6.1 Syntax and variables

Background

In 1995, Brendan Eich created JavaScript so the Netscape Navigator browser could dynamically respond to user events. Ex: The web page's content could change when the user clicked a button or hovered over an image.

JavaScript was standardized by Ecma International in 1997 and called ***ECMAScript***. Ecma International continues to improve ECMAScript, releasing a new version each year. JavaScript is an implementation of the ECMAScript specification.

Today, JavaScript is one of the most popular programming languages. JavaScript is supported by every major web browser and makes web applications like Gmail and Google Maps possible. JavaScript is also popular outside the web browser. Ex: Node.js, which runs JavaScript, is a popular technology for creating server-side web applications.

JavaScript is executed by an interpreter. An ***interpreter*** executes programming statements without first compiling the statements into machine language. Modern JavaScript interpreters (also called ***JavaScript engines***) use ***just-in-time (JIT) compilation*** to compile the JavaScript code at execution time into another format that can be executed quickly.

ECMAScript name

The name "ECMAScript" was a compromise between Netscape, Microsoft, and other organizations involved in the standardization of JavaScript. Brendan Eich once commented that "ECMAScript was always an unwanted trade name that sounds like a skin disease." Despite ECMAScript's similarity to eczema (a group of related skin diseases), the name has stuck.

Quote source: Archived email (Oct 3, 2006)

PARTICIPATION ACTIVITY 6.1.1: JavaScript background.

1) ECMAScript and JavaScript are the same thing.

- ○ True
- ○ False

2) JavaScript is only used for programs that run in a web browser.

- ○ True
- ○ False

3) All browsers must use the same JavaScript engine.

- ○ True
- ○ False

4) JavaScript and Java are the same programming language.

- ○ True
- ○ False

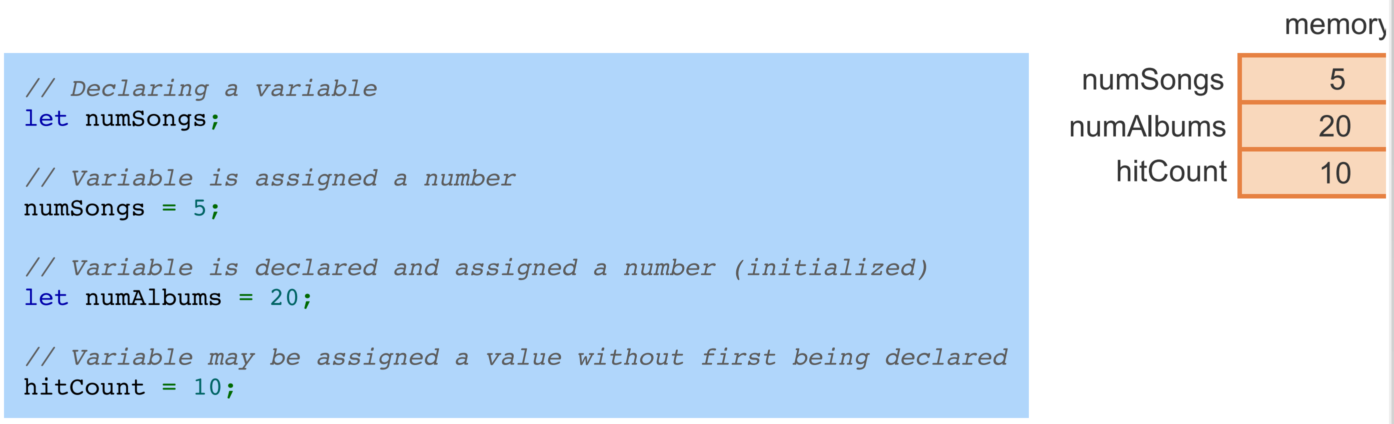
Variables

A ***variable*** is a named container that stores a value in memory. A ***variable declaration*** is a statement that declares a new variable with the keyword ***let*** followed by the variable name. Ex: `let score` declares a variable named score.

A variable may be assigned a value after being declared. An ***assignment*** assigns a variable with a value, like `score = 2`. A variable may also be assigned a value on the same line when the variable is declared, which is called ***initializing*** the variable. Ex: `let maxValue = 5;` initializes `maxValue` to 5.

A variable may be assigned a value without first declaring the variable, but good practice is to always declare a variable before assigning a value to the variable.

PARTICIPATION ACTIVITY 6.1.2: Declaring variables and assigning values.



```
// Declaring a variable
let numSongs;

// Variable is assigned a number
numSongs = 5;

// Variable is declared and assigned a number (initialized)
let numAlbums = 20;

// Variable may be assigned a value without first being declared
hitCount = 10;
```

Animation content:

The following code is displayed:
// Declaring a variable
let numSongs;

// Variable is assigned a number
numSongs = 5;

// Variable is declared and assigned a number (initialized)
let numAlbums = 20;

// Variable may be assigned a value without first being declared
hitCount = 10;

The first step runs: let numSongs;. The variable numSongs gets allocated a block of memory.
The second step runs: numSongs = 5;. The block of memory that was allocated to numSongs now stores the value 5.
The third step runs: let numAlbums = 20. The variable numAlbums gets allocated memory and assigned to 20 at the same time.
The fourth step runs: hitCount = 10. The allocation and assignment of memory is shown just like the third step but with variable hitCount and value 10.

Animation captions:

1. The numSongs variable is declared with the "let" keyword.
2. numSongs is assigned with 5.

3. numAlbums is initialized with 20.
4. When hitCount is assigned with 10, hitCount is implicitly declared. Good practice is to explicitly declare all variables with "let".

A name created for an item like a variable is called an ***identifier***. JavaScript imposes the following rules for identifiers:

- An identifier can be any combination of letters, digits, underscores, or $.
- An identifier may not start with a digit.
- An identifier may not be a reserved word like `let`, `function`, or `while`.

A JavaScript coding convention is to name JavaScript variables with camel casing, where the identifier starts with a lowercase letter, and subsequent words begin with a capital letter. Ex: `lastPrice` is preferred over `LastPrice` or `last_price`.

A ***constant*** is an initialized variable whose value cannot change. A JavaScript constant is declared with the ***const*** keyword. Ex: `const slicesPerPizza = 8;` creates a constant `slicesPerPizza` that is always 8.

var keyword

A variable may also be declared with the `var` keyword, which is covered elsewhere in this material.

PARTICIPATION ACTIVITY 6.1.3: Declaring and naming variables.

1) Which statement declares the variable `sum` without assigning a value to `sum`?

- ○ `sum;`
- ○ `let sum;`
- ○ `sum = 0;`

2) Which identifier is illegally named?

- ○ `star_destroyer`
- ○ `ADDRESS`
- ○ `$save`
- ○ `9to5`

3) Which variable is named with the preferred JavaScript naming conventions?

- ○ `total_points`
- ○ `$totalPoints`
- ○ `totalPoints`

4) Which statement declares a constant for Earth's gravity?

- ○ `let earthGravity = 9.8;`
- ○ `const earthGravity = 9.8;`
- ○ `const earthGravity;`

5) Which code segment contains an error?

○
```
let numLives = 9;
numLives = 8;
```

○
```
const numLives = 9;
let livesLeft =
numLives;
```

○
```
numLives = 9;
let numLives;
```

Data types

Variables are not explicitly assigned a data type. JavaScript uses ***dynamic typing***, which determines a variable's type at run-time. A variable can be assigned a value of one type and reassigned a value of another type. Ex: `x = 5; x = "test";` assigns `x` with a number type, then a string type.

Table 6.1.1: Example JavaScript data types.

Data type	Description	Example
string	Group of characters delimited with 'single' or "double" quotes	`let name = "Naya";` `let quote = 'He asked, "Shall we play a game?"';`
number	Numbers with or without decimal places	`let highScore = 950;` `let pi = 3.14;`
boolean	true or false	`let hungry = true;` `let thirsty = false;`
array	List of items	`let teams = ["Broncos", "Cowboys", "49ers"];`
object	Collection of property and value pairs	`let movie = { title:"Sing", rating:"PG" };`
undefined	Variable that has not been assigned a value	`let message;`
null	Intentionally absent of any object value	`let book = null;`

PARTICIPATION ACTIVITY 6.1.4: Variable data types.

1) What is the data type of `population`?

```
let population = 650000;
```

- ○ int
- ○ string
- ○ number

2) What is the data type of `z`?

```
let z;
```

- ○ undefined
- ○ string
- ○ number

3) What is the data type of `x`?

```
y = false;
x = y;
```

- ○ string
- ○ boolean
- ○ number

4) What is syntactically wrong with the following code?

```
name = 'Danny O'Sullivan';
```

- ○ `name` is assigned a value without being declared first.
- ○ Variables may not be assigned strings delimited with single quotes.
- ○ The single quotation mark in O'Sullivan is an error.

Comments and semicolons

A ***comment*** is any text intended for humans that is ignored by the JavaScript interpreter. JavaScript uses the `//` and `/* */` operators to produce comments in code.

Figure 6.1.1: Comments.

```
// Single line
comment

/* Multi-line
   comment
*/
```

JavaScript does not require that statements be terminated with a semicolon. Only when two statements reside on the same line must a semicolon separate the two statements. *Good practice is to avoid placing two statements on the same line.* Some developers prefer to use semicolons at the end of statements, and others do not. *Good practice is to consistently use semicolons or not throughout the code.*

Figure 6.1.2: Using semicolons.

```
let totalPoints = 10;

// No semicolon is required
let totalLives = 3

// Two statements on the same line require a
semicolon
totalPoints = 5; totalLives = 2
```

PARTICIPATION ACTIVITY | 6.1.5: Detect the error.

Indicate if the statements contain an error or not.

1)
```
/* a is assigned 2
a = 2;
```
- ○ Error
- ○ No error

2)
```
3.12 = pi;
```
- ○ Error
- ○ No error

3)
```
x = 10; let y = 20;
```
- ○ Error
- ○ No error

Input and output

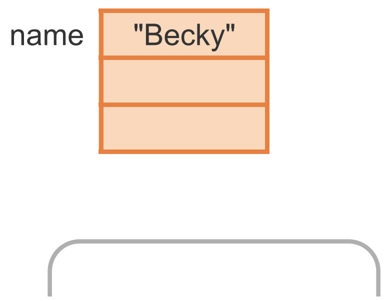
A JavaScript program may obtain text input from the user with the `prompt()` function. The ***prompt()*** function prompts the user with a dialog box that allows the user to type a single line of text and press OK or Cancel. The `prompt()` function returns the string the user typed or `null` if the user pressed Cancel.

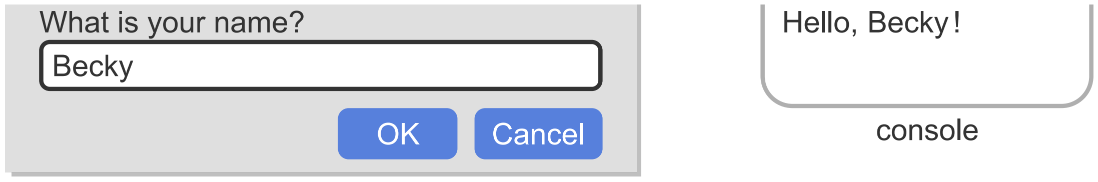
Output may be produced using the function ***console.log()***, which displays text or numbers in the console. The ***console*** is a location where text output is displayed. Web browsers have a console (accessible from the browser's development tools) that displays output from code the browser executes. This chapter's activities display the console output in the web page.

PARTICIPATION ACTIVITY 6.1.6: Prompting for input and displaying output.

```
// Display the prompt dialog box
let name = prompt("What is your name?");

// Output to the console
console.log("Hello, " + name + "!");
```





Animation content:

The following code is displayed:
// Display the prompt dialog box
let name = prompt("What is your name?");

// Output to the console
console.log("Hello, " + name + "!");

The first step of the animation runs: let name = prompt("What is your name?");. This opens a dialog box that displays What is your name? and a place to input text below.
The second step shows the string Becky being input for the dialog box.
The third step shows the variable name getting allocated memory and getting assigned the value Becky.
The fourth step runs: console.log("Hello, " + name + "!"); and displays Hello, Becky! in the console

Animation captions:

1. The prompt() function displays a dialog box with the given prompt text.
2. The user types her name and presses the OK button.
3. The name variable is assigned with the entered text.
4. console.log() outputs "Hello, ", then the value of the name variable, then "!" to the console.

PARTICIPATION ACTIVITY 6.1.7: prompt() and console.log().

1) Write the code to prompt the user with the `question` variable and retrieve the user's age.

```
question = "How old are
you?";
age =
                    ;
```

Check Show answer

2) Write the code to display "You are X", where X is the value of the `age` variable.

```
age = 21;
console.log(
);
```

Check Show answer

PARTICIPATION ACTIVITY 6.1.8: JavaScript practice.

The JavaScript code below initializes the variable `tvShow` to a popular TV show. Then, an if-else statement displays a message in the console if `tvShow` is `null`, otherwise the value of `tvShow` is displayed in the console. Change the code to prompt the user for the user's favorite TV show. Then, display "____ is your favorite TV show!" in the console. Press "Run JavaScript" to run your code.

Note: The console will display an error message if the JavaScript interpreter detects a syntax error. A ***syntax error*** is the incorrect typing of a programming statement. Ex: Forgetting to place "quotes" around a string value is a syntax error.

```
// Call prompt() to get the user's favorite TV show
let tvShow = "The Office";

if (tvShow === null) {
   console.log("You did not enter a TV show.");
}
else {
   console.log(tvShow);
}

```

Run JavaScript Reset code

Your console output

```
The Office
```

▶ View solution

CHALLENGE ACTIVITY 6.1.1: prompt() and console.log().

550544.4142762.qx3zqy7

Start

Write a statement that displays: My name is Bob

```

/* Your solution goes here */

```

1 2 3

Check Next

Exploring further:

- A Brief History of JavaScript from auth0
- JavaScript Lexical Grammar from MDN

6.2 Arithmetic

Arithmetic operators

An ***expression*** is a combination of items like variables, numbers, operators, and parentheses, that evaluates to a value like 2 * (x + 1). Expressions are commonly used on the right side of an assignment statement, as in `y = 2 * (x + 1)`.

An ***arithmetic operator*** is used in an expression to perform an arithmetic computation. Ex: The arithmetic operator for addition is `+`. JavaScript arithmetic operators are summarized in the table below.

Table 6.2.1: JavaScript arithmetic operators.

Arithmetic operator	Description	Example

`+`	Add	`// x = 3`
		`x = 1 + 2;`

`-`	Subtract	`// x = 1`
		`x = 2 - 1;`

`*`	Multiply	`// x = 6`
		`x = 2 * 3;`

`/`	Divide	`// x = 0.5`
		`x = 1 / 2;`

`%`	Modulus (remainder)	`// x = 0`
		`x = 4 % 2;`

`**`	Exponentiation	`// x = 2 * 2 * 2 = 8`
		`x = 2 ** 3;`

`++`	Increment	`// Same as x = x + 1`
		`x++;`

`--`	Decrement	`// Same as x = x - 1`
		`x--;`

Expressions are computed using the same rules as basic arithmetic. Expressions in parentheses `()` have highest precedence, followed by exponentiation (`**`). Multiplication (`*`), division (`/`), and modulus (`%`) have precedence over addition (`+`) and subtraction (`-`). Ex: The expression `7 + 3 * 2` = 7 + 6 = 13 because `*` has precedence over `+`, but `(7 + 3) * 2` = 10 * 2 = 20 because `()` has precedence over `*`.

PARTICIPATION ACTIVITY | 6.2.1: Arithmetic practice.

What is `points` at the end of each code segment?

1)
```
points = 3 + 5 * 2;
```

Check Show answer

2)
```
points = 4;
points = (3 + points) %
5;
```

Check Show answer

3)
```
scale = 5;
points = 3 ** 2 * scale;
```

Check Show answer

4)
```
points = 10;
points--;
```

Check Show answer

5)
```
points = 6;
points++;
```

Check Show answer

Compound assignment operators

A ***compound assignment operator*** combines an assignment statement with an arithmetic operation. Common JavaScript compound assignment operators are summarized in the table below.

Table 6.2.2: Compound assignment operators.

Assignment operator	Description	Example

`+=`	Add to	`// Same as x = x + 2`
		`x += 2;`

`-=`	Subtract from	`// Same as x = x - 2`
		`x -= 2;`

`*=`	Multiply by	`// Same as x = x * 3`
		`x *= 3;`

`/=`	Divide by	`// Same as x = x / 3`
		`x /= 3;`

`%=`	Mod by	`// Same as x = x % 4`
		`x %= 4;`

PARTICIPATION ACTIVITY | 6.2.2: Practice with compound assignment operators.

1) What compound assignment operator makes `points` become 2.5?

```
points = 5;
points ___ 2;
```

Check Show answer

2) What is `points`?

```
points = 2;
points *= 3 + 1;
```

Check Show answer

3) What is `points`?

```
points = 4;
points %= 2;
```

Check Show answer

Arithmetic with numbers and strings

The + operator is also the string concatenation operator. ***String concatenation*** appends one string after the end of another string, forming a single string. Ex: `"back" + "pack"` is `"backpack"`.

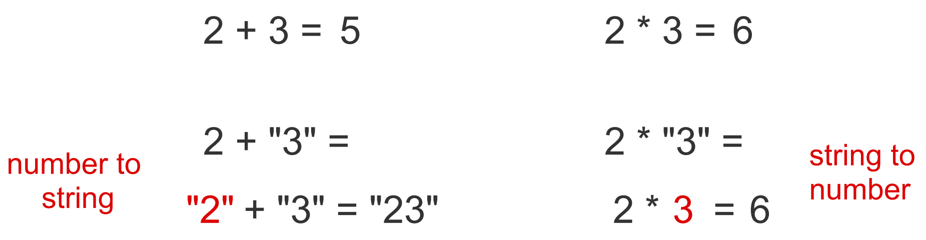
The JavaScript interpreter determines if + means "add" or "concatenate" based on the operands on either side of the operator. An ***operand*** is the value or values that an operator works on, like the number 2 or variable x.

- If both operands are numbers, + performs addition. Ex: 2 + 3 = 5.
- If both operands are strings, + performs string concatenation. Ex: "2" + "3" = "23".
- If one operand is a number and the other a string, + performs string concatenation. The

number is converted into a string, and the two strings are concatenated into a single string. Ex: "2" + 3 = "2" + "3" = "23".

For all other arithmetic operators, combining a number and a string in an arithmetic expression converts the string operand to a number and then performs the arithmetic operation. Ex: "2" * 3 = 2 * 3 = 6.

PARTICIPATION ACTIVITY 6.2.3: Type conversion in arithmetic operations.



Animation content:

Step 1 shows an example of a number + number expression:
2 + 3 = 5.
Step 2 shows an example of a number + string expression:
2 + "3" =
"2" + "3" = "23".
The number 2 gets automatically converted to a string then the string 3 is concatenated to the converted string 2.
Step 3 shows an example of the number * number expression:
2 * 3 = 6.
Step 4 shows an example of a number * string expression:
2 * "3" =
2 * 3 = 6.
The string 3 gets automatically converted to a number and multiplied with the number 2.

Animation captions:

1. number + number = number

2. number + string = string
3. number * number = number
4. number * string = number

The JavaScript functions ***parseInt()*** and ***parseFloat()*** convert strings into numbers. Ex: `parseInt("5") + 2 = 5 + 2 = 7`, and `parseFloat("2.4") + 6 = 2.4 + 6 = 8.4`.

If `parseInt()` or `parseFloat()` are given a non-number to parse, the functions return `NaN`. ***NaN*** is a JavaScript value that means Not a Number. Ex: `parseInt("dog")` is `NaN`.

The JavaScript function ***isNaN()*** returns `true` if the argument is not a number, `false` otherwise. When the `isNaN()` argument is non-numeric, the function attempts to convert the argument into a number. Ex: `isNaN("dog")` is `true` because the non-numeric value "dog" cannot be converted into a number. But `isNaN("123")` is `false` because "123" can be converted into the number 123.

PARTICIPATION ACTIVITY | 6.2.4: Arithmetic practice with numbers and strings.

What is `secretCode` at the end of each code segment? Type "quotes" around strings. If not a number, type NaN.

1)
```
secretCode = 10 + "ten";
```

Check Show answer

2)
```
secretCode = "3" / "6";
```

Check Show answer

3)
```
secretCode = "3" + 5 * 2;
```

Check Show answer

4)
```
secretCode =
parseFloat("3.2") +
parseInt("2.7");
```

Check Show answer

5)
```
secretCode = 3 +
parseInt("pig");
```

Check Show answer

6)
```
// true = 1, false = 0
secretCode = 2 +
isNaN("oink") +
isNaN("5");
```

Check Show answer

CHALLENGE ACTIVITY | 6.2.1: Arithmetic operators.

550544.4142762.qx3zqy7

Start

Write a statement that assigns operationResult with the sum of value1 and value2. Ex: If value1 is 6 and value2 is 2, operationResult is 8. Note: Your code will be tested with more values for value1 and value2.

```
let value1 = 6;
let value2 = 2;

/* Replace 0 with an expression in the following line */
let operationResult = 0;

```

1 2 3 4

Check Next

Exploring further:

- Arithmetic operators from MDN
- MDN documentation for parseInt(), parseFloat(), and isNaN()

6.3 Conditionals

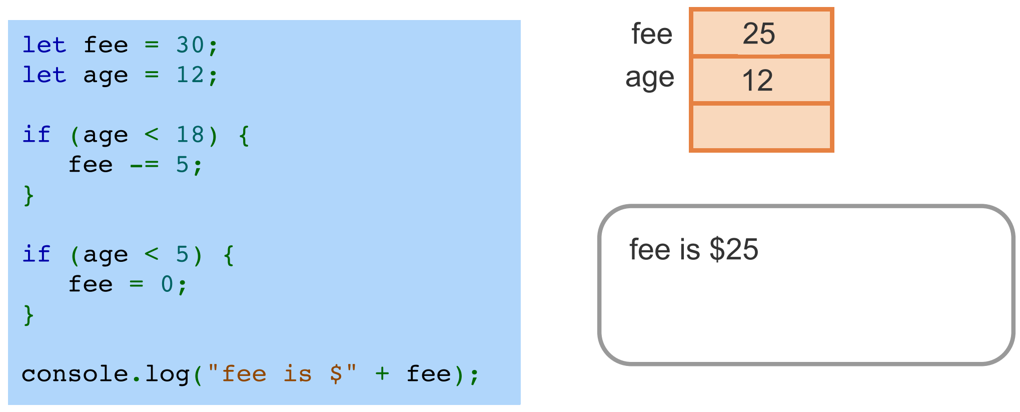
If statement

An ***if statement*** executes a group of statements if a condition is true. Braces `{ }` surround the group of statements. *Good practice is to indent statements in braces using a consistent number of spaces.* This material indents 3 spaces.

