nums]);
```

output

```
[ 1, 2, 3, 4, 5, 6 ]
```

Fibonacci Generator

- Build a generator function that returns an iterator, which emits the numbers of the fibonacci sequence
- Protip

Two ago	1	0	1	1	2	3
One ago	0	1	1	2	3	5
Number	1	1	2	3	5	8

```
let it = getFibSequence();

it.next().value; // 1
it.next().value; // 1
it.next().value; // 2
it.next().value; // 3
it.next().value; // 5
```

```
npm test fib
```

🤹‍♀️ React + TypeScript

React - Stateless Functional Components

- Interfaces used to describe props

```
import * as React from 'react';

interface IMyComponentProps {
  name: string;
}

const MyComponent: React.SFC<IMyComponentProps> = ( props ) ⇒ {
  return (  <div> {props.name} </div>  )
};
```

React - Stateless Functional Components

- Interfaces used to describe props

```
const MyComponent: React.SFC<IMyComponentProps> = ( props ) ⇒ {
  return (  <div> {props.name} </div>  )
};

const App: React.SFC = ( ) ⇒ {
  return ( <MyComponent name="foo" /> );
};
```

React - Stateless Functional Components

DEMO

Autocomplete I

- Build a stateless functional react component for the PlaceDetails type
- It should include at least 5 pieces of data from the interface to the right
- Look at the tests/__snapshots__ folder for guidance on HTML structure

```
export interface PlaceDetails {
  id: string;
  rating: number;
  icon: string;
  name: string;
  url: string;
  vicinity: string;
  website?: string;
}
```

```
npm start autocomplete-sfc
```

React - Stateful Components

- Interfaces used to describe props AND state

```
import * as React from 'react';

interface IMyComponentProps {  name: string; }
interface IMyComponentState {  time: Date; }

class MyComponent extends React.Component<IMyComponentProps, IMyComponentState> {
  componentDidMount() {
    this.setState({ time: new Date() });
  }
  render() {
    return (  <div> {this.props.name} - { this.state.time.toISOString() } </div>  );
  }
};
```

React - Stateful Components

DEMO

Autocomplete II

8

- Build fill in the `PlaceSearchResultList` component
- App component should pass in `trySearch` function, and `PlaceSearchResultList` should bind it to an input's onChange event
- Be sure to handle "not yet searched", "in progress" and "we have results" scenarios
- The existing use of async/await is fine here

```
npm start autocomplete-2
```

Autocomplete III

- Build a `PlaceSearchContainer` component, which manages state, but does nothing to trigger its own re-rendering
- Fill in the `beginSearch` function, and rely on the async function in the `./autocomplete.ts` module to return a promise that resolves to `PlaceDetails[]`.
- The existing use of async/await is fine here

```
npm test autocomplete-3
```

Autocomplete IV

- We must build our own function in the `task.ts` module, which takes a generator function as an argument.
- Allow yield within the generator function to behave exactly as await would in an async function
- `task()` should return a promise that resolves to the last value yielded (or the value returned) by the generator function

```
npm start autocomplete-4
```

⚙️ Configuring TypeScript

tsconfig.json

Transform JSX

Enable "strict" features

Forbid implicit any

Check + compile JS

Output module type

