

8.1 Regular expressions

Introduction to regular expressions

Programs often need to determine if a string conforms to a pattern. Ex: A user is asked for their phone number, and the program must recognize if the input is formatted like a phone number. Or perhaps a program needs to search through a large collection of DNA sequences and replace defective gene sequences with correct sequences. Developers use regular expressions to solve these types of problems.

A **regular expression** (often shortened to **regex**) is a string pattern that is matched against a string. Regular expressions may be defined with a **RegExp** object or between two forward slashes. Ex: `let re = new RegExp("abc");` or more simply: `let re = /abc/;` The pattern "abc" matches any string that contains "abc". Ex: "abcde" matches but "abd" does not. The `RegExp` method **test(str)** returns true if the string `str` matches the regex, and false otherwise.

PARTICIPATION ACTIVITY

8.1.1: Searching an array for the pattern 'ab'.



```
let words = ["ban", "babble", "make", "flab"];  
let re = /ab/;  
words.forEach(function(word) {  
  if (re.test(word)) {  
    console.log(word + " matches!");  
  }  
});
```

babble matches!
flab matches!

Animation content:

The following code snippet is displayed.

```
let words = ["ban", "babble", "make", "flab"];
```

```
let re = /ab/;
```

```
words.forEach(function(word) {  
  if (re.test(word)) {  
    console.log(word + " matches!");  
  }  
});
```

Step 5: The console displays: babble matches!

Step 6: The console displays: flab matches!

Animation captions:

1. Define a regular expression that matches words containing "ab".
2. Loop through each word in the words array.
3. `re.test("ban")` returns false because "ban" does not contain "ab".
4. `re.test("babble")` returns true because "babble" contains "ab".
5. `re.test("make")` returns false because "make" does not contain "ab".
6. `re.test("flab")` returns true because "flab" contains "ab".

PARTICIPATION ACTIVITY

8.1.2: Simple regular expressions.



1) The regex `/run/` matches the string "pruning".

- ☐ True
☐ False



2) The regex `/run/` matches the string "pruning".

- ☐ True
☐ False



3) What pattern would match the string "Regular Expression"?

- ☐ Exp
- ☐ exp

4) What value is **x**?

```
let re = /regex/;  
let x = re.test("regular  
expression");
```

- ☐ true
- ☐ false

5) What value is **x**?

```
let re = /c\/1/;  
let x = re.test("abc/123");
```

- ☐ true
- ☐ false

Special characters

Regular expressions use characters with special meaning to create more sophisticated patterns. The **+** character matches the preceding character at least once. Ex: **/ab+c/** matches one "a" followed by at least one "b" and one "c", so "abc" and "abbbbc" both match. However, "ac" does not match because the required "b" is missing.

Parentheses are used in a regex to match consecutive characters with *****, **+**, and **?**. Ex: **/(ab)+/** matches one or more "ab", so "abab" and "abbb" both match. However, "acb" does not match because the "c" is between "a" and "b".

Table 8.1.1: Selected special characters in regex patterns.

Character	Description	Example
*	Match the preceding character 0 or more times.	<code>/ab*c/</code> matches "abc", "abbbbc", and "ac".
+	Match the preceding character 1 or more times.	<code>/ab+c/</code> matches "abc" and "abbbbc" but not "ac".
?	Match the preceding character 0 or 1 time.	<code>/ab?c/</code> matches "abc" and "ac", but not "abbc".
^	Match at the beginning.	<code>/^ab/</code> matches "abc" but not "cab".
\$	Match at the end.	<code>/ab\$/</code> matches "cab" but not "abc".
	Match string on the left OR string on the right.	<code>/ab cd/</code> matches "abc" and "bcd".

**PARTICIPATION
ACTIVITY**

8.1.3: Regex with special characters.

1) What string does NOT match the regex `/grea*t/?`

- ☐ gret
- ☐ greaaat
- ☐ grat

2) What string does NOT match the regex `/w+hy/?`

- ☐ why
- ☐ hy
- ☐ wwwwhy

3) What string does NOT match the regex `/cat|bat|mat/`?

- ☐ `bbatt`
- ☐ `at`
- ☐ `mat`

4) What regex matches the string `"boom"`?

- ☐ `/ba?oom/`
- ☐ `/b\?oom/`
- ☐ `/bo?m/`

5) What regex matches the string `"sleep like a baby"`?

- ☐ `/^like/`
- ☐ `/like$/`
- ☐ `/^sleep/`

6) What regex matches the string `"that's easy"`?

- ☐ `/b?eas$/`
- ☐ `/b?sy$/`
- ☐ `/e+sy$/`

7) What regex matches the string `"breaeak"`?

- ☐ `/bre+a+k/`
- ☐ `/br(ea)+k/`
- ☐ `/er|bk/`

8) What regex matches the string
"what?"?

- ☐ /what?\$/
- ☐ /what\?\$/
- ☐ /what\$/

Character ranges

Square brackets are used in regular expressions to match any character in a range of characters. Ex: `/[aeiou]/` matches any vowel, and `/[0-9]/` matches any digit. The not operator (`^`) negates a range. Ex: `/[^str]/` matches any character except s, t, or r.

PARTICIPATION ACTIVITY

8.1.4: Regex with brackets.

Fill in the blank so the regex matches the description.

1) Match only the digits 0 through
5.

/[____]/

Check

Show answer

2) Match only the letters a through
f.

/[____]/

Check

Show answer

3) Match anything except a vowel (a, e, i, o, u). List characters in the regex alphabetically.



/ [____] /

Check

Show answer

Metacharacters

A **metacharacter** is a character or character sequence that matches a class of characters in a regular expression. Ex: The period character matches any single character except the newline character. So `/ab.c/` matches "abZc" and "ab2c", but not "abc" since the period must match a single character. Other metacharacters begin with a backslash.

Table 8.1.2: Selected metacharacters in regex patterns.

Metacharacter	Description	Example
.	Match any single character except newline.	/a.b/ matches "aZb" and "a b".
\w	Match any word character (alphanumeric and underscore).	/a\wb/ matches "aAb" and "a5b" but not "a b".
\W	Match any non-word character.	/a\Wb/ matches "a-b" and "a b" but not "aZb".
\d	Match any digit.	/a\db/ matches "a2b" and "a9b", but not "aZb".
\D	Match any non-digit.	/a\Db/ matches "aZb" and "a b", but not "a2b".
\s	Match any whitespace character (space, tab, form feed, line feed).	/a\s b/ matches "a b" but not "a4b".
\S	Match any non-whitespace character.	/a\S b/ matches "a!b" but not "a b".

PARTICIPATION
ACTIVITY

8.1.5: Metacharacters in regex.



Match the regex to a string the regex matches.

If unable to drag and drop, refresh the page.

/1\w+/

/\d\s\?/

/\d\./div>/1\s\d/

/\W\W\D/

/9\d+/

123break

https://learn.zybooks.com/zybook/CIS192_193_Spring_2024/chapter/8/print

Page 8 of 223

	923 break
	break1 9
	()break
	5!?5break
	0.break

Reset

Mode modifiers

A **mode modifier** (sometimes called a **flag**) changes how a regex matches and is placed after the second slash in a regex. Ex: `/abc*/i` specifies the mode modifier `i`, which performs case-insensitive matching.





Table 8.1.3: Selected mode modifiers.

Mode modifier	Description	Example
i	Case insensitivity - Pattern matches upper or lowercase.	<code>/aBc/i</code> matches "abc" and "AbC".
m	Multiline - Pattern with <code>^</code> and <code>\$</code> match beginning and end of any line in a multiline string.	<code>/^ab/m</code> matches the second line of "cab\nabc", and <code>/ab\$/m</code> matches the first line.
g	Global search - Pattern is matched repeatedly instead of just once.	<code>/ab/g</code> matches "ab" twice in "cababc".

PARTICIPATION
ACTIVITY

8.1.6: Regex with mode modifiers.



- 1) What string does NOT match the regex `/great/i`? 
- ☐ Great
 - ☐ greatT
 - ☐ TREAT
- 2) What string does NOT match the regex `/ly$/m`? 
- ☐ quick
 - ☐ most
first
 - ☐ quick
 - ☐ mostly
first
 - ☐ quick
 - ☐ most
firstly
- 3) How many times does the regex `/moo/g` find a match in "Moo the cow mooed at the moon." 
- ☐ 1
 - ☐ 2
 - ☐ 3
- 4) How many times does the regex `/moo/gi` find a match in "Moo the cow mooed at the moon." 
- ☐ 1
 - ☐ 2
 - ☐ 3

A criminal organization is using Reddit to communicate. To keep from being detected, the criminals are posting comments that look innocuous but use a secret pattern.

- The pattern contains one or more digits followed by any number of characters, followed by the word "star". Ex: "3stars" and "99 bright stars!" should both match.
- The letters in the word "star" may be separated by a single space. Ex: "1 blast ark" and "1 s t a r" should match.
- The comments can include upper or lowercase characters. Ex: "2 STar" should match.

Loop through the Reddit posts in the `posts` array and output to the console the lines that match the criminal's pattern. Use a single regex to identify the suspected posts. Hint: The 2nd, 3rd, and 5th lines should match the regex.

```
1 let posts = [  
2   "The starting time was 6pm.",  
3   "If the 2nd string QB gets hurt, they have no starting QB.",  
4   "My email is sis1@yahoo.com. Last are first.",  
5   "Stare into the abyss 1 time.",  
6   "90210 for Beverly Hills. Thick as TAR."  
7 ];  
8  
9 // Modify to output only lines that match regex  
10 posts.forEach(function(line) {  
11   console.log(line);  
12 });  
13
```

[Run JavaScript](#)[Reset code](#)

Your console output

```
The starting time was 6pm.  
If the 2nd string QB gets hurt, they have no starting QB.  
My email is sisl@yahoo.com. Last are first.  
Stare into the abyss 1 time.  
90210 for Beverly Hills. Thick as TAR.
```

► View solution

CHALLENGE ACTIVITY

8.1.1: Regular expressions.



550544.4142762.qx3zqy7

[Start](#)

Assign `re` with a regular expression that contains a letter (a-z or A-Z).

```
1 let re /* Your solution goes here */ ;
```

1

2

3

4

[Check](#)[Next](#)

Determining what matches

The `RegExp` method **`exec(str)`** determines what part of the string `str` matches a regex. The **`exec ()`** method returns a result array, or returns `null` if the pattern does not match.

Figure 8.1.1: Using `exec()` to discover what characters matched the regex.

```
let re = /t.+r/;
let result = re.exec("Raise the bar
high.");

if (result === null) {
    console.log("No match.");
}
else {
    // Index 0 is the full string that
    matched
    console.log(result[0]);    // the bar
}
```

Parentheses in a regex are used to "remember" different parts of a matched string. Ex: `/a(b+)c/` remembers anything matching `(b+)`, so "bbb" is remembered when applying the regex to "abbbc". The remembered parts are accessible from the result array returned by `exec()`. The first array element is the complete matched string, and the following elements are the remembered parts. If the regex contains no parentheses, the returned array contains only the complete string that matches.

Figure 8.1.2: Remembering matches in a regex.

```
let re = /(B.+)'s (.+day)/;
let result = re.exec("When is Becky's
birthday?");

// Index 0 is the full string that matched
console.log(result[0]);    // Becky's birthday

// Index 1 is the first remembered part
console.log(result[1]);    // Becky

// Index 2 is the second remembered part
console.log(result[2]);    // birthday
```



Twitter wants to know which hashtags are currently trending and what websites are tweeted most often. A selection of tweets are given in the `tweets` array. Create two regular expressions that will:

1. Extract all the hashtags used in the tweets. A hashtag begins with a pound sign and contains all following word characters. Ex: `#myHashTag`. Output each hashtag to the console.
2. Extract all the domain names from the URLs in the tweets. A URL begins with a protocol and double slash: `"http://"` or `"https://"`. The domain name is the string of characters immediately after the double slash and before the next forward slash (/). Ex: The domain name for `https://en.wikipedia.org/wiki/URL` is `en.wikipedia.org`. Output each domain name to the console.

Multiple hashtags and URLs may exist in a single tweet, so use the "g" mode modifier on both regexes and loop until the pattern is no longer found. To extract the domain name, use `.+?` to match the characters after the double slash and before the first slash. The `+` operator is "lazy" and matches as few characters as possible, whereas `+` matches as many characters as possible.

```
1 const tweets = [  
2   "Thank you to the Academy and the incredible cast & crew of #TheRe  
3   "@HardingCompSci department needs student volunteers for #HourOfCo  
4   "Checkout the most comfortable earbud on #Kickstarter and boost yo  
5   "Bootstrap 4 Cheat Sheet https://t.co/MFyKHvd50H",  
6   "Curious to see how #StephenCurry handles injury. http://mashable.  
7 ];  
8  
9 // Extract hashtags and domain names from URLs  
10 tweets.forEach(function(tweet) {  
11   console.log(tweet);  
12 });  
13
```

[Run JavaScript](#)[Reset code](#)

Your console output

Thank you to the Academy and the incredible cast & crew of #TheRevenant. #Oscar
@HardingCompSci department needs student volunteers for #HourOfCode https://hou
Checkout the most comfortable earbud on #Kickstarter and boost your #productivi
Bootstrap 4 Cheat Sheet https://t.co/MFyKHvd50H
Curious to see how #StephenCurry handles injury. http://mashable.com/2016/04/25

[► View solution](#)

What is output to the console?

1)

```
re = /\w+y/;  
result = re.exec("zing zany  
zone");  
console.log(result[0]);
```

- ☐ zing
- ☐ zing zany
- ☐ zany

2)

```
re = /\w+y/;  
result = re.exec("zing zane  
zone");  
console.log(result);
```

- ☐ undefined
- ☐ null
- ☐ false

3)

```
re = /\w+y/;  
result = re.exec("zing zany  
zone");  
console.log(result[1]);
```

- ☐ undefined
- ☐ null
- ☐ false

4)

```
re = /z(\w+)/;  
result = re.exec("zing zany  
zone");  
console.log(result[1]);
```

- ☐ zing
- ☐ ing
- ☐ ing zany zone

5)

```
re = /z\w+/g;
str = "zing zany zone";
result = re.exec(str);
while (result !== null) {
  console.log(result[0]);
  result = re.exec(str);
}
```

- ☐ zing
- ☐ zing
- ☐ zany
- ☐ zone
- ☐ Infinite loop

6)

```
re = /\d+?/;
str = "0123";
result = re.exec(str);
console.log(result[0]);
```

- ☐ 0
- ☐ 3
- ☐ 0123

CHALLENGE ACTIVITY

8.1.2: Determining what matches.

550544.4142762.qx3zqy7

[Start](#)

Write a statement to display the month from the regular expression match in result. Note: The given date is in month, day, year order.

```
1 const re = /(\d+)\V(\d+)\V(\d+)/;  
2 const date = "4/15/91"; // Code will also be tested with value "11/2/2  
3 const result = re.exec(date);  
4  
5 /* Your solution goes here */  
6
```

1

2

3

[Check](#)[Next](#)

String methods that use regex

Several String methods work with regular expressions:

- **match()** returns an array of the matches made when matching the string against a regex.
- **replace()** returns a new string that replaces matching strings with a replacement string.
- **search()** returns the index of the first match between the regex and the given string, or -1 if no match is found.
- **split()** returns an array of strings created by separating the string into substrings based on a regex.

Exploring further:

- [Regular Expressions \(MDN\)](#).
- [RegExp](#) - For testing regular expressions

8.2 Classes

Constructor functions

A JavaScript **class** is a special function, called a constructor function, that defines properties and methods from which an object may inherit. A **constructor function** is a function that initializes a new object when an object is instantiated with the **new** operator. The **this** keyword refers to the current object and is used to access properties inside the class.

PARTICIPATION ACTIVITY

8.2.1: Creating a Person class with a constructor function.



```
function Person(name, age) {  
  this.name = name;  
  this.age = age;  
  
  this.sayHello = function() {  
    console.log("Hello, " + this.name);  
  };  
};  
  
let bob = new Person("Bob", 21);  
let sue = new Person("Sue", 40);  
  
bob.sayHello();  
sue.sayHello();
```

bob	name: "Bob"	age: 21	sayHello: func
sue	name: "Sue"	age: 40	sayHello: func

Hello, Bob
Hello, Sue

Animation content:

The following code snippet is displayed.

```
function Person(name, age) {  
  this.name = name;  
  this.age = age;  
  
  this.sayHello = function() {  
    console.log("Hello, " + this.name);  
  };  
};  
  
let bob = new Person("Bob", 21);  
let sue = new Person("Sue", 40);  
  
bob.sayHello();  
sue.sayHello();
```

Step 2: The constructor function is passed arguments "Bob" and 21.

Step 3: The constructor function is passed arguments "Sue" and 40.

Step 4: The console outputs:

Hello, Bob
Hello, Sue

Animation captions:

1. A constructor function called `Person` takes two parameters: `name` and `age`.
2. New object `bob` is instantiated by calling the `Person` constructor function.
3. New object `sue` is instantiated by calling the `Person` constructor function.
4. `bob` and `sue`'s `sayHello()` methods are called and output to the console.

**PARTICIPATION
ACTIVITY**

8.2.2: JavaScript classes.

Refer to the `Person` constructor function from the animation above.

- 1) "Vail" is output to the console.

```
let student = new
Person("Vail", 30);
console.log(student.name);
```

- ☐ True
- ☐ False

- 2) 55 is output to the console.

```
let instructor =
Person("Rodriguez", 55);
console.log(instructor.age);
```

- ☐ True
- ☐ False

- 3) What outputs the `Person`'s age to the console when `showAge()` is called?

```
this.showAge = function() {
  console.log("I am " + _____
+ " years old.");
};
```

- ☐ `this.name`
- ☐ `this.age`

Prototype object

Every JavaScript object is associated with a second object called the prototype. The **prototype** object contains properties that an associated object inherits when the associated object is created.

When an object is instantiated with the **new** keyword, the object is assigned the **prototype** property that is associated with the constructor function. Ex: When a date object is instantiated with **new Date()**, the date object is assigned the **Date.prototype** from the **Date** constructor function. All date objects have access to the same methods like **getDate()** and **getFullYear()** because **getDate()** and **getFullYear()** are methods assigned to **Date.prototype**.

Developers often assign methods to the class' **prototype** instead of the constructor function because prototype methods are more memory efficient. The JavaScript interpreter must allocate memory for each method defined in a constructor function, but a prototype method is only allocated memory once and is shared by all objects created with the same constructor function.

Figure 8.2.1: Assigning methods to the prototype object is more memory efficient.

```
function Person(name, age) {  
  this.name = name;  
  this.age = age;  
  
  // Regular method  
  this.sayHello = function() {  
    console.log("Hello, " +  
this.name);  
  };  
};  
  
let bob = new Person("Bob", 21);  
let sue = new Person("Sue", 40);  
let pam = new Person("Pam", 32);
```

```
function Person(name, age) {  
  this.name = name;  
  this.age = age;  
};  
  
// Prototype method  
Person.prototype.sayHello =  
function() {  
  console.log("Hello, " +  
this.name);  
};  
  
let bob = new Person("Bob", 21);  
let sue = new Person("Sue", 40);  
let pam = new Person("Pam", 32);
```

bob	name: "Bob", age: 21, sayHello: func
sue	name: "Sue", age: 40, sayHello: func
pam	name: "Pam", age: 32, sayHello: func

Method duplicated
multiple times in memory

bob	name: "Bob", age: 21
sue	name: "Sue", age: 40
pam	name: "Pam", age: 32
Person.prototype	sayHello: func

Prototype method
appears once in memory

PARTICIPATION
ACTIVITY

8.2.3: The prototype object.

Refer to the `Person` constructor function.

```
function Person(name, age) {  
  this.name = name;  
  this.age = age;  
};
```

- 1) What creates a prototype method for `Person` called `showAge`?

```
_____ = function() {  
  console.log("I am " +  
  this.age + " years old.");  
};
```

- ☐ `this.showAge`
- ☐ `Person.showAge`
- ☐ `Person.prototype.showAge`

- 2) How is the `showAge()` method called to get Professor X's age?

```
Person.prototype.showAge =  
function() {  
  console.log("I am " + this.age +  
  " years old.");  
};  
  
let profX = new Person("Professor  
X", 60);
```

- ☐ `profX.showAge();`
- ☐ `profX.prototype.showAge();`
- ☐ `Person.prototype.showAge();`

3) What is output to the console?



```
Person.prototype.species =  
"Homo sapiens";  
let james = new  
Person("James", 16);  
console.log(james.species);
```

- ☐ 16
- ☐ Homo sapiens
- ☐ undefined

4) What is output to the console?



```
Person.prototype.toString =  
function() {  
    return this.name + " - " +  
this.age;  
};  
  
let james = new  
Person("James", 16);  
console.log(james);
```

- ☐ [object Object]
- ☐ Nothing
- ☐ James - 16

5) What produces the console output below?



```
console.log(Person.prototype._____);
```

```
let Person = function(name, age) {  
    this.name = name;  
    this.age = age;  
};
```

- ☐ Person
- ☐ constructor
- ☐ function



The JavaScript code defines a constructor function for a **Student** class.

1. Add a **Course** constructor function that takes a **title** as a parameter. The **Course** class should have two properties: a **title** and **students** array. The constructor function should initialize the **title** property to the **title** parameter and initialize the **students** array to an empty array.
2. Add a method to the constructor function called **addStudent** that takes a single **student** parameter. The **addStudent()** method should push the **student** to the back of the **students** array property.
3. Add a prototype method called **displayStudents** that takes no parameters. The **displayStudents()** method should output the course title and all students' names and GPAs to the console.
4. Instantiate two different courses with any titles, and add a few students (**Student** objects) to the courses using the **addStudent()** method.
5. Display all the students in the two courses using the **displayStudents()** method.

```
1 // Student class
2 function Student(name, gpa) {
3     this.name = name;
4     this.gpa = gpa;
5 }
6
```

[Run JavaScript](#)[Reset code](#)

Your console output

► [View solution](#)

Private properties and closures

A **private property** is a property that is only accessible to object methods but is not accessible from

outside the class. Private properties may be simulated in JavaScript by creating local variables in a constructor function with getters and setters to get and set the properties.

Figure 8.2.2: Creating a private property called "secret" with a getter and setter.

```
function Person(name, age) {  
  this.name = name;  
  this.age = age;  
  
  // private  
  let secret;  
  
  // public methods have access to private  
  properties  
  this.setSecret = function(s) {  
    secret = s;  
  };  
  
  this.getSecret = function() {  
    return secret;  
  };  
};  
  
let bob = new Person("Bob", 21);  
bob.setSecret("I have mutant powers!");  
console.log(bob.getSecret()); // I have mutant  
powers!  
console.log(bob.secret);      // undefined
```

Private class variables can be simulated in JavaScript because of closures. A **closure** is a special object that is automatically created and maintains a function's local variables and values after the function has returned. The `secret` variable defined in the `Person` constructor function above is remembered because of a closure that remembers the `Person` constructor function's local variables.

PARTICIPATION ACTIVITY

8.2.5: Private properties and closures.



Refer to the figure above.

- 1) The `getPartialSecret()` method below returns part of the `secret` property.



```
this.getPartialSecret =  
function() {  
    return this.secret.substr(0,  
5) + "...";  
};
```

- ☐ True
- ☐ False

- 2) The `getPartialSecret()` prototype method returns part of the `secret` property.



```
Person.prototype.getPartialSecret  
= function() {  
    return secret.substr(0, 5) +  
    "...";  
};
```

- ☐ True
- ☐ False

- 3) What is output to the console?



```
bob.setSecret("I can't  
swim.");  
bob.secret = "I'm  
ambidextrous.";  
console.log(bob.getSecret());
```

- ☐ I'm ambidextrous.
- ☐ I can't swim.

- 4) Local variables in a function maintain values after the function terminates because of _____.



- ☐ closures
- ☐ private variables

Inheritance

Inheritance creates a new child class that adopts properties of a parent class. Ex: A Student class (child) may inherit from a Person class (parent), so a Student class has the same properties of a Person and may add even more properties.

Implementing inheritance in JavaScript is more complicated than most other programming languages. For a child class to inherit from a parent class, 3 operations must be performed:

- 1. The child class calls the parent class' constructor function from the child's constructor function using the `call()` method.
- 2. The **Object.create()** method copies the parent's prototype, and the new copy is assigned to the child's prototype to give the child class the same functionality as the parent class.
- 3. The child class' `prototype.constructor` is explicitly set to the child's constructor function.

PARTICIPATION
ACTIVITY

8.2.6: Student class inherits properties from the Person class.

```
// Parent class
function Person(name) {
  this.name = name;
}

Person.prototype.sayHello = function() {
  console.log("Hello. My name is " + this.name);
};
Person.prototype.sayGoodbye = function() {
  console.log("Goodbye!");
};

// Child class
function Student(name, gpa) {
  Person.call(this, name);
  this.gpa = gpa;
}

// Duplicate functionality of parent
Student.prototype = Object.create(Person.prototype);
Student.prototype.constructor = Student;

// Replace the parent's sayHello with a new method
Student.prototype.sayHello = function() {
  console.log("Hi, I'm " + this.name + " with a "
    + this.gpa + " GPA!");
}
```

Person	func
Person.prototype	constructor: Person() func sayHello: func sayGoodbye: func
Student	func
Student.prototype	constructor: Student() func sayHello: func: "Hi, I'm _ with a sayGoodbye: func
bob	name: "Bob" gpa: 3.5

```
let bob = new Student("Bob", 3.5);  
bob.sayHello();  
bob.sayGoodbye();
```

Hi, I'm Bob with a 3.5 GPA!
Goodbye!

Animation content:

The following code snippet is displayed.

```
function Person(name) {  
  this.name = name;  
}
```

```
Person.prototype.sayHello = function() {  
  console.log("Hello. My name is " + this.name);  
};  
Person.prototype.sayGoodbye = function() {  
  console.log("Goodbye!");  
};
```

```
// Child class  
function Student(name, gpa) {  
  Person.call(this, name);  
  this.gpa = gpa;  
}
```

```
// Duplicate functionality of parent  
Student.prototype = Object.create(Person.prototype);  
Student.prototype.constructor = Student;
```

```
// Replace the parent's sayHello with a new method  
Student.prototype.sayHello = function() {  
  console.log("Hi, I'm " + this.name + " with a "  
    + this.gpa + " GPA!");  
}
```

```
let bob = new Student("Bob", 3.5);  
bob.sayHello();
```

```
bob.sayGoodbye();
```

Step 6: "Hi, I'm _ with a _ GPA!" is added to student.prototype.sayHello.

Step 9: The console output reads:

Hi, I'm Bob with a 3.5 GPA!

Goodbye!

Animation captions:

1. Constructor function Person takes a name parameter. Person.prototype is created with a constructor property that references the Person constructor function.
2. Methods sayHello() and sayGoodbye() are added to Person.prototype.
3. Constructor function Student takes a name and gpa parameter. Student.prototype is created with a constructor property that references the Student constructor function.
4. Object.create() creates a new Person.prototype object, and the new object is assigned to Student.prototype.
5. Student.prototype.constructor is reset back to the Student constructor function.
6. A new sayHello() method is defined that outputs the student's name and GPA.
7. Object bob is instantiated using the Student function constructor.
8. Person constructor function is called with "this" referencing the current object. Person constructor function initializes name property, and Student constructor function initializes gpa property.
9. bob's sayHello() and sayGoodbye() methods are called.

PARTICIPATION ACTIVITY

8.2.7: Inheritance.

1) JavaScript supports _____ inheritance, meaning a child class may only inherit from a single parent class.

- ☐ single
- ☐ multiple
- ☐ virtual

- 2) Suppose the program from the animation above was missing the call to `Person.call()` in the `Student` constructor function. What would be output to the console by the code below?



```
let sue = new Student("Sue",  
3.9);  
sue.sayHello();
```

- ☐ Hi, I'm Sue with a 3.9 GPA!
 - ☐ Hi, I'm undefined with a 3.9 GPA!
 - ☐ No output: The program throws an exception.
- 3) Suppose the code below was added to the end of the program in the animation above. What would be output to the console by `bob.highFive()`?



```
Person.prototype.highFive =  
function() {  
  console.log("High five!");  
};  
  
bob.highFive();
```

- ☐ No output: An exception is thrown.
- ☐ Hi, I'm Bob with a 3.5 GPA!
- ☐ High five!

PARTICIPATION ACTIVITY

8.2.8: Practice with inheritance and private properties.



The JavaScript code defines a **Game** class and two methods.

1. Add a **VideoGame** class and all the necessary code so **VideoGame** inherits from the **Game** class.
2. Add a private variable to the **VideoGame** class called **totalPoints**, and initialize **totalPoints** to 0.
3. Add a getter method called **getScore()** to get the **totalPoints** variable.
4. Add a method called **addToScore(points)** that adds the **points** to **totalPoints**.
5. Instantiate a new **VideoGame** object with the title "Pac-Man". Call the appropriate methods to:
 1. Start playing the game.
 2. Show the score (should be 0).
 3. Add 20 points.
 4. Add 50 points.
 5. Show the score (should be 70).
 6. Stop playing the game.

©zyBooks 04/15/24 16:42 2071381
Marco Aguilar
CIS192_193_Spring_2024

```
1 function Game(title) {  
2     this.title = title;  
3 }  
4  
5 Game.prototype.startPlaying = function() {  
6     console.log("Now playing " + this.title + "!");  
7 };  
8  
9 Game.prototype.stopPlaying = function() {  
10    console.log("Taking a break.");  
11 };  
12
```

[Run JavaScript](#)[Reset code](#)

Your console output

► [View solution](#)

**CHALLENGE
ACTIVITY**

8.2.1: Classes.



