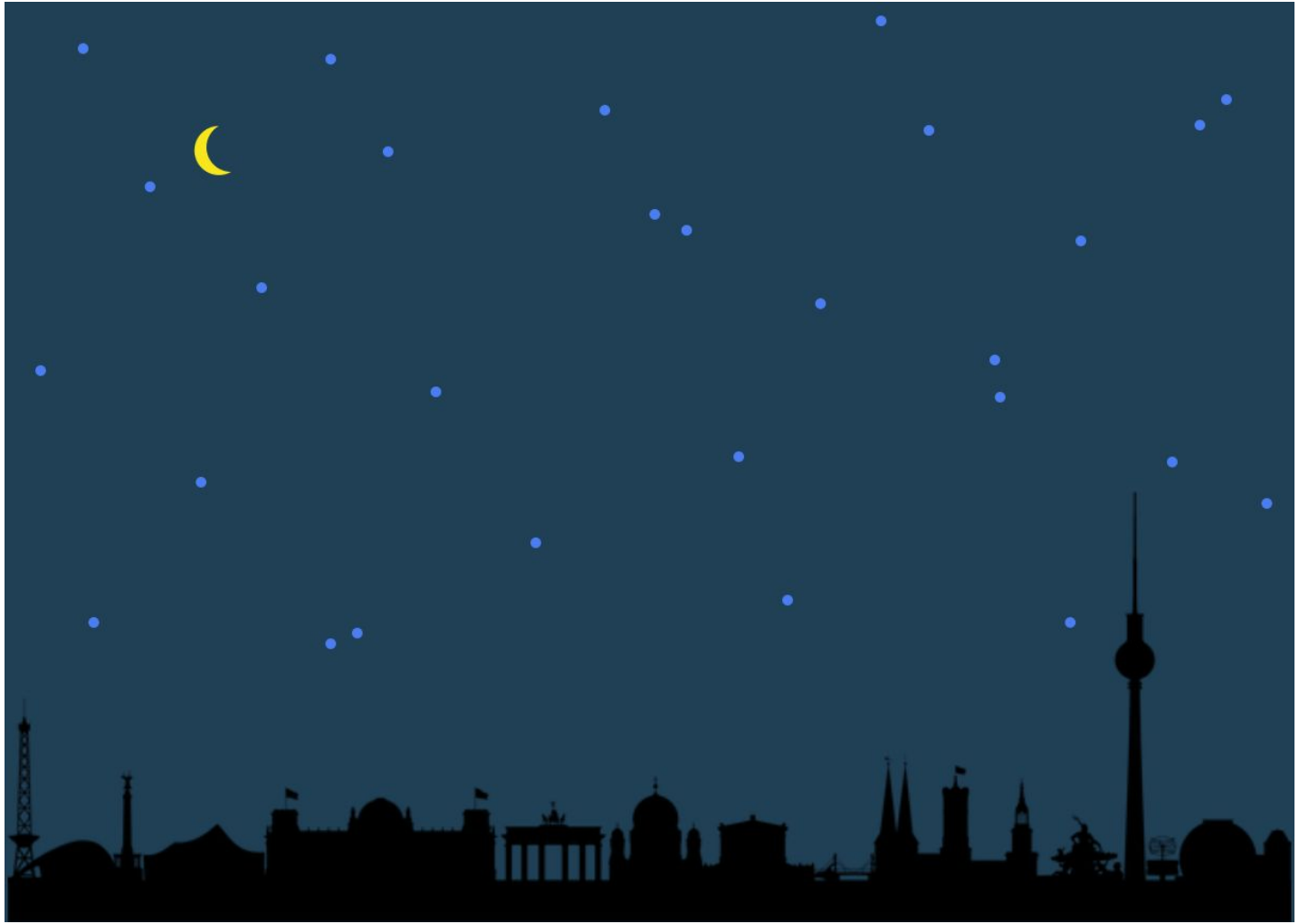
Scope

Darkhan Zhursin
daratechn@gmail.com

Scope

An important idea in programming is *scope*. Scope defines where variables can be accessed or referenced. While some variables can be accessed from anywhere within a program, other variables may only be available in a specific context.

You can think of scope like the view of the night sky from your window. Everyone who lives on the planet Earth is in the global scope of the stars. The stars are accessible *globally*. Meanwhile, if you live in a city, you may see the city skyline or the river. The skyline and river are only accessible *locally* in your city, but you can still see the stars that are available globally.



Blocks and Scope

Before we talk more about scope, we first need to talk about *blocks*.

We've seen blocks used before in functions and `if` statements. A block is the code found inside a set of curly braces `{}`. Blocks help us group one or more statements together and serve as an important structural marker for our code.