Construct 6.3.1: if statement.

```
if (condition) {
   // Statements to execute when condition is
true
}
```

PARTICIPATION ACTIVITY 6.3.1: Evaluating if statements.



```
let fee = 30;
let age = 12;

if (age < 18) {
   fee -= 5;
}

if (age < 5) {
   fee = 0;
}

console.log("fee is $" + fee);
```

```
fee is $25
```

Animation content:

The following code is displayed:
let fee = 30;
let age = 12;

if (age < 18) {
   fee -= 5;
}

```
if (age < 5) {
   fee = 0;
}

console.log("fee is $" + fee);
```

Variables fee and age are allocated memory and initialized with values 25 and 12, respectively. The string fee is $25 is displayed without quotes in the console after the code is run.

Animation captions:

1. Variable fee is assigned 30, and age is assigned 12.
2. Evaluate the condition: 12 < 18.
3. 12 < 18 is true, so the block under "if" executes, and fee is assigned 30 - 5 = 25.
4. Evaluate the condition: 12 < 5.
5. 12 < 5 is false, so the block under "if" does not execute. Variable fee remains 25.

PARTICIPATION ACTIVITY 6.3.2: If statement.

What is the final value of `numItems`?

1)
```
bonus = 19;
numItems = 1;
if (bonus > 10) {
   numItems = numItems +
3;
}
```

Check Show answer

2)
```
bonus = 0;
numItems = 1;
if (bonus > 10) {
   numItems = numItems +
3;
}
```

Check Show answer

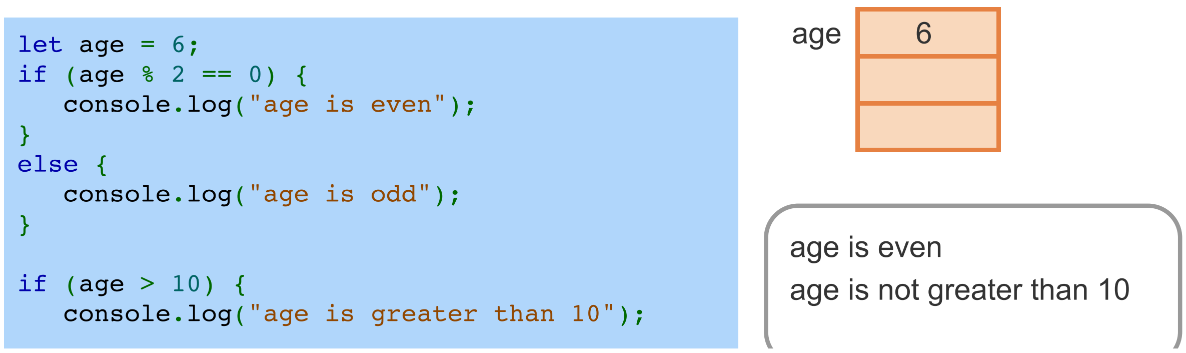
If-else statement

An ***if-else statement*** executes a block of statements if the statement's condition is true, and executes another block of statements if the condition is false.

Construct 6.3.2: if-else statement.

```
if (condition) {
   // statements to execute when condition is
true
}
else {
   // statements to execute when condition is
false
}
```

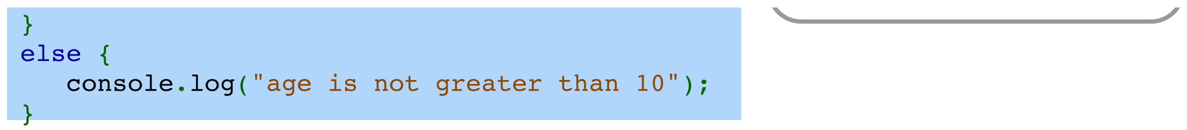
PARTICIPATION ACTIVITY 6.3.3: Evaluating if-else statements.



```
let age = 6;
if (age % 2 == 0) {
   console.log("age is even");
}
else {
   console.log("age is odd");
}

if (age > 10) {
   console.log("age is greater than 10");
```

```
age is even
age is not greater than 10
```



```
}
else {
   console.log("age is not greater than 10");
}
```

Animation content:

The following code is displayed:
let age = 6;
if (age % 2 == 0) {
  console.log("age is even");
}
else {
  console.log("age is odd");
}

if (age > 10) {
  console.log("age is greater than 10");
}
else {
  console.log("age is not greater than 10");
}
After the first line runs, the variable age is shown in memory initialized with value 6. Step 3 runs console.log("age is even"); and the string age is even is displayed in the console. Step 5 runs console.log("age is not greater than 10"); and the string age is not greater than 10 is displayed on a new line in the console.

Animation captions:

1. Variable age is assigned 6.
2. Evaluate the condition: 6 % 2 == 0.
3. 0 == 0 is true, so the block under "if" executes.
4. Evaluate the condition: age > 10.
5. 6 > 10 is false, so the block under "else" executes.

PARTICIPATION ACTIVITY | 6.3.4: If-else statements.

What is the final value of `numItems`?

1)
```
bonus = 5;
if (bonus < 12) {
   numItems = 100;
}
else {
   numItems = 200;
}
```

Check Show answer

2)
```
bonus = 12;
if (bonus < 12) {
   numItems = 100;
}
else {
   numItems = 200;
}
```

Check Show answer

3)
```
bonus = 15;
numItems = 44;
if (bonus < 12) {
   numItems = numItems +
3;
}
else {
   numItems = numItems +
6;
}
numItems = numItems + 1;
```

Check Show answer

4)
```
bonus = 5;
if (bonus < 12) {
   bonus = bonus + 2;
   numItems = 3 * bonus;
}
else {
   numItems = bonus + 10;
}
```

Check Show answer

Using { } around if and else blocks

JavaScript does not require braces `{}` *around if or else blocks with a single statement. Good practice is to always use braces, which results in more readable code that is less susceptible to logic errors.*

```
// Braces not required around single statements
if (vote == "M")
   memberCount++;
else
   nonMemberCount++;
```

Comparison operators

If and if-else statements commonly use comparison operators. A ***comparison operator*** compares two operands and evaluates to a Boolean value, meaning either true or false.

Table 6.3.1: Comparison operators.

Comparison operator	Name	Example

`==`	Equality	`2 == 2 // true`
		`"bat" == "bat" // true`

`!=`	Inequality	`2 != 3 // true`
		`"bat" != "zoo" // true`

`===`	Identity	`2 === 2 // true`
		`"2" === 2 // false`

`!==`	Non-identity	`2 !== 2 // false`
		`"2" !== 2 // true`

`<`	Less than	`2 < 3 // true`
		`"bat" < "zoo" // true`

`<=`	Less than or equal	`2 <= 3 // true`
		`"bat" <= "bat" // true`

`>`	Greater than	`3 > 2 // true`
		`"zoo" > "bat" // true`

`>=`	Greater than or equal	`3 >= 2 // true`
		`"zoo" >= "zoo" // true`

When the equality operator `==` and inequality `!=` operator compare a number and a string, the string is first converted to a number and then compared. Ex: `3 == "3"` is true because "3" is converted to 3 before the comparison, and 3 and 3 are the same.

The ***identity operator*** `===` performs ***strict equality***. Two operands are strictly equal if the operands' data types and values are equal. Ex: `3 === 3` is true because both operands are numbers and the same value, but `"3" === 3` is false because "3" is a string, and 3 is a number. The ***non-identity operator*** `!==` is the opposite of the identity operator. Ex: `"3" !== "3"` is false because both operands are the same type and value, but `"3" !== 3` is true because "3" is a string, and 3 is a number.

Other comparison operators also convert a string to a number when comparing a string with a number. Ex: `2 < "12"` is true because 2 is less than the number 12. When comparing two strings, JavaScript uses Unicode values to compare characters. Ex: `"cat" <= "dog"` is true because "c" has a smaller Unicode value than "d".

A common error when comparing two values for equality is to use a single `=` instead of `==` or `===`. Ex: `if (name = "Sue")` assigns `name` with "Sue" instead of asking if `name` equals "Sue".

What is Unicode?

Unicode *is a computing industry standard that assigns a unique number to characters in over one hundred different languages, including multiple symbol sets and emoji. The Unicode numbers for capital A-Z range from 65 to 90, and lowercase a-z range from 97 to 122.*

PARTICIPATION ACTIVITY 6.3.5: Comparison operators.

Is the if statement true or false?

1)
```
score = 2;
if (score == "2") {
   score = 10;
}
```

- true
- false

2)
```
score = 2;
if (score = "50") {
   score = 100;
}
```

- ○ true
- ○ false

3)
```
score = 2;
if (score === "2") {
   score = 10;
}
```

- ○ true
- ○ false

4)
```
status = "10";
if (status > 5) {
   length = 0;
}
```

- ○ true
- ○ false

5)
```
status = "good";
if (status > "bad") {
   bonus += 2;
}
```

- ○ true
- ○ false

6)
```
status = "charge";
if (status <= "chance") {
   bonus += 2;
}
```

- ○ true
- ○ false

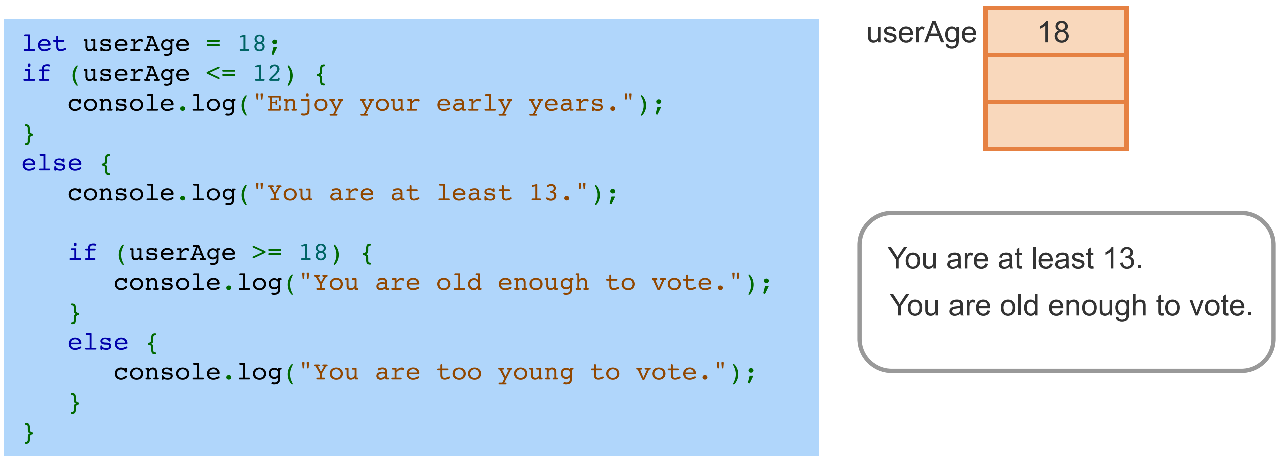
7)
```
lowerCaseLetters = "abc";
upperCaseLetters = "ABC";
if (lowerCaseLetters >
upperCaseLetters) {
   length++;
}
```

- true
- false

Nested statements and else-if statement

If and else block statements can include any valid statements, including another if or if-else statement. An if or if-else statement that appears inside another if or if-else statement is called a ***nested*** statement.

PARTICIPATION ACTIVITY | 6.3.6: Nested if-else statement example.



```
let userAge = 18;
if (userAge <= 12) {
   console.log("Enjoy your early years.");
}
else {
   console.log("You are at least 13.");

   if (userAge >= 18) {
      console.log("You are old enough to vote.");
   }
   else {
      console.log("You are too young to vote.");
   }
}
```

```
You are at least 13.
You are old enough to vote.
```

Animation content:

The following code is displayed:
let userAge = 18;
if (userAge <= 12) {
  console.log("Enjoy your early years.");
}

```
else {
  console.log("You are at least 13.");

  if (userAge >= 18) {
    console.log("You are old enough to vote.");
  }
  else {
    console.log("You are too young to vote.");
  }
}
```

In step 1, the variable userAge is allocated memory and initialized to 18. Step 2 runs the first line in the else block and the string You are at least 13. is displayed in the console. Step 3 runs the code in the nested if statement and the string You are old enough to vote. is displayed in the console.

Animation captions:

1. userAge is 18. Since 18 <= 12 is false, the outer if-else statement's else block executes.
2. After outputting to the console, the nested if-else statement executes.
3. 18 >= 18 is true, so the if block executes.
4. No more statements exist in the nested if-else statement or the outer if-else statement.

A common situation is when several nested if-else statements are needed to execute one and only one block of statements. The ***else-if*** statement is an alternative to nested if-else statements that produces an easier-to-read list of statement blocks.

In the example below, the `grade` variable is assigned with A, B, C, D, or F depending on the `score` variable. The code segment on the left uses nested if-else statements. The code segment on the right performs the same logic with else-if statements.

Figure 6.3.1: Nested if-else statements vs. else-if statements.

```
// Nested if-else statements
if (score >= 90) {
   grade = "A";
}
else {
   if (score >= 80) {
      grade = "B";
   }
   else {
      if (score >= 70) {
         grade = "C";
      }
      else {
         if (score >= 60) {
            grade = "D";
         }
         else {
            grade = "F";
         }
      }
   }
}
```

```
// else-if statements
if (score >= 90) {
   grade = "A";
}
else if (score >= 80) {
   grade = "B";
}
else if (score >= 70) {
   grade = "C";
}
else if (score >= 60) {
   grade = "D";
}
else {
   grade = "F";
}
```

PARTICIPATION ACTIVITY 6.3.7: Nested if-else practice.

Use a nested if-else statement or an else-if statement to examine the variable `golfScore`.

- If `golfScore` is above 90, output to the console "Keep trying!"
- Otherwise, if `golfScore` is above 80, output to the console "Nice job!"
- Otherwise, output to the console "Ready to go pro!"

Test your code with values above 90, between 81 and 90, and 80 and below to ensure your logic is correct.

```
let golfScore = 95;

// Write your if-else statements here!

```

Run JavaScript   Reset code

Your console output

▶ View solution

PARTICIPATION ACTIVITY 6.3.8: Nested if and if-else statements.

What is `numBoxes` at the end of each code segment?

1)
```
numApples = 2;
numOranges = 5;
numBoxes = 0;
if (numApples % 2 != 0) {
   numBoxes = 1;
}
else {
   if (numApples +
numOranges > 10) {
      numBoxes = 2;
   }
   else {
      numBoxes = 99;
   }
}
```

Check Show answer

2)
```
numApples = 2;
numOranges = 5;
numBoxes = 0;
if (numApples > 0) {
   if (numOranges > 10) {
      numBoxes = 4;
   }
   numBoxes++;
}
else {
   numBoxes = 99;
}
```

Check Show answer

3)
```
produce = "carrots";
if (produce == "apples")
{
   numBoxes = 1;
}
else if (produce ==
"bananas") {
   numBoxes = 2;
}
else if (produce ==
"carrots") {
   numBoxes = 3;
}
else {
   numBoxes = 4;
}
```

Check Show answer

Logical operators

JavaScript logical operators perform AND, OR, and NOT logic.

Table 6.3.2: Logical operators.

Logical operator	Name	Description	Example
`&&`	And	True if both sides are true	`(1 < 2 && 2 < 3)  // true`
`||`	Or	True if either side is true	`(1 < 2 || 2 < 0)  // true`
`!`	Not	True if expression is not true	`!(2 == 2)  // false`

Multiple `&&` and `||` conditions may be combined into a single ***complex condition***. Ex:

`(1 < 2 && 2 < 3 || 3 < 4)`. Complex conditions are evaluated from left to right, but `&&` has higher precedence than `||`, so `&&` is evaluated before `||`. *Good practice is to use parentheses `()` around conditions that use `&&` and `||` to explicitly indicate the order of evaluation. Ex: `(a < 0 || a > 1 && b > 2)` is better expressed as: `(a < 0 || (a > 1 && b > 2))`.*

Logic involving "not" can be difficult for humans to correctly read or understand. Ex: "Are you not hungry?" is more difficult for a human to understand than the equivalent "Are you satisfied?" *Good practice is to avoid using the not operator when possible. Ex: `!(score > 10)` is better expressed as: `score <= 10`.*

PARTICIPATION ACTIVITY 6.3.9: Evaluating complex conditions.

What is `decision` at the end of each code segment?

1)
```
homeTeam = 2;
visitingTeam = 5;
if (homeTeam > 10 ||
visitingTeam > 0) {
   decision = 1;
}
else {
   decision = 0;
}
```

Check Show answer

2)
```
homeTeam = 2;
visitingTeam = 5;
if (!(homeTeam > 10 &&
visitingTeam > 0)) {
   decision = 1;
}
else {
   decision = 0;
}
```

Check Show answer

3)
```
homeTeam = 2;
visitingTeam = 5;
if (homeTeam > 10 ||
(visitingTeam != 2 &&
visitingTeam > 0)) {
   decision = 1;
}
else {
   decision = 0;
}
```

Check Show answer

CHALLENGE ACTIVITY 6.3.1: Conditionals.

550544.4142762.qx3zqy7

Start

Write an if-else statement that if userTickets is less than or equal to 5, executes awardPoints = 10. Else, execute awardPoints = userTickets. Ex: If userTickets is 4, then awardPoints = 10.

```
let awardPoints = 0;
let userTickets = 4; // Code will be tested with values: 4, 5, and 6

/* Your solution goes here */

```

1 2 3 4

Check Next

6.4 More conditionals

Truthy and falsy

A ***truthy*** value is a non-Boolean value that evaluates to `true` in a Boolean context. Ex: `if (18)` evaluates to `true` because non-zero numbers are truthy values. A ***falsy*** value is a non-Boolean value that evaluates to `false` in a Boolean context. Ex: `if (null)` evaluates to `false` because `null` is a falsy value.

Table 6.4.1: Truthy values.

Example	Description
`if (32)`	Non-zero number
`if ("cat")`	Non-empty string
`if (myObject)`	Object variable
`if (myArray)`	Array variable

Table 6.4.2: Falsy values.

Example	Description
`if (0)`	Zero
`if ("")`	Empty string
`if (NaN)`	Not a number
`if (undefined)`	Variable that has not been assigned a value
`if (null)`	No object value

PARTICIPATION ACTIVITY 6.4.1: Truthy and falsy values.

Indicate if the `if` statement's condition evaluates to `true` or `false`.

1) `if (undefined)`

- ○ true
- ○ false

2) `if (999)`

- ○ true
- ○ false

3) `if (0)`

- ○ true
- ○ false

4) `if ("")`

- ○ true
- ○ false

5) `if (" ")`

- ○ true
- ○ false

6) `if ("false")`

- ○ true
- ○ false

7) `if (myArray)`

- ○ true
- ○ false

Conditional (ternary) operator

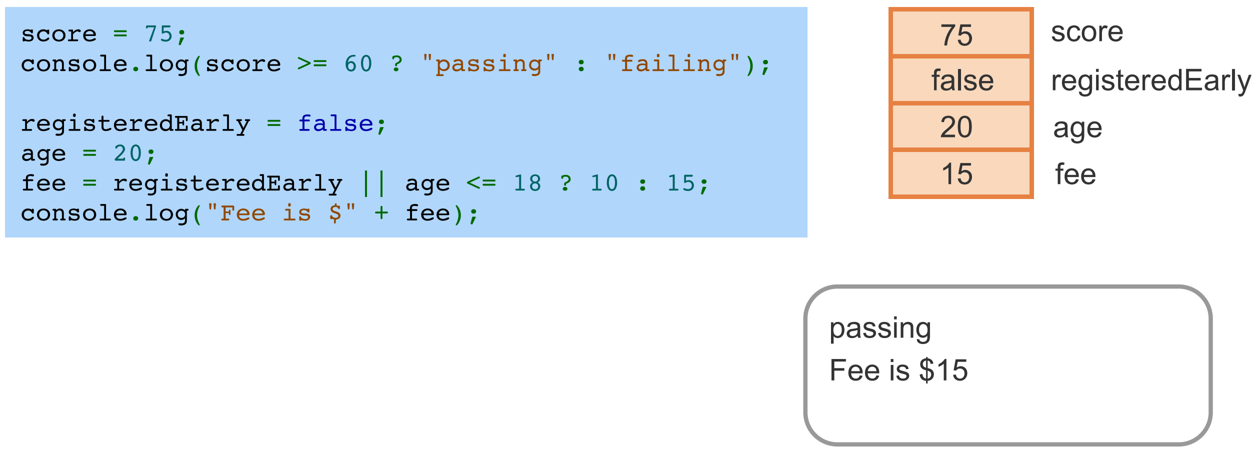
The conditional operator allows developers to write concise conditional statements. The ***conditional operator*** (or ***ternary operator***) has three operands separated by a question mark (`?`)

and colon (`:`). If the `condition` evaluates to `true`, then the value of `expression1` is returned, otherwise the value of `expression2` is returned.

Construct 6.4.1: Conditional (ternary) operator.

```
condition ? expression1 :
expression2
```

PARTICIPATION ACTIVITY 6.4.2: Evaluating the conditional operator.



```
score = 75;
console.log(score >= 60 ? "passing" : "failing");

registeredEarly = false;
age = 20;
fee = registeredEarly || age <= 18 ? 10 : 15;
console.log("Fee is $" + fee);
```

75	score
false	registeredEarly
20	age
15	fee

```
passing
Fee is $15
```

Animation content:

The following code is displayed:
score = 75;
console.log(score >= 60 ? "passing" : "failing");

registeredEarly = false;
age = 20;
fee = registeredEarly || age <= 18 ? 10 : 15;
console.log("Fee is $" + fee);

Step 1 and 2 run the first 2 lines. The left side of the ? operator, score >= 60, evaluates to true so

the entire expression evaluates to passing and passing is displayed to the console.
Step 3 and 4 run the remaining lines of code. The left side of the ? operator evaluates to false so the entire expression evaluates to 15. fee is assigned 15 and the last line displays Fee is $15 to the console.

Animation captions:

1. 75 >= 60 evaluates to true.
2. Ternary operator returns "passing", so "passing" is displayed in the console.
3. false || 20 <= 18 is false.
4. Ternary operator returns 15, so fee is assigned 15 and output to the console.

PARTICIPATION ACTIVITY 6.4.3: Conditional operator.

1) Complete the code to assign `lateStatus` with "yep" if `currTime` is greater than 60, and "nope" otherwise.

```
lateStatus = currTime > 60
        "yep" : "nope";
```

Check Show answer

2) Complete the code to assign `y` with `x` if `x` is greater than 0, and -1 otherwise.

```
y = (x > 0) ?
                  ;
```

Check Show answer

3) What is `boardType` after the following statements?

```
year = 1985;
boardType = year >= 2015
? "hoverboard" :
"skateboard";
```

Check Show answer

4) What is `priority` after the following statements?

```
attempt = 4;
priority = 2;
attempt > 3 ? priority++
: priority--;
```

Check Show answer

Switch statement

The switch statement is an alternative to writing multiple else-if statements. A ***switch statement*** compares an expression's value to several cases using strict equality (===) and executes the first matching case's statements. If no case matches, an optional default case's statements execute.

The ***break statement*** stops executing a case's statements and causes the statement immediately following the switch statement to execute. Omitting the break statement causes the next case's statements to execute, even though the case does not match.

