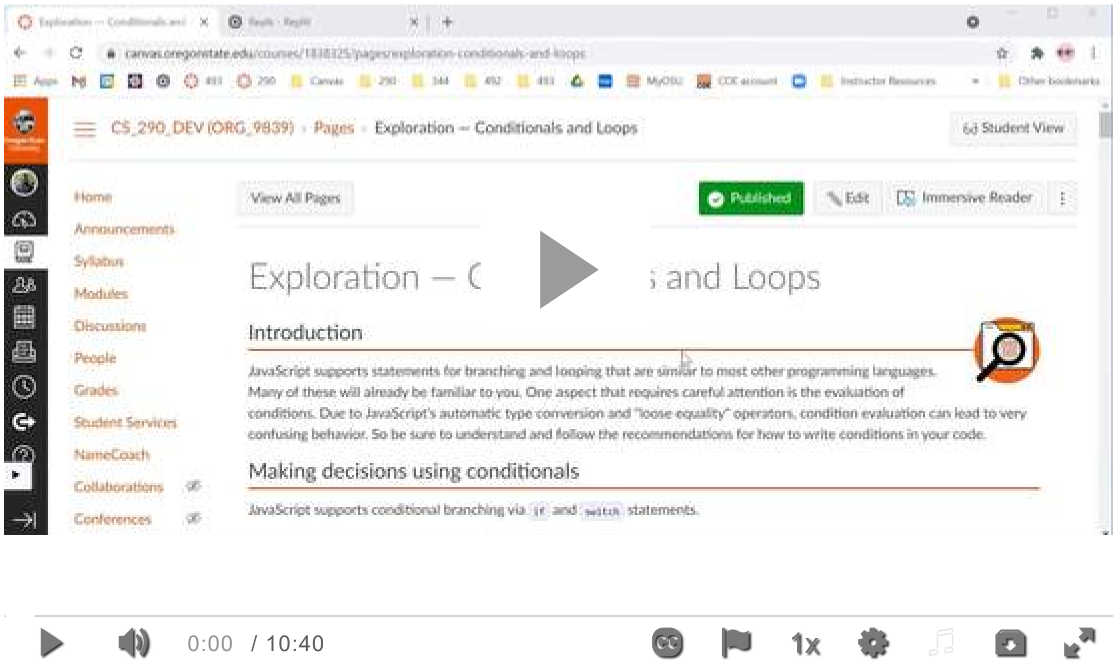
Exploration — Conditionals and Loops

Introduction

JavaScript supports statements for branching and looping that are similar to most other programming languages. Many of these will already be familiar to you. One aspect that requires careful attention is the evaluation of conditions. Due to JavaScript's automatic type conversion and "loose equality" operators, condition evaluation can lead to very confusing behavior. So be sure to understand and follow the recommendations for how to write conditions in your code.



Exploration — Conditionals and Loops

Introduction

JavaScript supports statements for branching and looping that are similar to most other programming languages. Many of these will already be familiar to you. One aspect that requires careful attention is the evaluation of conditions. Due to JavaScript's automatic type conversion and "loose equality" operators, condition evaluation can lead to very confusing behavior. So be sure to understand and follow the recommendations for how to write conditions in your code.

Making decisions using conditionals

JavaScript supports conditional branching via if and switch statements.

Making decisions using conditionals

JavaScript supports conditional branching via `if` and `switch` statements.