550544.4142762.qx3zqy7

[Start](#)

Create a Rabbit class with properties name and weight passed to the constructor.

```
1  
2 /* Your solution goes here */  
3
```

1

2

3

4

[Check](#)[Next](#)

Classes and inheritance in EcmaScript 6

EcmaScript 6 simplifies class creation and inheritance, introducing syntax that looks more familiar to a Java or C# programmer. Although the syntax is different, the underlying prototype model is not changed.

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  sayHello() {
    console.log("Hello. My name is " + this.name + ".");
  }
}

// Student inherits from Person
class Student extends Person {
  constructor(name, age, gpa) {
    super(name, age); // Call parent constructor
    this.gpa = gpa;
  }

  // Override parent's sayHello method
  sayHello() {
    console.log("Hi, I'm " + this.name + " with a " + this.gpa + " GPA!");
  }
}
```

Exploring further:

- [JavaScript Object Constructors](#) from w3schools
- [Closures \(MDN\)](#)

8.3 Classes (ES6)

Classes in EcmaScript 6

EcmaScript 6 (ES6) simplifies class declarations, introducing syntax that looks more familiar to a Java or C# programmer. Although the syntax is different, the underlying prototype model is not changed.

A class is declared by using the **class keyword** followed by a class name. The class is implemented by declaring methods within braces. Each method declaration is similar to a function declaration, but without the **function** keyword. The method name **constructor()** is reserved for the class constructor.

PARTICIPATION ACTIVITY

8.3.1: CityState ES6 class.

```
class CityState {  
  constructor(city, state) {  
    this.city = city;  
    this.state = state;  
  }  
  
  toHTML() {  
    return this.city + ",&nbsp;" + this.state;  
  }  
}  
  
let Austin = new CityState("Austin", "Texas");  
let Madison = new CityState("Madison", "Wisconsin");  
console.log(Austin.toHTML());
```

Austin:

city: "Austin"
state: "Texas"

Madison:

city: "Madison"
state: "Wisconsin"

Austin, Texas

Animation content:

The following code snippet is displayed.

```
class CityState {  
  constructor(city, state) {  
    this.city = city;  
    this.state = state;  
  }  
  
  toHTML() {  
    return this.city + ",&nbsp;" + this.state;  
  }  
}
```

```
}
```

```
let Austin = new CityState("Austin", "Texas");  
let Madison = new CityState("Madison", "Wisconsin");  
console.log(Austin.toHTML());
```

Step 1: The constructor method creates the city and state parameters.

Step 2: Object Austin contains parameters city: "Austin" and state: "Texas".

Step 3: Object Madison contains parameters city: "Madison" and state: "Wisconsin".

Step 4: The call toHTML() returns "Austin, Texas"

Animation captions:

1. The CityState class is declared with a constructor method and a toHTML() method.
2. new CityState is used to construct Austin, an instance of CityState. The method named "constructor" is called automatically.
3. The Madison object is constructed similarly.
4. The Austin object's toHTML() is called to return an HTML string containing the city and state name.

PARTICIPATION ACTIVITY

8.3.2: Classes.



Refer to the **CityState** class in the animation above.

1) Which line properly constructs a `CityState` instance, referenced by a variable named `Boise`, for Boise, Idaho?

- ☐

```
CityState Boise = new  
CityState("Boise",  
"Idaho");
```
- ☐

```
let Boise =  
CityState("Boise",  
"Idaho");
```
- ☐

```
let Boise = new  
CityState("Boise",  
"Idaho");
```

2) If `Boise` is an instance of `CityState`, the statement `console.log(Boise.state);` _____.

- ☐ logs `Boise`'s state, Idaho, to the console
- ☐ logs nothing, since `state` is a private property
- ☐ throws an exception. The proper statement is `console.log(Boise.getState());`

- 3) What is wrong with the following attempt to declare a `toString()` method for `CityState`?



```
class CityState {  
    constructor(city, state) {  
        this.city = city;  
        this.state = state;  
    }  
  
    toHTML() {  
        return this.city +  
        ",&nbsp;" + this.state;  
    }  
  
    function toString() {  
        return this.city + ", "  
+ this.state;  
    }  
}
```

- ☐ `toString()` requires at least 1 parameter.
- ☐ The "function" keyword must be removed.
- ☐ The name "toString" is reserved and must be changed.

Inheritance

The ***extends keyword*** allows one class to inherit from another. In the inheriting class' constructor, calling the ***super()*** function calls the parent class' constructor. ***super ()*** must be called before using the ***this*** keyword in the inheriting class' constructor.

Figure 8.3.1: Person class and inheriting Student class.

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    sayHello() {
        console.log("Hello. My name is " + this.name + ".");
    }
}

// Student inherits from Person
class Student extends Person {
    constructor(name, age, gpa) {
        super(name, age); // Call parent constructor
        this.gpa = gpa;
    }

    // Override parent's sayHello method
    sayHello() {
        console.log("Hi, I'm " + this.name + " with a " + this.gpa + " GPA!");
    }
}
```

**PARTICIPATION
ACTIVITY**

8.3.3: Inheritance.

Suppose a `GraduateStudent` class, that inherits from `Student` is being defined. The class adds a `status` member, which is set to either "Masters" or "PhD". Match the blank lines to the appropriate code pieces.

```
class GraduateStudent __ (1) __ {
    constructor(name, age, gpa, mastersOrPhd) {
        __ (2) __ (name, age, gpa);
        __ (3) __ = mastersOrPhd;
    }

    // Override parent's sayHello method
    sayHello() {
        __ (4) __ ("Hi, I'm " + this.name + ", a " + this.status +
            " student with a " + this.gpa + " GPA!");
    }
}
```

If unable to drag and drop, refresh the page.

2

3

1

4

`this.status``console.log``extends Student``super`

Reset

Getters and setters

A class method declaration preceded by the **get keyword** defines a getter method for a property. A class method declaration preceded by the **set keyword** defines a setter method for a property. Defining either a getter or setter method named X, adds a property named X to each class instance. Ex: A **Square** class may have a getter method declared as:

```
get area() {  
    return this.width * this.width;  
}
```

After creating a **Square** instance named **s1**, the expression **s1.area**, without parentheses, gets the square's area.

A get method must not have parameters. A set method must have 1 parameter. A property can be defined via a getter only, a setter only, or both a getter and setter.

PARTICIPATION ACTIVITY

8.3.4: Getters and setters can be used to convert an angle between degrees and radians.



```
class Angle {  
    constructor(angleRadians) {  
        this.radians = angleRadians;  
    }  
}
```

angle1:

radians: 4.71238898038469

```
}

get degrees() {
    return this.radians * 180.0 / Math.PI;
}

set degrees(angleDegrees) {
    this.radians = angleDegrees * Math.PI / 180.0;
}

let angle1 = new Angle(Math.PI);
console.log(angle1.degrees);
angle1.degrees = 270.0;
console.log(angle1.radians);
```

console:

```
180
4.71238898038469
```

Animation content:

The following code snippet is displayed.

```
class Angle {
    constructor(angleRadians) {
        this.radians = angleRadians;
    }

    get degrees() {
        return this.radians * 180.0 / Math.PI;
    }

    set degrees(angleDegrees) {
        this.radians = angleDegrees * Math.PI / 180.0;
    }
}

let angle1 = new Angle(Math.PI);
console.log(angle1.degrees);
angle1.degrees = 270.0;
console.log(angle1.radians);
```

Class Angle is created with a constructor, get degrees() method, and a set degrees(angleDegrees) method.

Step 1: Angle1 stores 3.141592653589793

Step 2: The angle converted from radians to degrees returns 180.

Step 3: Angle1 now stores 4.71238898038469.

Step 4: the code outputs the radians of angle1 to the console which is 4.7124.

Animation captions:

1. angle1 is constructed as an instance of the Angle class, storing the angle in radians.
2. Accessing angle1's "degrees" property calls the corresponding getter. The angle, converted from radians to degrees, is returned.
3. Assigning to the "degrees" property calls the setter to convert the angle to radians.
4. "radians" is the angle1 object's only numerical property. The get and set allow "degrees" to be accessed like a property, even though the object never internally stores the angle in degrees.

PARTICIPATION ACTIVITY

8.3.5: Getter and setter methods.

1) If a getter method for property X is added to a class, a setter method for X ____.

- ☐ must also be added
- ☐ can be added, but is not required
- ☐ cannot be added

2) A setter method ____.

- ☐ requires 0 parameters
- ☐ requires 1 parameter
- ☐ can have any number of parameters

3) Suppose `obj` is an instance of a class with a property named `value`, which has both a getter and setter. The statement below calls the ____.



```
obj.value += 10;
```

- ☐ getter only
- ☐ setter only
- ☐ getter first, then the setter
- ☐ setter first, then the getter

Static methods

A **static method** is a method that can be called without creating an instance of the class. A static method is declared with the **static** keyword preceding the method name. The method is called with the syntax: `ClassName.methodName(arguments)`.

PARTICIPATION ACTIVITY

8.3.6: StringOps class with static methods.



```
class StringOps {
  static isLowerCase(str) {
    return str.toLowerCase() === str;
  }

  static countChar(str, char) {
    let count = 0;
    for (let i = 0; i < str.length; i++) {
      if (str.charAt(i) === char) {
        count++;
      }
    }
    return count;
  }
}

console.log(StringOps.isLowerCase("abc"));
console.log(StringOps.isLowerCase("zyBooks"));
console.log(StringOps.countChar("zyBooks", "o"));
```

console:

```
true
false
2
```

Animation content:

The following code snippet is displayed.

```
class StringOps {  
  static isLowerCase(str) {  
    return str.toLowerCase() === str;  
  }  
  
  static countChar(str, char) {  
    let count = 0;  
    for (let i = 0; i < str.length; i++) {  
      if (str.charAt(i) === char) {  
        count++;  
      }  
    }  
    return count;  
  }  
}  
  
// Outputs true  
console.log(StringOps.isLowerCase("abc"));  
  
// Outputs false  
console.log(StringOps.isLowerCase("zyBooks"));  
  
// Outputs 2  
console.log(StringOps.countChar("zyBooks", "o"));
```

Class StringOps is created with a static isLowerCase() method that checks if the passed in string is all lowercase and the countChar() method that counts the occurrence of a specific letter in a string.

Animation captions:

1. The StringOps class has a static isLowerCase() method, which is called with the syntax: StringOps.isLowerCase(...).
2. StringOps.countChar("zyBooks", "o") is called to count the number of 'o' characters in the string "zyBooks".

**PARTICIPATION
ACTIVITY**

8.3.7: StringOps class and static method concepts.



- 1) The following is a valid static method declaration that could be inserted into the `StringOps` class:



```
static getClassName() {  
    return "StringOps";  
}
```

- ☐ True
- ☐ False
- 2) A class can have both static and non-static methods.
- ☐ True
- ☐ False
- 3) Non-static methods can also be called with the `ClassName.methodName(arguments)` syntax.
- ☐ True
- ☐ False



Exploring further:

- [Classes \(MDN\)](#).

**CHALLENGE
ACTIVITY**

8.3.1: Classes ES6.



550544.4142762.qx3zqy7

[Start](#)

Create a Puppy class with a constructor that takes two arguments and initializes properties name and gender in that order.

```
1  
2 /* Your solution goes here */  
3
```

1

2

3

4

[Check](#)[Next](#)

8.4 Classes (ES13)

Classes in EcmaScript 2022

EcmaScript 2022 (ES13) introduces class syntax that differs in some ways from earlier versions of EcmaScript. Although the syntax is different, the underlying prototype model is not changed.

A class is declared by using the **class keyword** followed by a class name. The class is implemented by declaring fields and methods within braces. A **field** is a variable that stores data for a class. Each method declaration is similar to a function declaration, but without the **function** keyword. The method name **constructor()** is reserved for the class constructor.

PARTICIPATION
ACTIVITY

8.4.1: City class.



```
class City {
  name;
  state;

  constructor(name, state) {
    this.name = name;
    this.state = state;
  }

  toHTML() {
    return this.name + ",&nbsp;" + this.state;
  }
}

let city1 = new City("Austin", "Texas");
let city2 = new City("Madison", "Wisconsin");
console.log(city1.toHTML());
```

city1

name: "Austin"

state: "Texas"

city2

name: "Madison"

state: "Wisconsin"

Austin,&nbsp;Texas

Animation content:

The following code snippet is displayed.

```
class CityState {
  name;
  state;

  constructor(city, state) {
    this.city = city;
    this.state = state;
  }

  toHTML() {
    return this.city + ",&nbsp;" + this.state;
  }
}

let city1 = new CityState("Austin", "Texas");
let city2 = new CityState("Madison", "Wisconsin");
console.log(city1.toHTML());
```

Step 1: The constructor method creates the city and state parameters.

Step 2: Object city1 contains arguments city: "Austin" and state: "Texas".

Step 3: Object Madison contains arguments city: "Madison" and state: "Wisconsin".

Step 4: The call toHTML() returns "Austin, Texas"

Animation captions:

1. The City class is declared with two fields, a constructor method, and a toHTML() method.
2. new City calls the method named "constructor" to construct city1, a City instance. The constructor() method assigns the object's name and state fields.
3. The city2 object is constructed similarly.
4. The call to city1.toHTML() returns an HTML string containing the name and state fields.

PARTICIPATION ACTIVITY

8.4.2: Classes.



Refer to the `City` class in the animation above.

1) Which line properly constructs a `City` instance, referenced by a variable named `homeTown`, for Boise, Idaho?



- ☐ `City homeTown = new City("Boise", "Idaho");`
- ☐ `let homeTown = City("Boise", "Idaho");`
- ☐ `let homeTown = new City("Boise", "Idaho");`

2) If `homeTown` is an instance of `City`, the statement `console.log(homeTown.state);` ____.

- ☐ logs `homeTown`'s state, Idaho, to the console
- ☐ logs nothing, since `state` is a private field
- ☐ throws an exception. The proper statement is `console.log(homeTown.getState());`

3) What is wrong with the following attempt to declare a `toString()` method for `City`?

```
function toString() {  
    return this.name + ", " +  
    this.state;  
}
```

- ☐ `toString()` requires at least one parameter.
- ☐ The "function" keyword must be removed.
- ☐ The name "toString" is reserved and must be changed.

Private fields and methods

By default, fields and methods are public, meaning the fields and methods are accessible from outside the class. Fields and methods can be made private, or inaccessible from outside the class, by prefixing the field or method name with `#`. Ex: `#privateField` and `#privateMethod()`.

The `City` class in the figure below declares a private field called `#foundingYear`. The public fields `name` and `state` are not declared because, unlike private fields, public field declarations are not required.

Figure 8.4.1: City class with private field.

```
class City {  
    #foundingYear;  
  
    constructor(name, state, foundingYear)  
    {  
        this.name = name;  
        this.state = state;  
        this.#foundingYear = foundingYear;  
    }  
  
    toString() {  
        return this.name + ", " +  
        this.state +  
        " (" + this.#foundingYear + ")";  
    }  
}
```

**PARTICIPATION
ACTIVITY**

8.4.3: Private fields and methods.

Refer to the figure above.

- 1) What does the code output to the console?

```
let myCity = new  
City("Norfolk", "Virginia",  
1682);  
console.log(myCity.toString());
```

- ☐ Norfolk, Virginia
- ☐ Norfolk, Virginia (#1682)
- ☐ Norfolk, Virginia (1682)

- 2) What does the code output to the console?



```
let myCity = new City("Norfolk",  
"Virginia", 1682);  
console.log(myCity.#foundingYear);
```

- ☐ 1682
- ☐ Nothing
- ☐ An error message

- 3) The methods below are added to `City`. What is missing from `getAge()`?



```
#getCurrentYear() {  
  const today = new Date();  
  return today.getFullYear();  
}  
  
getAge() {  
  return _____ -  
  this.#foundingYear + " years  
old";  
}
```

- ☐ `#getCurrentYear()`
- ☐ `this.#getCurrentYear()`
- ☐ `this.getCurrentYear()`

Getters and setters

A class method declaration preceded by the **get keyword** defines a getter method for a property. A class method declaration preceded by the **set keyword** defines a setter method for a property. Defining either a getter or setter method named X, adds a property named X to each class instance. Ex: A **Square** class may have a getter method declared as:

```
get area() {  
  return this.width * this.width;  
}
```

After creating a **Square** instance named `s1`, the expression `s1.area`, without parentheses, gets the square's area.

A get method must not have parameters. A set method must have one parameter. A property can be defined via a getter only, a setter only, or both a getter and setter.

**PARTICIPATION
ACTIVITY****8.4.4: Using getters and setters to convert between degrees and radians.**

```
class Angle {
  constructor(angleRadians) {
    this.radians = angleRadians;
  }

  get degrees() {
    return this.radians * 180.0 / Math.PI;
  }

  set degrees(angleDegrees) {
    this.radians = angleDegrees * Math.PI / 180.0;
  }
}

let angle1 = new Angle(Math.PI);
console.log(angle1.degrees);
angle1.degrees = 270.0;
console.log(angle1.radians);
```

angle1

radians: 4.71238898038469

console:

180
4.71238898038469**Animation content:**

The following code snippet is displayed.

```
class Angle {
  constructor(angleRadians) {
    this.radians = angleRadians;
  }

  get degrees() {
    return this.radians * 180.0 / Math.PI;
  }

  set degrees(angleDegrees) {
    this.radians = angleDegrees * Math.PI / 180.0;
  }
}
```

```
let angle1 = new Angle(Math.PI);  
console.log(angle1.degrees);  
angle1.degrees = 270.0;  
console.log(angle1.radians);
```

Class Angle is created with a constructor, get degrees() method, and a set degrees(angleDegrees) method.

Step 1: Angle1 stores 3.141592653589793

Step 2: The angle converted from radians to degrees returns 180.

Step 3: Angle1 now stores 4.71238898038469.

Step 4: the code outputs the radians of angle1 to the console which is 4.7124.

Animation captions:

1. angle1 is constructed as an instance of the Angle class, storing the angle in radians.
2. Accessing angle1's "degrees" property calls the corresponding getter. The angle, converted from radians to degrees, is returned.
3. Assigning to the "degrees" property calls the setter to convert the angle to radians.
4. "radians" is the angle1 object's only field. The get and set allow "degrees" to be accessed like a field, even though the object never internally stores the angle in degrees.

PARTICIPATION ACTIVITY

8.4.5: Getter and setter methods.

1) If a getter method for property X is added to a class, a setter method for X ____.

- ☐ must also be added
- ☐ can be added, but is not required
- ☐ cannot be added

2) A setter method ____.

- ☐ requires no parameters
- ☐ requires one parameter
- ☐ can have any number of parameters

3) Suppose `obj` is an instance of a class with a property named `value`, which has both a getter and setter. The statement below calls the ____.

```
obj.value += 10;
```

- ☐ getter only
- ☐ setter only
- ☐ getter first, then the setter
- ☐ setter first, then the getter

Static properties and methods

A **static property** is a property that can be accessed without creating a class instance. A **static method** is a method that can be called without creating a class instance. Static properties and methods are declared with the **static** keyword preceding the property's field or method declaration. A static property is accessed with the syntax: `ClassName.propertyName`. A static method is called with the syntax: `ClassName.methodName(arguments)`.

PARTICIPATION ACTIVITY

8.4.6: StringOps class with static methods.

```
class StringOps {  
    static description = "String operations";  
  
    static isLowerCase(str) {  
        return str.toLowerCase() === str;  
    }  
  
    static countChar(str, char) {  
        let count = 0;  
        for (let i = 0; i < str.length; i++) {  
            if (str.charAt(i) === char) {
```

console:

```
String operations  
true  
false  
2
```

```
        count++;
    }
}
return count;
}
}

console.log(StringOps.description);
console.log(StringOps.isLowerCase("abc"));
console.log(StringOps.isLowerCase("zyBooks"));
console.log(StringOps.countChar("zyBooks", "o"));
```

Animation content:

The following code snippet is displayed.

```
class StringOps {
    static description = "String operations";

    static isLowerCase(str) {
        return str.toLowerCase() === str;
    }

    static countChar(str, char) {
        let count = 0;
        for (let i = 0; i < str.length; i++) {
            if (str.charAt(i) === char) {
                count++;
            }
        }
        return count;
    }
}

// Outputs: String operations
console.log(StringOps.description);

// Outputs: true
console.log(StringOps.isLowerCase("abc"));

// Outputs: false
console.log(StringOps.isLowerCase("zyBooks"));
```

```
// Outputs: 2
console.log(StringOps.countChar("zyBooks", "o"));
```

Animation captions:

1. The StringOps class has a static property named "description", which is accessed via the class name.
2. The static method isLowerCase() is called with the syntax: StringOps.isLowerCase(...).
3. StringOps.countChar("zyBooks", "o") counts the number of 'o' characters in the string "zyBooks" and returns 2.

PARTICIPATION ACTIVITY

8.4.7: StringOps class and static method concepts.

- 1) The code below outputs "Rectangle" to the console.

```
class Rectangle {
    static displayName =
    "Rectangle";
}
let rect = new Rectangle();
console.log(rect.displayName);
```

- ☐ True
- ☐ False

- 2) The following is a valid static method declaration that could be inserted into the StringOps class:

```
static getClassName() {
    return "StringOps";
}
```

- ☐ True
- ☐ False

3) A class can have both static and non-static methods.

- ☐ True
- ☐ False

4) Non-static methods can also be called with the

`ClassName.methodName(arguments)` syntax.

- ☐ True
- ☐ False

Inheritance

The ***extends keyword*** allows one class to inherit from another. In the inheriting class' constructor, calling the ***super()*** function calls the parent class' constructor. ***super ()*** must be called before using the ***this*** keyword in the inheriting class' constructor.

Figure 8.4.2: Person class and inheriting Student class.

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    sayHello() {
        console.log("Hello. My name is " + this.name + ".");
    }
}

// Student inherits from Person
class Student extends Person {
    constructor(name, age, gpa) {
        super(name, age); // Call parent constructor
        this.gpa = gpa;
    }

    // Override parent's sayHello method
    sayHello() {
        console.log("Hi, I'm " + this.name + " with a " + this.gpa + "
GPA!");
    }
}
```

**PARTICIPATION
ACTIVITY**

8.4.8: Inheritance.



Suppose a `GraduateStudent` class, that inherits from `Student` is being defined. The class adds a `status` field, which is set to either "Masters" or "PhD". Match the blank lines to the appropriate code pieces.

```
class GraduateStudent __ (1) __ {
    constructor(name, age, gpa, mastersOrPhd) {
        __ (2) __ (name, age, gpa);
        __ (3) __ = mastersOrPhd;
    }

    // Override parent's sayHello method
    sayHello() {
        __ (4) __ ("Hi, I'm " + this.name + ", a " + this.status +
        " student with a " + this.gpa + " GPA!");
    }
}
```

If unable to drag and drop, refresh the page.

3

1

4

2

`this.status``console.log``extends Student``super`

Reset

Exploring further:

- [Classes \(MDN\)](#).

8.5 Inner functions, outer functions, and function scope

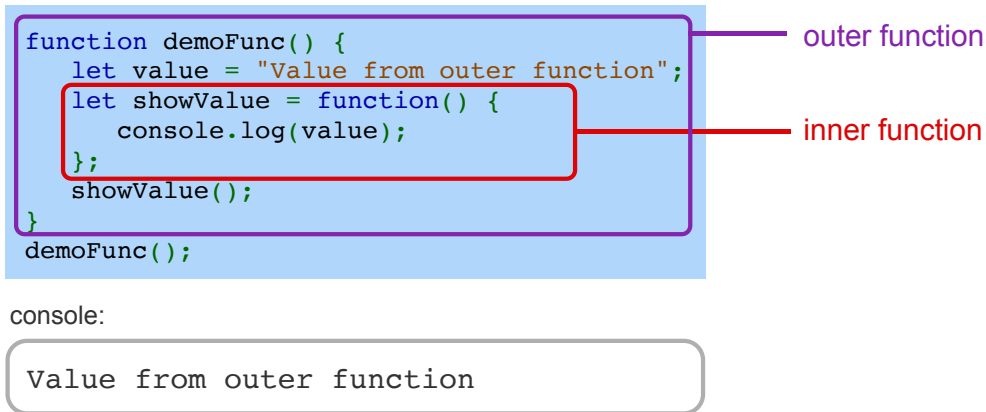
Inner and outer functions

An **inner function** (**nested function**) is a function declared inside another function. An **outer function** is a function containing an inner function. An inner function can access variables declared in the outer function.

PARTICIPATION
ACTIVITY

8.5.1: Inner and outer functions.





Animation content:

The following code snippet is displayed.

```
function demoFunc() {  
  let value = "Value from outer function";  
  let showValue = function() {  
    console.log(value);  
  };  
  showValue();  
}  
demoFunc();
```

Step 2: The console displays: "Value from outer function"

Animation captions:

1. showValue() is an inner function declared within the demoFunc() function. demoFunc() is the outer function.
2. Despite being declared outside of the showValue() function, value can still be accessed from within showValue().



```
function logzyBooks() {  
  console.log("zyBooks");  
}  
  
function logSum(x, y, z) {  
  const sum = x + y + z;  
  let actualLogger = function() {  
    console.log(sum);  
  }  
  actualLogger();  
}
```

If unable to drag and drop, refresh the page.

logSum()

logzyBooks()

actualLogger()

Inner function

Outer function

Neither an inner nor outer function

Reset

A function declaration or function expression can be used to declare an inner function

The examples above declare an inner function using a function expression. Inner functions can also be declared using function declarations, as shown in the example below.

Ex: Array filtering

Inner functions are commonly used for array filtering. An Array object's **filter()** method takes a filter function as an argument, calls the filter function for each array element, and returns a new array consisting only of elements for which the filter function returns true.

**PARTICIPATION
ACTIVITY**

8.5.3: Filtering an array of grades to get only passing grades.



```
function getPassingGrades(grades) {  
  function isPassing(number) {  
    return number >= 73;  
  }  
  
  return grades.filter(isPassing);  
}  
  
const grades = getPassingGrades([73.1, 86.4, 62.1, 59.6, 88.8, 99.9]);  
console.log("Passing grades: " + grades);
```

Console:

Passing grades: 73.1,86.4,88.8,99.9

Animation content:

The following code snippet is displayed:

```
function getPassingGrades(grades) {  
  function isPassing(number) {  
    return number >= 73;  
  }  
  
  return grades.filter(isPassing);  
}  
  
const grades = getPassingGrades([73.1, 86.4, 62.1, 59.6, 88.8, 99.9]);  
console.log("Passing grades: " + grades);
```

Step 3: The output to console is:

Passing grades: 73.1, 86.4, 88.8, 99.9

Animation captions:

1. Outer function `getPassingGrades()` declares `isPassing()` as an inner function. `isPassing()` returns true only if the number passed as an argument is `>= 73`.
2. `getPassingGrades()` is called with an array of grades. The array's `filter()` method is called

with the inner function passed as an argument.

3. The filter function is called for each element. The returned array consists only of the grades ≥ 73 .

**PARTICIPATION
ACTIVITY**

8.5.4: Array filtering using inner functions.



Consider the following code.

```
const strings = ["one", "two words", "three", "four five"];

function getSingleWords(stringArray) {
  const noSpace = function(element) {
    return element.indexOf(" ") === -1;
  };

  return stringArray.filter(noSpace);
}

function getStartingWith(stringArray, startString) {
  function startsWith(string) {
    return string.indexOf(startString) === 0;
  }

  return stringArray.filter(startsWith);
}
```

- 1) What does `getSingleWords(strings);` return?
 - ☐ ["one", "two words", "three", "four five"]
 - ☐ ["one", "three"]
 - ☐ "one"
- 2) Which inner function uses a variable from the outer function?
 - ☐ `noSpace()`
 - ☐ `startsWith()`
 - ☐ neither



- 3) What does `getStartingWith(strings, "t");` return?
- ☐ `["two words", "three"]`
 - ☐ `["three"]`
 - ☐ `[]`

Scope objects

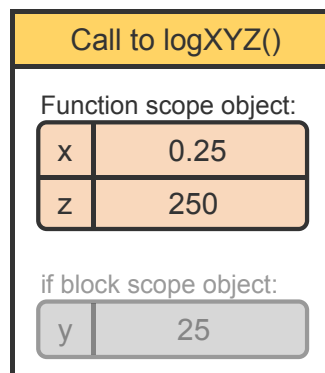
To store a collection of variables for a particular scope, JavaScript implementations commonly use a **scope object**. An object that stores a collection of variable names and corresponding values. Ex: Whenever a function with local variables is executed, the JavaScript runtime creates a scope object that stores the function's local variables.

Scope objects are behind-the-scenes objects used to implement the JavaScript runtime and are not accessible in JavaScript code.

PARTICIPATION ACTIVITY

8.5.5: Scope objects.

```
function logXYZ() {  
  let x = Math.random();  
  if (x < 0.5) {  
    let y = x * 100;  
    console.log(y);  
  }  
  
  let z = x * 1000;  
  console.log(z);  
}
```



console:

25
250

Animation content:

The following code snippet is displayed.