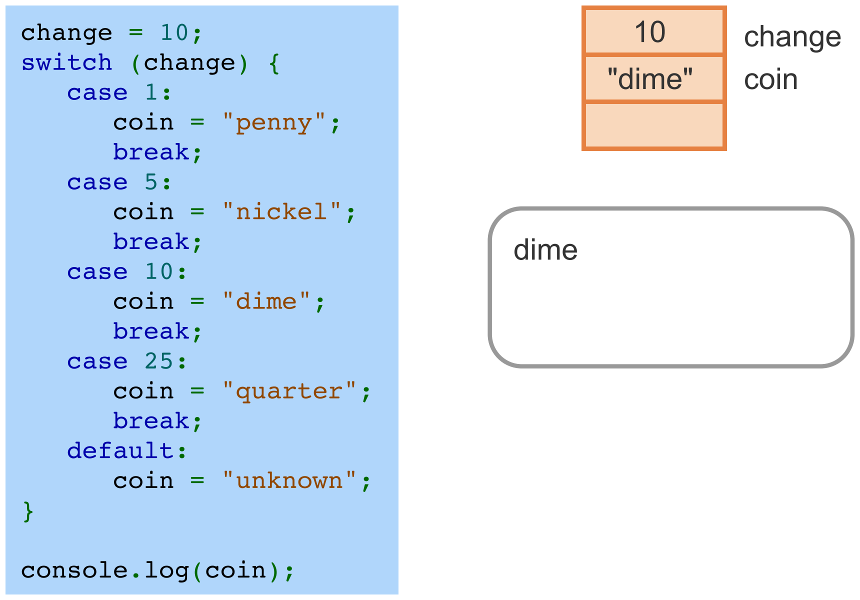
Construct 6.4.2: switch statement.

```
switch (expression) {
   case value1:
      // Statements executed when expression's value matches
value1
      break;    // optional
   case value2:
      // Statements executed when expression's value matches
value2
      break;    // optional

   // ...

   default:
      // Statements executed when no cases match
}
```

PARTICIPATION ACTIVITY 6.4.4: Evaluating the switch statement.



```
change = 10;
switch (change) {
   case 1:
      coin = "penny";
      break;
   case 5:
      coin = "nickel";
      break;
   case 10:
      coin = "dime";
      break;
   case 25:
      coin = "quarter";
      break;
   default:
      coin = "unknown";
}

console.log(coin);
```

Animation content:

The following code is displayed:

```
change = 10;
switch (change) {
  case 1:
    coin = "penny";
    break;
  case 5:
    coin = "nickel";
    break;
  case 10:
    coin = "dime";
    break;
  case 25:
    coin = "quarter";
    break;
  default:
    coin = "unknown";
}

console.log(coin);
```

Step 1 assigns change to 10. Steps 2 through 4 compare the value and variable type of change with the value of each case. change has the same type and value as 10 so the case 10: block is executed. coin is declared and assigned the string dime.
Step 5 runs the break; line and no more cases are checked or run. This includes the default: block. The last line of code is run and dime is displayed on the console.

Animation captions:

1. switch statement examines the change variable.
2. change === 1 is false, so the case does not match.
3. change === 5 is false, so the case does not match.
4. change === 10 is true, so the case matches, and the case's statements are executed.
5. Break statement stops executing the switch statement. The code after the switch executes, outputting "dime" to the console.

PARTICIPATION ACTIVITY | 6.4.5: switch statement.

Refer to the switch statement below.

```
switch (item) {
   case "apple":
   case "orange":
      fruits++;
      break;
   case "milk":
      drinks++;
   case "cheese":
      dairy++;
      break;
   case "beef":
   case "chicken":
      meat++;
      break;
   default:
      other++;
}
```

1) If `item` is "beef", what variables are incremented?

- ○ `other`
- ○ `meat` only
- ○ `meat` and `other`

2) If `item` is "milk", what variables are incremented?

- ○ `other`
- ○ `drinks` only
- ○ `drinks` and `dairy`

3) If `item` is "Apple", what variable is incremented?

- ○ `other`
- ○ `fruits`
- ○ Nothing is incremented.

PARTICIPATION ACTIVITY

6.4.6: Practice with the switch statement.

Convert the group of else-if statements into an equivalent switch statement.

```

// Get a number between 0 and 6 representing the day of the week (0 =
let currDay = new Date().getDay();

// Convert into an equivalent switch statement
if (currDay === 1) {
   console.log("I love Mondays!");
}
else if (currDay === 2 || currDay === 3 || currDay === 4) {
   console.log("Working hard!");
}
else if (currDay === 5) {
   console.log("TGIF!");
}
else {
   console.log("Time to relax!");
```

Run JavaScript

Reset code

Your console output

```
I love Mondays!
```

▶ View solution

CHALLENGE ACTIVITY 6.4.1: More conditionals.

550544.4142762.qx3zqy7

Start

Complete the conditional operator so ticketPrice is assigned with 13 if membershipLevel is 7 or above, or with 20 otherwise. Ex: If membershipLevel = 8, then ticketPrice is assigned with 13.

```
// Your code will be tested with 8 and other values
let membershipLevel = 8;

let ticketPrice = /* Your solution goes here */;

console.log(ticketPrice);
```

1 2

Check Next

6.5 Loops

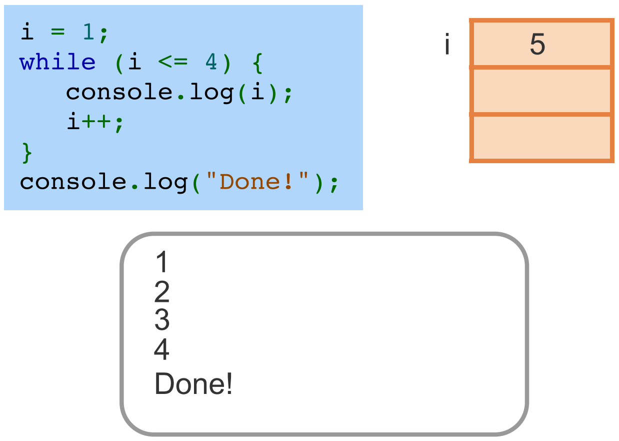
While loop

Three general-purpose looping constructs exist: while, do-while, and for loops. The ***while loop*** is a looping structure that checks if the loop's condition is true before executing the loop body, repeating until the condition is false. The ***loop body*** is the set of statements that a loop repeatedly executes.

Construct 6.5.1: while loop.

```
while (condition)
{
   body
}
```

PARTICIPATION ACTIVITY 6.5.1: Executing a while loop.



```
i = 1;
while (i <= 4) {
   console.log(i);
   i++;
}
console.log("Done!");
```

```
1
2
3
4
Done!
```

Animation content:

The following code is displayed:
i = 1;
while (i <= 4) {
  console.log(i);
  i++;
}
console.log("Done!");
Step 1 runs line 1 and variable i is allocated memory.

Step 3 displays the integer 1 in the console.
In step 4 i is now assigned the value 2.
Step 6 displays the strings 2, 3, 4, and Done! all displayed on new lines in the console.

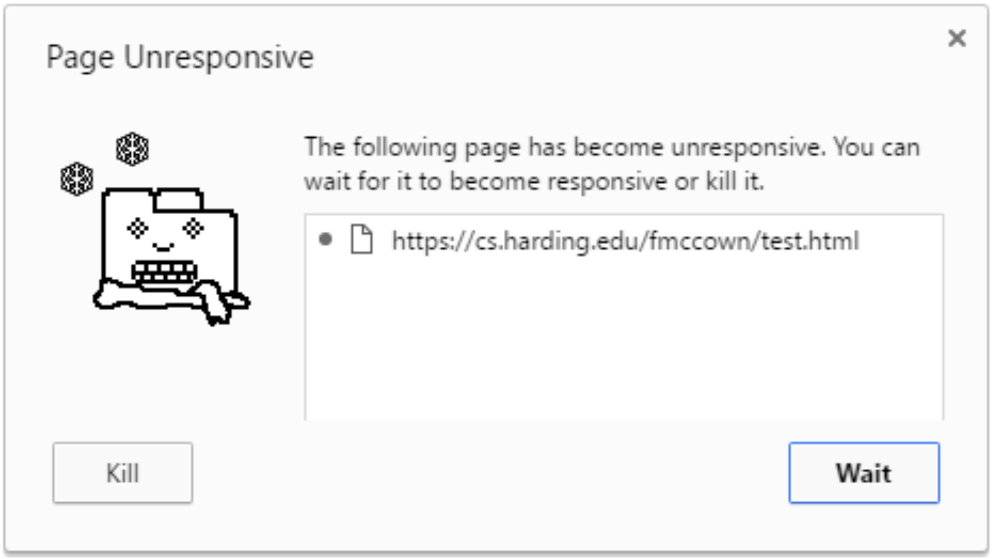
Animation captions:

1. Assign i with 1.
2. 1 <= 4 is true, so the loop's body executes.
3. Output i to the console.
4. Increment i.
5. End of loop so go back to the top and re-evaluate the condition.
6. Keep executing loop until i <= 4 is false.

Developers must be careful to avoid writing infinite loops. An ***infinite loop*** is a loop that never stops executing. Ex: `while (true);` is an infinite loop because the loop's condition is never false.

JavaScript infinite loop with Chrome's "Page Unresponsive" message

If the web browser is running JavaScript that contains an infinite loop, the web browser tab will cease to respond to user input.



Page Unresponsive

The following page has become unresponsive. You can wait for it to become responsive or kill it.

https://cs.harding.edu/fmccown/test.html

PARTICIPATION ACTIVITY 6.5.2: while loop.

1) What are the first and last numbers output by the code segment?

```
let c = 100;
while (c > 0) {
   console.log(c);
   c -= 10;
}
```

- ○ 100 and 0.
- ○ 90 and 0.
- ○ 100 and 10.

2) What condition makes the loop output the even numbers 2 through 20?

```
let c = 2;
while (_____) {
   console.log(c);
   c += 2;
}
```

- ○ `c >= 20`
- ○ `c <= 20`
- ○ `c < 20`

3) What is the value of `c` when the loop terminates?

```
let c = 10;
while (c <= 20); {
   console.log(c);
   c += 5;
}
```

- ○ 25
- ○ 20
- ○ The loop never terminates.

4) What is `c` when the loop terminates?

```
let c = 10;
while (c <= 20)
   console.log(c);
   c += 5;
```

- ○ 15
- ○ 20
- ○ The loop never terminates.

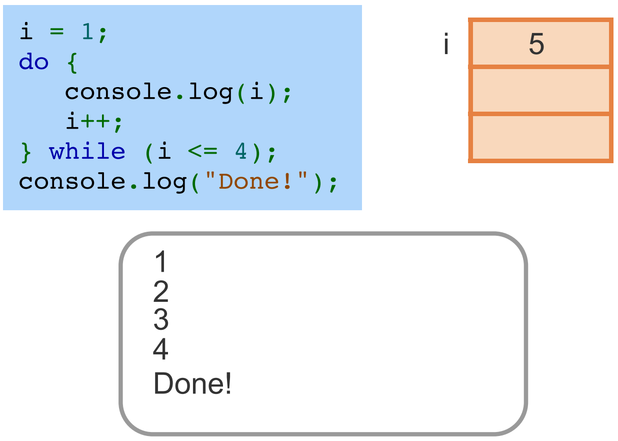
Do-while loop

The ***do-while loop*** executes the loop body before checking the loop's condition to determine if the loop should execute again, repeating until the condition is false.

Construct 6.5.2: do-while loop.

```
do {
   body
} while
(condition);
```

PARTICIPATION ACTIVITY 6.5.3: Executing a do-while loop.



```
i = 1;
do {
   console.log(i);
   i++;
} while (i <= 4);
console.log("Done!");
```

```
1
2
3
4
Done!
```

Animation content:

The following code is displayed:

```
i = 1;
do {
  console.log(i);
  i++;
} while (i <= 4);
console.log("Done!");
```

In step 2, i now stores 2 in memory and 1 is displayed in the console before the loop condition line is run.
In step 3, the strings 2, 3, 4, and Done are all displayed in the console on new lines.

Animation captions:

1. Assign i with 1.
2. do..while loop executes the loop body, evaluating the loop condition after the first execution.
3. The loop repeatedly executes until i <= 4 is false.

PARTICIPATION ACTIVITY 6.5.4: do-while loop.

1) What is the last number output to the console?

```
let c = 10;
do {
   console.log(c);
   c--;
} while (c >= 5);
```

Check Show answer

2) Write a condition that executes the do-while loop as long as the user enters a negative number.

```
let num;
do {
   num = prompt("Enter a
negative number:");
} while (______);
```

Check Show answer

3) What is the last number output to the console?

```
let x = 1;
do {
   let y = 0;
   do {
      console.log(x + y);
      y++;
   } while (y < 3);
   x++;
} while (x < 4);
```

Check Show answer

PARTICIPATION ACTIVITY 6.5.5: Practice with the while and do-while loops.

A given insect population doubles every week. If 2 insects exist on the first week, how many weeks will pass until the insect population exceeds 10,000 insects? Use a `while` loop to output the insect population each week until the population exceeds 10,000 insects.

Researchers have discovered that every 4 weeks a disease is killing 40% of the insect population after the population has reproduced. If 2 insects exist on the first week, how many weeks will pass until the insect population exceeds 10,000 insects? Write a second `do-while` loop that outputs the insect population each week until the population exceeds 10,000 insects. Decimal places will appear in the number of insects after removing 40% of the population on week 4.

```
// Write the while and do-while loops here!

```

Run JavaScript

Reset code

Your console output

▶ View solution

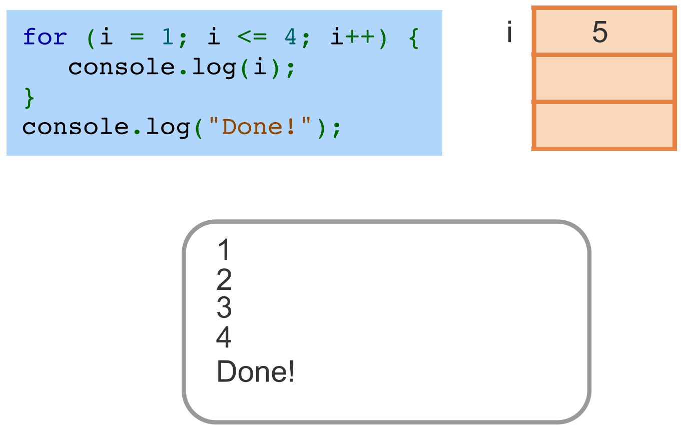
For loop

A for loop collects three parts — the loop variable initialization, loop condition, and loop variable update — all at the top of the loop. A ***for loop*** executes the initialization expression, evaluates the loop's condition, and executes the loop body if the condition is true. After the loop body executes, the final expression is evaluated, and the loop condition is checked to determine if the loop should execute again.

Construct 6.5.3: for loop.

```
for (initialization; condition;
finalExpression) {
   body
}
```

PARTICIPATION ACTIVITY 6.5.6: Executing a for loop.



```
for (i = 1; i <= 4; i++) {
   console.log(i);
}
console.log("Done!");
```

```
1
2
3
4
Done!
```

Animation content:

The following code is displayed:

```
for (i = 1; i <= 4; i++) {
  console.log(i);
}
```

console.log("Done!");
In step 1, i is allocated memory and stores 1.
In step 3, 1 is displayed in the console.
In step 4, i now stores the value 2.
In step 5, i keeps incrementing until i stores 5. 2, 3, 4, and Done! are displayed in the console on new lines

Animation captions:

1. for loop's initial expression assigns i with 1.
2. for loop's condition is then evaluated. 1 <= 4 is true, so the loop's statements are executed.
3. Output i to the console.
4. After the loop executes, the final expression is evaluated, which increments i.
5. Loop repeatedly executes until i <= 4 is false.

PARTICIPATION ACTIVITY 6.5.7: For loops.

1) Which loop always executes the loop body at least once?

- ○ `while`
- ○ `do-while`
- ○ `for`

2) What numbers are output by the code segment?

```
for (c = 5; c < 10; c += 2) {
   console.log(c);
}
```

- ○ 5, 7, 9
- ○ 5, 6, 7, 8, 9
- ○ Infinite loop

3) What condition causes the for loop to output the numbers 100 down to 50, inclusively?

```
for (c = 100; ______; c--) {
   console.log(c);
}
```

- ○ `c < 50`
- ○ `c > 50`
- ○ `c >= 50`

4) What numbers are output by the code segment?

```
for (x = 1; x <= 3; x++) {
   for (y = 2; y <= 4; y++) {
      console.log(y);
   }
}
```

- ○ 2, 3, 4
- ○ 2, 3, 4, 2, 3, 4, 2, 3, 4
- ○ 2, 3, 4, 5, 6, 7, 8, 9, 10

break and continue statements

Two *jump* statements alter the normal execution of a loop. The ***break*** statement breaks out of a loop prematurely. The ***continue*** statement causes a loop to iterate again without executing the remaining statements in the loop.

```
for (c = 1; c <= 5; c++) {
   if (c == 4) {
      break;    // Leave the loop
   }
   console.log(c);    // 1 2 3 (missing 4 and 5)
}

for (c = 1; c <= 5; c++) {
   if (c == 4) {
      continue;    // Iterate again immediately
   }
   console.log(c);    // 1 2 3 (missing 4) 5
}
```

Some developers use `break` and `continue` to write short and concise code, but jump statements may introduce subtle logic errors that are difficult to find. This material does not use jump statements.

CHALLENGE ACTIVITY 6.5.1: Loops.

550544.4142762.qx3zqy7

Start

Write a while loop that multiplies enteredNum by 3 while enteredNum is less than 60, displaying the result after each multiplication. Ex: For enteredNum = 5, output is: 15 45 135 (with each value output on a separate line)

```
let enteredNum = 5; // Code will be tested with values: 5, 10 and 80

/* Your solution goes here */

```

1 2 3

Check Next

6.6 Functions

Introduction to functions

A ***function*** is a named group of statements. JavaScript functions are declared with the `function` keyword followed by the function name and parameter list in parentheses `()`. A ***parameter*** is a variable that supplies the function with input. The function's statements are enclosed in braces `{}`.

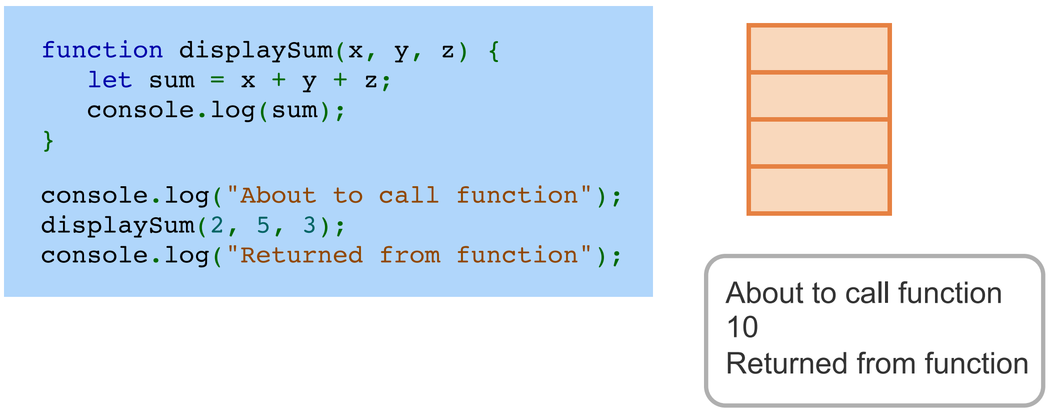
Invoking a function's name, known as a ***function call***, causes the function's statements to execute. An ***argument*** is a value provided to a function's parameter during a function call.

Construct 6.6.1: Function declaration.

```
function functionName(parameter1, parameter2,
...) {
   // Statements to execute when function is
called

}
```

PARTICIPATION ACTIVITY | 6.6.1: Declaring and calling a function.



```
function displaySum(x, y, z) {
   let sum = x + y + z;
   console.log(sum);
}

console.log("About to call function");
displaySum(2, 5, 3);
console.log("Returned from function");
```

```
About to call function
10
Returned from function
```

Animation content:

The following code is displayed:

function displaySum(x, y, z) {
  let sum = x + y + z;
  console.log(sum);
}
In step 2, About to call function is displayed in the console. 2, 5, and 3 are stored in memory as x, y, and z respectively.
In step 3, the lines of code defined inside the function displaySum are run. A new variable sum is declared and stores 10 in memory.
In step 4, 10 and Returned from function are each displayed on new lines in the console.

Animation captions:

1. A function named displaySum is declared with three parameters: x, y, and z.
2. displaySum() is called with arguments 2, 5, and 3, which are assigned to parameters x, y, and z.
3. The variable sum is assigned the sum of x, y, and z, which is 10.
4. sum is output to the console. No more code exist in the function, so the function is finished executing.

Good practice is to use function names that contain a verb and noun. Ex: ***`display`*** *is a vague function name, but* ***`displayAverage`*** *is better because* ***`displayAverage`*** *indicates what is being displayed.*

Good practice is to use camel case for JavaScript function names, where the name starts with a lowercase letter and subsequent words begin with a capital letter.

PARTICIPATION ACTIVITY 6.6.2: Declaring and calling functions.

1) Which function call displays the numbers 5, 4, 3, 2, 1.

```
function countDown(firstNum) {
   for (let count = firstNum;
count > 0; count--) {
      console.log(count);
   }
}
```

○ `countDown();`

○ `countDown(5);`

○ `countDown(5, 4, 3, 2, 1);`

2) Choose a better name for the function `test`.

```
function test(x, y) {
   if (x > y) {
      console.log(x);
   }
   else {
      console.log(y);
   }
}
```

- ○ `Largest`
- ○ `display_largest`
- ○ `displayLargest`

3) What is output to the console?

```
function sayHello(name,
greeting) {
   console.log(greeting + ", "
+ name + "!");
}

sayHello("Maria");
```

- ○ Hello, Maria!
- ○ Hello, undefined!
- ○ undefined, Maria!

4) The function below uses a default parameter value "Hello" that is assigned when the greeting is not supplied in the function call. What is output to the console?

```
sayHello("Sam");
sayHello("Juan", "Hola");

function sayHello(name,
greeting = "Hello") {
   console.log(greeting + ", "
+ name);
}
```

- ○ Hello, Sam
  Hello, Juan
- ○ Hello, Sam
  Hola, Juan
- ○ undefined

PARTICIPATION ACTIVITY 6.6.3: Function practice.

The code below produces a 5 x 10 box of question marks. Convert the code into a function called `drawBox()` that has three parameters:

1. `numRows` - The number of rows for the box.
2. `numCols` - The number of columns for the box.
3. `boxChar` - The character to use to create the box. If no argument is supplied, use "X".