```
{
  "compilerOptions": {
    "jsx": "react",
    "strict": true,
    "sourceMap": true,
    "noImplicitAny": true,
    "strictNullChecks": true,
    "allowJs": true,
    "types": [],
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "moduleResolution": "node",
    "target": "es2015"
  }
}
```

👥 Proxies

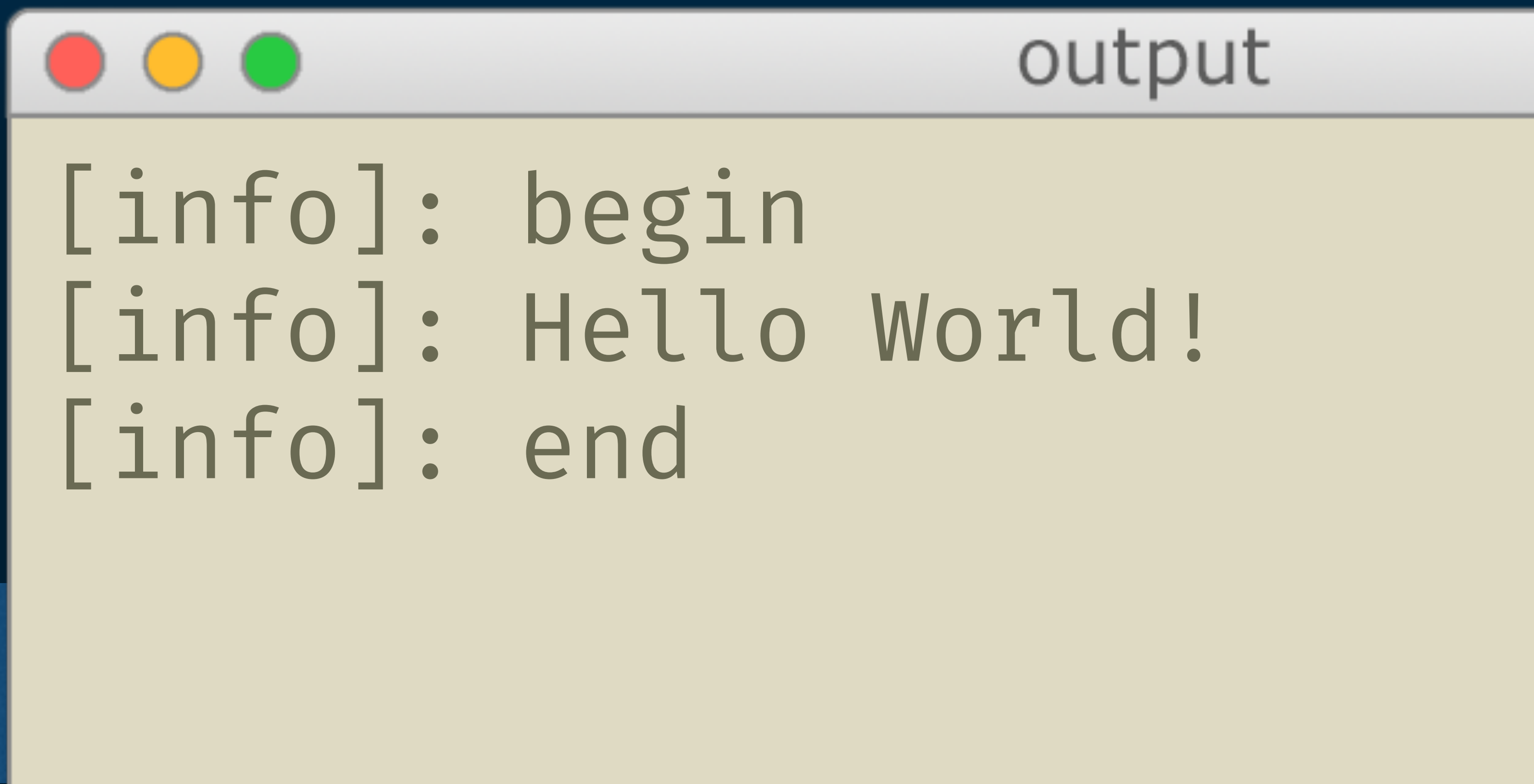
Proxy - Core Concept

ES2015

- Proxies allow a `handler` to intercept or customize certain operations, with respect to a `target` object, by implementing one or more `traps`.

```
let target = { level: 'info' };

let handler = {
  get(targ, prop) { // "get property" trap
    console.log(`[${targ.level}]: ${prop}`);
  }
};