```
function logXYZ() {  
  let x = Math.random();  
  if (x < 0.5) {
```

```
let y = x * 100;  
console.log(y);  
}
```

```
let z = x * 1000;  
console.log(z);  
}
```

Step 3: Y is assigned with x multiplied by 100.

Step 5: Z is assigned with x multiplied by 1000.

Step 6: The output displays the value of x as 25 and the value of z as 250.

Animation captions:

1. When `logXYZ()` is called, a scope object for the function is created. Variables `x` and `z` are scoped to the entire function and are stored in the scope object.
2. The scope object includes the variable values. `x` is assigned with the random number 0.25 on the first line, and `z` is initially undefined.
3. A new block scope object is created when execution enters the `if` block. Variable `y` is declared with `let` and is scoped to the `if` block.
4. The block scope object is used by the runtime to lookup `y`'s value for the `console.log()` call.
5. When execution leaves the `if` block, the block scope object with `y`'s value is removed. The variable `y` is out of scope and no longer available.
6. The function scope object is used to store and lookup values for `x` and `z`.

Avoid mixing 'var' and 'let' in practice

*Examples in this material, like the one above, may mix the use of **var** and **let** to illustrate technical concepts. While **var** and **let** can be used together, good practice is to avoid mixed usage and instead use only one of the two.*



1) Function `logXYZ()` always creates



- ☐ a single scope object to hold all of the function's variables
- ☐ two scope objects: one for `x` and `z` and the other for `y`
- ☐ one scope object for variables `x` and `z`, but also creates a second scope object for `y` if execution enters the if block

2) Calling function `logXYZ()` six times means that at least six distinct scope objects are created by the JavaScript runtime.



- ☐ True
- ☐ False

3) Consider the altered version of function `logXYZ()` below, which logs `x`'s value in the if block.



```
function logXYZ() {  
  let x = Math.random();  
  if (x < 0.5) {  
    let y = x * 100;  
    console.log(x);  
    console.log(y);  
  }  
  
  let z = x * 1000;  
  console.log(z);  
}
```

For the statement

`console.log(x);`, how many scope objects will be checked to find the value for `x`?

- ☐ 1
- ☐ 2

Scope chain

A **scope chain** is a linked list of scope objects used by the JavaScript runtime to store and lookup variable values when executing code. When a variable is needed, a search begins at the scope object at the beginning of the scope chain. If the variable is found, the corresponding value is used. Otherwise, the next object in the scope chain is searched. If the search reaches a null object at the end of the scope chain, the variable is not found and a `ReferenceError` is thrown.

The scope chain always contains the global scope object. Additional scope objects are prepended to the list as code executes. Ex: Calling a function prepends a new scope object for that function's local variables. A block scope object is prepended when execution enters a nested block.

The animation below illustrates how the scope chain works by using the same variable name in the function's block scope and the nested scope.

PARTICIPATION ACTIVITY

8.5.7: Scope chain.



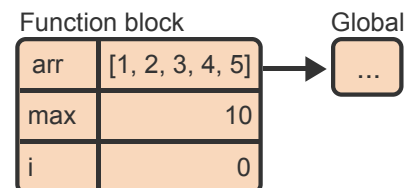
```
function shuffle(arr) {  
  const max = arr.length * 2;  
  let i = 0;  
  while (i < max) {  
    // max redeclared in new block  
    const max = arr.length;  
  
    // Create 2 random indices  
    const i1 = Math.floor(Math.random() * max);  
    const i2 = Math.floor(Math.random() * max);  
  
    // Swap two array elements  
    const temp = arr[i1];  
    arr[i1] = arr[i2];  
    arr[i2] = temp;  
  
    ++i;  
  }  
  return arr;  
}  
shuffle([1, 2, 3, 4, 5]);
```

Execution context scope chain...

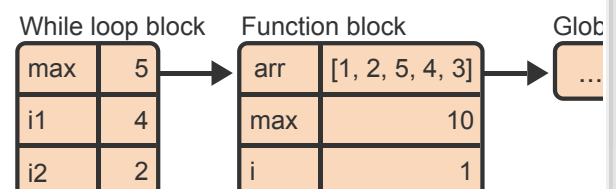
... just before the call to shuffle:



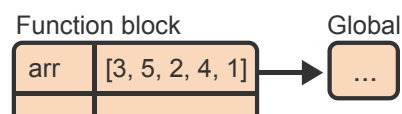
... just before entering the loop for the first time



... at the end of the first loop iteration:



... just before returning:



max	10
i	5

Animation content:

The following code snippet is displayed.

```
function shuffle(arr) {  
  const max = arr.length * 2;  
  let i = 0;  
  while (i < max) {  
    // max redeclared in new block  
    const max = arr.length;  
  
    // Create 2 random indices  
    const i1 = Math.floor(Math.random() * max);  
    const i2 = Math.floor(Math.random() * max);  
  
    // Swap two array elements  
    const temp = arr[i1];  
    arr[i1] = arr[i2];  
    arr[i2] = temp;  
  
    ++i;  
  }  
  return arr;  
}  
shuffle([1, 2, 3, 4, 5]);
```

Step 1: The array [1,2,3,4,5] is passed into the function shuffle().

Step 2: Max is set to 10. i is set to 0.

Step 6: i1 is set to 4. i2 is set to 2.

Step 8: The loop iterates 5 times. The final array contains [3,5,2,4,1].

Animation captions:

1. Just before calling shuffle(), code is executing in the global scope. The scope chain has one

- object containing global variables.
2. Entering the function's body prepends a scope object to the chain. Function-block-scoped variables `max` and `i` are included, along with parameter `arr`.
 3. Entering the while loop's block for an iteration prepends a new block scope object.
 4. `const` gives a variable block scope, so the redeclaration puts `max` into the block scope object with a value of 5.
 5. When retrieving the value of `max`, the scope chain is searched. Since `max` is found in the first object, no additional objects are searched.
 6. `i1` and `i2` are also in the object at the front of the scope chain.
 7. When using the `arr` and `i` variables, the scope chain is searched from the beginning. Both variables are scoped to the function block, and are found in the second scope chain object.
 8. The scope object for the last loop iteration is removed after exiting the loop. The scope chain again has two objects just before returning.

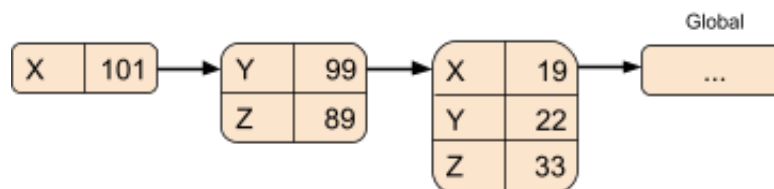
Scope objects reference the next/outer scope object

Arrows in the animation above illustrate the links between scope objects. Each scope object has a link to the next scope object in the list. The link is commonly referred to as the "outer" link.

PARTICIPATION ACTIVITY

8.5.8: Scope chain.

Suppose the following scope chain is being used when executing code.



1) The statement `console.log(X)` will ____.

- ☐ log 101 to the console
- ☐ log 19 to the console
- ☐ throw a ReferenceError

2) The statement `Y = 42;` will ____.

- ☐ add `Y` to the object at the front of the scope chain, with a value of 42
- ☐ replace `Y`'s value of 99 with 42 in the second scope object
- ☐ replace `Y`'s value of 22 with 42 in the third scope object

3) The scope chain with more than 2 scope objects implies that at least 1 function call was made.

- ☐ True
- ☐ False

Exploring further:

- [Nested functions and closures \(MDN\)](#)

8.6 Closures

Execution context and closures

An **execution context** is an object that stores information needed to execute JavaScript code, and includes, but is not limited to:

- information about code execution state, such as the line of code being executed and the line to return to when a function completes, and
- a reference to a scope chain.

Execution contexts are stored on an execution stack. The **current execution context** (**running execution context**) is the execution context at the top of the execution stack. The **current scope chain** is the scope chain of the current execution context.

A **closure** is a combination of a function's code and a reference to a scope chain. When a JavaScript function is declared, a closure is created that includes the function's code and a reference to the current scope chain. Closures are what allow an inner function to access the outer function's variables.

PARTICIPATION ACTIVITY

8.6.1: A closure allows an inner function to access the outer function's scope.



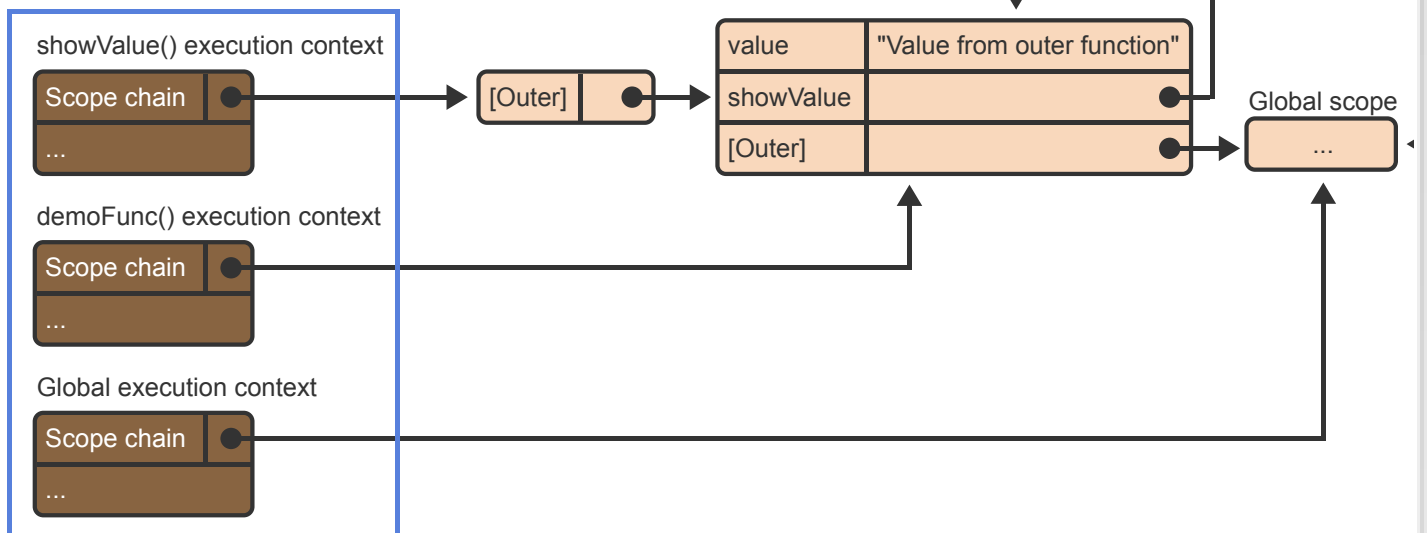
```
function demoFunc() {  
  let value = "Value from outer function";  
  let showValue = function() {  
    console.log(value);  
  };  
  showValue();  
}  
demoFunc();
```

- Scope object
- Function closure object
- Execution context object

Console:

Value from outer function

Execution stack



Animation content:

The following code snippet is displayed.

```
function demoFunc() {  
  let value = "Value from outer function";  
  let showValue = function() {  
    console.log(value);  
  };  
  showValue();  
}  
demoFunc()
```

Step 7: The console outputs: "Value from outer function".

Animation captions:

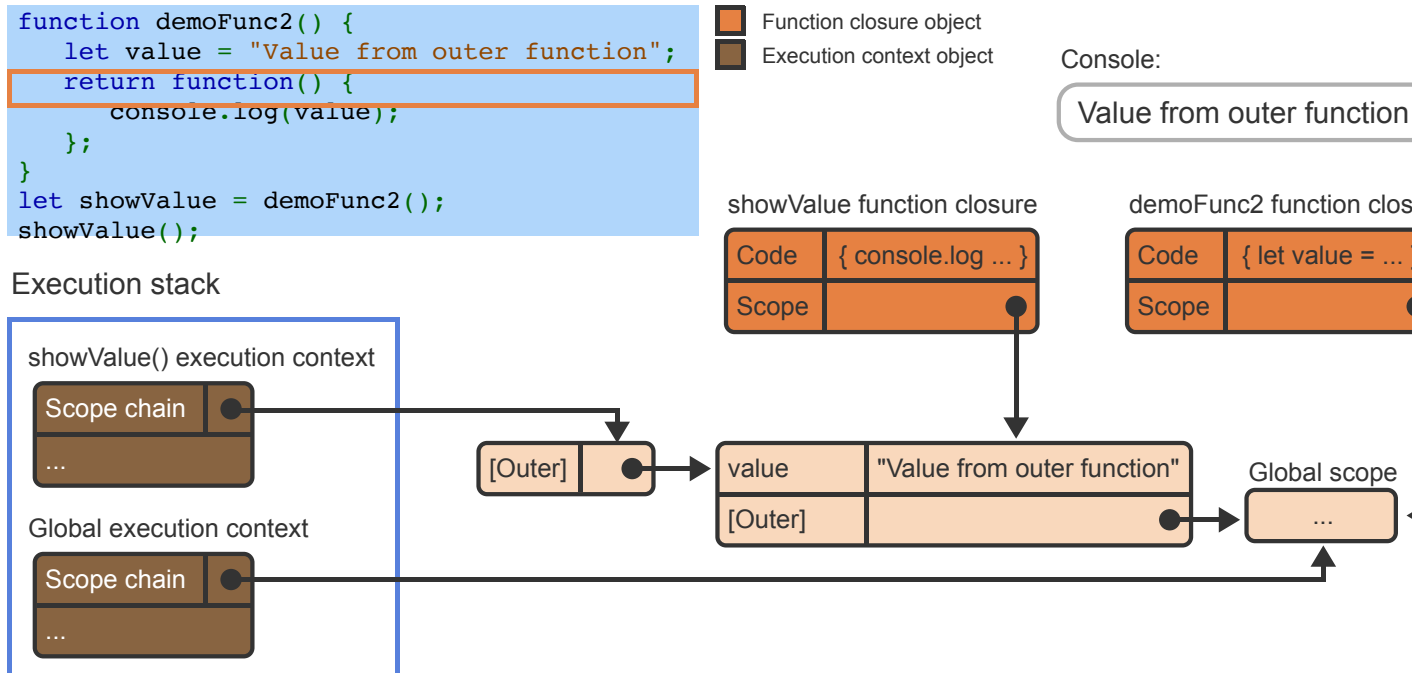
1. Initially, code is executing in the global context. So the execution stack has 1 execution context object. The referenced scope chain consists of only the global scope object.
2. When the `demoFunc()` function is declared, a closure is created. The closure's scope equals the execution context's scope chain, which consists of only the global scope object.
3. Calling `demoFunc()` does two things: First, a new execution context is pushed onto the execution stack, with the scope chain set to the `demoFunc()` function closure.
4. Second, a new scope object for `demoFunc`'s variables is prepended to the current scope chain. The value string is set inside the scope object.
5. Declaring `showValue` as an inner function creates a closure that references the scope chain of the current execution context.
6. Calling `showValue()` pushes a new execution context onto the stack, and prepends to the current scope chain. `showValue()` has no local variables, so the front of the scope chain has only an outer reference.
7. `showValue()` finds the value variable in the scope chain's second object.

PARTICIPATION ACTIVITY

8.6.2: Closures can access local variables from functions that have completed execution.



 Scope object



Animation content:

The following code is displayed.

```
function demoFunc2() {  
  let value = "Value from outer function";  
  return function() {  
    console.log(value);  
  };  
}  
let showValue = demoFunc2();  
showValue();
```

Step 7: The console outputs: "Value from outer function".

Animation captions:

1. A closure is created for the demoFunc2() function, referencing the execution context's scope chain, which consists of only the global scope object.
2. demoFunc2() is called, creating a new execution context and prepending a scope object to that context's scope chain.
3. An anonymous function is created inside demoFunc2(), referencing the current scope chain.

4. The closure is returned and assigned to the global showValue variable. The current execution context is now the global execution context.
5. demoFunc2() has completed, and the associated execution context has been popped off the stack. But the scope object from demoFunc2() stays in memory, due to being referenced by the showValue closure.
6. Calling showValue() first pushes a new execution context with a scope chain equal to that of the showValue() closure. Then a new scope object is prepended.
7. The value string can be found in the current scope chain and logged to the console.

**PARTICIPATION
ACTIVITY**

8.6.3: Changing a local variable after a closure's creation can affect functionality.



```
function demoFunc3() {  
  let value = "Initial string value";  
  const localFunc = function() {  
    console.log(value);  
  };  
  localFunc();  
  value = "Modified string value";  
  return localFunc;  
}  
let showValue = demoFunc3();  
showValue();
```

- Scope object
- Function closure object
- Execution context object

Console:

Initial string value
Modified string value

showValue function closure

Code	{ console.log ... }
Scope	●

demoFunc3 function closure

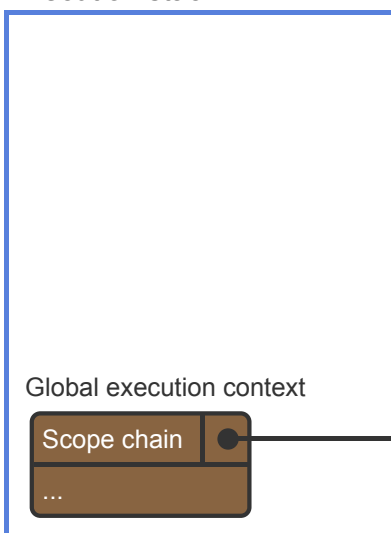
Code	{ let value = ... }
Scope	●

value	"Modified string value"
localFunc	●
[Outer]	●

Global scope

...

Execution stack



Animation content:

The following code snippet is displayed.

```
function demoFunc3() {  
  let value = "Initial string value";  
  const localFunc = function() {  
    console.log(value);  
  };  
  localFunc();  
  value = "Modified string value";  
  return localFunc;  
}  
let showValue = demoFunc3();  
showValue();
```

Step 2: The console outputs: Initial string value.

Step 3: The string value now contains: "Modified string value".

Step 5: The console outputs: Modified string value.

Animation captions:

1. A closure is created for demoFunc3(). demoFunc3() is then called, prepending a scope object to the current scope chain and setting the string's initial value.
2. localFunc is created with a reference to the current scope chain. Calling localFunc() immediately after logs the initial string value.
3. After calling localFunc(), demoFunc3() resumes execution and changes the string's value. The function closure object itself hasn't changed, but the referenced scope object has.
4. localFunc is returned and assigned to showValue.
5. Calling showValue() logs the modified string value.

PARTICIPATION ACTIVITY

8.6.4: Execution context and closures.



Assume the following code is executed before each question:

```
let inc = null;
let dec = null;
let log;
function createIncDecLog() {
  let number = 0;
  inc = function() { number++; };
  dec = function() { number--; };
  log = function() { console.log(number); };
}
createIncDecLog();
```

1) What does the following code log?



```
inc();
inc();
log();
```

- ☐ 0
- ☐ 2

2) What does the following code log?



```
inc();
dec();
dec();
inc();
inc();
log();
```

- ☐ -2
- ☐ 1
- ☐ 3

3) What does the following code log?



```
inc();
inc();
inc();
createIncDecLog();
log();
```

- ☐ 0
- ☐ 3

Closures and loops

Each time a loop iteration begins, a new block scope object is prepended to the current scope

chain. Block-scoped variables declared with **let** or **const** are stored in the block scope object. When a loop iteration ends, the block scope object is removed from the front of the current scope chain. Therefore, a loop that executes N iterations will have caused N distinct block scope objects to have been prepended and then removed from the current scope chain.

PARTICIPATION ACTIVITY

8.6.5: A new block scope is created each loop iteration, storing block-scoped variables.

```
let funcsArr = [null,null,null];

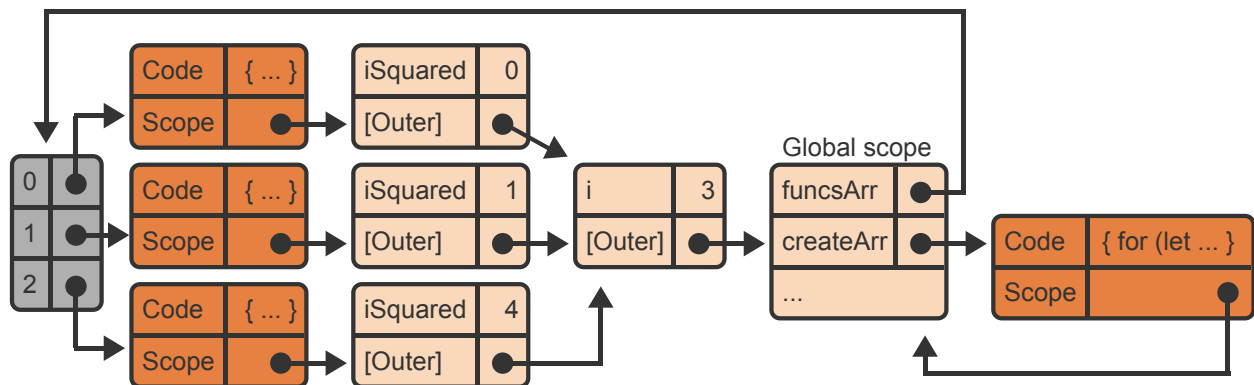
function createArr() {
  for (let i = 0; i < 3; i++) {
    const iSquared = i * i;
    funcsArr[i] = function() {
      console.log(iSquared);
    };
  }
}

createArr();
for (let func of funcsArr) {
  func();
}
```

Scope object
 Function closure object
 JavaScript array object

Console:

```
0
1
4
```



Animation content:

The following code snippet is displayed.

```
let funcsArr = [null,null,null];
```



```
function createArr() {  
  for (let i = 0; i < 3; i++) {  
    const iSquared = i * i;  
    funcsArr[i] = function() {  
      console.log(iSquared);  
    };  
  }  
}
```

```
createArr();  
for (let func of funcsArr) {  
  func();  
}
```

Animation captions:

1. funcsArr and createArr() are created in the global scope.
2. Calling createArr() prepends to the current scope chain. Entering the first loop iteration prepends again, adding a block scope object that stores iSquared.
3. The first closure is created, referencing the current scope chain.
4. Ending the first loop iteration removes the block scope from the current scope chain.
5. Starting the next iteration prepends a new block scope, and the next closure is created.
6. The third closure is created similarly.
7. The three functions are called, logging 0, 1, and 4.

PARTICIPATION ACTIVITY

8.6.6: Closures and loops.

- 1) At the start of each loop iteration, a new block scope object is prepended to the current scope chain.

- ☐ True
☐ False

- 2) At the end of each loop iteration, a block scope object is removed from the front of the current scope chain.
- ☐ True
- ☐ False
- 3) For a loop that executes N times, any variable declared inside a loop's body is stored in N distinct block scope objects.
- ☐ True
- ☐ False

Closures and loops: var declarations

Variables declared with **var** always have function scope and are never stored in a loop's block scope object. A common error is to declare a variable inside a loop with **var**, in an attempt to capture a variable's value at that point in time. However, each iteration reassigns the same variable in the function's scope object. Using **const** or **let** inside a loop can often solve this problem.




PARTICIPATION ACTIVITY

8.6.7: Variables declared with **var** have function scope, even if declared inside a loop.

```
var funcsArr = [null,null,null];

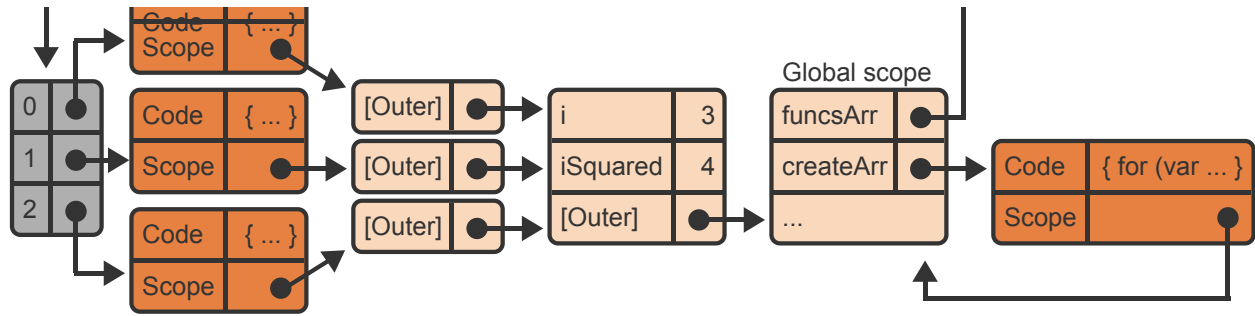
function createArr() {
  for (var i = 0; i < 3; i++) {
    var iSquared = i * i;
    funcsArr[i] = function() {
      console.log(iSquared);
    };
  }
}

createArr();
for (var func of funcsArr) {
  func();
}
```

-  Scope object
-  Function closure object
-  JavaScript array object

Console:

4
4
4



Animation content:

The following code snippet is displayed.

```
var funcsArr = [null,null,null];
```

```
function createArr() {
  for (var i = 0; i < 3; i++) {
    var iSquared = i * i;
    funcsArr[i] = function() {
      console.log(iSquared);
    };
  }
}
```

```
createArr();
for (var func of funcsArr) {
  func();
}
```

Animation captions:

1. The funcsArr array and the createArr closure are initialized the same as in the previous example. The only change is that iSquared is declared with var instead of const.
2. Calling createArr() prepends to the current scope chain. Entering the first loop iteration then prepends a block scope.
3. Only const and let create block-scoped variables. iSquared is declared with var and is therefore in the function scope object along with i.
4. After createArr() completes, 3 distinct closures reference 3 distinct, empty block scopes. Each scope references the outer scope, where iSquared is stored once with a value of 4.

5. Calling the functions logs 4 three times.

**PARTICIPATION
ACTIVITY**

8.6.8: Closures and loops.



1) What does the following code log to the console?



```
function getFunction() {  
    var functionToReturn =  
    null;  
    var i = 0;  
    while (i < 5) {  
        if (i === 0) {  
            functionToReturn =  
            function() { console.log(i);  
            };  
        }  
        i++;  
    }  
    return functionToReturn;  
}  
const theFunction =  
getFunction();  
theFunction();
```

- ☐ 0
- ☐ 4
- ☐ 5

2) What does the following code log to the console?



```
function getFunction() {  
  var functionToReturn =  
null;  
  var i = 0;  
  while (i < 5) {  
    var saved_i = i;  
    if (i === 0) {  
      functionToReturn =  
function() {  
console.log(saved_i); };  
    }  
    i++;  
  }  
  return functionToReturn;  
}  
const theFunction =  
getFunction();  
theFunction();
```

- ☐ 0
- ☐ 4
- ☐ 5

3) What does the following code log to the console?



```
function getFunction() {
  var functionToReturn =
null;
  var i = 0;
  while (i < 5) {
    const saved_i = i;
    if (i === 0) {
      functionToReturn =
function() {
console.log(saved_i); };
    }
    i++;
  }
  return functionToReturn;
}
const theFunction =
getFunction();
theFunction();
```

- ☐ 0
- ☐ 4

JavaScript runtimes may optimize scope chains

Some JavaScript debugging tools show a closure's scopes. When constructing a closure, the JavaScript runtime may optimize the scope chain by removing scope objects without any variables referenced by the closure's code. Therefore, debugging tools may show scope chains slightly different than those in this section's animations.

Exploring further:

- [Closures \(MDN\)](#)

8.7 Modules

Why modules?

A website may use a significant amount of JavaScript code to fetch data from a web server, interact with the user, perform computations, etc. Developers organize JavaScript code by placing related functionality into modules. A **module** is a JavaScript file that contains one or more variables, functions, or classes that are exported for use in other modules. Modules allow developers to write more maintainable and reusable code.

PARTICIPATION ACTIVITY

8.7.1: Using modules to organize a travel website's code.



flights.js

```
const max = 10;  
  
function purchaseTicket() { ... }  
  
...
```

Separate files

hotels.js

```
const max = 10; ← Error  
  
function bookRoom() { ... }  
  
...
```

```
<script src="flights.js"></script>  
<script src="hotels.js"></script>
```

flights.js

```
const max = 10;  
  
function purchaseTicket() { ... }  
  
...  
  
export { purchaseTicket };
```

Separate modules

hotels.js

```
const max = 10;  
  
function bookRoom() { ... }  
  
...  
  
export { bookRoom };
```

```
<script type="module">  
  import { purchaseTicket } from "../flights.js";  
  import { bookRoom } from "../hotels.js";  
  ...  
</script>
```

Animation content:

Static Figure:

The text Separate files is displayed.

Begin JavaScript code for file flights.js:

```
const max = 10;  
function purchaseTicket() { ... }
```

...

End JavaScript code.

Begin JavaScript code for file hotels.js:

```
const max = 10;  
function bookRoom() { ... }
```

...

End JavaScript code.

A red arrow points to the line of code, `const max = 10;`, in `hotels.js`.

Begin HTML code:

```
<script src="flights.js"></script>  
<script src="hotels.js"></script>
```

End HTML code.

The text Separate modules is displayed.

Begin JavaScript code for file flights.js:

```
const max = 10;  
function purchaseTicket() { ... }
```

...

```
export { purchaseTicket };
```

End JavaScript code.

Begin JavaScript code for file hotels.js:

```
const max = 10;  
function bookRoom() { ... }
```

...


```
export { bookRoom };  
End JavaScript code.
```

Begin HTML code:

```
<script type="module">  
  import { purchaseTicket } from "./flights.js";  
  import { bookRoom } from "./hotels.js";  
  ...  
</script>  
End HTML code.
```

Step 1: A travel website puts code for managing flights in flights.js and code for managing hotels in hotels.js.

The text Separate files is displayed.

Code for flights.js is displayed.

Begin JavaScript code for file flights.js:

```
const max = 10;  
function purchaseTicket() { ... }  
...  
End JavaScript code.
```

Code for hotels.js is displayed.

Begin JavaScript code for file hotels.js:

```
const max = 10;  
function bookRoom() { ... }  
...  
End JavaScript code.
```

Step 2: A webpage uses <script> tags to download the two JS files.

HTML code is displayed.

Begin HTML code:

```
<script type="module">  
  import { purchaseTicket } from "./flights.js";  
  import { bookRoom } from "./hotels.js";  
  ...  
</script>  
End HTML code.
```

Step 3: However, a browser error reports that the max variable in hotels.js conflicts with the identically named variable in flights.js.

A red arrow is displayed, pointing to the line of code, `const max = 10;` , in hotels.js.

Step 4: Converting the JS files into modules gives the developer control over which items are exported from the JS files.

The text `Separate modules` is displayed.

Code for flights.js is displayed.

Begin JavaScript code for file flights.js:

```
const max = 10;  
function purchaseTicket() { ... }
```

...