if statements

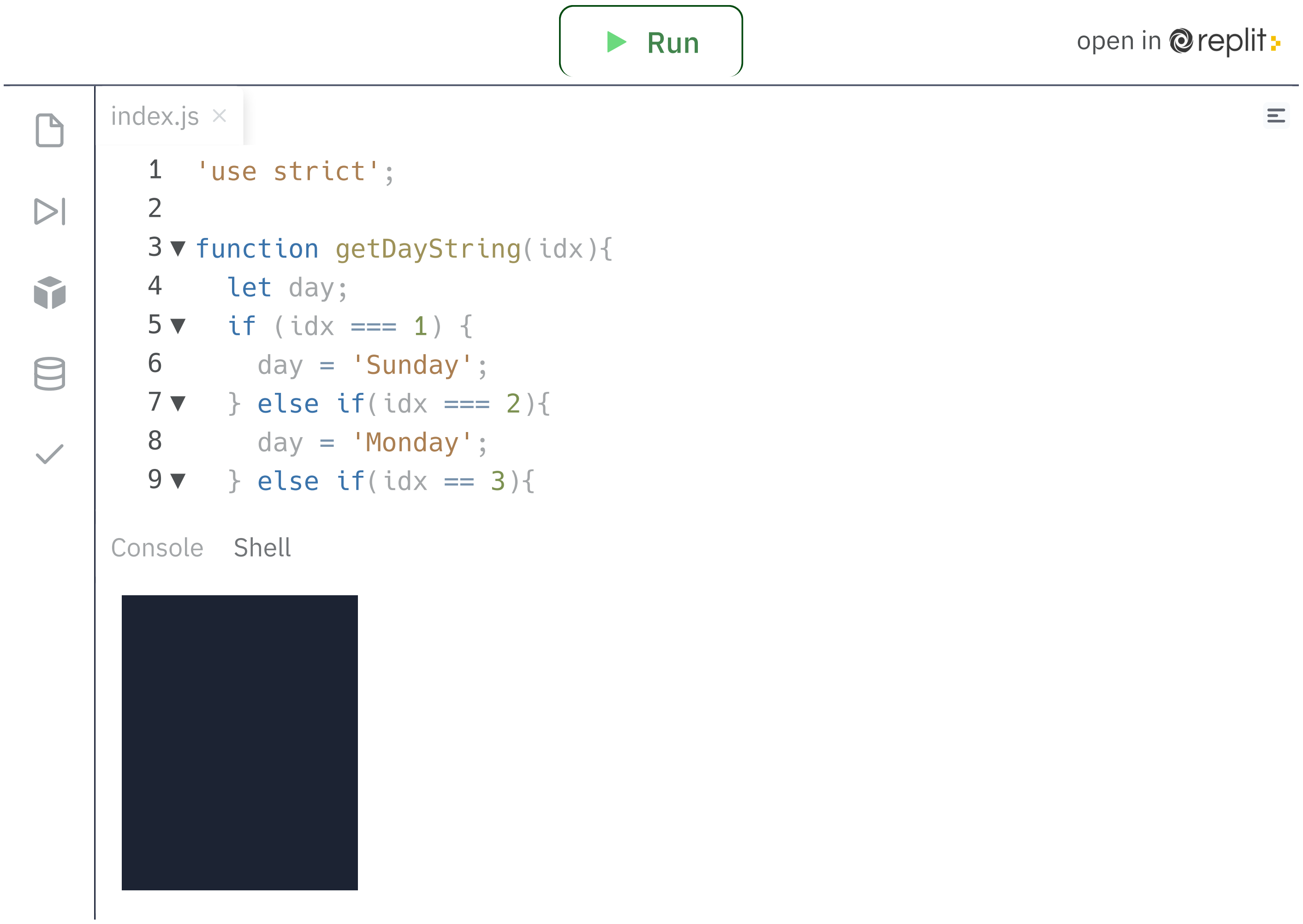
JavaScript **if statements** **(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/if...else)** are very similar to other programming languages. The general syntax is as follows:

```
if (expr1) {
    //...
} else if (expr2) {
    // ...
} else {
    // ...
}
```

We can use the basic `if` statement which will execute if the provided expression evaluates to `true`. We can also use the familiar `if-then-else` statement, where we have additional `else if` branches with expressions, and optionally an `else` branch without an expression which is executed if none of the expressions provided with the other branches evaluate to `true`.

Note: If you are coming from Python, note that conditional expressions in JavaScript must be enclosed within parentheses.

Example



```
'use strict';

function getDayString(idx){
  let day;
  if (idx === 1) {
    day = 'Sunday';
  } else if(idx === 2){
    day = 'Monday';
  } else if(idx == 3){
```

Exercise

In the above program, what is the value printed to the console by the statement `console.log(getDayString(10));`. Explain why that particular value is printed?