let logger = new Proxy(target, handler);
```

```
logger.begin;
logger['Hello!'];
logger.end;
```



```
[info]: begin
[info]: Hello World!
[info]: end
```

Proxies - Traps

ES2015

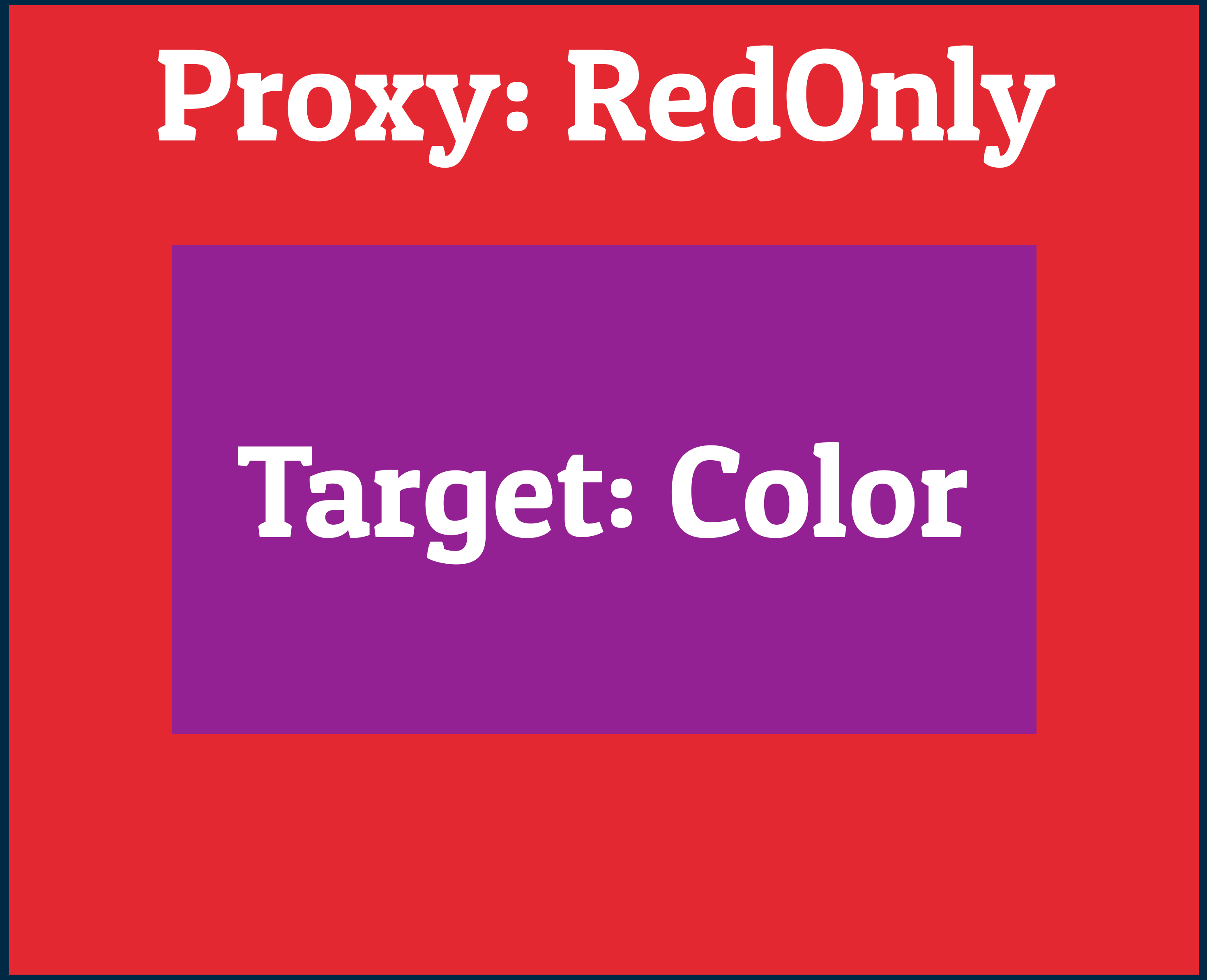
- When implementing "traps", we must stick to certain established conventions called invariants.
- Not doing so will cause a TypeError.