```
export { purchaseTicket };
```

End JavaScript code.

Code for hotels.js is displayed.

Begin JavaScript code for file hotels.js:

```
const max = 10;  
function bookRoom() { ... }
```

...

```
export { bookRoom };
```

End JavaScript code.

Step 5: A webpage imports functions from the modules, but no error results from using identically named variables in the two modules.

HTML code is displayed.

Begin HTML code:

```
<script type="module">  
  import { purchaseTicket } from "./flights.js";  
  import { bookRoom } from "./hotels.js";
```

...

```
</script>
```

End HTML code.

Animation captions:

1. A travel website puts code for managing flights in flights.js and code for managing hotels in

hotels.js.

2. A webpage uses `<script>` tags to download the two JS files.
3. However, a browser error reports that the `max` variable in `hotels.js` conflicts with the identically named variable in `flights.js`.
4. Converting the JS files into modules gives the developer control over which items are exported from the JS files.
5. A webpage imports functions from the modules, but no error results from using identically named variables in the two modules.

PARTICIPATION ACTIVITY

8.7.2: Modules vs. separate files.

Refer to the animation above.

- 1) An error occurs when `flights.js` and `hotels.js` are downloaded as two separate files because _____.
 - ☐ both files contain a **max** variable
 - ☐ both files contain syntax errors
 - ☐ a browser cannot download more than one JS file
- 2) No error occurs when functions from `flights.js` and `hotels.js` are imported because _____.
 - ☐ the **max** variable no longer exists in `hotels.js`
 - ☐ the **max** variables are not imported
 - ☐ modules never contain syntax errors

Module exports

A module uses an **export** statement to list items to be exported.

- If exporting more than one item, braces must surround the items. Ex:
`export { item1, item2, item3 };`. Alternatively, the **export** keyword may be used before each item's declaration. Ex: `export function item1() { ... }.`
- If exporting a single item, the **default** keyword is used in an export statement to name the one item to export.. Ex: `export default item;`. Alternatively, the **export default** keywords may be used before the item's declaration. Ex:
`export default function item() { ... }.`

The figure below declares two modules. The hello.js module exports two functions. The goodbye.js module exports a default function.

Figure 8.7.1: hello.js and goodbye.js modules.

hello.js

```
function sayHello() {  
    alert("Hello!");  
}  
  
function sayHi() {  
    alert("Hi!");  
}  
  
export { sayHello, sayHi  
};
```

goodbye.js

```
function sayGoodbye() {  
    alert("Goodbye!");  
}  
  
export default  
sayGoodbye;
```

Competing module formats

Several different module formats have been used in the past. **ES6 modules**, first introduced in 2015, is a module format supported by all major browsers. Node.js has traditionally used the CommonJS format but now supports ES6 modules. This section covers ES6 modules.



Refer to the figure above.

1) Which statement exports a constant for hello.js?

- ☐ `export const name = "Alex";`
- ☐ `const export name = "Alex";`
- ☐ `const name = "Alex" export;`

2) Can the function below be added to goodbye.js without causing an error?

```
function sayAdios() {  
  alert("Adios");  
}
```

- ☐ Yes, `sayAdios()` becomes the default export.
- ☐ Yes, but `sayAdios()` is not exported.
- ☐ No, adding `sayAdios()` causes an error.

3) Can the function below be added to goodbye.js without causing an error?

```
export default function  
sayAdios() {  
  alert("Adios");  
}
```

- ☐ Yes, `sayAdios()` becomes the default export.
- ☐ Yes, but `sayAdios()` is not exported.
- ☐ No, adding `sayAdios()` causes an error.

Module imports

A module imports items from another module using an import statement. An **import** statement indicates which items are to be imported from a module.

- If importing non-default items, braces must surround the items. Ex:
`import { item1, item2 } from "./my_module.js";`
- If importing a default item, no braces are used. Ex:
`import defaultItem from "./my_module.js";`

In a webpage, the script element must use the **type** attribute for an import statement to work: `<script type="module">`. The browser downloads a module when executing an import statement. If the browser is executing the script locally using the `file://` protocol, the browser cannot import the module. In the animation below, `index.html`, `hello.js`, and `goodbye.js` must all be located in the same directory on a web server for the code to run properly.

PARTICIPATION ACTIVITY

8.7.4: Importing functions from modules.



hello.js

```
function sayHello() {  
    alert("Hello!");  
}  
  
function sayHi() {  
    alert("Hi!");  
}  
  
export { sayHello, sayHi };
```

goodbye.js

```
function sayGoodbye() {  
    alert("Goodbye!");  
}  
  
export default sayGoodbye;
```

index.html

```
<!DOCTYPE html>  
<html lang="en">  
  <head>  
    <title>Module Demo</title>  
  </head>  
  <body>  
    <h1>Module Demo</h1>  
    <script type="module">  
      import { sayHello, sayHi } from "./hello.js";  
      import sayGoodbye from "./goodbye.js";  
  
      sayHello();  
      sayHi();  
      sayGoodbye();  
    </script>  
  </body>  
</html>
```

Animation content:

The code reads:

```
function Light(props) {  
  if (props.on) {  
    return <p>Light is on! </p>;  
  }  
  else {  
    return <p>Light is off. </p>;  
  }  
}  
  
function App() {  
  return (  
    <Light on={true} />  
  );  
}
```

The browser displays:
Light is on!

Animation captions:

1. hello.js exports sayHello() and sayHi(). goodbye.js exports sayGoodbye().
2. index.html uses a <script> tag with type="module" to indicate the enclosed code should be treated as a module.
3. The first import statement imports the sayHello() and sayHi() functions from hello.js. The "/" in front of hello.js indicates that hello.js is in the same directory as index.html.
4. The second import statement imports the default function from goodbye.js.
5. The imported functions are called, displaying alerts in the browser.

PARTICIPATION ACTIVITY

8.7.5: Importing from modules.



Refer to the animation above. Assume each question's proposed modification is independent of other questions.

1) What happens if the call to `sayHi()` is removed from `index.html`?

- ☐ An error occurs when importing `sayHi`.
- ☐ An error occurs when calling `sayHello()`.
- ☐ The code in `index.html` runs without errors.

2) What happens if `sayHi` and the preceding comma are removed from the import statement in `index.html`?

- ☐ An error occurs when importing `sayHello`.
- ☐ An error occurs when calling `sayHi()`.
- ☐ The code in `index.html` runs without errors.

3) What happens if `sayHi` is removed from the export statement in `hello.js`?

- ☐ An error occurs when importing `sayHi`.
- ☐ An error occurs when calling `sayHi()`.
- ☐ The code in `index.html` runs without errors.

4) What happens if `type="module"` is removed from the script element in `index.html`?

- ☐ An error occurs when executing the `hello.js` import statement.
- ☐ An error occurs when calling `sayHi()`.
- ☐ The code in `index.html` runs without errors.

5) What happens if the first import statement imports from `"hello.js"` instead of `"./hello.js"`?

- ☐ An error occurs when executing the `hello.js` import statement.
- ☐ An error occurs when calling `sayHi()`.
- ☐ The code in `index.html` runs without errors.

Module file extensions

Some developers use a `.mjs` file extension on modules to distinguish a module from a regular JavaScript file. However, a web server may require configuration to properly serve files with a `.mjs` file extension.

Exploring further:

- [JavaScript modules](#) from MDN

8.8 Strict mode

Applying strict mode

The flexible nature of JavaScript can lead to programming errors that are difficult to find.

PARTICIPATION ACTIVITY

8.8.1: Misspelling a variable name can be difficult to detect.

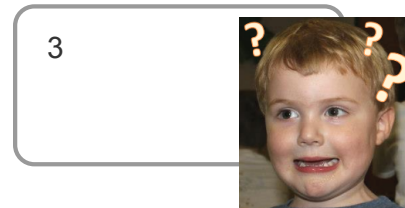


```
let humanLivesRemaining = 3;

// Set to 1 if greater than 0
if (humanLivesRemaining > 0) {
  humanLivesRemaing = 1;
}

console.log(humanLivesRemaining);
```

humanLivesRemaining	3
humanLivesRemaing	1



Animation content:

The following code snippet is displayed.

```
let humanLivesRemaining = 3;
```

```
// Set to 1 if greater than 0
if (humanLivesRemaining > 0) {
  humanLivesRemaing = 1;
}
```

```
console.log(humanLivesRemaining);
```

Animation captions:

1. Variable `humanLivesRemaining` is declared and initialized to 3.

2. A new variable `humanLivesRemaing` is created and assigned 1 unintentionally, because the programmer misspelled `"humanLivesRemaining"`.
3. The programmer may wonder "Why is `humanLivesRemaining` still 3?"

A programmer can make the JavaScript interpreter catch mistakes like mistyped variable names using strict mode. **Strict mode** makes a JavaScript interpreter apply a set of restrictive syntax rules to JavaScript code.

To enable strict mode for an entire script, the statement `"use strict"` must be placed before any other statements.

PARTICIPATION ACTIVITY

8.8.2: Strict mode causes misspelled variable assignment to throw an exception.



```
"use strict";

let humanLivesRemaining = 3;

// Set to 1 if greater than 0
if (humanLivesRemaining > 0) {
    humanLivesRemaing = 1;
}

console.log(humanLivesRemaining);
```

humanLivesRemaining

3

ReferenceError exception!

Animation content:

The following code snippet is displayed.

```
"use strict";
```

```
let humanLivesRemaining = 3;
```

```
// Set to 1 if greater than 0
if (humanLivesRemaining > 0) {
    humanLivesRemaing = 1;
}
```

```
console.log(humanLivesRemaining);
```

Step 4: The console displays:
ReferenceError exception!

Animation captions:

1. "use strict" enables strict mode.
2. Variable humanLivesRemaining is declared and initialized to 3.
3. Assignment to non-existing variable humanLivesRemaing causes the JavaScript interpreter to throw a ReferenceError.
4. Exceptions are displayed in the browser's developer console.

PARTICIPATION ACTIVITY

8.8.3: Strict errors.



Which scripts have strict errors?

1)

```
"use strict";  
x = 10;
```



- ☐ Error
- ☐ No error

2)

```
"use strict";  
let z;  
if (z > 0) {  
    console.log("true");  
}
```



- ☐ Error
- ☐ No error

3)

```
"use strict";  
let p = { x: 5, x: 10};
```



- ☐ Error
- ☐ No error

4) `"use strict";
function test(a, b, a) {
 return a - b;
}`

- ☐ Error
☐ No error

5) `"use strict";
let x = 2 + 04 + 8;`

- ☐ Error
☐ No error

6) `"use strict";
let p = {
 get x() { return 0; }
};

p.x = 1;`

- ☐ Error
☐ No error

Applying strict mode to functions

Strict mode may apply to only a function by placing `"use strict"` at the beginning of the function.

Figure 8.8.1: A strict function and a regular function.

```
function strict() {  
    "use strict";  
    // All code in this function is  
    strict  
}  
  
function notStrict() {  
    // Not in strict mode  
}
```

**PARTICIPATION
ACTIVITY**

8.8.4: Strict errors in functions.



Which scripts have strict errors?

1)

```
x = 5;
function test() {
  "use strict";
  let y = 1;
}
```



- ☐ Error
- ☐ No error

2)

```
function test(x) {
  "use strict";
  let x = 1;
}
```



- ☐ Error
- ☐ No error

3)

```
function test() {
  "use strict";
  let interface = 1;
}
```



- ☐ Error
- ☐ No error

4)

```
"use strict";
if (true) {
  function test() {
    let x = 1;
  }
}
```



- ☐ Error
- ☐ No error

5)

```
"use strict";
function zig() {
  function zag() {
    let x = 1;
  }
}
```

- ☐ Error
- ☐ No error

Exploring further:

- [Strict mode \(MDN\)](#).

8.9 Web storage

Web Storage API

The **Web Storage API** provides storage objects that allow JavaScript programs to securely store key/value pairs in the web browser. The Web Storage API supports two storage objects:

1. The **sessionStorage** object stores key/value pairs for an origin that are only available for the duration of the session. Closing the browser or browser tab ends the session.
2. The **localStorage** object stores key/value pairs for an origin that are stored indefinitely.

An **origin** is a combination of scheme, hostname, and port number in a URL. Each of the following are examples of different origins:

- <http://example.com/>
- <http://www.example.com/>
- <https://www.example.com/>
- <http://www.example.com:8080/>

The browser stores the data for each origin separately and does not share the data between origins.

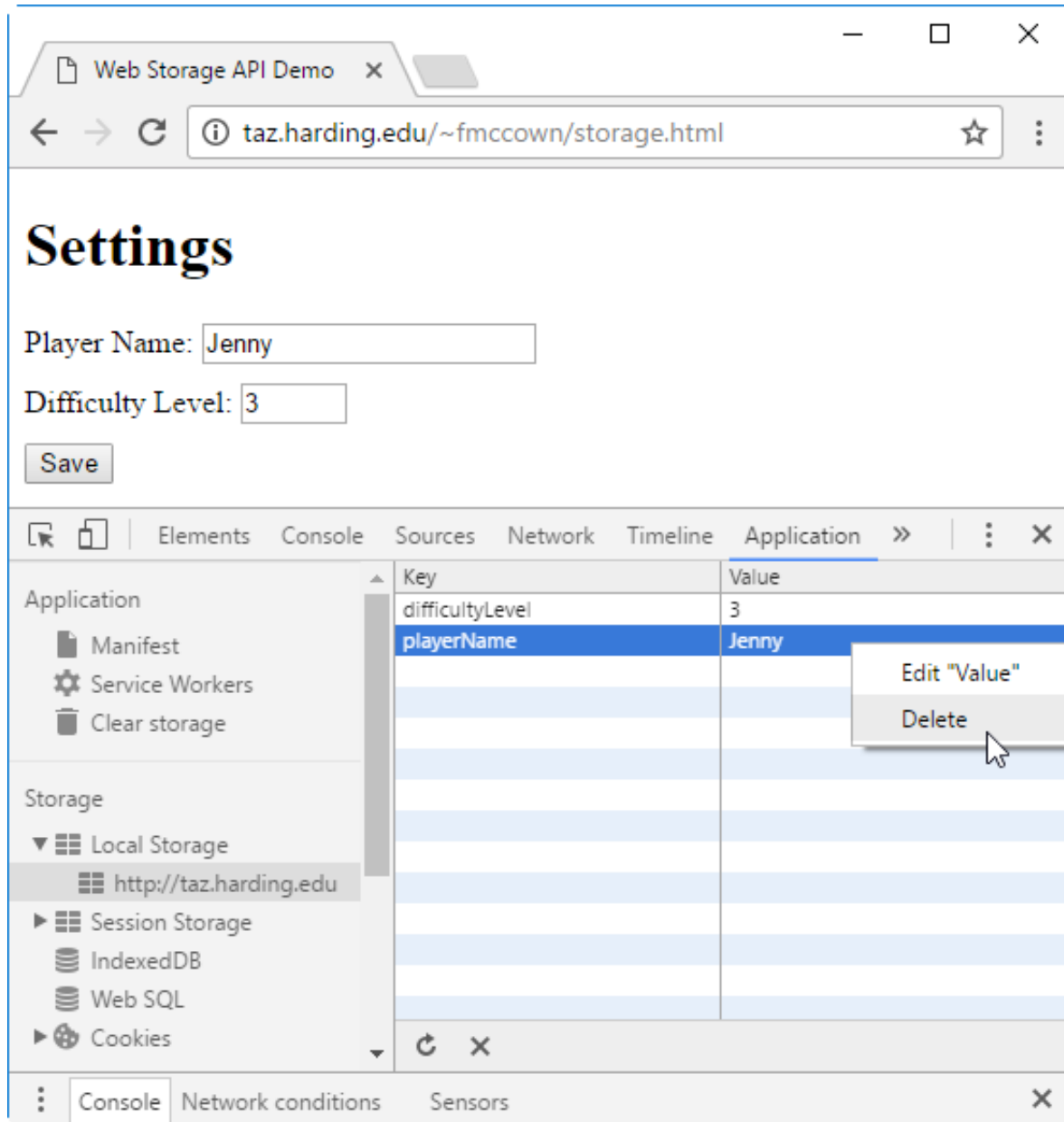
**PARTICIPATION
ACTIVITY**

8.9.1: Web storage.

- 1) Refreshing a webpage begins a new session.
☐ True
☐ False
- 2) Data that should remain after the user closes a web browser should be stored in sessionStorage.
☐ True
☐ False
- 3) The webpage from `http://google.com/` cannot access web storage data from `https://google.com/`.
☐ True
☐ False

Chrome DevTools displays web storage

Most web browsers allow developers to see the key/value pairs stored in web storage. Chrome's DevTools displays the key/value pairs stored for the origin `http://taz.harding.edu` below and allows the developer to edit and delete key/value pairs. *Good practice is to avoid storing sensitive data like social security numbers, financial data, and passwords in web storage, because the values can be easily seen by others.*



Private browsing

Many web browsers allow users to browse the web privately using "incognito" mode or "private windows". When browsing privately, web storage may be disabled or may be cleared when the user stops browsing privately. See the Exploring further section for information on detecting the availability of `localStorage`.

Accessing web storage data

The `localStorage` and `sessionStorage` objects provide methods for storing data, retrieving data, and removing data:

- **`setItem(key, value)`** stores the `key` string and associated `value` string in storage.
- **`getItem(key)`** returns the value associated with the `key` in storage or `null` if the `key` does not exist.
- **`removeItem(key)`** removes the `key` and associated value from storage.
- **`clear()`** removes all keys and associated values from storage.

PARTICIPATION ACTIVITY

8.9.2: Storing and retrieving values from localStorage.



```
<h1>Settings</h1>
<label>Player Name:
  <input type="text" id="playerName" value="Player 1">
</label>
<label>Difficulty Level:
  <input type="number" min="1" max="3" id="diffLevel" value="1">
</label>
<input type="button" value="Save" id="saveBtn">
```

Settings

Player Name:

Difficulty Level:

Save

```
let playerNameWidget = document.getElementById("playerName");
let difficultyLevelWidget = document.getElementById("diffLevel");

if (localStorage.getItem("playerName")) {
  playerNameWidget.value = localStorage.getItem("playerName");
  difficultyLevelWidget.value = localStorage.getItem("difficultyLevel");
}

document.getElementById("saveBtn").addEventListener("click", function() {
  localStorage.setItem("playerName", playerNameWidget.value);
  localStorage.setItem("difficultyLevel", difficultyLevelWidget.value);
});
```

local storage

playerName =
Jenny

difficultyLevel =
3

Animation content:

The following HTML is displayed.

<h1>Settings</h1>

Player Name:

<input type="text" id="playerName" value="Player 1">

Difficulty Level:

```
<input type="number" min="1" max="3" id="diffLevel" value="1">
```

```
<input type="button" value="Save" id="saveBtn">
```

The following JavaScript is displayed:

```
let playerNameWidget = document.getElementById("playerName");  
let difficultyLevelWidget = document.getElementById("diffLevel");
```

```
if (localStorage.getItem("playerName")) {  
    playerNameWidget.value = localStorage.getItem("playerName");  
    difficultyLevelWidget.value = localStorage.getItem("difficultyLevel");  
}
```

```
document.getElementById("saveBtn").addEventListener("click", function() {  
    localStorage.setItem("playerName", playerNameWidget.value);  
    localStorage.setItem("difficultyLevel", difficultyLevelWidget.value);  
});
```

The browser displays two input boxes labeled Player Name and Difficulty level. Below the input boxes is a Save button.

A box entitled local storage has the starting value of Empty in it.

Step 3: The user enters Jenny and 3 in the form.

Step 4: The name Jenny and difficult level 3 replace "Empty" in localStorage.

Step 5: The statement `if (localStorage.getItem("playerName"))` is true, so `playerNameWidget.value` and `difficultyLevelWidget.value` are assigned Jenny and 3.

Animation captions:

1. The form displays the default values "Player 1" and "1" in the browser.
2. `localStorage.getItem()` returns null because the playerName has not been previously saved in localStorage. The default HTML values remain in the browser.
3. The user changes the player name and difficulty level and clicks Save.
4. The Save button's click handler calls `localStorage.setItem()` to save the player name and

difficulty level to local storage.

5. When the page is reloaded, the player name and difficulty level are loaded from `localStorage` using `localStorage.getItem()`.

**PARTICIPATION
ACTIVITY**

8.9.3: Web storage methods.



Refer to the animation above.

- 1) If local storage is empty and the Settings webpage is loaded into the browser, what does `localStorage.getItem("difficultyLevel")` return?



- ☐ "1"
- ☐ `null`
- ☐ `false`

- 2) Suppose `localStorage` is empty, and the Settings webpage is loaded in the browser. If the user clicks the Save button without changing the difficulty level, what does `localStorage.getItem("difficultyLevel")` return?



- ☐ "1"
- ☐ "3"
- ☐ `null`

- 3) If the user sets the difficulty level to 2, clicks Save, closes the browser, re-opens the browser, and navigates to the Settings webpage, what difficulty level is displayed?



- ☐ 1
- ☐ 2
- ☐ `null`

4) If localStorage stores the difficulty level "3" and then `localStorage.clear()` is called, what does `localStorage.getItem("difficultyLevel")` return?

- ☐ "1"
- ☐ "3"
- ☐ null

**CHALLENGE
ACTIVITY**

8.9.1: Web storage.

550544.4142762.qx3zqy7

Start

Using sessionStorage, store a key of "age" with value of 22.

```
1  
2 /* Your solution goes here */  
3
```

1

2

3

Check

Next

Exploring further:

- [Web Storage API](#) from MDN
- [Feature-detecting localStorage](#) from MDN

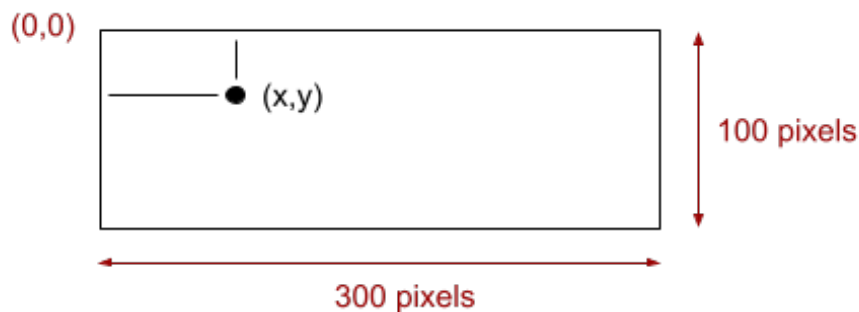
8.10 Canvas drawing

Canvas, context, and drawing rectangles

The **<canvas>** tag defines a rectangular area of a webpage where shapes, images, and text can be displayed using JavaScript. The canvas **origin** (0, 0) is located in the upper-left corner of the canvas.

Figure 8.10.1: Canvas uses a coordinate system with (0, 0) anchored in the upper-left corner.

```
<canvas id="myCanvas" width="300" height="100" style="border: 1px solid black"></canvas>
```



The canvas' **context** object represents the drawing surface of the canvas. The 2D context is used for drawing two dimensional graphics. The **context** object defines numerous methods for drawing text, lines, shapes, images, gradients, and shadows.

The context method **strokeRect()** draws an outlined rectangle, and the **fillRect()** method draws a filled rectangle. The methods' first two parameters specify the rectangle's top-left coordinate on the canvas, and the last two parameters specify the rectangle's width and height in pixels.

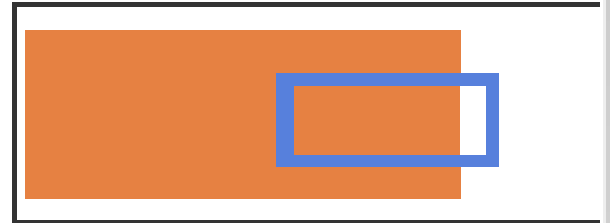
**PARTICIPATION
ACTIVITY**

8.10.1: Drawing rectangles on a canvas.



```
<canvas id="myCanvas" width="300" height="100" style="border: 2px solid black"></canvas>
```

```
let canvas = document.getElementById("myCanvas");  
let context = canvas.getContext("2d");  
  
context.fillStyle = "orange";  
context.fillRect(5, 10, 200, 75);  
  
context.strokeStyle = "royalblue";  
context.lineWidth = 6;  
context.strokeRect(120, 30, 100, 40);
```

**Animation content:**

The following code snippet is displayed.

```
<canvas id="myCanvas" width="300" height="100" style="border: 2px solid black"></canvas>
```

```
let canvas = document.getElementById("myCanvas");  
let context = canvas.getContext("2d");
```

```
context.fillStyle = "orange";  
context.fillRect(5, 10, 200, 75);
```

```
context.strokeStyle = "royalblue";  
context.lineWidth = 6;  
context.strokeRect(120, 30, 100, 40);
```

Step 1: The outline of a rectangle is displayed.

Step 4: This filled rectangle is drawn inside the previous, larger rectangle, and is filled in with an orange color.

Step 7: Creates a hollow rectangle with a blue border, overlapping the previous two rectangles.

Animation captions:

1. The canvas element defines a canvas with a width of 300 pixels and a height of 100 pixels.
2. context is assigned the 2d context from the canvas.
3. The fillStyle property sets the context's fill style (shape's interior color) to orange.
4. fillRect() draws a filled rectangle with upper-left coordinate (5, 10) that has a width of 200 pixels and a height of 75 pixels.
5. The strokeStyle property sets the context's stroke style (outline color) to royal blue.
6. The lineWidth property sets the line width to 6 pixels.
7. strokeRect() displays the outline of a rectangle with upper-left coordinate (120, 30) that has a width of 100 pixels and a height of 40 pixels.

Table 8.10.1: Rectangle methods and properties.

Method/Property	Description	Example
<code>fillRect(x, y, width, height)</code>	Draw a filled rectangle with top-left corner at (x, y)	<code>context.fillRect(0, 5, 20, 30);</code>
<code>fillStyle</code>	Interior color of filled rectangle	<code>context.fillStyle = "maroon";</code>
<code>lineWidth</code>	Outline width of stroked rectangle	<code>context.lineWidth = 5;</code>
<code>strokeRect(x, y, width, height)</code>	Draw a stroked (outlined) rectangle with top-left corner at (x, y)	<code>context.strokeRect(0, 5, 20, 30);</code>
<code>strokeStyle</code>	Outline color of stroked rectangle	<code>context.strokeStyle = "blue";</code>

**PARTICIPATION
ACTIVITY**

8.10.2: Canvas and context.

1) The canvas must be surrounded by a border.

- ☐ True
- ☐ False

2) The context's `fillStyle` determines the interior color of a rectangle displayed with the `strokeRect()` method.

- ☐ True
- ☐ False

3) What color is the rectangle?

```
context.fillStyle = "rgb(255, 0, 0)";  
context.fillRect(10, 20, 40, 60);
```

- ☐ red
- ☐ blue

4) How many pixels from the top of the canvas is the rectangle's top edge?

```
context.strokeStyle = "rgb(255, 0, 0)";  
context.strokeRect(10, 20, 40, 60);
```

- ☐ 10
- ☐ 20

5) What is the rectangle's width?

```
context.strokeStyle = "rgb(255, 0, 0)";  
context.strokeRect(10, 20, 40, 60);
```

- ☐ 40
- ☐ 60

Drawing lines and shapes with paths

Lines and shapes can be drawn on a canvas using a path. A **path** is a list of points that are connected by lines. A path is created using various context methods:

1. The **beginPath()** method creates a new path.
2. The **moveTo()** method defines the path's starting point at coordinate (x, y).
3. The **lineTo()** method adds a line to the path from the last point to a new point at coordinate (x, y).
4. Additional calls to **moveTo()** and/or **lineTo()** create additional points or lines.
5. The **closePath()** method adds a line from the current point to the path's starting point.

The path does not appear on the canvas until **stroke()** or **fill()** is called. The **stroke()** method draws a path's outline. The **fill()** method fills a path's interior.

PARTICIPATION ACTIVITY

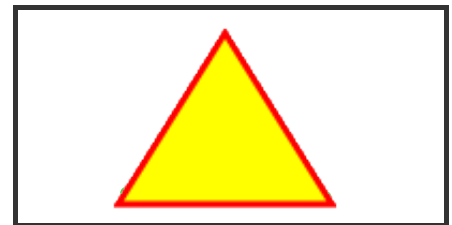
8.10.3: Using a path to draw a triangle.



```
<canvas id="myCanvas" width="200" height="100" style="border: 2px solid black">
```

```
let canvas = document.getElementById("myCanvas");
let context = canvas.getContext("2d");

context.beginPath();
context.moveTo(50, 90);
context.lineTo(100, 10);
context.lineTo(150, 90);
context.closePath();
context.fillStyle = "yellow";
context.fill();
context.strokeStyle = "red";
context.lineWidth = 3;
context.stroke();
```



Animation content:

The following code snippet is displayed.