Ex: `drawBox(5, 4, "!")` and `drawBox(2, 6)` should display the boxes pictured below.

```
!!!!
!!!!
!!!!
!!!!
!!!!
XXXXXX
XXXXXX
```

```
// Convert into a drawBox function
for (let r = 0; r < 5; r++) {
   let line = "";
   for (let c = 0; c < 10; c++) {
      line += "?";
   }
   console.log(line);
}

```

Run JavaScript   Reset code

Your console output

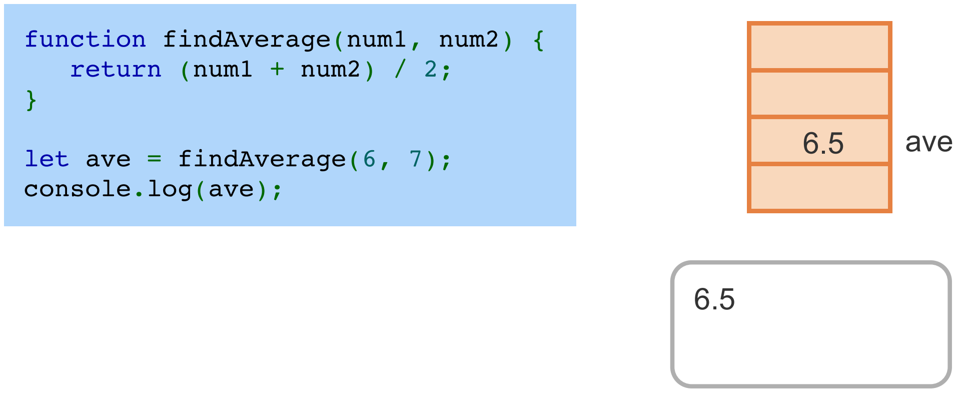
```
??????????
??????????
??????????
??????????
??????????
```

▶ View solution

Returning a value

A function may return a single value using a ***return*** statement. A function that is missing a return statement returns `undefined`.

PARTICIPATION ACTIVITY 6.6.4: Function that returns a value.



```
function findAverage(num1, num2) {
   return (num1 + num2) / 2;
}

let ave = findAverage(6, 7);
console.log(ave);
```

Animation content:

The following code is displayed:
function findAverage(num1, num2) {
  return (num1 + num2) / 2;
}

let ave = findAverage(6, 7);
console.log(ave);
In step 2, the parameters for findAverage, num1 and num2, are allocated memory and get assigned 6 and 5 respectively.
In step 3, the function findAverage finishes running. The blocks of memory used by the parameters num1 and num2 are deallocated and free for use by other variables. The variable avg is allocated memory and stores 6.5. 6.5 is displayed in the console.

Animation captions:

1. A function named findAverage is declared with two parameters: num1 and num2.
2. findAverage() is called with arguments 6 and 7, which are assigned to parameters num1 and num2.
3. The return statement returns the average of num1 and num2, which is 6.5.

PARTICIPATION ACTIVITY 6.6.5: Functions that return values.

1) What is output to the console?

```
console.log(findSmallest(5,
2));

function findSmallest(x, y) {
   if (x < y) {
      return x;
   }
   else {
      return y;
   }
}
```

- 2
- 5
- undefined

2) What is the correct way to call `factorial()` and output the factorial of 5?

```
function factorial(num) {
   let result = 1;
   for (let count = 1; count
<= num; count++) {
      result *= count;
   }
   return result;
}
```

○ `factorial(5);`
`console.log(result);`

○ `let answer =`
`factorial();`
`console.log(answer);`

○ `let answer =`
`factorial(5);`
`console.log(answer);`

3) What is output to the console?

```
function factorial(num) {
   let result = 1;
   for (let count = 1; count
<= num; count++) {
      result *= count;
   }
   return result;
}

let answer = factorial(8 -
factorial(3));
console.log(answer);
```

○ 2

○ 6

○ 8

4) What is output to the console?

```
console.log(sayHello("Sam"));

function sayHello(name) {
   console.log("Hello, " +
name + "!");
}
```

- ○ Hello, Sam!
- ○ Hello, Sam!
  undefined
- ○ undefined

Function expressions and anonymous functions

JavaScript functions may be assigned to a variable with a function expression. A ***function expression*** is identical to a function declaration, except the function name may be omitted. A function without a name is called an ***anonymous function***. Anonymous functions are often used with arrays and event handlers, discussed elsewhere in this material.

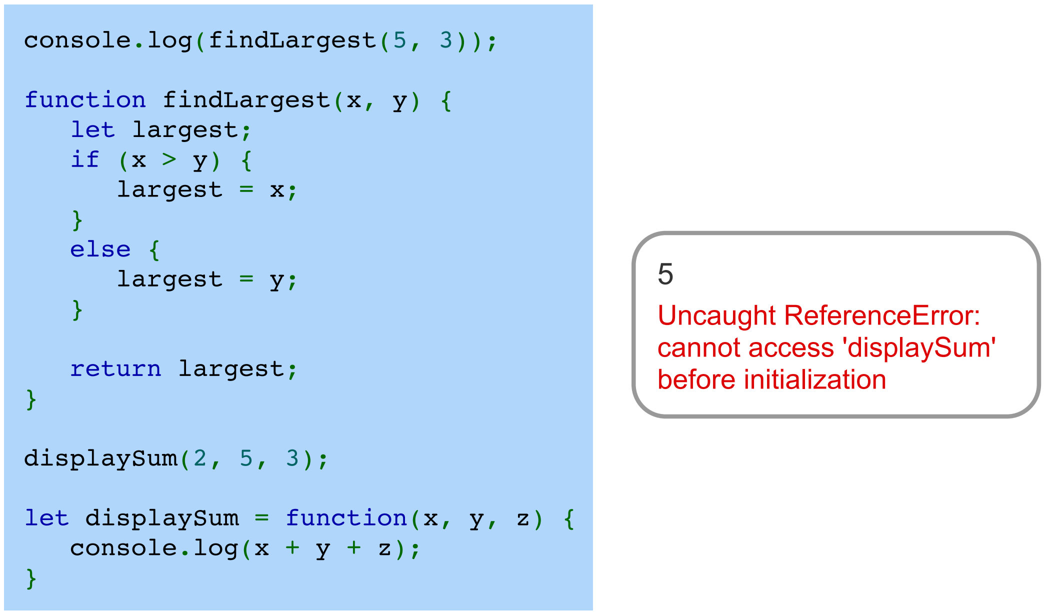
Figure 6.6.1: Assigning a function expression to a variable.

```
// Function name is omitted
let displaySum = function(x, y, z)
{
   console.log(x + y + z);
}

// Function call
displaySum(2, 5, 3);
```

Unlike functions declared with a function declaration, a variable assigned with a function expression cannot be used until after the variable is assigned. Using a variable before the variable is assigned with a function expression causes an exception.

PARTICIPATION ACTIVITY 6.6.6: Using a function expression before assignment.



```
console.log(findLargest(5, 3));

function findLargest(x, y) {
   let largest;
   if (x > y) {
      largest = x;
   }
   else {
      largest = y;
   }

   return largest;
}

displaySum(2, 5, 3);

let displaySum = function(x, y, z) {
   console.log(x + y + z);
}
```

```
5
Uncaught ReferenceError:
cannot access 'displaySum'
before initialization
```

Animation content:

The following code is displayed:

```
console.log(findLargest(5, 3));

function findLargest(x, y) {
  let largest;
  if (x > y) {
    largest = x;
  }
  else {
    largest = y;
  }

  return largest;
}

displaySum(2, 5, 3);

let displaySum = function(x, y, z) {
  console.log(x + y + z);
}
```

In step 2, 5 is displayed on the console.
In step 3, the error messageUncaught ReferenceError: cannot access 'displaySum' before initialization is displayed on the console

Animation captions:

1. findLargest() may be called before the findLargest() function declaration.
2. Since x > y, findLargest() returns 5, and 5 is output to the console.
3. Calling displaySum() before displaySum is assigned with a function expression produces an exception.

PARTICIPATION ACTIVITY 6.6.7: Function expressions.

1) The variable `result` is assigned 4.

```
let square = function(num) {
   return num * num;
}
let result = square(2);
```

- ○ True
- ○ False

2) The variable `result` is assigned 9.

```
let result = square(3);
let square = function(num) {
   return num * num;
}
```

- ○ True
- ○ False

3) The variable `result` is assigned 9.

```
let square = function(num) {
   return num * num;
}
let result = square;
```

- True
- False

Arrow functions

An ***arrow function*** is an anonymous function that uses an arrow => to create a compact function. An arrow function's parameters are listed to the left of the arrow. The right side of the arrow may be a single expression or multiple statements in braces.

Construct 6.6.2: Arrow function declaration that returns a single expression.

```
(parameter1, parameter2, ...) =>
expression
```

Construct 6.6.3: Arrow function with multiple statements.

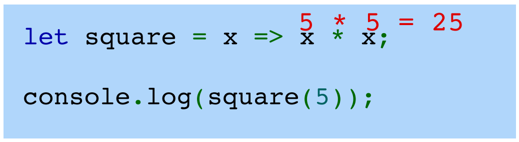
```
(parameter1, parameter2, ...) => {
statements; }
```

PARTICIPATION ACTIVITY 6.6.8: Arrow functions that sum two numbers and square a number.

```
                              3 + 6 = 9
let findSum = (a, b) => a + b;

let sum = findSum(3, 6);
console.log(sum);
```

```
9
25
```



```
                          5 * 5 = 25
let square = x => x * x;

console.log(square(5));
```

Animation content:

The following code is displayed:
let findSum = (a, b) => a + b;

let sum = findSum(3, 6);
console.log(sum);

let square = x => x * x;

console.log(square(5));

Step 5 runs let sum = findSum(3, 6); console.log(sum); and 9 is displayed in the console.
Step 6 runs console.log(square(5)); and 25 is displayed in the console.

Animation captions:

1. An arrow function may be assigned to a variable, just like a function expression.
2. The function parameters are listed in parenthesis to the left of the arrow =>.
3. An expression listed by itself is the value returned by the arrow function.
4. An arrow function is called the same as any other function. The arguments 3 and 6 are assigned to parameters a and b.
5. The arrow function returns the sum of a and b, which is 9.
6. An arrow function with only one parameter does not require parentheses around the one parameter.

PARTICIPATION ACTIVITY 6.6.9: Arrow functions.

1) Complete the arrow function.

```
let max =          =>
a > b ? a : b;
```

Check Show answer

2) Complete the arrow function.

```
let countCapitals =
                   => {
   let count = 0;
   for (let i = 0; i <
str.length; i++) {
      let ch =
str.charAt(i);
      if (ch >= 'A' && ch
<= 'Z') {
         count++;
      }
   }
   return count;
}
```

Check Show answer

3) Convert `isEven()` into an equivalent arrow function.

```
function isEven(num) {
   return num % 2 === 0;
}
```

```
let isEven = num =>
                    ;
```

Check Show answer

CHALLENGE ACTIVITY 6.6.1: Functions.

550544.4142762.qx3zqy7

Start

Call the favoriteColor function with argument "green".

```
function favoriteColor(myColor) {
   console.log("My favorite color is " + myColor + ".");
}

/* Your solution goes here */

```

1 2 3 4

Check Next

Exploring further:

- Functions (MDN)

6.7 Scope and the global object

The var keyword and scope

In addition to declaring variables with `let`, a variable can be declared with the ***var*** keyword. Ex: `var x = 6;` declares the variable `x` with an initial value of 6. When JavaScript was first created, `var` was the only way to declare a variable. The `let` keyword was added to JavaScript in 2015.

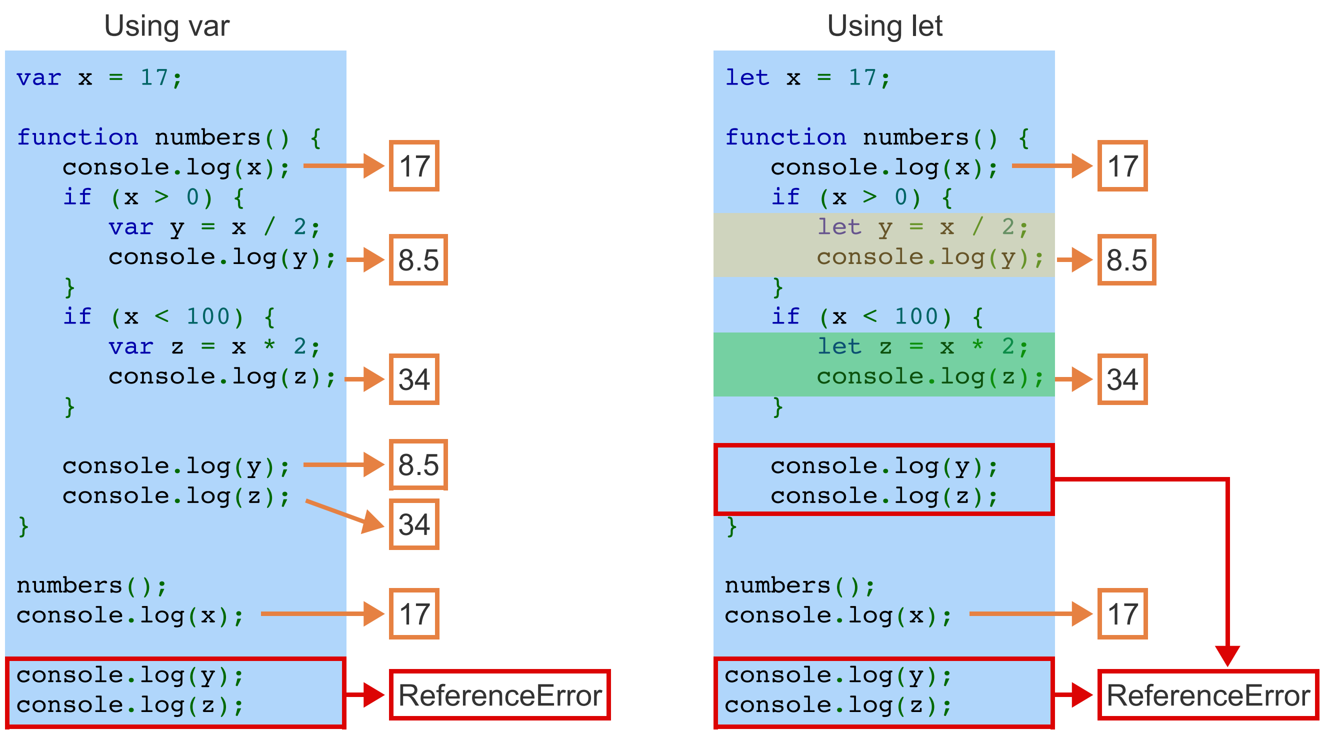
Both `let` and `var` declare variables but with differing scope. A JavaScript variable's ***scope*** is the context in which the variable can be accessed.

A variable declared inside a function has ***local scope***, so only the function that defines the variable has access to the ***local variable***. A variable declared outside a function has ***global scope***, and all functions have access to a ***global variable***.

A variable declared inside a function with `var` has ***function scope***: the variable is accessible anywhere within the function, but not outside. A variable declared inside a function with `let` has ***block scope***: the variable is accessible only within the enclosing pair of braces.

A variable declared using `var` or `let` that is not inside a function creates a global variable that is accessible from anywhere in the code.

PARTICIPATION ACTIVITY 6.7.1: var vs. let scoping.


Using var

```
var x = 17;

function numbers() {
   console.log(x);
   if (x > 0) {
      var y = x / 2;
      console.log(y);
   }
   if (x < 100) {
      var z = x * 2;
      console.log(z);
   }

   console.log(y);
   console.log(z);
}

numbers();
console.log(x);

console.log(y);
console.log(z);
```

Using let

```
let x = 17;

function numbers() {
   console.log(x);
   if (x > 0) {
      let y = x / 2;
      console.log(y);
   }
   if (x < 100) {
      let z = x * 2;
      console.log(z);
   }

   console.log(y);
   console.log(z);
}

numbers();
console.log(x);

console.log(y);
console.log(z);
```

Animation content:

The following code is displayed to show the scope of variables using var declarations:

```
// Outside a function so declared with global scope
var x = 17;

function numbers() {
// x has global scope so x is accessible anywhere
 console.log(x);
 if (x > 0) {
   // y is declared inside a function and inside an if block. y has function scope because y was
declared with var
   var y = x / 2;
   console.log(y);
 }
 if (x < 100) {
   // Same case as y, z is declared in a function using var. z has function scope.
   var z = x * 2;
   console.log(z);
 }
// y and z are accessible. y and z both have function scope and are accessed and declared in the
same function
 console.log(y);
 console.log(z);
}

numbers();
// x has global scope so x is accessible
console.log(x);

// y and z are accessed outside of the function y and z were declared in. A ReferenceError is
thrown.
console.log(y);
console.log(z);
```

The following code is displayed to show the scope of variables using let declarations:

```
// Outside a function so declared with global scope
let x = 17

function numbers() {
```

```
// x has global scope so x is accessible anywhere
 console.log(x);
 if (x > 0) {
      // y is declared inside a function and inside an if block. y has block scope because y was
declared with let
    let y = x / 2;
    console.log(y);
 }
 if (x < 100) {
     // Same case as y, z is declared in a function using var. z has block scope
    let z = x * 2;
    console.log(z);
 }
// y and z have block scope and are accessed outside the block y and z were declared in. A
ReferenceError is thrown.
 console.log(y);
 console.log(z);
}

numbers();
// x has global scope so x is accessible
console.log(x);
// y and z are once again accessed outside of the block y and z were declared in. A
ReferenceError is thrown.
console.log(y);
console.log(z);
```

Animation captions:

1. var x = 17; declares x with global scope. x is accessible everywhere, so each console.log(x) statement logs x as 17.
2. The var y declaration exists inside the numbers() function. So both console.log(y) statements inside the function log y as 8.5.
3. Similarly, the var z statement is inside the function, so both console.log(z) statements inside the function log z as 34.
4. y and z are not accessible outside the numbers() function. The console.log() statements that exist outside the function throw a ReferenceError when executed.
5. Code that uses let instead of var has similar behavior for the global variable x.
6. The first log statement for y is in y's scope (yellow), and the first log statement for z is in z's

scope (green). So, 8.5 and 34 are logged.

7. All remaining calls to log y or z are out of scope and throw a ReferenceError.

PARTICIPATION ACTIVITY 6.7.2: Local and global variables.

Refer to the code below.

```
function multiplyNumbers(x, y) {
   var answer = x * y;
   return answer;
}

var z = multiplyNumbers(2, 3);
console.log(answer);
```

1) The `answer` variable has _____ scope.

- ○ global
- ○ local

2) The `z` variable has _____ scope.

- ○ global
- ○ local

3) The `console.log(answer);` line _____.

- ○ logs 6
- ○ logs undefined
- ○ throws a ReferenceError

PARTICIPATION ACTIVITY 6.7.3: var vs. let scoping.

1) In the code below, which variables are in scope on the blank line?

```
function oneToTen() {
   let a = 1;
   for (var i = a; i <= 10;
i++) {
      console.log(i);
      _____
   }
}
```

- ○ `a` only
- ○ `i` only
- ○ both `a` and `i`

2) After calling functions `f1()` and `f2()` in the code below, which variable has global scope?

```
function f1() {
   let x = 100;
}

function f2() {
   var y = 200;
}
```

- ○ `x`
- ○ `y`
- ○ neither

3) On the blank line in the code below, variable `k` _____.

```
function sumOfSquares() {
   let sum = 0;
   for (let i = 1; i < 5; i++)
{
      let j = i * i;
      sum += j;
      var k = sum;
   }
   _____
}
```

- ○ is out of scope and not accessible
- ○ is in scope and has a value of 30
- ○ is in scope, but has an undefined value

Global variables and the global object

Before developer code is run, JavaScript implementations create ***the global object***: an object that stores certain global variables, functions, and other properties. When running JavaScript code in a web browser, global variables are usually assigned as properties to the global `window` object. Therefore, a global variable called `test` is accessible as `window.test`.

Developers must be careful when assigning global variables, because a global variable could replace an existing window property. Ex: `window.location` contains the URL the browser is displaying. Assigning `location = "Texas"` causes the web browser to attempt to load a web page with the URL "Texas", which likely does not exist.

Three cases exist when assigning to a global variable X:

- X has been declared with `var`, in which case a property named "X" is added to the global object.
- X has been declared with `let`, in which case a property named "X" is not added to the global object, but X is still accessible from anywhere in the code.
- X has not been declared with `var` or `let`, in which case the variable becomes a property of the global object, even if assigned to inside a function.

Figure 6.7.1: Example with accidental global variable.

```
function calculateTax(total) {
   // Missing "var" so tax becomes a global variable!
   tax = total * 0.06;
   return tax;
}

var totalTax = calculateTax(10);

// tax is accessible because tax is global
console.log(tax);
```

```
0.6
```

Good practice is to always declare variables used in functions with `var` or `let`, so the variables do not become global.

PARTICIPATION ACTIVITY 6.7.4: Variable scope and functions.

1) What is output to the console?

```
function addNumber(x) {
   sum += x;
}

let sum = 0;
addNumber(2);
addNumber(5);
console.log(sum);
```

Check Show answer

2) What is output to the console?

```
function
multiplyNumbers(x, y) {
   answer = x * y;
   return answer;
}

var z =
multiplyNumbers(2, 3);
console.log(answer);
```

Check Show answer

3) If `window` is the global object, what is the value of `window.result` after running the following code?

```
function
subtractNumbers(x, y) {
   result = x - y;
}

var a =
subtractNumbers(7, 6);
var b =
subtractNumbers(11, 3);
var c =
subtractNumbers(9, 1);
```

Check Show answer

Exploring further:

- var (MDN)
- Global object (MDN)

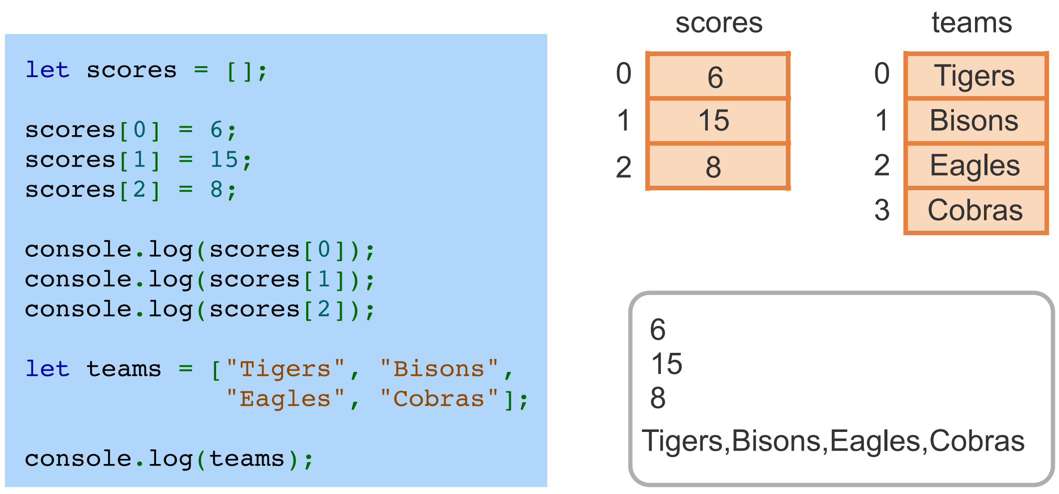
6.8 Arrays

Array introduction

An ***array*** is an ordered collection of values called ***elements***. Each array element is stored in a numeric location called an ***index***. An array is initialized by assigning an array variable with brackets `[ ]` containing comma-separated values.