A block of code could be a function, like this:

```
const logSkyColor = () => {
  let color = 'blue';
  console.log(color); // blue
}
```

Notice that the function body is actually a block of code.

Observe the block in an `if` statement:

```
if (dusk) {
  let color = 'pink';
  console.log(color); // pink
}
```

Global Scope

Scope is the context in which our variables are declared. We think about scope in relation to blocks because variables can exist either outside of or within these blocks.

In *global scope*, variables are declared outside of blocks. These variables are called *global variables*. Because global variables are not bound inside a block, they can be accessed by any code in the program, including code in blocks. Example of global scope:

```
const color = 'blue';

const returnSkyColor = () => {

  return color; // blue

};

console.log(returnSkyColor()); // blue
```

- Even though the `color` variable is defined outside of the block, it can be accessed in the function block, giving it global scope.
- In turn, `color` can be accessed within the `returnSkyColor` function block.

Block Scope

The next context we'll cover is *block scope*. When a variable is defined inside a block, it is only accessible to the code within the curly braces `{}`. We say that variable has *block scope* because it is *only* accessible to the lines of code within that block.

Variables that are declared with block scope are known as *local variables* because they are only available to the code that is part of the same block. Block scope works like this:

```
const logSkyColor = () => {
  let color = 'blue';
  console.log(color); // Prints "blue"
};
logSkyColor(); // Prints "blue"
console.log(color); // throws a ReferenceError
```

You'll notice:

- We define a function `logSkyColor()`.
- Within the function, the `color` variable is only available within the curly braces of the function.
- If we try to log the same variable outside the function, it throws a `ReferenceError`.

Scope Pollution

It may seem like a great idea to always make your variables accessible, but having too many global variables can cause problems in a program.

When you declare global variables, they go to the *global namespace*. The global namespace allows the variables to be accessible from anywhere in the program. These variables remain there until the program finishes which means our global namespace can fill up really quickly.

Scope pollution is when we have too many global variables that exist in the global namespace, or when we reuse variables across different scopes. Scope pollution makes it difficult to keep track of our different variables and sets us up for potential accidents. For example, globally scoped variables can collide with other variables that are more locally scoped, causing unexpected behavior in our code. Let's look at an example of scope pollution in practice so we know how to avoid it:

```
let num = 50;

const logNum = () => {

  num = 100; // Take note of this line of code

  console.log(num);

};

logNum(); // Prints 100

console.log(num); // Prints 100
```

You'll notice:

- We have a variable `num`.
- Inside the function body of `logNum()`, we want to declare a new variable but forgot to use the `let` keyword.
- When we call `logNum()`, `num` gets reassigned to `100`.
- The reassignment inside `logNum()` affects the global variable `num`.
- Even though the reassignment is allowed and we won't get an error, if we decided to use `num` later, we'll unknowingly use the new value of `num`.

While it's important to know what global scope is, it's best practice to not define variables in the global scope.

Practice Good Scoping

Given the challenges with global variables and scope pollution, we should follow best practices for scoping our variables as tightly as possible using block scope.

Tightly scoping your variables will greatly improve your code in several ways:

- It will make your code more legible since the blocks will organize your code into discrete sections.
- It makes your code more understandable since it clarifies which variables are associated with different parts of the program rather than having to keep track of them line after line!
- It's easier to maintain your code, since your code will be modular.
- It will save memory in your code because it will cease to exist after the block finishes running.

Here's another example of how block scope works, as defined within an `if` block:

```
const logSkyColor = () => {
  const dusk = true;
  let color = 'blue';
  if (dusk) {
    let color = 'pink';
    console.log(color); // Prints "pink"
  }
  console.log(color); // Prints "blue"
};
console.log(color); // throws a ReferenceError
```

- We create a variable `color` inside the `logSkyColor()` function.
- After the `if` statement, we define a new code block with the `{}` braces. Here we assign a new value to the variable `color` if the `if` statement is truthy.
- Within the `if` block, the `color` variable holds the value `'pink'`, though outside the `if` block, in the function body, the `color` variable holds the value `'blue'`.
- On the last line, we attempt to print the value of `color` outside both the `if` statement and the definition of `logSkyColor()`. This will throw a `ReferenceError` since `color` only exists within the scope of those two blocks — it is never defined in the global scope.
- While we use block scope, we still pollute our namespace by reusing the same variable name twice. A better practice would be to rename the variable inside the block.

Block scope is a powerful tool in JavaScript, since it allows us to define variables with precision, and not pollute the global namespace. If a variable does not need to exist outside a block— it shouldn't!