```
<canvas id="myCanvas" width="200" height="100" style="border: 2px solid black">
```

```
let canvas = document.getElementById("myCanvas");
let context = canvas.getContext("2d");
```

```
context.beginPath();
context.moveTo(50, 90);
```

```
context.lineTo(100, 10);
context.lineTo(150, 90);
context.closePath();
context.fillStyle = "yellow";
context.fill();
context.strokeStyle = "red";
context.lineWidth = 3;
context.stroke();
```

Step 2: A green dot is displayed at the coordinate (50,90).

Step 3: A blue line is drawn between the two coordinates.

Step 4: A blue line is drawn between the two coordinates.

Step 5: A blue line is drawn between the last and first coordinates, creating a triangle shape.

Step 6: The triangle between all 3 blue lines is filled with yellow.

Step 7: The blue outline of the yellow triangle is now red with a line width of 3px.

Animation captions:

1. `beginPath()` creates a new path.
2. `moveTo()` starts the path at (50, 90).
3. `lineTo()` defines a line from (50, 90) to (100, 10).
4. `lineTo()` defines a line from (100, 10) to (150, 90).
5. `closePath()` closes the path with a final line that ends at the starting point (50, 90).
6. `fill()` draws the path with a yellow interior color.
7. `stroke()` outlines the path with a red 3px border.

Table 8.10.2: Path methods and properties.

Method/Property	Description	Example
<code>beginPath()</code> <code>closePath()</code>	Begin and close a path	<pre>context.beginPath(); ... context.closePath();</pre>
<code>fill()</code>	Draw a filled shape defined by the current path	<pre>context.fill();</pre>
<code>fillStyle</code>	Interior color of filled shape	<pre>context.fillStyle = "blue";</pre>
<code>lineTo(x, y)</code>	Draw a line from the last point in the path to (x, y)	<pre>context.lineTo(100, 50);</pre>
<code>moveTo(x, y)</code>	Move the path to (x, y)	<pre>context.moveTo(50, 20);</pre>
<code>stroke()</code>	Draw a stroked (outlined) shape defined by the current path	<pre>context.stroke();</pre>
<code>strokeStyle</code>	Outline color of stroked shape	<pre>context.strokeStyle = "orange";</pre>

**PARTICIPATION
ACTIVITY**

8.10.4: Drawing lines.



Refer to the code segment.

```
context.beginPath();
context.moveTo(100, 10);
context.lineTo(50, 90);
context.moveTo(150, 90);
context.lineTo(150, 10);
context.closePath();
context.fillStyle = "pink";
context.fill();
context.lineWidth = 5;
context.strokeStyle = "purple";
context.stroke();
```

1) How many lines are drawn?

- ☐ One
- ☐ Two
- ☐ Three

2) What color are the lines?

- ☐ Purple
- ☐ Pink
- ☐ Black

3) Which line is completely vertical?

- ☐ First line
- ☐ Second line
- ☐ Neither line

4) What is displayed when the second `moveTo()` call is removed from the code segment?

- ☐ A pink triangle.
- ☐ A shape with four sides.
- ☐ No change.

**PARTICIPATION
ACTIVITY**

8.10.5: Practice drawing paths and rectangles.

The webpage below shows a bar chart of adult obesity rates in Arkansas over a 12 year

period. Each bar is a filled rectangle with a shadow using the context properties `shadowBlur`, `shadowOffsetX`, `shadowOffsetY`, and `shadowColor`.

The Arkansas obesity rates are stored in the `arkansasRates` array, and the `coloradoRates` array stores Colorado's obesity rates over the same time period. Add the JavaScript to create a diamond (using a path) on top of each bar indicating Colorado's obesity rates as shown in the expected webpage. Make the diamonds' colors change just as the bars' colors do. Then, modify the shadows to point in the same direction as the expected webpage.

Source: stateofobesity.org

HTML

JavaScript

```
1 <h2>Arkansas & Colorado Obesity Rates</h2>
2 <canvas id="myCanvas" width="400" height="200" style="border: 1px solid
3
```

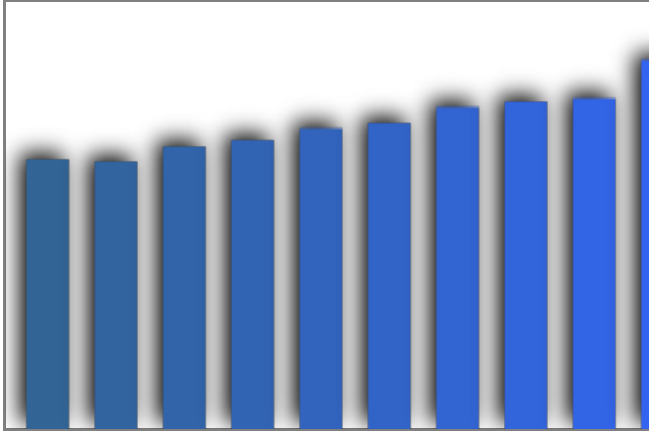
Render webpage

Reset code

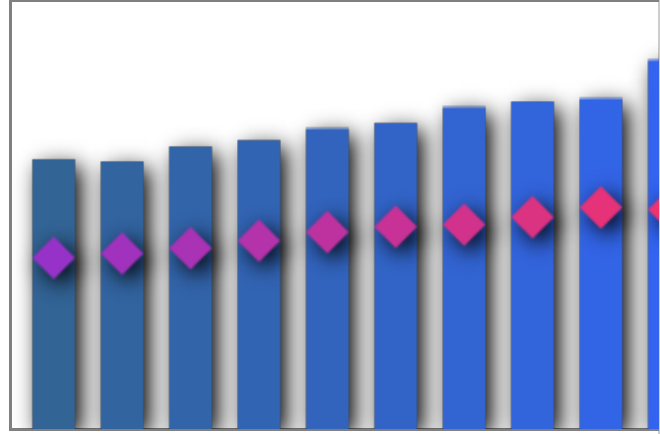
Your webpage

Expected webpage

Arkansas & Colorado Obesity Rates



Arkansas & Colorado Obesity Rates



► View solution

Drawing arcs and circles

The context method ***arc()*** adds an arc to a path when given the following parameters:

- ***x*** - Center's x coordinate
- ***y*** - Center's y coordinate
- ***radius*** - Arc's radius
- ***startAngle*** - Angle in radians where the arc begins, measured clockwise from the positive x axis
- ***endAngle*** - Angle in radians where the arc ends, measured clockwise from the positive x axis
- ***anticlockwise*** - Optional boolean parameter that causes the arc to be drawn counter-clockwise between the 2 angles when true. By default the arc is drawn clockwise.

An arc is displayed with ***stroke()*** or ***fill()***.

Figure 8.10.2: `.arc(x, y, radius, startAngle, endAngle)` draws an arc or circle.

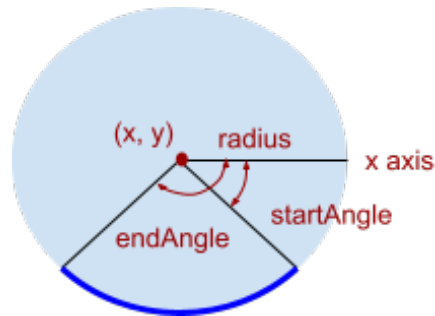
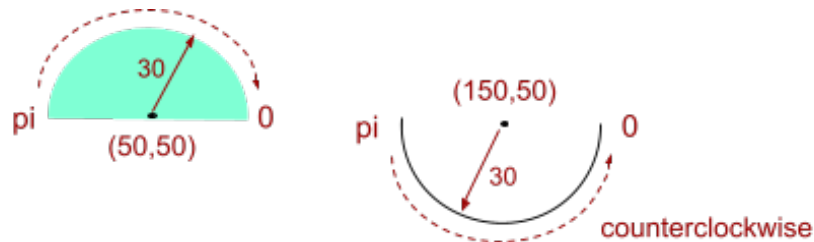


Figure 8.10.3: Two semicircles drawn with the `arc()` method.

```
// Aquamarine semicircle
context.beginPath();
context.arc(50, 50, 30, Math.PI, 0);
context.fillStyle = "aquamarine";
context.fill();

// Outline of semicircle
context.beginPath();
context.arc(150, 50, 30, Math.PI, 0, true);
context.stroke();
```



Converting degrees to radians.

Several of the context methods like `arc()` have an angle parameter that is expected to be in radians, not in degrees. Use the following equation to convert degrees to radians: $\text{radians} = \pi/180 \times \text{degrees}$. Ex: 90 degrees = $\pi/180 \times 90 = \pi/2$ radians. Or use an online [unit converter](#).

Table 8.10.3: Arc methods and properties.

Method/Property	Description	Example
<code>arc(x, y, radius, startAngle, endAngle, [anticlockwise])</code>	Adds an arc to the path	<code>context.arc(50, 100, 30, 0, Math.PI, true);</code>
<code>fill()</code>	Draw a filled arc	<code>context.fill();</code>
<code>fillStyle</code>	Interior color of arc	<code>context.fillStyle = "maroon";</code>
<code>stroke()</code>	Draw a stroked (outlined) arc	<code>context.stroke();</code>
<code>strokeStyle</code>	Outline color of stroked arc	<code>context.strokeStyle = "blue";</code>

PARTICIPATION
ACTIVITY

8.10.6: Drawing arcs and circles.



- 1) What is missing to draw the solid blue circle below?



```
context.arc(50, 50, 30,
_____, Math.PI * 2);
context.fillStyle =
"blue";
context.fill();
```



Check

Show answer

- 2) What is missing to draw the red semicircle below with a diameter of 50?



```
context.arc(50, 50, _____,
3 * Math.PI / 2, Math.PI
/ 2);
context.fillStyle =
"red";
context.fill();
```



Check

Show answer

3) What is missing to draw the brown semicircle below?



```
context.arc(50, 50, 60,  
3*Math.PI/2, Math.PI/2,  
____);  
context.strokeStyle =  
"brown";  
context.lineWidth = 3;  
context.stroke();
```



Check

Show answer

Drawing images

The context method ***drawImage()*** draws an **Image** object with the image's top-left corner anchored at the given (x, y) coordinate. Optional **width** and **height** arguments draw the image with the given dimensions.

Figure 8.10.4: Drawing an image with the drawImage() method.

```
<canvas id="myCanvas" width="400" height="200"></canvas>

<!-- Use CSS to keep the dog image from being displayed -->

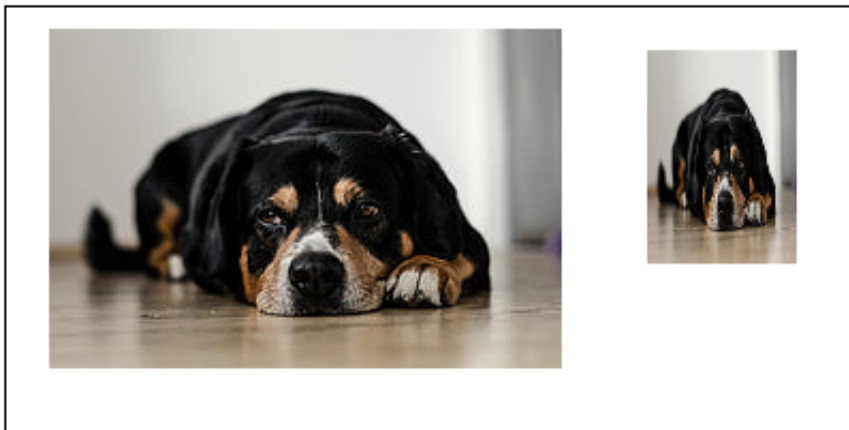
```

```
// Draw after dog image has finished loading
window.addEventListener("load", draw);

let canvas = document.getElementById("myCanvas");
let context = canvas.getContext("2d");
let image = document.getElementById("dogImg");

function draw() {
    // Draw the image 20 pixels from the left canvas edge, and 10
    // pixels from the top canvas edge. Image is 250 x 160.
    context.drawImage(image, 20, 10);

    // Draw image with width 70 and height 100
    context.drawImage(image, 300, 20, 70, 100);
}
```



**PARTICIPATION
ACTIVITY**

8.10.7: Drawing images.

Refer to the figure above.

- 1) Why must `drawImage()` be called after the load event triggers?
- ☐ Nothing can be drawn on the canvas until load triggers.
 - ☐ The load event does not need to trigger before calling `drawImage()`.
 - ☐ The image must be downloaded before `drawImage()` can display the image.
- 2) Where does the image appear on the canvas?

```
context.drawImage(image, 0, 50);
```

- ☐ Upper-left corner
 - ☐ Lower-left corner
 - ☐ Bottom-right corner
- 3) Where does the image appear on the canvas?

```
context.drawImage(image, 100, 75, 200, 50);
```

- ☐ Upper-left corner
- ☐ Centered
- ☐ Bottom-right corner

Drawing text

The context method **fillText()** draws filled text on a canvas. The **strokeText()** method draws the outline of text. These methods draw the given text beginning at the given bottom-left (x, y) coordinate. Context properties that modify the text's appearance include:

- **textAlign** - Changes the text's horizontal alignment. Possible values: "left" (default), "center", "right".
- **textBaseline** - Changes the text's baseline. Possible values: "bottom" (default), "middle", "top".
- **font** - Sets the text's font style, size, and typeface

Figure 8.10.5: Drawing text with fillText() and strokeText() methods.

```
// Red text
context.font = "italic 24pt Courier New";
context.fillStyle = "red";
context.fillText("Red text", 0, 30);

// Navy text
context.font = "bold 30pt Arial";
context.textAlign = "center";
context.strokeStyle = "navy";
context.lineWidth = 2;
context.strokeText("Navy text", canvas.width / 2,
70);

// Green text
context.font = "12pt Verdana";
context.textAlign = "right";
context.textBaseline = "top";
context.fillStyle = "green";
context.fillText("Green text", canvas.width, 80);
```



Table 8.10.4: Text methods and properties.

Method/Property	Description	Example
<code>font(fontProperties)</code>	Set the font's style, size, typeface	<code>context.font("italic 12pt Times New Roman");</code>
<code>fillText(text, x, y)</code>	Draw filled text	<code>context.fillText("filled", 10, 20);</code>
<code>fillStyle</code>	Interior color of text	<code>context.fillStyle = "gold";</code>
<code>strokeText(text, x, y)</code>	Draw stroked (outlined) text	<code>context.strokeText("stroked", 50, 50);</code>
<code>strokeStyle</code>	Outline color of stroked text	<code>context.strokeStyle = "brown";</code>
<code>textAlign</code>	Text's horizontal alignment	<code>context.textAlign = "center";</code>
<code>textBaseline</code>	Text's baseline	<code>context.textBaseline = "top";</code>

PARTICIPATION
ACTIVITY

8.10.8: Drawing text.

- 1) The `strokeText()` method draws the outline of the given text with no interior.
- ☐ True
- ☐ False

- 2) The code below draws "Hello" horizontally and vertically centered in the canvas.



```
context.font = "24pt Times New Roman";
context.textBaseline = "middle";
context.textAlign = "center";
context.fillText("Hello", 0, canvas.height / 2);
```

- ☐ True
- ☐ False

- 3) The code below draws "Goodbye" in the top-right corner of the canvas.



```
context.font = "24pt Times New Roman";
context.textBaseline = "top";
context.textAlign = "right";
context.fillText("Goodbye", canvas.width, 0);
```

- ☐ True
- ☐ False

PARTICIPATION ACTIVITY

8.10.9: Practice drawing images and text.



The webpage below displays a canvas with several playing cards. Two key functions are provided:

1. `drawCards()` - Called after all the suit images have downloaded. `drawCards()` draws a background on the entire canvas with the `createLinearGradient()` method, which produces a smooth transition from one color to another. Then `drawCards()` calls `drawCard()` several times to display various cards on the canvas.
2. `drawCard(x, y, rank, suit)` - Displays a card with the given `rank` and `suit` at coordinate `(x, y)`. The card's rank is displayed in the corner with the `fillText()` method, and the suit image is displayed with `drawImage()`.

Make the following modifications:

1. Change the gradient background color in the `drawCards ()` function to any colors you like.
2. Modify the `drawCard ()` function to display the rank with black text if the `suit` parameter is 2 (clubs) or 3 (spades). Currently the text color is always red.
3. Display a smaller suit image immediately below the rank in the upper-left corner of the card. Also draw the rank and suit in the bottom-right corner of the card.
4. Modify `drawCards ()` to display a variety of cards in any pattern you like. The expected webpage displays Ace through King using randomly selected suits.
5. Finally, display the text "Let's play cards!" somewhere on the canvas.

HTML

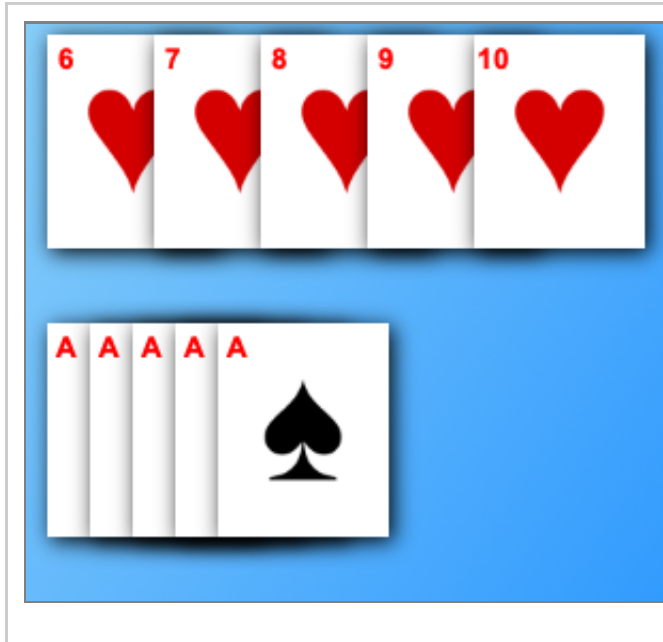
JavaScript

```
1 <canvas id="myCanvas" width="410" height="270" style="border: 1px solid
2
3 <div style="display:none;">
4   
9
```

Render webpage

Reset code

Your webpage



Expected webpage



► View solution

Exploring further:

- [Canvas element \(MDN\)](#)
- [Canvas Context2D properties and methods \(MDN\)](#)

8.11 Canvas transformations and animation

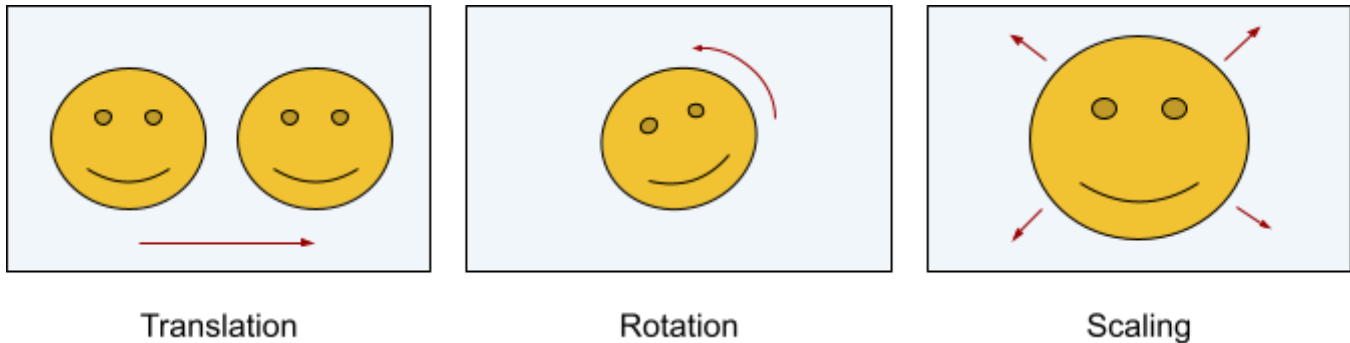
Transformations

A **transformation** is a graphical operation that alters the position, shape, or orientation of an object. The canvas' context supports 3 common transformation operations:

1. **Translation** - Moving the canvas' origin to another location to draw a graphic at a different location
2. **Rotation** - Rotating the canvas to draw a graphic at an angle

3. **Scaling** - Increasing or decreasing the canvas' grid to draw a graphic larger or smaller

Figure 8.11.1: Three transformations: Translation, rotation, and scaling.



Translation

The ***translate()*** method translates the origin of the canvas to the given (x, y) coordinate, relative to the current origin. When a graphic is drawn on a canvas that has been translated, the location of the graphic will appear in a different location than if the canvas had not been translated.

PARTICIPATION ACTIVITY

8.11.1: Translating the origin to draw 3 circles.

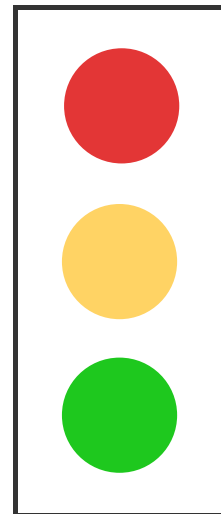
```
<canvas id="myCanvas" width="100" height="260"
style="border: 2px solid black"></canvas>
```

```
let canvas = document.getElementById("myCanvas");
let context = canvas.getContext("2d");

context.beginPath();
context.arc(50, 50, 30, 0, 2 * Math.PI);
context.fillStyle = "red";
context.fill();

context.beginPath();
context.translate(0, 80);
context.arc(50, 50, 30, 0, 2 * Math.PI);
context.fillStyle = "yellow";
context.fill();

context.beginPath();
context.translate(0, 80);
context.arc(50, 50, 30, 0, 2 * Math.PI);
```



```
context.fillStyle = "green";  
context.fill();
```

Animation content:

The following code snippet is displayed.

```
<canvas id="myCanvas" width="100" height="260"  
style="border: 2px solid black"></canvas>  
let canvas = document.getElementById("myCanvas");  
let context = canvas.getContext("2d");
```

```
context.beginPath();  
context.arc(50, 50, 30, 0, 2 * Math.PI);  
context.fillStyle = "red";  
context.fill();
```

```
context.beginPath();  
context.translate(0, 80);  
context.arc(50, 50, 30, 0, 2 * Math.PI);  
context.fillStyle = "yellow";  
context.fill();
```

```
context.beginPath();  
context.translate(0, 80);  
context.arc(50, 50, 30, 0, 2 * Math.PI);  
context.fillStyle = "green";  
context.fill();
```

Step 1: The canvas is currently an empty vertical rectangle.

Step 2: The circle is placed in the top third of the rectangle.

Step 3: The origin is now in the same x position, but lower y position. The y level is beneath the red circle now.

Step 4: A yellow circle now appears in the middle third of the rectangle.

Step 5: The origin is now in the same x position, but lower y position. The y level is beneath the

yellow circle now.

Step 6: A green circle now appears in the lower third of the rectangle.

Animation captions:

1. The origin (0, 0) by default is the upper-left corner of the canvas.
2. `fill()` draws a red circle with radius 30 at (50, 50).
3. `translate()` translates the origin 80 pixels lower than the previous origin.
4. `fill()` draws a yellow circle at (50, 50), which appears lower than the red circle because the origin is lower.
5. `translate()` translates the origin to 80 pixels below the previous origin.
6. `fill()` draws a green circle at (50, 50), which appears lower than the yellow circle because the origin is lower.

PARTICIPATION ACTIVITY

8.11.2: Translating.



Refer to the animation above.

- 1) Removing all calls to `translate()` results in a single green circle appearing on the canvas.



- ☐ True
☐ False

- 2) The final call to `context.translate(0, 80)` moved the origin to (0, 80).



- ☐ True
☐ False

3) Changing the first `translate()` call to `context.translate(-40, 80)` moves the yellow and green circles to the left 40 pixels.

- ☐ True
- ☐ False

**PARTICIPATION
ACTIVITY**

8.11.3: Practice translating.

The webpage below displays a canvas with 5 squares that are all placed on top of each other. Add a call to `translate()` inside the `for` loop to draw the squares moving down and to the right as shown in the expected webpage.

HTML

JavaScript

```
1 <canvas id="myCanvas" width="410" height="270" style="border: 1px solid  
2
```

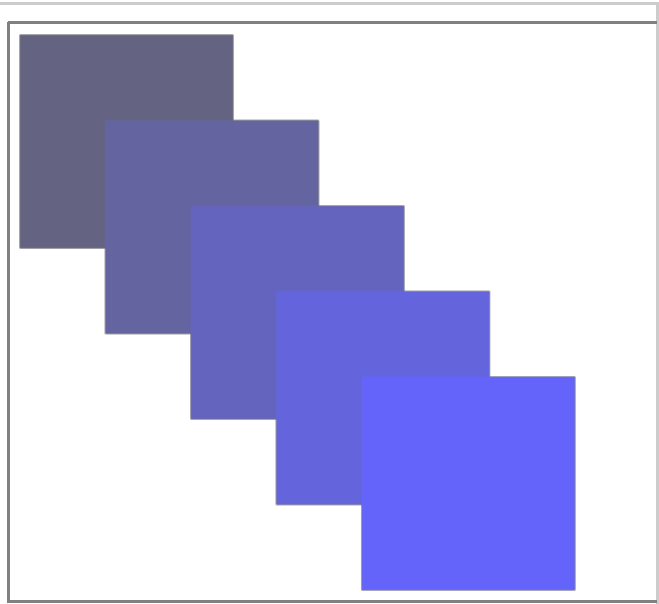
Render webpage

Reset code

Your webpage



Expected webpage



► View solution

Rotation

The context method ***rotate()*** rotates the canvas clockwise about the current origin by the given radian angle.

Figure 8.11.2: Using `rotate()` to display rotated squares.

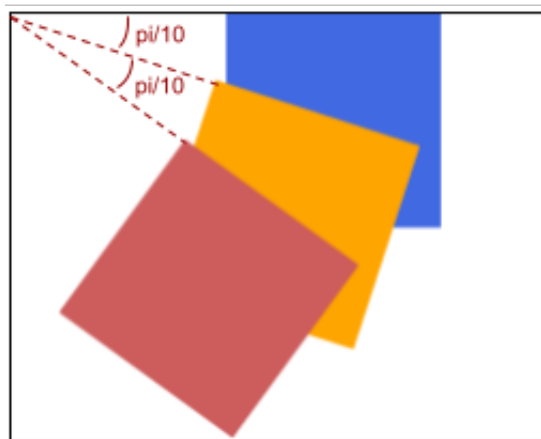
```
context.fillStyle = "royalblue";
context.fillRect(100, 0, 100,
100);

// Rotate clockwise 18 degrees
context.rotate(Math.PI / 10);

context.fillStyle = "orange";
context.fillRect(100, 0, 100,
100);

// Rotate clockwise 18 degrees
context.rotate(Math.PI / 10);

context.fillStyle = "indianred";
context.fillRect(100, 0, 100,
100);
```



The `translate()` and `rotate()` methods are used together to rotate a graphic about different points. The figure above rotated about the canvas origin at $(0, 0)$, but a square can be rotated about the square's center by moving the origin to the square's center before rotating:

1. Translate the origin to the center of the square
2. Rotate
3. Translate the origin back to the origin's original location
4. Draw the square

Figure 8.11.3: Combining translate() and rotate() to rotate squares about the center of the squares.

```
<canvas id="myCanvas" width="200" height="200" style="border: 1px solid black"></canvas>
```

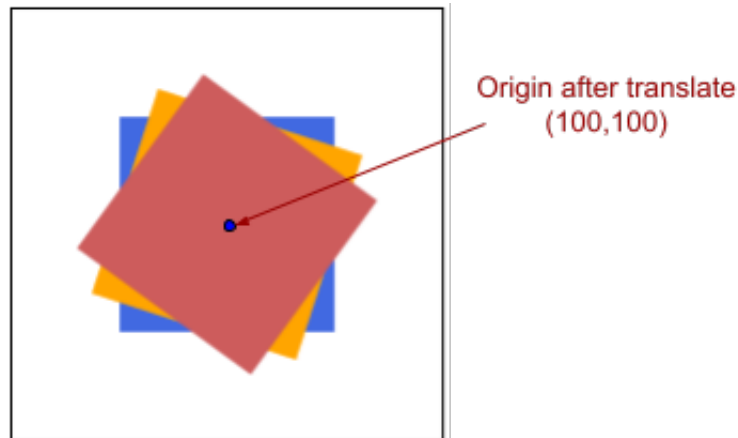
```
// Blue square is not rotated
context.fillStyle = "royalblue";
context.fillRect(50, 50, 100, 100);

// Translate to the center of the blue square, rotate, translate back
context.translate(100, 100);
context.rotate(Math.PI / 10);
context.translate(-100, -100);

// Orange square has same coordinates as blue square but is rotated
context.fillStyle = "orange";
context.fillRect(50, 50, 100, 100);

// Translate to the center of the blue square, rotate, translate back
context.translate(100, 100);
context.rotate(Math.PI / 10);
context.translate(-100, -100);

// Red square has same coordinates as blue square but is rotated
context.fillStyle = "indianred";
context.fillRect(50, 50, 100, 100);
```



- 1) What is missing to draw the pink rectangle below?



```
context.rotate(Math.PI /
_____);
context.fillStyle =
"hotpink";
context.fillRect(100, 0,
70, 30);
```

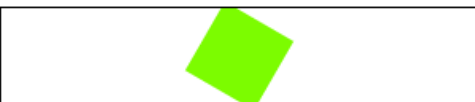
**Check**[Show answer](#)

- 2) The green square is rotated about the center of the canvas that is 300 x 60 pixels. What number is missing from the calls to `translate()`?



```
context.translate(150,
_____);
context.rotate(-Math.PI /
3);
context.translate(-150, -
_____);

context.fillStyle =
"lawngreen";
context.fillRect(125, 5,
50, 50);
```

**Check**[Show answer](#)

3) What is the missing y value to display the rotated cat as pictured in the canvas?