```
Object.getPrototypeOf(Employee);
Object.setPrototypeOf(Employee);
Object.isExtensible({});
Object.preventExtensions({});
Object.getOwnPropertyDescriptor({}, "foo");
Object.defineProperty({}, "foo", descriptor);
"house key" in keyring; // in operator
person.firstName; // getting props
person.firstName = "Mike" // setting props
delete person.firstName; // deleting props
Object.getOwnPropertNames(person) // own keys
makeRequest(); // function call
new Person(); // new operator
```

Proxy - Example

ES2015

- Proxies allow interception or customization of certain operations, w/ respect to a target object
- EXAMPLE: single-channel filters for colors
  - Target - the original color
  - Proxy - RedOnly, BlueOnly, only lets one color channel through
  - Handler - the thing we'll put in place to set blue and green channels to 0

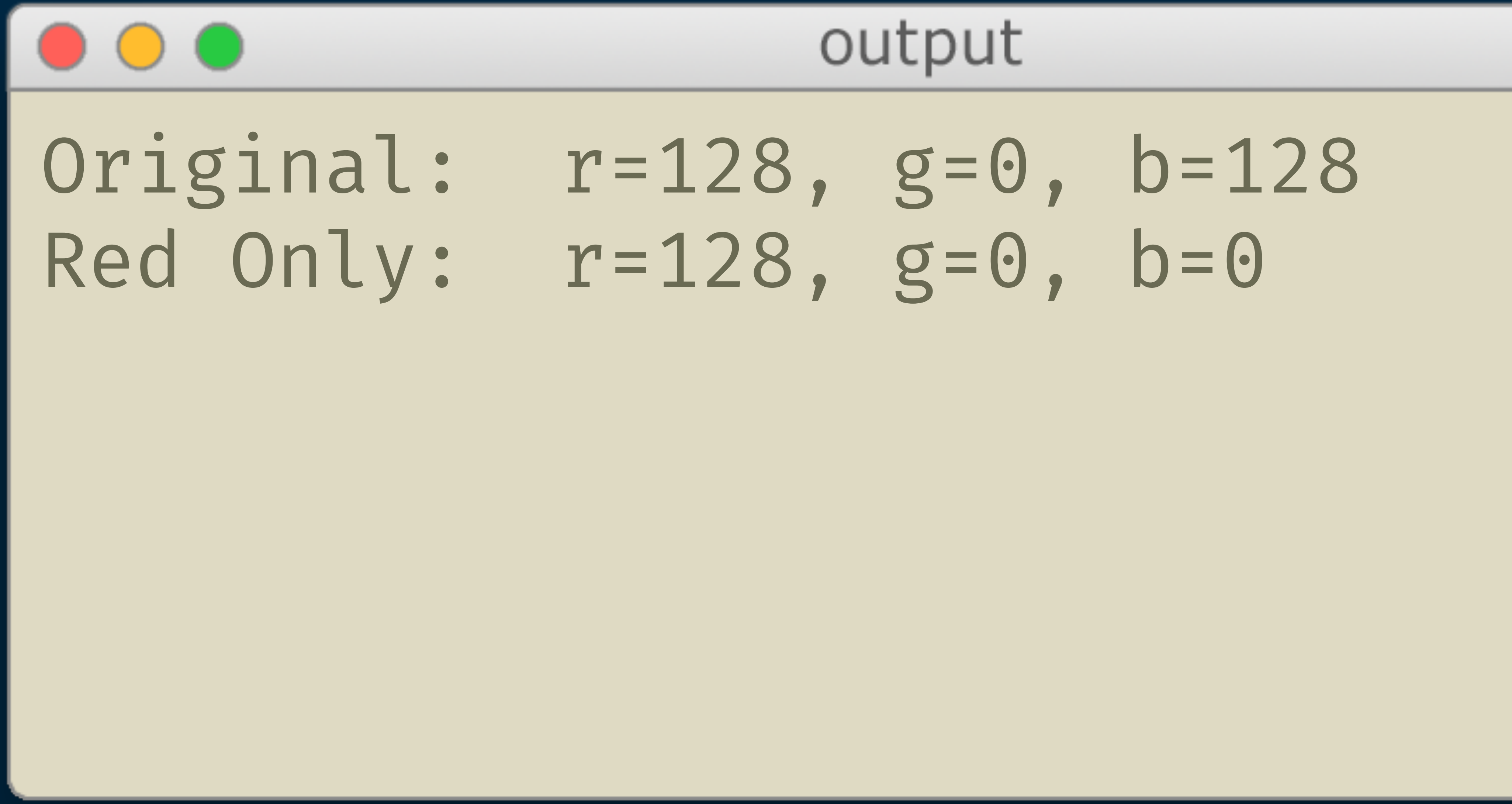


Proxy - Example

ES2015

```
class Color {
  constructor({r, g, b}) {
    this.r = r; this.g = g; this.b = b;
  }
  toString() {
    let { r, g, b } = this;
    return `r=${r}, g=${g}, b=${b}`;
  }
}

let color = new Color({r: 128, g: 0, b: 128 });
console.log(`Original:  ${color}`);

let redHandler = {
  get(target, prop) {
    if (prop === 'b' || prop === 'g') return 0;
    else return target[prop];
  }
}
```