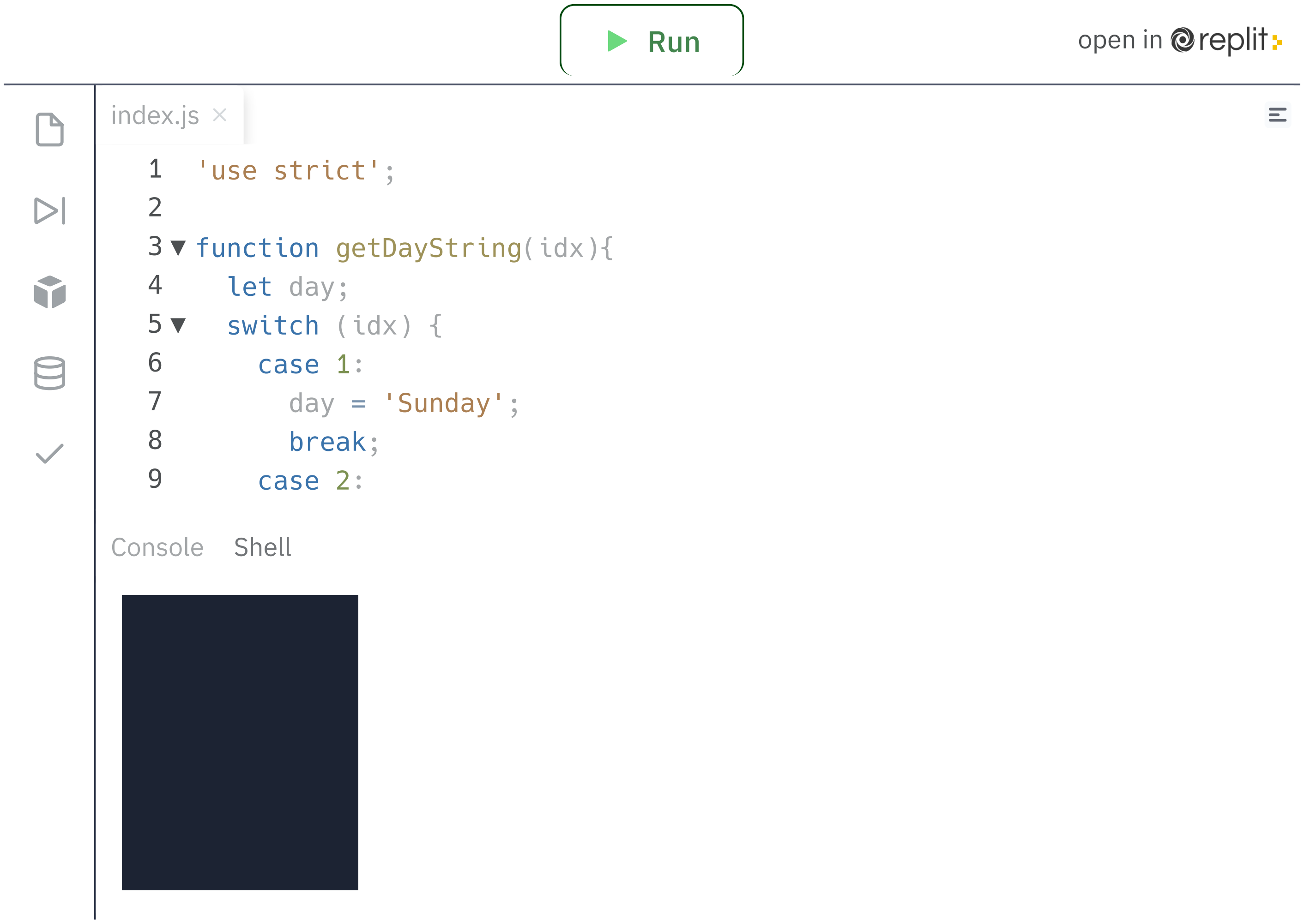
In the above program, note the parentheses around the Boolean expression. These are mandatory. The example also uses braces for the code block following the condition. If a code block contains only one statement, we can omit the braces. The style of putting a brace on the same line as the condition is called the "one true brace style." Not using this style can cause some issues if the code is pasted into a JavaScript console. In this course, we will use the "one true brace style" even if the code blocks contains one statement. We recommend that you do the same.

switch statements

In many programs, we have an `if-then-else` statement that compares the same variable to different values. In fact, the `if-then-else` statement in the previous program does exactly this. For such comparisons, JavaScript provides a **switch statement** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/switch) just like the `switch` statement in many other languages such as C, Java, C#, etc.

Example

Here is a program that uses a `switch` statement to code the same logic as is coded in the previous example using an `if-then-else` statement.



```
'use strict';

function getDayString(idx){
  let day;
  switch (idx) {
    case 1:
      day = 'Sunday';
      break;
    case 2:
```

Note that the expression is evaluated once. Then the matching `case` branch is executed. The code to be executed for the branch ends with a `break` statement. If none of the case branches match, the `default` branch is executed.

Conditional (or Ternary) Operator

The **Conditional Operator (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator)**, also called the Ternary Operator, is a very succinct way of coding a simple decision. The general format of the conditional operator is as follows:

`condition ? expression1 : expression2`

- The condition is evaluated.
- If it is true, then `expression1` is evaluated and its value is returned.
- Otherwise, `expression2` is evaluated and its value is returned.