```
context.drawImage(img, 0,
0, 200, 100);
context.rotate(Math.PI /
2);
context.drawImage(img, 0,
_____, 100, 50);
```



Check

Show answer

Scaling

The **scale()** method uses horizontal and vertical multipliers to increase or decrease the size of graphics on a canvas. Multipliers less than 1.0 make graphics appear smaller on the canvas, and values greater than 1.0 make graphics appear larger.

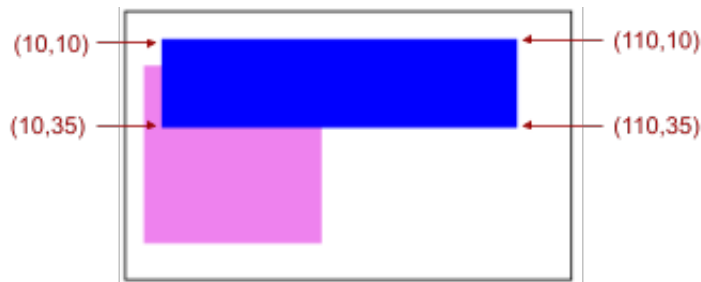
When the canvas origin is at (0, 0), the (x, y) coordinate of each shape's endpoints are multiplied by the scale's parameters; x values are multiplied by the horizontal multiplier and y values by the vertical multiplier. In the figure below, the blue and violet rectangles have the same top-left endpoint at (5, 20), but the blue rectangle is scaled by **scale(2, 0.5)**, transforming the location of the rectangle's top-left coordinate, width, and height:

- Top-left coordinate (5, 20) becomes $(5 \times 2, 20 \times 0.5) = (10, 10)$
- Width becomes $50 \times 2 = 100$
- Height becomes $50 \times 0.5 = 25$

Figure 8.11.4: Doubling the width and halving the height of a square with `scale()`.

```
<canvas id="myCanvas" width="200" height="80" style="border: 1px solid black"></canvas>
```

```
// Violet square  
context.fillStyle = "violet";  
context.fillRect(5, 20, 50, 50);  
  
// Double x values, halve y values  
context.scale(2, 0.5);  
  
// Blue square appears wider and shorter  
context.fillStyle = "blue";  
context.fillRect(5, 20, 50, 50);
```



PARTICIPATION ACTIVITY

8.11.5: Scaling.

Refer to the code segment.

```
context.fillStyle = "orange";  
context.fillRect(0, 0, 50, 50);  
context.scale(xScale, yScale);  
context.fillStyle = "blue";  
context.fillRect(0, 0, 50, 50);
```

1) Is the orange square scaled by the call to `scale()`?

- ☐ Yes
- ☐ No

2) What is the blue rectangle's width and height when `xScale = 1` and `yScale = 1`?

- ☐ width = 50, height = 50
- ☐ width = -50, height = -50
- ☐ width = 0, height = 0

3) What is the blue square's top-left coordinate when `xScale = 0.5` and `yScale = 3`?

- ☐ (0.5, 3)
- ☐ (50, 50)
- ☐ (0, 0)

4) What values make the blue rectangle's width 200 pixels and height 100 pixels?

- ☐ `xScale = 0, yScale = 0`
- ☐ `xScale = 4, yScale = 2`
- ☐ `xScale = 200, yScale = 100`

PARTICIPATION ACTIVITY

8.11.6: Practice scaling.

The webpage below displays a sun in the corner of the canvas. When the user moves a slider, the `drawSun()` function is called, which draws the sun with the new x and y scale values. The `drawSun()` function calls a few new context methods:

- **`clearRect()`** clears the canvas of any previous graphics
- **`save()`** saves the current context settings
- **`restore()`** restores the saved context settings

The `save()` and `restore()` methods are useful when a developer wants to undo transformations that have been applied to the context.

The sun is currently scaling out from the origin at (0, 0). Make the following modifications so the sun appears in the center of the canvas and scales about the sun's center:

1. Call `context.translate(130, 30)` before `scale()` so the origin is at (130, 30), and the sun appears in the center of the canvas. Click "Render webpage" and note that the sun is now initially centered but scales as before.
2. Call `context.translate(70, 70)` before `scale()` so the origin is in the center of the sun before scaling. Then, the call to `scale()` will cause the sun to scale about the center of the sun instead of the upper-left corner of the sun. Call `context.translate(-70, -70)` immediately after `scale()` to move the origin back to the upper-left corner of the sun. Click "Render webpage" and note that the sun is now scaling from the center of the sun.

HTML

JavaScript

```
1 <div>
2   <label for="xSlider" style="width:100px; display:inline-block">
3     Scale x: <b id="xScale">1</b>
4   </label>
5   <input id="xSlider" type="range" min="0" max="3" step="0.1" value=
6 </div>
7 <div>
8   <label for="ySlider" style="width:100px; display:inline-block">
9     Scale y: <b id="yScale">1</b>
10  </label>
11  <input id="ySlider" type="range" min="0" max="3" step="0.1" value=
12 </div>
13
14 <canvas id="myCanvas" width="400" height="200" style="border: 1px sol
15
```


Render webpage

Reset code

Your webpage

Scale x: 1


Scale y: 1



Expected webpage

Scale x: 1

Scale y: 1

[► View solution](#)

Animation

The canvas transformation methods can be used to create animations by repeatedly and quickly re-drawing the canvas, making small changes in the location of the graphics that should appear to move. The human eye perceives smooth animation when the screen updates at least 60 times a second, so each **animation frame** is visible for only 1/60 of a second.

The following steps summarize the animation process:

1. Clear the canvas with `clearRect()`
2. Save the canvas state with `save()`
3. Draw the animated graphics
4. Restore the canvas state with `restore()`
5. Repeat until the animation is finished

The **`window.requestAnimationFrame()`** method specifies a callback function that is called by the JavaScript engine to draw a single animation frame on the canvas. The callback may be called up to 60 times a second to produce a complete animation.




```
<canvas id="myCanvas" width="200" height="100" style="border: 2px solid black"></canvas>
```

```
let canvas = document.getElementById("myCanvas");
let context = canvas.getContext("2d");
let circleX = 50;

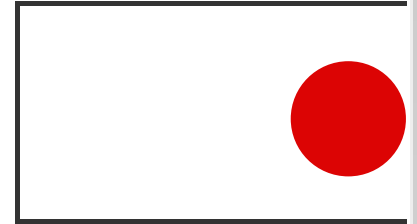
window.requestAnimationFrame(drawFrame);

function drawFrame() {
  context.clearRect(0, 0, canvas.width, canvas.height);
  context.save();

  context.beginPath();
  context.arc(circleX, 50, 30, 0, 2 * Math.PI);
  context.fillStyle = "red";
  context.fill();

  context.restore();

  circleX++;
  if (circleX < 150) {
    window.requestAnimationFrame(drawFrame);
  }
}
```



circleX = 15

Animation content:

The following code snippet is displayed.

```
<canvas id="myCanvas" width="200" height="100" style="border: 2px solid black"></canvas>
```

```
let canvas = document.getElementById("myCanvas");
let context = canvas.getContext("2d");
let circleX = 50;
```

```
window.requestAnimationFrame(drawFrame);
```

```
function drawFrame() {
  context.clearRect(0, 0, canvas.width, canvas.height);
  context.save();

  context.beginPath();
  context.arc(circleX, 50, 30, 0, 2 * Math.PI);
```

```
context.fillStyle = "red";
context.fill();

context.restore();

circleX++;
if (circleX < 150) {
    window.requestAnimationFrame(drawFrame);
}
}
```

Step 4: The circle is on the left half of the canvas, with circleX still 50.

Step 6: The circle slides to the right side of the canvas.

Animation captions:

1. circleX is assigned 50, the circle's initial x coordinate.
2. requestAnimationFrame() requests that drawFrame() be called the next time the canvas is re-drawn.
3. The JavaScript engine calls drawFrame() when the canvas is ready to be drawn.
4. After clearing the canvas and saving the canvas settings, a red circle is drawn at (circleX, 50).
5. After restoring the canvas settings, circleX is incremented to 51.
6. While circleX is < 150, drawFrame() will be called repeatedly for each canvas update. The circle moves 1 pixel to the right each time until circleX = 150. Then the animation stops.

PARTICIPATION ACTIVITY

8.11.8: Canvas animation.

1) The `requestAnimationFrame()` method must be called to start an animation.

- ☐ True
- ☐ False

2) In the animation above, removing the `if` statement and the call to `requestAnimationFrame()` causes the red circle to remain at (50, 50).

- ☐ True
- ☐ False

3) The `requestAnimationFrame()` method's callback is called 60 times a second, even when the browser tab is not visible.

- ☐ True
- ☐ False

PARTICIPATION ACTIVITY

8.11.9: Practice animating.

The webpage below shows a basketball image that bounces left and right when the user mouses over the canvas. The basketball logic is encapsulated in a `ball` object that has 2 methods:

- `move()` - Moves the x coordinate of the ball left or right and accounts for bouncing the ball off the left and right sides of the canvas
- `draw()` - Draws the ball at the ball's `x` and `y` location on the canvas

The ball's methods are called in `drawFrame()`, the callback function supplied to `requestAnimationFrame()`. When the mouse is moved off the canvas, the `cancelAnimationFrame()` method is called, which cancels the animation request.

Make the following modifications so the ball bounces off all walls, rotates, and scales larger:

1. Add code to the ball's `move()` method so the y property is incremented/decremented just like the ball's x property. Add an `if` statement to multiply `yInc` by -1 if the y property causes the ball to be drawn past the top or bottom edges of the canvas.
2. Add code to the ball's `draw()` method to rotate the ball by adding 0.01 to the ball's

`rotation` property. Then, translate to the ball's center, rotate the canvas by `rotation`, and translate back before drawing the ball. The ball's center can be calculated as:

```
centerX: this.x + this.img.width / 2;  
centerY: this.y + this.img.height / 2;
```

3. Finally, make the ball scale up in size by 0.1 when the ball hits a canvas edge. Modify the ball's `scale` property when the ball bounces. The ball should scale about the ball's center, so place the call to `scale()` near the code to rotate the ball.

HTML

JavaScript

```
1 <canvas id="myCanvas" width="410" height="270" style="border: 1px solid  
2  
3 <div style="display:none;">  
4   
  </div>
</body>
```

```
let url = "wss://echo.websocket.events/";
let websocket = new WebSocket(url);

websocket.onopen = function() {
```

Web browser

WebSocket Demo

Connected

Hello, WebSockets!

Disconnected

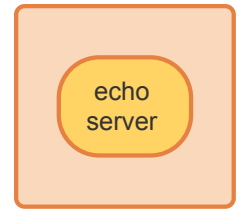
```
    displayMessage("Connected");
    websocket.send("Hello, WebSockets!");
};

websocket.onclose = function() {
    displayMessage("Disconnected");
};

websocket.onmessage = function(e) {
    displayMessage(e.data);
};

setTimeout(function() {
    websocket.close(); }, 2000);

function displayMessage(message) {
    let para = document.createElement("p");
    para.innerHTML = message;
    document.getElementById("output").appendChild(para);
}
```



websocket.events

Animation content:

The following code HTML is displayed:

```
<body>
  <h1>WebSocket Test</h1>
  <div id="output"></div>
</body>
```

The following JavaScript is displayed:

```
let url = "wss://echo.websocket.events/";
let websocket = new WebSocket(url);
```

```
websocket.onopen = function() {
    displayMessage("Connected");
    websocket.send("Hello, WebSockets!");
};
```

```
websocket.onclose = function() {
    displayMessage("Disconnected");
};
```

```
websocket.onmessage = function(e) {
    displayMessage(e.data);
};
```

```
setTimeout(function() {  
    websocket.close()}, 2000);  
  
function displayMessage(message) {  
    let para = document.createElement("p");  
    para.innerHTML = message;  
    document.getElementById("output").appendChild(para);  
}
```

Animation captions:

1. The web browser creates a WebSocket object that attempts to connect to the WebSocket server at URL `wss://echo.websocket.events/`.
2. When the connection is established, the open event is triggered, and the onopen callback function outputs "Connected" to the browser.
3. The browser calls `send()` to send the message "Hello, WebSockets!" to the echo server.
4. The echo server replies with the same message back to the browser.
5. The message event is triggered when data is received from the server. The browser displays the "Hello, WebSockets!" message from the event's data property.
6. `close()` is called after a two second delay to close the WebSocket connection.
7. Closing the connection triggers the close event, and the onclose callback function displays "Disconnected".

PARTICIPATION ACTIVITY

8.12.2: Using the WebSocket object.



Refer to the animation above.

1) What does a WebSocket URL begin with?



- ☐ `http://` or `https://`
- ☐ `ws://` or `wss://`
- ☐ Anything

2) What event must be triggered before `websocket.send()` is called?

- ☐ message
- ☐ connect
- ☐ close

3) What event is triggered by `websocket.send()`?

- ☐ message
- ☐ connect
- ☐ No event is triggered.

4) What event, which is not implemented in the code above, is triggered when the client does not successfully connect to the server?

- ☐ error
- ☐ connect
- ☐ close

WebSocket handshake

The browser and server perform a **WebSocket handshake** to establish a WebSocket connection. The browser requests a WebSocket connection by sending the server a WebSocket handshake request, an HTTP request that contains several headers like:

- **Connection** - Value set to "Upgrade"
- **Sec-WebSocket-Key** - String produced by the client and used by the server to produce a **Sec-WebSocket-Accept** string that the client expects in the response
- **Upgrade** - Request to upgrade from HTTP to WebSocket protocol

If the server accepts the WebSocket connection request, the server responds with a WebSocket handshake response that contains a 101 status code and several headers:

- **Connection** - Value set to "Upgrade"
- **Sec-WebSocket-Accept** - String produced by the server based on the

Sec-WebSocket-Key string

- Upgrade - Upgrade to WebSocket protocol

After the connection is established, the WebSocket protocol messages are sent in a binary format.

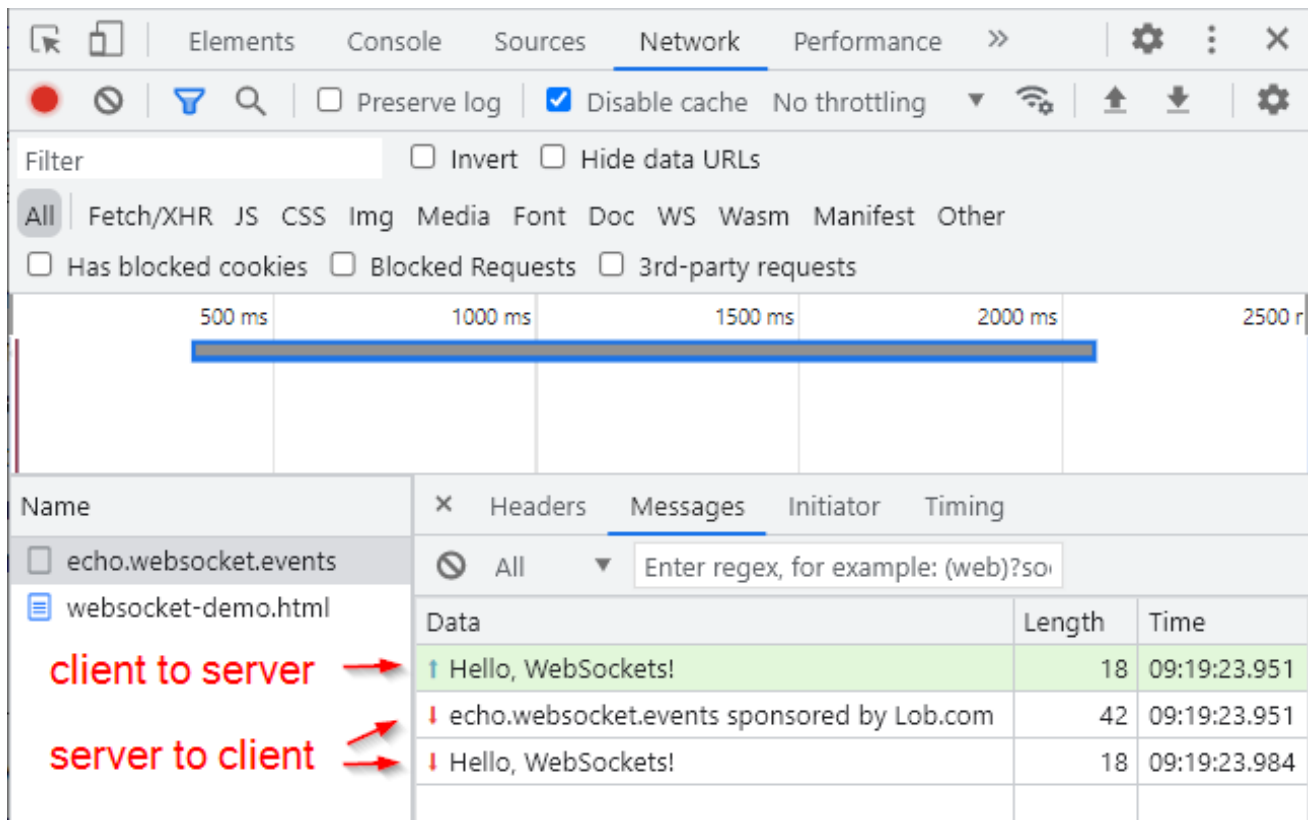
Figure 8.12.2: Viewing a WebSocket handshake in Chrome DevTools (Network tab).

The screenshot displays the Chrome DevTools Network tab for a request to `echo.websocket.events/`. The request is a GET method with a status code of 101, indicating a successful upgrade to the WebSocket protocol. The response headers show the server's acceptance of the upgrade, including the `Sec-WebSocket-Accept` key and the `Upgrade: websocket` header. The request headers show the client's upgrade request, including the `Sec-WebSocket-Key`, `Sec-WebSocket-Version: 13`, and `Upgrade: websocket` headers. The `Connection: Upgrade` header is also present in both the request and response.

Name	Headers	Messages	Initiator	Timing
echo.websocket.events	▼ General			
websocket-demo.html	Request URL: <code>wss://echo.websocket.events/</code>			
	Request Method: GET			
	Status Code: 101 Switching Protocols			
	▼ Response Headers View source			
	Connection: Upgrade			
	Sec-WebSocket-Accept: <code>z3sz8y30FqVDIcg/m0AjbpAzdeg=</code>			
	Upgrade: websocket			
	Via: 1.1 vegur			
	▼ Request Headers View source			
	Accept-Encoding: gzip, deflate, br			
	Accept-Language: en-US,en;q=0.9			
	Cache-Control: no-cache			
	Connection: Upgrade			
	Host: echo.websocket.events			
	Origin: null			
	Pragma: no-cache			
	Sec-WebSocket-Extensions: <code>permessage-deflate; client_max_window_bits</code>			
	Sec-WebSocket-Key: <code>enFko0a9HDsIQq064u8Z2w==</code>			
	Sec-WebSocket-Version: 13			
	Upgrade: websocket			
	User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/53			

2 requests | 830 B transferred

Figure 8.12.3: Viewing WebSocket messages in Chrome DevTools.

**PARTICIPATION
ACTIVITY**

8.12.3: WebSocket handshake.

- 1) A WebSocket handshake _____ contains a **Sec-WebSocket-Key** HTTP header.
☐ request
☐ response
- 2) **Connection** and **Upgrade** headers are in both the HTTP request and response in a WebSocket handshake.
☐ True
☐ False

- 3) The code below causes the browser to send a WebSocket handshake request to the server.



```
websocket.send("User 123 likes photo ABC");
```

- ☐ True
- ☐ False
- 4) A web server that supports WebSockets will return a ____ status code.
- ☐ 200
- ☐ 101
- 5) The Chrome DevTools display all WebSocket messages between the client and server.
- ☐ True
- ☐ False



PARTICIPATION ACTIVITY

8.12.4: Create a parrot chat app with WebSockets.



Create a web application that allows the user to chat with a parrot. The parrot only replies with what the parrot hears, so the `echo.websockets.org` WebSocket server is perfect for implementing the chat application.

Add the following JavaScript:

1. Create a WebSocket connection to `wss://echo.websocket.events/` in the Connect button's callback function with the `websocket` variable.
2. Implement an open event callback function that sets the `connectionStatus.innerHTML` to "Connected".
3. Implement a close event callback function that sets the `connectionStatus.innerHTML` to "Disconnected".

4. Implement a message event callback function that sets the `response.innerHTML` to the message received from the server.
5. Finally, add code to close the WebSocket connection when the user clicks the Disconnect button.

After rendering the webpage, click the Connect button and verify the Status changes from "Disconnected" to "Connected". Type a message in the text box and click Send. Verify the same message is displayed after "Parrot's response:". Send a few more messages before clicking Disconnect, and verify the Status changes from "Connected" to "Disconnected".

HTML

JavaScript

```
1 <p>
2   <button id="connectBtn">Connect</button>
3   <button id="disconnectBtn">Disconnect</button>
4 </p>
5
6 <p>
7   Status: <span id="connectionStatus"></span>
8 </p>
9
10 <p>
11   <input type="text" id="message">
12   <button id="sendBtn">Send</button>
13 </p>
14
15 <p>
16   Parrot's response: <strong id="response"></strong>
```

Render webpage

Reset code

Your webpage

Status: Disconnected

Parrot's response:

► View solution

Exploring further:

- [WebSocket standard](#)
- [websocketd](#) - Command-line tool to create a WebSocket server
- [Writing WebSocket servers](#)

8.13 Promises

Synchronous and asynchronous functions

A **synchronous function** is a function that completes an operation before returning. Ex: A function to sort an array will return only after the entire array is sorted.

An **asynchronous function** is a function that starts an operation and potentially returns before the operation completes. The operation completes in the background, allowing other code to execute in

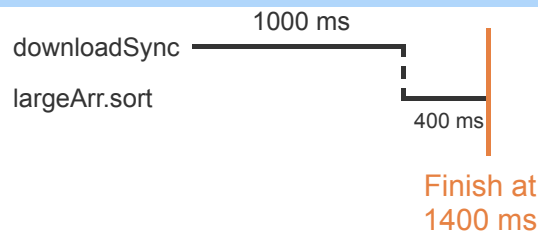
the meantime. Ex: A function to download data is often implemented asynchronously, allowing other code to execute while the download runs in the background. A callback function is commonly passed to an asynchronous function and is called when the operation completes.

PARTICIPATION ACTIVITY

8.13.1: Asynchronous functions are commonly used to download data in the background.

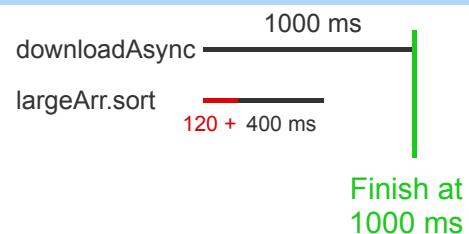


```
function doWork(downloadURL, largeArr) {
  let data = downloadSync(downloadURL);
  largeArr.sort();
  // do something with data and largeArr
}
```



```
function doWork(downloadURL, largeArr) {
  downloadAsync(downloadURL, listener);
  largeArr.sort();
}

function listener() {
  // do something with data and largeArr
}
```



Animation content:

The following JavaScript is displayed on the left:

```
function doWork(downloadURL, largeArr) {
  let data = downloadSync(downloadURL);
  largeArr.sort();
  // do something with data and largeArr
}
```

The following JavaScript is displayed on the right:

```
function doWork(downloadURL, largeArr) {
  downloadAsync(downloadURL, listener);
  largeArr.sort();
}
```

```
function listener() {
```

```
// do something with data and largeArr  
}
```

Animation captions:

1. Downloading data from the web is often a slow task. The `doWork()` implementation on the left uses `downloadSync()` to synchronously download data.
2. The `doWork()` implementation on the right uses `downloadAsync()` to download data asynchronously.
3. If the data takes 1000 milliseconds (ms) to download, the `downloadSync()` call returns after 1000 ms.
4. After the download, sorting the array takes another 400 ms. The entire function finishes at 1400 ms.
5. The asynchronous download function starts the download but returns just after starting. The download still takes 1000 ms.
6. `downloadAsync()` takes 120ms to start the operation, but then returns. So the array sort can start after about 120 ms. The download continues in the background.
7. `downloadAsync()` calls the listener function when the download completes. The array sort executed concurrently with the download, so only 1000 ms were required for both operations.

PARTICIPATION ACTIVITY

8.13.2: Asynchronous function result data.

1) In the animation above, the **data** variable is declared in the synchronous **doWork()** function, but not the asynchronous. The data variable in the asynchronous code should be ____.

- ☐ declared as a global variable
- ☐ passed as an argument to the **listener()** function
- ☐ declared as a local variable in **doWork()**, the same as the synchronous version

**PARTICIPATION
ACTIVITY**

8.13.3: Synchronous and asynchronous functions.

Consider the following code.

```
function doWork2(downloadURL) {  
  let dataArray = download(downloadURL);  
  dataArray.sort();  
}
```

1) Suppose the `doWork2()` function is intended to download an array of data and then sort the downloaded array. For proper functionality, the download function ____.

- ☐ must be synchronous
- ☐ must be asynchronous
- ☐ can be either synchronous or asynchronous

2) Assume the `download()` function is synchronous but the `sort()` function is asynchronous. If no errors occur, then by the time `doWork2()` finishes, ____.

- ☐ the data array is guaranteed to be downloaded and sorted
- ☐ the data array is guaranteed to be downloaded, but may not be sorted
- ☐ the data array is not guaranteed to be downloaded or sorted

Promise object

An asynchronous function cannot return the result of the operation since the function may return before the operation completes. So asynchronous functions often return a **Promise object**. An

object representing the eventual completion of the asynchronous operation.

A Promise object can be in one of three states: pending, fulfilled, or rejected.

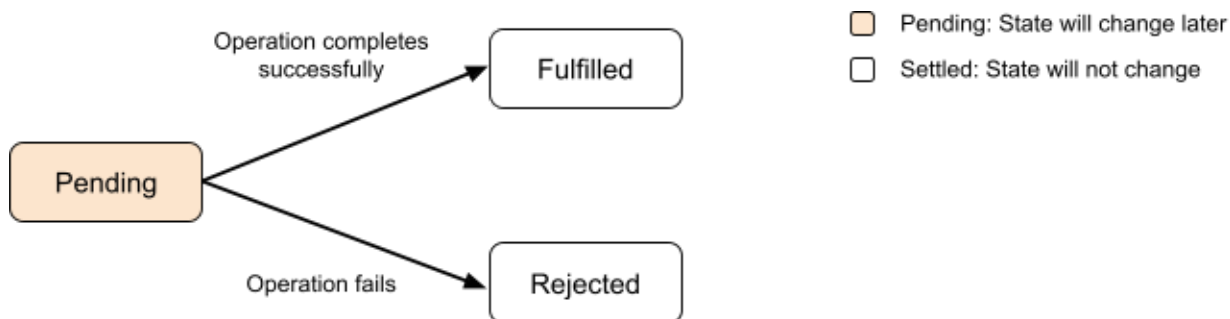
- **Pending** means that the asynchronous operation is still running.
- **Fulfilled** means that the asynchronous operation has completed successfully.
- **Rejected** means that the asynchronous operation has ended in failure to produce the intended result.

Once reaching the fulfilled or rejected state, the Promise object is **settled**, and the state will not change again.

The Promise constructor has a single parameter, an **executor function** that executes in the background. The executor function has two parameters:

- **resolve** - Function to call when the executor function has completed successfully (state becomes fulfilled)
- **reject** - Function to call when the executor function has completed unsuccessfully (state becomes rejected)

Figure 8.13.1: Promise state transition diagram.



PARTICIPATION ACTIVITY

8.13.4: Function that returns a Promise object.

```
function saveToCloudAsync(userData) {  
  return new Promise(function(resolve, reject) {  
    if (!userData) {  
      reject();  
    }  
  })  
}
```

```
// Simulate saving taking 2 seconds
setTimeout(resolve, 2000);
});
}
```

saveToCloudAsync(null);

Promise state:

pending

rejected

saveToCloudAsync({data: 5});

Promise state:

pending

fulfilled

Animation content:

The following code is displayed.

```
function saveToCloudAsync(userData) {
  return new Promise(function(resolve, reject) {
    if (!userData) {
      reject();
    }

    // Simulate saving taking 2 seconds
    setTimeout(resolve, 2000);
  });
}
```

Step 3: The function call `saveToCloudAsync(null)` assigns `null` to parameter `userData`.

Step 5: The function call `saveToCloudAsync({data:5})` assigns `{data: 5}` to parameter `userData`.

Animation captions:

1. The function `saveToCloudAsync()` simulates saving user data to a remote server.
2. `saveToCloudAsync()` returns a Promise object. The Promise constructor's executor function has `resolve` and `reject` parameters.
3. When `saveToCloudAsync()` is called and the Promise object is first created, the Promise is in the pending state.
4. Since `userData` is `null`, `reject()` is called. The Promise moves to the rejected state.
5. When `saveToCloudAsync()` is called with an object, the new Promise object is in the pending state.
6. Since `userData` is not `null`, `setTimeout()` is called to simulate an executor function that saves `userData` to a remote server.

7. `resolve()` is called after 2 seconds, which changes the Promise state to fulfilled.

**PARTICIPATION
ACTIVITY**

8.13.5: Promise object.



- 1) A Promise object can represent an asynchronous operation that has ____.
- ☐ not started
 - ☐ completed
- 2) A Promise may change state if in the ____ state.
- ☐ pending
 - ☐ fulfilled
 - ☐ rejected
- 3) A settled Promise will not be in the ____ state.
- ☐ pending
 - ☐ fulfilled
 - ☐ rejected
- 4) In the animation above, how long does the Promise stay in the pending state when calling `saveToCloudAsync({data: 5})`?
- ☐ Less than 1 second
 - ☐ 2 seconds
 - ☐ More than 3 seconds



Promise.then() method

A Promise object's **then()** method can be called to request notifications about the state. Two arguments are passed to the **then()** method. The first is a function to be called if the Promise is fulfilled, and the second is a function to be called if the Promise is rejected.

The **then()** method can be called when the Promise object is in any state. The fulfilled callback function will eventually be called if either of the following is true:

- the Promise is already fulfilled when **then()** is called, or
- the Promise is pending when **then()** is called and is eventually fulfilled.

Similarly, the rejected callback function will eventually be called if either of the following is true:

- the Promise is already rejected when **then()** is called, or
- the Promise is pending when **then()** is called and is eventually rejected.

PARTICIPATION ACTIVITY

8.13.6: Calling then() with fulfilled and rejected callbacks.