```
let redOnly =
  new Proxy(color, redHandler);
console.log(`Red Only:  ${redOnly}`);
```



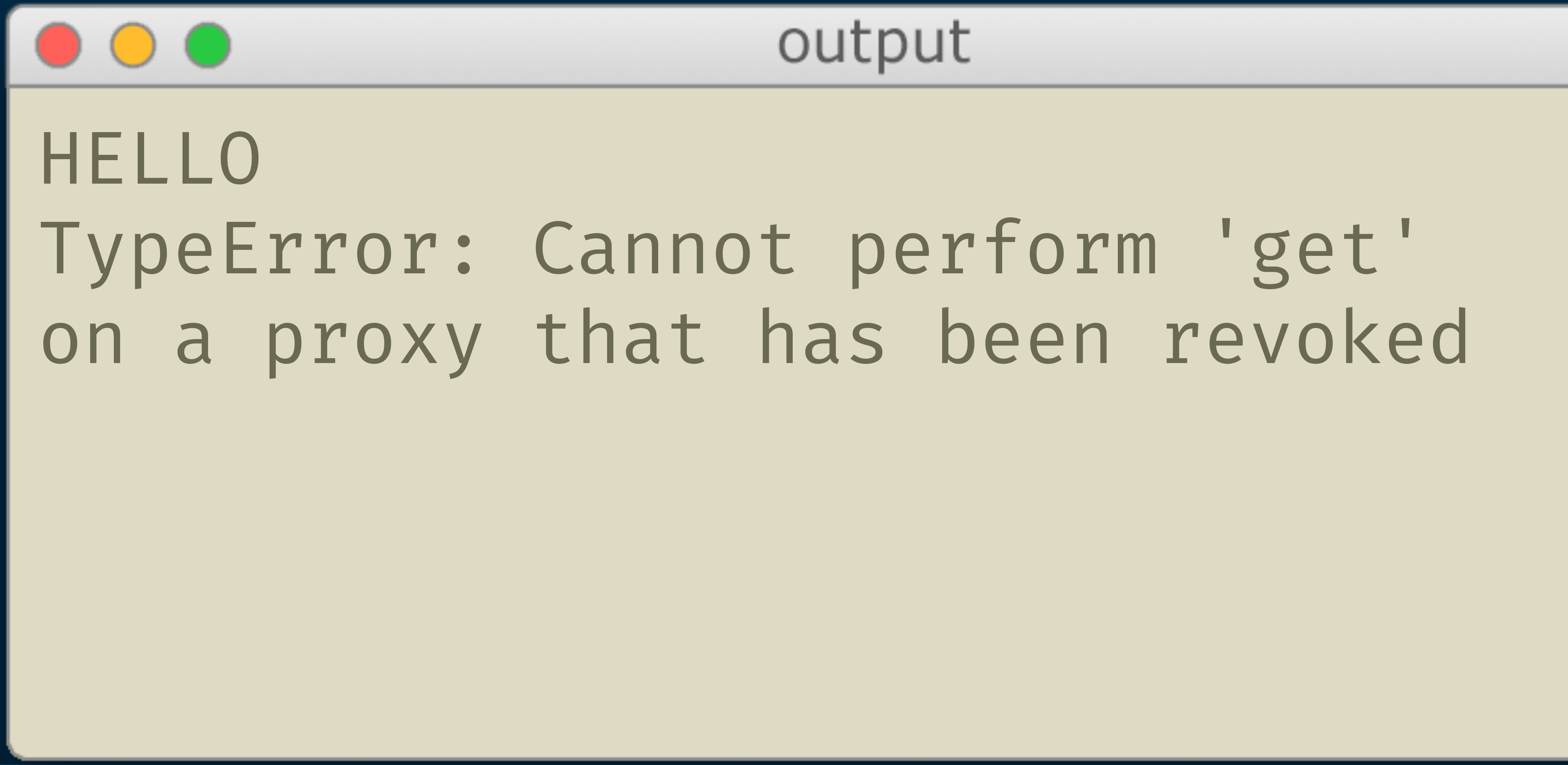
```
Original:  r=128, g=0, b=128
Red Only:  r=128, g=0, b=0
```

Proxies - Revoking

ES2015

- Proxies can be revokable
- Revoking a proxy makes it inoperable. Every trap throws a `TypeError`.

```
let { revoke, proxy } = Proxy.revocable({}, {
  get: function(target, name) {
    return name.toUpperCase();
  }
});
console.log(proxy.hello);
revoke();
console.log(proxy.bye);
```



```
HELLO
TypeError: Cannot perform 'get'
on a proxy that has been revoked
```