Array elements may be of the same type or different types. Arrays increase in size as elements are added and decrease as elements are removed.

PARTICIPATION ACTIVITY 6.8.1: Initializing and displaying array elements.



```
let scores = [];

scores[0] = 6;
scores[1] = 15;
scores[2] = 8;

console.log(scores[0]);
console.log(scores[1]);
console.log(scores[2]);

let teams = ["Tigers", "Bisons",
             "Eagles", "Cobras"];

console.log(teams);
```

```
6
15
8
Tigers,Bisons,Eagles,Cobras
```

Animation content:

The following code is shown below:
let scores = [];

scores[0] = 6;
scores[1] = 15;

scores[2] = 8;

console.log(scores[0]);
console.log(scores[1]);
console.log(scores[2]);

let teams = ["Tigers", "Bisons",
            "Eagles", "Cobras"];

console.log(teams);

Step 2 runs the following code:
scores[0] = 6;
scores[1] = 15;
scores[2] = 8;.
The memory allocated to the array scores grows as more indices in the array get assigned values. Values 6, 15, and 8 are shown stored under the variable scores at indices 0, 1 and 2 respectively.
In step 3, the following lines of code are run:
console.log(scores[0]);
console.log(scores[1]);
console.log(scores[2]);
6, 15, and 8 are displayed in the console.
In step 4, a new array teams is allocated memory to store 4 strings. The 4 strings are assigned to indices in the array in the order they are listed.
In step 5, the array teams is logged to the console. Tigers,Bisons,Eagles,Cobras is displayed in console

Animation captions:

1. An empty array called "scores" is declared with [ ].
2. Three elements are added to the scores array at indexes 0, 1, and 2.
3. The three elements are output to the console.
4. The teams array is initialized with four elements.
5. All four elements are output to the console separated by commas.

PARTICIPATION ACTIVITY

6.8.2: Initializing and displaying array elements.

1) Initialize `names` to an empty array.

```
let names = ;
```

Check Show answer

2) How many elements does `names` have?

```
let names = [];
names[0] = "Sue";
names[1] = "Bob";
names[2] = "Jeff";
```

Check Show answer

3) What is output to the console?

```
let names = [];
names[0] = "Sue";
names[1] = "Bob";
names[2] = "Jeff";
console.log(names[0] +
names[1]);
```

Check Show answer

4) What is the largest number `nameIndex` can be to output a name?

```
let names = ["Maria",
"Braden", "Jaden", "Aditya"];
console.log(names[nameIndex]);
```

Check Show answer

5) What is output to the console?

```
let nums = [5, 2, 9, 3];
nums[0] = nums[1] +
nums[2];
console.log(nums[0]);
```

Check Show answer

6) Write the simplest code to output all elements in `pets`.

```
let pets = ["cats",
"dogs", "fish"];
console.log(          );
```

Check Show answer

Adding and removing array elements

An array is an ***Array object*** that defines numerous methods for manipulating arrays. A ***method*** is a function that is attached to an object and operates on data stored in the object. Methods are called by prefacing the method with the object. Ex: `myArray.method();`.

Table 6.8.1: Array methods for adding and removing array elements.

Method	Description	Example

push(value)	Adds a value to the end of the array	`let nums = [2, 4, 6];`
		`nums.push(8);  // nums = [2, 4, 6, 8]`

pop()	Removes the last array element and returns the element	`let nums = [2, 4, 6];`
		`let x = nums.pop();  // returns 6, nums = [2, 4]`

unshift(value)	Adds a value to the beginning of the array	`let nums = [2, 4, 6];`
		`nums.unshift(0);  // nums = [0, 2, 4, 6]`

shift()	Removes the first array element and returns the element	`let nums = [2, 4, 6];`
		`let x = nums.shift();  // returns 2, nums = [4, 6]`

	Adds or removes elements	

splice(startingIndex, numElemToDelete, valuesToAdd)	from anywhere in the array and returns the deleted elements (if any)	`let nums = [2, 4, 6, 8, 10];`
		
		`// Deletes all elements from index 3 to the end`
		`nums.splice(3);    // nums = [2, 4, 6]`
		
		`// Deletes 2 elements starting at index 0`
		`nums.splice(0, 2);    // nums = [6]`
		
		`// Adds 3, 5 starting at index 0`
		`nums.splice(0, 0, 3, 5);    // nums = [3, 5, 6]`
		
		`// Adds 7, 9, 11 starting at index 2`
		`nums.splice(2, 0, 7, 9, 11);    // nums = [3, 5, 7, 9, 11, 6]`

PARTICIPATION ACTIVITY 6.8.3: Adding and removing array elements.

The six individuals in the `line` array are waiting in line. Write the JavaScript code to add or remove elements to/from the array to simulate the following events:

1. The person at the front of the line (index 0) leaves the line (`shift`).
2. The person at the end of the line cuts in front of the person at the front of the line (`pop` and `unshift`).
3. Two new people named "Poe" and "Snoke" cut into line behind the second person in line (`splice`).
4. The fifth person in line leaves the line (`splice`).
5. A new person named "Han" enters the back of the line (`push`).

Finally, display the contents of the `line` array to view the new line occupants. A correct solution will show: Leia, Finn, Poe, Snoke, Maz, Han.

```
// People waiting in line (Kylo is in front, Leia at the end)
let line = ["Kylo", "Finn", "Rey", "Maz", "Leia"];

// Show entire line
console.log(line);

```

Run JavaScript Reset code

Your console output

```
Kylo,Finn,Rey,Maz,Leia
```

▶ View solution

Looping through an array

The array property ***length*** contains the number of elements in the array. The `length` property is helpful for looping through an array using a for loop.

Figure 6.8.1: Looping through an array with a for loop.

```
let groceries = ["bread", "milk", "peanut butter"];

// Display all elements in groceries array
for (i = 0; i < groceries.length; i++) {
   console.log(i + " - " + groceries[i]);
}
```

```
0 - bread
1 - milk
2 - peanut butter
```

The ***for-of loop*** is a simplified for loop that loops through an entire array. The array name is placed after the `of` keyword in a for-of loop. Each time through the loop, the next array element is assigned to the variable in front of the `of` keyword.

Figure 6.8.2: Looping through an array with a for-of loop.

```
let groceries = ["bread", "milk", "peanut butter"];

// Display all elements in groceries array
for (let item of groceries) {
   console.log(item);
}
```

```
bread
milk
peanut butter
```

The `Array` method ***forEach()*** also loops through an array. The `forEach()` method takes a function as an argument. The function is called for each array element in order, passing the element and the element index to the function.

Figure 6.8.3: Looping through an array with the forEach() method.

```
let groceries = ["bread", "milk", "peanut butter"];

// Display all elements in groceries array
groceries.forEach(function(item, index) {
   console.log(index + " - " + item);
});
```

```
0 - bread
1 - milk
2 - peanut butter
```

PARTICIPATION ACTIVITY 6.8.4: Looping through an array.

1) What is `autos.length`?

```
let autos = ["Chevrolet",
"Dodge", "Ford", "Ram"];
```

- 0
- 3
- 4

2) What is output to the console?

```
let autos = ["Chevrolet",
"Dodge", "Ford", "Ram"];
for (i = 0; i < 2; i++) {
   console.log(autos[i]);
}
```

- Chevrolet, Dodge
- Chevrolet, Dodge, Ford
- Chevrolet, Dodge, Ford, Ram

3) What is output to the console?

```
let autos = ["Chevrolet",
"Dodge", "Ford", "Ram"];
for (i = 0; i < autos.length;
i++) {
   if (i % 2 == 0) {
      console.log(autos[i]);
   }
}
```

- ○ Chevrolet, Dodge, Ford, Ram
- ○ Chevrolet, Ford
- ○ Dodge, Ram

4) What is output to the console?

```
let autos = ["Chevrolet",
"Dodge", "Ford", "Ram"];
autos.forEach(function(item,
index) {
   if (index % 3 == 0) {
      console.log(item);
   }
});
```

- ○ Chevrolet, Dodge, Ford, Ram
- ○ Chevrolet, Ford
- ○ Chevrolet, Ram

5) What is missing in the for-of loop to display all the elements in the `autos` array?

```
let autos = ["Chevrolet",
"Dodge", "Ford", "Ram"];
for (____) {
   console.log(auto);
}
```

- ○ `let auto of autos`
- ○ `let item of autos`
- ○ `let autos of auto`

6) Which loop is best for looping through an array in reverse order (from last element to first element)?

- ○ for loop
- ○ for-of loop
- ○ forEach() loop

PARTICIPATION ACTIVITY 6.8.5: Practice looping.

Duke and North Carolina have a famous basketball rivalry dating back to 1920. The number of points each team has scored in head-to-head competition over five years is provided in the `dukeScores` and `ncScores` arrays. Ex: North Carolina won the first game 76-72 since `dukeScores[0]` is 72 and `ncScores[0]` is 76.

Write a for loop that examines the `dukeScores` and `ncScores` arrays and places "D" in the `winningTeam` array if Duke won or "N" if North Carolina won, for every game. Ex: `winningTeam[0]` should be "N" because North Carolina won 76-72, and `winningTeam[1]` should be "D" because Duke won 74-73.

Then display the contents of the `winningTeam` array using a for-of or forEach() loop.

```
let dukeScores  = [72, 74, 84, 92, 93, 66, 69, 73, 70, 85, 75, 67, 79
let ncScores    = [76, 73, 77, 90, 81, 74, 53, 68, 88, 84, 58, 81, 73
let winningTeam = [];

// Who won the first game?
if (dukeScores[0] > ncScores[0]) {
   console.log("Duke won " + dukeScores[0] + "-" + ncScores[0] + ".")
}
else {
   console.log("North Carolina won " + ncScores[0] + "-" + dukeScores
}
```

Run JavaScript Reset code

Your console output

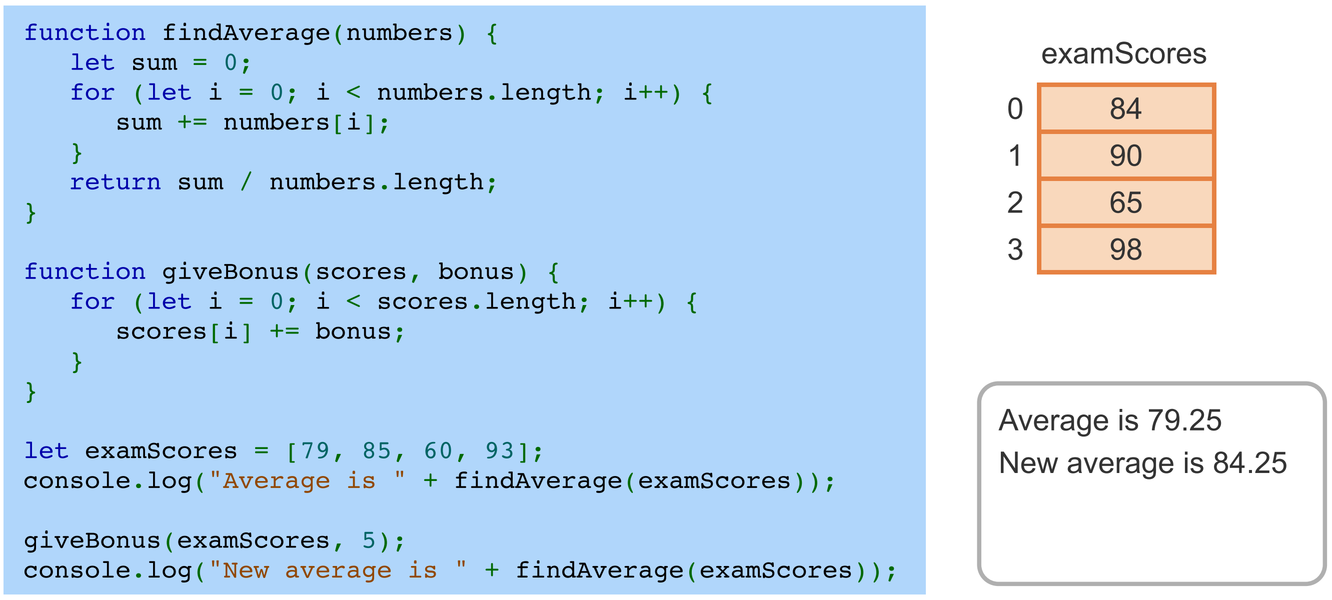
```
North Carolina won 76-72.
```

▶ View solution

Passing arrays to functions

An array can be passed to a function as an argument. A function may modify an array argument's elements.

PARTICIPATION ACTIVITY 6.8.6: Passing arrays to functions.



```
function findAverage(numbers) {
   let sum = 0;
   for (let i = 0; i < numbers.length; i++) {
      sum += numbers[i];
   }
   return sum / numbers.length;
}

function giveBonus(scores, bonus) {
   for (let i = 0; i < scores.length; i++) {
      scores[i] += bonus;
   }
}

let examScores = [79, 85, 60, 93];
console.log("Average is " + findAverage(examScores));

giveBonus(examScores, 5);
console.log("New average is " + findAverage(examScores));
```

```
Average is 79.25
New average is 84.25
```

Animation content:

The following code is displayed:

```
function findAverage(numbers) {
  let sum = 0;
  for (let i = 0; i < numbers.length; i++) {
    sum += numbers[i];
  }
  return sum / numbers.length;
}

function giveBonus(scores, bonus) {
  for (let i = 0; i < scores.length; i++) {
    scores[i] += bonus;
  }
}
```

```
let examScores = [79, 85, 60, 93];
console.log("Average is " + findAverage(examScores));

giveBonus(examScores, 5);
console.log("New average is " + findAverage(examScores));
```

The findAverage() parameter numbers and the giveBonus() parameter scores both reference the same memory used by the examScore argument. The function findAverage() is able to get values stored in the indices of examScores. The function giveBonus() is able to change values in examScores because the parameter scores reference the same block of memory as examScores. The parameter stores a copy of the reference to examScores rather than making a copy of the array examScores.

Animation captions:

1. The examScores array is passed to the findAverage() function.
2. findAverage's numbers parameter refers to the same array as examScores.
3. findAverage() sums the scores in the numbers array, divides the sum by 4, and returns the result.
4. examScores is passed to the giveBonus() function. The scores parameter and examScores argument refer to the same array.
5. The for loop adds 5 to each element in scores, which also changes examScores.
6. Since examScores is changed, the new average is 5 points higher than before.

PARTICIPATION ACTIVITY 6.8.7: Passing arrays to functions.

Match the value to the corresponding array element after the code below executes.

```
function makeOdd(numbers) {
   for (let i = 0; i < numbers.length; i++) {
      if (numbers[i] % 2 == 0) {
         numbers[i]++;
      }
   }
}

let ages = [10, 3, 5, 8];
makeOdd(ages);
```

If unable to drag and drop, refresh the page.

3 9 5 11

	ages[0]
	ages[1]
	ages[2]
	ages[3]

Reset

Searching an array

The array methods ***indexOf()*** and ***lastIndexOf()*** search an array and return the index of the first found value or -1 if the value is not found. `indexOf()` searches from the beginning of the array to the end. `lastIndexOf()` searches from the end of the array to the beginning. Both functions take two arguments:

1. `searchValue` - The value to search for
2. `startingPosition` - Optional argument that indicates the index at which the search should begin (default is 0 for `indexOf()` and `array.length - 1` for `lastIndexOf()`)

Figure 6.8.4: Searching for array elements.

```
let scores = [80, 92, 75, 64, 88,
92];

s = scores.indexOf(92);          // 1
s = scores.indexOf(92, 2);       // 5
s = scores.indexOf(100);         //
-1
s = scores.lastIndexOf(92);      // 5
s = scores.lastIndexOf(92, 4);   // 1
s = scores.lastIndexOf(50);      //
-1
```

PARTICIPATION ACTIVITY 6.8.8: Searching an array.

Refer to the `artists` array.

```
let artists = ["Raphael", "Titian", "Masaccio", "Botticelli", "Titian"];
```

1) `artists.indexOf("Botticelli")` returns 3.

- ○ True
- ○ False

2) `artists.lastIndexOf("Titian")` returns 1.

- ○ True
- ○ False

3) `artists.indexOf("Michaelangelo")` returns NaN.

- ○ True
- ○ False

PARTICIPATION ACTIVITY 6.8.9: Practice searching an array.

The `validCredentials()` function contains two parallel arrays of usernames and passwords. Modify `validCredentials()` to use the `indexOf()` method to search the `usernames` array for the given `enteredUsername`. If the username is found, the same location in the `passwords` array should contain the `enteredPassword`. Return `true` if the passwords are equal, `false` otherwise. `validCredentials()` should also return `false` if the given username was not found.

```
// Return true if the given username and password are in the database
// false otherwise.
function validCredentials(enteredUsername, enteredPassword) {

   // Database of usernames and passwords
   let usernames = ["smith",  "tron",      "ace",      "ladyj",    "c
   let passwords = ["qwerty", "EndOfLine", "year1942", "ladyj123", "F

   // Search the usernames array for enteredUsername

   // Only return true if the enteredUsername is in username, and the
   // same location in passwords is enteredPassword
   return true;
}


```

Run JavaScript Reset code

Your console output

```
Login for ladyj: true
Login for ace: true
Login for jake: true
```

▶ View solution

Sorting an array

The array method ***sort()*** sorts an array in ascending (increasing) order. `sort()`'s default behavior is to sort each element as a string using the string's Unicode values. Sorting by Unicode values may yield unsatisfactory results for arrays that store numbers. Ex: 10 is sorted before 2 because "10" is < "2" when comparing the Unicode values of "1" to "2".

The `sort()` method can sort elements in other ways by passing a comparison function to `sort()`. The comparison function returns a number that helps `sort()` determine the sorting order of the array's elements:

- Returns a value < 0 if the first argument should appear before the second argument.
- Returns a value > 0 if the first argument should appear after the second argument.
- Returns 0 if the order of the first and second arguments does not matter.

Figure 6.8.5: Sorting an array of numbers.

```
let numbers = [200, 30, 1000, 4];

// Sort based on Unicode values: [1000, 200, 30, 4]
numbers.sort();

// Sort numbers in ascending order: [4, 30, 200,
1000]
numbers.sort(function(a, b) {
   return a - b;
});
```

PARTICIPATION ACTIVITY 6.8.10: Sorting arrays.

1) What is output to the console?

```
let names = ["Sue",
"Bob", "Jeff"];
names.sort();
console.log(names[0]);
```

Check Show answer

2) What is output to the console?

```
let names = ["Sue",
"Bob", "Jeff"];
names.sort(function(a, b)
{
   if (a > b) {
      return -1;
   }
   else if (a < b) {
      return 1;
   }
   return 0;
});
console.log(names[0]);
```

Check Show answer

3) What is output to the console?

```
let totals = [90, 4, 21,
34];
totals.sort();
console.log(totals[0]);
```

Check Show answer

4) What is output to the console?

```
let totals = [99, 4, 250,
38];
totals.sort(function(a,
b) {
   return b - a;
});
console.log(totals[0]);
```

Check Show answer

CHALLENGE ACTIVITY | 6.8.1: Arrays.

550544.4142762.qx3zqy7

Exploring further:

- Array object (MDN)

6.9 Objects

Objects and properties

An ***object*** is an unordered collection of properties. An object ***property*** is a name-value pair, where the name is an identifier and the value is any data type. Objects are often defined with an object literal. An ***object literal*** (also called an ***object initializer***) is a comma-separated list of property name and value pairs.

PARTICIPATION ACTIVITY | 6.9.1: Creating an object with an object literal.

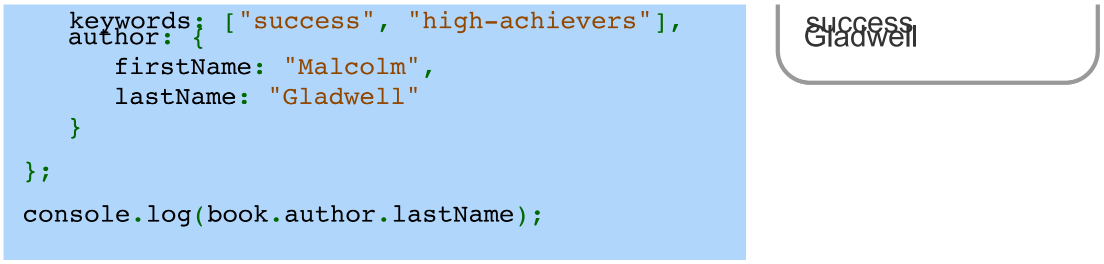
```
let book = {};

book = {
   title: "Outliers",
   published: 2011,
   keywords: ["success", "high-achievers"]
};

console.log(book.title);
console.log(book.keywords[0]);


book = {
   title: "Outliers",
   published: 2011,
```

```
Outliers
```



```
   keywords: ["success", "high-achievers"],
   author: {
      firstName: "Malcolm",
      lastName: "Gladwell"
   }
};
console.log(book.author.lastName);
```

```
success
Gladwell
```

Animation content:

The following code is displayed:
let book = {};

book = {
  title: "Outliers",
  published: 2011,
  keywords: ["success", "high-achievers"]
};

console.log(book.title);
console.log(book.keywords[0]);

book = {
  title: "Outliers",
  published: 2011,
  keywords: ["success", "high-achievers"],
  author: {
    firstName: "Malcolm",
    lastName: "Gladwell"
  }
};

console.log(book.author.lastName);

Step 1 runs the code: let book = {};.
Step 2 runs the code: book = {
  title: "Outliers",
  published: 2011,
  keywords: ["success", "high-achievers"]

};.
Step 3 runs the code: console.log(book.title);
console.log(book.keywords[0]);.
Outliers and success are displayed in console, each on new lines.
Step 4 runs the code:
book = {
  title: "Outliers",
  published: 2011,
  keywords: ["success", "high-achievers"],
  author: {
    firstName: "Malcolm",
    lastName: "Gladwell"
  }
};.
Step 5 runs the last line of code and Gladwell is displayed on a new line in the console

Animation captions:

1. book is assigned an empty object literal.
2. book is assigned an object literal with three properties: title, published, keywords.
3. Display the title and first keyword of the book object.
4. book is assigned an object literal with an embedded object literal that is assigned to the author property.
5. Display the last name of the book's author.

PARTICIPATION ACTIVITY 6.9.2: Accessing object properties.

Refer to the `book` object below.

```
let book = {
   title: "Hatching Twitter",
   published: 2013,
   keywords: ["origins", "betrayal", "social media"],
   author: {
      firstName: "Nick",
      lastName: "Bilton"
   }
};
```

1) Which statement changes the published year to 2014?

- ○ `book.Published = 2014;`
- ○ `book.published = 2014;`
- ○ `book.published : 2014;`

2) Which statement adds a new property called "isbn" with the value "1591846013"?