```
function saveUserData(data) {  
  let promise = saveToCloudAsync(data);  
  promise.then(dataSaved, saveFailed);  
}
```

```
function dataSaved() {  
  console.log("Data saved to cloud!");  
}
```

```
function saveFailed() {  
  console.error("Failed to save data to cloud");  
}
```

Order of function calls:

1. saveUserData (starts)
2. saveToCloudAsync
3. promise.then
4. saveUserData (completes)
5. **dataSaved**
OR
saveFailed

Animation content:

The following code snippet is displayed.

```
function saveUserData(data) {  
  let promise = saveToCloudAsync(data);  
  promise.then(dataSaved, saveFailed);  
}
```

```
function dataSaved() {  
  console.log("Data saved to cloud!");  
}  
  
function saveFailed() {  
  console.error("Failed to save data to cloud");  
}
```

Order of function list:

1. saveUserData(starts)
2. saveToCloudAsync
3. Promise.then
4. saveUserData(completes)
5. dataSaved OR saveFailed

Animation captions:

1. The saveUserData() function calls saveToCloudAsync(), which returns a Promise object.
2. The Promise object's then() method is called with 2 functions provided as arguments. saveUserData() returns while the async save attempt continues in the background.
3. If the save attempt succeeds, the dataSaved() function is eventually called.
4. If the save attempt fails, saveFailed() is eventually called.

PARTICIPATION ACTIVITY

8.13.7: Promise.then() method.



Refer to the animation above.

- 1) If the Promise object returned from **saveToCloudAsync ()** was rejected before calling **then ()**, then neither **dataSaved ()** nor **saveFailed ()** would be called.



- ☐ True
- ☐ False

- 2) A partially successful save would result in both `dataSaved()` and `saveFailed()` being called.
- ☐ True
- ☐ False
- 3) Reversing argument order and calling `promise.then(saveFailed, dataSaved)` would result in `saveFailed()` being called on fulfillment, and `dataSaved()` being called on rejection.
- ☐ True
- ☐ False
- 4) The `saveUserDataNew()` function below works like `saveUserData()` from the animation, except `saveUserDataNew()` uses anonymous functions.



```
function saveUserDataNew(data)
{
    let promise =
saveToCloudAsync(data);
    promise.then(
        function() {
            console.log("Data
saved to cloud!");
        },
        function() {
            console.error("Failed
to save data to cloud");
        }
    );
}
```

- ☐ True
- ☐ False

- 5) The `promise` variable is not necessary to call `then()`.



```
function saveUserData(data) {  
  saveToCloudAsync(data).then(  
    function() {  
      console.log("Data  
saved to cloud!");  
    },  
    function() {  
      console.error("Failed  
to save data to cloud");  
    }  
  );  
}
```

- ☐ True
- ☐ False

Promise fulfillment values and rejection reasons

A pending Promise is either fulfilled with a value or rejected with a reason. A fulfilled Promise passes the fulfillment value as an argument to the fulfilled callback function. A rejected Promise passes the rejection reason as an argument to the rejected callback function.

A rejection reason is commonly an Error object. The type of a fulfillment value varies based on the type of Promise. Ex: An asynchronous function that downloads data may return a Promise that passes the downloaded data string as the fulfillment value.

PARTICIPATION ACTIVITY

8.13.8: Promise fulfillment values and rejection reasons.



```
function loadFromCloudAsync(filename) {  
  return new Promise(function(resolve, reject) {  
    if (filename === "Hello.txt") {  
      // Simulate loading taking 2 seconds  
      setTimeout(() => resolve("Hello World!"), 2000);  
    }  
    else {  
      reject(new Error("File does not exist"));  
    }  
  });  
}
```

Cloud storage:

Hello.txt file contents:

Hello World!

(OtherFile.txt does not
exist)


```
function loadString(filename) {
    let promise = loadFromCloudAsync(filename);
    promise.then(dataLoaded, loadFailed);
}

function dataLoaded(value) {
    console.log("Data loaded from cloud: " + value);
}

function loadFailed(reason) {
    console.error(reason.toString());
}
```

```
loadString("Hello.txt");
```

Console (on success):

Data loaded from cloud: Hello World!

```
loadString("OtherFile.txt");
```

Console (on failure):

Error: File does not exist

Animation content:

The following JavaScript is displayed:

```
function loadFromCloudAsync(filename) {
    return new Promise(function(resolve, reject) {
        if (filename === "Hello.txt") {
            // Simulate loading taking 2 seconds
            setTimeout(() => resolve("Hello World!"), 2000);
        }
        else {
            reject(new Error("File does not exist"));
        }
    });
}
```

```
function loadString(filename) {
    let promise = loadFromCloudAsync(filename);
    promise.then(dataLoaded, loadFailed);
}
```

```
function dataLoaded(value) {
    console.log("Data loaded from cloud: " + value);
}
```

```
function loadFailed(reason) {  
    console.error(reason.toString());  
}
```

There is a display box labeled Cloud storage.

Step 3: Hello World is loaded into the cloud storage.

Step 4: The console displays "Data loaded from cloud: Hello World!"

Step 5: The console displays "Error: File does not exist".

Animation captions:

1. The `loadFromCloudAsync()` function simulates loading a string from a file in cloud storage. `loadFromCloudAsync()` returns a Promise object.
2. `loadString()` calls `loadFromCloudAsync()` and `then()` on the returned Promise.
3. Since filename is `Hello.txt`, `loadFromCloudAsync()` calls `resolve()` with a string argument after 2 seconds.
4. `dataLoaded()` is called with the argument "Hello World!". `dataLoaded()` logs the string to the console.
5. An attempt to load a non-existent file results in a `reject()` call with an Error object.
6. `loadFailed()` is called with the Error object as an argument. `loadFailed()` logs the error message to the console.

PARTICIPATION ACTIVITY

8.13.9: Promise fulfillment values and rejection reasons.

Refer to the animation above.

- 1) If the cloud storage operation loaded only part of the file and then got disconnected, the `dataLoaded()` function would likely be called.

- ☐ True
- ☐ False

2) If `loadFromCloudAsync()` calls `resolve()` with an array, `dataLoaded()`'s `value` parameter would be an array.

- ☐ True
☐ False

3) If `loadFromCloudAsync()` calls `reject()` with a string, `loadFailed()`'s `reason` parameter is still an Error object.

- ☐ True
☐ False

4) A function that actually loads data from the cloud might result in `loadFailed()` being called, even when the filename exists.

- ☐ True
☐ False

Promise.catch() method

The second parameter to the `then()` method is optional. If omitted, the parameter defaults to a function that throws an exception.

A Promise object's **`catch()`** method takes a single argument that is a function to call if the Promise is rejected or if the fulfilled handler throws an exception. Consider the two statements:

1. `promiseObj.then(okFunc, failFunc);`
2. `promiseObj.then(okFunc).catch(failFunc);`

While having some similarity, the two statements are not equivalent. The first statement will call either `okFunc()` or `failFunc()`, but not both. The second statement will call `okFunc()` if `promiseObj` is fulfilled, and then also call `failFunc()` if `okFunc()` throws an exception. Both will call only `failFunc()` if `promiseObj` is rejected.

Statements of the form `promiseObj.then(okFunc).catch(failFunc);` are commonly

used when working with Promises.

Table 8.13.1: Comparison of Promise object's then() and catch() usage scenarios.

Code	Scenario	Function(s) called	Uncaught exception?
<code>promise1.then(okFunc, failFunc);</code>	promise1 fulfilled, okFunc() does NOT throw an exception	okFunc() only	No
	promise1 fulfilled, okFunc() throws an exception	okFunc() only	Yes
	promise1 rejected, failFunc() does NOT throw an exception	failFunc() only	No
	promise1 rejected, failFunc() throws an exception	failFunc() only	Yes
	promise1 fulfilled, okFunc() does NOT throw an	okFunc() only	No

<pre>promise1.then(okFunc).catch(failFunc);</pre>	exception		
	promise1 fulfilled, okFunc() throws an exception	okFunc() first, then failFunc()	No
	promise1 rejected, failFunc() does NOT throw an exception	failFunc() only	No
	promise1 rejected, failFunc() throws an exception	failFunc() only	Yes

**PARTICIPATION
ACTIVITY**

8.13.10: Promise.catch() method.



Suppose `promise2` is a Promise object in the pending state. Consider the following functions:

```
function noThrow(arg) {  
    return arg;  
}  
function yesThrow() {  
    throw new Error("Error message");  
}
```

If unable to drag and drop, refresh the page.

```
promise2.then(noThrow).catch(yesThrow);
```

```
promise2.then(yesThrow, noThrow);
```

```
promise2.then(yesThrow).catch(noThrow);
```

Results in an uncaught exception if
promise2 is fulfilled.

Results in an uncaught exception if
promise2 is rejected.

Does not result in an uncaught
exception, regardless of whether
promise2 is fulfilled or rejected.

[Reset](#)

Exploring further:

- [Promise \(MDN\)](#).
- [Using promises \(MDN\)](#).

8.14 async and await

Async function

An async function provides simpler syntax for creating an asynchronous function that returns a Promise. An **async function** is a function that is declared with the **async** keyword. The **async keyword** must appear in a function declaration before the **function** keyword. An async function

fulfills the Promise by returning a value, or rejects the Promise by throwing an exception.

**PARTICIPATION
ACTIVITY**

8.14.1: Async function returns a Promise.



```
async function loadFromCloudAsync(filename) {
  if (filename === "Hello.txt") {
    return "Hello World!";
  }
  else {
    throw Error("File does not exist");
  }
}

function dataLoaded(value) {
  console.log("Data loaded: " + value);
}

function loadFailed(reason) {
  console.error(reason.toString());
}
```

```
loadFromCloudAsync("Hello.txt")
  .then(dataLoaded).catch(loadFailed);
```

Data loaded: Hello World!

```
loadFromCloudAsync("Otherfile.txt")
  .then(dataLoaded).catch(loadFailed);
```

Error: File does not exist

Animation content:

The following code snippet is displayed.

```
async function loadFromCloudAsync(filename) {
  if (filename === "Hello.txt") {
    return "Hello World!";
  }
  else {
    throw Error("File does not exist");
  }
}

function dataLoaded(value) {
```

```
    console.log("Data loaded: " + value);  
  }  
  
  function loadFailed(reason) {  
    console.error(reason.toString());  
  }  
}
```

Step 3: The console displays "Data loaded: Hello World!"

Step 5: The console displays "Error: File does not exist".

Animation captions:

1. The `async` keyword appears before the function declaration, so `loadFromCloudAsync()` returns a `Promise`.
2. Calling `loadFromCloudAsync()` with "Hello.txt" returns a fulfilled `Promise` with value "Hello World!".
3. `dataLoaded()` logs the fulfilled value to the console.
4. For all other filenames, `loadFromCloudAsync()` returns a rejected `Promise` with an `Error` reason by throwing the `Error`.
5. `loadFailed()` logs the error message to the console.

PARTICIPATION ACTIVITY

8.14.2: Async function.



Complete the `async` function, which returns a fulfilled `Promise` that resolves to `true` if the given number is even or `false` if the number is odd. If the argument is not a number, the `async` function rejects the `Promise`.


```
__A__ function isEvenAsync(num) {  
  if (!Number.isInteger(num)) {  
    __B__ Error("Argument is not an integer");  
  }  
  
  __C__ num % 2 == 0;  
}  
  
// Should resolve to false  
isEvenAsync(4)  
  .then(result => alert("Result: " + result))  
  .catch(error => alert(error));  
  
// Should be rejected  
isEvenAsync("dog")  
  .then(result => alert("Result: " + result))  
  .catch(error => alert(error));
```

If unable to drag and drop, refresh the page.

C

B

A

return

async

throw

Reset

await keyword

The `await` keyword simplifies waiting for a Promise to resolve. The ***await keyword*** is used in an `async` function to wait for a Promise object to resolve before proceeding to the next statement. Waiting for an asynchronous operation to complete pauses the `async` function's execution. Then, after the Promise eventually resolves, execution resumes inside the `async` function at the statement following the `await`.

**PARTICIPATION
ACTIVITY**

8.14.3: await allows an async function to wait for completion of an asynchronous operation.



```
function loadData(filename) {  
  console.log("Loading data...");  
  let promise = loadFromCloudAsync(filename);  
  promise.then(function(data) {  
    console.log("Data loaded: " + data);  
  });  
}
```

```
loadData("Hello.txt");
```

Loading data...
Data loaded: Hello World!

```
async function loadDataAsync(filename) {  
  console.log("Loading data...");  
  let data = await loadFromCloudAsync(filename);  
  console.log("Data loaded: " + data);  
}
```

```
loadDataAsync("Hello.txt");
```

Loading data...
Data loaded: Hello World!

Animation content:

The following code snippet is displayed.

```
function loadData(filename) {  
  console.log("Loading data...");  
  let promise = loadFromCloudAsync(filename);  
  promise.then(function(data) {  
    console.log("Data loaded: " + data);  
  });  
}  
  
async function loadDataAsync(filename) {  
  console.log("Loading data...");  
  let data = await loadFromCloudAsync(filename);  
  console.log("Data loaded: " + data);  
}
```

Step 1: The call `loadData("Hello.txt")` results in the console displaying "Loading data..."

Step 3: The console displays "Data loaded: Hello World!"

Step 5: The call `loadDataAsync("Hello.txt")` results in the console displaying "Loading data..."

Step 6: The console displays "Data loaded: Hello World!"

Animation captions:

1. `loadData()` calls `loadFromCloudAsync()`, which returns a Promise object representing an asynchronous load operation.
2. The `then()` method supplies a callback for when the Promise is fulfilled.
3. When the Promise is fulfilled, the data is logged to the console.
4. Alternatively, `loadData()` can be implemented without a callback function by using `await`. Any function using the `await` operator must be declared with the `async` keyword.
5. Execution begins similarly. When the `await` keyword is reached, execution leaves `loadDataAsync()` while the load completes.
6. When the load completes and the Promise resolves, execution resumes in `loadDataAsync()`.

PARTICIPATION ACTIVITY

8.14.4: await keyword.

Refer to the animation above.

- 1) `loadData()` and `loadDataAsync()` allow additional events to be processed while the load completes in the background.

- ☐ True
☐ False

- 2) `loadData()` and `loadDataAsync()` both risk an uncaught exception if `loadFromCloudAsync()` rejects the Promise.

- ☐ True
☐ False

3) An exception is thrown if the **async** keyword is removed from `loadDataAsync()`.

- ☐ True
- ☐ False

Handling rejected Promises

An async function uses a try-catch statement to catch the Error object from a rejected Promise.

PARTICIPATION ACTIVITY

8.14.5: Catching the Error from a rejected Promise.

```
async function loadDataAsync(filename) {  
  console.log("Loading data...");  
  let data = await loadFromCloudAsync(filename);  
  console.log("Data loaded: " + data);  
}
```

```
loadDataAsync("Unknown.txt");
```

Loading data...
Uncaught (in promise) Error:
File does not exist

```
async function loadDataAsync(filename) {  
  console.log("Loading data...");  
  try {  
    let data = await loadFromCloudAsync(filename);  
    console.log("Data loaded: " + data);  
  }  
  catch (exception) {  
    console.log(exception.toString());  
  }  
}
```

```
loadDataAsync("Unknown.txt");
```

Loading data...
Error: File does not exist

Animation content:

Step 1: Displays the following JavaScript:

```
async function loadDataAsync(filename) {  
  console.log("Loading data...");  
  let data = await loadFromCloudAsync(filename);  
  console.log("Data loaded: " + data);  
}
```

The call `loadDataAsync("Unknown.txt")` results in the console displaying "Uncaught (in promise) Error: File does not exist."

Step 2: Displays the following JavaScript:

```
async function loadDataAsync(filename) {  
  console.log("Loading data...");  
  try {  
    let data = await loadFromCloudAsync(filename);  
    console.log("Data loaded: " + data);  
  }  
  catch (exception) {  
    console.log(exception.toString());  
  }  
}
```

Step 3: The call `loadDataAsync("Unknown.txt")` results in the console displaying: "Error: File does not exist."

Animation captions:

1. If `loadFromCloudAsync()` rejects the Promise, the async function terminates immediately because the Error is not caught. The uncaught Error is typically logged to the console.
2. The async function can use a try-catch statement to catch the Error object.
3. If `loadFromCloudAsync()` rejects the Promise in the try block, the catch block executes. The async function logs the Error and then terminates normally.

PARTICIPATION ACTIVITY

8.14.6: Handling rejected Promises.



Refer to the animation above.

- 1) Which statement in the top `loadDataAsync()` does NOT execute when the Promise is rejected?
- ☐ `console.log("Loading data...");`
- ☐ `let data = await loadFromCloudAsync(filename);`
- ☐ `console.log("Data loaded: " + data);`
- 2) Which variable is assigned the Error object containing the rejected Promise's reason?
- ☐ `filename`
- ☐ `exception`
- ☐ `data`
- 3) What does the bottom `loadDataAsync()` function log when the Promise is fulfilled with string "XYZ"?
- ☐ "Data loaded: XYZ" only
- ☐ Error message only
- ☐ "Data loaded: XYZ" and error message

Exploring further:

- [async function \(MDN\)](#)
- [await \(MDN\)](#)

8.15 Fetch API

Introduction to Fetch

The **Fetch API** defines a **fetch()** method for sending HTTP requests and receiving HTTP responses. The `fetch()` method is a replacement for using the `XMLHttpRequest` object directly and is generally easier to use.

The `fetch()` method sends a GET request (by default) to the given URL argument and returns a Promise object. The Promise object resolves to a **Response** object, which contains information about the HTTP response and methods for retrieving the response body. Some properties and methods of **Response** include:

- The **response.status** property is the HTTP response's status code (200, 301, 404, etc.)
- The **response.ok** property is true if the HTTP response's status code is 2xx, false otherwise.
- The **response.headers** property is an object containing the HTTP response headers.
- The **response.text()** method returns a Promise that resolves with the textual body of the HTTP response.

The web browser restricts `fetch()` from sending a cross-origin HTTP request, which is a request made to another domain. Ex: If `fetch()` is called from JavaScript downloaded from `abc.com`, the browser restricts a cross-origin HTTP request to `xyz.com`. A web server can implement Cross-Origin Resource Sharing (CORS) to allow cross-origin requests.

PARTICIPATION ACTIVITY

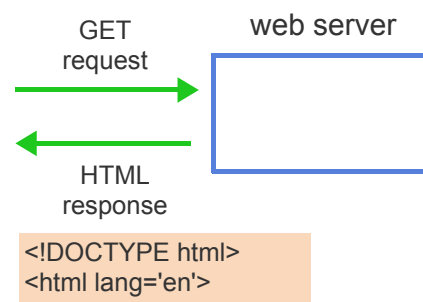
8.15.1: Fetch request for a webpage.



```
let url = "https://learn.zybooks.com/";
let response = await fetch(url);
console.log("status is " + response.status);

let html = await response.text();
console.log(html);
```

console



```
status is 200
<!DOCTYPE html>
<html lang='en'>
  <head>
    <meta charset='utf-8'>
    ...
```

```
<head>
  <meta charset='utf-8'>
  ...
```

Animation content:

The following JavaScript is displayed:

```
let url = "https://learn.zybooks.com/";
let response = await fetch(url);
console.log("status is " + response.status());
let html = await response.text();
console.log(html);
```

Step 3: The console displays "status is 200".

Step 4: The console displays the HTML file contents.

Animation captions:

1. The `fetch()` method sends an HTTP GET request to the `zybooks.com` web server. The `await` operator waits for the HTTP response to return before continuing.
2. The web server responds with the HTML for the URL `https://learn.zybooks.com`.
3. The `status` property is the HTTP status code. 200 means success.
4. The `text()` method returns the response body containing HTML as text.

Alternative syntax

Some developers prefer the alternative Promise syntax shown below when using the Fetch API.

```
let url = "https://learn.zybooks.com/";
fetch(url)
  .then(function(response) {
    console.log("status is " + response.status);
    return response.text();
  })
  .then(function(html) {
    console.log(html);
  })
  .catch(function(error) {
    console.log("Request failed", error)
  });
```

The same code can be written more concisely using arrow functions:

```
fetch(url)
  .then(response => {
    console.log("status is " + response.status);
    return response.text();
  })
  .then(html => console.log(html))
  .catch(error => console.log("Request failed", error));
```

PARTICIPATION ACTIVITY

8.15.2: Fetch.

1) Which HTTP method type does `fetch()` send by default?

- ☐ GET
- ☐ POST
- ☐ JSON

- 2) What is missing from the code segment to fetch the given URL?



```
let url =  
"https://www.example.com/";  
let response = _____;  
console.log(response.status);
```

- ☐ fetch()
- ☐ fetch(url)
- ☐ await fetch(url)

- 3) Why might the code below output "404" to the console?



```
let url =  
"https://www.example.com/trythis";  
let response = await fetch(url);  
console.log(response.status);
```

- ☐ fetch() is not called correctly
- ☐ The url variable is assigned a URL that points to an invalid resource
- ☐ response.text() has not been called

- 4) What is missing to output the HTML returned in the response?



```
let url =  
"https://www.example.com/";  
let response = await  
fetch(url);  
let html = _____;  
console.log(html);
```

- ☐ response.text()
- ☐ await response
- ☐ await response.text()

- 5) What happens if the code below is from zybooks.com, and example.com does not allow cross-origin requests?



```
let url =  
"https://www.example.com/";  
let response = await  
fetch(url);  
console.log(response.status);
```

- ☐ `fetch()` throws an exception
- ☐ `fetch()` works normally
- ☐ `response.status` is assigned 500

Fetching JSON

Developers often use the Fetch API to make requests to web APIs that return JSON responses. The **`response.json()`** method returns a Promise that resolves to an object created from parsing the response body as JSON.

The animation below makes a request to the [Random User Generator](#), a web API that generates random user data.

PARTICIPATION ACTIVITY

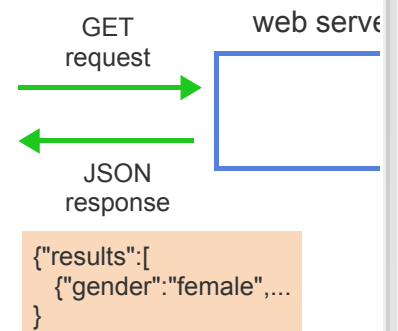
8.15.3: Fetch request that returns JSON.



```
let url = "https://randomuser.me/api/?results=3";  
let response = await fetch(url);  
let users = await response.json();  
for (let user of users.results) {  
  console.log(`name = ${user.name.first} ${user.name.last}`);  
  console.log(`gender = ${user.gender}`);  
  console.log(`email = ${user.email}`);  
}
```

console

```
name = Judy Welch  
gender = female  
email = judy.welch@example.com  
name = Phillip Brewer  
gender = male
```



```
email = phillip.brewer@example.com  
name = Auguste Riviere  
gender = male  
email = auguste.riviere@example.com
```

Animation content:

The following JavaScript is displayed:

```
let url = "https://randomuser.me/api/?results=3";  
let response = await fetch(url);  
let users = await response.json();  
for (let user of users.results) {  
  console.log(`name = ${user.name.first} ${user.name.last}`);  
  console.log(`gender = ${user.gender}`);  
  console.log(`email = ${user.email}`);  
}
```

Step 4: The resulting output of the code is a console log of the following:

```
Name = Judy Welch  
Gender = female  
Email = judy.welch@example.com  
Name = Phillip Brewer  
Gender = male  
Email = philip.brewer@example.com  
Name = Auguste Riviere  
Gender = male  
Email = auguste.riviere@example.com
```

Animation captions:

1. The `fetch()` method sends an HTTP GET request to the randomuser.me web server.
2. The web server returns JSON that encodes information for 3 randomized users.
3. The `json()` method parses the JSON response and returns an object containing the JSON data.
4. The for-of loop outputs the name, gender, and email address of the three users to the console.

PARTICIPATION
ACTIVITY

8.15.4: Fetch weather data.



The URL `https://wp.zybooks.com/weather.php?zip=XXXXX`, where XXXXX is a five digit ZIP code, returns JSON containing a randomly produced forecast for the given ZIP code. If the ZIP code is not given or is not five digits, the JSON response indicates the ZIP code is not found.

Successful request	Unsuccessful request
<pre>{ "success": true, "forecast": [{ "high": 90, "low": 72, "desc": "sunny" }, { "high": 92, "low": 73, "desc": "mostly sunny" }, { "high": 87, "low": 64, "desc": "rain" }, { "high": 88, "low": 65, "desc": "cloudy" }, { "high": 90, "low": 68, "desc": "partly cloudy" }] }</pre>	<pre>{ "success": false, "error": "ZIP code not found" }</pre>

Enter any ZIP code in the webpage below, and click the Search button. When Search is clicked, the `fetch()` method requests the URL above with the ZIP code entered in the form. The raw JSON response is displayed in the webpage, which is not ideal.

Make the following changes:

1. Replace the call to `response.text()` with the method that parses a JSON response:

```
//let json = await response.text();
let weather = await response.json();
```

2. Replace the code that appends the raw JSON to the `html` string with code that loops through the `weather.forecast` array and produces a numbered list with each day's forecast:

```
//html += json;
html += "<ol>";
for (let day of weather.forecast) {
    html += "<li>" + day.desc + ": high is " + day.high + ", low is "
+ day.low + "</li>";
}
html += "</ol>";
```

©zyBooks 04/15/24 16:42 2071381

Marco Aguilar

CIS192_193_Spring_2024

3. Render the webpage, and verify the changes you have made work correctly to show the weather for the ZIP code you enter.
4. Finally, modify the code to display an appropriate error message if the response indicates the ZIP code is not found. Hint: Examine the `weather.success` property.
5. Render the webpage, and verify that entering a bad ZIP code like "123" produces an error message.

HTML

JavaScript

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <title>Weather Forecast</title>
5 </head>
6 <body>
7   <p>
8     <label for="zip">ZIP code:</label>
9     <input type="text" id="zip" maxlength="5">
10    <button id="search">Search</button>
11  </p>
12  <div id="forecast"></div>
13 </body>
14 </html>
15
```

Render webpage

Reset code

Your webpage

ZIP code:

Expected webpage

ZIP code:

► View solution

**PARTICIPATION
ACTIVITY**

8.15.5: Fetching JSON.



A web API responds with the JSON below.

```
{
  "success": true,
  "questions": [
    { "text": "Who is often called the father of the computer?",
      "answer": "Charles Babbage" },
    { "text": "Who is considered the first programmer?", "answer": "Ada
Lovelace" },
    { "text": "What computer popularized the Graphical User Interface
in 1984?", "answer": "Macintosh" },
    { "text": "What is the name of the first modern-day digital
computer?", "answer": "ENIAC" }
  ]
}
```

- 1) The `fetch()` method must specify the returned response is expected to contain JSON.
- ☐ True
- ☐ False
- 2) The `response.json()` method returns an object with two properties: `success` and `questions`.
- ☐ True
- ☐ False
- 3) According to the code below, `trivia.questions[1].answer` is Charles Babbage.
- ```
let trivia = await
response.json();
```
- ☐ True
- ☐ False

## POST request

The `fetch()` method has an optional second parameter, an object that specifies options to modify the HTTP request. Some common `fetch()` options:

- The **`method`** option indicates the HTTP method. Ex: GET, POST, PUT, and DELETE.
- The **`headers`** option specifies various HTTP request headers. Ex: Content-Type and User-Agent.
- The **`body`** option specifies the HTTP request body, which could be form data, a JSON-encoded string, or binary data.

The animation below POSTs form data using `fetch()` and the `FormData` object. The JavaScript object **`FormData`** stores key/value pairs from a form submission.



PARTICIPATION  
ACTIVITY

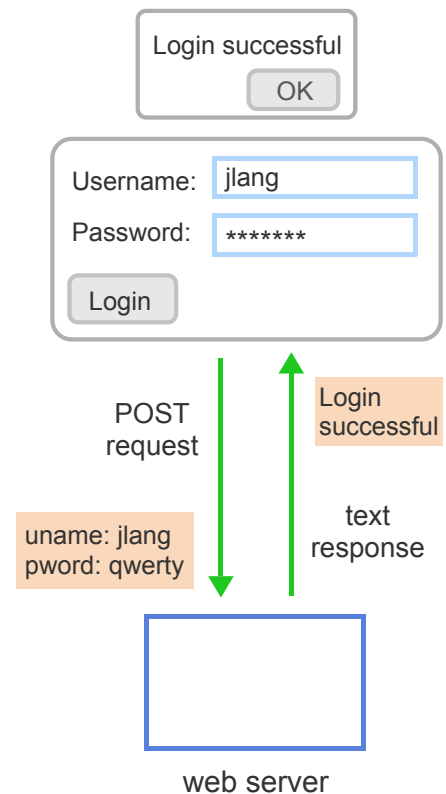
## 8.15.6: POSTing form data.



```
<form>
 <p>
 <label for="uname">Username:</label>
 <input type="text" id="uname" name="uname">
 </p>
 <p>
 <label for="pword">Password:</label>
 <input type="password" id="pword" name="pword">
 </p>
 <p>
 <input type="submit" value="Login">
 </p>
</form>
```

```
let form = document.querySelector("form");
form.addEventListener("submit", async function(e) {
 e.preventDefault();
 let response = await fetch("login", {
 method: "POST",
 body: new FormData(form)
 });

 let result = await response.text();
 alert(result);
});
```

**Animation content:**

The following code snippet is displayed.

```
<form>
 <p>
 <label for="uname">Username:</label>
 <input type="text" id="uname" name="uname">
 </p>
 <p>
 <label for="pword">Password:</label>
 <input type="password" id="pword" name="pword">
 </p>
```

```
<p>
 <input type="submit" value="Login">
</p>
</form>
```

```
let form = document.querySelector("form");
form.addEventListener("submit", async function(e) {
 e.preventDefault();
 let response = await fetch("login", {
 method: "POST",
 body: new FormData(form)
 });

 let result = await response.text();
 alert(result);
```

Step 1: The user inputs username jlang and password qwerty and clicks a login button.

