Program State

In this lesson, we will go over the concept of the state of the program and see its importance.

We'll cover the following

The previous program is an example of what is called *imperative programming*. In this paradigm, the programmer gives orders to the computer through a series of statements that modify the program state. Imperative programming focuses on describing how a program operates.

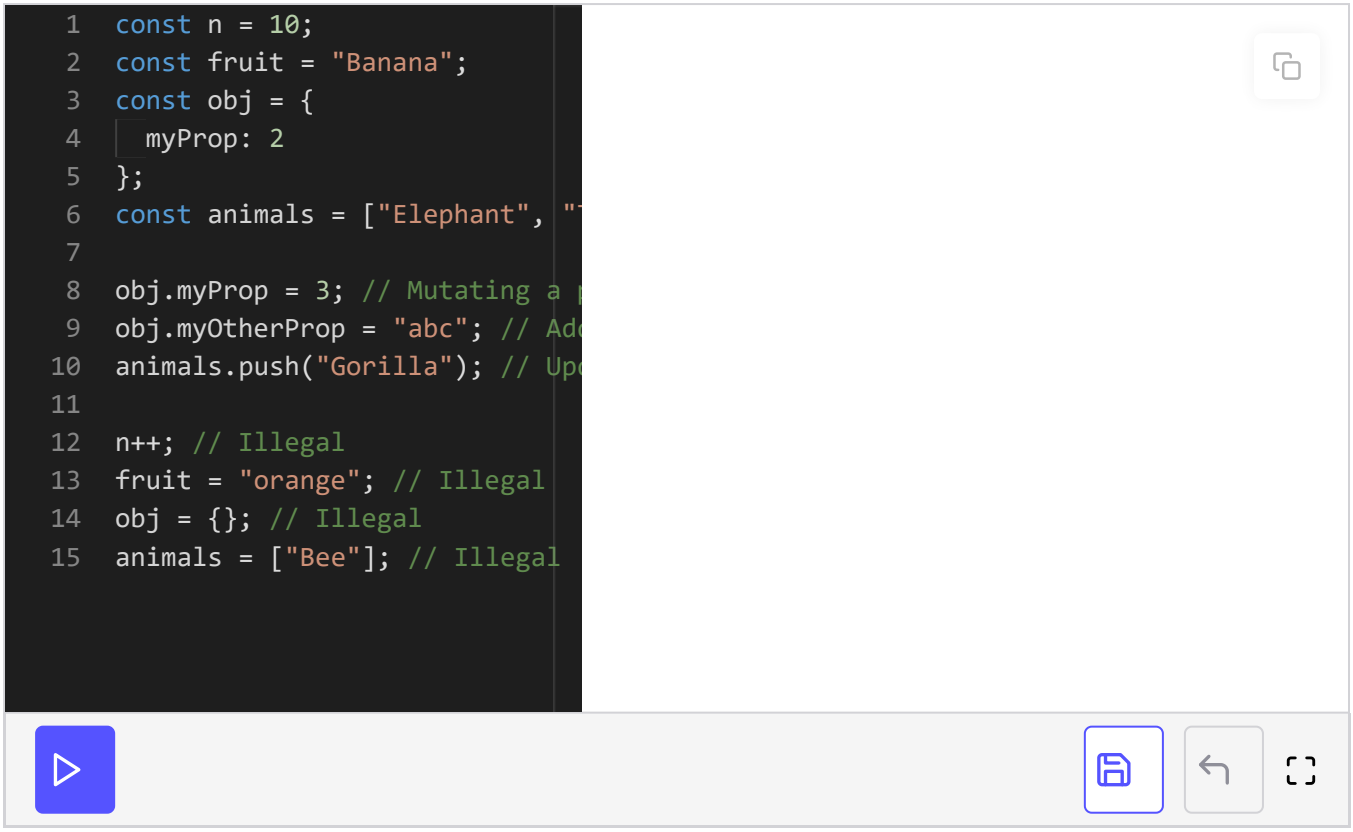
The concept of state is an important one. The *state* of a program is the value of its *global variables* (variables accessible everywhere in the code) at a given time. In our example, the values of `movieList`, `titles`, `nolanMovieCount`, `bestTitles`, `ratingSum` and `averageRating` form the state of the program. Any assignment to one of these variables is a state change, often called a *mutation*.

In imperative programming, the state can be modified anywhere in the source code. This is convenient, but can also lead to nasty bugs and maintenance headaches. As a program grows in size and complexity, it becomes easier for the programmer to mutate a part of the state by mistake and harder to monitor state modifications.

Limiting Mutations with `const` Variables

In order to decrease the risk of accidental state mutation, a first step is to favor `const` over `let` whenever applicable for variable declarations. A variable declared with the `const` keyword cannot be further reassigned. Array and object content can still be mutated, though. Check the following code for details.

```
const n = 10;
const fruit = "Banana";
const obj = {
  myProp: 2
};
const animals = ["Elephant", "
obj.myProp = 3; // Mutating a
obj.myOtherProp = "abc"; // Ad
animals.push("Gorilla"); // Up

n++; // Illegal
fruit = "orange"; // Illegal
obj = {}; // Illegal
animals = ["Bee"]; // Illegal
```



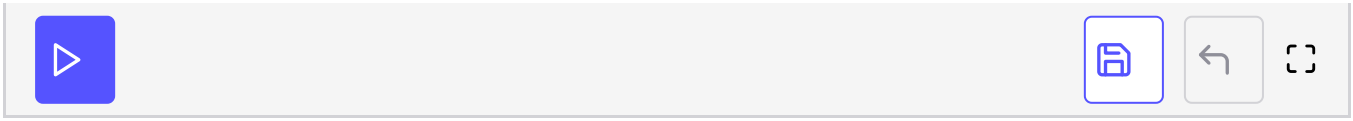
Splitting the Program into Functions

Another solution is to split the source code into subroutines called procedures or *functions*. This approach is called *procedural programming* and has the benefit of transforming some variables into *local variables*, which are only visible in the subroutine code.

Let's try to introduce some functions in our code.

```
// Get movie titles
const titles = () => {
  const titles = [];
  for (const movie of movieList) {
    titles.push(movie.title);
  }
  return titles;
};

const nolanMovieList = [];

// Get movies by Christopher Nolan
const nolanMovies = () => {
  for (const movie of movieList) {
    if (movie.director === "Christopher Nolan") {
      nolanMovieList.push(movie);
    }
  }
};

// Get titles of movies with an IMDB rating greater or equal to 7.5
const bestTitles = () => {
```

```
  const bestTitles = [];
  for (const movie of movieList) {
    if (movie.imdbRating >= 7.5) {
      bestTitles.push(movie.title);
    }
  }
  return bestTitles;
};

// Compute average rating of Christopher Nolan's movies
const averageNolanRating = () => {
  let ratingSum = 0;
  for (const movie of nolanMovieList) {
    ratingSum += movie.imdbRating;
  }
  return ratingSum / nolanMovieList.length;
};

console.log(titles());
nolanMovies();
console.log(nolanMovieList.length);
console.log(bestTitles());
console.log(averageNolanRating());
```



The state of our program is now limited to two variables: `movieList` and `nolanMovieList` (the latter being necessary in functions `nolanMovies()` and `averageNolanRating()`). The other variables are now local to the functions they are used into, which limits the possibility of an accidental state mutation.

Also, this version of the program is easier to understand than the previous one. Functions with appropriate names help describe a program's behavior. Comments are now less necessary than before.