- ○ `book.isbn = "1591846013";`
- ○ `isbn = "1591846013";`
- ○ `book.isbn("1591846013");`

3) What statement replaces "Nick" with "Jack"?

- ○ `book.author = "Jack";`
- ○ `book.author.firstName = "Jack";`
- ○ `book.author.lastName = "Jack";`

4) What is missing from the code below to remove "social media" from the book's keywords? The array method `pop()` removes the last element from an array.

```
_______.pop();
```

- ○ `keywords`
- ○ `book.keywords[2]`
- ○ `book.keywords`

Methods

Assigning an object's property name with an anonymous function creates a method. Methods access the object's properties using the keyword `this`, followed by a period, before the property name. Ex: `this.someProperty`.

Figure 6.9.1: Defining a method in an object literal.

```
let book = {
   title: "Quiet",
   author: {
      firstName: "Susan",
      lastName: "Cain"
   },

   // Define a method
   getAuthorName: function() {
      return this.author.firstName + " " +
this.author.lastName;
   }
};

// Call a method that returns "Susan Cain"
let name = book.getAuthorName();
```

Figure 6.9.2: Defining a method for an existing object.

```
let book = {
   title: "Quiet",
   author: {
      firstName: "Susan",
      lastName: "Cain"
   }
};

// Define a method
book.getAuthorName = function() {
   return this.author.firstName + " " +
this.author.lastName;
};

// Call a method that returns "Susan Cain"
let name = book.getAuthorName();
```

PARTICIPATION ACTIVITY 6.9.3: Object methods.

Refer to the above figures.

1) A method may be defined inside or outside an object literal.

- ○ True
- ○ False

2) The method below outputs "I'm reading 'Quiet'.".

```
book.read = function() {
   console.log("I'm reading '"
+ title + "'.");
};
```

- ○ True
- ○ False

3) The method below creates a new object property.

```
book.assignMiddleInitial =
function(middleInitial) {
   this.author.middleInitial =
middleInitial;
};

book.assignMiddleInitial("H");
```

- ○ True
- ○ False

PARTICIPATION ACTIVITY 6.9.4: Practice creating objects and methods.

Create an object called `game` that represents a competition between two opponents or teams. Add the following properties to `game`, and assign any value to each property:

1. `winner` - An object with properties `name` and `score`

2. `loser` - An object with properties `name` and `score`

Add the following methods to `game`:

1. `getMarginOfVictory()` - Returns the difference between the winner's score and the loser's score
2. `showSummary()` - Outputs to the console the winner's name and score, the loser's name and score, and the margin of victory

Call the two methods to verify the methods work correctly. Example output:

```
Broncos: 24
Panthers: 10
Margin of victory: 14
```

Accessor properties

An object property may need to be computed when retrieved, or setting a property may require executing some code to perform data validation. The `get` and `set` keywords define getters and setters for a property.

- A ***getter*** is a function that is called when an object's property is retrieved. Syntax to define a getter: `get property() { return someValue; }`.
- A ***setter*** is a function that is called when an object's property is set to a value. Syntax to define a setter: `set property(value) { ... }`.

An ***accessor property*** is an object property that has a getter or a setter or both.

Figure 6.9.3: Defining an accessor property called 'area'.

```
let rectangle = {
   width: 5,
   height: 8,
   get area() {
      return this.width * this.height;
   },
   set area(value) {
      // Set width and height to the square root of the value
      this.width = Math.sqrt(value);
      this.height = this.width;
   }
};

let area = rectangle.area;      // Calling getter returns 40
rectangle.area = 100;           // Calling setter sets width and height
to 10
console.log(rectangle.width);   // 10
```

PARTICIPATION ACTIVITY 6.9.5: Accessor properties.

Refer to the `game` object.

```
let game = {
   firstOpponent: "Serena Williams",
   firstOpponentScore: 2,
   secondOpponent: "Garbine Muguruza",
   secondOpponentScore: 0,
   get winner() {
      if (this.firstOpponentScore > this.secondOpponentScore) {
         return this.firstOpponent;
      }
      else if (this.secondOpponentScore > this.firstOpponentScore) {
         return this.secondOpponent;
      }
      else {
         return "Tie";
      }
   }
};
```

1) The code below outputs "Serena Williams".

```
console.log(game.winner());
```

- True
- False

2) The code below outputs "Maria Sharapova" .

```
game.winner = "Maria
Sharapova";
console.log(game.winner);
```

- True
- False

3) The `matchDate` setter below sets the `date` property to the given `value`.

```
let game = {
   ...
   date: "",
   set matchDate(value) {
      date = value;
   },
   ...
};
```

- True
- False

4) What sets the game's match date to the Date object?

```
let game = {
   ...
   date: "",
   set matchDate(value) {
      this.date = value;
   },
   ...
};

// Wimbledon 2016 women's
championship
let champDate = new Date(2016,
5, 9);
```

○ game.matchDate = champDate;

○ game.matchDate(champDate);

PARTICIPATION ACTIVITY 6.9.6: Practice creating accessor properties.

The `musicQueue` object contains a `songs` property listing all the songs in the music queue. Add an accessor property called "next" with the following functions::

- Getter - Returns the song in the `songs` array at index `nextSong`. Then increments `nextSong` by one so the next song in the queue will be retrieved the next time the getter is accessed. If `nextSong` is beyond the boundaries of the `songs` array, `nextSong` should be assigned 0.
- Setter - Sets `nextSong` to the given value. If the value is outside the `songs` array's bounds, `nextSong` should be assigned 0.

If the `next` property is implemented correctly, the for loop under the `musicQueue` will display each song three times. The code under the for loop tests the setter and should display the song in comments.

Passing objects to functions

JavaScript data types can be divided into two categories: primitives and references.

1. A ***primitive*** is data that is not an object and includes no methods. Primitive types include: boolean, number, string, null, and undefined.
2. A ***reference*** is a logical memory address. Only objects are reference types.

Assigning a variable with a primitive creates a copy of the primitive. Ex: If y is 2, then `x = y;` means x is assigned with a copy of y. Assigning a variable with a reference creates a copy of the reference. Ex: If y refers to an object, then `x = y;` means x is assigned with a copy of y's reference. Both x and y refer to the same object.

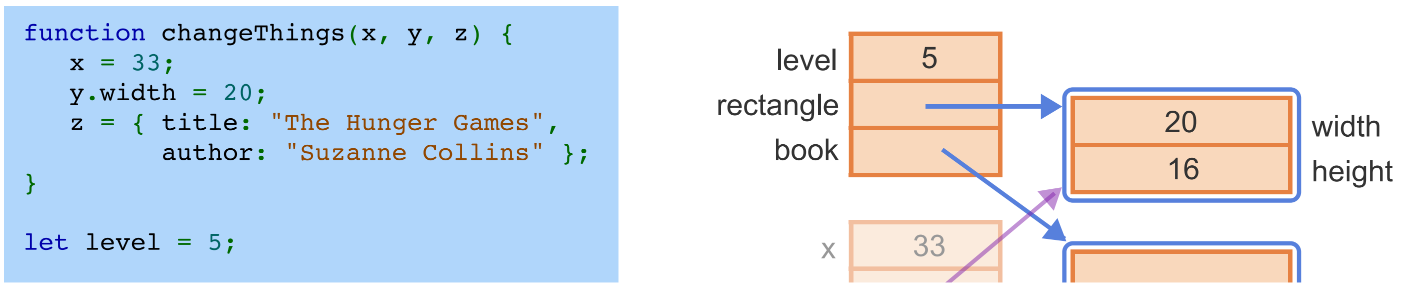
When a primitive is passed to a function, the parameter is assigned a copy of the argument. Changing the parameter does not change the argument.

When an object is passed to a function, the parameter is also assigned a copy of the argument. However, the parameter and argument are both a reference to the same object. Changing the parameter reference to a new object does not change the argument, but changing the parameter's properties *does* change the argument's properties.

Primitive wrappers

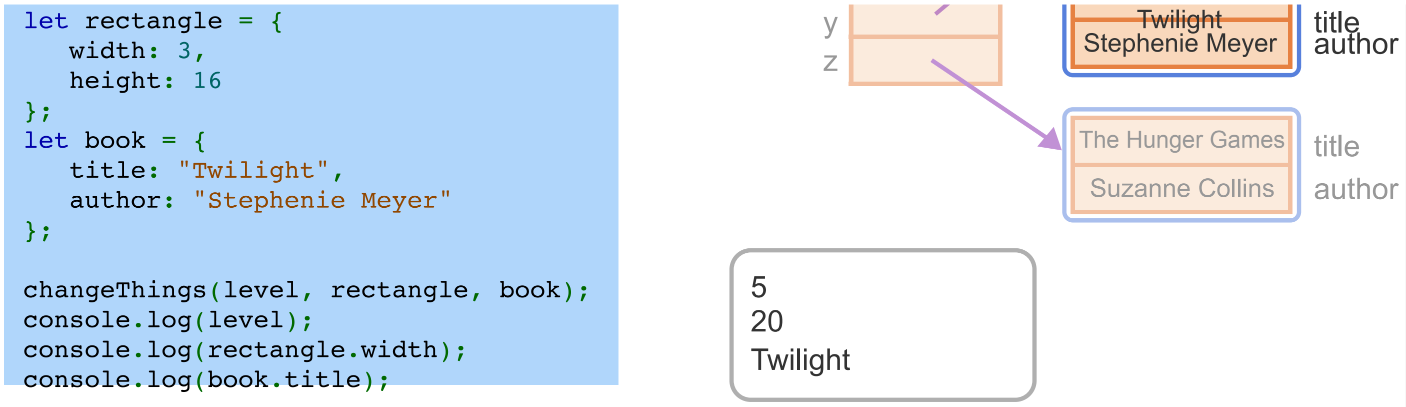
All primitives, except for null and undefined, have equivalent objects that wrap the primitive values and provide methods for interacting with the primitives. Ex: A string primitive has a `String` class that provides various methods for manipulating a string. Calling `"abc".toUpperCase()` converts the primitive string into a `String` object, calls the method, and returns the string primitive "ABC".

PARTICIPATION ACTIVITY 6.9.7: Passing primitives and references to a function.



```
function changeThings(x, y, z) {
   x = 33;
   y.width = 20;
   z = { title: "The Hunger Games",
         author: "Suzanne Collins" };
}

let level = 5;
```



```
let rectangle = {
   width: 3,
   height: 16
};
let book = {
   title: "Twilight",
   author: "Stephenie Meyer"
};

changeThings(level, rectangle, book);
console.log(level);
console.log(rectangle.width);
console.log(book.title);
```

```
5
20
Twilight
```

Animation content:

The following code is displayed:

```
function changeThings(x, y, z) {
  x = 33;
  y.width = 20;
  z = { title: "The Hunger Games",
        author: "Suzanne Collins" };
}

let level = 5;
let rectangle = {
  width: 3,
  height: 16
};
let book = {
  title: "Twilight",
  author: "Stephenie Meyer"
};

changeThings(level, rectangle, book);
console.log(level);
console.log(rectangle.width);
console.log(book.title);
```

Step 1 runs the following line of code:
Let level = 5;. Level is allocated memory and stores the value 5 at level's memory address.
Step 2 runs the following code:

let rectangle = {
  width: 3,
  height: 16
};
let book = {
  title: "Twilight",
  author: "Stephenie Meyer"
};.
Rectangle is allocated memory but stores the memory address that stores the values of width and height. The memory allocated to rectangle is shown pointing to the blocks of memory that are allocated to width and height. The same applies to the object book.
Step 3 runs the following code:
changeThings(level, rectangle, book.
The parameter y is shown pointing to the same block of memory rectangle points to and parameter z points to the same block of memory that book points to.
Step 4 runs the following code in changeThings: x = 33;.
Step 5 runs the following code in changeThings: y.width = 20;
Step 6 runs the following code in changeThings: z = { title: "The Hunger Games",
      author: "Suzanne Collins" };
In step, 7 the function changeThings returns and runs the following code:
console.log(level);
console.log(rectangle.width);
console.log(book.title);.

5, 20, and Twilight are displayed in the console.
After returning x, y, and z have their memory deallocated. Since the memory that stored the object z referred to no longer has a reference, the object gets deallocated memory.

Animation captions:

1. level is a number, which is a primitive type.
2. rectangle and book are objects. Each object refers to the object's location in memory.
3. The call to changeThings() assigns a copy of each argument to the x, y, and z parameters. y refers to the same object as rectangle, and z refers to the same object as book.
4. Assigning x a new number does not change level.
5. Assigning y.width a new number changes rectangle.width since both y and rectangle refer to the same object.
6. Assigning z a new object does not change book since z and book refer to different objects.
7. After returning from changeThings(), rectangle.width is the only value that has changed.

PARTICIPATION ACTIVITY 6.9.8: Passing objects to functions.

Refer to the code below.

```
function changeMovie(movie) {
   movie.title = "The Avengers";
   movie.released = 2012;
   movie = {
      title: "Avengers: Endgame",
      released: 2019 };
}

let avengersMovie = {
   title: "Avengers: Infinity War",
   released: 2018
};
```

1) What is output to the console?

```
changeMovie(avengersMovie);
console.log(avengersMovie.title);
```

- ○ Avengers: Infinity War
- ○ The Avengers
- ○ Avengers: Endgame

2) What is output to the console?

```
let myMovie = avengersMovie;
myMovie.title = "Avengers: Age of
Ultron";
console.log(avengersMovie.title);
```

- ○ Avengers: Infinity War
- ○ The Avengers
- ○ Avengers: Age of Ultron

CHALLENGE ACTIVITY 6.9.1: Objects.

550544.4142762.qx3zqy7

Exploring further:

- Working with objects (MDN)

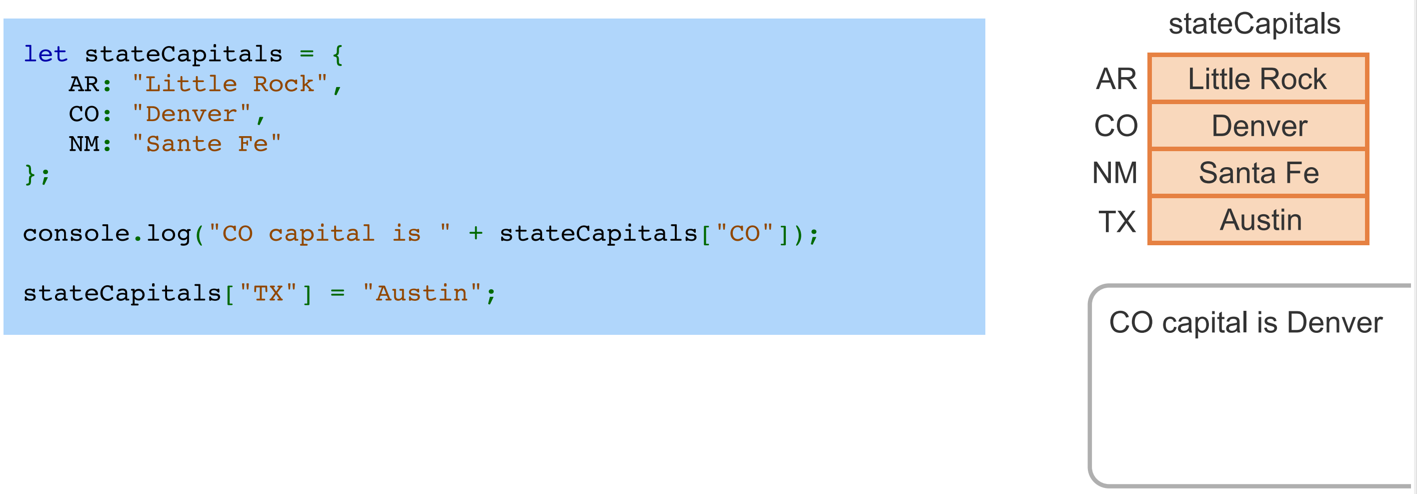
6.10 Maps

Objects as maps

A ***map*** or ***associative array*** is a data structure that maps keys to values. Each key/value pair in a map is called an ***element***.

JavaScript objects can be used as maps, in which the key is the object property and the value is the property's value. When an object is used as a map, individual elements are accessed by key using brackets. Ex: `myMap["key"]`.

PARTICIPATION ACTIVITY 6.10.1: State capitals in an object map.



```
let stateCapitals = {
   AR: "Little Rock",
   CO: "Denver",
   NM: "Sante Fe"
};

console.log("CO capital is " + stateCapitals["CO"]);

stateCapitals["TX"] = "Austin";
```

Animation content:

Static figure:

Begin code segment:
```
let stateCapitals = {
  AR: "Little Rock",
  CO: "Denver",
  NM: "Sante Fe"
};

console.log("CO capital is " + stateCapitals["CO"]);

stateCapitals["TX"] = "Austin";
```
End code segment.

Memory for stateCapitals shows four elements with two letter state codes as keys and capitals as values: AR/"Little Rock", CO/"Denver", NM/"Santa Fe", and TX/"Austin".

Console displays:
```
CO capital is Denver
```

Animation captions:

1. An object map called stateCapitals is initialized with three key/value pairs, creating three elements.
2. The map's value for key "CO" is "Denver".
3. The capital of Texas, with key "TX" and value "Austin", is added to the map.

PARTICIPATION ACTIVITY 6.10.2: Object maps.

Refer to the object map below.

```
let contacts = {
   Rosa: {
      phone: "303-555-4321",
      email: "rosa@gmail.com"
   },
   Dave: {
      phone: "501-533-9988",
      email: "dave@yahoo.com"
   },
   Li: {
      phone: "213-511-6758",
      email: "li@msn.com"
   }
};
```

1) What outputs Dave's email address?

```
console.log(______);
```

- ○ `["Dave"].email`
- ○ `contacts.email`
- ○ `contacts["Dave"].email`

2) What assigns a Twitter username to Rosa?

```
______ = "@rosaLuvsCats";
```

- ○ `contacts["Rosa"].twitter`
- ○ `contacts["rosa"].twitter`
- ○ `contacts["Rosa"].email`

3) What adds John to the `contacts` map?

```
______ = { phone: "111-2222",
email: "john@work.org" };
```

- ○ `contacts["John"].email`
- ○ `contacts`
- ○ `contacts["John"]`

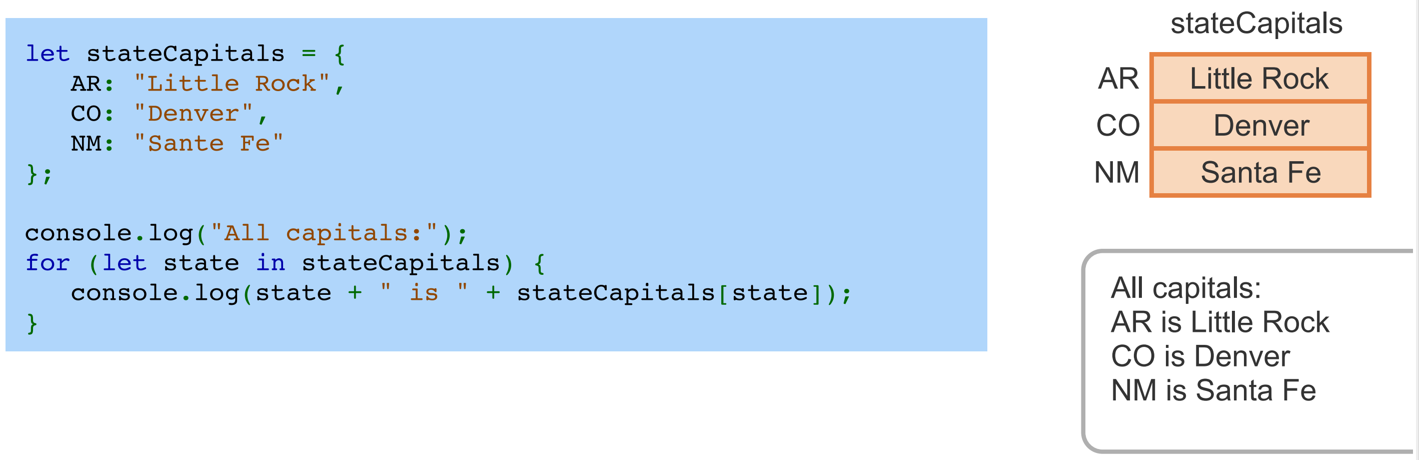
For-in loop

The ***for-in loop*** iterates over an object's properties in arbitrary order and is ideal for looping through an object map's elements. The for-in loop declares a variable on the left side of the `in` keyword and an object on the right. In each iteration, the variable is assigned with each of the object's properties.

Construct 6.10.1: for-in loop.

```
for (let variable in object)
{
   body
}
```

PARTICIPATION ACTIVITY 6.10.3: Looping through an object map.



```
let stateCapitals = {
   AR: "Little Rock",
   CO: "Denver",
   NM: "Sante Fe"
};

console.log("All capitals:");
for (let state in stateCapitals) {
   console.log(state + " is " + stateCapitals[state]);
}
```

```
All capitals:
AR is Little Rock
CO is Denver
NM is Santa Fe
```

Animation content:

Static figure:

Begin code snippet:
let stateCapitals = {
  AR: "Little Rock",
  CO: "Denver",

```
  NM: "Sante Fe"
};

console.log("All capitals:");
for (let state in stateCapitals) {
  console.log(state + " is " + stateCapitals[state]);
}
```

End code snippet.

Console displays:

```
All capitals:
AR is Little Rock
CO is Denver
NM is Sante Fe
```

Animation captions:

1. An object map called stateCapitals is initialized with three key/value pairs, creating three elements.
2. The for-in loop declares variable state inside the for-in statement.
3. The for-in loop assigns each key to the state variable, one at a time. The loop body outputs each element.

PARTICIPATION ACTIVITY 6.10.4: For-in loop.

Refer to the object map below.

```
let contacts = {
   Rosa: {
      phone: "303-555-4321",
      email: "rosa@gmail.com"
   },
   Dave: {
      phone: "501-533-9988",
      email: "dave@yahoo.com"
   },
   Li: {
      phone: "213-511-6758",
      email: "li@msn.com"
   }
};
```

1) Which expression loops through the `contacts` map to output all names and phone numbers?

```
for (______) {
   console.log(name + ": " +
contacts[name].phone);
}
```

- ○ `let contacts`
- ○ `let name in contacts`
- ○ `let contacts in name`

2) What is missing to assign `email` with each contact's email address?

```
for (let name in contacts) {
   let email = _____;
   if (email.includes("msn"))
{
      console.log(name);
   }
}
```

- ○ `contacts.email`
- ○ `name.email`
- ○ `contacts[name].email`

PARTICIPATION ACTIVITY 6.10.5: Practice with object maps.

Create an object map called `courses` that stores a university department's course number as the key and an object as the value. The object has three properties: `title`, `description`, `creditHours`. Example courses:

- 170 - Introduction to Programming, Develop algorithms for computers, 5.
- 250 - Web Development, Build web applications, 3.
- 310 - Operating Systems, Process management and memory management, 3.
- 430 - Artificial Intelligence, Simulate human thinking, 2.