A common usage of the conditional operator is to set the value of a variable to one of two expressions based on a condition. For example:

`let max = x > y ? x : y;`

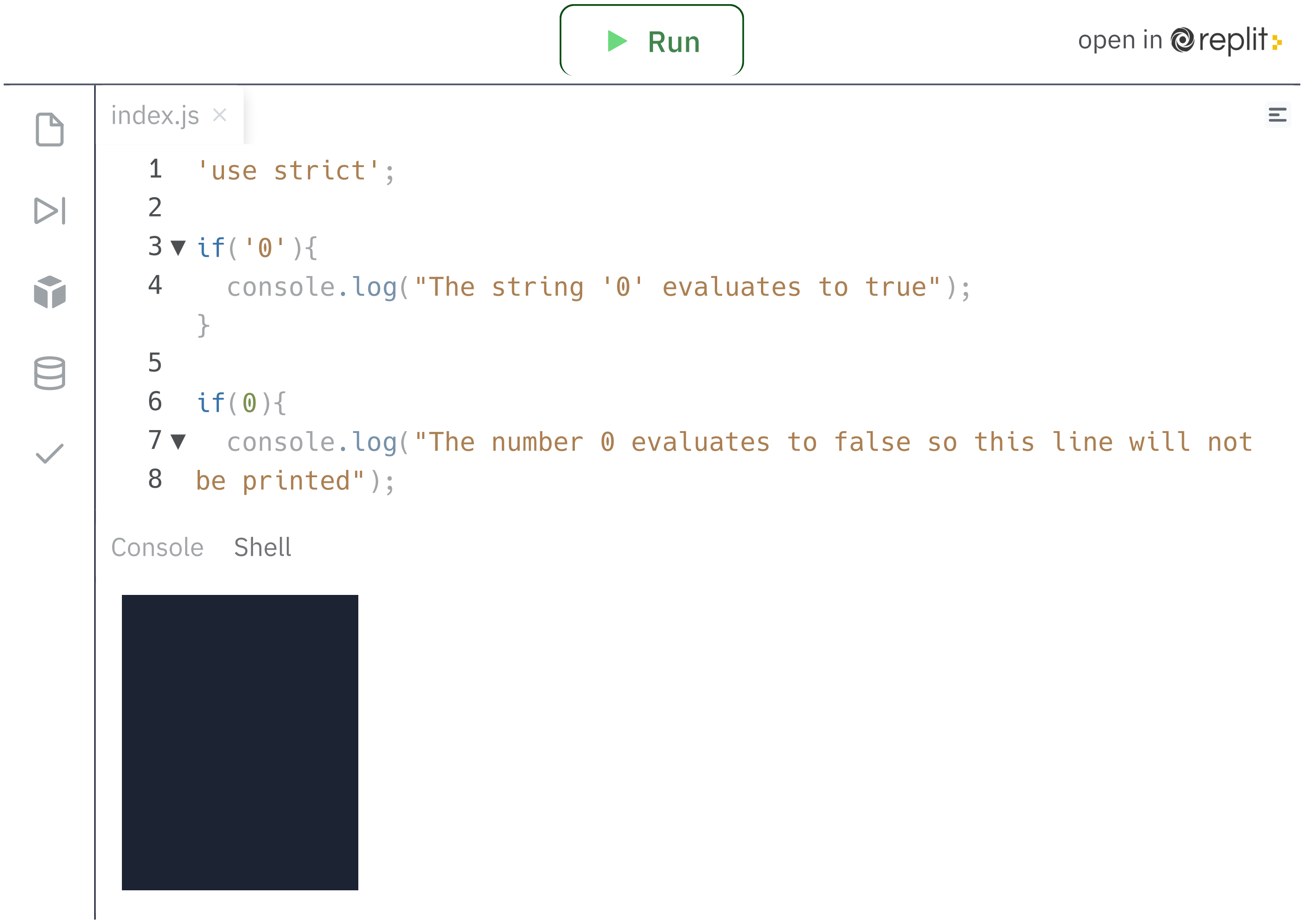
This is a lot more succinct that using an if-else statement to implement the same functionality:

```
let max;
if( x > y){
  max = x;
} else{
  max = y;
}
```

Conditions in JavaScript

JavaScript allows a condition to contain values of any type. A non-Boolean value is automatically converted to a Boolean value as needed. For example, the values `0`, `NaN`, `null`, `undefined` and the empty string are all converted to `false`. Instead of relying on the automatic conversion, we should code our conditions to explicitly produce either `true` or `false`.

Example



```
'use strict';

if('0'){
  console.log("The string '0' evaluates to true");
}

if(0){
  console.log("The number 0 evaluates to false so this line will not
be printed");
```

Equality Testing

JavaScript provides two sets of equality operators.