Step 5: The alert() method displays "Login Successful" as a popup.

### Animation captions:

1. The HTML form allows the user to submit a username and password.
2. When the form is submitted, e.preventDefault() prevents the browser from reloading the webpage.
3. fetch() creates a POST request, placing the form data in the request body. The form data is sent to the web server.
4. If the username and password are correct, the server responds with the text "Login successful".
5. The alert() method displays the text response in a dialog box.

#### PARTICIPATION ACTIVITY

#### 8.15.7: POST JSON data.



Complete the `postSong()` function, which sends a JSON-encoded song in a POST request to a web API. The web API responds with JSON indicating if the operation is successful.

```
__A__ function postSong() {
 let song = {
 title: "Georgia on My Mind",
 artist: "Ray Charles",
 releaseDate: "1930-11-15"
 };

 let response = await fetch("api/songs", {
 method: __B__,
 headers: {
 "Content-Type": __C__
 },
 body: __D__
 });

 let result = __E__;
 console.log(result.status);
}
```

If unable to drag and drop, refresh the page.

D

A

B

C

E

"application/json"

async

"POST"

await response.json()

JSON.stringify(song)

Reset

Exploring further:

- [Fetch Standard \(WHATWG\)](#)
- [Fetch API \(MDN\)](#)

## 8.16 Example: Lights Out game with canvas

### Revisiting the Lights Out game

The Lights Out game consists of a grid of lights. When clicked, a light and all orthogonally adjacent lights are toggled. When all lights are out, the game is won.

An earlier section provided an implementation of the Lights Out game where lights in the game grid were represented using an HTML table and CSS styles. This section provides an alternate implementation, using a class for the game logic and an HTML canvas for rendering.

### Page HTML

The LightsOutRevisited.html file has two script tags to reference JavaScript source files. Game logic is implemented in the LightsOutGame class, declared in LightsOutGame.js. Canvas rendering and event handling code is implemented in LightsOutRevisited.js.

Page elements include a **<canvas>** for rendering the game grid and processing input, a **<div>** to display a message when the game is won, a button to start a new game with a 3x3 grid of lights, and a button to start a new game with a 5x5 grid of lights.

Figure 8.16.1: LightsOutRevisited.html file.

```
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8">
 <title>Lights Out</title>
 <script src="LightsOutGame.js"></script>
 <script src="LightsOutRevisited.js"></script>
</head>
<body>
 <canvas id="gameCanvas" width="210" height="210"></canvas>
 <div id="information"></div>
 <input id="newGame3x3Button" type="button" value="New game
(3x3)">
 <input id="newGame5x5Button" type="button" value="New game
(5x5)">
</body>
</html>
```

**PARTICIPATION  
ACTIVITY**

## 8.16.1: LightsOutRevisited.html file.

1) How many files comprise the Lights Out game?

- ☐ 1
- ☐ 2
- ☐ 3

2) What is the ID of the canvas?

- ☐ canvas
- ☐ gameCanvas
- ☐ information

3) CSS style declarations are included in the page HTML to define rendering styles.

- ☐ True
- ☐ False

## LightsOutGame.js file

The `LightsOutGame` class represents a game of Lights Out. The constructor initializes the following properties:

- `rowCount`: Number of rows in the game grid.
- `columnCount`: Number of columns in the game grid.
- `lights`: Array of light objects. Each object has members `on` and `lastToggle`, and is frozen.
  - `on` is true if the light is on, false if the light is off.
  - `lastToggle` is a Date object representing when the light was most recently toggled.
- `startTime`: A Date object representing the time the constructor finishes executing and the game starts.

In addition to the constructor, the `LightsOutGame` class has three methods and one property getter:

- `allLightsOut()`: Method returns true if all lights are out, false otherwise.
- `getLight()`: Method takes row and column indices as arguments and returns the corresponding light object.
- `toggle()`: Method takes row and column indices as arguments and toggles the corresponding light and each orthogonally adjacent light.
- `won()`: Getter for game's `won` property, which is true only if all lights are out and the game is won.

`Object.freeze()` creates frozen light objects, preventing external changes that violate game rules

---

**`Object.freeze()`** makes an object a frozen object: an object that cannot have any properties added, removed, or changed. `Object.freeze()` is used to create frozen light objects.

*Good practice is to design the `LightsOutGame` class methods so that external code cannot modify the state in a way that violates game rules. Since each light object returned by `getLight()` is frozen, the caller cannot change the light's `on` value without calling `toggle()` to toggle the appropriate surrounding lights.*

---

Figure 8.16.2: LightsOutGame.js file.

```
class LightsOutGame {
 // Constructs a LightsOut instance for either a 5x5 or 3x3 game,
 // depending
 // on if the is5x5 argument is true or false, respectively
 constructor(is5x5) {
 if (is5x5) {
 this.rowCount = 5;
 this.columnCount = 5;
 }
 else {
 this.rowCount = 3;
 this.columnCount = 3;
 }

 // Allocate the light array, with all lights off
 const lightCount = this.rowCount * this.columnCount;
 this.lights = [];
 for (let i = 0; i < lightCount; i++) {
 // Each light is a frozen object with members "on" and
 // "lastToggle"
 const light = Object.freeze({
 "on": false,
 "lastToggle": new Date()
 });
 this.lights.push(light);
 }

 // Perform a series of random toggles to generate a winnable game
```

```

grid
while (this.allLightsOut()) {
 // Generate random lights
 for (let i = 0; i < this.lights.length * 2; i++) {
 const randRow = Math.floor(Math.random() * this.rowCount);
 const randCol = Math.floor(Math.random() * this.columnCount);

 // Toggle at the location
 this.toggle(randRow, randCol);
 }
}

// Store the start time
this.startTime = new Date();
}

// Returns true if all lights are out, false otherwise
allLightsOut() {
 for (let i = 0; i < this.lights.length; i++) {
 // Even 1 light being on implies that not all are out/off
 if (this.lights[i].on)
 return false;
 }

 // All lights were checked and none are on, so lights are out!
 return true;
}

// Gets the light object at the specified location. The light object
has
// members "on" and "lastToggle".
// Returns null if either index is out of bounds.
getLight(rowIndex, columnIndex) {
 if (rowIndex < 0 || rowIndex >= this.rowCount ||
 columnIndex < 0 || columnIndex >= this.columnCount) {
 return null;
 }

 // Return the light
 return this.lights[rowIndex * this.columnCount + columnIndex];
}

// Toggles the light at (row, column) and each orthogonally
// adjacent light
toggle(row, column) {
 const now = new Date();
 const locations = [
 [row, column], [row - 1, column], [row + 1, column],
 [row, column - 1], [row, column + 1]
];
 for (let location of locations) {
 row = location[0];
 column = location[1];
 if (row >= 0 && row < this.rowCount &&

```



```
 column >= 0 && column < this.columnCount) {
 // Compute array index
 const index = row * this.columnCount + column;

 // Toggle the light
 this.lights[index] = Object.freeze({
 "on": !this.lights[index].on,
 "lastToggle": now
 });
 }
 }
}

// Getter for the 'won' property, which is true if all lights are out,
// false otherwise
get won() {
 return this.allLightsOut();
}
}
```

**PARTICIPATION  
ACTIVITY**

8.16.2: LightsOutGame.js file.



Assume the following code is executed to create a 3x3 Lights Out game:

```
let game = new LightsOutGame(false);
```

- 1) Calling `game.getLight(2, 1)`  
returns the light at index \_\_\_\_ in  
`game.lights`.  

☐ 3

☐ 5

☐ 7
- 2) `game.getLight(0, 0).on = false`; turns off the light in the top  
row and leftmost column.  

☐ True

☐ False



- 3) Complete the code below to log a console message if the game is won.



```
if (_____)
 console.log("Game is
won!");
```

- ☐ `game.won`
- ☐ `game.won()`
- ☐ `game.isWon()`

## LightsOutRevisited.js file

The remainder of the game is implemented in `LightsOutRevisited.js`. An event listener for the `DOMContentLoaded` event is added at the start of the code. A global `game` instance is then declared, followed by five functions:

- `canvasClicked()`: Called when the canvas is clicked. Uses the click location to determine the clicked light and then calls `clickLight()` with appropriate row and column arguments.
- `clickLight()`: Calls `game.toggle()` and then checks to see if the game is won. If the game is won, a message is displayed to the player that includes the time taken to win the game.
- `domLoaded()`: Called when the page's DOM loads. Adds click event listeners for the buttons and the canvas, starts a new 3x3 game, and calls `window.requestAnimationFrame()` to begin rendering.
- `newGame()`: Resets the global `game` variable to a new instance of `LightsOutGame` and clears the information `<div>`.
- `render()`: Renders the game's grid of lights to the canvas.

`render()` first calls `window.requestAnimationFrame()`, which makes the browser continue to call the `render()` function, allowing for animation. Each light fades on or off when toggled.

Each light's `lastToggle` time is used to compute the elapsed time since the light was last toggled. If less than the desired animation duration, the time is scaled to the range `[0.0, 1.0]` and used to perform a weighted average between on and off colors.

Figure 8.16.3: `LightsOutRevisited.js` file.

```
// Add an event listener for the DOMContentLoaded event
window.addEventListener("DOMContentLoaded", domLoaded);
```

```

// Instance of LightsOutGame that stores information about the current
game.
let game = new LightsOutGame(false);

// Handles canvas click events
function canvasClicked(e) {
 // Get the canvas and bounding client rectangle
 const canvas = e.target;
 const rect = canvas.getBoundingClientRect();

 // Compute click coordinates, relative to the canvas
 const x = e.clientX - rect.left;
 const y = e.clientY - rect.top;

 // Convert from pixel coordinates to row, column
 const row = Math.floor(y * game.rowCount / canvas.height);
 const column = Math.floor(x * game.columnCount / canvas.width);

 // Call clickLight
 clickLight(row, column);
}

// Handles a click at the specified location. Toggles lights and checks
to see
// if the game is won.
function clickLight(row, column) {
 // Ignore if the game is already won
 if (game.won)
 return;

 // Toggle the appropriate lights
 game.toggle(row, column);

 // Check to see if the game is won
 if (game.won) {
 // Compute the time taken to solve the puzzle
 const timeTaken = Math.floor(((new Date()) - game.startTime) /
1000);

 // Display message
 const infoDIV = document.getElementById("information");
 infoDIV.textContent = "You win! Solved in " + timeTaken + "
seconds";
 }
}

// Called when the page's DOM loads. Adds click event listeners, starts a
new
// 3x3 game, and begins rendering.
function domLoaded() {
 // Add click event listeners for the two new game buttons
 const btn3x3 = document.getElementById("newGame3x3Button");
 btn3x3.addEventListener("click", function() {
 newGame(3, 3);
 });
}

```

```
 newGame(false);
 });
 const btn5x5 = document.getElementById("newGame5x5Button");
 btn5x5.addEventListener("click", function() {
 newGame(true);
 });

 // Add a click event listener for the canvas
 const canvas = document.getElementById("gameCanvas");
 canvas.addEventListener("click", canvasClicked);

 // Start a new 3x3 game
 newGame(false);

 // Begin rendering
 window.requestAnimationFrame(render);
}

// Resets to a random, winnable game with at least 1 light on
function newGame(is5x5) {
 // Create a new game instance
 game = new LightsOutGame(is5x5);

 // Clear the information <div>
 const infoDIV = document.getElementById("information");
 infoDIV.textContent = "";
}

function render() {
 // Request the next animation frame in advance
 window.requestAnimationFrame(render);

 // Lights fade in/out when toggled. A hard-coded animation duration of
 500
 // milliseconds is used. In practice, animation durations are often
 shorter.
 const animationDuration = 500;

 let canvas = document.getElementById("gameCanvas");
 let ctx = canvas.getContext("2d");

 // Compute width and height of a light's rectangle on the canvas
 const lightWidth = canvas.width / game.columnCount;
 const lightHeight = canvas.height / game.rowCount;

 // Render each light
 for (let row = 0; row < game.rowCount; row++) {
 for (let column = 0; column < game.columnCount; column++) {
 // Get the light
 const light = game.getLight(row, column);

 // Compute the time, in milliseconds, since the light's most
 recent
 // toggle
 }
 }
}
```

```
const timeSinceToggle = (new Date()) - light.lastToggle;

// Compute the light's intensity
let intensity;
if (timeSinceToggle >= animationDuration)
 intensity = light.on ? 255.0 : 0.0;
else if (light.on)
 intensity = timeSinceToggle / animationDuration * 255;
else
 intensity = (1.0 - (timeSinceToggle / animationDuration)) *
255;

// Set the light's color
ctx.fillStyle = `RGB(${intensity}, ${intensity}, 0)`;

// Fill the light's rectangle
ctx.fillRect(
 lightWidth * column, lightHeight * row,
 lightWidth, lightHeight);

// Draw a thin white border, so the lights are visually distinct
ctx.strokeStyle = "white";
ctx.strokeRect(
 lightWidth * column, lightHeight * row,
 lightWidth, lightHeight);
}
}
}
```

**PARTICIPATION  
ACTIVITY**

## 8.16.3: LightsOutRevisited.js functions.



If unable to drag and drop, refresh the page.

**newGame()****clickLight()****render()****domLoaded()****canvasClicked()**

Computes a light's row and column indices based on a click location.

Resets the global game variable to a new instance of `LightsOutGame`.

Draws each light on the canvas.

Begins rendering by calling  
`window.requestAnimationFrame()`.

Displays a message in the  
information `<div>` if the game is  
won.

[Reset](#)

## Lights Out full source.

The full source of `LightsOutRevisited.html`, `LightsOutGame.js`, and `LightsOutRevisited.js` are provided below, followed by the rendered version of the page.

[LightsOutRevisited.html](#)[LightsOutGame.js](#)[LightsOutRevisited.js](#)

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4 <meta charset="UTF-8">
5 <title>Lights Out</title>
6 <script src="LightsOutGame.js"></script>
7 <script src="LightsOutRevisited.js"></script>
8 </head>
9 <body>
10 <canvas id="gameCanvas" width="210" height="210"></canvas>
11 <div id="information"></div>
12 <input id="newGame3x3Button" type="button" value="New game (3x3)">
13 <input id="newGame5x5Button" type="button" value="New game (5x5)">
14 </body>
15 </html>
```

[Render webpage](#)[Reset code](#)

### Your webpage

New game (3x3)

New game (5x5)

## 8.17 Example: Weather Comparison (fetch)

### Weather Comparison overview

This section presents an example implementation of a Weather Comparison web app that compares the weather forecast between two cities. The OpenWeatherMap's free [5 day weather forecast API](#) is used to obtain weather forecasts using `fetch()`.

The OpenWeatherMap website provides documentation explaining how to use the forecast API using GET requests with various query string parameters. The API endpoint `https://api.openweathermap.org/data/2.5/forecast` returns the current weather based on the following query string parameters:

- q - City name
- units - Standard, metric, or imperial units to use for measurements like temperature and wind speed
- appid - Developer's API key

Other parameters are documented in the OpenWeatherMap website. The Weather API returns weather data in JSON format by default. The JSON response contains the weather forecast for 5

days with data every 3 hours. The figure below shows the first hour's forecast for Denver in the `list`.

Figure 8.17.1: URL and JSON response containing Denver forecast.

[http://api.openweathermap.org/data/2.5/forecast?](http://api.openweathermap.org/data/2.5/forecast?q=Denver,US&units=imperial&appid=APIKEY)

[q=Denver,US&units=imperial&appid=APIKEY](#)

```
{
 "cod": "200",
 "message": 0.0046,
 "cnt": 40,
 "list": [
 {
 "dt": 1545242400,
 "main": {
 "temp": 38.44,
 "temp_min": 37.08,
 "temp_max": 38.44,
 "pressure": 815.31,
 "sea_level": 1030.83,
 "grnd_level": 815.31,
 "humidity": 26,
 "temp_kf": 0.71
 },
 "weather": [{ "id": 500, "main": "Rain",
 "description": "light rain", "icon": "10d" }],
 "clouds": { "all": 88 },
 "wind": { "speed": 3.33, "deg": 175 },
 "rain": { "3h": 0.07 },
 "sys": { "pod": "d" },
 "dt_txt": "2018-12-19 18:00:00"
 },
 ...etc...
],
 "city": {
 "id": 5419384,
 "name": "Denver",
 "coord": { "lat": 39.7392, "lon": -104.9848 },
 "country": "US",
 "population": 600158
 }
}
```

forecasted temperature

weather type

time of forecast

city name

single forecast



**PARTICIPATION  
ACTIVITY**

## 8.17.1: JSON forecast.



Refer to the figure above.

- 1) The forecast is for 6:00 pm on December 19, 2018.  
☐ True  
☐ False
- 2) The forecast is for a clear sky.  
☐ True  
☐ False
- 3) The forecast temperature is shown in Celsius.  
☐ True  
☐ False
- 4) The next item in the `list` should be 3 hours after the currently shown `list` item.  
☐ True  
☐ False



## Page HTML and CSS

The HTML for the page has two text inputs and a Compare button so the user can enter the two cities to compare. The 5 day forecast will be displayed below in two tables. The web page uses an external stylesheet `styles.css` for styling the page.

HTML and CSS for Weather Comparison app.

The app currently does nothing when Compare is clicked.

weather.html

styles.css

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4 <meta charset="UTF-8">
5 <title>Weather Comparison</title>
6 <link rel="stylesheet" href="styles.css">
7 <!-- <script src="weather.js"></script> -->
8
9 </head>
10 <body>
11 <h1>Weather Comparison</h1>
12 <section id="weather-input">
13 <p>
14 <label for="city1">City 1:</label>
15 <input type="text" id="city1">
16 Enter a city
```

Render webpage

Reset code

## Your webpage

# Weather Comparison

City 1: City 2: PARTICIPATION  
ACTIVITY

8.17.2: HTML and CSS for Weather Comparison app.



- 1) What CSS class is hiding the "Enter a city" messages and the forecast section?
- ☐ `error-msg`
  - ☐ `hidden`
  - ☐ `display`
- 2) What does the web page display if the `hidden` class is removed from an "Enter a city" `<span>`?
- ☐ Nothing. The text remains hidden.
  - ☐ "Enter a city" under the textbox.
  - ☐ "Enter a city" next to the textbox.
- 3) What does the web page display if the `hidden` class is removed from the forecast `<section>`?
- ☐ Nothing. The section remains hidden.
  - ☐ Two empty tables.
  - ☐ Two "Loading..." messages and two empty tables.

## Handling the Compare button click

When the Compare button is clicked, the JavaScript in `weather.js` needs to:

1. Extract the cities from the text boxes.
2. If no city is entered in a text box, display an error message next to the text box.
3. If two cities are entered, hide the error messages and show the forecast section and only the loading messages.
4. Send two HTTP requests to the forecast API requesting the forecast for the two cities.

The figure below shows some of the code necessary to implement the above logic.

Figure 8.17.2: Compare button callback and supporting code.

```
// Called when Comapre button is clicked
function compareBtnClick() {

 // Get user input
 const city1 =
document.getElementById("city1").value.trim();
 const city2 =
document.getElementById("city2").value.trim();

 // Show error messages if city fields left blank
 if (city1.length === 0) {
 showElement("error-value-city1");
 }
 if (city2.length === 0) {
 showElement("error-value-city2");
 }

 // Ensure both city names provided
 if (city1.length > 0 && city2.length > 0) {
 showElement("forecast");
 hideElement("error-loading-city1");
 hideElement("error-loading-city2");
 showElement("loading-city1");
 showMessage("loading-city1", `Loading ${city1}...`);
 showElement("loading-city2");
 showMessage("loading-city2", `Loading ${city2}...`);
 hideElement("results-city1");
 hideElement("results-city2");

 // Fetch forecasts
 getWeatherForecast(city1, "city1");
 getWeatherForecast(city2, "city2");
 }
}

// Display the message in the element
function showMessage(elementId, message) {
 document.getElementById(elementId).innerHTML = message;
}

// Show the element
function showElement(elementId) {

document.getElementById(elementId).classList.remove("hidden");
}

// Hide the element
function hideElement(elementId) {
 document.getElementById(elementId).classList.add("hidden");
}
```

}

## Partially implemented Weather Comparison app.

Click Compare before entering a city name to see error messages. Then enter two city names and click Compare. The `getWeatherForecast( )` function has not been implemented yet, so no forecasts are displayed.

weather.html

styles.css

weather.js

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4 <meta charset="UTF-8">
5 <title>Weather Comparison</title>
6 <link rel="stylesheet" href="styles.css">
7 <script src="weather.js"></script>
8 </head>
9 <body>
10 <h1>Weather Comparison</h1>
11 <section id="weather-input">
12 <p>
13 <label for="city1">City 1:</label>
14 <input type="text" id="city1">
15 Enter a ci
16 </p>
```

Render webpage

Reset code

## Your webpage

# Weather Comparison

City 1: City 2: **PARTICIPATION  
ACTIVITY**

8.17.3: Compare button click.

1) Only one "Enter a city" error message can appear at a time.

- ☐ True  
☐ False

2) An "Enter a city" error message appears if the user just types a few spaces.

- ☐ True  
☐ False

3) The "Enter a city" error message only disappears when the user enters a city in the text box next to the message and clicks Compare again.

- ☐ True  
☐ False

4) Clicking the Compare button causes two HTTP requests to be sent to the forecast API.

- ☐ True
- ☐ False

## Requesting the forecast

The `getWeatherForecast()` function is responsible for sending an HTTP request to the forecast API. The `fetch()` method requests a URL with the given city. If the API returns a 2xx response code, `response.json()` is called to convert the JSON into a JavaScript object. Then the `displayForecast()` function, which appears in the next subsection, displays the forecast. If the API returns any other response code, an error message is displayed.

Figure 8.17.3: getWeatherForecast() function.

```
// Request this city's forecast
async function getWeatherForecast(city, cityId) {

 // Create a URL to access the web API
 const endpoint = "https://api.openweathermap.org/data/2.5/forecast";
 const apiKey = "Your API key goes here";
 const queryString = "q=" + encodeURIComponent(city) + "&units=imperial&appid="
+ apiKey;
 const url = endpoint + "?" + queryString;

 // Send http request to web API
 const response = await fetch(url);

 // No longer loading
 hideElement("loading-" + cityId);

 // See if forecast was successfully received
 if (response.ok) {
 const jsonResult = await response.json();
 displayForecast(cityId, jsonResult);
 }
 else {
 // Display appropriate error message
 const errorId = "error-loading-" + cityId;
 showElement(errorId);
 showMessage(errorId, `Unable to load city "${city}".`);
 }
}
```

**PARTICIPATION  
ACTIVITY**

## 8.17.4: Requesting the forecast.

1) How is `getWeatherForecast()` properly called?

- ☐ `getWeatherForecast("Denver");`
- ☐ `getWeatherForecast("Denver", true);`
- ☐ `getWeatherForecast("Denver", "city1");`



2) What does the query string look like if Boston is the city?

- ☐ ?q=Boston
- ☐ ?q=Boston&appid=APIKEY
- ☐ ?  
q=Boston&units=imperial&appid=APIKEY

3) If the API returns a 404 status code, the webpage displays \_\_\_\_.

- ☐ error message
- ☐ a default forecast
- ☐ nothing

## Displaying the forecast

The `displayForecast()` function displays the forecast information in the city's table by calling several helper functions:

- `getSummaryForecast()` - Retrieves a map of objects containing the day's high, low, and weather summary for 5 days.
- `getDayName()` - Converts a date like "2022-06-19" into a day name like "Sun".
- `showImage()` - Displays the weather image matching the given weather type.

Figure 8.17.4: `displayForecast()` and supporting functions.

```
// Display forecast received from JSON
function displayForecast(cityId, jsonResult) {
 showElement("results-" + cityId);

 const cityName = jsonResult.city.name;
 showMessage(cityId + "-name", cityName);

 // Get 5 day forecast map
 const forecastMap = getSummaryForecast(jsonResult.list);

 // Put forecast into the city's table
 let day = 1;
 for (const date in forecastMap) {
 // Only process the first 5 days
 if (day <= 5) {
 const dayForecast = forecastMap[date];
```

```

 const dayForecast = forecastMap[date];
 showMessage(`${cityId}-day${day}-name`, getDayName(date));
 showMessage(`${cityId}-day${day}-high`,
Math.round(dayForecast.high) + "°");
 showMessage(`${cityId}-day${day}-low`,
Math.round(dayForecast.low) + "°");
 showImage(`${cityId}-day${day}-image`, dayForecast.weather);
 }
 day++;
}
}

// Convert date string into Mon, Tue, etc.
function getDayName(dateStr) {
 const date = new Date(dateStr);
 return date.toLocaleDateString("en-US", { weekday: "short", timeZone:
"UTC" });
}

// Show the weather image that matches the weatherType
function showImage(elementId, weatherType) {

 // Images for various weather types
 const weatherImages = {
 Clear: "clear.png",
 Clouds: "clouds.png",
 Drizzle: "drizzle.png",
 Mist: "mist.png",
 Rain: "rain.png",
 Snow: "snow.png"
 };

 const imgUrl = "https://static-resources.zybooks.com/";
 const img = document.getElementById(elementId);
 img.src = imgUrl + weatherImages[weatherType];
 img.alt = weatherType;
}

```

The `getSummaryForecast()` function loops through the forecast data retrieved from the forecast API and creates a map of 5 or 6 objects that contain the highest and lowest temperature for each day and weather summary string like "Clear", "Rain", or "Snow".

Figure 8.17.5: `getSummaryForecast()` function.

```

// Return a map of objects with high, low, weather
properties
function getSummaryForecast(forecastList) {

 // Map for storing high, low, weather

```

```
const forecast = {};

// Determine high and low for each day
forecastList.forEach(function (item) {
 // Extract just the yyyy-mm-dd
 const date = item.dt_txt.substr(0, 10);

 // Extract temperature
 const temp = item.main.temp;

 // Has this date been seen before?
 if (date in forecast) {
 // Determine if the temperature is a new low or
high
 if (temp < forecast[date].low) {
 forecast[date].low = temp;
 }
 if (temp > forecast[date].high) {
 forecast[date].high = temp;
 }
 }
 else {
 // Initialize new forecast
 const temps = {
 high: temp,
 low: temp,
 weather: item.weather[0].main
 }

 // Add entry to map
 forecast[date] = temps;
 }
});

return forecast;
}
```

Example return value:

```
2022-01-22: {high: 43, low: 38.93, weather: "Rain"},
2022-01-23: {high: 47.02, low: 42.8, weather: "Rain"},
2022-01-24: {high: 47.44, low: 43.43, weather: "Clouds"},
2022-01-25: {high: 45.45, low: 40.39, weather: "Clouds"},
2022-01-26: {high: 42.79, low: 39.57, weather: "Clear"},
2022-01-27: {high: 41.33, low: 37.3, weather: "Clear"}
```

## Weather Comparison app.

The complete implementation is provided below. Enter two city names like "Denver" and "Seattle" and click Compare to see the forecasts compared.

The implementation uses zyBook's API key. You may need to replace the API key in `getWeatherForecast()` with your own key due to API query limits.

Weather images from [OpenWeatherMap.org](https://openweathermap.org)

[weather.html](#)[styles.css](#)[weather.js](#)

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4 <meta charset="UTF-8">
5 <title>Weather Comparison</title>
6 <link rel="stylesheet" href="styles.css">
7 <script src="weather.js"></script>
8 </head>
9 <body>
10 <h1>Weather Comparison</h1>
11 <section id="weather-input">
12 <p>
13 <label for="city1">City 1:</label>
14 <input type="text" id="city1">
15 Enter a ci
16 </p>
```

[Render webpage](#)[Reset code](#)

Your webpage

# Weather Comparison

City 1:

City 2:

Compare

**PARTICIPATION  
ACTIVITY**

## 8.17.5: Displaying the forecast.

- 1) `displayForecast()` displays the forecast's \_\_\_\_ on the table's top row.
- ☐ day abbreviations
  - ☐ high temperatures
  - ☐ low temperatures
- 2) What is the likely size of `jsonResult.list` when `getSummaryForecast(jsonResult.list)` is called?
- ☐ 1
  - ☐ 20
  - ☐ 40
- 3) The figure above shows an example return value from calling `getSummaryForecast()`. How many `jsonResult.list` forecast objects were likely examined to produce a high of 47.44 and a low of 43.43 on 2022-01-24?
- ☐ 1
  - ☐ 8
  - ☐ 40

## 8.18 LAB: User registration and validation with regex



This section's content is not available for print.

## 8.19 LAB: JavaScript SuperHero and SuperVillain classes



This section's content is not available for print.

## 8.20 LAB: Grocery list



This section's content is not available for print.

## 8.21 LAB: Snowman canvas



This section's content is not available for print.

## 8.22 LAB: Frog image rotation



This section's content is not available for print.

## 8.23 LAB: Circle with a Promise



This section's content is not available for print.

## 8.24 LAB: Quote web API (Fetch)



This section's content is not available for print.