Then, write a for-in loop that displays the course number and title for only those courses that are 3 credit hours.

Other object map operations

Other common operations performed on object maps include:

- **Get number of elements.** The ***Object.keys()*** method returns an array of an object's property names. The array's `length` property returns the number of elements in the object map.
- **Check for key.** The ***in operator*** returns `true` if an object contains the given property and returns `false` otherwise.
- **Remove element.** The ***delete operator*** removes a key/property from a map or object.

Figure 6.10.1: Object map operations example.

```
let stateCapitals = {
   AR: "Little Rock",
   CO: "Denver",
   NM: "Sante Fe"
};

let states =
Object.keys(stateCapitals);
console.log(states);          //
AR,CO,NM
console.log(states.length);   // 3

// Evaluates true
if ("NM" in stateCapitals) {
   console.log("NM exists");
}

// Remove the NM/Santa Fe pair
delete stateCapitals["NM"];

// Evaluates false
if ("NM" in stateCapitals) {
   console.log("NM exists");
}

// Outputs undefined
console.log(stateCapitals["NM"]);
```

PARTICIPATION ACTIVITY 6.10.6: in and delete operators.

Refer to the object map below.

```
let students = {
   123: { name: "Tiara",  gpa: 3.3 },
   444: { name: "Lee",    gpa: 2.0 },
   987: { name: "Braden", gpa: 3.1 }
};
```

1) Remove Lee from `students`.

```
delete ________;
```

Check Show answer

2) Assuming `students` has three elements before the code executes, what number is output to the console?

```
delete students["Braden"];
console.log(Object.keys(students).length);
```

Check Show answer

3) What is missing to check if student ID 888 is in `students`?

```
if (________) {
   console.log("Hello, "
+ students[888].name);
}
```

Check Show answer

4) What is output to the console?

```
for (let id in students)
{
   delete students[id];
}
if (123 in students) {
   console.log("yes");
}
else {
   console.log("no");
}
```

Check Show answer

Map object

The ***Map object*** is a newer alternative to using objects for storing key/value pairs. Common methods and properties of the `Map` object include:

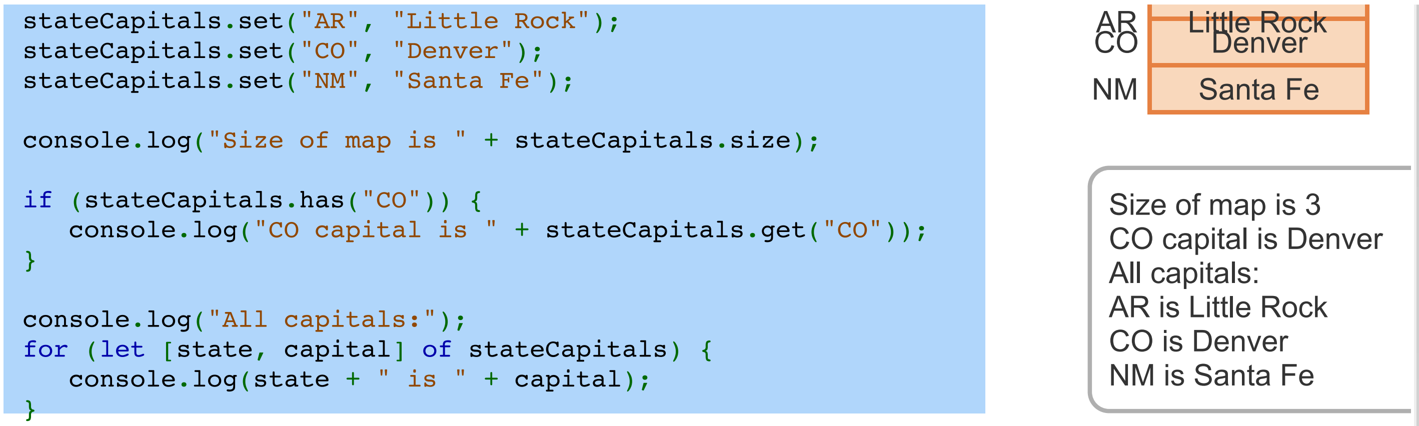
- The ***set(key, value)*** method sets a key/value pair. If the key is new, a new element is added to the map. If the key already exists, the new value replaces the existing value.
- The ***get(key)*** method gets a key's associated value.
- The ***has(key)*** method returns true if a map contains a key, false otherwise.
- The ***delete(key)*** method removes a map element.
- The ***size*** property is the number of elements in the map.

The for-of loop, which is often used to loop through an array, is ideal for looping through a `Map`. Each of the map's key/value pairs are assigned to the `[key, value]` variables declared in the for-of loop, as illustrated in the animation below.

PARTICIPATION ACTIVITY | 6.10.7: State capitals in a Map.

```
let stateCapitals = new Map();
```

stateCapitals



```
stateCapitals.set("AR", "Little Rock");
stateCapitals.set("CO", "Denver");
stateCapitals.set("NM", "Santa Fe");

console.log("Size of map is " + stateCapitals.size);

if (stateCapitals.has("CO")) {
   console.log("CO capital is " + stateCapitals.get("CO"));
}

console.log("All capitals:");
for (let [state, capital] of stateCapitals) {
   console.log(state + " is " + capital);
}
```

```
Size of map is 3
CO capital is Denver
All capitals:
AR is Little Rock
CO is Denver
NM is Santa Fe
```

Animation content:

The following code is displayed:
let stateCapitals = new Map();
stateCapitals.set("AR", "Little Rock");
stateCapitals.set("CO", "Denver");
stateCapitals.set("NM", "Santa Fe");

console.log("Size of map is " + stateCapitals.size);

if (stateCapitals.has("CO")) {
  console.log("CO capital is " + stateCapitals.get("CO"));
}

console.log("All capitals:");
for (let [state, capital] of stateCapitals) {
  console.log(state + " is " + capital);
}
Step 1 runs the first line and stateCapitals is initialized to a map object with nothing stored.
Step 2 runs the following code:
stateCapitals.set("AR", "Little Rock");
stateCapitals.set("CO", "Denver");
stateCapitals.set("NM", "Santa Fe");.
The map stateCapitals is populated with keys: AR, CO, and NM. Each key is associated with a memory location storing the strings: Little Rock, Denver, and Sante Fe.
Step 3 runs the following line of code:
console.log("Size of map is " + stateCapitals.size);.
Size of map is 3 is displayed in the console.

Step 4 runs the following if block:
if (stateCapitals.has("CO")) {
  console.log("CO capital is " + stateCapitals.get("CO"));
}.
CO capital is Denver is displayed in the console.
Step 5 runs the following code:
console.log("All capitals:");
for (let [state, capital] of stateCapitals) {
  console.log(state + " is " + capital);
}.
The following lines are displayed in the console:
All capitals:
AR is Little Rock
CO is Denver
NM is Sante Fe

Animation captions:

1. A new Map object is created with the Map() constructor.
2. The set() method adds three key/value pairs to the stateCapitals Map.
3. The size property returns 3 because stateCapitals has three key/value pairs.
4. The has() method returns true because "CO" is one of the keys in stateCapitals. The get() method returns the value associated with "CO".
5. The for-of loop assigns each key/value pair to variables state and capital.

PARTICIPATION ACTIVITY 6.10.8: Map object.

Refer to the map below.

```
let contacts = new Map([
   ["Rosa", { phone: "303-555-4321", email: "rosa@gmail.com" }],
   ["Dave", { phone: "501-533-9988", email: "dave@yahoo.com" }],
   ["Li",   { phone: "213-511-6758", email: "li@msn.com" }]
]);
```

1) What is output to the console?

```
console.log(contacts.size);
```

- ○ 0
- ○ 2
- ○ 3

2) What is output to the console?

```
contacts.set("Li", {
   phone: "213-444-6758",
   email: "li@email.com" });
console.log(contacts.size);
```

- ○ 0
- ○ 3
- ○ 4

3) What is output to the console?

```
contacts.delete("Li");
console.log(contacts.size);
```

- ○ 0
- ○ 2
- ○ 3

4) What outputs Dave's email address?

```
console.log(______);
```

- ○ `contacts.get("Dave")`
- ○ `contacts.get("Dave").email`
- ○ `contacts["Dave"].email`

5) What is output to the console?

```
if (contacts.has("John")) {

console.log(contacts.get("John").phone);
}
else {
   console.log("Missing");
}
```

- John's phone number
- The string "Missing"
- An empty string

6) The code below assigns a Twitter username to Rosa while leaving Rosa's `email` and `phone` properties unaltered. How can the code be simplified?

```
let rosa = contacts.get("Rosa");
rosa.twitter = "@rosaLuvsCats";
```

-
  ```
  contacts.get("Rosa").twitter
  = "@rosaLuvsCats";
  ```

-
  ```
  contacts.set("Rosa").twitter
  = "@rosaLuvsCats";
  ```

-
  ```
  contacts.set("Rosa", {
  twitter = "@rosaLuvsCats"
  });
  ```

7) Which expression loops through the `contacts` map to output all names and phone numbers?

```
for (______) {
   console.log(name + ": " +
contact.phone);
}
```

- ○ `let contact of contacts`
- ○ `let [name, contact] of contacts`
- ○ `let name of contacts`

CHALLENGE ACTIVITY 6.10.1: Maps.

550544.4142762.qx3zqy7

Exploring further:

- Map object (MDN)

6.11 String object

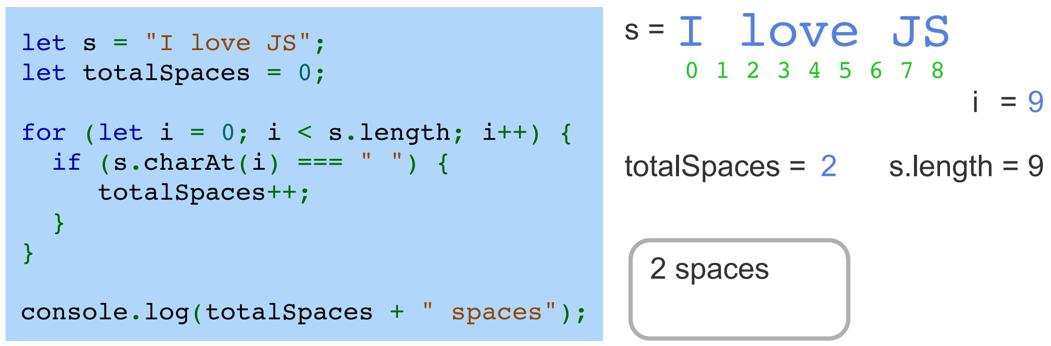
Introduction to the String object

The ***String*** object defines methods to manipulate strings, extract substrings, test for string inclusion, etc. A string literal (a string in "quotes") is automatically converted into a `String` object when a `String` method is invoked.

The `String` method ***charAt()*** returns the character at the specified index as a string. Ex: `"test".charAt(1)` returns the character "e" at index 1. The `String` property ***length*** returns the number of characters in a string. Ex: `"test".length` returns 4. Calling `charAt()` with an index ≥

the string's `length` returns an empty string.

PARTICIPATION ACTIVITY 6.11.1: Counting spaces in a string.



```
let s = "I love JS";
let totalSpaces = 0;

for (let i = 0; i < s.length; i++) {
  if (s.charAt(i) === " ") {
     totalSpaces++;
  }
}

console.log(totalSpaces + " spaces");
```

Animation content:

The following code is displayed:
let s = "I love JS";
let totalSpaces = 0;

for (let i = 0; i < s.length; i++) {
 if (s.charAt(i) === " ") {
  totalSpaces++;
 }
}

console.log(totalSpaces + " spaces");
In step 1, the string s is initialized and indices 0-8 are shown under each character in the string "I love JS"
Step 3 and 4 runs the following line:
for (let i = 0; i < s.length; i++) {.
i is initialized to 0, and s.length is 9. i is shown as a pointer under the index 0 in the string s.
In step 5, the if statement is checked when i = 0.
In step 6, the loop iterates with i now equal to 1, and totalSpaces becomes 1.
In step 7, the loop runs until i is incremented to 9. The string "2 spaces" is displayed in the console.

Animation captions:

1. The s variable is initialized to a string literal.
2. The totalSpaces variable is used to count how many spaces are in the string s.
3. Use variable i to iterate through the string s.
4. The loop continues until i is s.length, the number of characters in the string s.
5. s.charAt(0) is "I", not a space, so totalSpaces is not affected.
6. s.charAt(1) is a space, so totalSpaces is incremented to 1.
7. The for loop continues to check each character in the string. totalSpaces is 2 when the loop terminates.

PARTICIPATION ACTIVITY 6.11.2: String object.

1) What is the value of `s.length`?

```
let s = "";
```

- ○ 0
- ○ NaN
- ○ 1

2) What is the value of `s.charAt(1)`?

```
let s = "To be, or not to
be.";
```

- ○ space
- ○ T
- ○ o

3) What is the value of `s[1]`?

```
let s = "To be, or not to
be.";
```

- space
- T
- o

Searching and replacing

The `String` object provides methods to search and replace strings:

- The ***indexOf()*** method returns the index of the search string's first occurrence inside the `String` object or -1 if the search string is not found.
- The ***lastIndexOf()*** method returns the index of the search string's last occurrence inside the `String` object or -1 if the search string is not found.
- The ***replace()*** method replaces one string with another and returns the string with the replacement string inside.

Figure 6.11.1: Searching for a string with indexOf() and lastIndexOf().

```
let s = "Seek and you will find.";
s.indexOf("and");     // 5
s.indexOf("e");       // 1 (first occurrence)
s.lastIndexOf("e");   // 2 (last occurrence)
s.indexOf("SEEK");    // -1 (case-sensitive
search)
```

Figure 6.11.2: Replacing a string with replace().

```
let s = "Seek and you will find.";
s = s.replace("find", "discover");   // "Seek and you will discover"
s = s.replace("Seek", "Search");     // "Search and you will
discover"
s = s.replace("SEARCH", "search");   // No change (case-sensitive
search)
```

PARTICIPATION ACTIVITY | 6.11.3: Search and replace.

Enter the value assigned to **x** in each code segment.

```
let quote = "I am Spartacus!";
```

1)
```
x =
quote.indexOf("part");
```

Check Show answer

2)
```
x = quote.indexOf("I
am!");
```

Check Show answer

3)
```
x =
quote.lastIndexOf("a");
```

Check Show answer

4)
```
x = quote.replace("am",
"was");
```

Check Show answer

5)
```
x =
quote.replace("spart",
"mart");
```

Check Show answer

Other String methods

A variety of other `String` methods exist. Some of the common methods are summarized in the table below.

Table 6.11.1: Common String methods.

Method	Description	Example
substr()	Returns the substring that begins at a given index and has an optional given length.	`s = "As you wish.";` `s.substr(3, 3);   // "you"` `s.substr(3);    // "you wish." (length optional)`
substring()	Returns the substring between two indices, not including the second index.	`s = "As you wish.";` `s.substring(3, 6);   // "you"` `s.substring(3);    // "you wish." (2nd index optional)`
split()	Returns an array of strings formed by splitting the string into substrings. The given delimiter	`s = "As you wish.";` `s.split(" ");   // ["As", "you", "wish."]`

	separates substrings.	

toLowerCase()	Returns the string converted to lowercase characters.	`s = "What?";`
		`s.toLowerCase();   // "what?"`

toUpperCase()	Returns the string converted to uppercase characters.	`s = "What?";`
		`s.toUpperCase();   // "WHAT?"`

trim()	Returns the string with leading and trailing whitespace removed.	`s = "   test   ";`
		`s.trim();   // "test"`

PARTICIPATION ACTIVITY 6.11.4: String methods.

Match the return value to the `String` method.

```
let greeting = " Welcome Home! ";
```

If unable to drag and drop, refresh the page.

[" Welc", "me H", "me! "]   "Home"   "Home! "   "Welcome Home!"   " welcome home! "

	`greeting.trim()`

	`greeting.substr(9, 4)`
	`greeting.substr(9)`
	`greeting.toLowerCase()`
	`greeting.split("o")`

Reset

PARTICIPATION ACTIVITY | 6.11.5: Practice with String methods.

When creating an account online, most websites require the user to enter a "strong" password. A strong password is a password that someone is unlikely to guess. Strong passwords must be sufficiently long, contain upper and lowercase letters, contain punctuation, etc.

Complete the `testPassword()` function, which tests the strength of the given password. `testPassword()` should verify the password meets the criteria below in the order specified. If the criteria is not met, `testPassword()` should return an appropriate message indicating what is wrong with the password. If all the criteria are met, `testPassword()` should return an empty string.

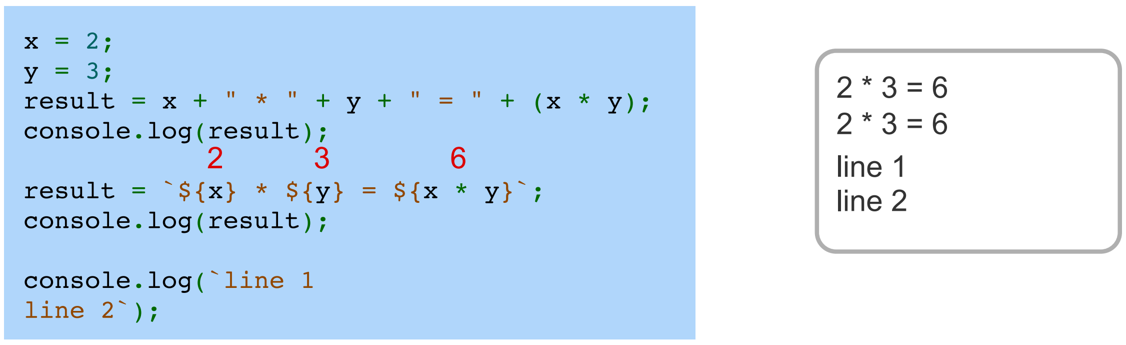
1. Minimum length of 6 characters - Use the `length` property to ensure the password is long enough.
2. No spaces - Use `indexOf()` to ensure the password does not contain any spaces.
3. Use at least one digit - Create a loop to examine each character of the password, and count how many times a digit character appears. JavaScript does not have a function to verify if a character is a digit, so use the `isSingleDigit()` function provided. The password should have at least one digit.
4. First 3 characters must not be repeated at the end - Use `substr()` to extract the string at the front and end of the password. Then, compare the substrings with ===. Ex: Password "abc123abc" is not acceptable because "abc" at the front of the password is the same as "abc" at the end of the password.

The code is currently testing the password "pass" which should fail because the password is only 4 characters long. Verify that `testPassword()` works by trying passwords that fail each of the four criteria.

Template literals

A ***template literal*** is a string literal enclosed by the back-tick (`` ` ``) that allows embedding expressions with a dollar sign and braces (`${expression}`). Ex: `` `test ${1 + 2}` `` evaluates to `"test 3"`. Template literals replace the need to produce a string with string concatenation.

PARTICIPATION ACTIVITY 6.11.6: Template literal simplifies syntax.



```
x = 2;
y = 3;
result = x + " * " + y + " = " + (x * y);
console.log(result);
              2      3        6
result = `${x} * ${y} = ${x * y}`;
console.log(result);

console.log(`line 1
line 2`);
```

```
2 * 3 = 6
2 * 3 = 6
line 1
line 2
```

Animation content:

The following code is displayed:
x = 2;
y = 3;
result = x + " * " + y + " = " + (x * y);
console.log(result);

result = `${x} * ${y} = ${x * y}`;
console.log(result);

console.log(`line 1
line 2`);

Step 1 runs the following code:
x = 2;
y = 3;
result = x + " * " + y + " = " + (x * y);
console.log(result);.
The string 2 * 3 = 6 is displayed in the console.
Step 2 runs the following code:
result = `${x} * ${y} = ${x * y}`;
console.log(result);
Step 3 runs the following code:
console.log(`line 1
line 2`);.
In the console, line 1 and line 2 are displayed in different lines despite only having 1 console.log line run.

Animation captions:

1. String concatenation is required to build a string showing the math equation.
2. A template literal simplifies the syntax to build the same string.
3. Newline characters inserted in a template literal create multi-line strings.

PARTICIPATION ACTIVITY 6.11.7: Template literal.

Rewrite each string assignment using a template literal.

1)
```
greeting = "Welcome, " +
name + "!";
```

```
greeting =
                    ;
```

Check Show answer

2)
```
answer = "sum is " + (x +
y);
```

answer =

;

Check Show answer

3)
```
yell = verb.toUpperCase()
+ "!";
```

yell =

;

Check Show answer

4)
```
letters = "a\nb\nc";
```

letters =

;

Check Show answer

CHALLENGE ACTIVITY 6.11.1: Strings.

550544.4142762.qx3zqy7

Exploring further:

- String object (MDN)
- Template literals (Template strings) (MDN)

6.12 Date object

Date object

A ***Date*** object represents a single moment in time, based on the number of milliseconds since the Unix Epoch (January 1, 1970 UTC). UTC (Coordinated Universal Time), also known as GMT (Greenwich Mean Time), is a 24-hour time standard. The `Date` object is created with the `new` operator and a constructor. A ***constructor*** is a function that creates an instance of an object.

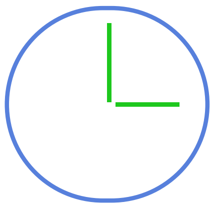
PARTICIPATION ACTIVITY 6.12.1: Date object constructor.

```
let currDateTime = new Date();
console.log(currDateTime);

let oneSecPastEpoch = new Date(1000);
console.log(oneSecPastEpoch);

// Feb 22, 1732
let georgeBirthday = new Date(1732, 1, 22);
console.log(georgeBirthday);

// Oct 21, 2035 at 7:28:00
let theFuture = new Date(2035, 9, 21, 7, 28, 0);
console.log(theFuture);
```



```
Thu Apr 18 2019 15:26:13 GMT-0500 (Central Daylight Time)
Wed Dec 31 1969 18:00:01 GMT-0600 (Central Standard Time)
Fri Feb 22 1732 00:00:00 GMT-0600 (Central Standard Time)
Sun Oct 21 2035 07:28:00 GMT-0500 (Central Daylight Time)
```

Animation content:

The following code is displayed:
let currDateTime = new Date();

```
console.log(currDateTime);

let oneSecPastEpoch = new Date(1000);
console.log(oneSecPastEpoch);

// Feb 22, 1732
let georgeBirthday = new Date(1732, 1, 22);
console.log(georgeBirthday);

// Oct 21, 2035 at 7:28:00
let theFuture = new Date(2035, 9, 21, 7, 28, 0);
console.log(theFuture);
```

Step 1 runs the following code:
let currDateTime = new Date();

Step 2 runs the following code:
console.log(currDateTime);.
The following line is displayed in the console:
Thu Apr 18 2019 15:26:13 GMT-0500 (Central Daylight Time).

Step 3 runs the following code:
let oneSecPastEpoch = new Date(1000);

Step 4 runs the following code:
console.log(georgeBirthday);.
The following line is displayed in the console:
Wed Dec 31 1969 18:00:01 GMT-0600 (Central Standard Time).