Strict Equality Operators

The two "strict equality" operators are

- `===` to check whether the two operands are strictly equal,
- `!==` to check whether the two operands are not strictly equal.

With strict equality operators:

- Operands of different type are never equal. For example:
  - `42 === '42' // false`
- Boolean values are equal when their values are the same.
- Numbers operands are equal when their values are equal. For example:
  - `42 === 42.0 // true`
  - `42 === 42.1 // false`
  - However, `NaN` is not equal to `NaN` . To check if a variable has the value `NaN` , use `Number.isNaN(x)` .

- String operands are equal when their values are equal.
- A `null` value is equal to another `null` value and an `undefined` value is equal to an `undefined` value. But a `null` value and an `undefined` value are not equal to each other.
- Objects are equal only when both the operands refer to the same object.

Example: Strict Equality of Objects

Consider the following two objects:

```
const student1 = { name: 'Harvey', age: 23 };
const student2 = { name: 'Harvey', age: 23 };
```

Since `student1` and `student2` refer to different objects, these two objects are not strictly equal, even though both the objects have the same properties and these properties have the same value in both these objects. Therefore:

- `student1 === student1 // true`
- `student2 === student2 // true`
- `student1 === student2 // false`

Example: Strict Equality of Arrays

Since arrays are objects, two arrays are strictly equal only when they refer to the same array.

Consider the following two arrays:

```
const majors1 = ['CS', 'Math'];
const majors2 = ['CS', 'Math'];
```

Since `majors1` and `majors2` refer to different objects, these two arrays are not strictly equal, even though both the arrays have the same elements in the same order. Therefore:

- `majors1 === majors1 // true`
- `majors2 === majors2 // true`
- `majors1 === majors2 // false`

Loose Equality Operators

The "loose equality" operators are

- `==` to check whether the two operands are loosely equal,
- `!=` to check whether the two operands are not loosely equal.

These operators can compare operands of different type by automatically converting them to a common type. For example, `42 == '42'` evaluates to `true`. But use of these operators can lead to very confusing code. For example, an empty array is loosely equal to an empty string, i.e., `[] == ''` evaluates to `true`!

We must follow the **Golden Rule 3** "Know your types and avoid automatic type conversion" and **do not** use the loose equality operators.

Comparison Testing

JavaScript support the standard comparison operators, `<`, `<=`, `>` and `>=`. When using these operators make sure that either both the operands are numbers or both the operands are strings. Otherwise, JavaScript will automatically convert the value. If needed, convert the operand explicitly. Also, never use these operators on values of any other type.

