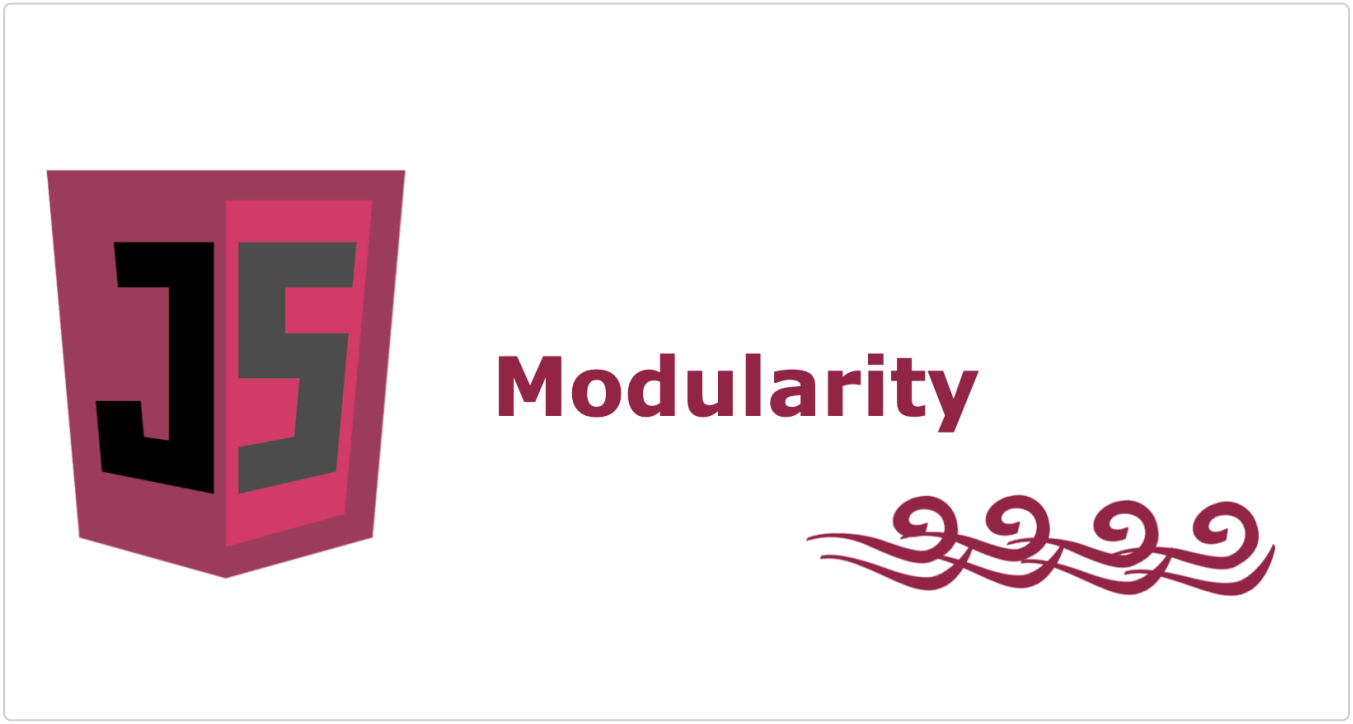
Modularity

We will cover modularity in this lesson.

WE'LL COVER THE FOLLOWING



Most curly-brace programming languages provide some kind of modularity. Besides the fact that you can encapsulate data and operations together in objects, you can also use concepts like namespaces to group objects together and use modules.

Modules have a well-defined public interface, a set of properties and operations that can be accessed from outside, and they may have private members that are intentionally concealed to be used internally; otherwise they would expose implementation details.

The JavaScript language does not have a concept of modules, but most developers use the module pattern that mimics the behavior of modules in

other programming languages.

Listing 8-21 demonstrates this concept.

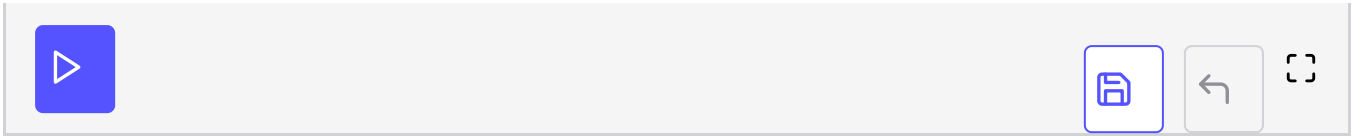
Listing 8-21: Exercise-08-21/index.html

```
<!DOCTYPE html>
<html>
<head>
  <title>Module pattern</title>
  <script>
    var CircleOps = (function () {
      // --- Private stuff
      var pi = 3.1415926;
      var radSquare = function (rad) {
        return rad * rad;
      }

      // --- Public stuff
      return {
        PI: pi,
        calcArea: function (rad) {
          return pi * radSquare(rad);
        },
        calcPerimeter: function (rad) {
          return pi * 2 * rad;
        }
      };
    })();

    var x = 1.5;
    CircleOps.PI = 2;
    var area = CircleOps.calcArea(x);
    var perim = CircleOps.calcPerimeter(x);
    var pi = CircleOps.PI;
    console.log("value of PI: " + pi);
    console.log("area: " + area);
    console.log("perimeter: " + perim);
  </script>
</head>
<body>
  Listing 8-21: View the console output
</body>
</html>
```

In this listing, `CircleOps` is an object with members like PI, `calcArea`, and `calcPerimeter`. The key of this pattern that makes `CircleOps` look and behave like a module is the form as the object is created:

```
var CircleOps = (function () {
  // --- Private stuff
  return {
    // --- Public stuff
  };
```

```
})();
```



This code defines a function expression and immediately executes the function. Every variable you define in the body of the function is scoped to the function and cannot be directly accessed. However, the object retrieved by the function contains properties and functions that are available to the external world.

The great thing about this pattern is that all properties and functions retrieved in the returned object are privileged members that can access the variables within this function.

This ensures that you can define the `pi` variable and the `calcSquare` function internally and use them through the returned object, but cannot access them directly through `CircleOps`.

In the *next lesson*, we see the BOM - browser object model.

See you there! :)