Step 5 runs the following code:
let georgeBirthday = new Date(1732, 1, 22);

Step 6 runs the following code:
console.log(georgeBirthday);
The following line is displayed in the console:
Fri Feb 22 1732 00:00:00 GMT-0600 (Central Standard Time).

Step 7 runs the following code:

let theFuture = new Date(2035, 9, 21, 7, 28, 0);
console.log(theFuture);
The following line is displayed in the console:
Sun Oct 21 2035 07:28:00 GMT-0500 (Central Daylight Time).

Animation captions:

1. Initialize the variable currDateTime to the current date and time using the Date constructor.
2. Display the currDateTime variable, which is in the local time zone. Central Daylight Time is 5 hours before Greenwich Mean Time (GMT).
3. Initialize the variable oneSecPastEpoch to 1000 milliseconds past Jan 1, 1970 using the Date constructor.
4. Central Standard Time is 6 hours before GMT. Daylight time (called Daylight Saving Time) is one hour different than standard time because clocks are turned forward one hour.
5. Initialize the variable georgeBirthday to Feb 22, 1732. The month parameter ranges from 0-11, so 1 = Feb.
6. georgeBirthday falls on a Friday and is 6 hours before GMT.
7. Initialize the variable theFuture to Oct 21, 2035 at 7:28:00. theFuture date falls on a Sunday.

PARTICIPATION ACTIVITY 6.12.2: Date object constructor.

1) The Date constructor must be passed at least one argument.

○ True

○ False

2) The following code initializes **x** to December 25, 2017.

```
let x = new Date(2017, 12,
25);
```

○ True

○ False

3) The code below displays the same string, regardless of the local time zone.

```
let x = new Date(2016, 5, 1,
15, 30, 45);
console.log(x);
```

- True
- False

Date methods

The `Date` object provides a number of methods to get and set `Date` properties.

Table 6.12.1: Date object getter and setter methods.

Method	Description	Example
getDate() ***setDate()***	Gets or sets the day relative to the current set month	`let day = new Date(2016, 0, 30);` `day.getDate(); // 30` `day.setDate(21); // 30 -> 21`
getDay()	Returns the day of the week (0-6)	`let day = new Date(2016, 0, 30);` `day.getDay(); // 6 = Saturday`
getFullYear() ***setFullYear()***	Gets or sets the 4 digit year	`let day = new Date(2016, 0, 30);` `day.getFullYear(); // 2016` `day.setFullYear(2017); // 2016 -> 2017`

getHours() ***setHours()***	Gets or sets the hour (0-23)	`let day = new Date(2016, 0, 30, 5, 0);` `day.getHours();     // 5` `day.setHours(2);    // 5 -> 2`
getMilliseconds() ***setMilliseconds()***	Gets or sets the milliseconds (0-999)	`let day = new Date(2016, 0, 1, 5, 20, 10, 250);` `day.getMilliseconds(); // 250` `day.setMilliseconds(500); // 250 -> 500`
getMinutes() ***setMinutes()***	Gets or sets the minutes (0-59)	`let day = new Date(2016, 0, 30, 5, 20);` `day.getMinutes();    // 20` `day.setMinutes(35);  // 20 -> 35`
getMonth() ***setMonth()***	Gets or sets the month (0-11)	`let day = new Date(2016, 0, 30, 5, 20);` `day.getMonth();   // 0` `day.setMonth(3);  // 0 (Jan) -> 3 (Apr)`
getSeconds() ***setSeconds()***	Gets or sets the seconds (0-59)	`let day = new Date(2016, 0, 1, 5, 20, 10, 250);` `day.getSeconds();    // 10` `day.setSeconds(45);  // 10 -> 45`
getTime() ***setTime()***	Gets or sets the number of milliseconds since Jan 1, 1970, 00:00:00 UTC	`let day = new Date(2016, 0, 30, 5, 20);` `day.getTime(); // 1454152800000` `day.setTime(1454153700000); // Sat Jan 30 2016 05:35:00 GMT-0600`

PARTICIPATION ACTIVITY 6.12.3: Practice with the Date object.

The `notablePeople` map contains a list of some notable individuals and birthdays. Ex: `notablePeople["Elvis Presley"]` contains Elvis' birthday, which is Jan 8, 1935.

1. The first for-in loop displays each person's name and birthday, but the format of the birthday is too wordy. Change the output format to MM/DD/YYYY. Ex: Elvis Presley: 1/8/1935.
2. Add a for-in loop to display each person born before Sonia Sotomayor and the rounded number of days difference in birth day. Ex:
   ```
   Elvis Presley was born 7108 days before Sonia Sotomayor
   Franklin D. Roosevelt was born 26443 days before Sonia Sotomayor
   ...
   ```
   - The getDifferenceInDays() utility function is provided in the code below. When passed a time difference in milliseconds, the function returns the rounded number of days difference. Ex: `getDifferenceInDays(person1Date.getTime() - person2Date.getTime(`
3. Add a for-in loop to display each person born after Sonia Sotomayor and the rounded number of days difference in birth day. Ex:
   ```
   Elon Musk was born 6212 days after Sonia Sotomayor
   Steve Jobs was born 244 days after Sonia Sotomayor
   ...
   ```

CHALLENGE ACTIVITY 6.12.1: Using Date methods.

550544.4142762.qx3zqy7

Exploring further:

- Date object (MDN)

6.13 Math object

Introduction to the Math object

The ***Math*** object provides properties for mathematical constants and methods to perform mathematical functions.

PARTICIPATION ACTIVITY 6.13.1: Math properties.

Match the Math property to the property's description.

If unable to drag and drop, refresh the page.

`Math.E` `Math.SQRT2` `Math.LOG10E` `Math.LN2` `Math.PI`

	Value of π, approximately 3.142
	Euler's number, approximately 2.718
	Natural logarithm of 2, approximately 0.693
	Base 10 logarithm of E, approximately 0.434
	Square root of 2, approximately 1.414

Reset

Math methods

The `Math` object has a range of trigonometric methods, including `sin()`, `cos()`, and `tan()`, and general calculation methods, including `log()` and `pow()`. Some commonly used `Math` methods are summarized in the table below.

Table 6.13.1: Common Math object methods.

Method	Description	Example
abs(x)	Returns the absolute value of x	`Math.abs(-5);  // 5`
ceil(x)	Returns x rounded up to the nearest integer	`Math.ceil(2.1);  // 3`
cos(x)	Returns the cosine of the radians x	`Math.cos(Math.PI)  // -1`
floor(x)	Returns x rounded down to the nearest integer	`Math.floor(2.9)  // 2`
log(x)	Returns the natural logarithm of x	`Math.log(Math.E)  // 1`
max(n1, n2, n3, ...)	Returns the largest number	`Math.max(5, 2, 8, 1) // 8`
min(n1, n2, n3, ...)	Returns the smallest number	`Math.min(5, 2, 8, 1) // 1`
pow(x, y)	Returns x to the power of y	`Math.pow(2, 3)  // 8`
round(x)	Returns x rounded to the nearest integer	`Math.round(3.5)  // 4`
sin(x)	Returns the sine of radians x	`Math.sin(Math.PI)  // 0`
sqrt(x)	Returns the square root of x	`Math.sqrt(25)  // 5`
tan(x)	Returns the tangent of radians x	`Math.tan(Math.PI / 4) // 1`

PARTICIPATION ACTIVITY

6.13.2: Math methods.

Enter the value assigned to `x` in each code segment.

1)
```
x = Math.sin(Math.PI /
2);
```

Check Show answer

2)
```
x = Math.pow(2, 3);
```

Check Show answer

3)
```
x = Math.sqrt(9);
```

Check Show answer

4)
```
x = Math.round(12.6) +
Math.floor(3.7);
```

Check Show answer

5)
```
x = Math.max(9, 14, -26);
```

Check Show answer

Producing random numbers

Many applications, especially games and simulations, need random numbers to simulate random processes. The ***Math.random()*** method returns a pseudo-random number ≥ 0 and < 1. A ***pseudo-random number*** is a number generated by an algorithm that approximates randomness, but is not truly random.

Figure 6.13.1: Display 5 random numbers with Math.random().

```
for (let i = 0; i < 5; i++) {
   console.log(Math.random());
}
```

```
0.5216294566239728
0.5399290004983317
0.05689844662407162
0.8711941395310085
0.7131957592778093
```

The figure below shows a `getRandomNumber()` function that performs the necessary calculations to generate a random integer between two integers.

Figure 6.13.2: Display five random numbers between 1 and 10.

```
// Return a random integer between min and max (inclusive).
function getRandomNumber(min, max) {
   return Math.floor(Math.random() * (max - min + 1)) +
min;
}

for (let i = 0; i < 5; i++) {
   console.log(getRandomNumber(1, 10));
}
```

```
7
3
1
8
4
```

PARTICIPATION ACTIVITY 6.13.3: Random numbers.

1) Numbers produced by `Math.random()` appear to be random.

   ○ True

   ○ False

2) `Math.floor(Math.random() * 10)` produces a random number between 0 and 10, inclusive.

   ○ True

   ○ False

3) `Math.floor(Math.random() * 10) + 2` produces a random number between 2 and 11, inclusive.

   ○ True

   ○ False

PARTICIPATION ACTIVITY 6.13.4: Practice with random numbers.

The `displayCard(rank, suit)` function displays a playing card to the console, given the rank (1-13) and suit (0-3).

Write a for loop that calls `displayCard()` 10 times, each time with a random rank and suit.

CHALLENGE ACTIVITY 6.13.1: Math object.

550544.4142762.qx3zqy7

Exploring further:

- Math object (MDN)

- Introduction to Randomness and Random Numbers

6.14 Exception handling

Exception and try-catch

An ***exception*** is an error that disrupts the normal flow of program execution. When an exception occurs, a program may need to execute code to handle the error. Ex: Display an error message, call a function, or shutdown. ***Exception handling*** is the process of catching and responding to an exception.

Figure 6.14.1: An exception is generated when calling a non-existing method.

```
// Oops!  Should be console.log()
console.Log("Will this work?");
```

```
Uncaught TypeError: console.Log is not a function (line
2)
```

The ***throw*** statement throws a user-defined exception. Syntax: `throw expression`. Ex: `throw "number is negative"` throws an exception with a string value.

A program halts when an exception is thrown unless a `try-catch` statement is used to catch/handle the exception. A ***try-catch*** statement has a `try` block to execute code that may throw an exception and a `catch` block that executes when an exception is thrown.

Construct 6.14.1: try-catch.

```
try {
   // Statements that might throw an
exception
}
catch (exception) {
   // Handle the exception
}
```

PARTICIPATION ACTIVITY 6.14.1: Throwing and catching an exception.

```
function findSum(numbers, startIndex, endIndex) {
   if (startIndex < 0) {
      throw "startIndex is less than 0.";
   }
   else if (endIndex >= numbers.length) {
      throw "endIndex is too large.";
   }

   let sum = 0;
   for (let i = startIndex; i <= endIndex; i++) {
      sum += numbers[i];
   }
   return sum;
}
```

```
let nums = [1, 2, 3, 4];

console.log(findSum(nums, 0, 2));

console.log(findSum(nums, 3, 4));

console.log("Done!");
```

```
6
Uncaught endIndex is too
large.
```

```
let nums = [1, 2, 3, 4];
try {
   console.log(findSum(nums, 3, 4));
}
catch (exception) {
   console.log(exception);
}

console.log("Done!");
```

```
endIndex is too large.
Done!
```

Animation content:

The following code is displayed:

```
function findSum(numbers, startIndex, endIndex) {
   if (startIndex < 0) {
      throw "startIndex is less than 0.";
   }
   else if (endIndex >= numbers.length) {
      throw "endIndex is too large.";
   }

   let sum = 0;
   for (let i = startIndex; i <= endIndex; i++) {
      sum += numbers[i];
   }
   return sum;
}



let nums = [1, 2, 3, 4];
console.log(findSum(nums, 0, 2));
console.log(findSum(nums, 3, 4));

console.log("Done!");
```

Step 1 runs the following lines:

```
let nums = [1, 2, 3, 4];
console.log(findSum(nums, 0, 2));.
```

6 is displayed in the console.

Step 2 runs the following line:

```
console.log(findSum(nums, 3, 4));
```

Step 3 runs lines of code in the function, findSum, called in step 2. The following lines were the lines run in step 3:

```
if (startIndex < 0) {
      throw "startIndex is less than 0.";
}
```

```
else if (endIndex >= numbers.length) {
    throw "endIndex is too large.";
}
```

Step 3 throws the exception inside the else if statement and exits the function. The exception thrown was not handled so Uncaught endIndex is too large. is displayed in the console.

Step 4 and 5 runs new code:

```
let nums = [1, 2, 3, 4];
try {
  console.log(findSum(nums, 3, 4));
}
catch (exception) {
  console.log(exception);
}

console.log("Done!");
```

The function findSum() is called like and the endIndex exception is thrown. The catch block handles the exception and endIndex is too large. is displayed in the console. The line of code after the catch block runs displays Done! in the console.

Animation captions:

1. The findSum() function adds the numbers in the nums array from index 0 to index 2.
2. findSum() is called with endIndex of 4. Since endIndex is equal to numbers.length, the throw statement throws an exception, indicating a problem with the endIndex value.
3. The code does not handle the exception, so the exception message is displayed in the console, and the program halts prematurely.
4. Using a try-catch statement, the catch block handles the exception and outputs the exception message.
5. After the catch block executes, the program execution continues.

PARTICIPATION ACTIVITY 6.14.2: Exception handling.

Refer to the animation above.

1) What happens when `findSum()` throws an exception, and `findSum()` is not inside a `try-catch` statement?

- ○ The uncaught exception is output to the console, and the program continues to run.
- ○ The uncaught exception is output to the console, and the program terminates.
- ○ The exception is ignored, and the program continues to run.

2) Complete the code to throw an exception when `endIndex` is smaller than `startIndex`?

```
if (endIndex < startIndex) {
   _________ "endIndex is
smaller than startIndex.";
}
```

- ○ `try`
- ○ `catch`
- ○ `throw`

3) Will the `catch` block execute if `findSum()` does not throw an exception?

- ○ Yes
- ○ No

4) Suppose the call to `findSum()` below throws an exception. What is output to the console?

```
try {
   console.log(findSum(nums,
-1, 2));
   console.log("got here");
}
catch (exception) {
   console.log("error");
}
console.log("done");
```

- ○
```
error
done
```
- ○
```
got here
error
done
```
- ○
```
error
```

PARTICIPATION ACTIVITY 6.14.3: Practice throwing and handling an exception.

Professor X has written a JavaScript function `findAverage()` to help his students compute homework score averages. The `findAverage()` function returns the average of an array of homework scores. Professor X's students complain that sometimes `findAverage()` returns `NaN` or unexpected answers. Professor X suspects the problems are due to students passing `findAverage()` an empty scores array or an array with improperly formatted scores.

Modify `findAverage()` to throw exceptions for the following reasons:

1. No scores are in the `scores` array.
2. A negative score was found in the `scores` array.
3. A non-integer was found in the `scores` array.

Wrap the existing function calls to `findAverage()` in a `try-catch` statement, and output any thrown exceptions to the console. The program should continue to try the next call to `findAverage()` regardless of any exceptions thrown.

To determine if a non-integer exists in the `scores` array, use ***Number.isInteger(n)***, which returns true if `n` is an integer, and false otherwise.

Finally block

A ***finally*** block may follow a `try` or `catch` block. The `finally` block executes regardless of whether an exception was thrown or not.

Developers use the `finally` block for any operations that must be executed, whether or not an exception was thrown. Ex: Releasing resources, closing files, and rolling back failed database transactions.

Construct 6.14.2: try-catch-finally.

```
try {
   // Statements to try
}
catch (exception) {
   // Optionally handle exceptions
}
finally {
   // Code that executes no matter
what
}
```

PARTICIPATION ACTIVITY 6.14.4: try-catch-finally execution.

```
function test() {
   try {
      console.log("try");
      throw "crash!!!";

      // Skips because exception is thrown
      console.log("after throw");
   }
   catch (exception) {
      console.log("catch");
   }
   finally {
      console.log("finally");
   }
```

```
try
catch
finally
after
done
```

```
   console.log("after");
}

test();
console.log("done");
```

Animation content:

The following code is displayed:

```
function test() {
  try {
    console.log("try");
    throw "crash!!!";

    // Skips because exception is thrown
    console.log("after throw");
  }
  catch (exception) {
    console.log("catch");
  }
  finally {
    console.log("finally");
  }

  console.log("after");
}

test();
console.log("done");
```

Step 1 calls the function test and runs the try block. try is displayed in the console and an exception is thrown.
Step 2 runs the catch block. The exception thrown in the try block is handled and catch is displayed in the console.
Step 3 runs the finally block. finally is displayed in the console.
Step 4 runs the remaining lines of code. after and done are displayed in console on new lines.

Animation captions:

1. The test function throws an exception with the throw statement.

2. The catch block catches the exception.
3. The finally block always executes at the end of a try-catch.
4. Because the exception was handled by the catch block, the program continues to execute.

Figure 6.14.2: finally without a catch.

```
function test() {
   try {
      console.log("try");
      throw "crash!!!";

      // Skips because an exception was thrown
      console.log("after throw");
   }
   finally {
      console.log("finally");
   }

   // Skips because an exception was thrown
   console.log("after");
}

// Exception is not caught, so program halts!
test();
console.log("done");
```

```
try
finally
Uncaught crash!!!
```

Developers find `finally` blocks especially helpful when the code in the `catch` block might throw an exception, because the `finally` block will execute even if an exception is thrown in the `catch` block.

Figure 6.14.3: finally block executes when the catch block throws an exception.

```
function test() {
   try {
      console.log("try");
      throw "crash!!!";

      // Skips because exception is thrown
      console.log("after throw");
   }
   catch (exception) {
      console.log("catch");
      throw "oops!!!";
   }
   finally {
      console.log("finally");
   }

   // Doesn't execute because exception thrown in catch block
   console.log("after");
}

test();
console.log("done");
```

```
try
catch
finally
Uncaught oops!!!
```

PARTICIPATION ACTIVITY 6.14.5: try-catch-finally.

Refer to the `displayValues()` function below. The `Number.toFixed(n)` method returns a number with `n` decimal places.

```
function displayValues(tax, total) {
   try {
      console.log(tax.toFixed(1) + "%");
      console.log("$" + total.toFixed(2));
   }
   catch (ex) {
      console.log("error");
   }
   finally {
      console.log("********");
   }
}
```

1) What is output by the call `displayValues(5.44, 123)`?

○
```
5.4%
$123.00
********
```

○
```
5.4%
$123.00
```

○
```
5.4%
$123.00
error
********
```

2) What is output by the call `displayValues("5.44", 123)`?

○
```
5.4%
$123.00
********
```

○
```
$123.00
error
********
```

○
```
error
********
```

3) What is output by the call `displayValues(2.89, "hot dog")`?

○
```
2.9%
hot dog
********
```

○
```
2.9%
error
********
```

○
```
error
********
```

Error object

The `throw` statement can throw any expression, but developers commonly throw an Error object. The ***Error object*** represents a runtime error, which is an error that occurs when the program is executing. An Error object has two properties:

- `name` - The error's name.
- `message` - The error's message.

The `Error` constructor takes a message parameter. Ex:
`err = new Error("My error message.");`

JavaScript defines several other Error constructors, including:

- ***RangeError*** - Thrown when a numeric variable or parameter is outside the valid range.
- ***InternalError*** - Thrown when an internal error occurs in the JavaScript interpreter.
- ***TypeError*** - Thrown when a variable or parameter is not the expected data type.

Figure 6.14.4: findAverage() throws an Error, TypeError, and RangeError.

```
// Returns the average of the scores array
function findAverage(scores) {
   if (!Array.isArray(scores)) {
      throw new TypeError("Must supply an array.");
   }

   if (scores.length === 0) {
      throw new Error("Must supply at least one score.");
   }

   let sum = 0;
   scores.forEach(function(score) {
      if (!Number.isInteger(score)) {
         throw new TypeError("Score '" + score + "' is not an integer.");
      }
      if (score < 0) {
         throw new RangeError("Negative score encountered.");
      }
      sum += score;
   });
   return sum / scores.length;
}

try {
   let ave = findAverage([50, "cow"]);
}
catch (ex) {
   console.log(ex.name + ": " + ex.message);
}
```

```
TypeError: Score 'cow' is not an integer.
```

PARTICIPATION ACTIVITY 6.14.6: Error object.

Refer to the figure above.

1) The code below throws a RangeError.

```
ave = findAverage();
```

○ True

○ False

2) The code below outputs "TypeError".

```
try {
   ave = findAverage([]);
}
catch (ex) {
   console.log(ex.name);
}
```

- ○ True
- ○ False

3) The code below outputs "Negative score encountered."

```
try {
   ave = findAverage([5, 2,
-4]);
}
catch (ex) {
   console.log(ex.message);
}
```

- ○ True
- ○ False

PARTICIPATION ACTIVITY | 6.14.7: Practice throwing an Error object.

The `caesarCipher()` function below uses a Caesar cipher to encrypt or decrypt a message. The `message` parameter is the message being encrypted or decrypted. The `key` parameter is the number of characters to shift each letter in the message up or down the alphabet. Ex: A key of 1 shifts each letter up by 1, so "A" becomes "B", "B" becomes "C", etc. A key of -2 shifts each letter down by 2, so "C" becomes "A", "D" becomes "B", etc. Letters at the end of the alphabet wrap back to the beginning, and vice versa.

The `caesarCipher()` function is called to encrypt a saying of Caesar's. The same message is decrypted using the negative of the key.

Improve the `caesarCipher()` function by adding exceptions:

1. Throw a `TypeError` if the `message` parameter is not a string. Use JavaScript's `typeof` operator to check if `message` is a string:

```
if (typeof message === "string") // true if message is a string
```

2. Throw a `TypeError` if the `key` is not an integer. Use the `Number.isInteger()` method to check the `key`.
3. Throw a `RangeError` if the `key` is not between -25 and 25.

Use a `try-catch` statement to call `caesarCipher()` and verify the proper exception is thrown when using incorrect arguments. Ex: `caesarCipher(123, 5)` should throw a `TypeError` because 123 is not a string. In the `catch` block, output one of the following Caesar quotes:

1. "Men willingly believe what they wish." if the exception thrown is a `TypeError`.
2. "The die is cast." if the exception thrown is a `RangeError`.

CHALLENGE ACTIVITY | 6.14.1: Throwing and catching exceptions.

550544.4142762.qx3zqy7

Exploring further:

- try...catch (MDN)
- Error object (MDN)
- typeof operator (MDN)

6.15 LAB: JavaScript loops

This section's content is not available for print.

6.16 LAB: JavaScript password strength

This section's content is not available for print.

6.17 LAB: JavaScript arrays

This section's content is not available for print.

6.18 LAB: JavaScript game object

This section's content is not available for print.

6.19 LAB: JavaScript maps

This section's content is not available for print.

6.20 LAB: JavaScript number guessing game