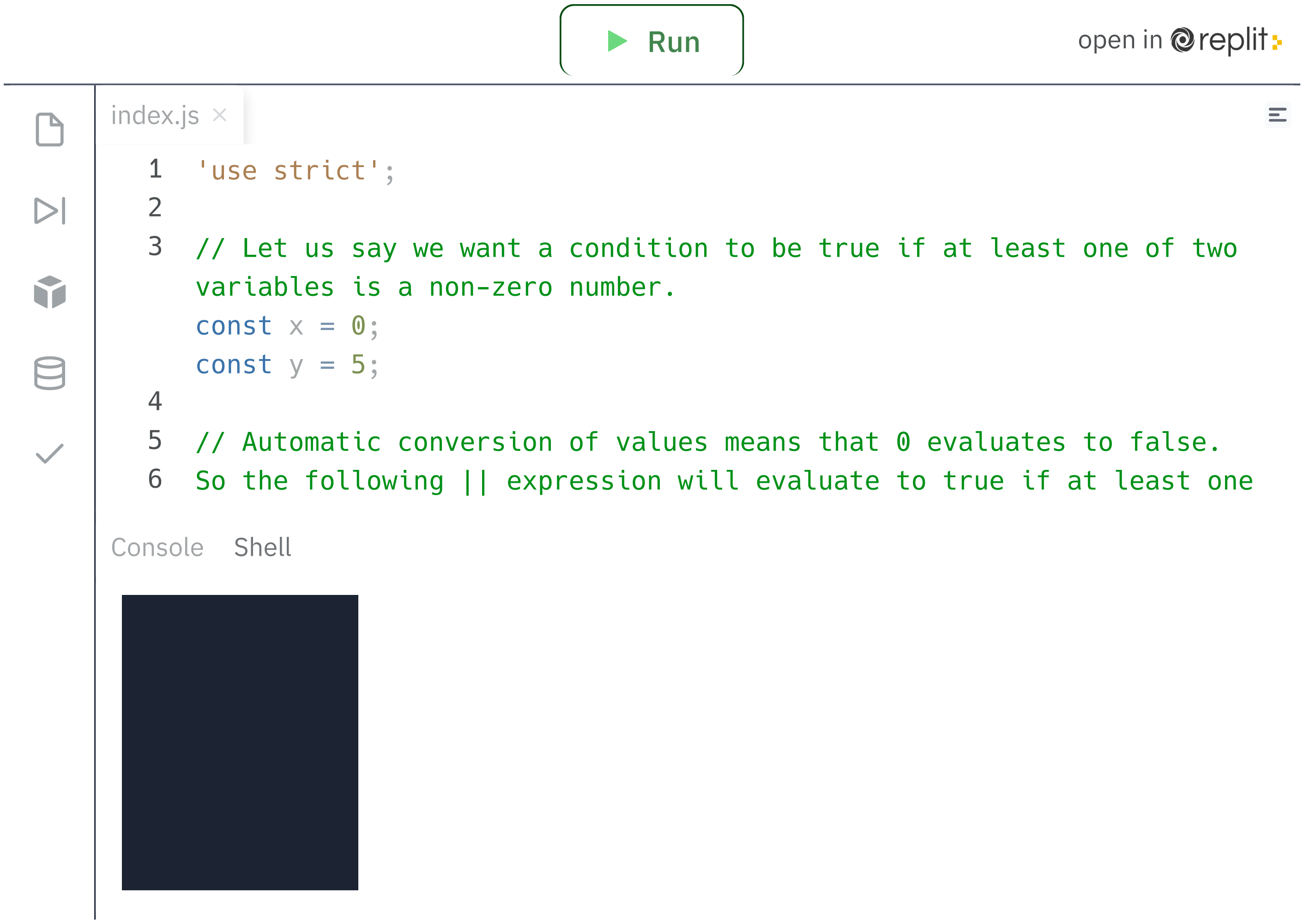
Boolean Operators

We can combine Boolean values using the usual operators:

- `&&` AND
- `||` OR
- `!` NOT

If the operands to these operators are not Boolean values, JavaScript will automatically convert them to Boolean values. Once more we emphasize that instead of allowing JavaScript to do automatic type conversion, use Boolean values as operands to these operators.

Example



```
'use strict';

// Let us say we want a condition to be true if at least one of two
variables is a non-zero number.
const x = 0;
const y = 5;

// Automatic conversion of values means that 0 evaluates to false.
So the following || expression will evaluate to true if at least one
```

Loops

Looping with **while**

The **while statement** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration#while_statement) in JavaScript is pretty similar to the while statement you have likely encountered in other languages. The basic syntax is:

```
while(expr) {
    Statement or statements to execute
}
```

The expression `expr` is evaluated. If it evaluates to `true` then the body of the `while` loop is executed; otherwise the loop ends. The `while` loop will keep executing the body as long as `expr` evaluates to `true`. If the body of the while loop has only one statement, braces aren't required.

Looping with **do while**

The **do while statement** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration#do...while_statement) is similar to the `while` statement with one crucial difference: the expression is evaluated after the body has been executed. The basic syntax is:

```
do {
    statement or statements to execute
} while(expr)
```

This means that a `do` loop will execute at least once. This will happen even if `expr` is false when the `do` loop is to be executed the first time.

Looping with **for** statements

The `for` loop is used to execute a block of code a certain number of times. A common use is for iterating over elements. JavaScript provides 3 variants of the `for` loop.

The **for** loop

Similar to many other languages, JavaScript supports the **for statement (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration#for_statement)** which has the following general form

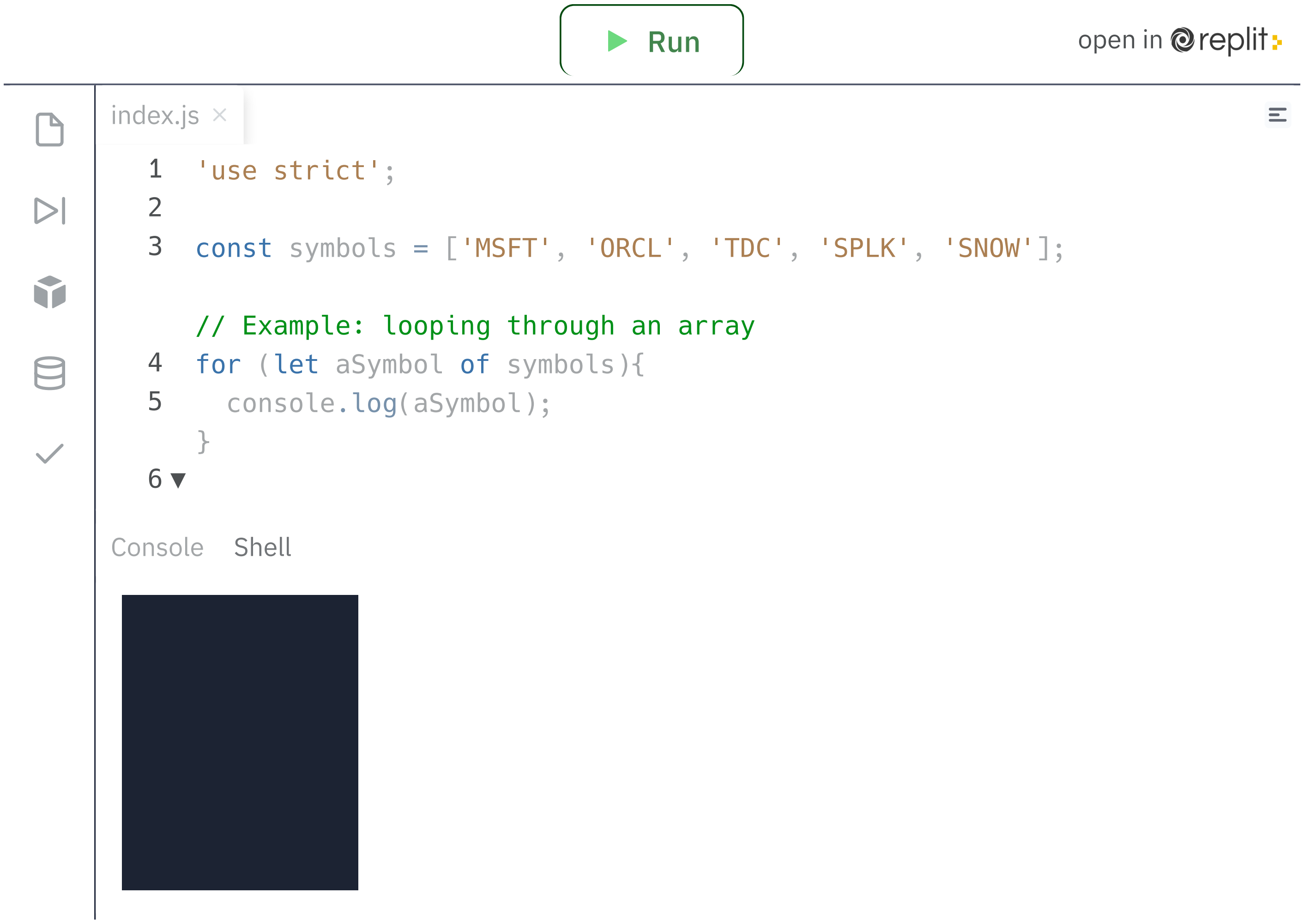
```
for(initialization statement; loop condition; repeating statement){
    Statement or statements to execute
}
```

The initialization statement is executed once. The loop condition is evaluated for each execution of the loop, including the first execution. If the condition evaluates to true, the body of the `for` loop is executed. After the body has been executed, the repeating statement is executed. Next the loop condition is evaluated again. The `for` loop continues until the loop condition evaluates to `false`.

The **for of** loop

The **for of statement (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration#for...of_statement)** is used to iterate over the elements of an **iterable** value, such as, a string or an array.

Example

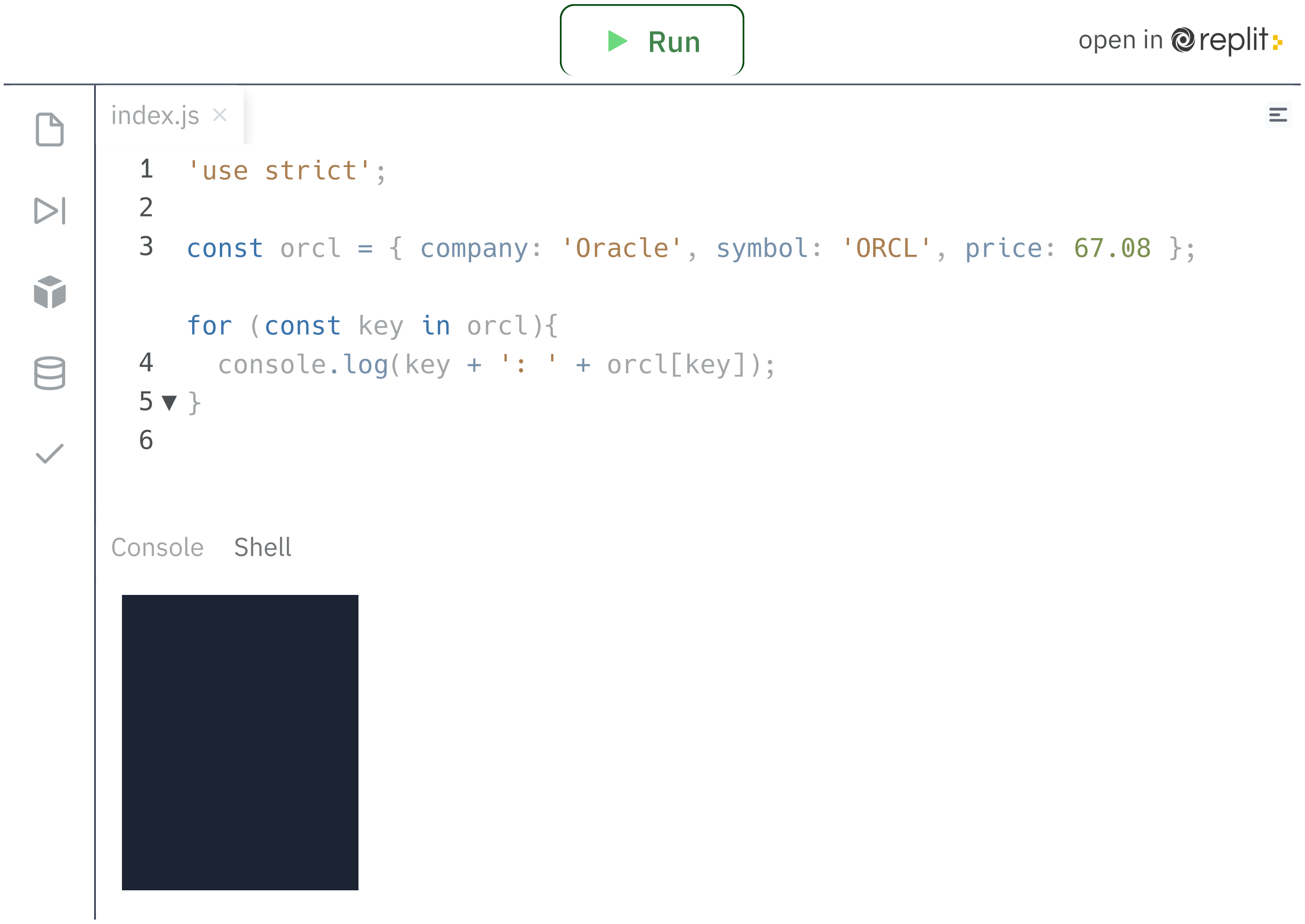


```javascript
'use strict';

const symbols = ['MSFT', 'ORCL', 'TDC', 'SPLK', 'SNOW'];

// Example: looping through an array
for (let aSymbol of symbols){
  console.log(aSymbol);
}
```

The **for in**

The **for in statement** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration#for...in_statement) is used to iterate over the properties of an object. We shouldn't assume that the loop will iterate over the properties in any particular order.

Example



```
'use strict';

const orcl = { company: 'Oracle', symbol: 'ORCL', price: 67.08 };

for (const key in orcl){
  console.log(key + ': ' + orcl[key]);
}

```

Exercise: Iterating Over the Properties of an Object Using Object.keys()

We just saw how we can iterate over the properties of an object using a `for in` statement. But another way to iterate over the properties of an object is to use a `for of` statement to iterate over the array returned by `Object.keys()`. Your task: print the same information being printed by the above example, but instead of a `for in` statement use a `for of` statement and `Object.keys()`.

Breaking out of a loop using break

The **break statement** **(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration#break_statement)** immediately breaks out of a loop even when the loop's condition is true.

Skipping to the next iteration of a loop using continue

The **continue statement** **(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration#continue_statement)** causes a loop to skip ahead to the next execution without breaking out of the loop. Any statements in the body of the loop that come after the `continue` statement will be skipped. This statement is used when we want to skip some statements in the body of the loop when a certain condition holds.

Summary

In this exploration, we look at the various statements provide by JavaScript for branching and looping. Most of the statements should already be familiar to someone who has programmed in other languages.

Additional Resources

Here are some references to learn more about the topics we discussed in this exploration.

- On the MDN website `if` and `switch` statements are discussed on the page on **Control flow and error handling** **(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling)** .
- Expressions and operators are discussed on MDN's page on **Expressions and Operators** **(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Expressions_and_Operators)**
- MDN's page on **Loops and Iteration** **(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration)** has comprehensive explanations and many examples for the the looping constructs.